RayPyNG

Simone Vadilonga, Ruslan Ovsyannikov

May 16, 2025

CONTENTS:

raypyng provides a simple python API to work with RAY-UI, a software for optical simulation of synchrotron beamlines and x-ray systems developed by Helmholtz-Zentrum Berlin.

INSTALLATION

raypyng will work only if using a Linux or a macOS distribution.

1.1 Install RAY-UI

Download the RAY-UI installer from this link, and run the installer.

1.2 Install xvfb

xvfb is a virtual X11 framebuffer server that let you run RAY-UI headless

Install xvfb:

```
sudo apt install xvfb
```

> ℹ **Note**
>
> xvfb-run script is a part of the xvfb distribution and runs an app on a new virtual X11 server.

1.3 Install raypyng

- You will need Python 3.8 or newer. From a shell ("Terminal" on OSX), check your current Python version.

  ```
  python3 --version
  ```

  If that version is less than 3.8, you must update it.

  We recommend installing raypyng into a "virtual environment" so that this installation will not interfere with any existing Python software:

  ```
  python3 -m venv ~/raypyng-tutorial
  source ~/raypyng-tutorial/bin/activate
  ```

  Alternatively, if you are a conda user, you can create a conda environment:

  ```
  conda create -n raypyng-tutorial "python>=3.8"
  conda activate raypyng-tutorial
  ```

- Install the latest versions of raypyng and ophyd. Also, install IPython (a Python interpreter designed by scientists for scientists).

```
python3 -m pip install --upgrade raypyng ipython
```

- Start IPython:

```
ipython --matplotlib=qt5
```

The flag `--matplotlib=qt5` is necessary for live-updating plots to work.

Or, if you wish you use raypyng from a Jupyter notebook, install a kernel like so:

```
ipython kernel install --user --name=raypyng-tutorial --display-name "Python␣
↪(raypyng)"
```

You may start Jupyter from any environment where it is already installed, or install it in this environment along-side raypyng and run it from there:

```
pip install notebook
jupyter notebook
```

TUTORIAL

2.1 Manipulate an RML file

Using the `RMLFile` class it is possible to manupulate an beamline file produced by RAY-UI.

```
In [8]: from raypyng.rml import RMLFile
 ...: rml = RMLFile('rml/elisa.rml')
 ...: rml
Out[8]: RMLFile('rml/elisa.rml',template='rml/elisa.rml')
```

The filename can be accesed with the filename `attribute`

```
In [9]: rml.filename
Out[9]: 'rml/elisa.rml'
```

and the beamline is available under:

```
In [10]: elisa = rml.beamline
In [11]: elisa
Out[11]: XmlElement(name = beamline, attributes = {}, cdata = )
```

It is possible to list all the element present in the beamlne using the `children()` method

```
In [14]: for i, oe in enumerate(elisa.children()):
   ...:     print('OE ',i, ':', oe.resolvable_name())
   ...:
OE  0 : Dipole
OE  1 : M1
OE  2 : PremirrorM2
OE  3 : PG
OE  4 : M3
OE  5 : ExitSlit
OE  6 : KB1
OE  7 : KB2
OE  8 : DetectorAtFocus
```

In a similar way one can print all the available paramters of a certain element. For instance, to print all the parameters of the Dipole:

```
In [15]: # print all the parameters of the Dipole
   ...: for param in elisa.Dipole.children():
   ...:     print('Dipole param: ', param.id)
```

```
   ...:
Dipole param:   numberRays
Dipole param:   sourceWidth
Dipole param:   sourceHeight
Dipole param:   verEbeamDiv
Dipole param:   horDiv
Dipole param:   electronEnergy
Dipole param:   electronEnergyOrientation
Dipole param:   bendingRadius
Dipole param:   alignmentError
Dipole param:   translationXerror
Dipole param:   translationYerror
Dipole param:   rotationXerror
Dipole param:   rotationYerror
Dipole param:   energyDistributionType
Dipole param:   photonEnergyDistributionFile
Dipole param:   photonEnergy
Dipole param:   energySpreadType
Dipole param:   energySpreadUnit
Dipole param:   energySpread
Dipole param:   sourcePulseType
Dipole param:   sourcePulseLength
Dipole param:   photonFlux
Dipole param:   worldPosition
Dipole param:   worldXdirection
Dipole param:   worldYdirection
Dipole param:   worldZdirection
```

Any parameter can be modified in this way:

```
In [17]: elisa.Dipole.photonEnergy.cdata
Out[17]: '1000'

In [18]: elisa.Dipole.photonEnergy.cdata = str(2000)

In [19]: elisa.Dipole.photonEnergy.cdata
Out[19]: 2000
```

Once you are done with the modifications, you can save the rml file using the `write()` method

```
rml.write('rml/new_elisa.rml')
```

2.2 RAY-UI API

Using the `RayUIRunner` and the `RayUIAPI` classes it is possible to interact with RAY-UI directly from python.

```
In [1]: import os
 ...: import time
 ...: from raypyng.runner import RayUIRunner, RayUIAPI
 ...:
 ...: r = RayUIRunner(ray_path=None, hide=True)
```

```
 ...: a = RayUIAPI(r)

In [2]: r.run()
Out[2]: <raypyng.runner.RayUIRunner at 0x7effd8f53b50>
```

Once an instance of RAY-UI is running, we can confirm that it is running and we can ask the `pid`

```
In [3]: r.isrunning
Out[3]: True

In [4]: r.pid
Out[4]: 20742
```

It is possible to load an rml file and trace it

```
In [5]: a.load('rml/elisa.rml')
 ...:
Out[5]: True

In [6]: a.trace(analyze=True)
 ...:
Out[6]: True
```

Export the files for the elements of interest:

```
In [7]: a.export("Dipole,DetectorAtFocus", "RawRaysOutgoing", '/home/simone/Documents/
→RAYPYNG/raypyng/examples', 'test_export')
 ...:
Out[7]: True
```

Save the rml file used for the simulation (this is useful because RAY-UI when it traces the beamline it updates the RML files with the latest parameters: for instance if you change the photon energy, it will update the source flux)

```
In [8]: a.save('rml/new_elisa')
Out[8]: True
```

And finally we can quit the RAY-UI instance that we opened:

```
In [9]: a.quit()
```

2.3 Simulations

2.3.1 Perform Simulations

raypyng is not able to create a beamline from scratch. To do so, use RAY-UI, create a beamline, and save it. What you save is `.rml` file, which you have to pass as an argument to the `Simulate` class. In the following example, we use the file for a beamline called *elisa*, and the file is saved in `rml/elisa.rml`. The `hide` parameter can be set to true only if *xvfb* is installed.

```
from raypyng import Simulate
rml_file = 'rml/elisa.rml'
```

```
sim = Simulate(rml_file, hide=True)
elisa = sim.rml.beamline
```

The elements of the beamline are now available as python objects, as well as their properties. If working in ipython, tab autocompletion is available. For instance to access the source, a dipole in this case:

```
# this is the dipole object
elisa.Dipole
# to acess its parameter, for instance, the photonFlux
elisa.Dipole.photonFlux
# to access the value
elisa.Dipole.photonFlux.cdata
# to modify the value
elisa.Dipole.photonFlux.cdata = 10
```

To perform a simulation, any number of parameters can be varied. For instance, one can vary the photon energy of the source, and set a a certain aperture of the exit slits:

```
# define the values of the parameters to scan
energy    = np.arange(200, 7201,250)
SlitSize  = np.array([0.1])

# define a list of dictionaries with the parameters to scan
params = [
          {elisa.Dipole.photonEnergy:energy},
          {elisa.ExitSlit.totalHeight:SlitSize}
       ]

#and then plug them into the Simulation class
sim.params=params
```

It is also possible to define coupled parameters. If for instance, one wants to increase the number of rays with the photon energy

```
# define the values of the parameters to scan
energy    = np.arange(200, 7201,250)
nrays     = energy*100
SlitSize  = np.array([0.1])

# define a list of dictionaries with the parameters to scan
params = [
          {elisa.Dipole.photonEnergy:energy, elisa.Dipole.numberRays:nrays},
          {elisa.ExitSlit.totalHeight:SlitSize}
       ]

#and then plug them into the Simulation class
sim.params=params
```

The simulations files and the results will be saved in a folder called *RAYPy_simulation_* and a name of your choice, that can be set. This folder will be saved, by default, in the folder where the program is executed, but it can eventually be modified

```
sim.simulation_folder = '/home/raypy/Documents/simulations'
sim.simulation_name = 'test'
```

This will create a simulation folder with the following path and name

```
/home/raypy/Documents/simulations/RAYPy_simulation_test
```

Sometimes, instead of using millions of rays, it is more convenient to repeat the simulations and average the results We can set which parameters of which optical elements can be exported. The number of rounds of simulations can be set like this:

```
# repeat the simulations as many times as needed
sim.repeat = 1
```

One can decide whether want RAY-UI or raypyng to do a preliminary analysis of the results. To let RAY-UI analyze the results, one has to set:

```
sim.analyze = True # let RAY-UI analyze the results
```

In this case, the following files are available to export:

```
print(sim.possible_exports)
> ['AnglePhiDistribution',
> 'AnglePsiDistribution',
> 'BeamPropertiesPlotSnapshot',
> 'EnergyDistribution',
> 'FootprintAbsorbedRays',
> 'FootprintAllRays',
> 'FootprintOutgoingRays',
> 'FootprintPlotSnapshot',
> 'FootprintWastedRays',
> 'IntensityPlotSnapshot',
> 'IntensityX',
> 'IntensityYZ',
> 'PathlengthDistribution',
> 'RawRaysBeam',
> 'RawRaysIncoming',
> 'RawRaysOutgoing',
> 'ScalarBeamProperties',
> 'ScalarElementProperties']
```

To let raypyng analyze the results set:

```
sim.analyze = False # don't let RAY-UI analyze the results
sim.raypyng_analysis=True # let raypyng analyze the results
```

In this case, only these exports are possible

```
print(sim.possible_exports_without_analysis)
> ['RawRaysIncoming', 'RawRaysOutgoing']
```

The exports are available for each optical element in the beamline, ImagePlanes included, and can be set like this:

```
## This must be a list of dictionaries
sim.exports  =  [{elisa.Dipole:['ScalarElementProperties']},
                 {elisa.DetectorAtFocus:['ScalarBeamProperties']}
                 ]
```

Finally, the simulations can be run using

```
sim.run(multiprocessing=5, force=True)
```

where the *multiprocessing* parameter can be an integer greater or equal to 1, corresponding to the number of parallel instances of RAY-UI to be used. Generally speaking, the number of instances of RAY-UI must be lower or equal than the number of available cores. If the simulation uses many rays, monitor the RAM usage of your computer. If the computation uses all the possible RAM of the computer the program may get blocked or not execute correctly.

Note on multiprocessing

The speed increase due to opening many RAY-UI instances is effective only when RAY-UI is not performing the analysis of the results.

```
sim.analyze = False # don't let RAY-UI analyze the results
```

There is little/no difference having RayPyNG analyzing the results .. code-block:: python

> sim.raypyng_analysis=True # let raypyng analyze the results

2.3.2 Simulation Output

Expect this folders and subfolders to be created:

```
RAYPy_simulation_mySimulation
├── round_0
│   ├── 0_*.rml
│   └── 0_*.csv
│   └── 0_*.dat (only if raypyng analyzes the results)
│   └── ...
│   └── looper.py
...
├── round_n
│   ├── 0_*.rml
│   └── 0_*.csv
│   └── 0_*.dat (only if raypyng analyzes the results)
│   └── ...
│   └── looper.py
├── input_param_1.dat
...
├── input_param_k.dat
├── output_simulation.dat (only if raypyng analyzes the results)
```

Analysis performed by RAY-UI

If you decided to let RAY-UI do the analysis, you should expect the following files to be saved in your simulation folder:

- one file for each parameter you set with the values that you passed to the program. If for instance, you input the Dipole numberRays, you will find a file called *input_param_Dipole_numberRays.dat*

- one folder called *round_n* for each repetition of the simulations. For instance, if you set `sim.repeat=2` you will have two folders *round_0* and *round_1*

- inside each *round_n* folder you will find the beamline files modified with the parameters you set in *sim.params*, these are the *.rml* files, that can be opened by RAY-UI.

- inside each *round_n* folder you will find your exported files, one for each simulation. If for instance, you exported the *ScalarElementProperties* of the Dipole, you will have a list of files *0_Dipole-ScalarElementProperties.csv*

- *looper.csv* each simulation and its parameters.

Analysis performed by raypyng

If you decided to let raypyng do the analysis, you should expect the following files to be saved in your simulation folder:

- one file for each parameter you set with the values that you passed to the program. If for instance, you input the Dipole numberRays, you will find a file called *input_param_Dipole_numberRays.dat*

- one folder called *round_n* for each repetition of the simulations. For instance, if you set `sim.repeat=2` you will have two folders *round_0* and *round_1*

- inside each *round_n* folder you will find the beamline files modified with the parameters you set in *sim.params*, these are the *.rml* files, that can be opened by RAY-UI.

- inside each *round_n* folder you will find your exported files, one for each simulation. If for instance, you exported the *RawRaysOutgoing* of the Dipole, you will have a list of files *0_Dipole-RawRaysOutgoing.csv*

- for each *RawRaysOutgoing* file, raypyng calculates some properties, and saves a corresponding file, for instance *0_Dipole_analyzed_rays.dat*. Each of these files contains the following information:

  - SourcePhotonFlux

  - NumberRaysSurvived

  - PercentageRaysSurvived

  - PhotonFlux

  - Bandwidth

  - HorizontalFocusFWHM

  - VerticalFocusFWHM

- In the simulation folder, all the for each exported element is united (and in case of more rounds of simulations averaged) in one single file. For the dipole, the file is called *Dipole.dat*

- *looper.csv* each simulation and its parameters.

2.4 Recipes

raypyng provides some recipes to make simulations, that simplify the syntax in the script. Two recipes are provided, one to make Resolving Power simulations, one to make Flux simulations.

2.5 List of available examples

In the example folder, the following examples are available:

- example_rml.py in this example is shown how to read, manipulate and save an rml file.

- example_runner.py in this example is shown how to use the RAY-UI API to start RAY-UI, load a file, trace it and export the desired results.

- example_simulation_analyze.py simulate a beamline, let Ray-UI do the analysis

- example_simulation_noanalyze.py simulate a beamline, let raypyng do the analysis

- example_eval_noanalyze_and_analyze.py plots the results of the two previous simulations

- example_simulation_Flux.py simulations using the flux recipe, useful if you intend to simulate the flux of your beamline

- example_simulation_RP.py simulations using the resolving power (RP) recipe, useful if you intend to simulate the RP of your beamline. The reflectivity of every optical element is switched to 100% and not calculated using the substrate and coating(s) material(s). The information about the Flux of the beamline is therefore not reliable.

- example_beamwaist.py raypyng can plot the beam waist of the x-rays across your beamline. It performs simulations using the beam waist recipe, and it exports the RawRaysOutgoing file from each optical element. It then uses a simple geometrical x-ray tracer to propagate each ray until the next optical element and plots the results (both top view and side view). This is still experimental and it may fail.

- example_waveHelper.py The WaveHelper class can help to inspect a folder produced by WAVE and provides a simple API to get the path of the simulation files depending on the harmonic and on the energy.

HOW TO GUIDES

To simplify the scripting, especially when repetitive, there is the possibility to write recipe for raypyng, to perform simulations, and automatize some tasks.

3.1 Write your own Recipe

3.1.1 Recipe Template

This the template to use to write a recipe. At the beginning of the file import `SimulationRecipe` from `raypyng` and define a the Simulation class as an empty dummy. This will ensure that you have access to all the methods of the `Simulation` class.

A recipe should containe at least the `__init__()` method and three more methods: `params()`, and `simulation_name()`, and they must have as an argument the simulate class.

Compose the simulation parameters in the `params` method: The simulation parameter must return a list of dictionaries, where the keys of the dictionaries are parameters of on abject present in the beamline, instances of `ParamElement` class. The items of the dictionary must be the values that the parameter should assume for the simulations.

Compose the simulation parameters in the `params()` method: The `params()` method must return a list of dictionaries. The keys of the dictionaries are parameters of on abject present in the beamline, instances of `ParamElement` class. The items of the dictionary must be the values that the parameter should assume for the simulations.

Compose the export parameters in the `exports()` method: The The `exports()` method must return a list of dictionaries, method must return a list of dictionaries. The keys of the dictionaries are parameters of on abject present in the beamline, instances of `ParamElement` class. The items of the dictionary is the name of the file that you want to export (print the output of `Simulation.possible_exports` and `possible_exports_without_analysis`.

Define the name to give to the simulation folder in `simulation_name()`

```python
from raypyng.recipes import SimulationRecipe

class Simulate: pass

class MyRecipe(SimulationRecipe):
    def __init__(self):
        pass

    def params(self,sim:Simulate):

        params = []

        return params
```

```python
    def exports(self,sim:Simulate):

        exports = []

        return exports

    def simulation_name(self,sim:Simulate):

        self.sim_folder = ...

        return self.sim_folder
```

3.1.2 How To Write a Recipe

An example of how to write a recipe that exports file for each element present in the beamline automatically.

```python
class ExportEachElement(SimulationRecipe):
    """At one defined energy export a file for each
    optical elements
    """
    def __init__(self, energy:float,/,nrays:int=None,sim_folder:str=None):
        """
        Args:
            energy_range (np.array, list): the energies to simulate in eV
            nrays (int): number of rays for the source
            sim_folder (str, optional): the name of the simulation folder. If None,
→ the rml filename will be used. Defaults to None.

        """

        if not isinstance(energy, (int,float)):
            raise TypeError('The energy must be an a int or float, while it is a',
→type(energy))

        self.energy = energy
        self.nrays  = nrays
        self.sim_folder = sim_folder

    def params(self,sim:Simulate):
        params = []

        # find source and add to param with defined user energy range
        found_source = False
        for oe in sim.rml.beamline.children():
            if hasattr(oe,"photonEnergy"):
            self.source = oe
                found_source = True
                break
        if found_source!=True:
            raise AttributeError('I did not find the source')
        params.append({self.source.photonEnergy:self.energy})
```

```python
    # set reflectivity to 100%
    for oe in sim.rml.beamline.children():
            for par in oe:
                try:
                    params.append({par.reflectivityType:0})
                except:
                    pass

    # all done, return resulting params
    return params

def exports(self,sim:Simulate):
    # find all the elements in the beamline
    oe_list=[]
    for oe in sim.rml.beamline.children():
        oe_list.append(oe)
    # compose the export list of dictionaries
    exports = []
    for oe in oe_list:
        exports.append({oe:'RawRaysOutgoing'})
    return exports

def simulation_name(self,sim:Simulate):
    if self.sim_folder is None:
        return 'ExportEachElement'
    else:
        return self.sim_folder
if __name__ == "__main__":
    from raypyng import Simulate
    import numpy as np
    import os

    rml_file = ('rml_file.rml')
    sim      = Simulate(rml_file, hide=True)


    sim.analyze = False

    myRecipe = ExportEachElement(energy=1000,nrays=10000,sim_folder=
↪'MyRecipeTest')

    # test resolving power simulations
    sim.run(myRecipe, multiprocessing=5, force=True)
```

3.2 How to work with Undulator File

The *WaveHelper* class helps to inspect a WAVE simulation folder and provides a simple way to extract the absolute path of the simulation files to feed to the *Undulator File*. In this example we use the *WAVE* folder provided in the example folder at this link. Inside the folder there are WAVE simulation files for the first, third and fifth harmonic, and the Undulator is called *U49*

```
import numpy as np

from raypyng.wave_helper import WaveHelper

WH = WaveHelper(wave_folder_path='WAVE', harmonic=3, undulator='U49')

WH.report_available_energies(verbose=True)
```

This produces the following output:

```
I found the following harmonics: dict_keys([1, 3, 5])
the energy points for each harmonic are equally spaced
Harmonic number 1, available energies:
start 80
stop 570
step 10
Harmonic number 3, available energies:
start 240
stop 1710
step 30
Harmonic number 5, available energies:
start 400
stop 2850
step 50
```

We can now extract the file location for all the energies or a subset of the energies available for the first harmonic of the undulator:

```
energies = np.arange(80,570,10)
energy_files = WH.convert_energies_to_file_list(1,energies)
```

energy_files contains the absolute path to the WAVE simulation file for each energy. This can be used to change the energy of an Undulator by calling the parameter *undulatorFile*.

API

4.1 Simulation

4.1.1 Simulate

class raypyng.simulate.**Simulate**(*rml=None*, *hide=False*, *ray_path=None*, ***kwargs*)

A class that takes care of performing the simulations with RAY-UI

> **Parameters**
>
> - **rml** (*RMLFile/string, optional*) – string pointing to an rml file with the beamline template, or an RMLFile class object. Defaults to None.
>
> - **hide** (*bool, optional*) – force hiding of GUI leftovers, xvfb needs to be installed. Defaults to False.
>
> - **ray_path** (*str, optional*) – the path to the RAY-UI installation folder. If None, the program will look for RAY-UI in the standard installation paths.

alignment_errors(*value*)

Switch the alignment errors of all the optical elements in the beamline on or off.

> **Parameters**
>
> **value** (*bool, optional*) – If *True*, the alignment errors are switched on. If *False*, the alignment errors are switched off.
>
> **Returns**
>
> None

property analyze

Turn on or off the RAY-UI analysis of the results. The analysis of the results takes time, so turn it on only if needed

> **Returns**
>
> True: analysis on, False: analysis off
>
> **Return type**
>
> bool

cleanup_child_processes()

Clean up all child processes initiated by this process and any specific Xvfb processes.

property efficiency

The parameters to scan, as a list of dictionaries.

For each dictionary the keys are the parameters elements of the beamline, and the values are the values to be assigned.

property exports

> Get the list of files to export after the simulation is complete.

property params

> The parameters to scan, as a list of dictionaries.
>
> For each dictionary the keys are the parameters elements of the beamline, and the values are the values to be assigned.

property path

> The path where to execute the simlations
>
> > **Returns**
> >
> > > by default the path is the current path from which the program is executed
> >
> > **Return type**
> >
> > > string

property possible_exports

> A list of the files that can be exported by RAY-UI
>
> > **Returns**
> >
> > > list of the names of the possible exports for RAY-UI
> >
> > **Return type**
> >
> > > list

property possible_exports_without_analysis

> A list of the files that can be exported by RAY-UI when the analysis option is turned off
>
> > **Returns**
> >
> > > list of the names of the possible exports for RAY-UI when analysis is off
> >
> > **Return type**
> >
> > > list

property raypyng_analysis

> Turn on or off the RAYPyNG analysis of the results.
>
> > **Returns**
> >
> > > True: analysis on, False: analysis off
> >
> > **Return type**
> >
> > > bool

reflectivity(*value*)

> Switch the reflectivity of all the optical elements in the beamline on or off.
>
> > **Parameters**
> >
> > > **value** (`bool, optional`) – If `True` the reflectivity is switched on, if `False` the reflectivity is switched off.

property repeat

> The simulations can be repeated an arbitrary number of times
>
> If the statitcs are not good enough using 2 millions of rays is suggested to repeat them instead of increasing the number of rays
>
> > **Returns**
> >
> > > the number of repetition of the simulations, by default is 1

> **Return type**
>> int

property rml

> RMLFile object instantiated in init

rml_list(*recipe=None*, *overwrite_rml=True*)

> Creates the folder structure and RML files needed for simulation. This method organizes simulation parameters into RML files and prepares the directory structure for simulations, which is useful for pre-simulation checks and manual adjustments.

> > **Parameters**
> > - **recipe** (`SimulationRecipe, optional`) – Recipe to use for setting up the simulation. Defaults to None.
> > - **overwrite_rml** (`bool, optional`) – If True, existing RML files will be overwritten. Defaults to True.

run(*recipe=None*, *multiprocessing=1*, *force=False*, *overwrite_rml=True*, *force_exit=True*, *remove_rawrays=False*, *remove_round_folders=False*)

> Execute simulations with optional recipe, multiprocessing, and file management options.

> This method orchestrates the setup and execution of simulations, managing multiprocessing, file generation, and progress tracking.

> > **Parameters**
> > - **recipe** (`SimulationRecipe, optional`) – Recipe for simulation setup. Defaults to None.
> > - **multiprocessing** (`int, optional`) – Number of processes for parallel execution. Defaults to 1.
> > - **force** (`bool, optional`) – Force re-execution of simulations. Defaults to False.
> > - **overwrite_rml** (`bool, optional`) – Overwrite existing RML files. Defaults to True.
> > - **force_exit** (`bool, optional`) – calls os.exit when the simulations are complete. Nothing else will run after it. Defaults to True.
> > - **remove_rawrays** (`bool, optional`) – removes RawRaysIncoming and RawRaysOutgoing files, if present.
> > - **remove_round_folders** (`bool, optional`) – remove the round folders after the simulations are done.

property simulation_name

> A string to append to the folder where the simulations will be executed.

slope_errors(*slope_errors*)

> Switch the slope errors of all the optical elements in the beamline on or off.

> > **Parameters**
> > **value** (`bool, optional`) – If *True* the slope errors are switched on, if *False* the slope errors are switched off.

property undulator_table

> The undulator table, as a pandas DataFrame.

4.1.2 SimulationParams

class raypyng.simulate.**SimulationParams**(*rml=None*, *param_list=None*, ***kwargs*)

> Handles the setup and management of simulation parameters for RAY-UI simulations.
>
> This class is responsible for organizing simulation parameters, both independent and dependent, and generating the necessary parameter sets for conducting simulations.
>
> **rml**
>
> > The RML file or the path to the RML file used as the template for simulations.
> >
> > **Type**
> > > *RMLFile* or str
>
> **params**
>
> > A list of dictionaries where each dictionary represents a set of parameters to simulate. Each key in the dictionary is a ParamElement, and its value is the parameter value(s) to simulate.
> >
> > **Type**
> > > list of dict
>
> **simulation_parameters_generator**()
>
> > Generates a dictionary of parameters for each simulation based on the input parameter list.
> >
> > **Yields**
> > > *dict* – A dictionary of parameters for a single simulation.

4.1.3 WaveHelper

Some source in RAY-UI can take as an input a file obtained by simulating the insertion device with WAVE. This class inspects the folder where the WAVES results are stored and provides a simple way to get a list of absolute paths to the simulation files.

class raypyng.wave_helper.**WaveHelper**(*wave_folder_path: str*, *harmonics: int*, *undulator: str*, ***kwargs*)

> Explore WAVE simulation folder and gives an easy way to convert energies into filenames
>
> The class expects the WAVE simulations to be stored in a folder called WAVE. For instance for an undulator called 'U49' and harmonic=3, this filestructure is expected:
>
> Wave ├── U49H1allrayfiles ├── U49H3allrayfiles ├── U49H5allrayfiles
>
> > **Parameters**
> >
> > - **wave_folder_path** (*str*) – the path to the WAVW folder where the WAVE simulations are stored
> >
> > - **harmonics** (*int*) – the number of harmonics present.(If harmonics=2, simulations for 1st and 3rd should exist)
> >
> > - **undulator** (*str*) – the undulator name as indicated in the simulation folders
>
> **convert_energies_to_file_list**(*harmonic: int*, *energies: list*)
>
> > Takes the harmonic and a list of energies and returns the files location
> >
> > > **Parameters**
> > >
> > > - **harmonic** (*int*) – the harmonic that you want to have
> > >
> > > - **energies** (*list*) – list of int, the x-ray energies
> > >
> > > **Raises**
> > > > **ValueError** – If an energy is not present in the WAVE simulation folder.

> **Returns**
>> list of absolute paths for each energy given as an input
>
> **Return type**
>> list

report_available_energies(*verbose=True*)

> Report about the availbale energies and explore the WAVE folder
>
>> **Parameters**
>>> **verbose** (`bool, optional`) – If Ture a report about the neergie is printed. Defaults to True.

4.2 Recipes

4.2.1 Resolving Power

class raypyng.recipes.**ResolvingPower**(*energy_range: range*, *exported_object:* ObjectElement, */ (Positional-only parameter separator (PEP 570)), *args*, *source:* ObjectElement = *None*, *sim_folder: str = None*)

> Recipe for resolving power simulations.
>
> The eflectivity is automatically switched off for all elements, the source if automatically recognized.
>
>> **Parameters**
>>
>> - **energy_range** (`np.array, list`) – the energies to simulate in eV
>> - **exported_object** (`ObjectElement`) – the object to export
>> - **source** (`ObjectElement, optional`) – the source object. If None is provided, an automatic recogniton of the source will be tried. Defaults to None.
>> - **sim_folder** (`str, optional`) – the name of the simulation folder. If None, the rml filename will be used. Defaults to None.

4.2.2 Flux

class raypyng.recipes.**Flux**(*energy_range: range*, *exported_object:* ObjectElement, */, *args*, *source:* ObjectElement = *None*, *sim_folder: str = None*)

> Recipe for flux simulations.
>
> The reflectivity is automatically switched on for all elements, and the source is automatically identified.
>
>> **Parameters**
>>
>> - **energy_range** (`np.array, list`) – the energies to simulate in eV
>> - **exported_object** (`ObjectElement`) – the object to export
>> - **source** (`ObjectElement, optional`) – the source object. If None is provided, an automatic recogniton of the source will be tried. Defaults to None.
>> - **sim_folder** (`str, optional`) – the name of the simulation folder. If None, the rml filename will be used. Defaults to None.

4.3 Process simulation files

4.3.1 PostProcess rays analyzed by raypyng

class raypyng.postprocessing.**PostProcess**

 class to post-process the data.

 It works only if the exported data are RawRaysOutgoing

 cleanup(*dir_path: str = None*, *repeat: int = 1*, *exp_elements: list = None*, *exported_file_type=None*)

  Reads all the results of the postprocessing process and summarize them in a single file for each exported object.

  This functions reads all the temporary files created by `self.postprocess_RawRays()` saves one file for each exported element in dir_path, and deletes the temporary files. If more than one round of simulations was done, the values are averaged.

  Parameters

   • **dir_path** (`str,` `optional`) – The path to the folder to cleanup. Defaults to None.

   • **repeat** (`int,` `optional`) – number of rounds of simulations. Defaults to 1.

   • **exp_elements** (`list,` `optional`) – the exported elements names as str. Defaults to None.

 extract_bw_from_source(*rml_filename*)

  Extract photon energy from rml file, find source automatically

  Parameters

   rml_filename (`str`) – the rml file to use to extract the photon flux

  Returns

   the photon energy

  Return type

   str

 extract_energy_from_source(*rml_filename*)

  Extract photon energy from rml file, find source automatically

  Parameters

   rml_filename (`str`) – the rml file to use to extract the photon flux

  Returns

   the photon energy

  Return type

   str

 extract_nrays_from_source(*rml_filename*)

  Extract photon flux from rml file, find source automatically

  Parameters

   rml_filename (`str`) – the rml file to use to extract the photon flux

  Returns

   the photon flux

  Return type

   str

 postprocess_RawRays(*exported_element: str = None*, *exported_object: str = None*, *dir_path: str = None*, *sim_number: str = None*, *rml_filename: str = None*, *suffix: str = None*, *remove_rawrays: bool = False*, *undulator_table=None*, *efficiency=None*)

PostProcess routine of the RawRaysOutgoing extracted files.

The method looks in the folder *dir_path* for a file with the filename:

```
filename = os.path.join(dir_path, sim_number + exported_element + '-' +
exported_object + '.csv')
```

For each file, it calculates the number of rays, bandwidth, horizontal and vertical focus size. It saves the results in an array formatted as:

```
[n_rays, bandwidth, hor_focus, vert_focus]
```

The array is then saved to:

```
os.path.join(dir_path, sim_number + exported_element + '_analyzed_rays.
npy')
```

Parameters

- **exported_element** (`list`, `optional`) – List of exported element names as strings. Defaults to None.

- **exported_object** (`str`, `optional`) – The exported object, tested only with RawRaysOutgoing. Defaults to None.

- **dir_path** (`str`, `optional`) – Folder where the file to process is located. Defaults to None.

- **sim_number** (`str`, `optional`) – Prefix of the file, typically the simulation number with a prepended _, e.g., `0_`. Defaults to None.

4.3.2 PostProcess rays analyzed by RAY-UI

class raypyng.postprocessing.**PostProcessAnalyzed**

class to analyze the data exported by RAY-UI

moving_average(*x*, *w*)

Computes the morivng average with window w on the array x

Parameters

- **x** (`array`) – the array to average

- **w** (`int`) – the window for the moving average

Returns

the x array once the moving average was applied

Return type

np.array

retrieve_bw_and_focusSize(*folder_name: str*, *oe: str*, *nsimulations: int*, *rounds: int*)

Extract the bandwidth and focus size from ScalarBeamProperties of an object.

Parameters

- **folder_name** (`str`) – the path to the folder where the simulations are

- **oe** (`str`) – the optical element name

- **nsimulations** (`int`) – the number of simulations

- **rounds** (`int`) – the number of rounds of simulations

Returns
the bandwidth foc_x np.array: the horizontal focus foc_y np.array: the vertical focus

Return type
bw np.array

retrieve_flux_beamline(*folder_name*, *source*, *oe*, *nsimulations*, *rounds=1*, *current=0.3*)

Extract the flux from ScalarBeamProperties and from ScalarElementProperties.

This function takes as arguments the name of the simulation folder, the exported objet in RAY-UI and there number of simulations and returns the flux at the optical element in percentage and in number of photons, and the flux produced by the dipole. It requires ScalarBeamProperties to be exported for the desired optical element, if the source is a dipole it requires ScalarElementProperties to be exported for the Dipole

Parameters

- **folder_name** (`str`) – the path to the folder where the simulations are

- **source** (`str`) – the source name

- **oe** (`str`) – the optical element name

- **nsimulations** (`int`) – the number of simulations

- **rounds** (`int`) – the number of rounds of simulations

- **current** (`float, optional`) – the ring current in Ampere. Defaults to 0.3.

Returns

photon_flux (np.array)
[the photon flux at the optical element] flux_percent (np.array) : the photon flux in percentage relative to the source source_Photon_flux (np.array) : the photon flux of the source

else:
flux_percent (np.array) : the photon flux in percentage relative to the source

Return type
if the source is a Dipole

4.3.3 RayProperties

class raypyng.postprocessing.**RayProperties**(*input=None*, *filename=None*, *undulator_table=None*)

concat(*other*)
Concatenates another RayProperties object to current.

save(*filename*)
Save current object into a CSV file.

4.4 RAY-UI API

4.4.1 RayUIRunner

class raypyng.runner.**RayUIRunner**(*ray_path=None*, *ray_binary='rayui.sh'*, *background=True*, *hide=False*)

RayUIRunner class implements all logic to start a RayUI process, load and rml file, trace and export.

Parameters

- **ray_path** (`str, optional`) – the path to the RAY-UI installation folder. Defaults to config.ray_path, that will look for the ray_path in the standard installation folders.

- **ray_binary** (`_type_`, `optional`) – the binary file of RAY-UI. Defaults to "rayui.sh".

- **background** (`bool`, `optional`) – activate background mode. Defaults to True.

- **hide** (`bool`, `optional`) – Hide the RAY-UI graphical instances. Available only if xvfb is installed. Defaults to False.

property isrunning

> Check weather a process is running and rerutn a boolean

> > **Returns**
> >
> > > returns True if the process is running, otherwise False
> >
> > **Return type**
> >
> > > bool

kill()

> kill a RAY-UI process

property pid

> Get process id of the RayUI process

> > **Returns**
> >
> > > PID of the process if it running, None otherwise
> >
> > **Return type**
> >
> > > int

run()

> Open one instance of RAY-UI using subprocess

> > **Raises**
> >
> > > **RayPyRunnerError** – if the RAY-UI executable is not found raise an error

4.4.2 RayUIAPI

class raypyng.runner.**RayUIAPI**(*runner:* RayUIRunner *= None*)

> RayUIAPI class implements (hopefully all) command interface of the RAY-UI

> **export**(*objects: str*, *parameters: str*, *export_path: str*, *data_prefix: str*, ***kwargs*)
>
> > Export simulation results from RAY-UI.
> >
> > > **Parameters**
> > >
> > > - **objects** (`str`) – string with objects list, e.g. "Dipole,DetectorAtFocus"
> > >
> > > - **parameters** (`str`) – stromg with parameters to export,
> > >
> > > - **"ScalarBeamProperties** (`e.g.`)
> > >
> > > - **ScalarElementProperties"**
> > >
> > > - **export_path** (`str`) – path where to save the data
> > >
> > > - **data_prefix** (`str`) – prefix for the putput files

> **load**(*rml_path*, ***kwargs*)
>
> > Load an rml file
> >
> > > **Parameters**
> > >
> > > > **rml_path** (`str`) – path to the rml file

quit()

>   quit RAY-UI if it is running

save(*rml_path*, ***kwargs*)

>   Save an rml file
>
>   > **Parameters**
>   >
>   > > **rml_path** (`path`) – path to save the rml file

trace(*analyze=True*, ***kwargs*)

>   Trace an rml file (must have been loaded before).
>
>   > **Parameters**
>   >
>   > > **analyze** (`bool, optional`) – If True RAY-UI will perform analysis of the rays. Defaults
>   > > to True.

4.5 RML

4.5.1 RMLFile

class raypyng.rml.**RMLFile**(*filename: str = None*, */*, *template: str = None*)

>   Read/Write wrapper for the Ray RML files
>
>   > **Parameters**
>   >
>   > > - **filename** (`str, optional`) – path to rml file. Defaults to None.
>   > >
>   > > - **template** (`str, optional`) – path to rml file to use as template. Defaults to None.

read(*file: str = None*)

>   Read rml file
>
>   > **Parameters**
>   >
>   > > **file** (`str, optional`) – file name to read. If set to None will use template file name defined
>   > > during initilizatino of the class. Defaults to None.

write(*file: str = None*)

>   Write the rml to `file`
>
>   > **Parameters**
>   >
>   > > **file** (`str, optional`) – filename . Defaults to None.

4.5.2 BeamlineElement

class raypyng.rml.**BeamlineElement**(*name: str*, *attributes: dict*, ***kwargs*)

add_cdata(*cdata*)

>   Store cdata

add_child(*element*)

>   Store child elements.

get_attribute(*key*)

>   Get attributes by key

get_elements(*name=None*)

>   Find a child element by name

get_full_path()

> Returns the full path of the xml object

> > **Returns**
> > > path of the xml object

> > **Return type**
> > > str

resolvable_name()

> Returns the name of the objects, removing lab.beamline.

> > **Returns**
> > > name of the object

> > **Return type**
> > > str

4.5.3 ObjectElement

class raypyng.rml.**ObjectElement**(*name: str*, *attributes: dict*, ***kwargs*)

add_cdata(*cdata*)

> Store cdata

add_child(*element*)

> Store child elements.

get_attribute(*key*)

> Get attributes by key

get_elements(*name=None*)

> Find a child element by name

get_full_path()

> Returns the full path of the xml object

> > **Returns**
> > > path of the xml object

> > **Return type**
> > > str

resolvable_name()

> Returns the name of the objects, removing lab.beamline.

> > **Returns**
> > > name of the object

> > **Return type**
> > > str

4.5.4 ParamElement

class raypyng.rml.**ParamElement**(*name: str*, *attributes: dict*, ***kwargs*)

add_cdata(*cdata*)

> Store cdata

add_child(*element*)

> Store child elements.

get_attribute(*key*)

> Get attributes by key

get_elements(*name=None*)

> Find a child element by name

get_full_path()

> Returns the full path of the xml object
>
> > **Returns**
> > path of the xml object
> >
> > **Return type**
> > str

resolvable_name()

> Returns the name of the objects, removing lab.beamline.
>
> > **Returns**
> > name of the object
> >
> > **Return type**
> > str