XICSRT

Release 0.8.8

May 02, 2023

Contents:

XICSRT is a general purpose, photon based, raytracing code intended for both optical and x-ray raytracing.

The best way to get started with XICSRT is with the `examples`.

Documentation: xicsrt.readthedocs.org
Git Repository: bitbucket.org/amicitas/xicsrt
Git Mirror: github.com/PrincetonUniversity/xicsrt

XICSRT provides a simple, extensible, optical and x-ray raytracing capability in Python. Input is a single python dictionary (which can be saved to a *json* file), output is a python dictionary (which can be saved to a *hdf5* file).

For interactive use XICSRT can run within a jupyter notebook. Simple examples for 2D and 3D plotting using the matplotlib and plotly libraries are included. A command line interface to XICSRT is also available.

XICSRT has been written with a primary goal of simplicity and ease of extensibility, rather than computational speed. That being said the code has been thoroughly vectorized and optimized, and most expensive calculations are performed through built-in numpy routines. Use across multiple processors can be achieved though the built-in *multiprocessing* capabilities.

> **Warning:** Documentation of XICSRT is still in progress. Please get involved and help us improve the documentation!

CHAPTER 1

Installation

XICSRT can be simply installed using *pip*

```
pip install xicsrt
```

Alternatively you can install from source using *setuptools*

```
python setup.py install
```

CHAPTER 2

Usage

XICSRT is run by supplying a *config* dictionary to *raytrace()*.

```
results = xicsrt.raytrace(config)
```

To learn how format the input and interpret the output, try the *examples* or download the XICSRT Tutorial.

CHAPTER 3

Tutorial

An XICSRT Tutorial presentation is available that introduces basic usage and concepts.

CHAPTER 4

Authors

XICSRT development is coordinated by Novimir A. Pablant. A full list of contributers can be found on the *Authors* page.

CHAPTER 5

Citation

If you use XICSRT for work leading to a publication, please use the following citation:

N. A. Pablant, M. Bitter, P. C. Efthimion, L. Gao, K. W. Hill, B. F. Kraus, J. Kring, M. J. MacDonald, N. Ose, Y. Ping, M. B. Schneider, S. Stoupin, and Y. Yakusevitch, **"Design and expected performance of a variable-radii sinusoidal spiral x-ray spectrometer for the National Ignition Facility"**, Review of Scientific Instruments 92, 093904 (2021) https://doi.org/10.1063/5.0054329

A list of publications relating to XICSRT can be found at the *List of Publications* page.

CHAPTER 6

License

XICSRT is open source software released under the MIT License. For the full text of the licence see the *License* page. Please help improve XICSRT by contributing to the codebase.

6.1 User Manual

This user manual is still incomplete. For now, please refer to the XICSRT Tutorial.

6.1.1 Command Line Interface

A command line interface is available for xicsrt. For a standard installation the command `xicsrt` will be available. The command line interface can also be invoked using `python -m xicsrt`.

A saved *config* dictionary file is required to use the command line interface (typially a *.json* file). Once defined we can simply pass this file to the `xicsrt` command.

```
xicsrt config.json
```

To aid in saving of *config* dictionaries to *.json* files the helper function *`xicsrt_io.save_config()`* is available.

xicsrt

A command line interface for the XICSRT raytracer.

usage:
xicsrt [-h] [–numruns N] [–numiter N] [–seed N] [–save]
[–images] [–suffix STR] [–path STR] [–multiprocessing]
[–processes N] [–version] [–debug]
[config_file]

xicsrt version 0.8.8

description:
Perform an XICSRT raytrace from the command line.

The input to this command should be an XICSRT configuration dictionary
in json format. (Pickle and hdf5 formats are also supported.)

example 1:
xicsrt config.json

example 2:
python -m xicsrt config.json

positional arguments:
config_file The path to the configuration file for this run.

optional arguments:
-h, –help show this help message and exit
–numruns N Number of runs.
–numiter N Number of iterations per run.
–seed N The random seed to use.
–save Save the results.
–images Save intersection images.
–suffix STR A suffix to add to the output files.
–path STR Directory in which to store output.
–multiprocessing, –mp
Use multiprocessing.
–processes N Number of processes to use for muliprocessing.
–version Show the version number.
–debug Show debugging output in the log.

6.1.2 XICSRT on Multiple Processors

XICSRT has built-in support for raytracing over multiple processors through the use of Python's multiprocessing library. To use this functionality one only needs to replace the call *xicsrt.raytrace()* with *xicsrt.raytrace_mp()*.

Windows Support

Multiprocessing on Windows requires that *xicsrt.raytrace_mp()* is wrapped in a `name == "__main__"` test. In a python script or jupyter notebook on Windows this means replacing the call to *xicsrt.raytrace()* with the following:

```
if __name__ == "__main__":
    results = xicsrt.raytrace_mp(config)
```

The *Command Line Interface* can be used without any modifications.

Cluster Computing

A command line interface to xicsrt is available to enable computations on a computer cluster.

A single call to xicsrt can utilize multiple processors but is (currently) limited to run on a single computational node. However, it is easy to combine results from multiple calls to xicsrt allowing multiple nodes to be used. The use of slurm Job Arrays is recommended.

To launch a single call to xicsrt on 16 processors using slurm the following command can be used:

```
srun -n1 -c16 xicsrt config.json --mp --numruns 16 --processes 16
```

For multiple parallel calls to xicsrt use the `--suffix` option to give all output files a unique name.

Note: For multiple calls to xicsrt make sure that the random seed is either

1. equal to `None` (the default)
2. set to a different value for each call using the `--seed` argument.

Job Array Example

Below is a simple example of a slurm batch file to run xicsrt on 64 processors over 4 nodes.

job.sh

```
#!/bin/bash
#SBATCH -J xicsrt
#SBATCH -o ./job_%A_%a.out
#SBATCH -e ./job_%A_%a.err
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --array=0-3

srun xicsrt config.json --mp --numruns 16 --processes 16 --suffix $SLURM_ARRAY_TASK_
↪ID &> xicsrt.log
```

To send this job to the queue type `sbatch job.sh` at the command line.

After the jobs are complete the saved results can be read using *`xicsrt_io.load_results()`* and combined using *`xicsrt_raytrace.combine_raytrace()`*.

If you use XICSRT for work leading to a publication, please use the following citation:

N. A. Pablant, M. Bitter, P. C. Efthimion, L. Gao, K. W. Hill, B. F. Kraus, J. Kring, M. J. MacDonald, N. Ose, Y. Ping, M. B. Schneider, S. Stoupin, and Y. Yakusevitch, **"Design and expected performance of a variable-radii sinusoidal spiral x-ray spectrometer for the National Ignition Facility"**, Review of Scientific Instruments 92, 093904 (2021) https://doi.org/10.1063/5.0054329

6.1.3 List of Publications

1. N. A. Pablant, M. Bitter, P. C. Efthimion, L. Gao, K. W. Hill, B. F. Kraus, J. Kring, M. J. MacDonald, N. Ose, Y. Ping, M. B. Schneider, S. Stoupin, and Y. Yakusevitch, **"Design and expected performance of a variable-radii sinusoidal spiral x-ray spectrometer for the National Ignition Facility"**, Review of Scientific Instruments 92, 093904 (2021) https://doi.org/10.1063/5.0054329

2. N. A. Pablant, A. Langenberg, J. A. Alonso, M. Bitter, S. A. Bozhenkov, O. P. Ford, K. W. Hill, J. Kring, O. Marchuck, J. Svensson, P. Traverso, T. Windisch, Y. Yakusevitch, and the W7-X Team, **"Correction and verification of x-ray imaging crystal spectrometer analysis on Wendelstein 7-X through x-ray ray tracing"**, Review of Scientific Instruments 92, 043530 (2021) https://doi.org/10.1063/5.0043513

3. J. Kring, N. Pablant, A. Langenberg, J. Rice, L. Delgado-Aparicio, D. Maurer, P. Traverso, M. Bitter, K. Hill, and M. Reinke, **"In situ wavelength calibration system for the X-ray Imaging Crystal Spectrometer (XICS) on W7-X"**, Review of Scientific Instruments 89, 10F107 (2018) https://doi.org/10.1063/1.5038809

6.1.4 Similar Software

There are a number of scientific x-ray raytracing codes available with similarities to XICSRT. Each of these codes has a different set of goals, strengths, and advantages.

SHADOW

SHADOW is perhaps the most well known scientific x-ray raytracing code and is used extensively within the synchrotron compunity. SHADOW has been in continuous development since 1982 and and has been thoroughly validated against experiment.

SHADOW is written in FORTRAN and released under the opensource MIT License. A graphical user interface is available through the OASYS software suite and a Python API is available for programmatic interaction.

- https://www.aps.anl.gov/Science/Scientific-Software/OASYS
- https://github.com/oasys-kit/shadow3

A new version of SHADOW (SHADOW4) is currently under development in which the code base is being completely rewritten in python.

- https://github.com/srio/shadow4

Xrt (XRayTracer)

xrt (XRayTracer) is a python software library for ray tracing and wave propagation in x-ray regime. It is primarily meant for modeling synchrotron sources, beamlines and beamline elements. Includes a GUI for creating a beamline and interactively viewing it in 3D.

XRT is written in Python and release under the opensource MIT License.

- https://xrt.readthedocs.io/

McXtrace

McXtrace is an extension of McStas, a neutron raytracing code. *McStas is a general tool for simulating neutron scattering instruments and experiments. It is actively supported by DTU Physics, NBI KU, ESS, PSI and ILL.*

McXtrace/McStas is written in C and release under an opensource GPL 2.0 License.

- https://www.mcxtrace.org/

6.1.5 Development Projects

A list of needed improvements for XICSRT can be found at *List of Todo Items*. Also see the open issues on the bitbucket git repository.

Programming Projects

Here are a list of projects for XICSRT improvements. These are particularly well suited for a undergraduate summer student, or anyone looking for a nice self-contained improvement project.

Time estimates are for someone who has experience running XICSRT, has a very strong python/numpy programming background, but who is not familiar with the XICSRT code base. Time estimates include time for testing and verification. I expect that it will take at *least* 2-3 times longer for most students or new contributors who will also be learning all the various python programming and numerical programming concepts!

Add a cylindrical reflector object

Time Estimate: **1 week**
Create a cylindrical shape object named *ShapeCylindrical*. Object should be very similar to `ShapeSphere` but with cylindrical geometry. The object should be defined with a radius. Test against `ShapePlane`, *ShapeToroidal*.

added: 2021-01-24 by Novimir

Add a toroidal reflector object

In progress by S. Mishra

Time Estimate: **2 weeks**
Create a toroidal shape named *ShapeToroidal*. Object should be very similar to `ShapeSphere` but with toroidal geometry. The object should be defined with a major and minor radius. Test against `ShapePlane`, `ShapeSphere` and *ShapeCylindrical*.

added: 2021-01-24 by Novimir

Add plotting routines for all aperture definitions

Time Estimate: **1 weeks**
The 2d plotter knows how to plot a few aperture shapes but, not all shapes have been added. Thes addition shapes need to be added to the function *_get_aperture_plotlist*.

added: 2021-07-02 by Novimir

Improve algorithm for isotropic emission with x & y limits

Time Estimate: **2 weeks**
An important vector distribution used in XICSRT is the isotropic distribution with separate x & y angular bounds (rectangular cone, pyramid). The function that implements this can be found in `vector_dist_isotropic_xy`, The current algorithm uses filtering from an emission cone with circular cross-section. This is accurate but highly inefficient, especially if the x & y spread are very different.

A more efficient algorithm is needed. This is almost certainly a solved problem so the first thing to do is to search the literature and look at other ray-tracing projects to find an existing example.

If an example cannot be found I see three possibilities for a solution:

1. Calculate the Joint Cumulative Distribution Function (CDF) on a plane of constant z. Use this to draw random points on the plane. A good (free) text on probability distributions can be found here: probabilitycourse.com.
2. Pull points on a unit-sphere only within the boundary of the rectangular- cone intersection. I have no idea how to approach this other than falling back on solution 1.
3. Continue using a filtering scheme, but start with a different boundary shape than a circle that is closer to the one needed for the rectangular cone.

It is important that the final algorithm is accurate to machine precision.

added: 2021-01-24 by Novimir

Improve mesh-grid pre-selection algorithm

Time Estimate: **2 weeks**
Mesh-grid optics in XICS use a mesh-refinement alorithm that uses a course grid to pre-select faces to test on the full mesh. The current algorithm is lossy, and often tests more faces than are actually required.

The goal of this project is to improve the pre-selection algorithm to eliminate ray losses. This can likely be done while also improving performance and allowing coarser pre-selection grids.

The specific methods in `ShapeMesh` that need improvement is `find_near_faces` however to achive this change will also be needed in *_mesh_precalc* and `mesh_intersect_2`.

Note: For a very course pre-selection grid and oblique incidence some ray loss will be expected even for this new algorithm.

Note: Consider how the new algorithm will perform with grids in which the x & y point densities are very different. The current algorithm behaves especially poorly in terms of losses in those cases.

added: 2021-01-24 by Novimir

Develop a numba accelerated version of XICSRT

Time Estimate: **2 months**
Performance of XICSRT can likely be dramatically improved by using the the numba package. Numba provides just-in-time compilation of python code and is highly integrated with numpy, making it well suited for inclusion in XICSRT.

Numba can often provide acceleration by just adding the *@jit* decorator. To really achieve acceleration, it is likely that some code changes are required. When available, use the *@vectorize* or *@guvectorize* decorators. Consider how this code will perform on multiple cpus or gpus. Consider the use of *prange* when approprate.

Development should be done in separate branch so as not to affect the master branch (though any code improvements that are not numba specific should still be made in the master branch).The new numba branch should contain a way to turn off numba, and care should be taken that the code still works seamlessly with numba turned off. Performance should be measured between the non-numba version, the numba version, and the numba version with numba turned off.

Note: XICSRT is already highly vectorized and utilizes numpy array manipulations whenever possible. These operations are already very fast, and some are even optimized for multiple processors. For this reason it is unclear how much speed improvement is actually achievable with numba. During development of the numba branch please also look into optimizing the standard numpy code.

Note: The main goals of XICSRT project are readability, easy development, cross-platform compatiblity, and pure python. Code changes that improve performance but make the code very complex should be avoided.

added: 2021-01-24 by Novimir

Make sure that RayDict is used everywhere

Time Estimate: **< 1 week**
In XICSRT the rays that we are tracing are always kept in a dictionary with some standard entries such as 'origin', 'direction', 'mask', etc.. This dictionary should always be an instance of the RayDict object. Right now this is inconsistent; in some places the RayDict is used, in others a regular dict is used instead. This project is to go through the code and make sure that RayDict is used everywhere.

Note: Right now, 2020-02-01, the object is called `RayArray`. This should be renamed to something better and more understandable such as *RayDict*, *RayObject*, *XicrtRays* etc.

Note: In general XICSRT should always treat RayDict as a regular dict. The reason for using RayDict is primarily for consistency, but also so that the RayDict object can eventually contain some convenience methods.

added: 2021-02-01 by Novimir

Better logging for xicsrt_multiprocessing

Time Estimate: **1 week**

Currently When running a raytrace using `xicsrt_multiprocessing.raytrace` in a Jupyter notebook there is not useful progress information displayed. We need to figure out a way to provide at least some minimial information on the number of runs and interations completed that showup in the notebook while execution is ongoing.

Simply using the image suffix and a 'run XX is complete' would be enough here. Calculating a percentage or estimated time might be a bonus, but we need to be careful not to introduce overly complicated code.

Note: Some logging information is displayed within the terminal session in which the Jupyter notebook is running, but not in the notebook webpage. This is of course not sufficient since many users will not be running Jupyter from a terminal, or that terminal session will be hidden.

added: 2021-02-04 by Novimir

Create an Aperture Optic

DONE!! (Thanks to Nathan Bartlett)

Time Estimate: **2 weeks**

Create an object named *XicsrtOpticAperature* that can act as an aperture to filter rays. The shape of the aperture should be implemented as a configuration option. Most of the coding for this should actually be implemented into *XicsrtOpticGeneric* so that the code can also be used to control the size of optics. This aperture object should inherit from *XicsrtOpticMesh*, and will probably not have any differences except for the default config options.

The options need to support at least rectangular and circular aperture shapes and should be implemented in such a way that it is:

1. Easy to add additional simple shapes in the base code.
2. Easy for a user to extend to complex shapes by creating a subclass of the object.

The mechanism used for this object should also be applicable to set the size of optics objects. This brings up some additional considerations:

3. Make sure that the aperture check is done as early as possible so that rays are excluded before any other calculations (such as reflection, Bragg check, etc). Of course the ray intersection needs to be calculated before the aperture check.
4. Aperture needs to be compatible with mesh optics. Make sure to check how the aperture fits in with the code in *XicsrtOpticMesh*.
5. Implement a way to deal with the pixel grid size used for image output. Currently this is based on a rectangular aperture.
6. Consider the possibility that some future optics types may need both an entrance-aperture and an exit-aperture. This capability is not currently needed, but make the code easily extensible to this idea if needed.

Finally we need to consider how to deal with the *size* specification for the aperture and more generaly the optic size. Currently only a rectangular optic shape is supported and the shape is defined by the *xsize*, *ysize* and *zsize* config options. These names don't make sense for a circular aperture. I have two ideas for how to handle this:

a. Use a single *size* option that now becomes an array. The interpretation of this array will depend on on the shape specification. For example a rectangular aperture would interpret *config['size'] = [0.1, 0.2]* as as an *xsize* and *ysize*, while a circular aperture would interpret *config['size'] = 0.1* as a radius.

b. Introduce new *-size* options as needed for each aperture shape. So for a circular aperture introduce an *rsize* config option.

I tend to prefer option (a), but would like some feedback. Option (a) is good because there are no unused *-size* specifications floating around to cause confusion. We don't need to check whether the right *-size* option is being specified by the user. However, option (a) means that *size* now has a variable length which is potentially confusing to the user and now requires some parsing code similar to `xicsrt_spread`.

added: 2021-01-29 by Novimir

6.1.6 List of Todo Items

A list of needed improvements for XICSRT.

Please also see the open issues on the bitbucket git repository.

Todo: InteractMosaicCrystal efficiency could be improved by including a pre-filter. The pre-filter would use a step-function rocking curve to exclude rays that are outside the likely range of reflection with the current mosaic spread.

---

original entry

Todo: XicsrtOpticMesh: Improve the pre-selection (mesh refinement algorithm) to eliminate ray losses. The current method is as follows:

1. Calculate intersection with coarse grid.
2. Find the point on the fine grid closest to the intersection.
3. Test all faces on the fine grid that contain this point.

The problem is that the closest point may not always be part of the face that actually has the intersection. This can happen if the fine and coarse grid have very different densities, but also even in the perfect case if the ray hits near the edge of a face and the grid density is not even in the x and y directions.

What is needed is a better selection of nearby faces. There is also a potential to improve computational speed slightly by testing fewer faces on the fine grid.

---

original entry

Todo:

- The config docstrings should all be indented follow the help() standard.
- Would it be helpful to show which inherited class the options came from?

---

original entry

Todo: Replace vector_dist_isotropic_xy with a more efficent calculation. A possible approach is to calculate the 2D Joint Cumulative Distribution Function for isotropic emission on a flat plane.

original entry

6.1.7 Testing

integrated_test_00 `integrated_test_00.ipynb`

Check ray generation for plasma sources.

integrated_test_01 `integrated_test_01.ipynb`

Perform a simple raytrace using all (most) defined optics.

6.2 Examples

This set of examples is meant to be used as a tutorial of sorts. These start very simple and get increasingly more complex as they introduce additional raytracing features.

6.2.1 example_00

Download a Jupyter notebook with this example: `example_00.ipynb`.

Please follow the source comments for description and instruction.

Source Code

```
# -*- coding: utf-8 -*-
"""
..  Authors:
    Novimir Antoniuk Pablant <npablant@pppl.gov>

A simple example consisting only of a point source and a spherical crystal.

Description
-----------

1.
Create a new user configuration dictionary.

The entries that we put into this config will overwrite the defaults
that are defined within xicsrt. The config can potentially contain the
following sections:

- general
- sources
- optics
- filters
```

(continues on next page)

(continued from previous page)

```
- scenario

2.
Create a section that contains the general raytracer configuration.

number_of_iter
  Perform raytracing the given number of times. The output from all
  the iterations will be combined. Performing multiple iterations allows
  a large number of rays to be traced without running into memory limits.
save_images
  If set to true, images will be saved to the output directory (which
  we have not specified in this example.

3.
Create the section that contains the sources.
Then define a source, cleverly named 'source'.

class_name
  The type of source object to create.
intensity
  The number of rays to launch in each iteration.
wavelength
  The nominal wavelength of the source emission.
spread
   The angular spread of the source (in radians).

4.
Create the section that contains the optics.
In this case we only define one optic: a detector.

class_name
  The type of optic object to create.
origin
  The location of this optic.
zaxis
  The direction the optics is pointing. For all of the standard
  optics that come with xicrt, the zaxis is the normal direction.
xsize
  The size of the optic along the xaxis.
  Corresponds to the 'width' of the optic.
ysize
  The size of the optic along the yaxis.
  Corresponds to the 'height' of the optic.

5.
Finally we pass the configuration to the XICSRT raytracer to perform
the actual raytracing. The `results` is a dictionary with the full
trace history along with images at the detector.
"""

import numpy as np
import xicsrt
xicsrt.warn_version('0.8')

# 1.
config = {}

```

(continues on next page)

(continued from previous page)

```
# 2.
config['general'] = {}
config['general']['number_of_iter'] = 5
config['general']['save_images'] = False

# 3.
config['sources'] = {}
config['sources']['source'] = {}
config['sources']['source']['class_name'] = 'XicsrtSourceDirected'
config['sources']['source']['intensity'] = 1e3
config['sources']['source']['wavelength'] = 3.9492
config['sources']['source']['spread'] = np.radians(5.0)

# 4.
config['optics'] = {}
config['optics']['detector'] = {}
config['optics']['detector']['class_name'] = 'XicsrtOpticDetector'
config['optics']['detector']['origin'] = [0.0, 0.0, 1.0]
config['optics']['detector']['zaxis']  = [0.0, 0.0, -1]
config['optics']['detector']['xsize']  = 0.2
config['optics']['detector']['ysize']  = 0.2

# 5.
results = xicsrt.raytrace(config)

```

6.2.2 example_01

Download a Jupyter notebook with this example: `example_01.ipynb`.

Please follow the source comments for description and instructions.

Source Code

```
# -*- coding: utf-8 -*-
"""
..  Authors:
    Novimir Antoniuk Pablant <npablant@pppl.gov>


A slightly more complicated example with an x-ray Bragg reflection from a
spherical crystal.

This configuration has a point source, a spherical crystal, and a detector.
"""

import numpy as np
import xicsrt
xicsrt.warn_version('0.8')

# 1.
config = dict()

# 2.
```

(continues on next page)

(continued from previous page)

```
config['general'] = {}
config['general']['number_of_iter'] = 5
config['general']['save_images'] = False

# 3.
config['sources'] = {}
config['sources']['source'] = {}
config['sources']['source']['class_name'] = 'XicsrtSourceDirected'
config['sources']['source']['intensity'] = 1e4
config['sources']['source']['wavelength'] = 3.9492
config['sources']['source']['spread'] = np.radians(10.0)
config['sources']['source']['xsize'] = 0.00
config['sources']['source']['ysize'] = 0.00
config['sources']['source']['zsize'] = 0.00

# 4.
config['optics'] = {}
config['optics']['crystal'] = {}
config['optics']['crystal']['class_name'] = 'XicsrtOpticCrystalSpherical'
config['optics']['crystal']['check_size'] = True
config['optics']['crystal']['origin'] = [0.0, 0.0,         0.80374151]
config['optics']['crystal']['zaxis']  = [0.0, 0.59497864, -0.80374151]
config['optics']['crystal']['xsize']  = 0.2
config['optics']['crystal']['ysize']  = 0.2
config['optics']['crystal']['radius'] = 1.0

# Rocking curve FWHM in radians.
# This is taken from x0h for quartz 1,1,-2,0
# Darwin Curve, sigma: 48.070 urad
# Darwin Curve, pi:    14.043 urad
config['optics']['crystal']['crystal_spacing'] = 2.45676
config['optics']['crystal']['rocking_type'] = 'gaussian'
config['optics']['crystal']['rocking_fwhm'] = 48.070e-6

# 5.
config['optics']['detector'] = {}
config['optics']['detector']['class_name'] = 'XicsrtOpticDetector'
config['optics']['detector']['origin'] = [0.0,  0.76871290, 0.56904832]
config['optics']['detector']['zaxis']  = [0.0, -0.95641806, 0.29200084]
config['optics']['detector']['xsize']  = 0.4
config['optics']['detector']['ysize']  = 0.2

# 6.
results = xicsrt.raytrace(config)
```

6.2.3 example_02

Download a Jupyter notebook with this example: `example_02.ipynb`.

Please follow the source comments for description and instruction.

Source Code

```
# -*- coding: utf-8 -*-
"""
..  Authors:
    Novimir Antoniuk Pablant <npablant@pppl.gov>

An example showing how to define a complex aperture.
"""

import numpy as np
import xicsrt
xicsrt.warn_version('0.8')

config = {}

config['general'] = {}
config['general']['number_of_iter'] = 5
config['general']['save_images'] = False
config['general']['random_seed'] = 0

config['sources'] = {}
config['sources']['source'] = {}
config['sources']['source']['class_name'] = 'XicsrtSourceDirected'
config['sources']['source']['intensity'] = 1e3
config['sources']['source']['wavelength'] = 3.9492
config['sources']['source']['angular_dist'] = 'isotropic_xy'
config['sources']['source']['spread'] = np.radians(6.0)

config['optics'] = {}
config['optics']['aperture'] = {}
config['optics']['aperture']['class_name'] = 'XicsrtOpticAperture'
config['optics']['aperture']['origin'] = [0.0, 0.0, 0.8]
config['optics']['aperture']['zaxis'] = [0.0, 0.0, -1]
config['optics']['aperture']['aperture']=[
    {'shape':'circle', 'size':[0.075], 'logic':'and'},
    {'shape':'circle', 'size':[0.065], 'origin':[-0.010, -0.01],  'logic':'not'},
    {'shape':'circle', 'size':[0.048], 'origin':[-0.027, -0.01],  'logic':'or'},
    {'shape':'circle', 'size':[0.044], 'origin':[-0.032, -0.015], 'logic':'not'},
    {'shape':'circle', 'size':[0.034], 'origin':[-0.041, -0.013], 'logic':'or'},
    {'shape':'circle', 'size':[0.032], 'origin':[-0.045, -0.018], 'logic':'not'},
    {'shape':'circle', 'size':[0.025], 'origin':[-0.038, -0.020], 'logic':'or'},
    ]

config['optics']['detector'] = {}
config['optics']['detector']['class_name'] = 'XicsrtOpticDetector'
config['optics']['detector']['origin'] = [0.0, 0.0, 1.0]
config['optics']['detector']['zaxis'] = [0.0, 0.0, -1]
config['optics']['detector']['xsize'] = 0.2
config['optics']['detector']['ysize'] = 0.2


results = xicsrt.raytrace(config)
```

example_00

A simple example consisting only of a point source and a spherical crystal.

example_01

A slightly more complicated example with x-rays. This configuration has a point source, a spherical crystal, and a detector.

example_02

An example showing how to define a complex aperture.

6.3 XICSRT API Documentation

6.3.1 command

Command Line Interface

A command line interface for the XICSRT raytracer.

usage:
xicsrt [-h] [–numruns N] [–numiter N] [–seed N] [–save]
[–images] [–suffix STR] [–path STR] [–multiprocessing]
[–processes N] [–version] [–debug]
[config_file]

xicsrt version 0.8.8

description:
Perform an XICSRT raytrace from the command line.

The input to this command should be an XICSRT configuration dictionary
in json format. (Pickle and hdf5 formats are also supported.)

example 1:
xicsrt config.json

example 2:
python -m xicsrt config.json

positional arguments:
config_file The path to the configuration file for this run.

optional arguments:
-h, –help show this help message and exit
–numruns N Number of runs.
–numiter N Number of iterations per run.
–seed N The random seed to use.
–save Save the results.

–images Save intersection images.
–suffix STR A suffix to add to the output files.
–path STR Directory in which to store output.
–multiprocessing, –mp
Use multiprocessing.
–processes N Number of processes to use for muliprocessing.
–version Show the version number.
–debug Show debugging output in the log.

run()
Parse command line arguments and run XICSRT.

6.3.2 modules

xicsrt

xicsrt

The top level module, *xicsrt*, provides convenient access to several functions that are defined in other modules.

raytrace(*config*)
Perform a series of ray tracing runs.

Each run will rebuild all objects, reset the random seed and then perform the requested number of iterations.

If the option 'save_images' is set, then images will be saved at the completion of each run. The saving of these run images is one reason to use this routine rather than just increasing the number of iterations: periodic outputs during long computations.

Also see *raytrace()* for a multiprocessing version of this routine.

raytrace_mp(*config*, *processes=None*)
Perform a series of ray tracing runs using the *multiprocessing* cpu pool.

Each run will rebuild all objects, reset the random seed and then perform the requested number of iterations.

If the option 'save_images' is set, then images will be saved at the completion of each run.

Also see *raytrace()* for a single process version of this routine.

xicsrt_raytrace

xicsrt.xicsrt_raytrace

Entry point to XICSRT. Contains the main functions that are called to perform raytracing.

raytrace(*config*)
Perform a series of ray tracing runs.

Each run will rebuild all objects, reset the random seed and then perform the requested number of iterations.

If the option 'save_images' is set, then images will be saved at the completion of each run. The saving of these run images is one reason to use this routine rather than just increasing the number of iterations: periodic outputs during long computations.

Also see *raytrace()* for a multiprocessing version of this routine.

raytrace_single(*config*, *_internal=False*)
Perform a single raytrace run consisting of multiple iterations.

If history is enabled, sort the rays into those that are detected and those that are lost (found and lost). The found ray history will be returned in full. The lost ray history will be truncated to allow analysis of lost ray pattern while still limiting memory usage.

private keywords

_internal [bool (False)] Used when calling this function from *raytrace* as part of the execution of multiple runs. Controls how *history_max_lost* is handled along with how *save_config* and *save_results* are interpreted.

combine_raytrace(*input_list*)
Produce a combined results dictionary from a list of raytrace results.

Example

results_1 = xicsrt.raytrace(config_1) results_2 = xicsrt.raytrace(config_2) results = xicsrt_raytrace.combine_raytrace([results_1, results_2])

Utility Members

These functions are used internally and are not typically needed by the user. They may, however, be useful in some circumstances. An example is combining of multiple separate raytracing runs.

combine_raytrace(*input_list*)
Produce a combined results dictionary from a list of raytrace results.

Example

results_1 = xicsrt.raytrace(config_1) results_2 = xicsrt.raytrace(config_2) results = xicsrt_raytrace.combine_raytrace([results_1, results_2])

check_config(*config*)
Check the general section of the configuration dictionary.

print_raytrace(*results*)
Print out some information and statistics from the raytracing results.

Private Members

_raytrace_iter(*config*, *sources*, *optics*)
Perform a single iteration of raytracing with the given sources and optics. The returned rays are unsorted.

_sort_raytrace(*input*, *max_lost=None*)
Sort the rays into 'lost' and 'found' rays, then truncate the number of lost rays.

xicsrt_multiprocessing

xicsrt.xicsrt_multiprocessing

raytrace(*config*, *processes=None*)
Perform a series of ray tracing runs using the *multiprocessing* cpu pool.

Each run will rebuild all objects, reset the random seed and then perform the requested number of iterations.

If the option 'save_images' is set, then images will be saved at the completion of each run.

Also see *raytrace()* for a single process version of this routine.

Private Members

xicsrt_public

xicsrt.xicsrt_public

A collections of routines to simplify interactive use of XICSRT.

get_element(*config_user*, *name*, *section=None*, *initialize=True*)
Retrieves an raytracing element (source, optic or filter) object.

Private Members

_find_element_section(*config*, *name*)
Search config for the given element name and return the section.

xicsrt_io

xicsrt.xicsrt_io

Description

Handle reading and writing of files for XICSRT.

load_config(*filename*)

save_config(*config*, *filename=None*, *path=None*, *mkdir=None*, *overwrite=None*)

save_results(*output*, *filename=None*, *path=None*, *mkdir=None*, *overwrite=None*)

load_results(*filename=None*, *path=None*, *config=None*)
Load an XICSRT results file.

Keywords

filename The name of the file to load. filename can be a full path; otherwise if path is given it will be prepended to the filename. If filename is not provided and a config is given, then a filename will be auto generated based on the config.

path If provided the path will be prepended to the filename.

config If provided (and filename is not), then the filename will be auto generated using the config settings.

save_images(*output*, *rotate=True*, *path=None*, *mkdir=None*)
Save images from the raytracing output.

generate_filename(*config*, *kind=None*, *name=None*, *path=None*)

path_exists(*path*)

Private Members

_dict_from_file(*filename*)

_file_from_dict(*data*, *filename*, *mkdir=False*, *overwrite=False*)

_make_output_path(*config*)

_make_path(*filename*)

xicsrt_config

xicsrt.xicsrt_config

default_config()

number_of_iter [int (1)] Number of raytracing iterations to perform for each raytracing run. Iterations are performed in a single process and with a single initialization of the raytracing objects and the random seed. All iterations will be combined before saving images or output files.

number_of_runs [int (1)] Number of raytracing runs to perform. For each run, the specified number of iterations will be performed. Raytracing runs can be performed in separate processes enabling the use of multiprocessing. At the start of each run all raytracing objects will be created and initialized. Images and output files will be saved after each run.

random_seed [int (None)] A random seed to initialize the pseudo-random number generator. If random_seed is equal to None then a random seed will be provided by the python/numpy internals. When a integer seed is provided and multiple raytracing runs are performed, the random seed will be incremented by one for each successive run. This seed can be used to make raytracing runs reproducible on the level of individual rays; however, reproducibility is not guaranteed between different versions of XICSRT or different versions of Python. Random seed initialization performed using *np.random.seed()*.

pathlist [list (list())] A list of paths that contain user defined raytracing modules (sources, optics, filters, apertures, etc.). These paths will be searched for filenames that match the requested 'class_name' in the object config. User defined paths are searched before the builtin or contrib paths.

pathlist_default [list] A list of paths to the builtin and contributed raytracing objects. This option should not be changed by the user, but can be useful for inspection to see what directories are actually being searched.

strict_config_check [bool (True)] Use strict checking to ensure that the config dictionary only contains valid options for the given object. This helps avoid unexpected behavior as well and alerting to typos in the configuration. When set to *False*, unrecognized config options will be quietly ignored.

output_path [str (None)] Path in which to save images and results from the raytracing run. If note then the current working path will be used. Use the option *make_directories* to control directory creation.

output_prefix [str ("xicsrt")] Filenames for images and results are automatically generated. Use this option to add a prefix to the beginning of all filenames. An underscore will be automatically added after the prefix. For images the following format will be used: "prefix_optic_suffix_run.tif". For example: "xicsrt_detector_scan01_0000.tif".

output_suffix [str (None)] If present this string will be added to automatically generated filenames after the optic name and before the run_suffix. See the option *output_prefix* for more information.

output_run_suffix [str (None)] This option is used internally and should not be set by the user.

image_ext [str ('.tif')] Controls the file format of the saved images. Any format supported by the *pillow* package can be used; .tif images are recommended.

results_ext [str ('.hdf5')] Controls the file format for saving the results dictionary. Currently hdf5 (.hdf5, .h5), pickle (.pickle, .pkl) and json (.json) file formats are supported. The json format is not recommended as it may lead to very large file sizes!

make_directories [bool (False)] Controls whether the output path should be created if it does not already exist. If False an error will be raised if the output path does not exist.

keep_meta [bool (True)] Controls whether to calculate and keep metadata and statistics relating to the raytracing.

keep_images [bool (True)] Controls whether to generate and keep pixelated images of the ray intersections at each optic. Control of image generation for individual optics can also be set within the object specific config sections.

keep_history [bool (True)] Controls whether to calculate and keep the raytracing history. Rays will be sorted into 'lost' and 'found' rays, where found rays are those that reach the final optic element. The 'found' ray history will be kept in full, the 'lost' ray history will be truncated (see option *history_max_lost*).

The ray history provides a great deal of information about the raytracing and enables ray plotting (2d and 3d) and post processing. However, turning on the ray history also *greatly* increases memory usage since the rays must be duplicated and saved for every optical element. If only final intersection images are required, consider setting this option to *False* to improve raytracing performance.

history_max_lost [int (10000)] Number of 'lost' rays to retain in the raytrace history. Lost rays are those that are launched from the source but that do not reach the last optical element (typically a detector). For many x-ray raytracing applications the number of lost rays will be very large, and retention of all lost rays would quickly exhaust available system memory. To avoid memory issues, while still retaining some lost rays for diagnostic purposes, a randomized truncation of the lost rays is performed.

save_config [bool (False)] Option whether or not to save the config dictionary. Output format is currently limited to json format (hdf5 and pickle coming soon).

save_images [bool (False)] Controls saving of images. Images will be saved for every run, and a combined image will be saved at the conclusion of all runs. Control of output for individual optics can be set within the object specific config sections. Image format will be determined by the option *image_ext* (default .tif). Images will be saved to the *output_path*.

save_results [bool (False)] Controls saving of the raytracing results dictionary. The contents of the results dictionary are controlled by *keep_meta*, *keep_images* and *keep_history*. Results will only be saved after all runs are completed. Output format will be determined by the option *results_ext* (default .hdf5). OUtput will be saved to the *output_path*.

print_results [bool (True)] Control text output to terminal of raytracing summary and optics specific information. (Note: control of logging/debugging output is controlled through a separate option that is not yet implemented.)

version: string The version number of xicsrt when this config was created. This option is set internally and should not be modified by the user.

get_config(*config_user=None*)

refresh_config(*config_new*)

When a config file is loaded from a another system or from a different user it may contain default values that are not appropriate for the current environment. This function will overwrite these options with new default values where appropriate.

get_pathlist_default()

Return a list of the default sources and optics directories. These locations will be based on the location of this

module.

config_to_numpy(*obj*)

config_from_numpy(*obj*)

update_config(*config*, *config_new*, *strict=None*, *update=None*, *ignore_none=None*)
Overwrite any values in the given config dict with the values in the config_new dict. This will be done recursively to allow nested dictionaries.

keywords:

strict (True) If True then an error will be raised if an option is found in the user dict that is not found in the default dict.

update (False) If True any unmatched options that are found will be retained. When False they will simply be ignored. This option has no effect unless strict = False.

ignore_none (False) If True any options found in config_new with a value of None will be ignored.

Private Members

_add_pathlist_builtin(*pathlist*)

_add_pathlist_contrib(*pathlist*)

_update_config_dict(*config*, *config_new*, *strict=None*, *update=None*, *ignore_none=None*)
Recursive worker function for *update_config*.

6.3.3 packages

xicsrt.sources

Contains the built-in source objects.

Additional sources are available as part of the xicsrt_contrib package.

Built-in Source Objects

XicsrtPlasmaCubic

xicsrt.sources._XicsrtPlasmaCubic.XicsrtPlasmaCubic

New Members

class XicsrtPlasmaCubic(**args*, ***kwargs*)
Bases: *xicsrt.sources._XicsrtPlasmaGeneric.XicsrtPlasmaGeneric*

A cubic plasma.

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution. Warning: Only the 'isotropic' distribution is currently supported!

spread [float (None) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius [float (None) [meters]] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

bundle_generate(*bundle_input*)

New Private Members

class XicsrtPlasmaCubic

Inherited Members

class XicsrtPlasmaCubic

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

bundle_filter(*bundle_input*)

bundle_generate(*bundle_input*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

generate_rays()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

setup_bundles()

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtPlasmaCylindrical

xicsrt.sources._XicsrtPlasmaCylindrical.XicsrtPlasmaCylindrical

New Members

class XicsrtPlasmaCylindrical(**args*, ***kwargs*)
Bases: *xicsrt.sources._XicsrtPlasmaGeneric.XicsrtPlasmaGeneric*

A cylindrical plasma oriented along the Y axis.

> **Warning:** This class is broken and out of date and needs to be updated.

This class is meant only to be used as an example for generating more complicated classes for specific plasmas.

plasma normal = absolute X plasma x orientation = absolute Z plasma y orientation = absolute Y

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution. Warning: Only the 'isotropic' distribution is currently supported!

spread [float (None) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in *XicsrtSourceGeneric* for detailed documentation.

spread_radius [float (None) [meters]] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

bundle_generate(*bundle_input*)

New Private Members

class XicsrtPlasmaCylindrical

Inherited Members

class XicsrtPlasmaCylindrical

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

bundle_filter(*bundle_input*)

bundle_generate(*bundle_input*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

generate_rays()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

setup_bundles()

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtPlasmaGeneric

xicsrt.sources._XicsrtPlasmaGeneric.XicsrtPlasmaGeneric

New Members

class XicsrtPlasmaGeneric(**args*, ***kwargs*)

Bases: *xicsrt.objects._GeometryObject.GeometryObject*

A generic plasma object.

Plasma object will generate a set of ray bundles where each ray bundle has the properties of the plasma at one particular real-space point.

Each bundle is modeled by a SourceFocused object.

Note: If a *voxel* type bundle is used rays may be generated outside of the defined plasma volume (as defined by xsize, ysize and zsize). The bundle *centers* are randomly distributed throughout the plasma volume, but this means that if a bundle is (randomly) placed near the edges of the plasma then the bundle voxel volume may extend past the plasma boundary. This behavior is expected. If it is important to have a sharp plasma boundary then consider using the 'point' bundle_type instead.

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution. Warning: Only the 'isotropic' distribution is currently supported!

spread [float (None) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in *XicsrtSourceGeneric* for detailed documentation.

spread_radius [float (None) [meters]] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

initialize()
Initialize the object.

setup_bundles()

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

bundle_generate(*bundle_input*)

bundle_filter(*bundle_input*)

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

generate_rays()

New Private Members

class XicsrtPlasmaGeneric

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Inherited Members

class XicsrtPlasmaGeneric

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

bundle_filter(*bundle_input*)

bundle_generate(*bundle_input*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

generate_rays()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

setup_bundles()

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtPlasmaToroidal

xicsrt.sources._XicsrtPlasmaToroidal.XicsrtPlasmaToroidal

New Members

class XicsrtPlasmaToroidal(**args*, ***kwargs*)
Bases: *xicsrt.sources._XicsrtPlasmaGeneric.XicsrtPlasmaGeneric*

A plasma object with toroidal geometry and a circular cross-section.

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution. Warning: Only the 'isotropic' distribution is currently supported!

spread [float (None) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in *XicsrtSourceGeneric* for detailed documentation.

spread_radius [float (None) [meters]] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *Xicsrt-SourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

flx_from_car(*point_car*)

rho_from_car(*point_car*)

car_from_flx(*point_flx*)

bundle_generate(*bundle_input*)

New Private Members

class XicsrtPlasmaToroidal

Inherited Members

class XicsrtPlasmaToroidal

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

bundle_filter(*bundle_input*)

bundle_generate(*bundle_input*)

car_from_flx(*point_flx*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

flx_from_car(*point_car*)

generate_rays()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

rho_from_car(*point_car*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

setup_bundles()

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtPlasmaToroidalDatafile

xicsrt.sources._XicsrtPlasmaToroidalDatafile.XicsrtPlasmaToroidalDatafile

New Members

class XicsrtPlasmaToroidalDatafile(**args*, ***kwargs*)
Bases: *xicsrt.sources._XicsrtPlasmaToroidal.XicsrtPlasmaToroidal*

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution. Warning: Only the 'isotropic' distribution is currently supported!

spread [float (None) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius [float (None) [meters]] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

get_emissivity(*rho*)

get_temperature(*rho*)

New Private Members

class XicsrtPlasmaToroidalDatafile

Inherited Members

class XicsrtPlasmaToroidalDatafile

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

bundle_filter(*bundle_input*)

bundle_generate(*bundle_input*)

car_from_flx(*point_flx*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

create_sources(*bundle_input*)
Generate rays from a list of bundles.

bundle_input a list containing dictionaries containing the locations, emissivities, temperatures and velocitities and of all ray bundles to be emitted.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

angular_dist [string ('isotropic')] The type of angular distribution to use for the emitted rays. Available distributions: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian', and 'gaussian_flat'. See *XicsrtSourceGeneric* for documentation of each distribution.

Warning: Only the 'isotropic' distribution is currently supported!

spread: float (None) [radians] The angular spread for the emission cone. The spread defines the half-angle of the cone. See 'angular_dist' in `XicsrtSourceGeneric` for detailed documentation.

spread_radius: float (None) [meters] If specified, the spread will be calculated for each bundle such that the spotsize at the target matches the given radius. This is useful when working with very extended plasma sources. This options is incompatible with 'spread'.

use_poisson No documentation yet. Please help improve XICSRT!

wavelength_dist [string ('voigt')] No documentation yet. Please help improve XICSRT!

wavelength [float (1.0) [Angstroms]] No documentation yet. Please help improve XICSRT!

mass_number [float (1.0) [au]] No documentation yet. Please help improve XICSRT!

linewidth [float (0.0) [1/s]] No documentation yet. Please help improve XICSRT!

emissivity [float (0.0) [ph/m^3]] No documentation yet. Please help improve XICSRT!

temperature [float (0.0) [eV]] No documentation yet. Please help improve XICSRT!

velocity [float (0.0) [m/s]] No documentation yet. Please help improve XICSRT!

time_resolution [float (1e-3) [s]] No documentation yet. Please help improve XICSRT!

bundle_type [string ('voxel')] Define how the origin of rays within the bundle should be distributed. Available options are: 'voxel' or 'point'.

bundle_volume [float (1e-3) [m^3]] The volume in which the rays within the bundle should distributed. if bundle_type is 'point' this will not affect the distribution, though it will still affect the number of bundles if bundle_count is set to None.

bundle_count [int (None)] The number of bundles to generate. If set to *None* then this number will be automatically determined by volume/bundle_volume. This default means that each bundle represents exactly the given *bundle_volume* in the plasma. For high quality raytracing studies this value should generally be set to a value much larger than volume/bundle_volume!

max_rays [int (1e7)] No documentation yet. Please help improve XICSRT!

max_bundles [int (1e7)] No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

flx_from_car(*point_car*)

generate_rays()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

get_emissivity(*rho*)

get_temperature(*rho*)

get_velocity(*rho*)

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

rho_from_car(*point_car*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

setup_bundle_spread(*bundle_input*)
Calculate the spread and solid angle for each bundle.

If the config option 'spread_radius' is provide the spread will be determined for each bundle by a spotsize at the target.

Note: Even if the idea of a spread radius is added to the generic source object we still need to calculate and save the results here so that we can correctly calcuate the bundle intensities.

setup_bundles()

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtSourceDirected

xicsrt.sources._XicsrtSourceDirected.XicsrtSourceDirected

New Members

class XicsrtSourceDirected(**args*, ***kwargs*)
Bases: *xicsrt.sources._XicsrtSourceGeneric.XicsrtSourceGeneric*

An extended rectangular ray source with rays emitted in a preferred direction.

This is similar to the SourceGeneric except that an explicit direction can be provided instead of always emitting rays along the z-axis.

This is different from a SourceFocused in that the emission cone is always aimed in a fixed direction for every location in the source. The SourceFocused instead aims the emission cone at a specific target so that the aiming direction changes for different locations within the source.

Configuration Options:

direction The direction in which to emit rays. This direction will define the center of the emission code with angular spread *spread*.

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

spatial_dist [string ('uniform')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'uniform'):

uniform
Uniform spatial distribution of rays within a rectangular cuboid defined by xsize, ysize and zsize. The sizes are interpreted as full widths.

gaussian
Gaussian spatial distribution of rays with a fwhm in each dimension defined by xsize, ysize and zsize. The sizes are interpreted as full-width-at-half-max (fwhm).

angular_dist [string ('isotropic')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'isotropic'):

isotropic
: Isotropic emission (uniform spherical) emitted in a cone (circular cross-section) with a half-angle of 'spread'. The axis of the emission cone is aligned along the z-axis. 'spread' must be a single value (scalar or 1-element array).

isotropic_xy
: Isotropic emission (uniform spherical) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values:

  s or [s]
  : A single value that will be used for both the x and y directions.

  [x, y]
  : Two values values that will be used for the x and y directions.

  [xmin, xmax, ymin, ymax]
  : For values that define the asymmetric exent in x and y directions. Example: [-0.1, 0.1, -0.5, 0.5]

flat
: Flat emission (uniform planar) emitted in a cone (circular cross-section) with a half-angle of 'spread'.

flat_xy
: Flat emission (uniform planar) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values, see above.

gaussian
: Emission with angles away from the z-axis having a Gaussian distribution (circular cross-section). The 'spread' defines the Half-Width-at-Half-Max (HWHM) of the distribution. 'spread' must be a single value (scalar or 1-element array).

gaussian_flat
: !! Not implemented !!
  Cross-section of emission (intersection with constant-z plane) will have a Gaussian distribution.

spread [float or array (np.pi) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the emission cone. See 'angular_dist' for detailed documentation.

intensity [int or float] The number of rays for this source to emit. This should be an integer value unless *use_poisson = True*. Note: If filters are attached, this will be the number of rays emitted before filtering.

use_poisson [bool (False)] If *True* the *intenisty* will be treated as the expected value for a Poisson distribution and the number of rays will be randomly picked from a Poisson distribution. This is setting is typically only used internally for Plasma sources.

wavelength_dist [str ('voigt')] The type of wavelength distribution for this source. Possible values are: 'voigt', 'uniform', 'monochrome'. Note: A monochrome distribution can also be achieved by using a 'voigt' distribution with zero linewidth and temperature.

wavelength [float (1.0) [angstroms]] Only used if *wavelength_dist = "monochrome" or "voigt"* Central wave-

length of the distribution, in Angstroms.

wavelength_range [tuple [angstroms]] Only used if *wavelength_dist = "uniform"* The wavelength range of the distribution, in Angstroms. Must be a 2 element tuple, list or array: (min, max).

linewidth [float (0.0) [1/s]] Only used if *wavelength_dist = "voigt"* The natural width of the emission line. This will control the Lorentzian contribution to the the overall Voigt profile. If linewidth == 0, the resulting wavelength distribution will be gaussian. To convert from a fwhm in [eV]: linewidth = 2*pi*e/(h*fwhm_ev) To translate from linewidth to gamma in the Voigt equation: gamma = linewidth * wavelength**2 / (4*pi*c*1e10)

mass_number [float (1.0) [au]] Only used if *wavelength_dist = "voigt"* The mass of the emitting atom in atomic units (au). This mass in used to convert temperature into line width. See temperature option.

temperature [float (0.0) [eV]] Only used if *wavelength_dist = "voigt"* The temperature of the emission line. This will control the Gaussian contribution to the overall Voigt profile. If temperature == 0, the resulting wavelength distribution will be Lorentzian. To translate from temperature to sigma in the Voigt equation: sigma = np.sqrt(temperature/mass_number/amu_kg/c**2*ev_J)*wavelength

velocity No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

direction The direction in which to emit rays. This direction will define the center of the emission code with angular spread *spread*.

initialize()
Initialize the object.

make_normal()

New Private Members

class XicsrtSourceDirected

Inherited Members

class XicsrtSourceDirected

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

direction The direction in which to emit rays. This direction will define the center of the emission code with angular spread *spread*.

generate_direction(*origin*)

generate_mask()

generate_origin()

generate_rays()

generate_wavelength(*direction*)

generate_weight()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

make_normal()

point_to_external(*point_local*)

point_to_local(*point_external*)

random_direction(*normal*)

random_wavelength_cauchy(*size=None*)

random_wavelength_normal(*size=None*)

random_wavelength_voigt(*size=None*)

ray_filter(*rays*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtSourceFocused

xicsrt.sources._XicsrtSourceFocused.XicsrtSourceFocused

New Members

class XicsrtSourceFocused(**args*, ***kwargs*)

Bases: *xicsrt.sources._XicsrtSourceGeneric.XicsrtSourceGeneric*

An extended rectangular ray source that allows focusing towards a target.

This is different to a SourceDirected in that the emission cone is aimed at the target for every location in the source. The SourceDirected instead uses a fixed direction for emission.

Configuration Options:

target The target at which to aim the emission cone at each point in the source volume. The emission cone aimed at the target will have an angular spread defined by *spread*.

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

spatial_dist [string ('uniform')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'uniform'):

uniform
: Uniform spatial distribution of rays within a rectangular cuboid defined by xsize, ysize and zsize. The sizes are interpreted as full widths.

gaussian
: Gaussian spatial distribution of rays with a fwhm in each dimension defined by xsize, ysize and zsize. The sizes are interpreted as full-width-at-half-max (fwhm).

angular_dist [string ('isotropic')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'isotropic'):

isotropic
: Isotropic emission (uniform spherical) emitted in a cone (circular cross-section) with a half-angle of 'spread'. The axis of the emission cone is aligned along the z-axis. 'spread' must be a single value (scalar or 1-element array).

isotropic_xy

Isotropic emission (uniform spherical) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values:

s or [s]
: A single value that will be used for both the x and y directions.

[x, y]
: Two values values that will be used for the x and y directions.

[xmin, xmax, ymin, ymax]
: For values that define the asymmetric exent in x and y directions. Example: [-0.1, 0.1, -0.5, 0.5]

flat
: Flat emission (uniform planar) emitted in a cone (circular cross-section) with a half-angle of 'spread'.

flat_xy
: Flat emission (uniform planar) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values, see above.

gaussian
: Emission with angles away from the z-axis having a Gaussian distribution (circular cross-section). The 'spread' defines the Half-Width-at-Half-Max (HWHM) of the distribution. 'spread' must be a single value (scalar or 1-element array).

gaussian_flat
: !! Not implemented !! Cross-section of emission (intersection with constant-z plane) will have a Gaussian distribution.

spread [float or array (np.pi) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the emission cone. See 'angular_dist' for detailed documentation.

intensity [int or float] The number of rays for this source to emit. This should be an integer value unless *use_poisson = True*. Note: If filters are attached, this will be the number of rays emitted before filtering.

use_poisson [bool (False)] If *True* the *intenisty* will be treated as the expected value for a Poisson distribution and the number of rays will be randomly picked from a Poisson distribution. This is setting is typically only used internally for Plasma sources.

wavelength_dist [str ('voigt')] The type of wavelength distribution for this source. Possible values are: 'voigt', 'uniform', 'monochrome'. Note: A monochrome distribution can also be achieved by using a 'voigt' distribution with zero linewidth and temperature.

wavelength [float (1.0) [angstroms]] Only used if *wavelength_dist = "monochrome" or "voigt"* Central wavelength of the distribution, in Angstroms.

wavelength_range [tuple [angstroms]] Only used if *wavelength_dist = "uniform"* The wavelength range of the distribution, in Angstroms. Must be a 2 element tuple, list or array: (min, max).

linewidth [float (0.0) [1/s]] Only used if *wavelength_dist = "voigt"* The natural width of the emission line. This will control the Lorentzian contribution to the the overall Voigt profile. If linewidth == 0, the resulting wavelength distribution will be gaussian. To convert from a fwhm in [eV]: linewidth = 2*pi*e/(h*fwhm_ev) To translate from linewidth to gamma in the Voigt equation: gamma = linewidth * wavelength**2 / (4*pi*c*1e10)

mass_number [float (1.0) [au]] Only used if *wavelength_dist = "voigt"* The mass of the emitting atom in

atomic units (au). This mass in used to convert temperature into line width. See temperature option.

temperature [float (0.0) [eV]] Only used if *wavelength_dist = "voigt"* The temperature of the emission line. This will control the Gaussian contribution to the overall Voigt profile. If temperature == 0, the resulting wavelength distribution will be Lorentzian. To translate from temperature to sigma in the Voigt equation: sigma = np.sqrt(temperature/mass_number/amu_kg/c**2*ev_J)*wavelength

velocity No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

target The target at which to aim the emission cone at each point in the source volume. The emission cone aimed at the target will have an angular spread defined by *spread*.

generate_direction(*origin*)

make_normal_focused(*origin*)

New Private Members

class XicsrtSourceFocused

Inherited Members

class XicsrtSourceFocused

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

target The target at which to aim the emission cone at each point in the source volume. The emission cone aimed at the target will have an angular spread defined by *spread*.

generate_direction(*origin*)

generate_mask()

generate_origin()

generate_rays()

generate_wavelength(*direction*)

generate_weight()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

make_normal()

make_normal_focused(*origin*)

point_to_external(*point_local*)

point_to_local(*point_external*)

random_direction(*normal*)

random_wavelength_cauchy(*size=None*)

random_wavelength_normal(*size=None*)

random_wavelength_voigt(*size=None*)

ray_filter(*rays*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtSourceGeneric

xicsrt.sources._XicsrtSourceGeneric.XicsrtSourceGeneric

New Members

class XicsrtSourceGeneric(**args*, ***kwargs*)

Bases: *xicsrt.objects._GeometryObject.GeometryObject*

Configuration Options:

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

spatial_dist [string ('uniform')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'uniform'):

uniform
: Uniform spatial distribution of rays within a rectangular cuboid defined by xsize, ysize and zsize. The sizes are interpreted as full widths.

gaussian
: Gaussian spatial distribution of rays with a fwhm in each dimension defined by xsize, ysize and zsize. The sizes are interpreted as full-width-at-half-max (fwhm).

angular_dist [string ('isotropic')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'isotropic'):

isotropic
: Isotropic emission (uniform spherical) emitted in a cone (circular cross-section) with a half-angle of 'spread'. The axis of the emission cone is aligned along the z-axis. 'spread' must be a single value (scalar or 1-element array).

isotropic_xy
: Isotropic emission (uniform spherical) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values:
  s or [s]
  : A single value that will be used for both the x and y directions.
  [x, y]
  : Two values values that will be used for the x and y directions.
  [xmin, xmax, ymin, ymax]
  : For values that define the asymmetric exent in x and y directions. Example: [-0.1, 0.1, -0.5, 0.5]

flat
: Flat emission (uniform planar) emitted in a cone (circular cross-section) with a half-angle of 'spread'.

flat_xy

Flat emission (uniform planar) emitted in a truncated-cone
(rectangular cross-section) with different x and y half-angles.
'spread' can contain either 1, 2 or 4 values, see above.

gaussian
Emission with angles away from the z-axis having a Gaussian
distribution (circular cross-section). The 'spread' defines the
Half-Width-at-Half-Max (HWHM) of the distribution. 'spread' must
be a single value (scalar or 1-element array).

gaussian_flat
!! Not implemented !!
Cross-section of emission (intersection with constant-z plane) will
have a Gaussian distribution.

spread [float or array (np.pi) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the emission cone. See 'angular_dist' for detailed documentation.

intensity [int or float] The number of rays for this source to emit. This should be an integer value unless *use_poisson* = *True*. Note: If filters are attached, this will be the number of rays emitted before filtering.

use_poisson [bool (False)] If *True* the *intenisty* will be treated as the expected value for a Poisson distribution and the number of rays will be randomly picked from a Poisson distribution. This is setting is typically only used internally for Plasma sources.

wavelength_dist [str ('voigt')] The type of wavelength distribution for this source. Possible values are: 'voigt', 'uniform', 'monochrome'. Note: A monochrome distribution can also be achieved by using a 'voigt' distribution with zero linewidth and temperature.

wavelength [float (1.0) [angstroms]] Only used if *wavelength_dist* = *"monochrome" or "voigt"* Central wavelength of the distribution, in Angstroms.

wavelength_range [tuple [angstroms]] Only used if *wavelength_dist* = *"uniform"* The wavelength range of the distribution, in Angstroms. Must be a 2 element tuple, list or array: (min, max).

linewidth [float (0.0) [1/s]] Only used if *wavelength_dist* = *"voigt"* The natural width of the emission line. This will control the Lorentzian contribution to the the overall Voigt profile. If linewidth == 0, the resulting wavelength distribution will be gaussian. To convert from a fwhm in [eV]: linewidth = 2*pi*e/(h*fwhm_ev) To translate from linewidth to gamma in the Voigt equation: gamma = linewidth * wavelength**2 / (4*pi*c*1e10)

mass_number [float (1.0) [au]] Only used if *wavelength_dist* = *"voigt"* The mass of the emitting atom in atomic units (au). This mass in used to convert temperature into line width. See temperature option.

temperature [float (0.0) [eV]] Only used if *wavelength_dist* = *"voigt"* The temperature of the emission line. This will control the Gaussian contribution to the overall Voigt profile. If temperature == 0, the resulting wavelength distribution will be Lorentzian. To translate from temperature to sigma in the Voigt equation: sigma = np.sqrt(temperature/mass_number/amu_kg/c**2*ev_J)*wavelength

velocity No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

spatial_dist [string ('uniform')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'uniform'):

uniform
Uniform spatial distribution of rays within a rectangular cuboid defined by xsize, ysize and zsize. The sizes are interpreted as full widths.

gaussian
Gaussian spatial distribution of rays with a fwhm in each dimension defined by xsize, ysize and zsize. The sizes are interpreted as full-width-at-half-max (fwhm).

angular_dist [string ('isotropic')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'isotropic'):

isotropic
Isotropic emission (uniform spherical) emitted in a cone (circular cross-section) with a half-angle of 'spread'. The axis of the emission cone is aligned along the z-axis. 'spread' must be a single value (scalar or 1-element array).

isotropic_xy
Isotropic emission (uniform spherical) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values:

s or [s]
A single value that will be used for both the x and y directions.

[x, y]
Two values values that will be used for the x and y directions.

[xmin, xmax, ymin, ymax]
For values that define the asymmetric exent in x and y directions. Example: [-0.1, 0.1, -0.5, 0.5]

flat
Flat emission (uniform planar) emitted in a cone (circular cross-section) with a half-angle of 'spread'.

flat_xy
: Flat emission (uniform planar) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values, see above.

gaussian
: Emission with angles away from the z-axis having a Gaussian distribution (circular cross-section). The 'spread' defines the Half-Width-at-Half-Max (HWHM) of the distribution. 'spread' must be a single value (scalar or 1-element array).

gaussian_flat
: !! Not implemented !! Cross-section of emission (intersection with constant-z plane) will have a Gaussian distribution.

spread [float or array (np.pi) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the emission cone. See 'angular_dist' for detailed documentation.

intensity [int or float] The number of rays for this source to emit. This should be an integer value unless *use_poisson = True*.

Note: If filters are attached, this will be the number of rays emitted before filtering.

use_poisson [bool (False)] If *True* the *intenisty* will be treated as the expected value for a Poisson distribution and the number of rays will be randomly picked from a Poisson distribution. This is setting is typically only used internally for Plasma sources.

wavelength_dist [str ('voigt')] The type of wavelength distribution for this source. Possible values are: 'voigt', 'uniform', 'monochrome'.

Note: A monochrome distribution can also be achieved by using a 'voigt' distribution with zero linewidth and temperature.

wavelength [float (1.0) [angstroms]] Only used if *wavelength_dist = "monochrome" or "voigt"* Central wavelength of the distribution, in Angstroms.

wavelength_range: tuple [angstroms] Only used if *wavelength_dist = "uniform"* The wavelength range of the distribution, in Angstroms. Must be a 2 element tuple, list or array: (min, max).

linewidth [float (0.0) [1/s]] Only used if *wavelength_dist = "voigt"* The natural width of the emission line. This will control the Lorentzian contribution to the the overall Voigt profile. If linewidth == 0, the resulting wavelength distribution will be gaussian.

To convert from a fwhm in [eV]: linewidth = 2*pi*e/(h*fwhm_ev)

To translate from linewidth to gamma in the Voigt equation: gamma = linewidth * wavelength**2 / (4*pi*c*1e10)

mass_number [float (1.0) [au]] Only used if *wavelength_dist = "voigt"* The mass of the emitting atom in atomic units (au). This mass in used to convert temperature into line width. See temperature option.

temperature [float (0.0) [eV]] Only used if *wavelength_dist = "voigt"* The temperature of the emission line. This will control the Gaussian contribution to the overall Voigt profile. If temperature == 0, the resulting wavelength distribution will be Lorentzian.

To translate from temperature to sigma in the Voigt equation: sigma = np.sqrt(temperature/mass_number/amu_kg/c**2*ev_J)*wavelength

velocity No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

initialize()
Initialize the object.

generate_rays()

generate_origin()

generate_direction(*origin*)

make_normal()

random_direction(*normal*)

generate_wavelength(*direction*)

random_wavelength_voigt(*size=None*)

random_wavelength_normal(*size=None*)

random_wavelength_cauchy(*size=None*)

generate_weight()

generate_mask()

ray_filter(*rays*)

New Private Members

class XicsrtSourceGeneric

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Inherited Members

class XicsrtSourceGeneric

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

xsize The size of this element along the xaxis direction.

ysize The size of this element along the yaxis direction.

zsize The size of this element along the zaxis direction.

spatial_dist [string ('uniform')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'uniform'):

uniform
: Uniform spatial distribution of rays within a rectangular cuboid defined by xsize, ysize and zsize. The sizes are interpreted as full widths.

gaussian
: Gaussian spatial distribution of rays with a fwhm in each dimension defined by xsize, ysize and zsize. The sizes are interpreted as full-width-at-half-max (fwhm).

angular_dist [string ('isotropic')]

The type of angular distribution to use for the emitted rays.
Available distributions (default is 'isotropic'):

isotropic
: Isotropic emission (uniform spherical) emitted in a cone (circular cross-section) with a half-angle of 'spread'. The axis of the emission cone is aligned along the z-axis. 'spread' must be a single value (scalar or 1-element array).

isotropic_xy
: Isotropic emission (uniform spherical) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values:

  s or [s]
  : A single value that will be used for both the x and y directions.

  [x, y]
  : Two values values that will be used for the x and y directions.

  [xmin, xmax, ymin, ymax]
  : For values that define the asymmetric exent in x and y directions. Example: [-0.1, 0.1, -0.5, 0.5]

flat
: Flat emission (uniform planar) emitted in a cone (circular cross-section) with a half-angle of 'spread'.

flat_xy
: Flat emission (uniform planar) emitted in a truncated-cone (rectangular cross-section) with different x and y half-angles. 'spread' can contain either 1, 2 or 4 values, see above.

gaussian
: Emission with angles away from the z-axis having a Gaussian distribution (circular cross-section). The 'spread' defines the Half-Width-at-Half-Max (HWHM) of the distribution. 'spread' must be a single value (scalar or 1-element array).

gaussian_flat
: !! Not implemented !!

Cross-section of emission (intersection with constant-z plane) will have a Gaussian distribution.

spread [float or array (np.pi) [radians]] The angular spread for the emission cone. The spread defines the half-angle of the emission cone. See 'angular_dist' for detailed documentation.

intensity [int or float] The number of rays for this source to emit. This should be an integer value unless *use_poisson = True*.

Note: If filters are attached, this will be the number of rays emitted before filtering.

use_poisson [bool (False)] If *True* the *intenisty* will be treated as the expected value for a Poisson distribution and the number of rays will be randomly picked from a Poisson distribution. This is setting is typically only used internally for Plasma sources.

wavelength_dist [str ('voigt')] The type of wavelength distribution for this source. Possible values are: 'voigt', 'uniform', 'monochrome'.

Note: A monochrome distribution can also be achieved by using a 'voigt' distribution with zero linewidth and temperature.

wavelength [float (1.0) [angstroms]] Only used if *wavelength_dist = "monochrome" or "voigt"* Central wavelength of the distribution, in Angstroms.

wavelength_range: tuple [angstroms] Only used if *wavelength_dist = "uniform"* The wavelength range of the distribution, in Angstroms. Must be a 2 element tuple, list or array: (min, max).

linewidth [float (0.0) [1/s]] Only used if *wavelength_dist = "voigt"* The natural width of the emission line. This will control the Lorentzian contribution to the the overall Voigt profile. If linewidth == 0, the resulting wavelength distribution will be gaussian.

To convert from a fwhm in [eV]: linewidth = 2*pi*e/(h*fwhm_ev)

To translate from linewidth to gamma in the Voigt equation: gamma = linewidth * wavelength**2 / (4*pi*c*1e10)

mass_number [float (1.0) [au]] Only used if *wavelength_dist = "voigt"* The mass of the emitting atom in atomic units (au). This mass in used to convert temperature into line width. See temperature option.

temperature [float (0.0) [eV]] Only used if *wavelength_dist = "voigt"* The temperature of the emission line. This will control the Gaussian contribution to the overall Voigt profile. If temperature == 0, the resulting wavelength distribution will be Lorentzian.

To translate from temperature to sigma in the Voigt equation: sigma = np.sqrt(temperature/mass_number/amu_kg/c**2*ev_J)*wavelength

velocity No documentation yet. Please help improve XICSRT!

filters No documentation yet. Please help improve XICSRT!

generate_direction(*origin*)

generate_mask()

generate_origin()

generate_rays()

generate_wavelength(*direction*)

generate_weight()

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

make_normal()

point_to_external(*point_local*)

point_to_local(*point_external*)

random_direction(*normal*)

random_wavelength_cauchy(*size=None*)

random_wavelength_normal(*size=None*)

random_wavelength_voigt(*size=None*)

ray_filter(*rays*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

xicsrt.optics

Contains the built-in optics objects.

Additional optics are available as part of the xicsrt_contrib package.

Built-in Optics Objects

XicsrtOpticAperture

New Members

class XicsrtOpticAperture(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractNone.InteractNone*, *xicsrt.optics._ShapePlane.ShapePlane*

An optic that can be used to set an aperture.

All of the implementation for the aperture is in *TraceObject*, so nothing is needed in this subclass for the time being.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticAperture

Inherited Members

class XicsrtOpticAperture

__init__ (*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point (*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture (*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds (*X*, *mask*)

check_config ()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param ()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size (*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config ()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticDetector

New Members

class XicsrtOpticDetector(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractNone.InteractNone*, *xicsrt.optics._ShapePlane.ShapePlane*

A detector optic.

For now the detector class simply records intersections with a plane. In the future this class may be expanded to include effects such as quantum efficiency, readout noise, dark noise, etc.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticDetector

Inherited Members

class XicsrtOpticDetector

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much

easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticMeshCrystal

New Members

class XicsrtOpticMeshCrystal(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.optics._InteractCrystal.InteractCrystal*, *xicsrt.optics._ShapeMesh.ShapeMesh*

A meshgrid crystal optic.

Configuration Options:

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticMeshCrystal

Inherited Members

class XicsrtOpticMeshCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

Note: This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

Note: Actually at this point only 'xop' is supported. np 2020-10-13

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticMeshMirror

New Members

class XicsrtOpticMeshMirror(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMirror.InteractMirror*, *xicsrt.optics._ShapeMesh.ShapeMesh*

A meshgrid perfect mirror optic.

Configuration Options:

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticMeshMirror

Inherited Members

class XicsrtOpticMeshMirror

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()
mesh_points mesh_faces mesh_normals

mesh_coarse_points mesh_coarse_faces mesh_coarse_normals

mesh_interpolate mesh_refine

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticMeshMosaicCrystal

New Members

class XicsrtOpticMeshMosaicCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMosaicCrystal.InteractMosaicCrystal*, *xicsrt.optics._ShapeMesh.ShapeMesh*

A meshgrid mosaic crystal optic.

Configuration Options:

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff [float (None)] A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticMeshMosaicCrystal

Inherited Members

class XicsrtOpticMeshMosaicCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff: float (None) A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Model reflections from a mosaic crystal using a multi-layer model. This is meant to simulate the penetration of x-rays into the HOPG until the rays either encounter a crystalite that satisfies the Bragg condition or get absorbed. This method of calculation replicates both the HOPG 'focusing' qualities as well as the expected throughput.

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

mosaic_normals(*normals*, *mask*, *copy=True*)
Add mosaic spread to the normals. Generates a list of crystallite normal vectors in a Gaussian distribution around the nominal surface normals.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticMeshSphericalCrystal

New Members

class XicsrtOpticMeshSphericalCrystal(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.optics._InteractCrystal.InteractCrystal*, *xicsrt.optics._ShapeMeshSphere.ShapeMeshSphere*

A meshgrid spherical crystal optic. This Optic is only meant to be used as an example of how to implement geometry using a meshgrid. The analytical Optic `ShapeSphere` should be used for all normal raytracing purposes.

The geometry of spherical mesh was implemented as a separate Shape object (`ShapeMeshSphere`) to allow mix-and-match with various Interactions. It would have also been possible to simply inherit `XicsrtOpticMeshCrystal` and define the geometry here instead. Doing so would have avoided the need to create two separate classes and files, but would have limit reuse of the defined geometry.

Configuration Options:

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

radius [float (1.0)] The radius of the sphere.

mesh_size [(float, float) ((11,11))] The number of mesh points in the x and y directions.

mesh_coarse_size [(float, float) ((5,5))] The number of mesh points in the x and y directions.

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticMeshSphericalCrystal

Inherited Members

class XicsrtOpticMeshSphericalCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

Note: This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

---

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

Note: Actually at this point only 'xop' is supported. np 2020-10-13

---

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

generate_mesh(*meshsize*)
Create a spherical meshgrid in local coordinates.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticPlanarCrystal

New Members

class XicsrtOpticPlanarCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractCrystal.InteractCrystal*, *xicsrt.optics._ShapePlane.ShapePlane*

A planar crystal optic.

Configuration Options:

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticPlanarCrystal

Inherited Members

class XicsrtOpticPlanarCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

Note: This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

Note: Actually at this point only 'xop' is supported. np 2020-10-13

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticPlanarMirror

New Members

class XicsrtOpticPlanarMirror(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.optics._InteractMirror.InteractMirror*, *xicsrt.optics._ShapePlane.ShapePlane*

A planar perfect mirror optic.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticPlanarMirror

Inherited Members

class XicsrtOpticPlanarMirror

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticPlanarMosaicCrystal

New Members

class XicsrtOpticPlanarMosaicCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMosaicCrystal.InteractMosaicCrystal*, *xicsrt.optics._ShapePlane.ShapePlane*

A planar mosaic crystal optic.

Configuration Options:

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff [float (None)] A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticPlanarMosaicCrystal

Inherited Members

class XicsrtOpticPlanarMosaicCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff: float (None) A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Model reflections from a mosaic crystal using a multi-layer model. This is meant to simulate the penetration of x-rays into the HOPG until the rays either encounter a crystalite that satisfies the Bragg condition or get absorbed. This method of calculation replicates both the HOPG 'focusing' qualities as well as the expected throughput.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mosaic_normals(*normals*, *mask*, *copy=True*)
Add mosaic spread to the normals. Generates a list of crystallite normal vectors in a Gaussian distribution around the nominal surface normals.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticSphericalCrystal

New Members

class XicsrtOpticSphericalCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractCrystal.InteractCrystal*, *xicsrt.optics._ShapeSphere.ShapeSphere*

A spherical crystal optic.

Configuration Options:

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticSphericalCrystal

Inherited Members

class XicsrtOpticSphericalCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

Note: This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

---

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

Note: Actually at this point only 'xop' is supported. np 2020-10-13

---

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calulate the distance to the intersection of the rays with the spherical optic.

This calculation is copied from: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticSphericalMirror

New Members

class XicsrtOpticSphericalMirror(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMirror.InteractMirror*, *xicsrt.optics._ShapeSphere.ShapeSphere*

A spherical perfect mirror optic.

Configuration Options:

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticSphericalMirror

Inherited Members

class XicsrtOpticSphericalMirror

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much

easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calulate the distance to the intersection of the rays with the spherical optic.

This calculation is copied from: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtOpticSphericalMosaicCrystal

New Members

class XicsrtOpticSphericalMosaicCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMosaicCrystal.InteractMosaicCrystal*, *xicsrt.optics._ShapeSphere.ShapeSphere*

A shperical mosaic crystal optic.

Configuration Options:

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff [float (None)] A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class XicsrtOpticSphericalMosaicCrystal

Inherited Members

class XicsrtOpticSphericalMosaicCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff: float (None) A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Model reflections from a mosaic crystal using a multi-layer model. This is meant to simulate the penetration of x-rays into the HOPG until the rays either encounter a crystalite that satisfies the Bragg condition or get absorbed. This method of calculation replicates both the HOPG 'focusing' qualities as well as the expected throughput.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and `intersect_normal()` methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calulate the distance to the intersection of the rays with the spherical optic.

This calculation is copied from: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mosaic_normals(*normals*, *mask*, *copy=True*)
Add mosaic spread to the normals. Generates a list of crystallite normal vectors in a Gaussian distribution around the nominal surface normals.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

Interact Objects

InteractNone

New Members

class InteractNone(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractObject.InteractObject*

No interaction with surface, rays will pass through unchanged.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

New Private Members

class InteractNone

Inherited Members

class InteractNone

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

InteractMirror

New Members

class InteractMirror(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractObject.InteractObject*

A perfect mirror interaction.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

New Private Members

class InteractMirror

Inherited Members

class InteractMirror

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

InteractCrystal

New Members

class InteractCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractMirror.InteractMirror*

Model for simple Bragg reflections.

Configuration Options:

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

> **Note:** This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

> **Note:** Actually at this point only 'xop' is supported. np 2020-10-13

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

New Private Members

class InteractCrystal

Inherited Members

class InteractCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

crystal_spacing: float [angstroms] The spacing between crystal planes.

Note: This is the nominal 'd' crystal spacing, not the '2d' spacing often used in the literature.

---

reflectivity: float (1.0) A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg: bool (True) Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type: str ('gaussian') The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm: float [rad] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file: str or list A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype: str The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'.

Note: Actually at this point only 'xop' is supported. np 2020-10-13

---

rocking_mix: float A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

InteractMosaicCrystal

New Members

class InteractMosaicCrystal(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._InteractCrystal.InteractCrystal*

A class to handle Mosaic Crystal optics.

Todo: InteractMosaicCrystal efficiency could be improved by including a pre-filter. The pre-filter would use a step-function rocking curve to exclude rays that are outside the likely range of reflection with the current mosaic spread.

Configuration Options:

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff [float (None)] A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

crystal_spacing [float [angstroms]] The spacing between crystal planes. .. Note:

```
This is the nominal 'd' crystal spacing, not the '2d' spacing
often used in the literature.
```

reflectivity [float (1.0)] A reflectivity factor for this optic. The reflectivity will modify the probability that a ray will reflect from this optic.

check_bragg [bool (True)] Switch between x-ray Bragg reflections and optical reflections for this optic. If True, a rocking curve will be used to determine the probability of reflection for rays based on their incident angle. If false this optic will act as a perfect mirror.

rocking_type [str ('gaussian')] The type of shape to use for the crystal rocking curve. Allowed types are 'step', 'gaussian' and 'file'.

rocking_fwhm [float [rad]] The width of the rocking curve, in radians. This option only used when rocking_type is 'step' or 'gaussian'.

rocking_file [str or list] A filename from which to read rocking curve data. A list may be used if sigma and pi data are in separate files.

rocking_filetype [str] The type of rocking curve file to be loaded. The following formats are currently supported: 'xop', 'x0h', 'simple'. .. Note:

```
Actually at this point only 'xop' is supported. np 2020-10-13
```

rocking_mix [float] A mixing factor to combine the sigma and pi reflectivities. This value will be interpreted as sigma/pi and will mix the reflection probabilities linearly. ref = sigma*mix + pi*(1-mix)

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff: float (None) A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

interact(*rays*, *xloc*, *norm*, *mask=None*)

Model reflections from a mosaic crystal using a multi-layer model. This is meant to simulate the penetration of x-rays into the HOPG until the rays either encounter a crystalite that satisfies the Bragg condition

or get absorbed. This method of calculation replicates both the HOPG 'focusing' qualities as well as the expected throughput.

mosaic_normals(*normals*, *mask*, *copy=True*)
Add mosaic spread to the normals. Generates a list of crystallite normal vectors in a Gaussian distribution around the nominal surface normals.

New Private Members

class InteractMosaicCrystal

Inherited Members

class InteractMosaicCrystal

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

angle_calc(*rays*, *norm*, *mask=None*)

angle_check(*rays*, *norm*, *mask=None*)

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

mosaic_spread [float (0.0) [radians]] The fwhm of the Gaussian distribution of crystalite normals around the nominal surface normal.

mosaic_depth [int (15)] The number of crystalite layers to model. This value will depend on the crystal structure and the incident x-ray energy.

mosaic_cutoff: float (None) A numerical probability cutoff used to avoid calculation of the mosaic reflection for angles far away from the nominal Bragg angle. A value of 1e-8 would provide a 6-sigma cutoff in angles, while a value of 1e-14 would provide an 8-sigma cutoff. By default no cutoff is used.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Model reflections from a mosaic crystal using a multi-layer model. This is meant to simulate the penetration of x-rays into the HOPG until the rays either encounter a crystalite that satisfies the Bragg condition or get absorbed. This method of calculation replicates both the HOPG 'focusing' qualities as well as the expected throughput.

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mosaic_normals(*normals*, *mask*, *copy=True*)
Add mosaic spread to the normals. Generates a list of crystallite normal vectors in a Gaussian distribution around the nominal surface normals.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

reflect_vectors(*rays*, *xloc*, *normals*, *mask=None*)

rocking_curve_filter(*incident_angle*, *bragg_angle*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

Shape Objects

ShapePlane

New Members

class ShapePlane(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._ShapeObject.ShapeObject*

A planar shape. This class defines intersections with a plane.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

New Private Members

class ShapePlane

Inherited Members

class ShapePlane

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calculate the distance to an intersection with a plane.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
The planar optic is flat, so the normal direction is always the zaxis.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

ShapeSphere

New Members

class ShapeSphere(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._ShapeObject.ShapeObject*

A shperical shape. This class defines intersections with a sphere.

Configuration Options:

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

initialize()
Initialize the object.

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calulate the distance to the intersection of the rays with the spherical optic.

This calculation is copied from: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

New Private Members

class ShapeSphere

Inherited Members

class ShapeSphere

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

default_config()

radius [float (1.0)] The radius of the sphere.

convex [bool (False)] If True the optic will have a convex curvature, if False the surface will have a concave curvature (the default).

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the `intersect_location()` and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_distance(*rays*)
Calulate the distance to the intersection of the rays with the spherical optic.

This calculation is copied from: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

ShapeMesh

New Members

class ShapeMesh(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.optics._ShapeObject.ShapeObject*

A shape that uses a mesh grid instead of an analytical shape.

Programming Notes

Raytracing of mesh optics is fundamentally slow, because of the need to find which mesh face is intersected by each ray. For the simplest implementations this requires testing each ray against each mesh face leading to the speed scaling as the number of mesh faces (or equivilently mesh_density^2).

Some optimization of the basic calculation been completed. The mesh_intersect_1 method implements the Möller–Trumbore algorithm and is the fastest pure python algorithm found so far. Other variations that have been tried are saved in the archive folder, along with some documentation on performance.

To further improve performance this class (optionally) also makes use of pre-selection with a coarse mesh. First the intersection with the coarse mesh is found for each ray. Then only the 8 nearby faces on the full mesh are checked for the final intersection location. This method improves the performance to (num_faces_coarse + 8).

The current algorithm for pre-selection (mesh refinement) is not perfect in that the nearby faces are not always appropriately chosen leading to a small number of rays being 'lost'. These errors can be minimized by increasing the resolution of the coarse mesh and ensuring that the grid spacing is approximately equal in the x and y directions.

Further performance improvement could be gained by using numba or jax. This would allow the Möller–Trumbore algorithm to be implemented as a loop (instead of being vectorized) where the calculation can be terminated early when no hit is found.

Todo: XicsrtOpticMesh: Improve the pre-selection (mesh refinement algorithm) to eliminate ray losses. The current method is as follows:

1. Calculate intersection with coarse grid.
2. Find the point on the fine grid closest to the intersection.
3. Test all faces on the fine grid that contain this point.

The problem is that the closest point may not always be part of the face that actually has the intersection. This can happen if the fine and coarse grid have very different densities, but also even in the perfect case if the ray hits near the edge of a face and the grid density is not even in the x and y directions.

What is needed is a better selection of nearby faces. There is also a potential to improve computational speed slightly by testing fewer faces on the fine grid.

Configuration Options:

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()
mesh_points mesh_faces mesh_normals

mesh_coarse_points mesh_coarse_faces mesh_coarse_normals

mesh_interpolate mesh_refine

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

initialize()
Initialize the object.

intersect(*rays*)
Calculate ray intersections with the mesh.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

find_near_faces(*X*, *mesh*, *mask*)

New Private Members

class ShapeMesh

_mesh_precalc(*points*, *normals*, *faces*)

Inherited Members

class ShapeMesh

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()
mesh_points mesh_faces mesh_normals

mesh_coarse_points mesh_coarse_faces mesh_coarse_normals

mesh_interpolate mesh_refine

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

ShapeMeshSphere

New Members

class ShapeMeshSphere(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.optics._ShapeMesh.ShapeMesh*

A spherical crystal implemented using a mesh-grid.

This class meant to be used for two reasons: - As an example and template for how to implement a mesh-grid optic. - As a verification of the mesh-grid implementation.

The analytical `ShapeSphere` object should be used for all normal raytracing purposes.

Configuration Options:

radius [float (1.0)] The radius of the sphere.

mesh_size [(float, float) ((11,11))] The number of mesh points in the x and y directions.

mesh_coarse_size [(float, float) ((5,5))] The number of mesh points in the x and y directions.

mesh_points

mesh_faces

mesh_normals

mesh_coarse_points

mesh_coarse_faces

mesh_coarse_normals

mesh_interpolate

mesh_refine

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

radius [float (1.0)] The radius of the sphere.

mesh_size [(float, float) ((11,11))] The number of mesh points in the x and y directions.

mesh_coarse_size [(float, float) ((5,5))] The number of mesh points in the x and y directions.

setup()
Perform any setup actions that are needed prior to initialization.

generate_mesh(*meshsize*)
Create a spherical meshgrid in local coordinates.

New Private Members

class ShapeMeshSphere

Inherited Members

class ShapeMeshSphere

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

_mesh_precalc(*points*, *normals*, *faces*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

radius [float (1.0)] The radius of the sphere.

mesh_size [(float, float) ((11,11))] The number of mesh points in the x and y directions.

mesh_coarse_size [(float, float) ((5,5))] The number of mesh points in the x and y directions.

find_near_faces(*X*, *mesh*, *mask*)

find_point_faces(*p_idx*, *faces*, *mask=None*)
Find all of the the faces that include a given mesh point.

generate_mesh(*meshsize*)
Create a spherical meshgrid in local coordinates.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)
Calculate ray intersections with the mesh.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

mesh_get_index(*hits*, *faces*)
Match faces to face indexes, with a loop over faces.

mesh_initialize()
Pre-calculate a number of mesh properties that are needed in the other mesh methods.

mesh_interpolate(*X*, *mesh*, *mask*)

mesh_intersect_1(*rays*, *mesh*)
Find the intersection of rays with the mesh using the Möller–Trumbore algorithm.

mesh_intersect_2(*rays*, *mesh*, *mask*, *faces_idx*, *faces_mask*)
Check for ray intersection with a list of mesh faces.

Programming Notes

Because of the mesh indexing, the arrays here have different dimensions than in mesh_intersect_1, and need a different vectorization.

At the moment I am using an less efficient mesh intersection method. This should be updated to use the same method as mesh_intersect_1, but with the proper vectorization.

mesh_normals(*hits*, *mesh*, *mask*)

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

Base Objects

InteractObject

New Members

class InteractObject(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._TraceObject.TraceObject*

The base class for interactions of rays with surfaces in XICSRT.

This base class should be used to define behavior such as reflection, transmission and absorption.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

New Private Members

class InteractObject

Inherited Members

class InteractObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask=None*)
Evaluate interaction with a surface. The base-class has no interaction, rays just pass through.

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

ShapeObject

New Members

class ShapeObject(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.optics._TraceObject.TraceObject*

The base class for intersections of rays with surfaces in XICSRT.

This base class should be used to define intersections with various shapes such as planes, spheres and toroids.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the *intersect_location()* and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

New Private Members

class ShapeObject

Inherited Members

class ShapeObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)
Calculate the location and normal of the surface at the ray intersections.

Specific shape objects can reimplement this method, or alternatively reimplement the *intersect_location()* and *intersect_normal()* methods.

Programming Notes

Currently the expectation is that intersect has made copies of ray['origin'] and ray['mask'] before any calculations. This is done for two reasons: 1. provide more information for the interactions. 2. it is much easier to read and understand the code this way. From a memory efficiency standpoint it would be better to modify these arrays in place instead.

intersect_location(*rays*)
Calculate the surface location at the ray intersections.

This base-class just returns a copy of the ray origin.

intersect_normal(*xloc*, *mask*)
Calculate the surface normal at the ray intersection locations.

Normals are not defined for this base-class; an array of np.nan will always be returned.

location_from_distance(*rays*, *dist*, *mask=None*)
Calculate 3D locations given a distance along the rays.

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

TraceObject

New Members

class TraceObject(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.objects._GeometryObject.GeometryObject*

A generic optical element and base class for raytracing objects in XICSRT.

Optical elements have a position and rotation in 3D space and a finite extent. Additional properties, such as as crystal spacing, rocking curve, and reflectivity, should be defined in derived classes.

Configuration Options:

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size [float (None)] The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture [dict or array (None)] Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local [bool (False)] If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates. The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture [bool (true)] Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

initialize()
Initialize the object.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

intersect(*rays*)

interact(*rays*, *xloc*, *norm*, *mask*)

check_bounds(*X*, *mask*)

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

New Private Members

class TraceObject

Inherited Members

class TraceObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_aperture(*X_local*, *mask*)
Check if the ray intersection is within the aperture as set by the 'aperture' config option.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly instead use *check_bounds*.

---

check_bounds(*X*, *mask*)

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

check_size(*X_local*, *mask*)
Check if the ray intersection is within the optic bounds as set by the xsize, ysize and zsize config options.

Note: This method expects to be given the ray intersections in local coordinates. Generally this method should not be called directly, instead use *check_bounds*.

---

default_config()

xsize The size of this element along the xaxis direction. Typically corresponds to the 'width' of the optic.

ysize The size of this element along the yaxis direction. Typically corresponds to the 'height' of the optic.

zsize The size of this element along the zaxis direction. Typically not required, but if needed will correspond to the 'depth' of the optic.

pixel_size: float (None) The pixel size, used for binning rays into images. This is currently a single number signifying square pixels.

aperture: dict or array (None) Define one or more apertures to to apply to this optic. Each aperture is defined as a dictionary with the following keys: shape, size, origin, logic. The origin and logic field keys are optional. The interpretation of size will depend on the provided shape.

trace_local: bool (False) If true: transform rays to optic local coordinates before raytracing, do raytracing in local coordinates, then transform back to global coordinates.

The default is 'false' as most built-in optics can perform raytracing in global coordinates. This option is convenient for optics with complex geometry for which intersection and reflection equations are easier or more clear to program in fixed local coordinates.

check_size: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' and 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

check_aperture: bool (true) Perform a check for whether the rays intersect the optic within the defined bounds (usually defined by 'xsize' an 'ysize'). If set to *False* all rays with a defined reflection/transmission condition will be traced if an intersection can be determined.

filters No documentation yet. Please help improve XICSRT!

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

interact(*rays*, *xloc*, *norm*, *mask*)

intersect(*rays*)

make_image(*rays*)
Collect the rays that intersect with this optic into a pixel array that can be used to generate an intersection image.

Programming Notes

It is important that this calculation is compatible with intersect_check in terms of floating point errors. The simple way to achieve this is to ensure that both use the same calculation method.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

trace(*rays*)
The main method that performs raytracing for this optic.

Raytracing here may be done in global or local coordinates depending on the how the optic is designed and the value of the configuration option: 'trace_local'.

trace_global(*rays*)
This is method that is called by the dispacher to perform ray-tracing for this optic. Rays into and out of this method are always in global coordinates.

It may be convenient for some optics object to do raytracing in local coordinates rather than in global coordinates. This method facilitates this by implementing the 'trace_local' configuration option.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

xicsrt.filters

Contains the built-in filter objects.

Additional filters are available as part of the xicsrt_contrib package.

Built-in Filters Objects

XicsrtBundleFilter

xicsrt.filters._XicsrtBundleFilter.XicsrtBundleFilter

New Members

class XicsrtBundleFilter(*config=None*, *strict=None*, *initialize=None*)

Bases: *xicsrt.objects._GeometryObject.GeometryObject*

A base class for bundle filters.

Configuration Options:

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis (optional) A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction.

If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]).

The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

filter(*bundle_input*)

This is the main filtering method that must be reimplemented for specific filter objects.

New Private Members

class XicsrtBundleFilter

Inherited Members

class XicsrtBundleFilter

__init__(*config=None*, *strict=None*, *initialize=None*)

Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)

Set the Z-Axis to aim at a particular point.

check_config()

Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()

Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis (optional) A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction.

If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]).

The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

filter(*bundle_input*)
This is the main filtering method that must be reimplemented for specific filter objects.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

XicsrtBundleFilterSightline

xicsrt.filters._XicsrtBundleFilterSightline.XicsrtBundleFilterSightline

New Members

class XicsrtBundleFilterSightline(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.filters._XicsrtBundleFilter.XicsrtBundleFilter*

A bundle filter based on proximity to sightline vectors.

Configuration Options:

radius The radius of the cylindrical sightline.

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

radius The radius of the cylindrical sightline.

filter(*bundle_input*)
Filter ray bundles that do not originate inside the cylindrical sightline.

New Private Members

class XicsrtBundleFilterSightline

Inherited Members

class XicsrtBundleFilterSightline

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

radius The radius of the cylindrical sightline.

filter(*bundle_input*)
Filter ray bundles that do not originate inside the cylindrical sightline.

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

xicsrt.objects

Contains various base objects used in XICSRT.

Objects

ConfigObject

xicsrt.objects._ConfigObject.ConfigObject

New Members

class ConfigObject(*config=None*, *strict=None*, *initialize=None*)
Bases: `object`

A base class for any objects with a configuration.

Configuration Options:

class_name Automatically generated.

yo_mama Is a wonderful person!

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

default_config()

class_name Automatically generated.

yo_mama Is a wonderful person!

get_config()

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

setup()
Perform any setup actions that are needed prior to initialization.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

initialize()
Initialize the object.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

New Private Members

class ConfigObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

Inherited Members

class ConfigObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

class_name Automatically generated.

yo_mama Is a wonderful person!

get_config()

initialize()
Initialize the object.

setup()
Perform any setup actions that are needed prior to initialization.

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

GeometryObject

xicsrt.objects._GeometryObject.GeometryObject

New Members

class GeometryObject(*config=None*, *strict=None*, *initialize=None*)
Bases: *xicsrt.objects._ConfigObject.ConfigObject*

The base class for any geometrical objects used in XICSRT.

Configuration Options:

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis [(optional)] A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction. If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]). The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

class_name Automatically generated.

yo_mama Is a wonderful person!

default_config()

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis (optional) A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction.

If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]).

The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

setup()
Perform any setup actions that are needed prior to initialization.

set_orientation(*zaxis*, *xaxis=None*)

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

point_to_external(*point_local*)

point_to_local(*point_external*)

vector_to_external(*vector*)

vector_to_local(*vector*)

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

New Private Members

class GeometryObject

Inherited Members

class GeometryObject

__init__(*config=None*, *strict=None*, *initialize=None*)
Initialize self. See help(type(self)) for accurate signature.

aim_to_point(*aim_point*, *xaxis=None*)
Set the Z-Axis to aim at a particular point.

check_config()
Check the config before copying to the internal param. This is called during object instantiation (*__init__*) and therefore before *setup* is called.

check_param()
Check the internal parameters prior to initialization. This will be called after *setup* and before *initialize*.

default_config()

origin The x,y,x origin of this element in global coordinates.

zaxis A unit-vector defining the z-axis of the element in global coordinates. For most optics: z-axis defines the surface normal direction.

xaxis (optional) A unit-vector defining the x-axis of the element in global coordinates. For most optics: x-axis defines the 'width' direction.

If xaxis is not provided it will be automatically generated by: cross(zaxis, [0,1,0]).

The *yaxis* is always automatically generated and defined by: cross(zaxis, xaxis)

get_config()

get_default_xaxis(*zaxis*)
Get the X-axis using a default definition.

In order to fully define the orientation of a component both, a z-axis and an x-axis are expected. For certain types of components the x-axis definition is unimportant and can be defined using a default definition.

initialize()
Initialize the object.

point_to_external(*point_local*)

point_to_local(*point_external*)

ray_to_external(*ray_local*, *copy=False*)

ray_to_local(*ray_external*, *copy=False*)

set_orientation(*zaxis*, *xaxis=None*)

setup()
Perform any setup actions that are needed prior to initialization.

to_ndarray(*vector_in*)

to_vector_array(*vector_in*)
Convert a vector to a numpy vector array (if needed).

update_config(*config_new*, ***kwargs*)
Overwrite any config values in this object with the ones given. This will be done recursively for all nested dictionaries.

vector_to_external(*vector*)

vector_to_local(*vector*)

RayArray

xicsrt.objects._RayArray.RayArray

New Members

class RayArray(**args*, ***kwargs*)
Bases: `dict`

The base class for an Ray array.

The RayArray object is essentially a dictionary of numpy arrays. Some convenience methods have been added.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

initialize()
Initialize the ray array. This will ensure that all properties are present and of the correct type.

zeros(*num*)

copy() → a shallow copy of D

extend(*ray_in*)

New Private Members

class RayArray

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Inherited Members

class RayArray

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

extend(*ray_in*)

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

initialize()
Initialize the ray array. This will ensure that all properties are present and of the correct type.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem()
Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

zeros(*num*)

Dispatcher

xicsrt.objects._Dispatcher.Dispatcher

New Members

class Dispatcher(*config=None*, *section=None*)
Bases: `object`

A class to help find, initialize and then dispatch calls to raytracing objects.

A dispatcher is used within XICSRT to find and instantiate objects based on their specification within the config dictionary. These objects are then tracked within the dispatcher, allowing methods to be called on all objects sequentially.

__init__(*config=None*, *section=None*)
Initialize self. See help(type(self)) for accurate signature.

instantiate(*names=None*)

find_xicsrt_objects(*pathlist*)
Return a dictionary with all the XICSRT objects found in the given list of paths. Objects are identified by looking for python files that start with '_Xicsrt' prefix.

Programming Notes

If a given path does not exist glob will just return and empty list. For this reason no path existence checking is needed (unless we want to raise a user friendly error).

get_object(*name*)

check_config(**args*, ***kwargs*)

check_param(**args*, ***kwargs*)

get_config(**args*, ***kwargs*)

setup(**args*, ***kwargs*)

initialize(**args*, ***kwargs*)

generate_rays(*keep_meta=None*, *keep_history=None*)
Generates rays from all sources.

trace(*rays*, *keep_meta=None*, *keep_history=None*, *keep_images=None*)
Perform raytracing for each object in sequence.

apply_filters(*filters*)

New Private Members

class Dispatcher

__init__(*config=None*, *section=None*)
Initialize self. See help(type(self)) for accurate signature.

_instantiate_single(*obj_info*, *config*, *strict=None*)
Instantiate an object from a list of filenames and a class name.

Inherited Members

class Dispatcher

__init__(*config=None*, *section=None*)
Initialize self. See help(type(self)) for accurate signature.

_instantiate_single(*obj_info*, *config*, *strict=None*)
Instantiate an object from a list of filenames and a class name.

apply_filters(*filters*)

check_config(**args*, ***kwargs*)

check_param(**args*, ***kwargs*)

find_xicsrt_objects(*pathlist*)
Return a dictionary with all the XICSRT objects found in the given list of paths. Objects are identified by looking for python files that start with '_Xicsrt' prefix.

Programming Notes

If a given path does not exist glob will just return and empty list. For this reason no path existence checking is needed (unless we want to raise a user friendly error).

generate_rays(*keep_meta=None*, *keep_history=None*)
Generates rays from all sources.

get_config(**args*, ***kwargs*)

get_object(*name*)

initialize(**args*, ***kwargs*)

instantiate(*names=None*)

setup(**args*, ***kwargs*)

trace(*rays*, *keep_meta=None*, *keep_history=None*, *keep_images=None*)
Perform raytracing for each object in sequence.

xicsrt.visual

Contains a set of visualization modules for both 2D and 3D plotting.

Visualization Modules

xicsrt_2d__matplotlib

xicsrt.visual.xicsrt_2d__matplotlib

A set of tools for 2d visualization of the XICSRT results

plot_example(*results*, *name*)
A simplified plotting routine to serve as an example of how to develop xicsrt visualizations. This function will plot found ray intersections.

plot_intersect(**args*, ***kwargs*)
Plot the intersection of rays with the given optic.

Parameters

- **results** – The restults dictionary from *raytrace()* that include the ray history.
- **Keywords** –
- **--------** –
- **name** (*string (None)*) – The name of the optic or source for which to plot intersections. The name refers to the key of the entry in the config dictionary. For example the name 'detector' will refer to config['optics']['detector'].
- **section** (*string (None)*) – [Optional] The name of the config section in which to search for *name*. This should typically be either 'optics' or 'sources'. If no section is given then then 'optics' will be searched first, then 'sources'.
- **options** (*dict (None)*) – [Optional] A dictionary containing plot options. All options can also be passed individually as keywords.

Returns Will return a plotlist with the full plot definition.

Return type plotlist

plot_image(*results*, *name=None*, *section=None*, *options=None*, ***kwargs*)
Plot an intersection image along with column and row summation plots.

Private Members

_get_aperture_plotlist(*obj*, *scale=None*)

_get_bounds_plotlist(*obj*, *scale=None*)

_get_hist(*obj*, *results*, *opt*, *raytype='found'*, *axis=0*)

_get_intersect_plotlist(*results*, *name=None*, *section=None*, *options=None*, *_noaspect=False*, ***kwargs*)
Return a plotlist for `plot_intersect()`.

_get_rays_plotlist(*obj*, *results*, *opt*, *raytype='found'*)

_on_ylims_change(*event_ax*)
An Axes callback to update the data limits after a change in the the data limits. This is primarily meant to allow retention af a fixed aspect after the user has zoomed into a region.

_truncate_mask(*mask*, *max_num*)

_update_lim_aspect(*ax*)
Update the data limits (xlim & ylim) to produce an equal aspect given a fixed plot size.

xicsrt_3d__plotly

xicsrt.visual.xicsrt_3d__plotly

These are a set of routines for 3D visualization using the plotly library.

Example

Example code for using this module within a Jupyter notebook.

plot(*results*, ***kwargs*)
Create a 3d plot using default options.

Any keywords provided will be passed to *add_rays*. For more control over plotting options it is recommended to perform the plotting steps manually as shown by the example in the *xicsrt_3d* module docstring.

figure(*showbackground=False*, *visible=False*)

show(*figure=None*)

add_rays(*results*, ***kwargs*)

add_optics(*config*, *figure=None*, ***kwargs*)

add_sources(*config*, *figure=None*, ***kwargs*)

add_fluxsurfaces(*config*, *figure=None*, ***kwargs*)

add_object(*config*, *name*, *section*, *figure=None*, ***kwargs*)

Private Members

_add_fluxsurf_single(*config*, *name*, *section=None*, *figure=None*, *alpha=None*, *flatshading=None*, ***kwargs*)
Plot the 3D flux surfaces of a plasma source. This should work for any object that has a 'car_from_flx' method.

_add_trace_mesh(*obj*, *figure=None*, *name=None*)
Add a meshgrid to the 3D plot.

_add_trace_volume(*obj*, *figure*, *name=None*, *opacity=0.5*)

_gen_fluxsurface_mesh(*obj*, *s*, *range_m=None*, *range_n=None*)
Generate points on a flux surface. The given input object must have a method 'car_from_flx'.

_make_plotly_color(*color*, *alpha=None*)

_plot_ray_history(*history*, *lost=None*, *figure=None*, *color=None*, *lost_color=None*, *found_color=None*)

_thin_mask(*mask*, *max_num*)
Reduce the number of True elements in a mask array.

Ray thinning is done randomly. Used to reduce the number of rays plotted.

xicsrt_3d__ipyvolume

xicsrt.visual.xicsrt_3d__ipyvolume

Private Members

xicsrt.tools

A set of mathematical tools for XICSRT.

Raytracing Tools

xicsrt_aperture

xicsrt.tools.xicsrt_aperture

A set of apertures for ray filtering.

aperture_mask(*X_local*, *m*, *aperture_info*)
Generate a mask array for the given aperture (or array of apertures).

aperture_selector(*X_local*, *m*, *aperture*, *_internal=False*)
Will call the appropriate aperture function for the given aperture name.

Note: This selector and all the individual function will modify the mask array in place.

aperture_none(*X_local*, *m*, *aperture*)
An empty aperture object.

aperture_circle (*X_local*, *m*, *aperture*)
A circular Aperture.

name: 'circle'

size: [radius] Contains the radius of the aperture.

aperture_square (*X_local*, *m*, *aperture*)
A square Aperture.

name: 'square'

size: [x, y] Contains the x and y size (full width) of the aperture.

aperture_rectangle (*X_local*, *m*, *aperture*)
A rectangular Aperture.

name: 'rectangle'

size: [x, y] Contains the x and y size (full width) of the aperture.

aperture_ellipse (*X_local*, *m*, *aperture*)
An elliptical Aperture.

name: 'ellipse'

size: [x, y] Contains the x and y size (full width) of the aperture.

aperture_triangle (*X_local*, *m*, *aperture*)
An triangular aperture defined by three vertices.

name: 'ellipse'

vertices: [[x0, y0], [x1,y1], [x2,y2]] Contains the three vertices of the aperture.

Private Members

_aperture_defaults (*aperture*)

xicsrt_spread

xicsrt.tools.xicsrt_spread

A set of algorithms to generate vector distributions.

Note: The term 'spread' here denotes the angular range of emission. The term 'divergence' is not used because I normally think of a divergence as a gaussian distributed probability distribution of angles. This type of distribution is available, but for generality and consistency 'spread' will be used throughout.

---

vector_distribution (*spread*, *number*, *name=None*)
A convenience function to retrieve vector distributions by name.

Parameters

- **spread** (*float or array [radians]*) – Can be a scalar or an array. See individual distributions for format and definitions.
- **number** (*int*) – The number of vectors to generate.

- **name** (*string ('isotropic')*) – The name of the vector distribution. Available names: 'isotropic', 'isotropic_xy', 'flat', 'flat_xy', 'gaussian'.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

solid_angle(*spread*, *name=None*)
A convenience function to retrieve solid angles that correspond to the various vector distributions.

Units: [sr]

vector_dist_isotropic(*spread*, *number*)
Return unit vectors from an isotropic (uniform spherical) distribution that fall within an angular spread (divergence) of theta.

The ray cone is aligned along the z-axis.

Parameters

- **spread** (*float [radians]*) – The half-angle of the emitted cone of vectors (axis to edge).
- **number** (*int*) – The number of vectors to generate.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

solid_angle_isotropic(*spread*)
Calculate the solid angle for the vector_dist_isotropic distribution.

Parameters **spread** (*float [radians]*) – The half-angle of cone of vectors (axis to edge).

Returns Units: [sr]

Return type solid_angle

vector_dist_isotropic_xy(*spread*, *number*)
Return random unit vectors from an isotroptic (uniform spherical) distribution that fall within a given x and y angular spread.

The truncated-cone of vectors is aligned along the z-axis.

Note: This routine repeatedly filters from a circular distribution, which is accurate but not efficent. Efficiency goes down for more unequal values of the x and y spread.

Todo: Replace vector_dist_isotropic_xy with a more efficent calculation. A possible approach is to calculate the 2D Joint Cumulative Distribution Function for isotropic emission on a flat plane.

Parameters

- **spread** (*float or array [radians]*) –

  The half-angles in the x and y directions that define the extent of the truncated-cone of vectors. Spread can be contain either 1,2 or 4 values.

  s or [s]
  : A single value that will be used for both the x and y directions.

  [x, y]

Two values values that will be used for the x and y directions.

[xmin, xmax, ymin, ymax]

For values that define the asymmetric exent in x and y directions.
Example: [-0.1, 0.1, -0.5, 0.5]

- **number** (*int*) – The number of vectors to generate.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

solid_angle_isotropic_xy(*spread*)

Calculate the solid angle for the vector_dist_isotropic_xy distribution.

Units: [sr]

vector_dist_flat(*spread*, *number*)

Return unit vectors from an flat (uniform planar) distribution that fall within an angular spread.

The ray cone is aligned along the z-axis.

Parameters

- **spread** (*float [radians]*) – The half-angle of the emitted cone of vectors (axis to edge).
- **number** (*int*) – The number of vectors to generate.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

vector_dist_flat_xy(*spread*, *number*)

Return random unit vectors from an flat (uniform planar) distribution that fall within a given x and y angular spread.

The truncated-cone of vectors is aligned along the z-axis.

Note: This distribution is identical to that used by the SHADOW raytracing code for both the 'flat' and 'uniform' distributions (as of 2021-01).

Parameters

- **spread**(*float or array [radians]*) –

  The half-angles in the x and y directions that define the extent of the truncated-cone of vectors. Spread can be contain either 1,2 or 4 values.

  s or [s]

  A single value that will be used for both the x and y directions.

  [x, y]

  Two values values that will be used for the x and y directions.

  [xmin, xmax, ymin, ymax]

  For values that define the asymmetric exent in x and y directions.
  Example: [-0.1, 0.1, -0.5, 0.5]

- **number** (*int*) – The number of vectors to generate.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

vector_dist_flat_gaussian (*spread*, *num_samples*)
Create distribution of vectors with a Gaussian distribution on a flat plane. The ray code is aligned with the z-axis, so the distribution is Gaussian in the x-y directions.

For small angles this distribution will approximate the Kent distribution and will be approximately gaussian in angle.

Parameters **spread** (*float [radians]*) – The half-with-at-half-max (hwhm) of the Gaussian angular distribution.

Returns A numpy array of shape (number, 3) containing the generated unit vectors.

Return type ndarray

Private Members

_parse_spread_single (*spread*)
Parse the number of input values in spread and return a standard array. Use only for distribution with a single spread value.

_parse_spread_xy (*spread*)
Parse the number of input values in spread and return a standard array. Use only for assymetric xy distributions.

_to_ndarray (*spread*)
Convert input value to an numpy array. Scalars will be transformed to a one element array.

xicsrt_bragg

xicsrt.tools.xicsrt_bragg

A set of utility routines to load bragg reflection data files from several external applications including x0h, XOP and SHADOW.

Data will be returned in a standardized dictionary.

read (*filename*, *filetype=None*)
The main routine to read Bragg reflection data files.

This will switch between the various format specific routines based on the filetype parameter.

Parameters **filename** (*str*) – The file to read.

Keyword Arguments **filetype** (*str*) – The source of the given file. If not provided, the filetype will be guessed. Currently supported types are: 'xop', 'x0h'.

read_xop (*filename*)
Read a data file from XOP and return a standard rocking-curve dict.

It is expected that the given file is diff_pat.dat file from XOP.

Private Members

_guess_filetype (*filename*)

_read_xop__diff_pat_dat__data (*filename*)

_read_xop__diff_pat_dat__header (*filename*)

Mathematical Tools

xicsrt_voigt

xicsrt.tools.xicsrt_voigt

A set of routines for related to Voigt distributions.

voigt(*x*, *intensity=None*, *location=None*, *sigma=None*, *gamma=None*)
The Voigt function is also the real part of w(z) = exp(-z^2) erfc(iz), the complex probability function, which is also known as the Faddeeva function. Scipy has implemented this function under the name wofz()

voigt_cdf_tab(*gamma*, *sigma*, *gridsize=None*, *cutoff=None*)

voigt_cdf_interp(*gamma*, *sigma*, *gridsize=None*)

voigt_invcdf_interp(*gamma*, *sigma*, *gridsize=None*)

voigt_cdf_numeric(*x*, *gamma*, *sigma*, *gridsize=None*)

voigt_invcdf_numeric(*x*, *gamma*, *sigma*, *gridsize=None*)

voigt_random(*gamma*, *sigma*, *size*, ***kwargs*)
Draw random samples from a Voigt distribution.

The tails of the distribution will be clipped; the clipping level can be adjusted with the cutoff keyword. The default values is 1e-5.

Private Members

xicsrt_math

xicsrt.tools.xicsrt_math

A set of mathematical utilities and vector convenience functions for XICSRT.

distance_point_to_line(*origin*, *normal*, *point*)
Find the closest distnace between a point and a line in 3D.

intersect_ray_plane(*ray*, *plane*)
Find the intersection between a ray and a plane in 3D.

toarray_1d(*a*)
Convert the input to a ndarray with at least 1 dimension. This is similar to the numpy function atleast_1d, but has less overhead and is jax compatible.

vector_angle(*a*, *b*)
Find the angle between two vectors.

vector_rotate(*a*, *b*, *theta*)
Rotate vector a around vector b by an angle theta (radians)

Programming Notes

u: parallel projection of a on b_hat. v: perpendicular projection of a on b_hat. w: a vector perpendicular to both a and b.

magnitude(*vector*)
Calculate magnitude of a vector or array of vectors.

normalize(*vector*)
Normalize a vector or an array of vectors. If an array of vectors is given it should have the shape (N,M) where | N: Number of vectors | M: Vector length

sinusoidal_spiral(*phi*, *b*, *r0*, *theta0*)

rotation_matrix(*axis*, *theta*)
Return the rotation matrix associated with counterclockwise rotation about the given axis by theta radians.

bragg_angle(*wavelength*, *crystal_spacing*)
The Bragg angle calculation is used so often that it deserves its own function.

Note: The crystal_spacing here is the true spacing, not the 2d spacing that is sometimes used in the literature.

---

cyl_from_car(*point_car*)
Convert from cartesian to cylindrical coordinates.

car_from_cyl(*point_cyl*)
Convert from cylindrical to cartesian coordinates.

tor_from_car(*point_car*, *major_radius*)
Convert from cartesian to toroidal coordinates.

Parameters

- **point_car** (*array [meters]*) – Cartesian coordinates [x,y,z]
- **major_radius** (*float [meters]*) – Torus Major Radius

Returns **point_tor** – Toroidal coordinates [r_min, theta_poloidal, theta_toroidal]

Return type array [meters]

car_from_tor(*point_tor*, *major_radius*)
Convert from toroidal to cartesian coordinates.

Parameters

- **point_tor** (*array [meters]*) – Toroidal coordinates [r_min, theta_poloidal, theta_toroidal]
- **major_radius** (*float [meters]*) – Torus Major Radius

Returns **point_car** – Cartesian coordinates [x,y,z]

Return type array [meters]

point_in_triangle_2d(*pt*, *p0*, *p1*, *p2*)
Determine if a point (or set of points) fall within a triangle as specified by three vertices. Calculation is performed in two dimensions (2D).

Private Members

xicsrt_math_jax

xicsrt.tools.xicsrt_math_jax

A set of mathematical function with jax acceleration. Many of these functions are exact copies or slight modification of the functions in xicsrt_math. Other function are specific to this module.

Programming Notes

These module was developed to support some specific modeling work by N. Pablant and is not used in any of the built-in xicsrt code. There is no plan to support jax generally within xicsrt, so I am not really sure of the best way to handle this module for the moment. Maybe move it into xicsrt_contrib?

toarray_1d(*a*)
Convert the input to a ndarray with at least 1 dimension. This is similar to the numpy function atleast_1d, but has less overhead and is jax compatible.

vector_angle(*a*, *b*)
Find the angle between two vectors.

vector_rotate(*a*, *b*, *theta*)
Rotate vector a around vector b by an angle theta (radians)

Programming Notes

u: parallel projection of a on b_hat. v: perpendicular projection of a on b_hat. w: a vector perpendicular to both a and b.

sinusoidal_spiral(*phi*, *b*, *r0*, *theta0*)

point_to_external(*point_local*, *orientation*, *origin*)

point_to_local(*point_external*, *orientation*, *origin*)

vector_to_external(*vector*, *orientation*)

vector_to_local(*vector*, *orientation*)

Private Members

Programmatic Tools

xicsrt_doc

xicsrt.tools.xicsrt_doc

A set of tools to help with automatic API documentation of XICSRT.

Description

XICSRT uses sphinx for documentation, and API docs are based on the idea of code self documentation though *python* doc strings. This module contains a set of decorators and helper function to aid in self documentation.

The most important part of this module is the *@dochelper* decorator which should be used for all element classes.

Todo:

- The config docstrings should all be indented follow the help() standard.
- Would it be helpful to show which inherited class the options came from?

---

dochelper(*cls*)
A functional wrapper for the DocHelper class. Intended to be used as a decorator.

This decorator does the following:

1. Adds a 'Configuration Options' section to the class docstring that contains all options defined in default_config and any class ancestors.

class DocHelper(*cls*)
A class to help generate docstrings for XICSRT.

This is expected to be used through the *@dochelper* class decorator.

update_class_docstring(*cls*)
Update the class docstring. This method does the following: 1. Creates a new section 'Configuration Options' which contains

combined documentation for all config options defined in any ancestor.

Private Members

xicsrt_string

xicsrt.tools.xicsrt_string

A set of tools to facilitate string handling in XICSRT.

simplify_strings(*value*)
Recursively simplify strings in the given variable.

This will do the following:

1. Make all strings lower case.

Private Members

xicsrt.util

Contains a set of "external" libraries and tools that are included in the XICSRT source code. These are included here either because they are not currently available on pipy or because they have been highly modified.

Programming Notes

Make sure not to add anything here that is incompatible with the MIT license!

Libraries and Utilities

profiler

xicsrt.util.profiler

Authors:

Novimir Antoniuk Pablant <npablant@pppl.gov>

Purpose: Create a simple profiler module.

Description: This module is meant to enable manual profiling with very low overhead.

isEnabled()

startProfiler(*reset=False*)

stopProfiler()

resetProfiler()

report(*flush=True*)

getTimeTotal(*name*)

getTimeSingle(*name*)

start(*name*)

stop(*name*)

Private Members

_newProfile(*name*)

mircolor

xicsrt.util.mircolor

A module for dealing with colors.

This module is an extension/modification of parts of: matplotlib.colors matplotlib.cm matplotlib.pyplot

There are a number of things that I don't like about how the matplotlib versions handle things, so this is my attempt to correct some of these issues.

Maybe someday I'll try to tackle the matplotlib code directly and create a replacement for the original code.

Example

norm = mircolor.Normalize(0.0, 1.0) grad = mircolor.getColorGradient(norm, 'tab10') color = grad.to_rgba(1.0)

getColorGradient(*norm=None*, *cmap=None*)

class ColorGradient

class LinearSegmentedColorGradient(*norm=None*, *segmentdata=None*)

> **rgba_keys = ['red', 'blue', 'green', 'alpha']**
>
> **to_rgba**(*value*, *alpha=None*)
>
> **setSegmentData**(*segmentdata_in*)

Private Members

mirplot

xicsrt.util.mirplot

An interface to matplotlib that allows specification of complex plots though a list of parameter dictionaries.

Example

The simplest example:

```
import numpy as np
import mirplot

x = np.arange(10)
y = x
plotlist = [{'x':x, 'y':y}]
fig = mirplot.plot_to_screen(plotlist)
```

Any supported plot properties can be added to the plot dictionary:

To add multiple plots to a single figure add parameter dicts ta the plotlist:

If axes names are provided then plots will be added to separate subfigures (stacked vertically). Each unique axes name will result in a new subfigure.

mirplot can also be used with predifined axes. For this purpose the axes must be placed into a dictionary and passed to *plot_to_axes*.

mirplot properties

A set of unique plot and axes properties are defined by mirplot to enable a complete dictionary definition.

type [str ('line')] Allowed Values: line, errorbar, scatter, fill_between, hline, vline, hspan, vspan.

legend [bool (false)] Set to true to show the legend in this subplot.

matplotlib properties

Any matplotlib plot or axes property that can be set using a simple *set_prop(value)* method is supported. Certain properties requiring a more complex set call are also supported.

plot_to_screen(*plotlist*, *show=True*)

plot_to_file(*plotlist*, *filename*)

plot_to_axes(*plotlist*, *axesdict*)

Private Members

_apply_axes_prop(*prop*, *axes*)

_apply_fig_prop(*prop*, *ax*)

_apply_plot_prop(*prop*, *axes*)

_autoname_plots(*plotlist*, *sequential=False*)
Automatically name any plots that were not given a name by the user.

_clean_plot_prop(*prop*)
Check the plot properties and cleanup or provides errors.

_get_figure_size(*numaxes*)
Return the default figure size. Width: 8 units Height: 3 units for every subplot or max 9 units :returns: The figure size in inches. :rtype: (width, height)

_make_axes(*namelist*, *fig*)

_make_figure(*namelist*)

_set_plot_defaults(*prop*)

mirlogging

xicsrt.util.mirlogging

A logging module for XICSRT.

For now, the purpose of this module is simply is set default logging options for interactive use.

defaultConfig(*level=None*, *long=False*, *force=False*)

Private Members

6.4 Authors

6.4.1 Primary Authors

- **Novimir Pablant** <npablant@pppl.gov>
- James Kring <jdk0026@tigermail.auburn.edu>
- Yevgeniy Yakusevich <eugenethree@gmail.com>
- Nathan Bartlett <nbb0011@auburn.edu>

6.4.2 Contributors

- Tanner Cordova <cordova12@llnl.gov>
- Collin S. Dunn <cdunn314@gatech.edu>
- Jio Gallardy <g2gallardy@gmail.com>
- Mike MacDonald <macdonald10@llnl.gov>
- Sapna Mishra <sapna.mishra@iter-india.org>
- Matt Slominski <mattisaacslominski@yahoo.com>

6.5 License

This software is licensed under the MIT Licence

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use,

copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 7

Indices and tables:

- genindex
- modindex
- search

Python Module Index

e

t

x

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

X

Z