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.

CHAPTER

ONE

INSTALL

raypyng will work only if using a linux or a MacOS distribustion.

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 distribuion 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, "Command Prompt" on Windows), check your current Python version.

```
python3 --version
```

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

  We recommend install bluesky into a "virtual environment" so 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, that you have to pass as an argumet to the *Simulate* 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 acess 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 acess the value
elisa.Dipole.photonFlux.cdata
# to modify the value
elisa.Dipole.photonFlux.cdata = 10
```

To perform simulation, any number of parameters can varied. For instance one can vary the photon energy of the source, and set 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_* plusa name at your choce, that can be set. This folder will be saved, by default, in the folder where the program is executed, but it can be 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 The 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 time as needed
sim.repeat = 1
```

One can decide weather want RAY-UI or raypyng to do a prelimanary 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 exports are available:

```
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 simulation using 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 perfomed by RAY-UI

If you decided to let RAY-UI doing 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 perfomed by raypyng

If you decided to let raypyng doing 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 followin informations:
  - 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 fill is called *Dipole.dat*

2.2.3 Recipees

Documentation still not available, see the examples.

2.2.4 List of examples avalable 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 Ray-UI do the analysis
- *example_eval_noanalyze_and_analyze.py* plot the results of the two previous simulations
- *example_simulation_Flux.py* simulations using the flux recipee, useful if you intent is to simulate the flux of your beamline
- *example_simulation_RP.py* simulations using the resolving power (RP) recipee, useful if you intent is to simulate the RP of your beamline. The reflectivity of every optical element is switched to 100% and not calculated using the substrate and coiating(s) material(s). The information about the Flux of the beamline are therefore not reliable.
- *example_beamwaist.py*: raypyng is able to plot the beamwaist of the x-rays across your beamline. It performs simulations using the beamwaist recipee, and it exports the raw raysoutgoing 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

SIMULATION

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

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

__init__(*rml=None*, *hide=False*, ***kwargs*) → None

Initialize the class with a rml file :param rml: string pointing to an rml file with the beamline template, or an RMLFile class object. Defaults to None. :type rml: RMLFile/string, optional :param hide: force hiding of GUI leftovers, xvfb needs to be installed. Defaults to False. :type hide: bool, optional

Raises
Exception – If the rml file is not defined an exception is raised

__weakref__

list of weak references to the object (if defined)

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.

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.

__init__(*rml=None*, *param_list=None*, ***kwargs*) → None

summary

Parameters

- **rml** (*RMLFile/string, optional*) – string pointing to an rml file with the beamline template, or an RMLFile class object. Defaults to None.
- **param_list** (*list, optional*) – list of dictionaries containing the parameters and values to simulate. Defaults to None.

__weakref__

list of weak references to the object (if defined)

_calc_loop(*verbose: bool = True*)

Calculate the simulations loop

Returns
idependent and dependent parameters self.simulations_param_list (list): parameters values for each simulation loop

Return type
self.param_to_simulate (list)

_check_if_enabled(*param*)

Check if a parameter is enabled

Parameters
param (*RML object*) – an parameter to simulate

Returns
True if the parameter is enabled, False otherwise

Return type
(bool)

_check_param()

Check that self.param is a list of dictionaries, and convert the items of the dictionaries to lists, otherwise raise an exception.

_enable_param(*param*)

Set enabled to True in a beamline object, and auto to False

Parameters
param (*RML object*) – beamline object

_extract_param(*verbose: bool = False*)

Parse self.param and extract dependent and independent parameters

Parameters
verbose (*bool, optional*) – If True print the returned objects. Defaults to False.

Returns
indieendent parameter values self.ind_par (list): independent parameters self.dep_param_dependency (dict): dictionary of dependencies self.dep_value_dependency (list): dictionaries of dependent values self.dep_par (list): dependent parameters

Return type
self.ind_param_values (list)

_write_value_to_param(*param*, *value*)

Write a value to a parameter, making sure enable is T and auto is F

Parameters

- **param** (*RML object*) – beamline object
- **value** (*str,int,float*) – the value to set the beamline object to

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

FOUR

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

__detect_ray_path() → str

Internal function to autodetect installation path of RayUI

Raises
RayPyRunnerError – is case no ray installations can be detected

Returns
string with the detected ray installation path

Return type
str

__init__(*ray_path=None*, *ray_binary='rayui.sh'*, *background=True*, *hide=False*) → None

__weakref__

list of weak references to the object (if defined)

_readline() → str

read a line from the stdout of the process and convert to a string

Returns
line read from the input

Return type
str

_write(*instr: str*, *endline='\n'*)

Write command to RayUI interface

Parameters

- **instr** (*str*) – _description_
- **endline** (*str, optional*) – _description_. Defaults to endline character.

Raises
RayPyRunnerError – _description_

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

FIVE

POSTPROCESSING

class raypyng.postprocessing.**PostProcess**

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

__init__() → None

__weakref__

list of weak references to the object (if defined)

_extract_bandwidth_fwhm(*rays_bw: array*)

calculate the fwhm of the rays_bw.

Parameters
(np (*rays_bw*) – array): the energy of the x-rays

Returns
fwhm

Return type
float

_extract_focus_fwhm(*rays_pos: array*)

calculate the fwhm of rays_pos

Parameters
rays_pos (*np.array*) – contains positions of the x-rays

Returns
fwhm

Return type
float

_extract_intensity(*rays: array*)

calculate how many rays there are

Parameters
rays (*np.array*) – contains rays information

_list_files(*dir_path: str*, *end_filename: str*)

List all the files in dir_path ending with end_filename

Parameters

- **dir_path** (*str*) – path to a folder
- **end_filename** (*str*) – the listed files end with end_filename

Returns
list of files in dir_path ending eith end_filename

Return type
res (list)

`_load_file`(*filepath*)

Load a .npy file and returns the array

Parameters
filepath (*str*) – the path to the file to load

Returns
The loaded numpy array

Return type
arr (np.array)

`_save_file`(*filename: str*, *array: array*, *header: Optional[str] = None*)

This function is used to save files,

Parameters

- **filename** (*str*) – file name(path)
- **array** (*np.array*) – array to save
- **header** (*str*) – header for the file

`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') :param exported_element:

a list of containing the exported elements name as str. Defaults to None. :type exported_element: list, optional :param exported_object: the exported opbject, tested only with RawRaysOutgoing. Defaults to None. :type exported_object: str, optional :param dir_path: the folder where the file to process is located. Defaults to None. :type dir_path: str, optional :param sim_number: the prefix of the file, that is the simulation number with a _prepended, ie "0_". Defaults to None. :type sim_number: str, optional

INDEX

Symbols

A

C

E

I

K

P

R

S