raypyng-bluesky

Simone Vadilonga, Ruslan Ovsyannikov

Dec 02, 2022

CONTENTS:

This is a raypyng-Bluesky interface library that allows using the Bluesky data acquisition framework to run RAY-UI simulations. It can be used to implement a digital twin at the beamline. This package is based on Bluesky, raypyng and RAY-UI.

Bluesky Raypyng Ray-UI

CHAPTER

ONE

INSTALLATION

raypyng-bluesky 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-bluesky

- 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-bluesky-tutorial
  source ~/raypyng-bluesky-tutorial/bin/activate
  ```

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

  ```
  conda create -n raypyng-bluesky-tutorial "python>=3.8"
  conda activate raypyng-bluesky-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-bluesky 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-bluesky-tutorial --display-name
↪"Python (raypyng-bluesky)"
```

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 Setup an Ipython profile

The code is thought to be used in an environment where bluesky is setup. For doing this it is convenient to create an ipython profile and modify the startup files. An rml file created with RAY-UI containg a beamline is also needed. The code in the following example and an rml file ready to use is available at this link In the startup folder of the ipython profile create a file called `0-bluesky.py` that contains a minimal setup of bluesky and raypyng-bluesky.

The first part of the file contains a minimal installation of bluesky

```
import os
from bluesky import RunEngine
from raypyng_bluesky.RaypyngOphydDevices import RaypyngOphydDevices

RE = RunEngine({})

# Send all metadata/data captured to the BestEffortCallback.
from bluesky.callbacks.best_effort import BestEffortCallback
bec = BestEffortCallback()


# Make plots update live while scans run.
from bluesky.utils import install_kicker
install_kicker()

# create a temporary database
from databroker import Broker
db = Broker.named('temp')
RE.subscribe(db.insert)
RE.subscribe(bec)

# import the magics
from bluesky.magics import BlueskyMagics
get_ipython().register_magics(BlueskyMagics)

# import plans
from bluesky.plans import (
    relative_scan as dscan,
    scan, scan as ascan,
    list_scan,
    rel_list_scan,
```

(continues on next page)

(continued from previous page)

```
    rel_grid_scan,  rel_grid_scan as dmesh,
    list_grid_scan,
    adaptive_scan,
    rel_adaptive_scan,
    inner_product_scan            as a2scan,
    relative_inner_product_scan   as d2scan,
    tweak)

# import stubs
from bluesky.plan_stubs import (
    abs_set,rel_set,
    mv, mvr,
    trigger,
    read, rd,
    stage, unstage,
    configure,
    stop)
```

The last part contains the the two lines of code used to create RaypyngOphyd devices. See the API documentation for more details about `RaypyngOphydDevices`. If you already have an ipython profile with Bluesky you can just add these lines.

```
# insert here the path to the rml file that you want to use
rml_path = '...rml/elisa.rml'

RaypyngOphydDevices(RE=RE, rml_path=rml_path, temporary_folder=None, name_space=None,␣
↪prefix=None, ray_ui_location=None)
```

The ipython profile can be started using:

```
ipython --profile=raypyng-bluesky-tutorial --matplotlib=qt5
```

All the elements present in the rml file as ophyd devices. If you set `prefix=None`, the prefix `rp_` is automatically prepended to the name of the optical elements found in the rml file to create the dame of the object in python. If you have a Dipole called `Dipole`, then the name would be: `rp_Dipole`. You can now use the simulated motors as you would normally do in bluesky.

To see a list of the implemented motors and detectors use the ipython autocompletion by typing in the ipython shell

```
rp_
```

and pressing `tab`.

2.2 What can go wrong, and how to correct it

If once you setup the ipython profile as explained in the section above no elements are found, might be that the `RaypyngOphydDevices` class fails to insert the ophyd devices in the correct namespace. In this case try to call the classes passing explicitly the correct namespace like this:

```
import sys
RaypyngOphydDevices(RE=RE, rml_path=rml_path, temporary_folder=None, name_space=sys._
↪getframe(0), prefix=None, ray_ui_location=None)
```

If when you start a scan (see section below in this tutorial), RAY-UI is not found, it is because you installed it in a non-standard location. In this case simply pass the absolute path of the folder where you installed RAY-UI to the class:

```
ray_path = ... # here the path to RAY-UI folder
RaypyngOphydDevices(RE=RE, rml_path=rml_path, temporary_folder=None, name_space=None,␣
↪prefix=None, ray_ui_location=ray_path)
```

2.3 RaypyngOphyd - Motors

Presently only a subset of the parameters available in rml file in RAY-UI are implemented as motor axes. To see which ones are available, use the tab-autocompletion. For instance, to see what axes are available for the motor `rp_Dipole` write in the ipython shell:

```
rp_Dipole.
```

and press tab: among the other things you will see that are implemented `rp_Dipole.nrays`, the number of rays to use in the simulation, and `p_Dipole.en`, the photon energy in eV. You can of course also use the `.get()` and `.set()` methods:

```
In [1]: rp_Dipole.en.get()
Out[1]: 1000.0

In [2]: rp_Dipole.en.set(1500)
Out[2]: <ophyd.sim.NullStatus at 0x7fbf4c25adc0>

In [3]: rp_Dipole.en.get()
Out[3]: 1500.0
```

For a complete description of the axis available for each optical element see the API documentation

2.4 RaypyngOphyd - Detectors

When an `ImagePlane`, or an `ImagePlaneBundle` is found in the rml file, a detector is created. Each detector can return four properties of the x-ray beam. For instance, for the `DetectorAtFocus`: - `rp_DetectorAtFocus.intensity`: the intensity [Ph/s/A/BW] - `rp_DetectorAtFocus.bw`: the bandwidth [eV] - `rp_DetectorAtFocus.hor_foc`: the horizontal focus [um] - `rp_DetectorAtFocus.ver_foc`: the vertical focus [um]

2.5 A scan in Bluesky

It is possible to do scan using the simulation engine RAY-UI as it is normally done in bluesky. For instance you can scan the photon energy and see the intensity at the source and and the sample position. While at the beamline to change the energy we would simply ask the monochromator to do it, for the simulations one needs to change the energy of the source

```
RE(scan([rp_DetectorAtSource.intensity,rp_DetectorAtFocus.intensity], rp_Dipole.en, 200,␣
↪2200, 11))
```

CHAPTER

THREE

HOW TO GUIDES

3.1 Change grating

This feature is stil experimental, and the implementation is somehow poor. However, the method can still be used to implement a grating change.

When the `RaypyngOphydDevices` class is called, Ophyd devices are automatically created .. code:: python

> from raypyng_bluesky.RaypyngOphydDevices import RaypyngOphydDevices
>
> # define here the path to the rml file rml_path = ('...rml/elisa.rml')
>
> RaypyngOphydDevices(RE=RE, rml_path=rml_path, temporary_folder=None, name_space=None, ray_ui_location='/home/simone/RAY-UI-development')#sys._getframe(0))

In this case we know that inside the `elisa.rml` file we have a plane grating monochromator, and an elment that is the a plane grating called `PG`, and that an Ophyd device called `rp_PG` is therefore created. The gratings can hold any number of different configurations, and the configuration found in the rml file is saved with the name `'default'` We can rename the default grating to a more meaningful name, in this case since it is a 1200 lines/mm blazed grating we will call it simply '1200' .. code:: python

> rp_PG.rename_default_grating('1200')

the second grating is a laminar grating with a pitch of 400 lines/mm.

```
rrp_PG.gratings=('400', {'lineDensity':400,
                     'orderDiffraction':1,
                     'lineProfile':'laminar',
                     'aspectAngle':90,
                     'grooveDepth':15,
                     'grooveRatio':0.65,}
            )
```

To change the gratings then, one can use the method implmented in the grating Ophyd device to change the grating, giving as argument the pitch of the grating. To use the blazed grating use:

```
rp_PG.change_grating('1200')
```

while to use the laminar grating:

```
rp_PG.change_grating('400')
```

Four different kind of gratings can be implemented: `blazed`, `laminar`, `sinus`, and `unknown`. Each grating needs slightly different parameters:

```
grating_dict_keys_blazed = {'name':
                                {'lineDensity':value,
                                'orderDiffraction':value,
                                'lineProfile':value,
                                'blazeAngle':value,
                                'aspectAngle':value,
                                }
                            }
grating_dict_keys_laminar = {'name':
                                {'lineDensity':value,
                                'orderDiffraction:value',
                                'lineProfile':value,
                                'aspectAngle':value,
                                'grooveDepth':value,
                                'grooveRatio':value,
                                }
                            }
grating_dict_keys_sinus   = {'name':
                                {'lineDensity':value,
                                'orderDiffraction':value,
                                'lineProfile':value,
                                'grooveDepth':value,
                                }
                            }

grating_dict_keys_unknown = {'name':
                                {'lineDensity':value,
                                'orderDiffraction':value,
                                'lineProfile':value,
                                'gratingEfficiency':value
                                }
                            }
```

CHAPTER

FOUR

API

4.1 Create Ophyd Devices from rml file

4.1.1 RaypyngOphydDevices

class raypyng_bluesky.RaypyngOphydDevices.**RaypyngOphydDevices**(**args*, *RE*, *rml_path*, *temporary_folder=None*, *name_space=None*, *prefix=None*, *ray_ui_location=None*, ***kwargs*)

Create ophyd devices from a RAY-UI rml file and adds them to a name space.

If you are using ipython `sys._getframe(0)` returns the name space of the ipython instance. (Remember to `import sys`)

Parameters

- **RE** (*RunEngine*) – Bluesky RunEngine
- **rml_path** (*str*) – the path to the rml file
- **temporary_folder** (*str*) – path where to create temporary folder. If None it is automatically set into the ipython profile folder. Default to None.
- **name_space** (*frame, optional*) – If None the class will try to understand the correct namespace to add the Ophyd devices to. If the automatic retrieval fails, pass `sys._getframe(0)`. Defaults to None.
- **prefix** (*str*) – the prefix to prepend to the oe names found in the rml file
- **ray_ui_location** (*str*) – the location of the RAY-UI installation folder. If None the program will try to find it automatically. Deafault to None.

append_preprocessor()

Add supplemental data to the RunEngine to trigger the simulations

create_raypyng_elements_from_rml()

Iterate through the raypyng objects created by RMLFile and create corresponding Ophyd Devices

Returns
the Ophyd devices created

Return type
OphydDevices

create_trigger_detector()

Create a trigger detector called RaypyngTriggerDetector

prepend_to_oe_name()

Prepend a prefix to the name of all the Ophyd object created

trigger_detector()

Return the RaypyngTriggerDetector

Returns
the trigger detector

Return type
RaypyngTriggerDetector (*RaypyngTriggerDetector*)

4.1.2 RaypyngDictionary

class raypyng_bluesky.RaypyngOphydDevices.**RaypyngDictionary**(**args*, ***kwargs*)

A class defining a dictionary of the differen elements in rayui and the classe to be used as Ophyd devices

4.2 Ophyd Signals

4.2.1 RayPySignal

class raypyng_bluesky.signal.**RayPySignal**(**args*, ***kwargs*)

get()

The readback value

put(**args*, ***kwargs*)

Put updates the internal readback value

The value is optionally checked first, depending on the value of force. In addition, VALUE subscriptions are run.

Extra kwargs are ignored (for API compatibility with EpicsSignal kwargs pass through).

Parameters

- **value** (*any*) – Value to set
- **timestamp** (*float, optional*) – The timestamp associated with the value, defaults to time.time()
- **metadata** (*dict, optional*) – Further associated metadata with the value (such as alarm status, severity, etc.)
- **force** (*bool, optional*) – Check the value prior to setting it, defaults to False

set(*value*)

Set is like *put*, but is here for bluesky compatibility

Returns
st – This status object will be finished upon return in the case of basic soft Signals

Return type
Status

4.2.2 RayPySignalRO

class raypyng_bluesky.signal.**RayPySignalRO**(**args*, ***kwargs*)

put(*value*, ***, *timestamp=None*, *force=False*)

Put updates the internal readback value

The value is optionally checked first, depending on the value of force. In addition, VALUE subscriptions are run.

Extra kwargs are ignored (for API compatibility with EpicsSignal kwargs pass through).

Parameters

- **value** (*any*) – Value to set
- **timestamp** (*float, optional*) – The timestamp associated with the value, defaults to time.time()
- **metadata** (*dict, optional*) – Further associated metadata with the value (such as alarm status, severity, etc.)
- **force** (*bool, optional*) – Check the value prior to setting it, defaults to False

set(*value*, ***, *timestamp=None*, *force=False*)

Set is like *put*, but is here for bluesky compatibility

Returns
st – This status object will be finished upon return in the case of basic soft Signals

Return type
Status

4.3 Ophyd Axes

4.3.1 Axes

class raypyng_bluesky.axes.**RaypyngAxis**(**args*, ***kwargs*)

The Axis used by all the Raypyng devices.

At the moment it is a comparator, in the future some other positioner will be used, probably a SoftPositioner. The class defines an empty dictionary, the `axes_dict` that will be then filled by each device.

get()

return the value of a certain axis as in the RMLFile

Returns
the value of the axis in the RML file

Return type
float

property position

The current position of the motor in its engineering units :returns: **position** :rtype: any

set(*value*)

Write a value in the RMLFile for a certain element/axis

Parameters
value (*float, int*) – the value to set to the axis

set_axis(*obj*, *axis*)

Set what axis should be used, based on the axes_dict

Parameters

- **obj** (*_type_*) – _description_
- **axis** (*_type_*) – _description_

4.3.2 SimulatedAxisSource

class raypyng_bluesky.axes.**SimulatedAxisSource**(**args*, ***kwargs*)

Define basic properties of the source.

Available axes: - number of rays - photon energy [eV] - bandwidth [% of photon energy]

4.3.3 class SimulatedAxisMisalign(RaypyngAxis):

class raypyng_bluesky.axes.**SimulatedAxisMisalign**(**args*, ***kwargs*)

Define misalignment axes

- translationXerror
- translationYerror
- translationZerror
- rotationXerror
- rotationYerror
- rotationZerror

4.3.4 SimulatedAxisAperture

class raypyng_bluesky.axes.**SimulatedAxisAperture**(**args*, ***kwargs*)

Define basic properties of the aperture, the width and the height.

4.3.5 SimulatedAxisGrating

class raypyng_bluesky.axes.**SimulatedAxisGrating**(**args*, ***kwargs*)

Define basic properties of the gratings:

- lineDensity
- orderDiffraction
- cFactor
- lineProfile
- blazeAngle
- aspectAngle

- grooveDepth
- grooveRatio

4.4 Ophyd Detectors

4.4.1 Detector

class raypyng_bluesky.detector.**RaypyngDetector**(**args*, *information_to_extract='intensity'*, *parent_detector_name=None*, ***kwargs*)

Defining a raypyng detector using a signal. The detector is created when an Image Plane or an Image Plane Bundle is found in the rml file.

get()

The readback value

put(*value*, ***, *timestamp=None*, *force=False*)

Put updates the internal readback value

The value is optionally checked first, depending on the value of force. In addition, VALUE subscriptions are run.

Extra kwargs are ignored (for API compatibility with EpicsSignal kwargs pass through).

Parameters

- **value** (*any*) – Value to set
- **timestamp** (*float, optional*) – The timestamp associated with the value, defaults to time.time()
- **metadata** (*dict, optional*) – Further associated metadata with the value (such as alarm status, severity, etc.)
- **force** (*bool, optional*) – Check the value prior to setting it, defaults to False

set(*value*, ***, *timestamp=None*, *force=False*)

Set is like *put*, but is here for bluesky compatibility

Returns

st – This status object will be finished upon return in the case of basic soft Signals

Return type

Status

trigger()

Call that is used by bluesky prior to read()

4.4.2 Trigger Detector

class raypyng_bluesky.detector.**RaypyngTriggerDetector**(**args*, *rml*, *temporary_folder*, ***kwargs*)

The trigger detector is used to start the simulations. The simulations are done on the machine where bluesky is running.

get()

The readback value

put(*value*, ***, *timestamp=None*, *force=False*)

Put updates the internal readback value

The value is optionally checked first, depending on the value of force. In addition, VALUE subscriptions are run.

Extra kwargs are ignored (for API compatibility with EpicsSignal kwargs pass through).

Parameters

- **value** (*any*) – Value to set
- **timestamp** (*float, optional*) – The timestamp associated with the value, defaults to time.time()
- **metadata** (*dict, optional*) – Further associated metadata with the value (such as alarm status, severity, etc.)
- **force** (*bool, optional*) – Check the value prior to setting it, defaults to False

set(*value*, ***, *timestamp=None*, *force=False*)

Set is like *put*, but is here for bluesky compatibility

Returns
st – This status object will be finished upon return in the case of basic soft Signals

Return type
Status

trigger()

Call that is used by bluesky prior to read()

4.5 Ophyd Devices

4.5.1 MisalignComponents

class raypyng_bluesky.devices.**MisalignComponents**(**args*, *obj*, ***kwargs*)

Define the misalignment components of an optical element using SimulatedAxisMisalign

4.5.2 SimulatedPGM

class raypyng_bluesky.devices.**SimulatedPGM**(**args*, *obj*, ***kwargs*)

Define the Plan Grating Monochromator using SimulatedAxisMisalign and SimulatedAxisGrating.

Additionaly defines a two dictionaries to store different grating parameters and a method `change_grating()` to switch between them.

change_grating(*grating_name*)

Change between gratings based on the line density

Parameters
grating_name (*str*) – the name you of the grating you want to use

rename_default_grating(*new_name*)

Rename the default grating

Parameters
new_name (*str*) – the new name for the default grating

rename_grating(*new_name*, *old_name*)

Rename any grating

Parameters

- **new_name** (*str*) – the new name for the default grating
- **old_name** (*str*) – the old name of the grating

4.5.3 SimulatedApertures

class raypyng_bluesky.devices.**SimulatedApertures**(**args*, *obj*, ***kwargs*)

Define the apertures using SimulatedAxisMisalign and SimulatedAxisAperture.

4.5.4 SimulatedMirror

class raypyng_bluesky.devices.**SimulatedMirror**(**args*, *obj*, ***kwargs*)

Define the mirrors using SimulatedAxisMisalign.

4.5.5 SimulatedSource

class raypyng_bluesky.devices.**SimulatedSource**(**args*, *obj*, ***kwargs*)

Define the source using SimulatedAxisSource.

4.6 Preprocessor

4.6.1 MisalignComponents

raypyng_bluesky.preprocessor.**trigger_sim**(*plan*, *trigger_detector*)

Trigger simulations for raypyng plans

This function is composed of four steps:

1- populate_raypyng_devices_list_at_stage:
at the 'stage message' each device is classified and saved into two list. One list is dedicated to raypng devices, and one for all the others

2- prepare_simulations_at_open_run:
when the message is 'open_run', if both raypyng devices and normal devices have been staged raise an exception. Otherwise the list of exports is prepared(consists of detector names included in the plan) and passed to the trigger detector. The done simulation file is removed from the temporary folder.

3- insert_before_first_det_trigger:
before the first detector is triggered, a trigger message for the raypyng trigger detector is inserted in the same group as the other detectors

4- cleanup_at_close_run:
when the message is 'close_run' the simulation_done file is removed and the list containing the raypyng and other devices, created at point 1, are cleared.

Parameters

- **plan** (*bluesky.plan*) – the plan that is being executed
- **trigger_detector** (RaypyngTriggerDetector) – the trigger detector

Raises

- **ValueError** – if in the plan a mix of raypyng devices are other devices
- **are used raise an exeption** –

4.6.2 SupplementalDataRaypyng

class raypyng_bluesky.preprocessor.**SupplementalDataRaypyng**(**args*, *trigger_detector*, ***kwargs*)

Supplemental data for raypyng.

The Run engine is needed to be able to include the trigger detector automatically

Parameters
trigger_detector (RaypyngTriggerDetector) – The detector to trigger raypyng

INDEX

T