RayPyNG

Simone Vadilonga, Ruslan Ovsyannikov

Oct 15, 2022

CONTENTS:

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

CHAPTER

ONE

INSTALL

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
```

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 alongside raypyng and run it from there:

```
pip install notebook
jupyter notebook
```

CHAPTER TWO

TUTORIAL

2.1 Minimal example

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',
```

(continues on next page)

(continued from previous page)

```
> '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 set either to False or to an int, 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 than the number of cores available. 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.

2.2 Exports

2.2.1 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*

2.2.2 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*

2.3 Recipes

Documentation is still not available, see the examples.

2.4 List of examples available and short explanation

In the example folder, the following examples are available:

- *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.

CHAPTER

THREE

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 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

```

(continues on next page)

(continued from previous page)

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

        exports = []

        return exports

    def simulation_name(self,sim:Simulate):

        self.sim_folder = ...

        return self.sim_folder
```

3.2 How To Write a Recipe

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

```
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')
```

(continues on next page)

(continued from previous page)

```
        params.append({self.source.photonEnergy:self.energy})

        # 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)
```

CHAPTER FOUR

SIMULATION

4.1 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

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

property exports

The files to export once the simulation is complete. for a list of possible files check self.possible_exports and self.possible_exports_without_analysis.

It is expeceted a list of dictionaries, and for each dictionary the key is the element to be exported and the valuee are the files to be exported

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

`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()`

This function creates the folder structure and the rml files to simulate. It requires the param to be set. Useful if one wants to create the simulation files for a manual check before starting the simulations.

`run`(*recipe=None*, */*, *multiprocessing=True*, *force=False*)

This method starts the simulations. params and exports need to be defined.

Parameters

- **recipe** (*SimulationRecipe, optional*) – If using a recipee pass it as a parameter. Defaults to None.
- **multiprocessing** (*boolint, optional*) – If True all the cpus are used. If an integer n is provided, n cpus are used. Defaults to True.
- **force** (*bool, optional*) – If True all the simlations are performed, even if the export files already exist. If False only the simlations for which are missing some exports are performed. Defaults to False.

`save_parameters_to_file`(*dir*)

save all the user input parameters to file. It takes the values from the SimulationParams class

Parameters
dir (*str*) – the folder where to save the parameters

`property simulation_name`

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

4.2 SimulationParams

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

A class that takes care of the simulations parameters, makes sure that they are written correctly, and returns the the list of simulations that is requested by the user.

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 rml

RMLFile object instantiated in init

CHAPTER FIVE

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.

5.1 Resolving Power

`class raypyng.recipes.`**`ResolvingPower`**`(`*energy_range: range*, *exported_object: ObjectElement*, */*, **args*, *source: Optional[ObjectElement] = None*, *sim_folder: Optional[str] = None*`)`

Resolving Power Simulations, reflectivity is automatically switched off for all elements

5.2 Flux

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

Flux simulations, reflectivity is automatically switched on for all elements

CHAPTER

SIX

PROCESS SIMULATION FILES

6.1 PostProcess rays analyzed by raypyng

class raypyng.postprocessing.**PostProcess**

class to post-process the data. At the moment works only if the exported data are RawRaysOutgoing

cleanup(*dir_path: Optional[str] = None*, *repeat: int = 1*, *exp_elements: Optional[list] = None*)

This functions reads all the temporary files created by `self.postptocess_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_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: Optional[str] = None*, *exported_object: Optional[str] = None*, *dir_path: Optional[str] = None*, *sim_number: Optional[str] = None*, *rml_filename: Optional[str] = None*)

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, the bandwidth, and the horizontal and vertical focus size, it saves it in an array that is composed by `[n_rays,bandwidth,hor_focus,vert_focus]`, that is then saved to `os.path.join(dir_path, sim_number+exported_element+'_analyzed_rays.npy')`

Parameters

- **exported_element** (*list, optional*) – a list of containing the exported elements name as str. Defaults to None.

- **exported_object** (*str, optional*) – the exported object, tested only with RawRaysOutgoing. Defaults to None.
- **dir_path** (*str, optional*) – the folder where the file to process is located. Defaults to None.
- **sim_number** (*str, optional*) – the prefix of the file, that is the simulation number with a _prepended, ie *0_*. Defaults to None.

6.2 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 if RAY-UI was run in analyze mode. It requires ScalarBeamProperties to be exported for the desired optical element

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*)

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

CHAPTER

SEVEN

RAY-UI API

7.1 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

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
type

run()

Open one instance of RAY-UI using subprocess

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

CHAPTER

EIGHT

RML

8.1 RMLFile

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

Read/Write wrapper for the Ray RML files

`read`(*file: Optional[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: Optional[str] = None*)

Write the rml to `file`

Parameters

`file` (*`str, optional`*) – filename . Defaults to None.

8.2 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

8.3 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

8.4 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

INDEX