XICSRT: x-ray raytracing in Python
A Tutorial

Novimir A. Pablant
npablant@pppl.gov

N.A. Pablant[1], N. Bartlett[2], J. Kring[2], Y. Yakusevich[3]
T. Cordova[3], C.S. Dunn[4], J. Gallardy[5], M. MacDonald[6], S. Mishra[7]

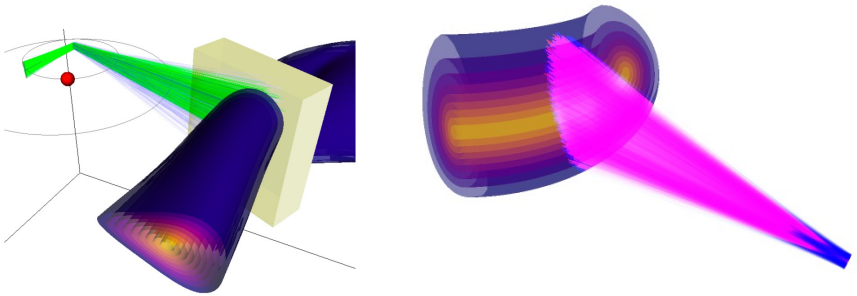
[1] Princeton Plasma Physics Laboratory
[2] Auburn University
[3] University of California San Diego
[4] Georgia Institute of Technology
[5] University of California Los Angeles
[6] Lawrence Livermore National Laboratory
[7] ITER-India, Institute for Plasma Research

last Update: 2021-08-09
Tutorial Version: 1.2
XICSRT Version: 0.8.0

XICSRT is a general-purpose scientific raytracing code intended for both optical and x-ray raytracing

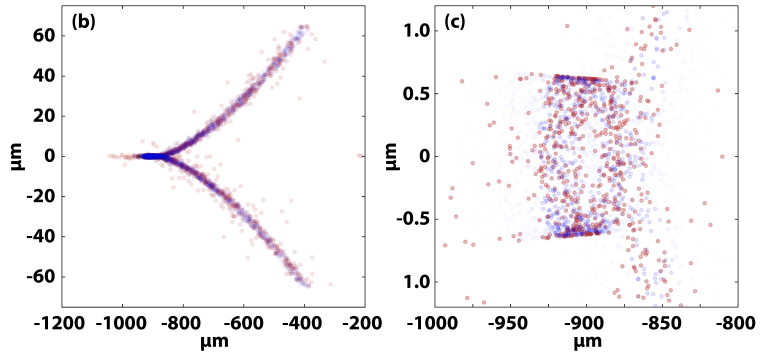
XICSRT can model complex x-ray diagnostics

- X-Ray Bragg reflections of arbitrarily complicated optic shapes
- Complex 3D ray sources for example tokamak, stellarator or HEDP plasmas
- Careful attention to preserving photon statistics throughout, allowing raytracing of plasma sources in real units



Written in Python and designed for simplicity, modularity and extensibility

- Released as an open-source tool for general scientific use.
- Validated against the mature synchrotron x-ray raytracing code SHADOW



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

XICSRT Basics

XICSRT can be installed as a normal python package

xicsrt is written in pure python and is cross-platform compatible.

- Tested on osx, linux and windows

Installation using pip is recommended

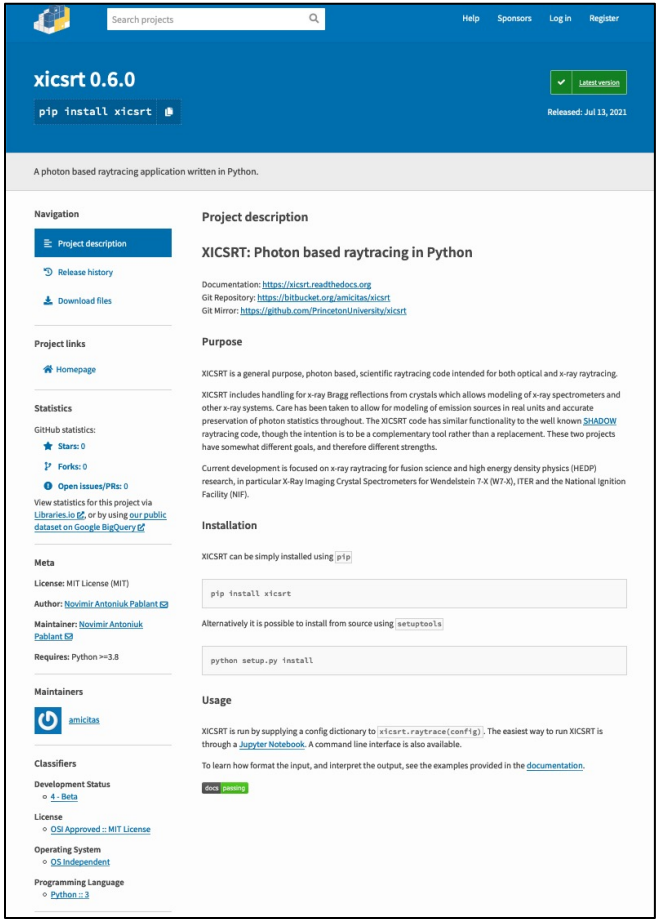
```
pip install xicsrt
```

Requirements:

- `python >= 3.8`
- `numpy, scipy, pillow`

Recommended:

- `jupyter`
- `matplotlib, ploty, h5py`



Documentation is available on the web

Documentation for **xicsrt** can be found at:

https://xicsrt.readthedocs.org

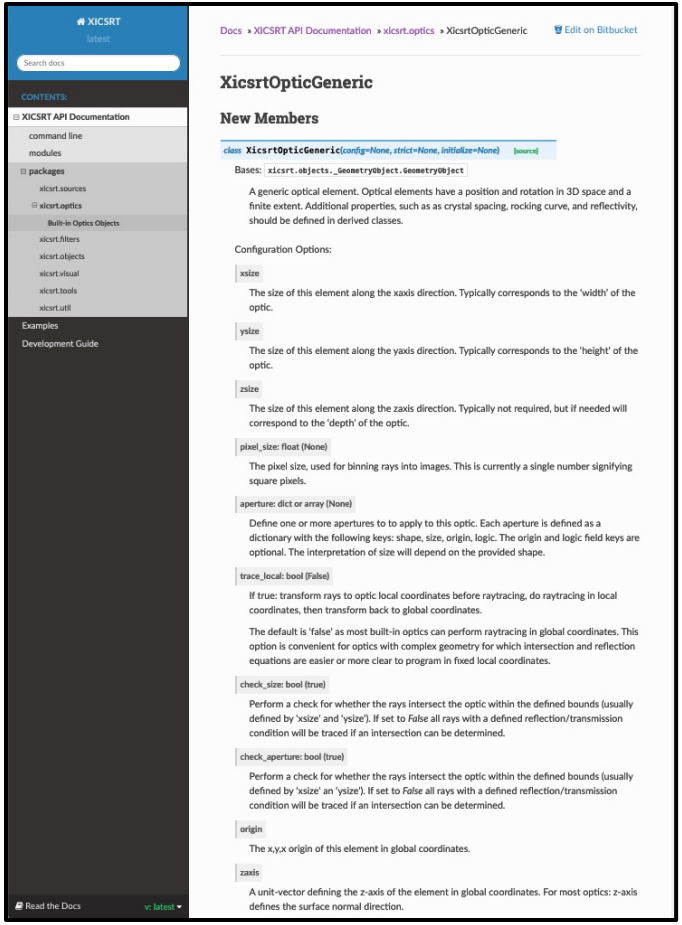
Contains:

- Basic instructions, Examples, API Documentation

All API Documentation can also be accessed using pythons built-in `help()` function.

- For example: `help(xicsrt.raytrace)`

xicsrt documentation is still rather incomplete.
Please help us improve it!



XICSRT can be run from the command line or from within an interactive Python session

The easiest way to run **xicsrt** is using a Jupyter Notebook

xicsrt is run by supplying a `config` dictionary to `raytrace()`:

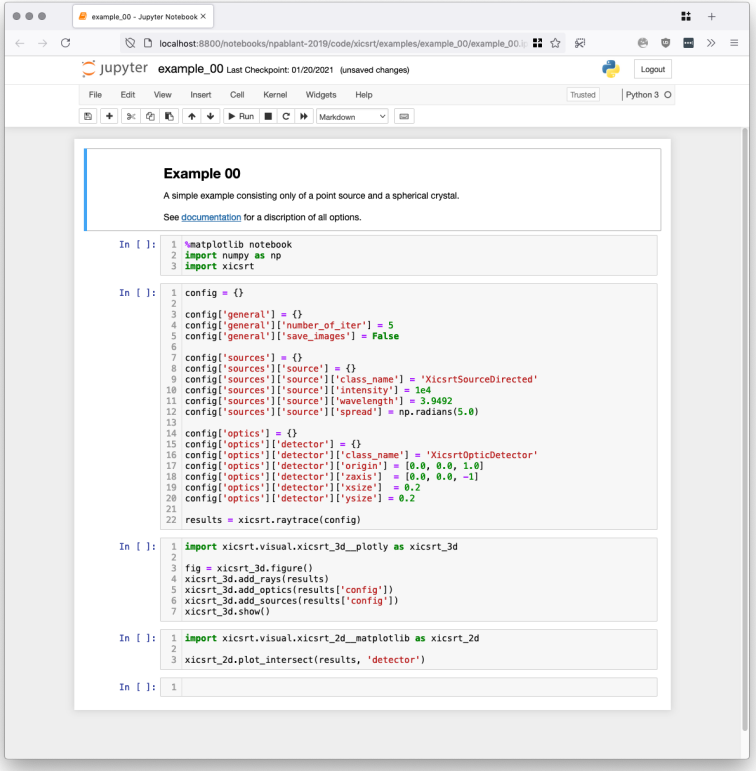
```
results = xicsrt.raytrace(config)
```

The `config` dictionary contains ***all information*** needed to perform the desired raytrace

The `results` are a dictionary that include:

- full raytracing history
- pixelated images at each component
- metadata

The pixelated images of the ray intersections at each element can be automatically saved in (almost) any format.



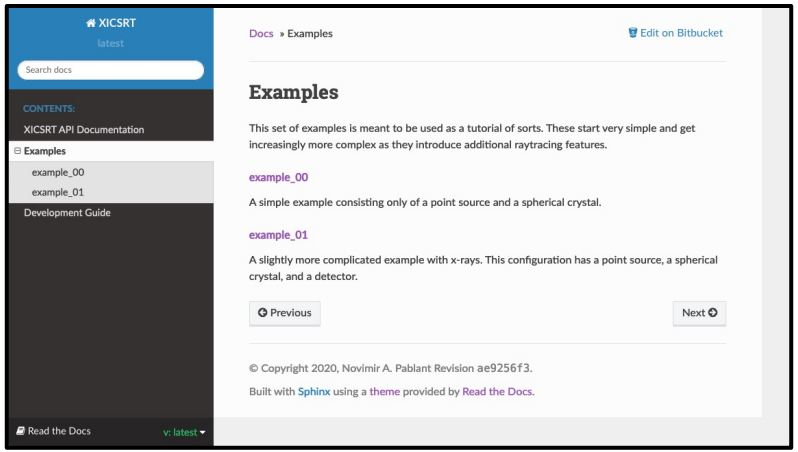
The core **xicsrt** project does *not* provide a graphical user interface. Adding a gui layer (as a separate project) could be easily achieved by manipulating the config dictionary.

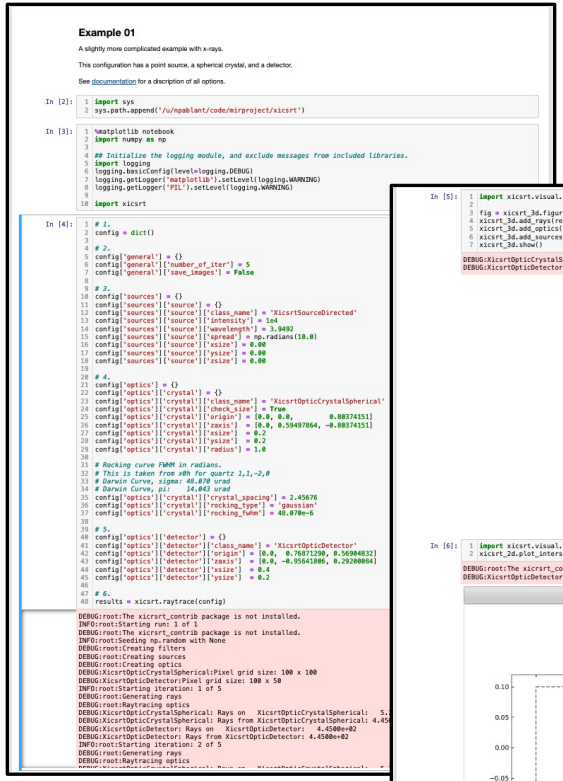
The best way to get started as an XICSRT user is to run the examples

Several examples are provided with the documentation

https://xicsrt.readthedocs.io/en/latest/apidoc/examples.html

Examples include a working Jupyter notebook that can be used to run **xicsrt** and perform simple visualizations.





The config dictionary must include at least three sections: general, sources and optics

Raytracing requires at least one ray source and one optic

- All optics also function as detectors!

Required config sections:

config['general']

- Options that define the overall raytracing behavior.

config['sources']

- Contains a dictionary that defines the ray source.

config['optics']

- Contains a dictionary that defines all optics in sequence.

Optional config sections:

config['filters']

- Defines filters that can be attached to sources or optics

config['scenario']

- Not used by **xicsrt**; meant for external scripting options

Example of a minimal config dictionary:

```
config = {}

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

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(5.0)

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

results = xicsrt.raytrace(config)
```

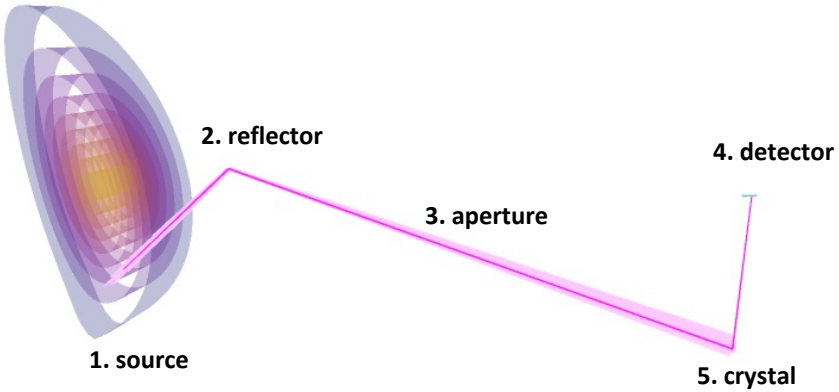
Optics in XICSRT are traced in sequential order

When defining `optics` in the `config`, the order in which they are added determines the raytracing order

Example: plasma → reflector → aperture → crystal → detector

```
config['sources'] = {}
config['sources']['plasma'] = {}

config['optics'] = {}
config['optics']['reflector'] = {}
config['optics']['aperture'] = {}
config['optics']['crystal'] = {}
config['optics']['detector'] = {}
```



Raytracing of ITER XRCS-Core spectrometer with pre-reflector

Only rays that are transmitted/reflected are sent to the next optic

- **xicsrt** uses probabilistic filtering (explained later in this tutorial)
- A mask array is used to track which rays have been lost

The `results` dictionary contains a full history of the ray intersections at each element

The contents of the **results** are controlled by the options:

`keep_meta`, `keep_history`, `keep_images`

- Turning off the **history** or **images** can improve performance

The ray **history** is divided into **lost** and **found** rays

- **Lost** rays are **randomly filtered** to conserve memory
- (explained later in this tutorial)

This ray **history** allows complete **2D** and **3D** analysis or plotting

The **results** dictionary can be saved as an **hdf5** file

- Simple to reload into python with included tools
- Enables post-analysis in other software environments

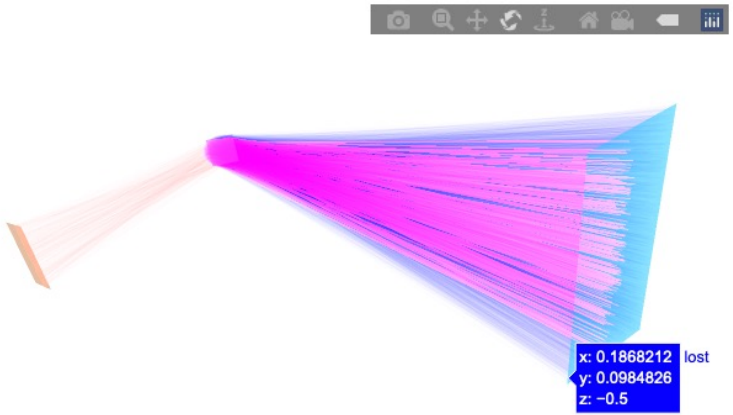
Example of a results dictionary structure:

```
results =
|--config:
|  |...
|--total:
|  |--meta:
|  |  |...
|  |--image:
|     |--crystal: [...]
|     |--detector: [...]
|--found:
|  |--history:
|     |--source:
|     |  |--origin: [...]
|     |  |--direction: [...]
|     |  |--wavelength: [...]
|     |  |--mask: [...]
|     |--crystal:
|     |  |--origin: [...]
|     |  |...
|     |--detector:
|        |--origin: [...]
|        |...
|--lost:
   |--history:
      |--source:
      |  |--origin:[...]
      |  |...
      |--crystal:
      |  |--origin: [...]
      |  |...
      |--detector:
      |  |--origin: [...]
      |  |...
```

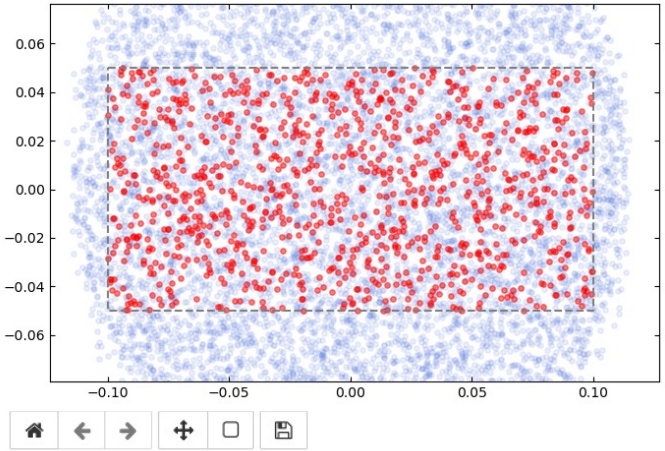
Simple 2d and 3d plotting functions are available as examples of raytracing visualization

Interactive visualization is performed using the `matplotlib` and `plotly` libraries

```
import xicsrt.visual.xicsrt_3d__plotly as xicsrt_3d
xicsrt_3d.plot(results)
```




```
import xicsrt.visual.xicsrt_2d__matplotlib as xicsrt_2d
xicsrt_2d.plot_intersect(results, 'detector')
```



Built-in plotting routines should be considered as examples only!

- You are encouraged to develop more advanced visualizations using the information in `results`

Additional XICSRT components are available as part of a contributed repository: `xicsrt_contrib`

The **xicsrt_contrib** package contains additional sources, optics and filters contributed by **xicsrt** users.

Contains extra objects have not been included as built-in objects for one of the following reasons:

- Usage is too specific for general use
- Non-standard external dependencies
- Performance and stability not at production quality

Installation through `pip` recommended

```
pip install xicsrt_contrib
```

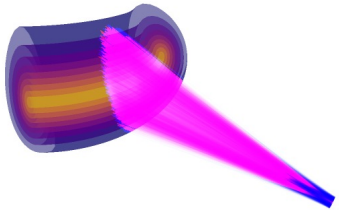
Once installed, the contributed modules will be natively available in xicsrt

Git Repository: https://bitbucket.org/amicitas/xicsrt_contrib
Git Mirror: https://github.com/PrincetonUniversity/xicsrt_contrib

Examples of contributed sources:

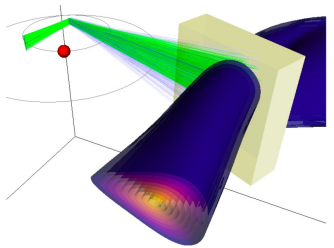
`XicsrtPlasmaImas`

- ITER equilibrium taken from the IMAS database.
- Can be run directly on the ITER high performance cluster (HPC) or used with a previously generated IMAS save file.



`XicsrtPlasmaVmec`

- Stellarator VMEC equilibrium defined by a `wout` file.
- Requires the package `stelltools` which is written in FORTRAN and must be compiled.



XICSRT Concepts

Sources and optics are placed within a global 3D coordinate system

All components must have a location and orientation

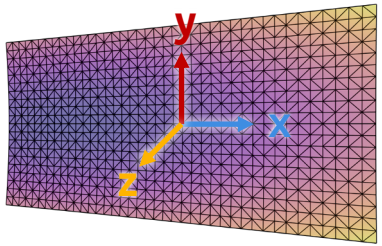
- Defined by: `'origin'`, `'zaxis'` and `'xaxis'`*
- `yaxis` is defined by the cross-product of `xaxis` and `zaxis` and never needs to be specified

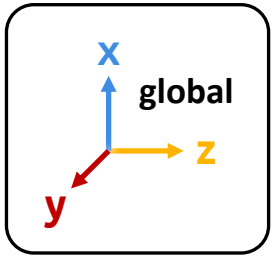
These vectors also define a local coordinate system for each optic

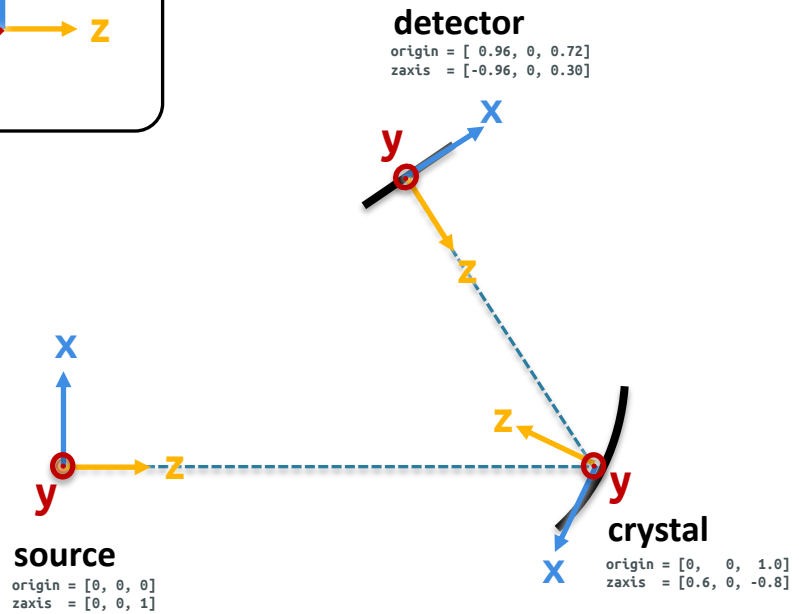
- Useful for plotting and visualization

Conventions in **xicsrt** (when applicable):

- The `origin` is centered on the optic surface
- The `zaxis` is the "normal" direction of the optic







* If `xaxis` is not specified it will be determined by the cross-product of `zaxis` and the global y-direction

Ray sources generate random rays with a given angle and wavelength distribution

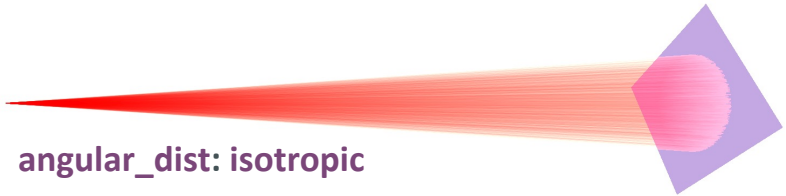
Angular distributions define the cone of rays emitted from any given point within the source

- Specified through `angluar_dist`
- **Half-angle** of distribution defined by `spread`
- Available: `isotropic`, `isotropic_xy`, `flat`, `flat_xy`, `gaussian`

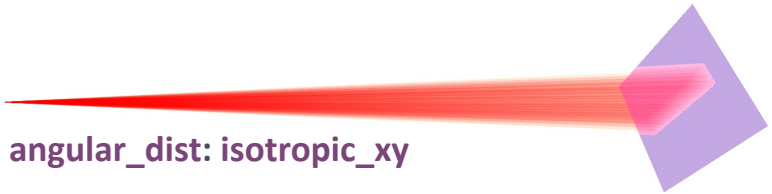
Several **wavelength distributions** are also available

- Specified through `wavelength_dist`
- Additional options depend on the distribution
- Available: `voigt`, `uniform`, `monochrome`
- In progress: `spectrum_file`, `voigt_energy`, `uniform_energy`
- (`voigt` can also be used for Gaussians or Lorentzians)

It is easy to program new distributions in **xicsrt**!



angular_dist: isotropic



angular_dist: isotropic_xy

angular_dist: gaussian

XICRT uses probabilistic ray filtering when considering reflection or transmission through an optic

(While the discussion below is for reflective optics, transmissive optics would behave similarly.)

Rays are treated as individual photons that have a finite probability of reflection at each element

- Ray reflection is binary: reflected or not-reflected
- Rays are randomly filtered according to this reflection probability at each intersection
- *No ray weighting is used within* ***xicsrt***

Ray reflection probability can be different for each ray

- For Bragg reflection the probability will be dependent on the wavelength and incident angle

The use of probabilistic filtering means that the counts in any saved images exactly follow a Poisson distribution

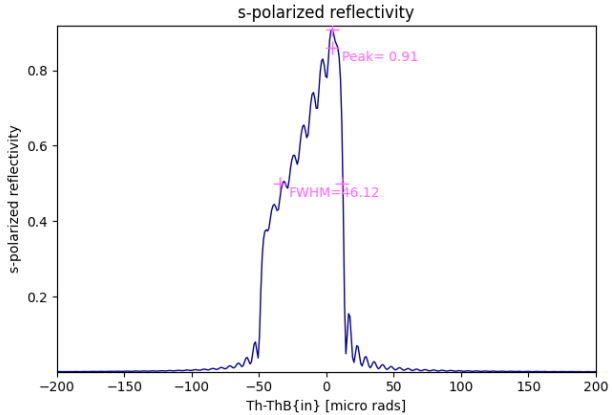
- Error analysis is easy with no additional information

This behavior is different from other raytracing codes

- SHADOW uses ray weighting instead

Advantages of probabilistic ray filtering

- Much more memory efficient
- Potentially much more computationally efficient



Rocking curve from XOP: reflection probability vs angle

After raytracing rays are sorted into lost and found rays

X-Ray diagnostics are often highly inefficient

- These systems often have a small effective étendue
- Consequence of narrow rocking curves
- *Most* generated rays will not reach the detector
- Large number of rays are needed to built statistics

Found: rays that *reach* the detector

Lost: rays that *don't reach* the detector

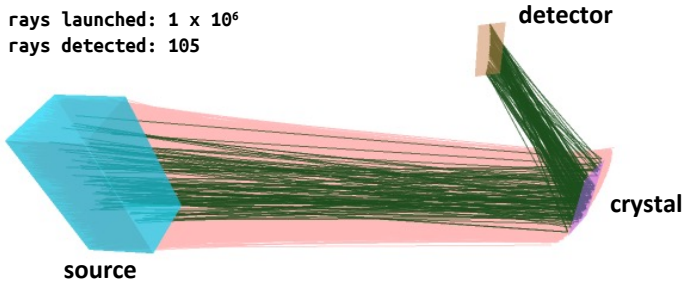
The ray `history` is split into lost and found rays

- Lost ray history is randomly filtered to keep only `max_lost_rays`
- All the found rays are always retained

Lost rays are helpful for diagnostic and visualization purposes

- Not all lost rays can be kept due to memory limitations
- But typically not all lost rays are needed!

Saved ***images*** show total ray intersections (both found and lost rays)



simplified results dictionary

```
results =
|--found:
|  |--history:
|     |--source:
|     |  |--origin: [...]
|     |  |--direction: [...]
|     |  |--wavelength: [...]
|     |  |--mask: [...]
|     |--crystal:
|     |  |--origin: [...]
|     |  |...
|     |--detector:
|        |--origin: [...]
|        |...
|--lost:
   |--history:
      |--source:
      |  |...
      |--crystal:
      |  |...
      |--detector:
      |  |...
```

Ray-bundles are used when modeling complex plasma ray sources

Standard ray sources generate rays uniformly within a volume

- Good for modeling of spatially uniform sources
- Example: XicsrtSourceFocused

Plasma sources can have complex spatial distributions

- Based on physics parameters (emissivity, temperature, velocity)
- Example: `XicsrtPlasmaCubic`

Determination of plasma parameters from real-space coordinates can be computationally expensive

Ray-bundles are used to enable modeling of complex plasmas

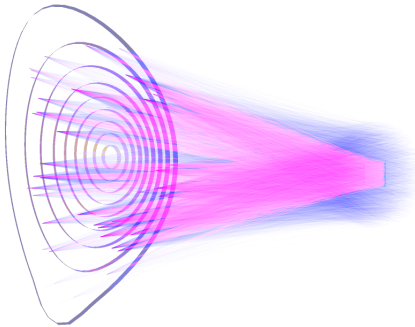
1. Spatial locations within the plasma are uniformly generated
2. Plasma parameters are determined at each location
3. A ray-bundle is generated with the appropriate wavelength distribution and number of rays

Ray-bundles represent a small volume within the plasma

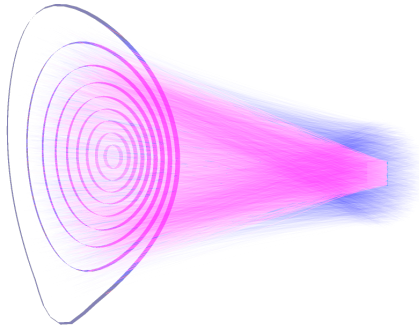
- Number of rays in each bundle is calculated to represent the total emissivity from the bundle volume

Rays from each bundle can be emitted as either a `point` or small volumetric (`voxel`) source

- Voxel bundles produce a more even ray distribution
- But also limit the spatial resolution of the simulation



100 bundles, point sources



100,000 bundles, voxel sources

Mesh optics can be used to model arbitrarily complex surface shapes

Optics with simple geometry are modeled analytically

- Fast and accurate: plane, sphere, cylinder, torus, etc.

For complex geometries it is not always easy or possible to analytically calculate ray intersections and surface normals

Mesh optics utilize a triangulated mesh surface

- Can be used to model *any* complex surface shape
- No analytical solutions for ray intersections are necessary

Mesh surfaces require only a set of x, y, z points on the surface

- Triangulated mesh faces can be provided by the user or automatically generated

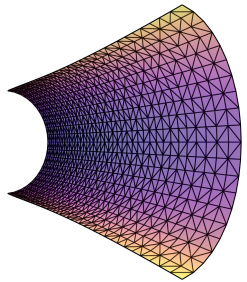
Example class: `XicsrtOpticMeshCrystal`

Raytracing *accuracy* can be improved by additionally providing a surface normal vector at each mesh point

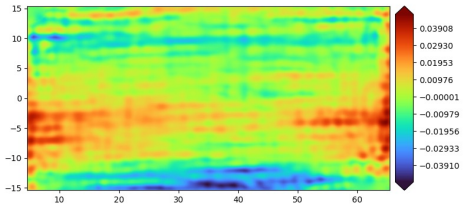
- Normal vectors are interpolated to the ray intersections
- Fewer mesh points are required when normals are provided

Raytracing *performance* can be improved by additionally providing points on a coarse mesh

- Coarse mesh is used to pre-select the faces of the fine mesh that need to be tested for intersections



Mesh optic for VR-Spiral geometry



Measured crystal surface variations imported into XICSRT using a high-resolution mesh

Apertures can be defined to give optics complex outlines

All optics (optionally) have an `xsize`, `ysize` and `zsize`

- Can be used to define rectangular bounds for raytracing
- Will be used to generate pixelated images with `pixel_size`

More complex shapes are defined by an `aperture` list

- Each entry in the list defines a logical operation
- Aperture entries are option dictionaries
- Many shapes and logical operators are available

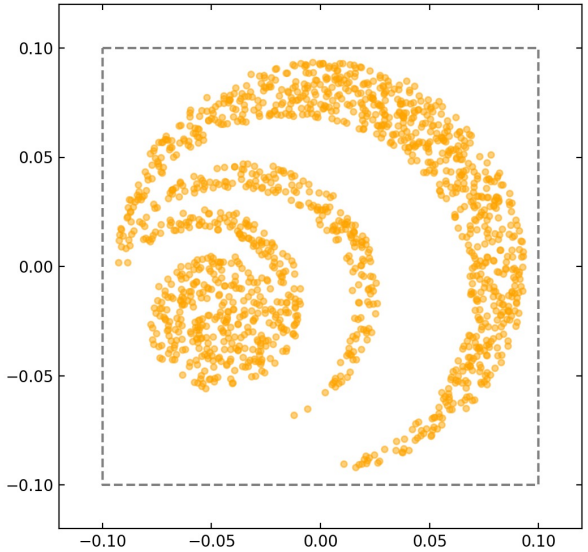
Included aperture shapes

- square, rectangle, triangle, circle, ellipse

Size and aperture checking are controlled with:

- `check_size` and `check_aperture`

```
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'},
    ]
```



Filters can be used to add behavior to sources and optics*

Filters allow a way to provide reusable extensions than can be applied to multiple sources and/or optics

Definition is the same as for sources and optics

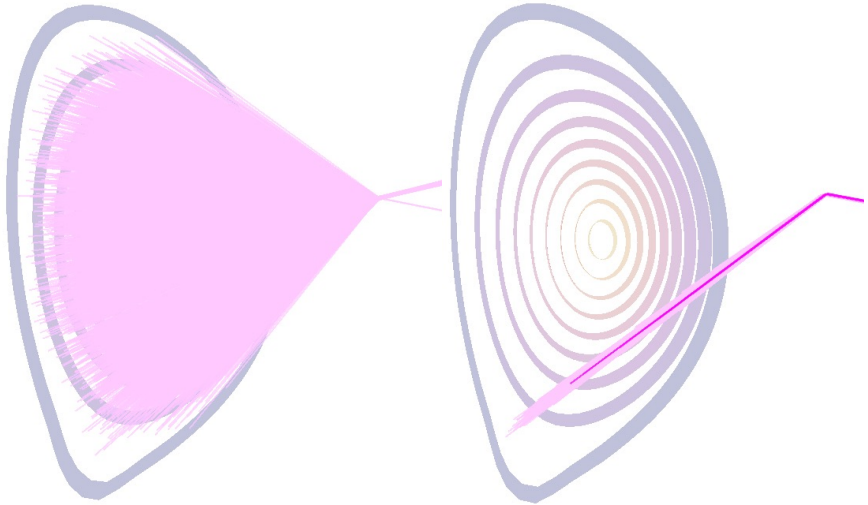
```
config['filters'] = {}
config['filters']['sightline'] = {...}
```

Filters then need to be attached to the desired element

```
config['sources']['plasma']['filters'] = ['sightline']
```

- Attachment of a filter to multiple elements is possible

ITER XRCS-Core raytracing



Without filter most rays are launched outside of the viewing volume. No rays reached the detector.

With filter only the region around the viewing volume is sampled. Efficiency is dramatically improved.

* No optics in **xicsrt** currently implement filters

Raytracing can use multiple processors though python multiprocessing

A multiprocessing version of `raytrace()` is available to take advantage of multiple processors

```
results = xicsrt.raytrace_mp(config, processes=5)
```

To use `raytrace_mp()` the raytracing needs to be performed using a series of runs

- All runs will be combined in the final results dictionary

The number of runs and iterations are defined in config

- `number_of_runs`, `number_of_iter`
- Each run will perform the specified number of iterations
- All raytracing objects are rebuilt for each run
- Objects are reused between iterations

Intersection images are created at the end of each run

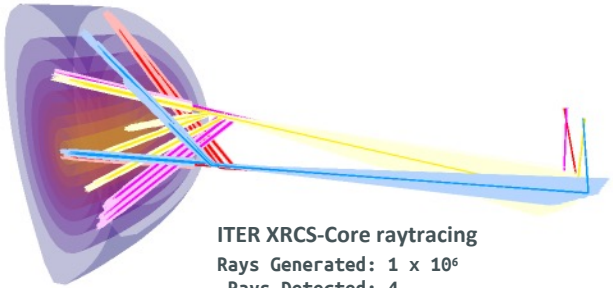
- Useful behavior for long runs when interruptions are possible

Be careful about excessive memory usage!

- Limit the number of rays per iteration

Multiprocessing will not provide a simple multiplier based on processors

- Some of the `numpy` routines used in **xicsrt** already use multiple processors
- There can be overhead in combining processes due to the large memory footprint



ITER XRCS-Core raytracing
Rays Generated: 1 x 10^6
Rays Detected: 4
Efficiency: 0.00042%

XICSRT Development

New optics or sources are defined by subclassing one of the existing XICSRT classes (object-oriented design)

New optics can be added without modifying any of the **xicsrt** source code.

New classes are put into specially named files and stored in a local directory

- Use the `config` option `'pathlist'` to add your local objects directory so that **xicsrt** can find the new objects
- Objects must be saved in files starting with '`_Xicsrt`'
- Example name: `_XicsrtSourceFocused.py`

All **xicsrt** objects use several layers of inheritance

- Use the online docs to easily navigate though inherited classes
- Only minimal code is needed to define new behavior

Simple examples of source and optics construction:

source:	`XicsrtSourceFocused`
plasma:	`XicsrtPlasmaGeneric`
optic:	`XicsrtOpticPlanarMirror`

See online documentation for more information *(coming soon)*.

`_XicsrtSourceFocused.py`

```python
# -*- coding: utf-8 -*-
"""
.. Authors
    Novimir Pablant <npablant@pppl.gov>
    James Kring <jdk0026@tigermail.auburn.edu>
    Yevgeniy Yakusevich <eugenethree@gmail.com>
"""

import numpy as np

from xicsrt.tools.xicsrt_doc import dochelper
from xicsrt.sources._XicsrtSourceGeneric import XicsrtSourceGeneric

@dochelper
class XicsrtSourceFocused(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.
    """

    def default_config(self):
        """
        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`.
        """
        config = super().default_config()
        config['target'] = None
        return config

    def generate_direction(self, origin):
        normal = self.make_normal_focused(origin)
        D = super().random_direction(normal)
        return D

    def make_normal_focused(self, origin):
        # Generate ray from the origin to the focus.
        normal = self.param['target'] - origin
        normal = normal / np.linalg.norm(normal, axis=1)[:, np.newaxis]
        return normal
```

XICSRT Validation

XICSRT has been validated against the SHADOW code

SHADOW is the premier code for x-ray raytracing

- Extensively used for synchrotron radiation beamlines
- In continuous development since 1982!
- Thoroughly validated against experimental data

An identical scenario was setup in both codes to model an x-ray spectrometer with a toroidal crystal

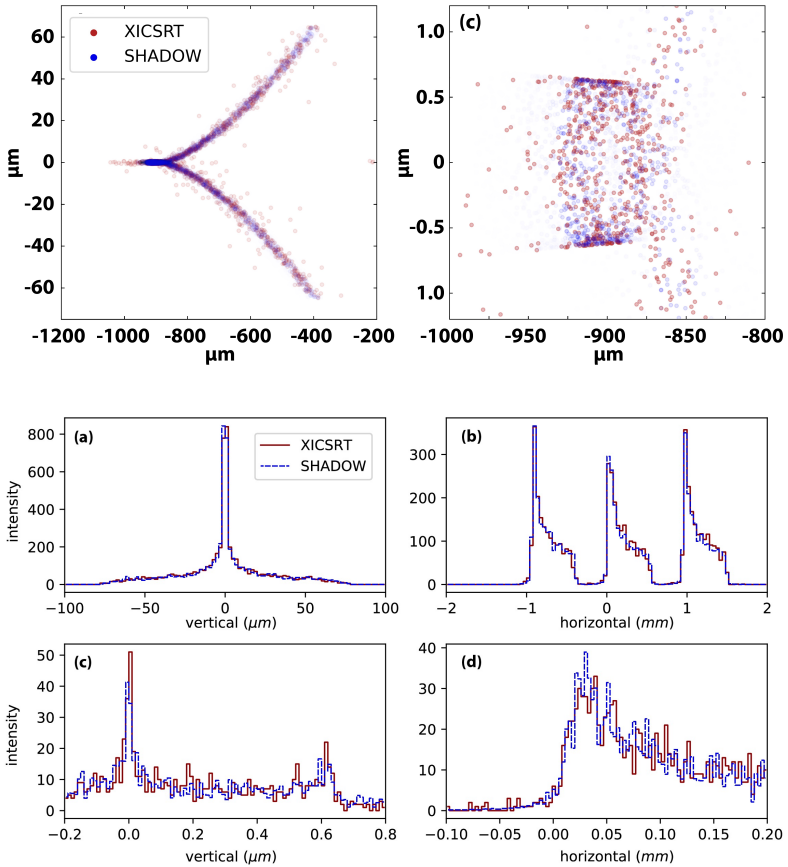
- Ge (4 0 0) crystal with full bent-crystal rocking curve from XOP
- Major Radius: 2200mm, Minor Radius: 200mm, Size: 60 x 30 mm
- Point source with central energy: 9818.8 eV (tungsten $L\beta_1$)
- 3 x 10^7 totally rays launched (5300 detected)

For XICSRT a mesh-grid representation was used for the crystal

- Mesh consisted of **121 x 61** points with normal vectors

Codes agree in both ray distribution and total intensity

- Detector ray patterns agree to within ***5nm***
- Intensity on detectors agree within Poisson noise



Concluding Remarks

XICSRT is ready for use, please give it a try

xicsrt is always being **improved** and **extended**

- New versions are released regularly, update often!

API is still being refined and major updates may require changes to your config dictionary or scripts.

Contributions to the **xicsrt** codebase are **encouraged**!

- Repository on bitbucket.org is used for development

Options for contributing:

1. Submit bug reports and feature requests
2. Fork the repository then submit pull requests
3. Ask for developer access to the **xicsrt** repo

Please **contact me** with any questions or issues

Novimir Pablant <npablant@pppl.gov>

XICSRT: Photon based raytracing in Python

Documentation:	https://xicsrt.readthedocs.org
Git Repository:	https://bitbucket.org/amicitas/xicsrt
Git Mirror:	https://github.com/PrincetonUniversity/xicsrt
PyPi:	https://pypi.org/project/xicsrt/
OSTI:	https://www.osti.gov/doecode/biblio/55767

Fin