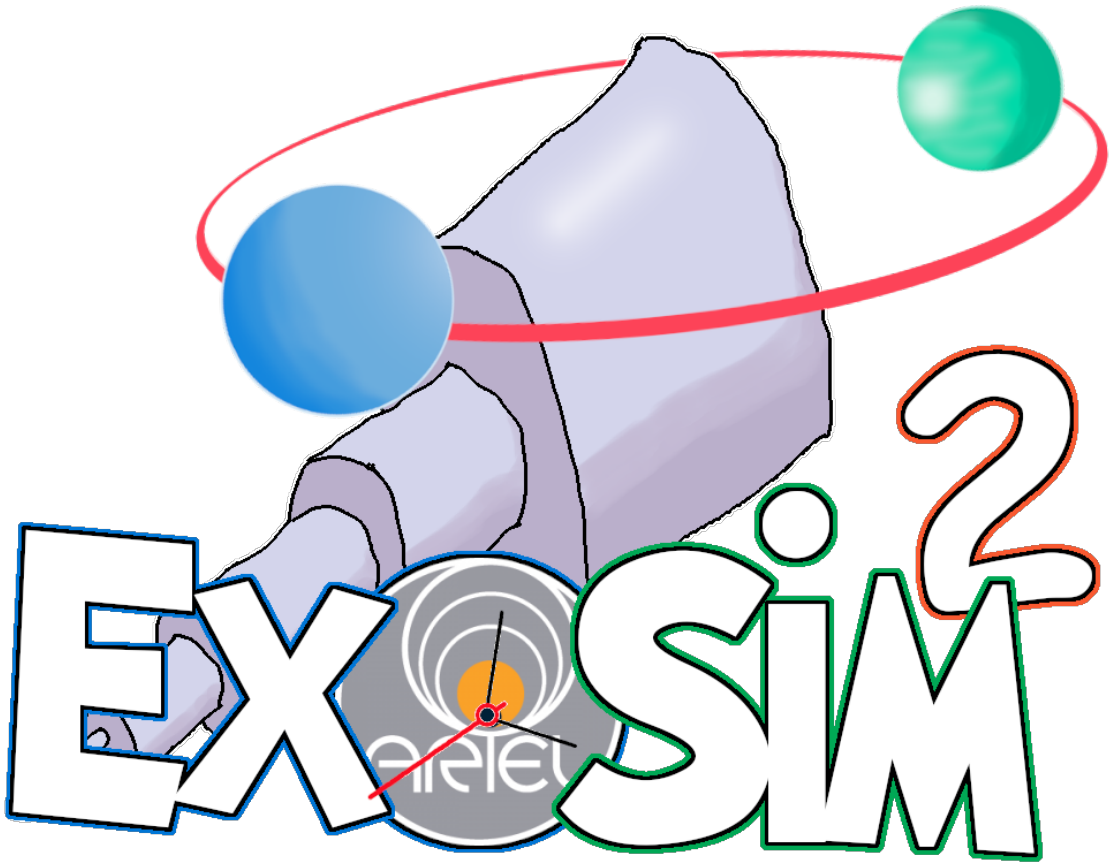


ExoSim2

Release 2.0.0rc1

L. V. Mugnai, E. Pascale, A. Bocchieri, A. Lorenzani, A. Papageo

Mar 06, 2024

CONTENTS

Version: 2.0.0rc1

ExoSim is the time domain simulator exoplanet observations first presented in Sarkar et al., 2021[1]. This refactored version is meant to be easy to use and largely customisable: almost every part of the code can be customised by the user.

This guide will walk you through the simulation steps with examples and descriptions of the simulation strategy. The guide goal is to train the user to customise the simulator according to the instrument of the observation needed.

Installation

New to ExoSim? Install ExoSim to start.

User Guide

Learn how to use ExoSim.

Developer Guide

[1] https://link.springer.com/article/10.1007/s10686-020-09690-9

Learn how to improve or customise ExoSim.

API Guide

Dig into the complete Api guide.

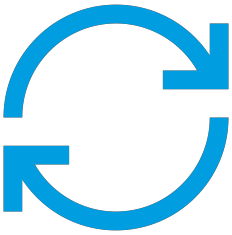
FAQs

Go To FAQs

License

Go To License

Changelog



Go To Changelog

CHAPTER

ONE

INSTALLATION & UPDATES

The following notes guide you toward the installation of *ExoSim* using a Python virtual environment. You have to have Python and *pip* ` installed already. Ask your computer administrator in case you need to install these components.

Note: The current implementation of ExoSim 2 is compatible with Python 3.8, 3.9 and 3.10.

1.1 Create the Virtual Environment

You can either create a Python Virtual Environment in your anaconda python or in a standard Python installation.

Tip: The Anaconda solution is cross-platform: the following instructions should work for Windows, iOS and Linux.

Assuming you have Anaconda[2] installed in your system, you can simply install ExoSimVE following this procedure. Open the Anaconda command shell, or if you are on a Unix system just open the console.

You can create a Virtual Environment as

```
conda create --name ExoSimVE python=3.9
```

The program will ask you if you want to install some standard packages: accept them.

You can now activate or deactivate the Virtual Environment as

```
conda activate ExoSimVE
conda deactivate
```

[2] https://www.anaconda.com/

Python venv

If you have a standard Python installation, you still can work with Virtual Environment. You have to have Python *virtualenv* ` installed. For Linux you can do that as:

```
mkdir ExoSimVE
virtualenv -p /usr/bin/python3.9 ExoSimVE
```

Then activate the virtual environment. If using csh, type

```
source ExoSimVE/bin/activate.csh
```

(check virtual environment documentation when using a different shell)

If you don't want to use a virtual environment, check *What if I don't want to create a python virtual environment?*

1.2 ExoSim package installation

1.2.1 Instal ExoSim

Install from PiPy

The ExoSim package is hosted on Pypi repository. You can install it by

```
pip install exosim
```

Install from Git

You can clone ExoSim from our main git repository

```
git clone https://github.com/arielmission-space/ExoSim2-public.git
```

Move into the ExoSim folder

```
cd /your_path/ExoSim2.0
```

Then, just do

```
pip install .
```

To test for correct setup you can do

```
python -c "import exosim"
```

If no errors appeared then it was successfully installed. Additionally the *exosim* program should now be available in the command line

```
exosim
```

1.2.2 Uninstall ExoSim

ExoSim is installed in your system as a standard python package: you can uninstall it from your Environment as

```
pip uninstall exosim
```

1.2.3 Upgrade ExoSim

Upgrade from PiPy 

If you have installed ExoSim from PyPi, now you can update the package simply as

```
pip install exosim --upgrade
```

Upgrade from Git 

If you have installed ExoSim from Git, you can download or pull a newer version of ExoSim over the old one, replacing all modified data.

Then you have to place yourself inside the installation directory with the console

```
cd /your_path/ExoSim2.0
```

Now you can update ExoSim simply as

```
pip install . --upgrade
```

or simply

```
pip install .
```

1.2.4 Modify ExoSim

You can modify ExoSim main code, editing it as you prefer, but in order to make the changes effective

```
pip install . --upgrade
```

or simply

```
pip install .
```

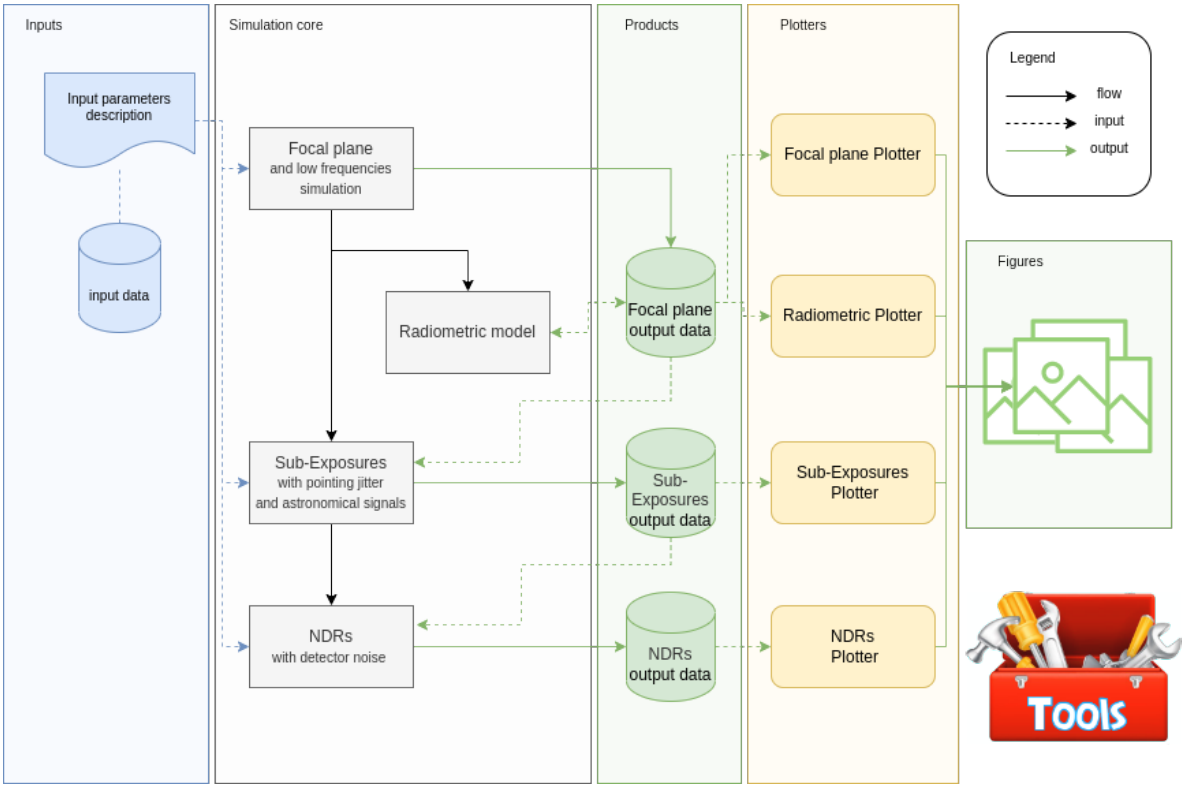
To produce new *ExoSim* functionalities and contribute to the code, please see *Contributing guidelines*.

CHAPTER

TWO

USER GUIDE

This guide will walk you through *ExoSim 2*. Here are listed the guides for all *ExoSim* pipelines and functionalities.



2.1 Quick start

In the following, we explore the main uses of the new *ExoSim* version. The code can be either used as stand-alone and run pre-made pipelines called *recipes*, or can be used as a library to build custom pipelines.

While in the next section of this documentation, we will focus on the functionalities contained in each recipe, and hence on the use of the code as a library, here we want to focus on the fast run.

2.1.1 Running ExoSim from console

After the installation, the user can run *ExoSim* from the console to verify the installed version:

```
exosim
```

To run pre-made pipelines (called *recipes'*) the user can call *exosim* from the console after the installation.

Different recipes have different commands:

command	description
`exosim-focalplane`	runs the recipe to create the low-frequency focal plane (*Focal plane creation*)
`exosim-radiometric`	runs the radiometric model recipe (*Radiometric Model*)
`exosim-sub-exposures`	create the sub-exposures from the focal plane (*Sub-Exposures*)
`exosim-nrds`	create the ndrs from the sub-exposures (*NDRS*)

Run it with the *help* flag to read the available options:

```
exosim-focalplane --help
```

Here are listed the command line main flags.

flag	description
`-c`, `--configuration`	Input payload description file
`-o`, `--output`	output file
`-P`, `--plot`	runs the associated plotter (*Plotters*)
`-n`, `--nThreads`	number of threads for parallel processing
`-d`, `--debug`	debug mode screen
`-l`, `--log`	store the log output on file

Where the configuration file shall be an *.xml* file and the output file and *.h5* file (see later in *The .h5 file*) *-n* must be followed by an integer. *-d* and *-l* do not need any argument as they simply need to be listed to activate their options.

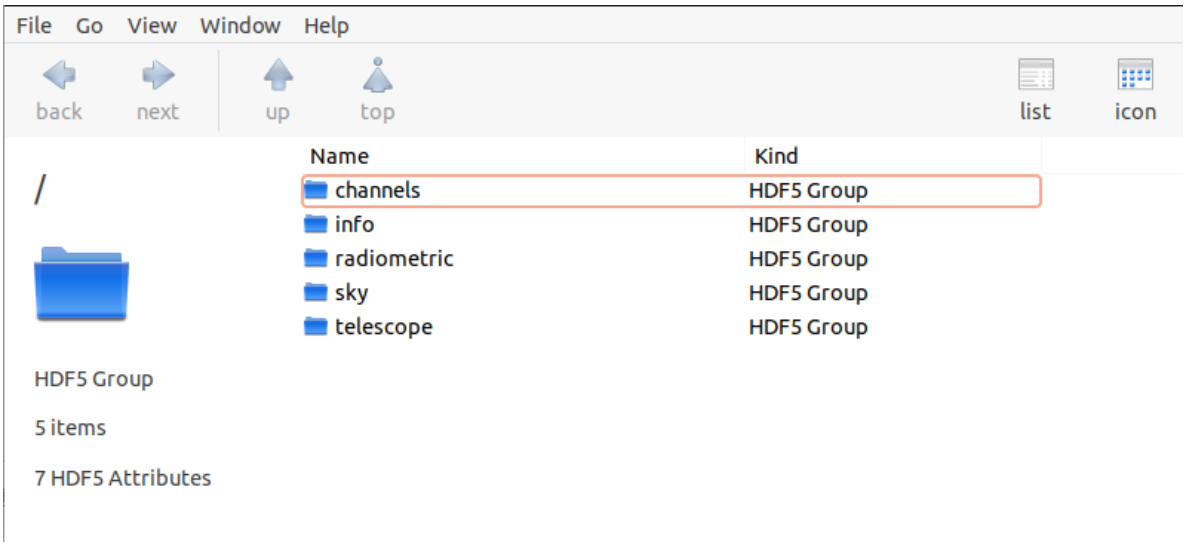
2.1.2 Understanding the outputs

It's easy.

The *.h5* file

The main output product is a HDF5[3] *.h5* file. This format has many viewers such as HDFView[4] or HDFCompass[5] and APIs such as Cpp[6], FORTRAN[7] and Python[8].

[3] https://www.hdfgroup.org/solutions/hdf5/
[4] https://www.hdfgroup.org/downloads/hdfview/
[5] https://support.hdfgroup.org/projects/compass/
[6] https://support.hdfgroup.org/HDF5/doc/cpplus_RM/index.html
[7] https://support.hdfgroup.org/HDF5/doc/fortran/index.html
[8] https://www.h5py.org/



To use the data, see *How can I load HDF5 data into my code?* in the *FAQs* section.

2.1.3 Running the examples

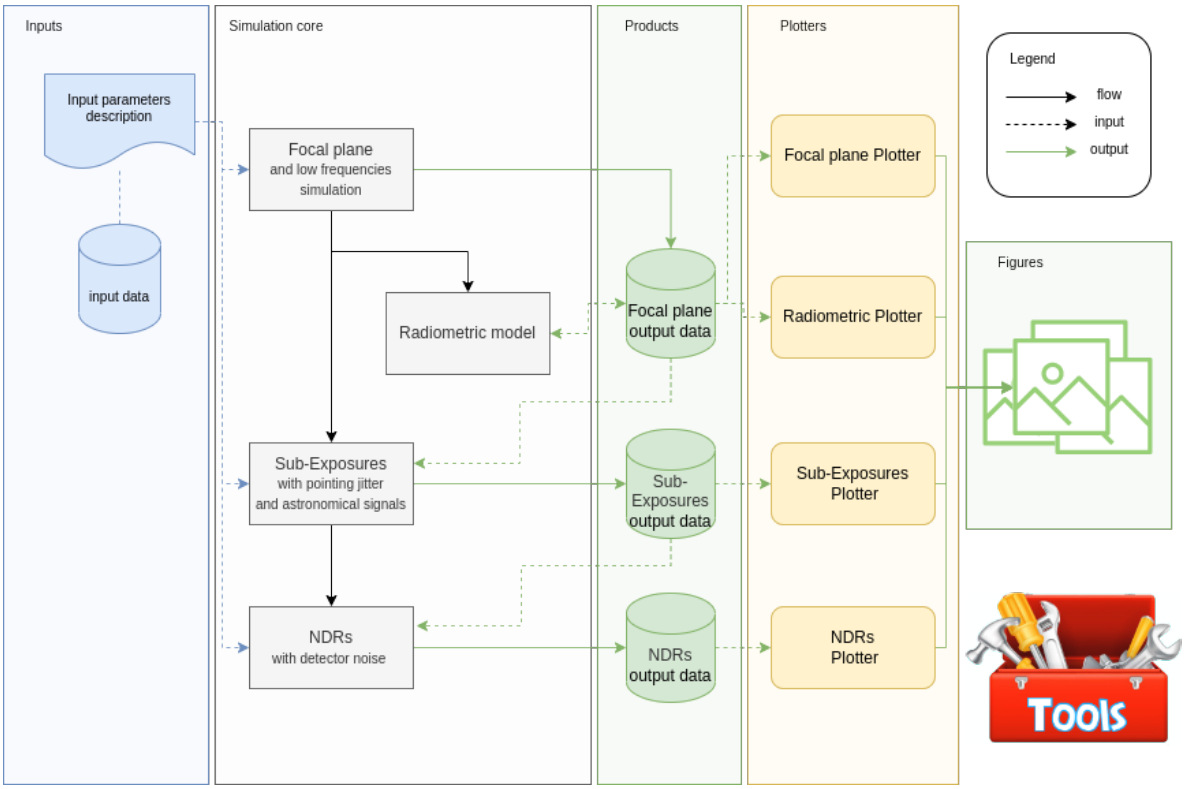
If you downloaded *ExoSim 2* from the GitHub[9] repository (see install git), you will find an *examples* folder in the root. If you installed *ExoSim 2* from Pypi (see install pip), you will have to download the folder from the GitHub[10] repository. Once you have downloaded the example folder, locate yourself there with the command console.

To run the example, you first need to change the path to the example folder in the *main_example.xml* file. Replace the path in the *main_example.xml* file with the path to the *examples* folder in your computer.

```
<ConfigPath>/path/to/ExoSim2/examples</ConfigPath>
```

Now, we will follow the Exosim diagram

[9] https://github.com/arielmission-space/ExoSim2-public
[10] https://github.com/arielmission-space/ExoSim2-public



From console

Focal plane

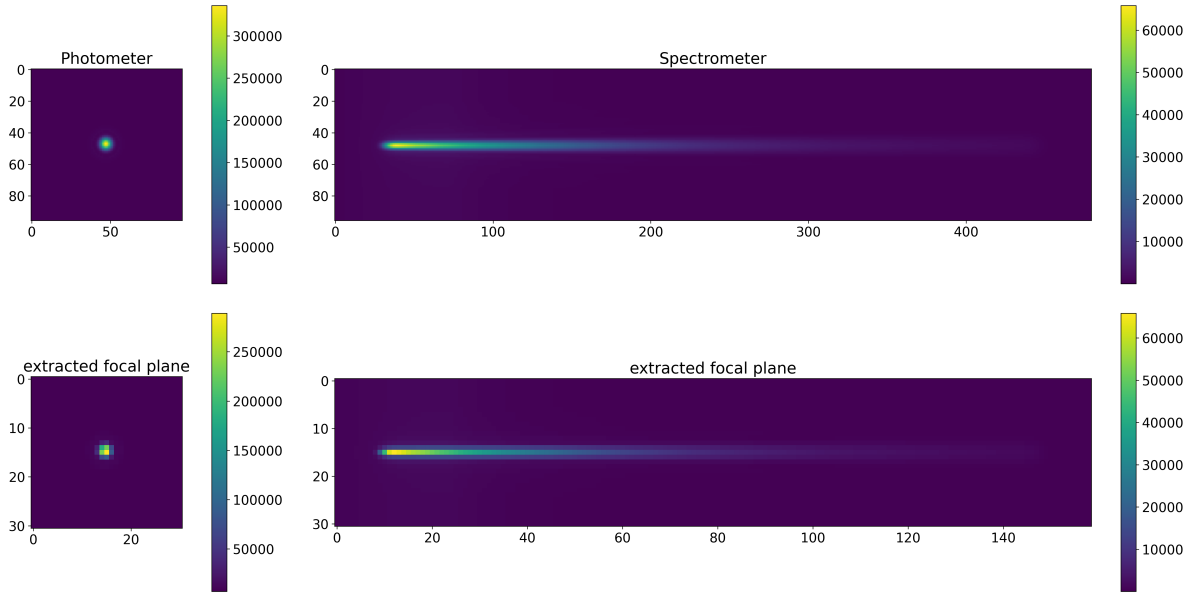
The first step is to build the focal plane (see *Focal plane creation*). You can do it by

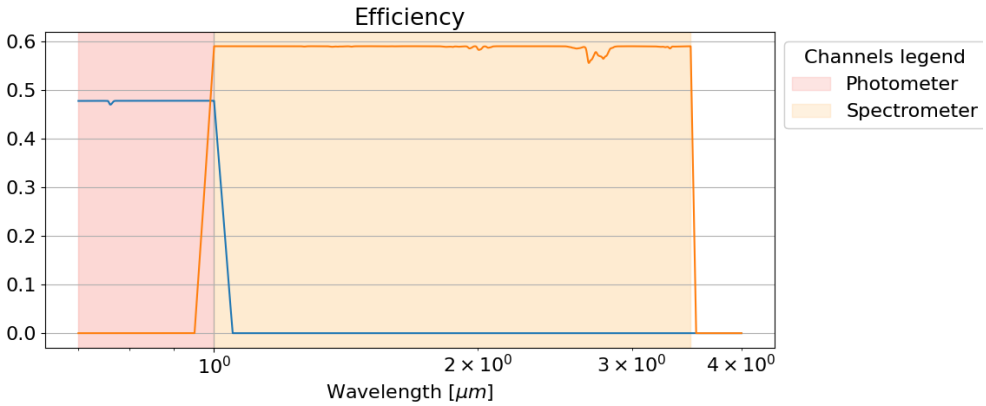
```
exosim-focalplane -c main_example.xml -o test_common.h5
```

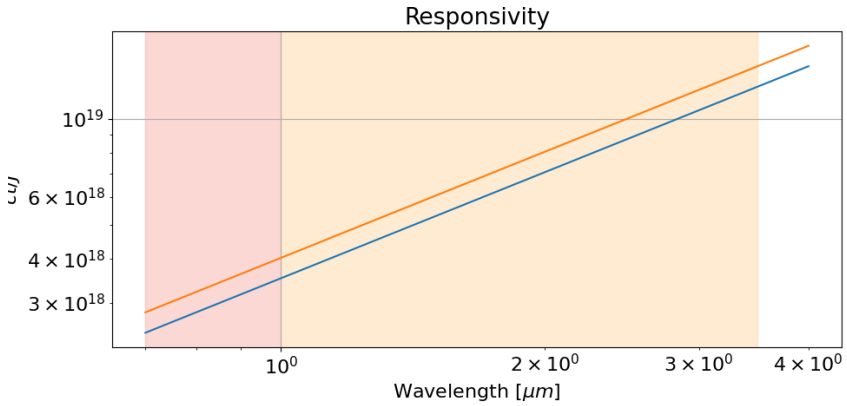
Then you will find the output file in the same folder. If you want to produce the plots, you can now run

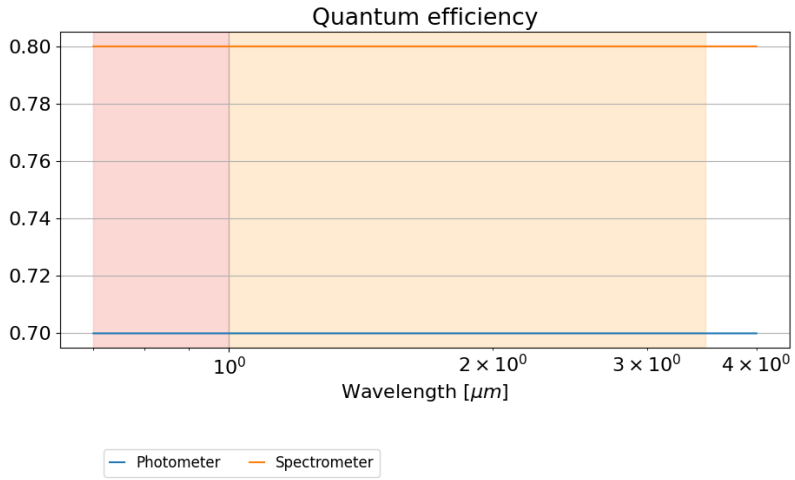
```
exosim-plot -i test_common.h5 -o plots/ --focal_plane -t 0
```

This will produce two plots: the first focal plane and the instrument efficiency vs wavelength:


Photometer
Spectrometer
extracted focal plane
extracted focal plane


Efficiency
0.6
0.5
0.4
0.3
0.2
0.1
0.0
10^0
2×10^0
3×10^0
4×10^0
Wavelength [μm]
Channels legend
Photometer
Spectrometer


Responsivity
10^{19}
6×10^{18}
4×10^{18}
3×10^{18}
10^0
2×10^0
3×10^0
4×10^0
Wavelength [μm]


Quantum efficiency
0.80
0.78
0.76
0.74
0.72
0.70
10^0
2×10^0
3×10^0
4×10^0
Wavelength [μm]
Photometer
Spectrometer

Radiometric model

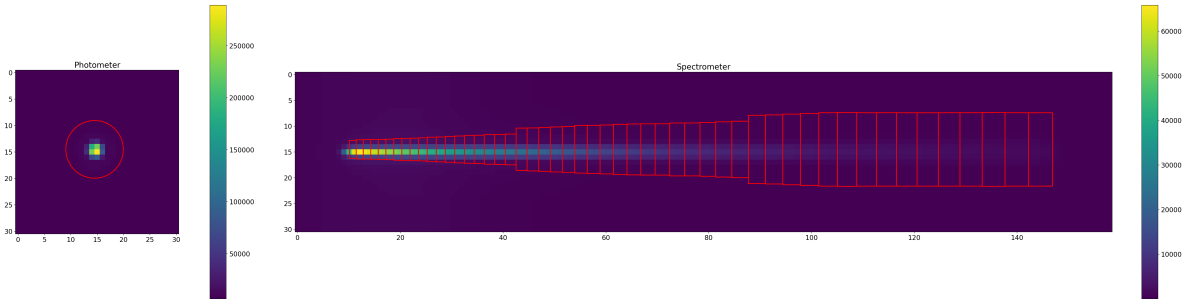
Now you can run the radiometric model on top of the produced focal plane (see *Radiometric Model*):

```
exosim-radiometric -c main_example.xml -o test_common.h5
```

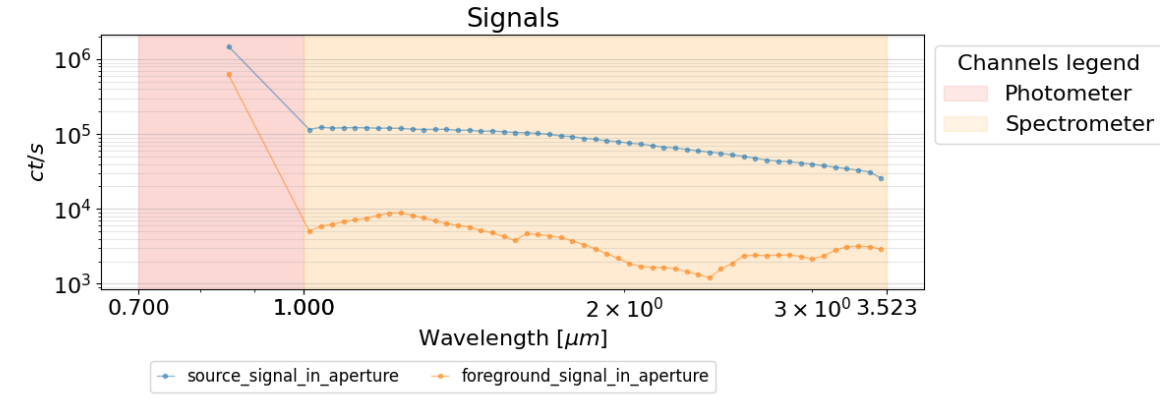
And, again, we can investigate the content by producing useful plots by

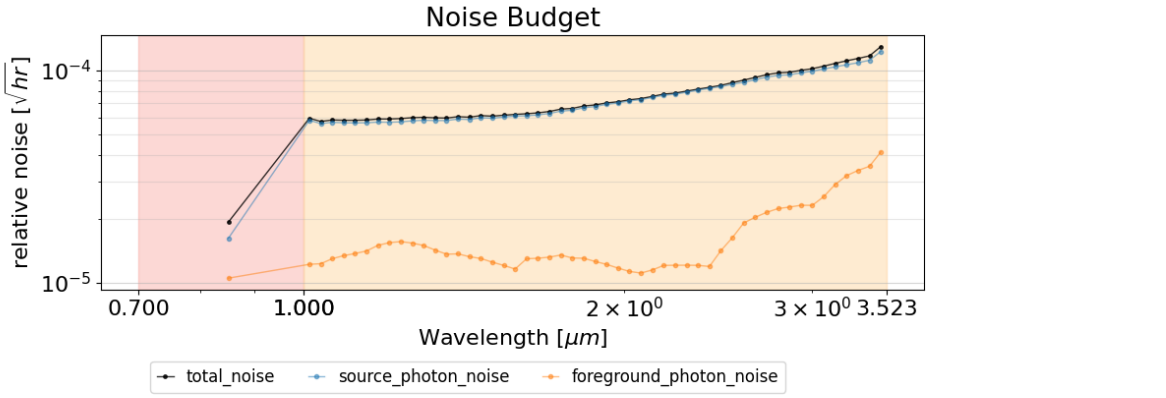
```
exosim-plot -i test_common.h5 -o plots/ --radiometric
```

This will plot the aperture used for the photometry



and the radiometric table





Sub-Exposure

As for the radiometric model, on top of the produced focal plane we can build the Sub-Exposures (see *Sub-Exposures*):

```
exosim-sub-exposures  -c main_example.xml -i test_common.h5 -o test_se.h5
```

And, again, we can use the dedicated plotter

```
exosim-plot -i test_se.h5 -o plots/ --subexposures
```

Which will produce an image of each Sub-Exposure for each channel and store it in the indicated folder

NDRs

Finally, we can build the NDRs (see *NDRS*) on top of the Sub-Exposures:

```
exosim-ndrs  -c main_example.xml -i test_se.h5 -o test_ndr.h5
```

And we can use the dedicated plotter

```
exosim-plot -i test_ndr.h5 -o plots/ --ndrs
```

Which will produce an image of each NDR for each channel and store it in the indicated folder.

From Python script

Alternatively, a Python script is included which follows the previous steps: *example_pipeline.py*.

The content of the scripts can be summarised as

```
import exosim.recipes as recipes
from exosim.plots import RadiometricPlotter, FocalPlanePlotter, \
                         SubExposuresPlotter, NDRsPlotter

# create focal plane
recipes.CreateFocalPlane('main_example.xml',
                         './test_common.h5')
# run focal plane plotter
focalPlanePlotter = FocalPlanePlotter(input='./test_common.h5')
focalPlanePlotter.plot_focal_plane(time_step=0)
focalPlanePlotter.save_fig('plots/focal_plane.png')
focalPlanePlotter.plot_efficiency()
focalPlanePlotter.save_fig('plots/efficiency.png')

# run radiometric model
recipes.RadiometricModel('main_example.xml',
                         './test_common.h5')
# run radiometric plotter
radiometricPlotter = RadiometricPlotter(input='./test_common.h5')
radiometricPlotter.plot_table(contribs=False)
radiometricPlotter.save_fig('plots/radiometric.png')
radiometricPlotter.plot_apertures()
```

(continues on next page)

(continued from previous page)

```
radiometricPlotter.save_fig('plots/apertures.png')

# create Sub-Exposures
recipes.CreateSubExposures(input_file='./test_common.h5',
                           output_file='./test_se.h5',
                           options_file='main_example.xml')
# run Sub-Exposures plotter
subExposuresPlotter = SubExposuresPlotter(input='./test_se.h5')
subExposuresPlotter.plot('plots/subexposures')

# create NDRs
recipes.CreateNDRs(input_file='./test_se.h5',
                   output_file='./test_ndr.h5',
                   options_file='main_example.xml')
# run NDRs plotter
ndrssPlotter = NDRsPlotter(input='./test_ndr.h5')
ndrssPlotter.plot('plots/ndrs')
```

From Jupyter notebook

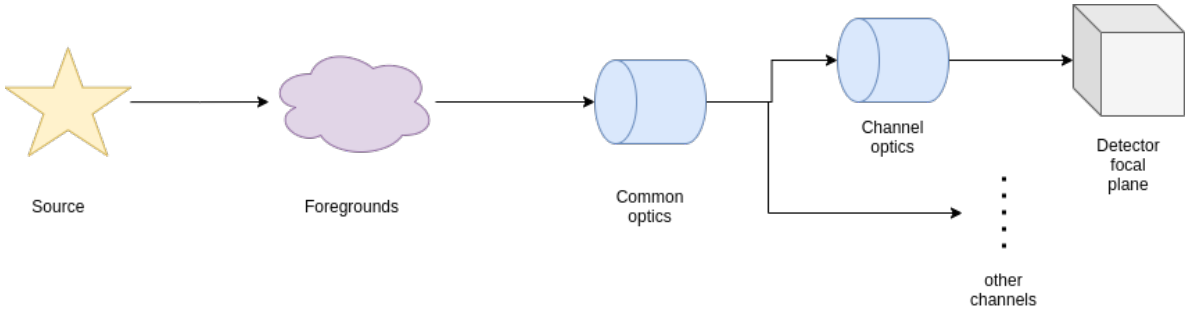
Finally, a Jupyter notebook is included containing the same scripts: *example_pipeline.ipynb*.

ExoSim Tools 

ExoSim 2 includes a list of tools useful to help the user in preparing the simulation (see *ExoSim Tools*). An example script to run the tools is included (*example_tools.py*), which refers to the tools configuration file (*tools_input_example.xml*).

2.2 Focal plane creation

The first step af an *ExoSim* simulation is the creation of the instrument focal planes. With *focal plane* here we are referring to the production of a time dependent focal plane that could take into account for low frequency time dependences. The focal plane creation is automatised by a recipe: `CreateFocalPlane`. In this section we explain each of the steps that lead to the focal plane creation.



First, we discuss the definition the light sources and their propagation

As explained in the figure above, we start from the light source. Next we explore the foreground and the common optic contributions. Then, for each channel we estimate the channel optical path and we produce the detector focal plane.

2.2.1 General settings

Here we will explore how to set up your machine for the simulation. To simulate a secific observation you need to define both the astroscene and the instrument payload. In ExoSim the user can set them both using xml file.

To start planing your simulation, you first need to setup a main configuration *.xml* file. This is an index where your settings are to be listed. This file will be parsed by `LoadOptions` into a dictionary,

Configuration path

The first thing to set is the configuration path:

```
<root>
    <ConfigPath> path/to/your/configs
        <comment>Main directory for the configuration files</comment>
    </ConfigPath>
</root>
```

This is the path that contains all the data you will need for the simulation. The *ConfigPath* content will be replaced by the parser (`LoadOptions`) in your string everytime you write `__ConfigPath__` in your *.xml*.

Wavelength grid

Then we need to set up a wavelength grid. This will be used for the production of emissions and signals from all the source in the simulation. These quantities will then be rebinned into the instrument wavelength grid when the focal plane is produced.

```
<root>
    <wl_grid>
        <wl_min unit="micron">0.45</wl_min>
        <wl_max unit="micron">10.0</wl_max>
        <logbin_resolution unit="">6000</logbin_resolution>
    </wl_grid>
</root>
```

This data will be fed into the `wl_grid` to produce the wavelength grid. The wavelength at the center of the spectral bins is defined as

$$\lambda_c = \frac{1}{2}(\lambda_j + \lambda_{j+1})$$

where λ_j is the wavelength at the bin edge defined by the recursive relation, and R is the *logbin_resolution* defined by the user.

$$\lambda_{j+1} = \lambda_j \left(1 + \frac{1}{R}\right)$$

And, given the maximum and minimum wavelengths, provided by the user, the number of bins is

$$n_{bins} = \frac{\log\left(\frac{\lambda_{max}}{\lambda_{min}}\right)}{\log\left(1 + \frac{1}{R}\right)} + 1$$

The python code to parse the wavelength grid will be:

```
import exosim.tasks.load as load
import exosim.utils as utils

loadOption = load.LoadOptions()
mainConfig = loadOption(filename='your_config_file.xml')

wl_grid = utils.grids.wl_grid(mainConfig['wl_grid']['wl_min'],
                              mainConfig['wl_grid']['wl_max'],
                              mainConfig['wl_grid']['logbin_resolution'])
```

Temporal grid

Now we need to set the temporal grid.

```
<root>
    <time_grid>
        <start_time unit="hour">0.0</start_time>
        <end_time unit="hour">10.0</end_time>
        <low_frequencies_resolution unit="second">60.0</low_frequencies_resolution>
    </time_grid>
</root>
```

This is going to be the focal plane temporal grid and should only use for low frequencies variation. For high frequency dependency a dedicated pipeline will be discussed later. This data will be fed into the *time_grid* to produce an equally sampled grid.

```
import exosim.tasks.load as load
import exosim.utils as utils

loadOption = load.LoadOptions()
mainConfig = loadOption(filename='your_config_file.xml')

time_grid = utils.grids.time_grid(mainConfig['time_grid']['start_time'],
                                  mainConfig['time_grid']['end_time'],
                                  mainConfig['time_grid']['low_frequencies_
↪resolution'])
```

If no time details are provided a single time step is assumed.

Sky and payload

Then we can describe the astroscene and the instrument payload by filling the keywords:

```
<root>
    <sky>
        <config>__ConfigPath__/sky_example.xml</config>
    </sky>

    <payload>
        <config>__ConfigPath__/payload_example.xml</config>
```

(continues on next page)

(continued from previous page)

```
    </payload>
</root>
```

In this example we use two different *.xml* files to describe the sky and the payload. We make use of the `__ConfigPath__` to point to file contained in the directory mentioned above. The *config* keyword tells to the parser (*LoadOptions*) to look for another *.xml* file.

The *sky* root contains all the information about the light sources and the sky foregrounds. The *payload* root contains the description of the instrument.

In particular, the *payload* root can contain both the common part of the instrument and the channel dedicated parts. In the following example, the payload contains a common optics path, which is the telescope, and two separated channels. Each of these part is described in a dedicated *.xml* configuration file.

```
<root>
    <Telescope> Common optics
        <config>__ConfigPath__/telescope.xml</config>
    </Telescope>

    <channel> channel 1
        <config>__ConfigPath__/channel_1.xml</config>
    </channel>
    <channel> channel 2
        <config>__ConfigPath__/channel_2.xml</config>
    </channel>
</root>
```

Preparing output

ExoSim can store all its product into an output file. At the moment of writing only *.hdf5* file are supported as output.

To prepare the output the following script can be used:

```
from exosim.output import SetOutput

output = SetOutput('output_file.h5')
```

This will set *output_file.h5* as the output file. To use the file the method *use* can be use as it return an *Output* class:

```
with output.use(append=True, cache=True) as out:
    ...
```

With the file in use, to produce sub-folders in the file the user can use.

```
out_group = out.create_group('group name')
```

For other functionalities refer to the *Output* class.

2.2.2 Sources

Inputs: describe the source

The target source must be described in an *.xml* file under *sky*, using the keyword *source*.

In the following example we simulate HD 209458:

```
<source> HD 209458
<source>
```

This file will be parsed by *LoadOptions* into a dictionary, and the star name is stored under the keyword *value*.

```
loadOptions = LoadOptions()
options = loadOptions(filename = 'path/to/file.xml')
options['value'] = 'HD 209458'
```

The dictionary is then feeded to the *ParseSource* task, that returs the source *Sed*.

ExoSim supports three different sources type:

- *Planck star*
- *Phoenix star*
- *Custom star*

The source type is to be indicated as:

```
<source> HD 209458
    <source_type> planck </source_type>
</source>
```

According to the indicated type, *ParseSource* will call a different *Task*.

Planck star

If *planck* star is used, then other information are needed to simulate the source:

```
<source> HD 209458
    <source_type> planck </source_type>
    <R unit="R_sun"> 1.18 </R>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 47 </D>
</source>
```

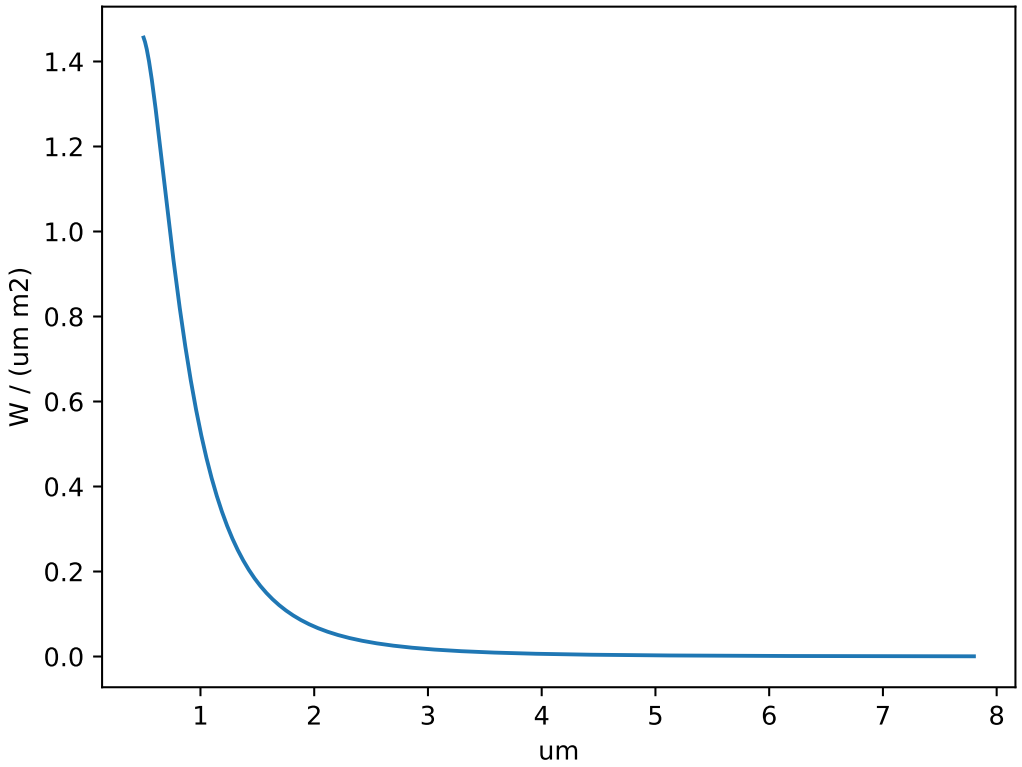
The planck star sed is created by *CreatePlanckStar*:

The star emission is simulated by `astropy.modeling.physical_models.BlackBody`[11]. The resulting sed is then converted into $W/m^2/sr/\mu m$ and scaled by the solid angle $\pi \left(\frac{R}{D}\right)^2$.

Here we report and example:

[11] https://docs.astropy.org/en/latest/api/astropy.modeling.physical_models.BlackBody.html#astropy.modeling.physical_models.BlackBody

```
from exosim.tasks.sed import CreatePlanckStar
import astropy.units as u
import numpy as np
createPlanckStar = CreatePlanckStar()
wl = np.linspace(0.5, 7.8, 10000) * u.um
T = 6086 * u.K
R = 1.18 * u.R_sun
D = 47 * u.au
sed = createPlanckStar(wavelength=wl, T=T, R=R, D=D)

import matplotlib.pyplot as plt
plt.plot(sed.spectral, sed.data[0,0])
plt.ylabel(sed.data_units)
plt.xlabel(sed.spectral_units)
plt.show()
```



Phoenix star

If *phoenix* is indicated, then *ExoSim* uses the Phoenic spectra to simulate the source. In this case we can either point to a specific Phoenix file using the *filename* keyword:

```
<source> HD 209458
    <source_type>phoenix </source_type>
    <filename> phoenix_filename </filename>

    <R unit="R_sun"> 1.18 </R>
    <D unit="pc"> 47 </D>
</source>
```

or we can point *ExoSim* to a path containing all the Phoenix spectra and provide it with all the information to select the best spectra to use:

```
<source> HD 209458
    <source_type>phoenix </source_type>
    <path> phoenix_path </path>

    <R unit="R_sun"> 1.18 </R>
    <M unit="M_sun"> 1.17 </M>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 47 </D>
    <z unit=""> 0.0 </z>
</source>
```

The Phoenix star sed is created by *LoadPhoenix*:

Custom star

If *custom* is indicated, then *ExoSim* will either look for a custom *Task* (see *Custom Tasks*), if *source_task* is present in the configuration file, or by default it uses *LoadCustom*. The *Task* loads a custom SED from a file and scaled it by the solid angle $\pi \left(\frac{R}{D}\right)^2$.

The default *LoadCustom* needs a filename containing the *Sed* to use.

```
<source> HD 209458
    <source_type>custom </source_type>
    <filename> custom_sed_filename </filename>

    <R unit="R_sun"> 1.18 </R>
    <D unit="pc"> 47 </D>
</source>
```

The custom sed file must be a *.ecsv* file with two columns: *Wavelength* and *Sed*.

Note: Depending on the computing power available, the user can decide to use a different number of wavelength and temporal points to simulate the source, incrementing the simulation accuracy.

Load star parameters from online databases

ExoSim can load star parameters from online databases. At the moment only exodb[12] is supported.

In this case, instead of the stellar parameter, the online database must be indicated:

```
<source> HD 209458
    <source_type>phoenix </source_type>
    <path>/usr/local/project_data/sed </path>

    <online_database>
        <url>https://exodb.space/api/v1/star</url>
        <x-access-tokens> your_token_here </x-access-tokens>
    </online_database>

</source>
```

Create your own source

Otherwise, in *Exosim* you can create your own source by using a customizable *Task*. To learn more about customizing tasks, please refer to *Custom Tasks*. To create a custom source, use *CreateCustomSource*.

As an example, we report here the default *CreateCustomSource* task. To enable it, write the following in your xml file:

```
<source> HD 209458
    <source_task> CreateCustomSource </source_task>
    <R unit="R_sun"> 1.17967 </R>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 47.4567 </D>
    <wl_min unit="um">0.5</wl_min>
    <wl_max unit="um">8</wl_max>
    <n_points >1000</n_points>
</source>
```

The *source_task* keyword will guide the code to the *Task* to use. In this case is the default tasks. If you write your own version, please write there the file containing your script. The default *CreateCustomSource* task will simply create a planck star using the input parameters.

Outputs: prepare the sources

Single source

As mentioned, the *.xml* file parsed by *LoadOptions*, for the planck case it will return a dictionary similar to

```
source_in = {
        'value': 'HD 209458',
        'source_type': 'planck',
        'R': 1.18 * u.R_sun,
        'D': 47 * u.pc,
```

(continues on next page)

[12] https://exodb.space/

(continued from previous page)

```
        'T': 6086 * u.K,
        }
```

The wavelength grid to use is provided by the *Wavelength grid*.

Then, we can use *ParseSource* task to produce the *Sed*. The result will be a dictionary with the star name as keys and *Sed* as key content.

```
from exosim.tasks.parse import ParseSource
import astropy.units as u
import numpy as np
parseSource = ParseSource()
wl = np.linspace(0.5, 7.8, 10000) * u.um
tt = np.linspace(0.5, 1, 10) * u.hr

source_out = parseSource(parameters=source_in,
                         wavelength=wl,
                         time=tt)

import matplotlib.pyplot as plt

plt.plot(source_out['HD 209458'].spectral, source_out['HD 209458'].data[0,0])
plt.ylabel(source_out['HD 209458'].data_units)
plt.xlabel(source_out['HD 209458'].spectral_units)
plt.show()
```

More sources

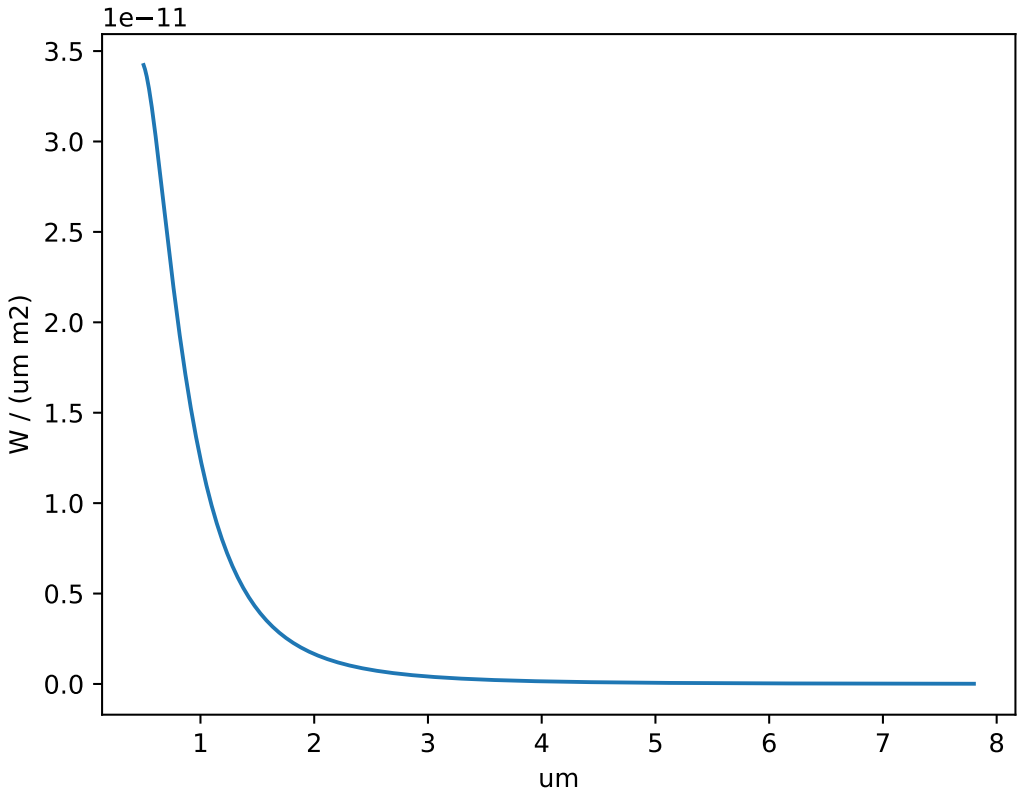
If more sources are listed, the xml file will look like this:

```
<source> HD 209458
    <source_type> planck </source_type>
    <R unit="R_sun"> 1.18 </R>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 47 </D>
</source>

<source> GJ 1214
    <source_type> planck </source_type>
    <R unit="R_sun"> 0.218 </R>
    <T unit="K"> 3026 </T>
    <D unit="pc"> 13 </D>
</source>
```

Then, the parsed dictionary will be:

```
from collections import OrderedDict
sources_in = OrderedDict({'HD 209458': {'value': 'HD 209458',
                                        'source_type': 'planck',
                                        'R': 1.18 * u.R_sun,
                                        'D': 47 * u.pc,
                                        'T': 6086 * u.K,
```

(continues on next page)


1e−11
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
W / (um m2)
1
2
3
4
5
6
7
8
um

(continued from previous page)

```
                                },
                            'GJ 1214': {'value': 'GJ 1214',
                                    'source_type': 'planck',
                                    'R': 0.218 * u.R_sun,
                                    'D': 13 * u.pc,
                                    'T': 3026 * u.K,
                                    },})
```

And this dictionary is fed into *ParseSources* to produce the following *Sed*:

```
import astropy.units as u
import numpy as np
from exosim.tasks.parse import ParseSources

wl = np.linspace(0.5, 7.8, 10000) * u.um
tt = np.linspace(0.5, 1, 10) * u.hr

parseSources = ParseSources()
sources_out = parseSources(parameters=sources_in,
                           wavelength=wl,
                           time=tt)

import matplotlib.pyplot as plt

for key in sources_out.keys():
    plt.plot(sources_out[key].spectral, sources_out[key].data[0, 0], label=key)
plt.ylabel(sources_out[key].data_units)
plt.xlabel(sources_out[key].spectral_units)
plt.legend()
plt.show()
```

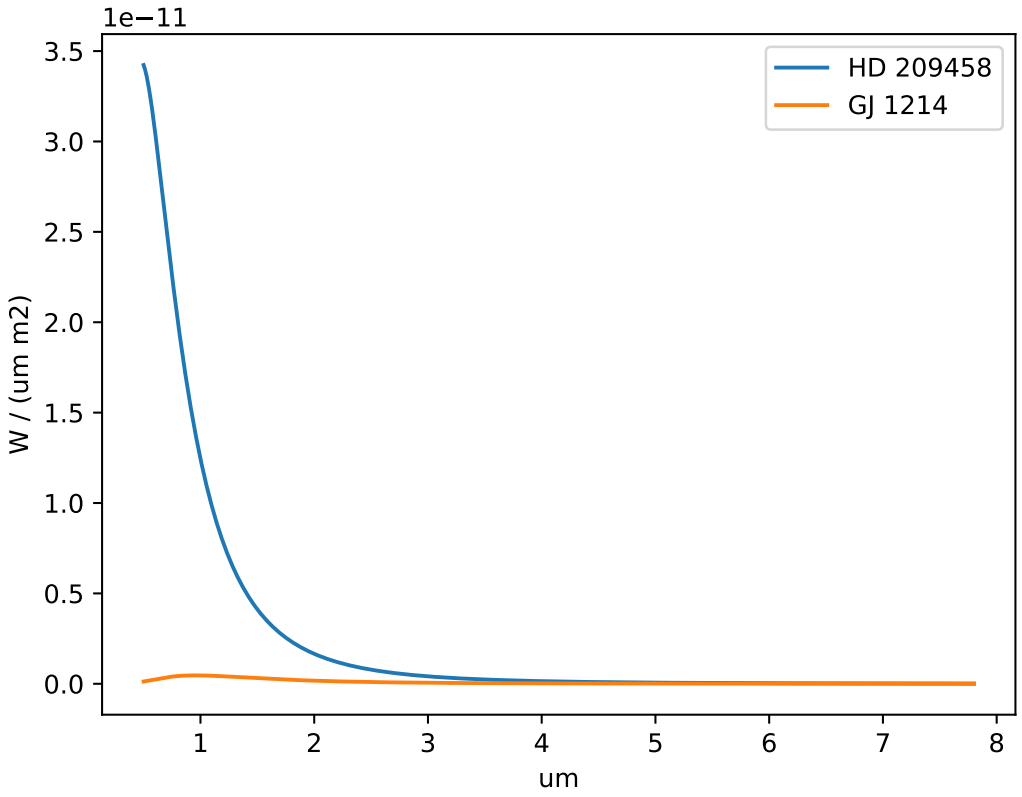
Note: In this example the sources are superimposed. If the sources have different position in the sky, see *Telescope pointing and multiple sources*. In that section is explained how to simulate multiple sources and the telescope pointing.

Parse from xml

Assuming the wavelength and temporal grids have already produced as described in *Wavelength grid* and *Temporal grid*, you can parse the configuration file to produce a dictionary of sources as

```
import exosim.tasks.parse as parse

with output.use(append=True, cache=True) as out:

    out_sky = out.create_group('sky')

    parseSources = parse.ParseSources()
    sources = parseSources(parameters=mainConfig['sky']['source'],
                           wavelength=wl_grid,
```

(continues on next page)


1e−11
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
W / (um m2)
1
2
3
4
5
6
7
8
um
HD 209458
GJ 1214

(continued from previous page)

```
                                time=time_grid,
                                output=out_sky)
```

Here we also assumed that the user selected an output file (as described in *Preparing output*) and wants to store the products in a dedicated subfolder.

2.2.3 Foregrounds

Defining the foregrounds

The foregrounds are to be listed in the *sky .xml* file, along with the *source*, but under the keyword *foregrounds*.

```
<foregrounds>
</foregrounds>
```

Foregrounds are parsed as optical elements, like the optics in the payload, by `ParseOpticalElement`. More foregrounds make an optical path, and therefore are parsed by `ParsePath`.

```
<foregrounds>
    <opticalElement> first_foreground_name
    </opticalElement>

    <opticalElement> second_foreground_name
    </opticalElement>
</foregrounds>
```

By default,`ExoSim`support user defined foregrounds and zodiacal foreground.

User defined foreground

An example of user defined foreground is reported in the package *examples* directory:

```
<foregrounds>
    <opticalElement> earthsky
        <task_model>LoadOpticalElement</task_model>
        <datafile>__ConfigPath__/foreground_file.ecsv</datafile>
        <wavelength_key>Wavelength</wavelength_key>
        <radiance_key>Radiance</radiance_key>
        <efficiency_key>Transmission</efficiency_key>
    </opticalElement>
</foregrounds>
```

In this case the *ExoSim* finds a foreground called *earthsky*, and uses the `Task` indicated in *task_model* to load it. The indicated `LoadOpticalElement` is the default `Task` included in *ExoSim* to load and optical element. An optical element is defined by it radiance and efficiency in function of the wavelength. Hence, this default class looks into the indicated *datafile* to load of the three quantities. The data file should contain a table such that the three quantities are identified by the keys reported in the *.xml* description. In this case the wavelength is reported under a column called *Wavelength*, the radiance under the column *Radiance* and the efficiency under the column *Transmission*.

The user can write a custom `Task` to load or estimate the foreground differently. This can be done by writing a new class inheriting from the default `LoadOpticalElement` and indicating in the *task_model* key the python

file containing such new class. The user shall only overwrite the *model* method in the new class. The output of a custom model method, as indicated in the `LoadOpticalElement` documentation, shall be a `Radiance` and a `Dimensionless`. The first containing the foreground radiance, and the second the foreground transmission. The two classes should be binned to the general *Wavelength grid* and *Temporal grid*. Notice that the binning can be handled by the `spectral_rebin` and `temporal_rebin` methods of the `Signal` class. To learn more about customizing tasks, please refer to *Custom Tasks*.

Caution: If the user doesn't include the *task_model* keyword in the optical element description, the default `LoadOpticalElement` task is used.

Zodiacal Foreground

If the foreground name is *zodi* or *zodiacal* the code will parse the element using `EstimateZodi` instead of `ParseOpticalElement`.

The zodiacal foreground radiance is estimate using a modified version of the JWST-MIRI Zodiacal model (Glasse et al., 2010), scaled according to the target position in the sky and the Zodi model of Kelsall et al. (1998):

$$I_{zodi}(\lambda) = A\left(3.5\cdot 10^{-14} BB(\lambda, 5500\,K) + 3.52\cdot 10^{-8} BB(\lambda, 270\,K)\right)$$

where $BB(\lambda, T)$ is the Planck black body law and A is the fitted coefficient.

The user can either specify the coefficient to use, as in the example:

```
<foregrounds>
    <opticalElement> zodiacal
        <zodiacal_factor>2.5</zodiacal_factor>
    </opticalElement>
</foregrounds>
```

or can specify the coordinates in rad and dec:

```
<foregrounds>
    <opticalElement> zodiacal
        <coordinates> (ra, dec) </coordinates>
    </opticalElement>
</foregrounds>
```

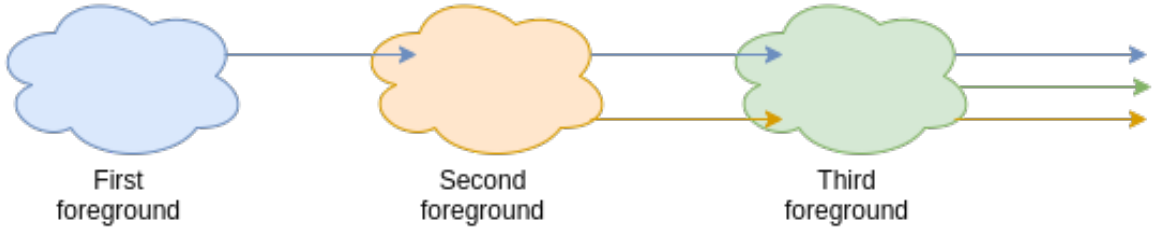
In this case the A coefficient is selected by a precompiled grid. The grid has been estimated by fitting our model with Kelsall et al. (1998) data. A custom map can be provided, to replace the default one, as long as it matches the format, by adding the keyword *zodi_map*.

Foregrounds propagation

Each parsed foreground contains a radiance in units of $W/m^2/\mu m/sr$, which is contained in a `Radiance` class, and a transmission, which is contained in a `Dimensionless` class. Both classes are children of the `Signal` class.

If more than a foreground is listed, the `ParsePath` class orders them in the same order used by the user in the *.xml* file and it propagates their light top to bottom. So the first element radiance is multiply by the second element transmission, then the second element radiance is summed. The obtained radiance is then multiply by the third element transmission and the third element radiance is summed to the result. The final transmission is the product of all the transmission. At the end of the pipeline we have a resulting radiance, still expressed

in units of $W/m^2/\mu m/sr$, and still contained in a `Radiance` class, which is the resulting radiance at the end of the chain, and a transmission that is the equivalent transmission of al the chain. This can be considered as a foreground equivalent to the full foregrounds chain.



The problem can be express with the recursive equation

$$I_{for,i+1} = I_{for,i+1} + I_{for,i} \cdot \Phi_{for,i+1}$$

$$\Phi_{for,i+1} = \Phi_{for,i+1} \cdot \Phi_{for,i}$$

Where $I_{for,i}$ is the radiance of i foreground and $\Phi_{for,i}$ is its transmission.

Note: Because of the way the light path is parsed. It's important to be careful of the order of writing for the optical element. Optical elements further from the detector should be write first in the *.xml* file.

Following the process presented in *Parse from xml*, we can parse the foregrounds as

```
import exosim.tasks.parse as parse

with output.use(append=True, cache=True) as out:

    out_sky = out.create_group('sky')

    parsePath = parse.ParsePath()
    for_contrib = parsePath(parameters=mainConfig['sky']['foregrounds'],
                            wavelength=wl_grid, time=time_grid,
                            output=out_sky,
                            group_name='foregrounds')
```

In this case, thanks to the *group_name* keyword, the contribution are saved in a dedicated folder called *foregrounds*.

The *for_contrib* element shall be propagated now through the telescope. *ExoSim2* handles this as the first optical element of the telescope optical chain.

2.2.4 Optical path

After the *Foregrounds*, comes the optical path. *ExoSim* considers two different paths: the common optics path and the channel optical path.

Both the path are described under the *payload* keyword in the configuration file, as mentioned in *Sky and payload*.

Optical elements

As the *Foregrounds*, each piece of the optical chain is parsed as an optical element by `ParseOpticalElement`. More optical elements make an optical path, and therefore are parsed by `ParsePath`.

Each optical element in the chain should be clearly defined in the *.xml* configuration file under the *optical_path* keyword.

```
<optical_path>
    <opticalElement> first_optical_element
    </opticalElement>

    <opticalElement> second_optical_element
    </opticalElement>
</optical_path>
```

Each parsed optical element contains a radiance in units of $W/m^2/\mu m/sr$, which is contained in a `Radiance` class, and a transmission, which is contained in a `Dimensionless` class. Both classes are children of the `Signal` class.

Note: *ExoSim 2* does not include an optic simulator. The optical path here is only used to estimate the system transmission and the instrument self emission. Any other effect on the performance due to the optical path will be provided to the system in the for of PSF.

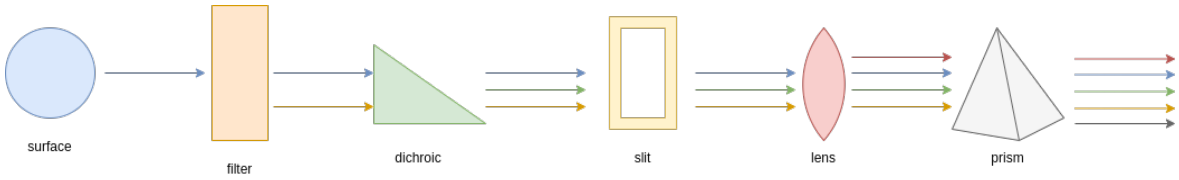
As already discussed in *User defined foreground*, `LoadOpticalElement` is the default `Task` included in *ExoSim* to load and optical element. An optical element is defined by it a radiance, that shall be expressed in units of $W/m^2/\mu m/sr$ and contained in a `Radiance` class, and an efficiency, contained in a `Dimensionless` class, both expressed in function of the wavelength. Hence, this default class looks into the indicated *datafile* to load of the three quantities. The data file should contain a table such that the three quantities are identified by the keys reported in the *.xml* description. In this case the wavelength is reported under a column called *Wavelength*, the radiance under the column *Radiance* and the efficiency under the column *Transmission*.

We recall here, that the user can write a custom `Task` to load or estimate the optical element differently. Each optical element can have its own dedicated class. This can be done by writing a new class inheriting from the default `LoadOpticalElement` and indicating in the *task_model* key the python file containing such new class. The user shall only overwrite the *model* method in the new class. The output of a custom model method, as indicated in the `LoadOpticalElement` documentation, shall be a `Radiance` and a `Dimensionless`. The first containing the optical element radiance, and the second the optical element transmission. The two classes should be binned to the general *Wavelength grid* and *Temporal grid*. Notice that the binning can be handled by the `spectral_rebin` and `temporal_rebin` methods of the `Signal` class. To learn more about customizing tasks, please refer to *Custom Tasks*.

Caution: If the user doesn't include the *task_model* keyword in the optical element description, the default `LoadOpticalElement` task is used.

Supported optical elements

The optical element type is indicated in the keyword *type*. The supported types are listed in the following image and discussed in the following.



Surface & filter

By default optical elements labelled as surfaces or filters are parsed by `LoadOpticalElement` to estimate the radiance and transmission. While the transmission can be simply read by the indicated data file, the radiance can either be provided by the user in the same data file, or it can be estimated by the code. In the second case, the user can provide an emissivity column in the data file and a temperature. Then the resulting radiance will be estimated as

$$I_{surf}(\lambda) = \epsilon(\lambda) \cdot BB(\lambda, T)$$

where ϵ is the indicated emissivity and $BB(\lambda, T)$ is the Planck black body law.

```
<optical_path>
    <opticalElement> mirror
        <type>surface</type>
        <task_model>LoadOpticalElement</task_model>
        <temperature unit='K'>60</temperature>
        <datafile>__ConfigPath__/mirror.ecsv</datafile>
        <wavelength_key>wavelength</wavelength_key>
        <emissivity_key>emissivity</emissivity_key>
        <efficiency_key>reflectivity</efficiency_key>
    </opticalElement>

    <opticalElement> filter
        <type>filter</type>
        <task_model>LoadOpticalElement</task_model>
        <temperature unit='K'>60</temperature>
        <datafile>__ConfigPath__/filter.ecsv</datafile>
        <wavelength_key>wavelength</wavelength_key>
        <emissivity_key>emissivity</emissivity_key>
        <efficiency_key>transmission</efficiency_key>
    </opticalElement>
</optical_path>
```

In *ExoSim* surface, filter, dichroic, lens and prisms are handled in the same way by default. However, because dichroics are used as beam splitter, they may compare multiple times in the payload description. In this case, the user should be carefully in indicating the right efficiency data column (transmission or reflectivity), according to the optical path branch the element is located in.

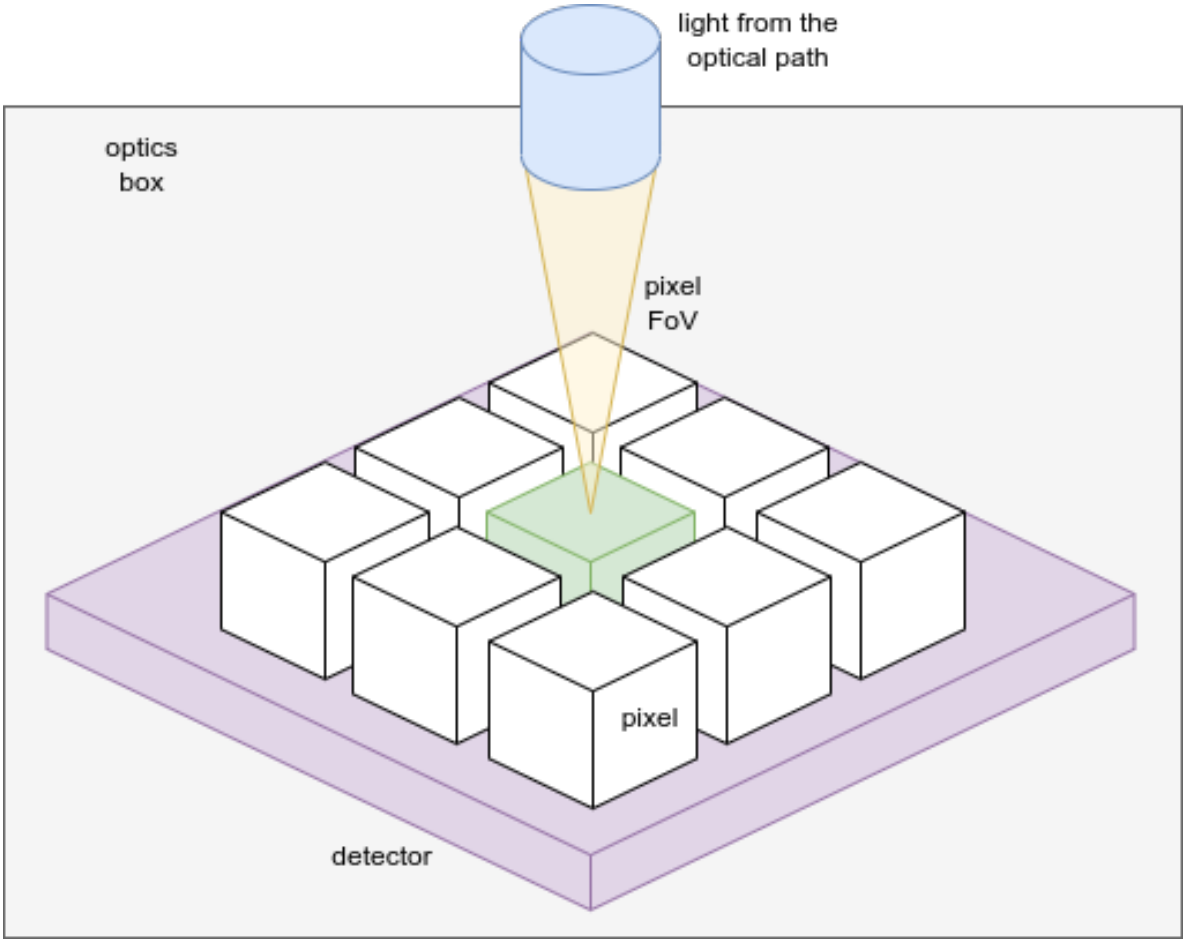
slit

ExoSim allow the introduction of slits into the payload configuration, to be used as field stops. The user must indicate the slit size on the focal plane in physical units.

```
<optical_path>
    <opticalElement> slit
        <type>slit</type>
        <width unit="mm">1.5</width>
</optical_path>
```

optics box & detector box

Other elements supported by *ExoSim* are the optics box and the detector box. In these cases the data file, here indicated as *black_box.ecsv* include emissivity and tranmission both set to 1 for every wavelegth. This are the box containing the optics and the detector and their light reach each pixel in the detector from a solid angle equal to $\pi - \Omega_{pix}$ for the optics box and π for the light coming from the detector box.



The image summarize the problem. The green detector is illuminated by the yellow cone from the optical path. The optics box, represented in gray, irradiates it from the top, except from the yellow cone, and hence $\pi - \Omega_{pix}$. The detector box, in purple, irradiates the pixel from the back: π.

```
<channel> channel_name
    <optical_path>
        <opticalElement>enclosure
            <type>optics box</type>
            <task_model>LoadOpticalElement</task_model>
            <temperature unit='K'>55</temperature>
            <datafile>__ConfigPath__/black_box.ecsv</datafile>
            <wavelength_key>wavelength</wavelength_key>
            <emissivity_key>emissivity</emissivity_key>
            <efficiency_key>transmission</efficiency_key>
            <solid_angle>pi-omega_pix</solid_angle>
        </opticalElement>

       <opticalElement>detector
            <type>detector box</type>
            <task_model>LoadOpticalElement</task_model>
            <temperature unit='K'>42</temperature>
            <datafile>__ConfigPath__/black_box.ecsv</datafile>
            <wavelength_key>wavelength</wavelength_key>
            <emissivity_key>emissivity</emissivity_key>
            <efficiency_key>transmission_eol</efficiency_key>
            <solid_angle>pi</solid_angle>
        </opticalElement>
    </optical_path>
</channel>
```

Custom solid angles can be indicated with steradians units:

```
<channel> channel_name
    <optical_path>
        <opticalElement>enclosure
            <type>optics box</type>
            <task_model>LoadOpticalElement</task_model>
            <temperature unit='K'>55</temperature>
            <datafile>__ConfigPath__/black_box.ecsv</datafile>
            <wavelength_key>wavelength</wavelength_key>
            <emissivity_key>emissivity</emissivity_key>
            <efficiency_key>transmission</efficiency_key>
            <solid_angle unit='sr'> 3.14 </solid_angle>
        </opticalElement>
</channel>
```

Parsing the path

If more optical elements are listed, the *ParsePath* class orders them in the same order used by the user in the *.xml* file and it propagates their light top to bottom. So the first element radiance is multiply by the second element transmission, then the second element radiance is summed. The obtained radiance is then multiply by the third element transmission and the third element radiance is summed to the result. The process is summarised in the previous figure. The final transmission is the product of all the transmission. At the end of the pipeline we have a resulting radiance, still expressed in units of $W/m^2/\mu m/sr$, and still contained in a *Radiance* class, which is the resulting radiance at the end of the chain, and a transmission that is the equivalent transmission of al the chain. This can be considered as an optical element equivalent to the full optical chain.

Similarly to what presented in *Foregrounds propagation*, we can summarise the process as

$$I_{opt,i+1} = I_{opt,i+1} + I_{opt,i} \cdot \Phi_{opt,i+1}$$

$$\Phi_{opt,i+1} = \Phi_{opt,i+1} \cdot \Phi_{opt,i}$$

Where $I_{opt,i}$ is the radiance of i optical element and $\Phi_{opt,i}$ is its transmission.

Note: Because of the way the light path is parsed. It's important to be careful of the order of writing for the optical element. Optical elements further from the detector should be write first in the *.xml* file.

The `ParsePath` class, allow to combine different optical path. If another optical path has been already parsed, for example, the *Foregrounds* path, the user can set that path as a starting point for the new one to be parser by using the "ligh_path" keyword. Combining more path allow us to have at the end of the cain, a single optical element in front of the detector. In this case, referring to the previous equations we can write

$$I_{opt,1} = I_{opt,1} + I_{prev} \cdot \Phi_{opt,1}$$

$$\Phi_{opt,1} = \Phi_1 \cdot \Phi_{prev}$$

Where $opt, 1$ identifies the first element of the new optical chain, while $prev$ is the result of the previous optical chain.

The output of `ParsePath` is not a single radiance an transmission, but a dictionary containing multiple radiances. In fact, when the light reach the slit, it's diffused. But because the estimation of the diffusion is computed on the focal plane, the code stores the information, and starts collecting the light after the slit as a new radiance. The same happens with the optics and detector box. Because are to be multiplied by different solid angles, and the information is not available to the code until the all channel is parsed, *ExoSim* separates the light in different radiances. So, at the end, we have a the radiance from the contributions before the slits in one key of the dictionary that is the output of `ParsePath`; a radiance from the contributions after the slit but the boxes; a radiance for the optics box and one for the detector boxes.

The user may also want to investigate the effects of a specific surface or contribution. In this case the user can use the keyword *isolate*:

```
<optical_path>
    <opticalElement>
        ...
        <isolate> True <isolate>
    </opticalElement>
</optical_path>
```

This forces the code to isolate that specific contribution and store it separately from the others in the output.

Common optics

The common optics path is describe under the *Telescope* keyword in the descriptions.

```
<Telescope>
    <optical_path>
        ...
    </optical_path>
</Telescope>
```

To optimise the code efficiency in case there are more channels in the payload, we estimate this contribution first. To estimate it we use `ParsePath`. If *Foregrounds* have been parsed before, they should be attached to this path.

```
import exosim.tasks.parse as parse

with output.use(append=True, cache=True) as out:

    payloadConfig = mainConfig['payload']
    out_payload = out.create_group('payload')

    parsePath = parse.ParsePath()
    common_path = parsePath(
        parameters=payloadConfig['Telescope']['optical_path'],
        wavelength=wl_grid, time=time_grid,
        output=out_payload, group_name='telescope',
        light_path=for_contrib )
```

Where *for_contrib* has been produced in *Foregrounds propagation* .

Channel optical path

For each channel a specific optical path can be define and parsed. This can either be estimate with *ParsePath* or using the *Channel* class.

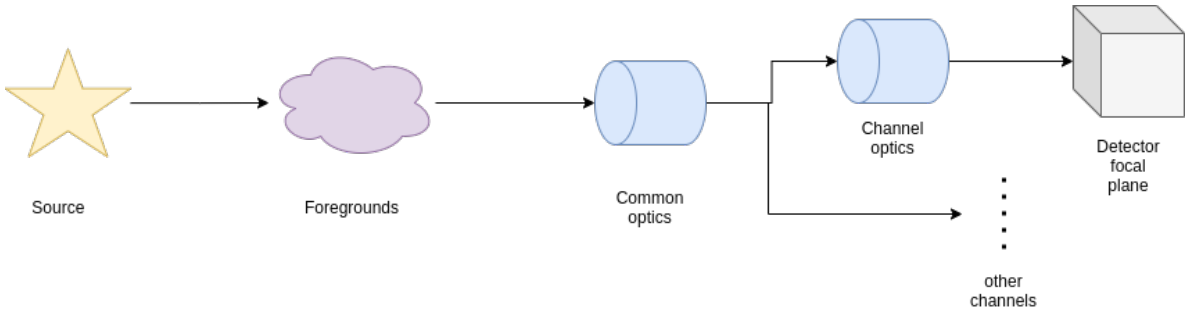
The *Channel* class contains all the routing to move forward to the focal plane production. The class can be instantiated providing a dictionary with the channel description and the *Wavelength grid* and *Temporal grid*. Then the path can be parsed using *parse_path*.

```
from exosim.models import Channel

with output.use(append=True, cache=True) as out:

    channel = Channel(parameters=payloadConfig['channel']['channel_name'],
                      wavelength=wl_grid, time=time_grid, output=out)
    channel.parse_path(light_path=common_path)
```

Other functions of the *Channel* class are discussed in *Channel*.

2.2.5 Channel

Remembering the road to the production of the focal plane, presented in *Focal plane creation*, after the parsing of source, the foregrounds and the common optics, we have to move inside each channel.



The channels are described in the *.xml* ad

```
<channel> channel_name
    <type>spectrometer</type>
</channel>
```

Where the channel *type* can either be *spectrometer* or *photometer*.

These are then handled by the *Channel* class.

As already mentioned in *Channel optical path*, the *Channel* class shall be initialised as

```
from exosim.models import Channel

with output.use(append=True, cache=True) as out:

    channel = Channel(parameters=payloadConfig['channel']['channel_name'],
                      wavelength=wl_grid, time=time_grid, output=out)
```

Where the output use and keyword are to use only if needed.

Optical path

The first action to do with a channel is to parse the dedicated optical path, as already shown in *Channel optical path*.

The channel optical path is to be reported under the channel section in the *.xml* configuration file.

```
<channel> channel_name

    <optical_path>
        <opticalElement> first_optical_element
        </opticalElement>

        <opticalElement> second_optical_element
        </opticalElement>
    </optical_path>

</channel>
```

The following script to be attach to the previous ans it assumes that all the previous script have been run already:

```
channel.parse_path(light_path=common_path)
```

This will parse the channel dedicated optical path and attach it after the common optical path already parsed and stored in *common_path*. The *parse_path* method populates the *path* attribute. The resulting transmission and radiances are datacubes of the size of *Wavelength grid* and *Temporal grid*, encapsulated in *Signal* classes. To learn more about *Signal* class, refer to *Signals*. This format allow for the use of wavelength and time dependent contributions.

Estimate responsivity

The channel responsivity refers to the detector quantum efficiency. Is to be described in the *.xml* configuration file, under the channel section as

```
<channel> channel_name

    <qe>
        <responsivity_task>LoadResponsivity</responsivity_task>
        <datafile>__ConfigPath__/qe.ecsv</datafile>
    </qe>

</channel>
```

As already seen in *User defined foreground* and discussed in *Optical elements*, the *responsivity_task* key identify a customizable task to load the detector quantum efficiency. To learn more about customizing tasks, please refer to *Custom Tasks*. The default task is `LoadResponsivity`. This task loads the *datafile*, that is *.csv* file containing a table of multiple columns. The first column is`Wavelength` and the other are named after the payload channels and contains their qe vs the wavelength. The default `LoadResponsivity` simply loads the right qe for each channel and convert it into $counts/Joule$. The resulting responsivity is a datacube of the size of *Wavelength grid* and *Temporal grid*, encapsulated in a `Signal` class. To learn more about `Signal` class, refer to *Signals*. This format allow for the use of wavelength and time dependent responsivities.

To run this action, the user shall call the `estimate_responsivity` method:

```
channel.estimate_responsivity()
```

Caution: If the user doesn't include the *responsivity_task* keyword in the description, the `estimate_responsivity` method automatically uses the default `LoadResponsivity` task.

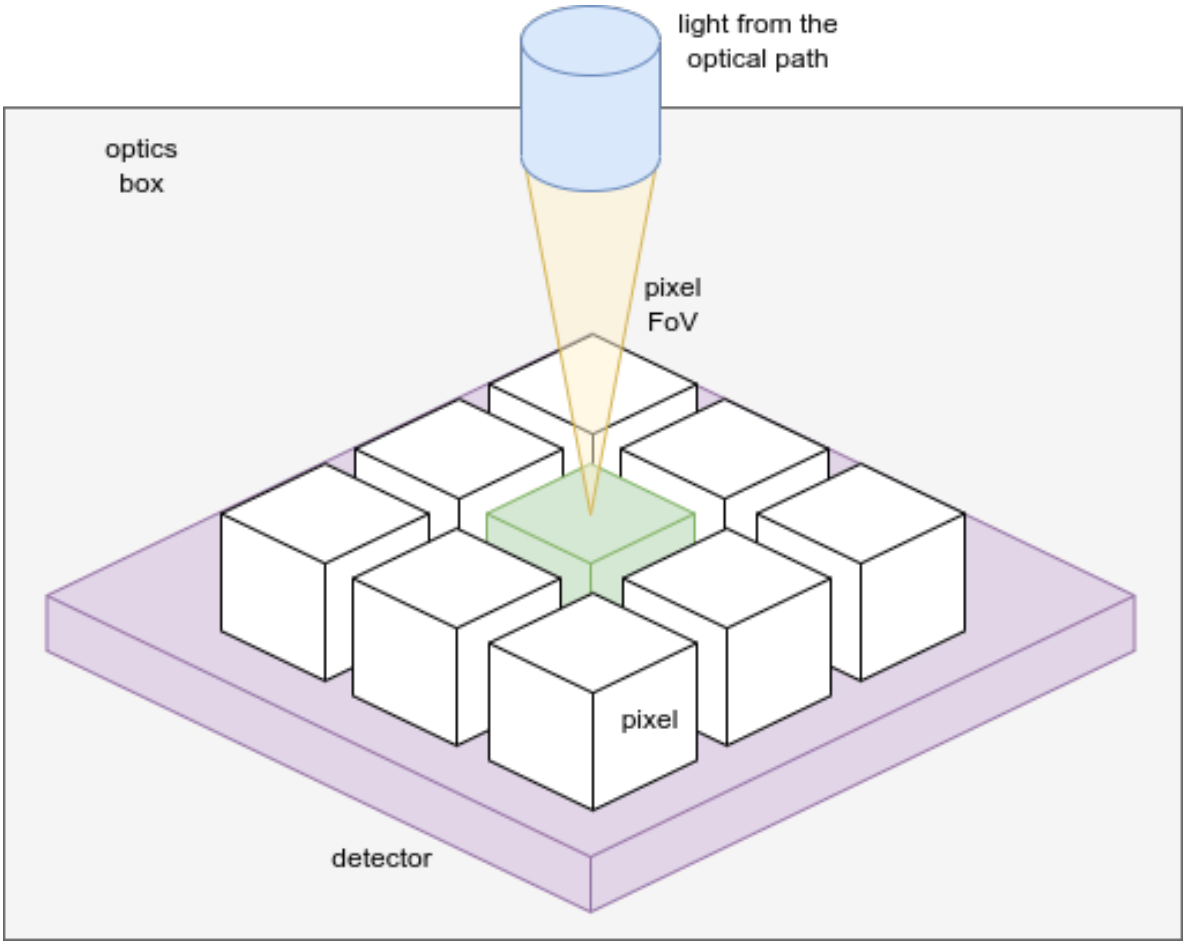
Propagate foreground

To propagate the foregrounds means to multiply the results of the optical path for the detector responsivity and the right solid angle.

$$S_{path,i} = A_{pix} \cdot \Omega_{pix} \cdot \nu \cdot I_{path,i}$$

Where A_{pix} is the pixel area, Ω_{pix} is the solid angle and ν is the detector responsivity. The pixel surface, A_{pix}, is computed from the detector section of *.xml* coinfiguration file:

```
<channel> channel_name
    <detector>
        <delta_pix unit="micron"> 18.0 </delta_pix>
    </detector>
</channel>
```



The solid angle, Ω_{pix}, as already discussed in *Supported optical elements*, depends on the position of the optical element respect to the detector. Everything the comes from the pixel field of view is multiply by the solid angle subtended by an elliptical aperture on-axis. The used algorithm is from Equation n. 56 of "John T. Conway. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 614(1), 17:27, 2010 (https://doi.org/10.1016/j.nima.2009.11.075). To estimate the Ω_{pix} from this equation, other information are required as the f-numbers in the two directions:

```
<channel> channel_name
   <Fnum_x>15.5</Fnum_x>
   <Fnum_y>15.5</Fnum_y>
</channel>
```

where *x* is the dispersion direction and *y* is the spatial direction. If only *x* is provided the two are assumed to be equal. If they are equal the solid angle for a circular aperture is estimated.

If the light comes from the optics box, it is then multiplied by $\pi - \Omega_{pix}$. It it comes from the back of the detector, the *detector box*, then is multiplied by π.

The results will be a dictionary containing the contributions of all the foregrounds (the light paths) expressed as $counts/s/\mu m$.

To run this action, the user shall call the `propagate_foreground` method:

```
channel.propagate_foreground()
```

This method will update the *path* attribute.

Propagate Source

Similarly to what seen before, to propagate the foreground means multiply the source SED by the instrument efficiency to get the density signal expressed as $counts/s/\mu m$.

For each source parsed as described in *Parse from xml*, the resulting density signal is

$$S_{source,i} = A_{tel} \cdot \Phi_{tot} \cdot \nu \cdot I_{source,i}$$

Where A_{tel} is the telescope aperture to be indicated in the *common optics* description in the *.xml* file:

```
<Telescope>
    <Atel unit="m**2">  0.63  </Atel>
    <optical_path>
        ...
    </optical_path>
</Telescope>
```

Φ_{tot} is the final transmission of the optical chain and ν is the detector responsivity.

To run this action, the user shall call the *propagate_sources* method:

```
channel.propagate_sources(sources = sources,
                          Atel = payloadConfig['Telescope']['Atel'])
```

This method will update the *sources* attribute.

2.2.6 Focal plane

Create focal planes

The next step in the the *Channel* pipeline is to create an empty focal planet to populate. This can be done with the method *create_focal_planes*

```
channel.create_focal_planes()
```

This method calls the *CreateFocalPlane* task. This tasks, first build the focal plane array, by using *CreateFocalPlaneArray*.

Detector geometry

The first step is the detector geometry that needs to be specified in the channel detector description:

```
<channel> channel_name

    <detector>
        <delta_pix unit="micron"> 18.0 </delta_pix>
        <spatial_pix>64</spatial_pix>
        <spectral_pix>364</spectral_pix>
        <oversampling>4</oversampling>
    </detector>

</channel>
```

In this case we are building a detector with 64 pixels in the spatial direction and 364 pixel in the dispersion direction, with pixel 18 micron wide. The *oversampling* key allows us to use sub-pixels. In this case we are splitting each pixel in 4 in each direction, hence having 16 sub-pixels.

Note: The main reason to have an oversampling factor is the jitter effect (see *Instantaneous readout*). The oversampling factor is only needed to assure that the PSF is Nyquist sampled (at least 2 per FWHM). The oversampling factor can be any number, but for efficiency reasons it should be a power of 2.

Wavelength solution

If the channel is a *spectrometer*, then the wavelength solution is used to find the wavelength collected by each pixel in the dispersion. The solution can be specified as

```
<channel> channel_name
    <wl_solution>
        <wl_solution_task>LoadWavelengthSolution</wl_solution_task>
        <datafile>__ConfigPath__/wl_sol.ecsv</datafile>
        <center>auto</center>
    </wl_solution>
</channel>
```

The *wl_sol.ecsv* file is a table with 3 columns: *Wavelength*, *x*, *y*, where *x* is the dispersion direction and *y* is the spatial direction. If *y* is set to 0 for each wavelength, the source light is assumed to be dispersed only along the dispersion direction, otherwise is to be dispersed also on the spatial direction. The wavelength associated to each pixel in the spectral and spatial directions are stored along the focal plane in the `Signal` class in the *spectral* and *spatial* attributes..

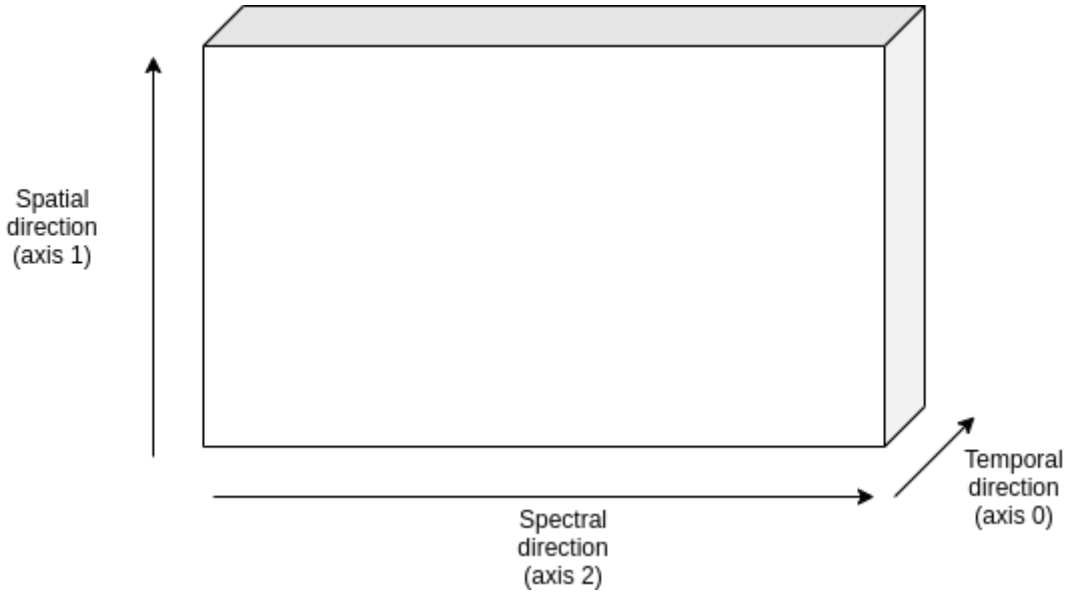
The *wl_solution_task* indicates the task to use to load the wavelength solution. By the default the `LoadWavelengthSolution` is used. This `Task` can be customised, as described in *Custom Tasks*.

The **center** key is used to set the central pixel in the spectral direction. If "auto" it sets the central wavelength of the channel in the center of the pixel array. If a wavelength is indicated, it centers the wl solution on that wavelength. Else, it shifts the pixel array by the indicated number of pixels.

If the channel is a *photometer* there is no need to specify the wavelength solution. The `CreateFocalPlaneArray` tasks will use the detector responsivity to estimate a wavelength solution to use for the next step (*Rescale Contributions*).

Source and foregrounds Focal planes

Once the array is built, the `CreateFocalPlane` task creates a stack of array along the temporal direction.



Finally, `create_focal_planes` duplicates it to produce a focal plane for the foreground contributions. This method populates the *focal_plane* and *frg_focal_plane* attributes in the `Channel` class.

Rescale Contributions

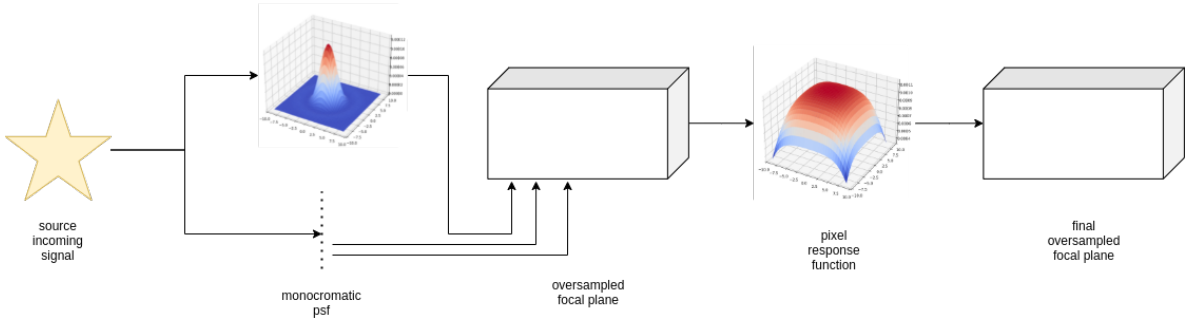
Knowing now the size of the focal planes and the wavelength solutions, we can rescale the incoming signal to convert them from signal densities ($counts/s/\mu m$) into proper signals ($counts/s/pixel$)

```
channel.rescale_contributions()
```

The `rescale_contributions` method updates the *sources* and the *path* keys in the `Channel` class by rebinning the the signals according to the focal plane dispersion binning. Then it estimates the wavelength solution gradient from the pixel wavelength solution and it multiply the signal by this gradient.

Populate focal plane

Next it is finally time to populate the source focal plane. We now follow the following scheme:



First we need to produce a monochromatic PSF for each wavelength sampled in the pixel wavelength solution. Then we multiply PSF by the source signal to the respective wavelength and we add the result on the relative

pixel. On the now populated focal plane, we then apply the Intra-pixel Response Function (IRF).

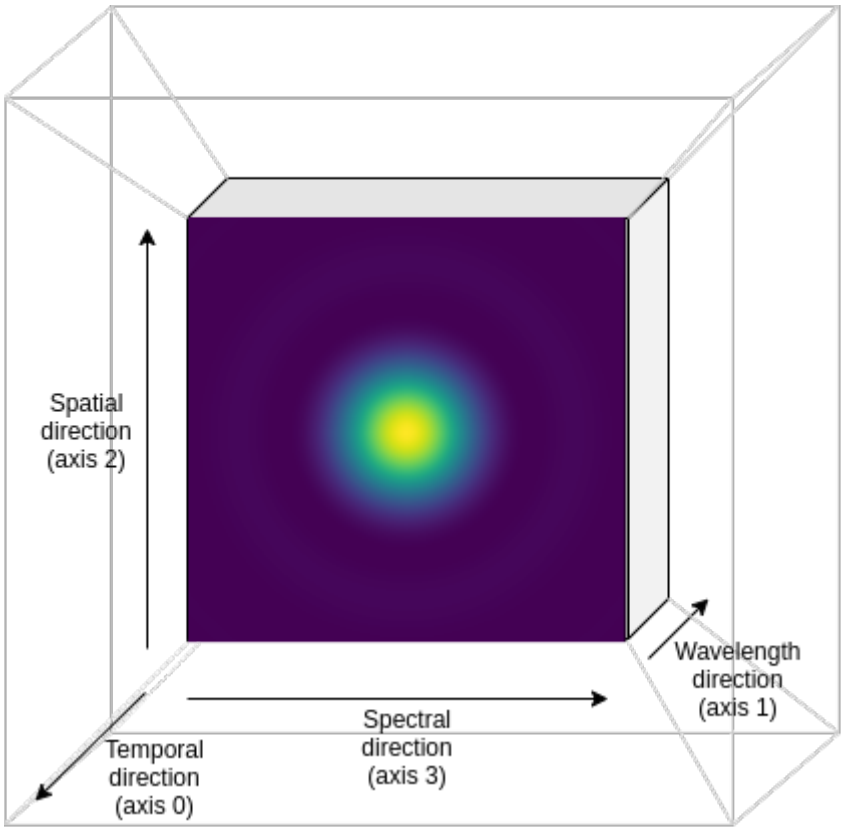
The first steps are handled by

```
channel.populate_focal_plane()
```

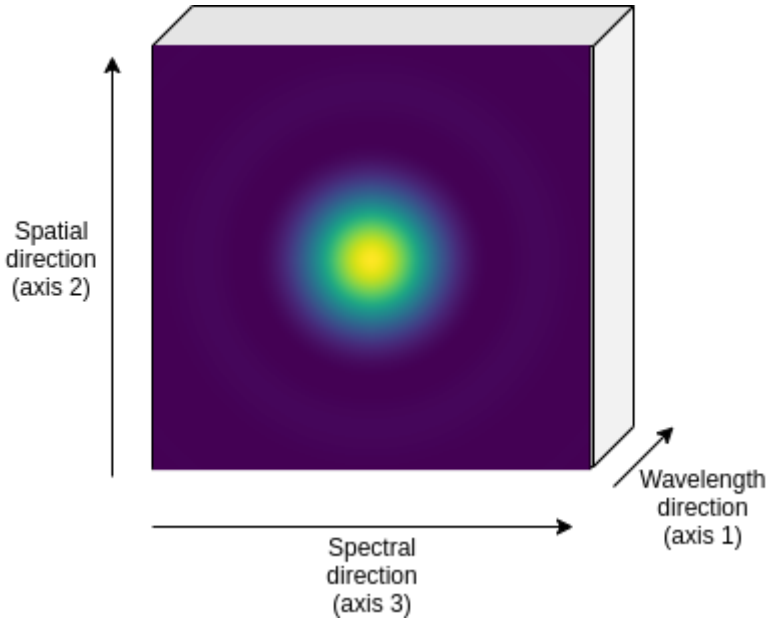
The *populate_focal_plane* method calls *PopulateFocalPlane* task.

PSF

The first step mentioned above is the production of the Point Spread Function ipercube.



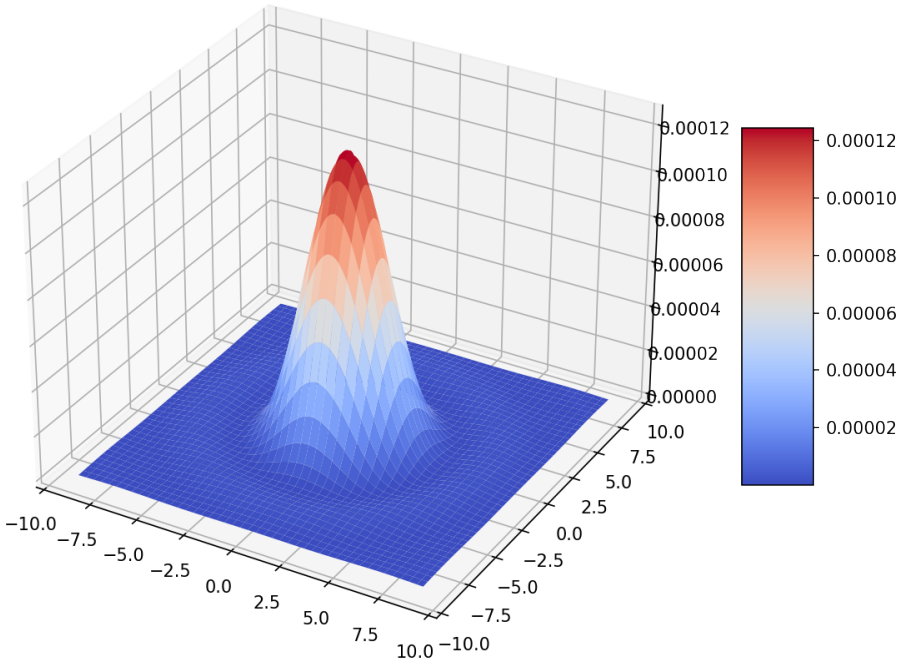
For each temporal step, the PSF cube is defined as in the following figure:



The PSF specifics are to be listed in the *psf* section of the *.xml* channel description. The simplest PSF are described by the *Airy* or by the *Gauss* functions.

```
<channel> channel_name
    <psf>
        <shape>Airy</shape>
    </psf>
</channel>
```

In this case, the `PopulateFocalPlane` task calls `create_psf` This function produce a PSF cube as the one showed before, where the volume of ech PSF is normalised to unity:



The *psf* section can be customised by adding the following keys:

```
<channel> channel_name
    <psf>
        <shape>Airy</shape>
        <nzero> 8 </nzero>
        <size_y> 64 </size_y>
        <size_x> 64 </size_x>
    </psf>
</channel>
```

Where, *nzero* indicates the numbers of zero in the Airy function, *size_x* and *size_y* are the size of the PSF cube in the spectral and spatial directions. *size_x* and *size_y* can also be set to *full* to use the full size of the focal plane.

However, the user may want to load specific PSF shapes. This can be done by writing a dedicated `LoadPsf` task. `LoadPsf` task produces an iper-cube, where to each temporal step of the focal plane is associated a PSF cube as the one in the previous picture. The native PSF format supported by *ExoSim* is PAOS format and the functionality is provided by `LoadPsfPaos`. In this case the user shall specify it in the *.xml* file as

```
<channel> channel_name
    <psf>
        <psf_task>LoadPsfPaos</psf_task>
        <filename>__ConfigPath__/paos_file.h5</filename>
    </psf>
</channel>
```

The `LoadPsfPaos` task loads the PSF cube provided by the *filename* data. The PSF are then interpolated over a grid matching the one used to produce the focal planes, to convert them into the physical units. Then the total volume of the interpolated PSF is rescaled to the total volume of the original one. This allow to take into account for loss in the transmission due to the optical path. The PSF are then interpolated over a wavelength grid

matching the one used to for the focal plane, producing the cube. This would fasten up the successive *ExoSim* steps. The default *LoadPsfPaos* task does not include a temporal dependency, and therefore the PSF cube is repeated on the temporal axis.

The user can define a temporal dependence by using a custom *LoadPsf* task. An example using PAOS PSF is reported in *LoadPsfPaosTimeInterp*.

Finally, the PSF obtained are stored in the output file.

Adding PSF to the focal plane

Once the PSF cube is ready, for each temporal step of the focal plane, we add a monochromatic PSF to the relative pixel multiplying it by the relative intensity of the source signal at the same temporal step. This allow us to produce a dispersed image in the case of a *spectrometer* or to cumulate the PSF in the case of a *photometer*. Also, if the source signal as a time dependent variation, this is propagated to the image on the focal plane thanks to the use of the same temporal step both in the focal plane and the source signal. The results will be an oversampled focal plane.

Intra-pixel Response Function

The pixels on the focal plane do not have an uniform responsivity to the incoming light on their surfaces. They are known to be more responsive at the center and less to the edges. This effect can be represented in *ExoSim* introducing the IRF.

This is handled by the *apply_irf* method:

```
channel.apply_irf()
```

Create IRF

The task to use to estimate the IRF is indicated as

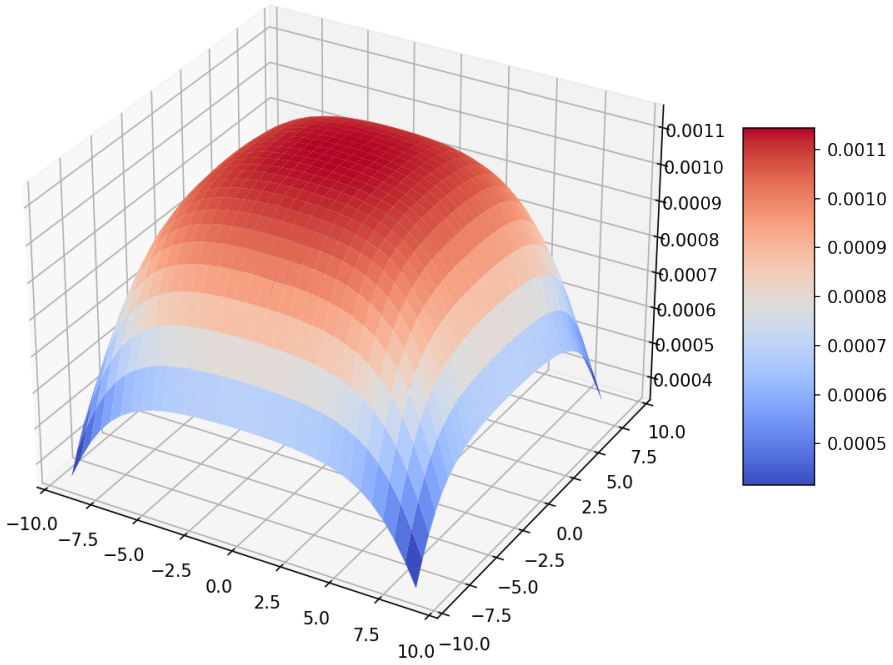
```
<channel> channel_name
    <detector>
        <irf_task>CreateIntrapixelResponseFunction</irf_task>
    </detector>
</channel>
```

where *CreateIntrapixelResponseFunction* is the default class. This tasks implements the equation presented in Barron et al., PASP, 119, 466-475, 2007 (https://doi.org/10.1086/517620). It required the pixel *diffusion length* and the *intra-pixel distance*:

```
<channel> channel_name
    <detector>
        <irf_task>CreateIntrapixelResponseFunction</irf_task>
        <diffusion_length unit="micron">1.7</diffusion_length>
        <intra_pix_distance unit="micron">0.0</intra_pix_distance>
    </detector>
</channel>
```

Two other default tasks are available to create the IRF: *CreateOversampledIntrapixelResponseFunction*. The first one is a simple oversampling of the IRF, while the second one is a oversampling of the IRF with a larger size.

The user can however specify its own tasks and the relative parameters. Notice that the IRF volume is expected to be normalised to unity. Here is an example of a resulting IRF:



Caution: If no *irf_task* key is provided in the channel description, the `apply_irf` method automatically uses the default `CreateIntrapixelResponseFunction` task.

IRF application

When the Pixel response function is produced, we apply it using the `ApplyIntraPixelResponseFunction`. This task performs a convolution between the focal plane and the IRF.

Now the source focal plane is completed.

Note: In the default recipe (*Focal plane automatic Recipe*), if no *irf_task* key is provided in the channel description, the IRF step is skipped.

The user can specify the convolution method to use:

```
<channel> channel_name
    <detector>
        <convolution_method>fftconvolve</convolution_method>
    </detector>
</channel>
```

The available methods are: *fftconvolve* (`scipy.signal.fftconvolve`[13]), *convolve* (`scipy.signal.`

[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html#scipy.signal.fftconvolve

convolve[14]), *ndimage.convolve* (scipy.ndimage.convolve[15]) and *fast_convolution* (*exosim.utils.convolution.fast_convolution*). If no convolution_method is specified, the default is *fftconvolve*.

Note: The *fast_convolution* method is the same implemented in *Sarkar et al., 2021 <https://link.springer.com/article/10.1007/s10686-020-09690-9>`__*. It is very accurate but it is slower than the other methods and requires a lot of memory. It is therefore recommended to use it only for small oversampling factor.

The *CreateIntrapixelResponseFunction* task creates a kernel compatible with both *fftconvolve* (scipy.signal.fftconvolve[16]), *convolve* (scipy.signal.convolve[17]) and *ndimage.convolve* (scipy.ndimage.convolve[18]). The tasks *CreateOversampledIntrapixelResponseFunction* is instead compatible with *fast_convolution* (*exosim.utils.convolution.fast_convolution*), which is a method developed specifically for ExoSim.

Populate foreground focal plane

To populate the foregrounds focal plane, we can call the *populate_foreground_focal_plane* method:

```
channel.populate_foreground_focal_plane()
```

This involves the *ForegroundsToFocalPlane* task, that simply adds the foregrounds contributions, stored in the *path* attribute, to the foreground focal planet, stored in the *frg_focal_plane* attribute.

If the *path* element to add is before a slit, the signal is dispersed. Therefore the contribution signal is convolved with a kernel of the width of the slit expressed as number of pixel, and then summed to the full array. If the slit width is expressed in number of pixel at the focal plane is L, and the spectral resolving power computed at a certain λ_0 is $R(\lambda_0)$, the detector received diffuse radiation over the wavelength range $\left(\lambda_j - \frac{L\lambda_0}{4R(\lambda_0)}\ ,\ \lambda_j + \frac{L\lambda_0}{4R(\lambda_0)}\right)$, and not over the full range of wavelength accepted by the filter. So, the j-th pixel sampling the λ_j wavelength the collected signal is

$$S(j) = \int_{\lambda_j - \frac{L\lambda_0}{4R(\lambda_0)}}^{\lambda_j + \frac{L\lambda_0}{4R(\lambda_0)}} S_{for}(\lambda) d\lambda$$

If the *path* element to add is after a slit, or if no slit is in the path, the signal integrated on the full wavelength range is simply added to each pixel:

$$S = \int S_{for}(\lambda) d\lambda$$

Now the foreground focal plane is completed.

[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html#scipy.signal.convolve
[15] https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html#scipy.ndimage.convolve
[16] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html#scipy.signal.fftconvolve
[17] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html#scipy.signal.convolve
[18] https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html#scipy.ndimage.convolve

Foreground sub focal planes

If at least one optical element has

```
<optical_path>
    <opticalElement>
        ...
        <isolate> True <isolate>
    </opticalElement>
</optical_path>
```

Then the sub focal planes are computed. The same *populate_foreground_focal_plane* method also populates a *frg_sub_focal_planes* attribute. This is a dictionary containing all the foreground signal contribution, highlighting the ones marked with `isolate=True`. The sum of all the sub focal planes matches *frg_focal_plane*.

This mode allows the user to investigate the effects of a single optical surface.

2.2.7 Telescope pointing and multiple sources

In this section we will discuss how to simulate the source position in the sky, relatively to the telescope pointing. In a real observation, the telescope is pointed to a certain location in the sky, where the target is expected to be, and other sources can be in the field.

Note: This pointing section has nothing to do with pointing stability simulations or with a pointing direction changing with time. Here we are only simulating the telescope ideal pointing.

The position of the target from the sky to the focal plane is handled in *PopulateFocalPlane* by the *ComputeSourcesPointingOffset*. The offset resolution is in integer multiple of the subpixel size.

Telescope pointing

In *exosim* the telescope pointing direction can be set in the main configuration *.xml* file (see *General settings*). Assuming we want to observe HD209458[19] we point the telescope to the target coordinates by adding

```
<root>
    <pointing>
        <ra> 22h03m10.8s </ra>
        <dec> +18d53m04s </dec>
    </pointing>
</root>
```

Then, we need to add the coordinates also to the star in the source description (see *Sources*):

```
<source> HD 209458
     <source_type> planck </source_type>

     <R unit="R_sun"> 1.17967 </R>
     <M unit="M_sun"> 1.1753 </M>
     <T unit="K"> 6086 </T>
     <D unit="pc"> 47.4567 </D>
```

(continues on next page)

[19] http://simbad.u-strasbg.fr/simbad/sim-id?Ident=HD%20209458

(continued from previous page)

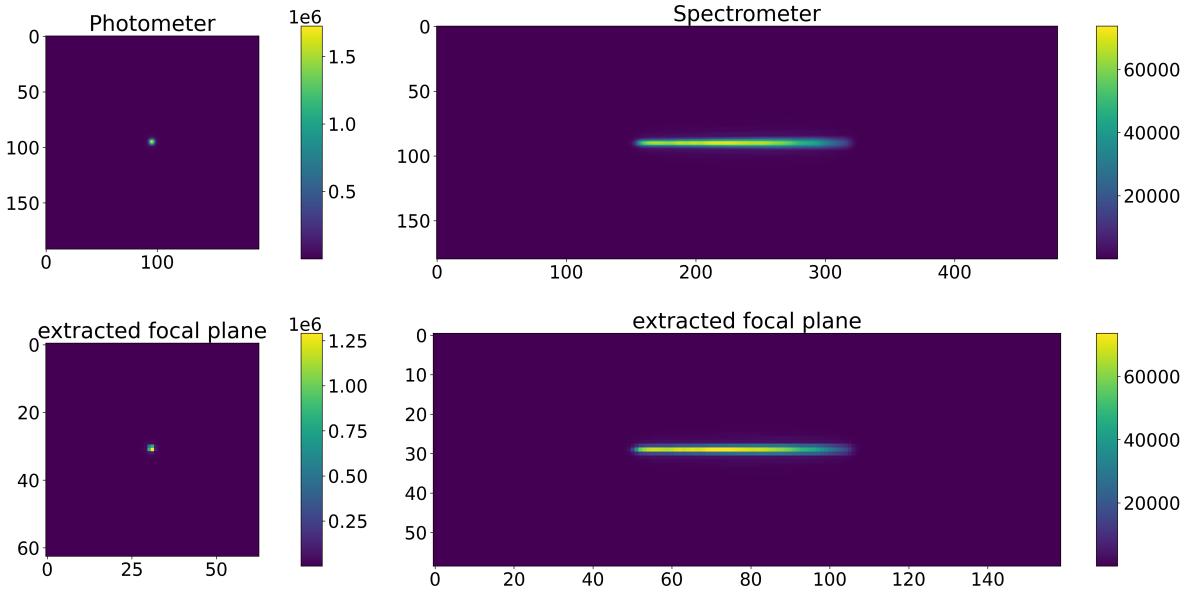
```
    <z unit=""> 0.0 </z>

    <ra> 22h03m10.8s </ra>
    <dec> +18d53m04s </dec>
</source>
```

Another important information is the channel plane scale. In fact, *exosim* needs to estimate the angle of view of each pixel to estimate the star position in the focal plane. Assuming the instrument has two channel: a photometer and a spectrometer, we can add this information under the *detector* section as

```
<channel> Photometer
    <type> photometer </type>
    <detector>
        <plate_scale unit="arcsec/micron"> 0.01 </plate_scale>
    <detector>
</channel>

<channel> Spectrometer
    <type> spectrometer </type>
    <detector>
        <plate_scale>
            <spatial unit="arcsec/micron"> 0.01 </spatial>
            <spectral unit="arcsec/micron"> 0.05 </spectral>
        </plate_scale>
    <detector>
</channel>
```

In this example, the spectrometer has different plate scales in the two detector directions.

Because we are pointing directly to the target, the star will be at the center of the focal plane:

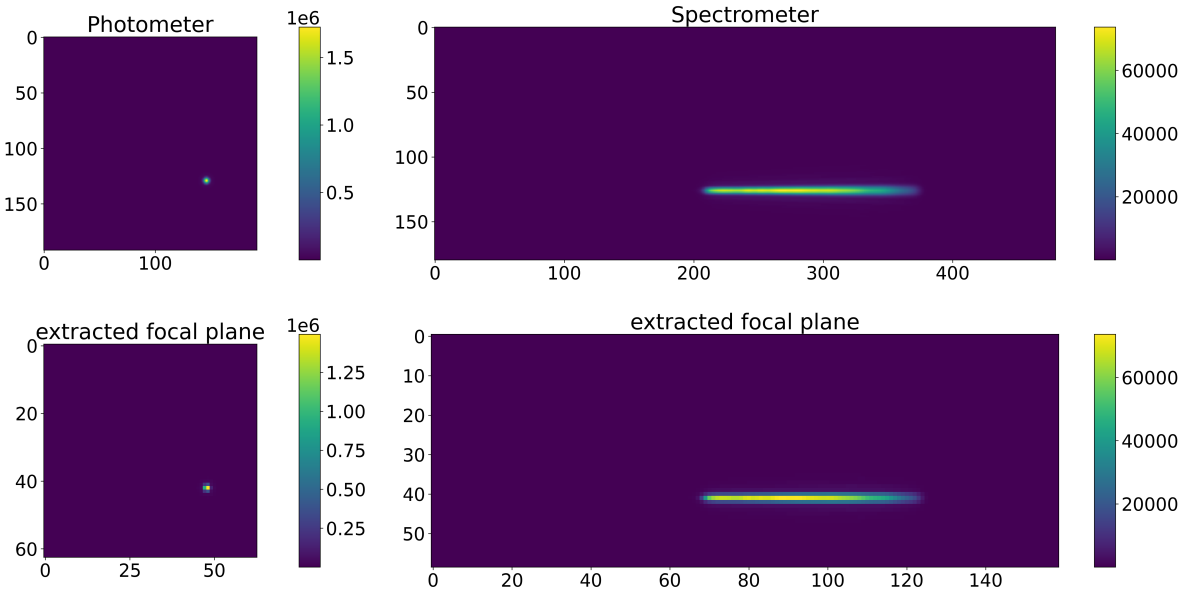


Pointing offset

If we want to simulate an offset of the source on the focal plane, we can mode the telescope pointing. In this example we simply changed the pointing in the main configuration *.xml* file:

```
<root>
    <pointing>
        <ra> 22h03m11s </ra>
        <dec> +18d53m06s </dec>
    </pointing>
</root>
```

The result will be a different location of the target on the focal plane



Multiple sources in the field

Another useful case is the simulation of multiple sources on the focal plane. In this example we add two other targets. To keep things simple, we add two HD209458 stars to the field: HD209458 1 and HD209458 2. We change the star distances a little, to differentiate them on the focal plane: HD209458 1 is set at 55 pc and HD209458 2 is set at 35 pc instead of 47 pc as the original star location. Also the star locations on the sky are little changed to generate the offsets:

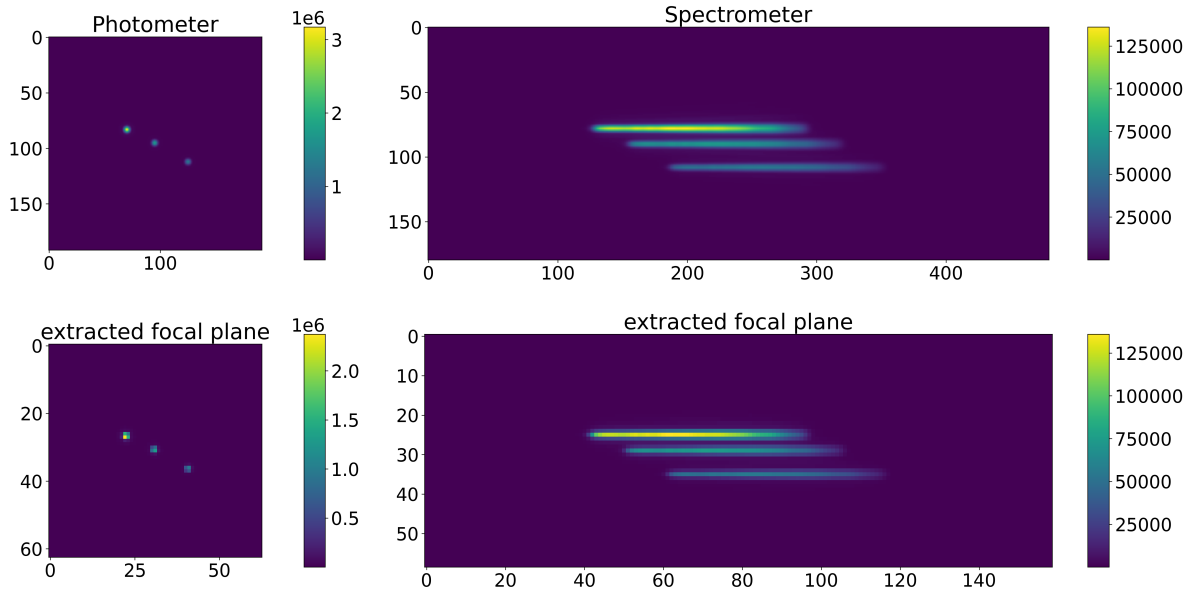
```
<source> HD 209458 1
    <source_type> phoenix </source_type>
    <path>/usr/local/project_data/sed </path>

    <R unit="R_sun"> 1.17967 </R>
    <M unit="M_sun"> 1.1753 </M>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 55 </D>
    <z unit=""> 0.0 </z>

    <ra> 22h03m10.68s </ra>
    <dec> +18d53m03s </dec>
```

(continues on next page)

(continued from previous page)

```
</source>


<source> HD 209458 2
    <source_type> phoenix </source_type>
    <path>/usr/local/project_data/sed </path>

    <R unit="R_sun"> 1.17967 </R>
    <M unit="M_sun"> 1.1753 </M>
    <T unit="K"> 6086 </T>
    <D unit="pc"> 35 </D>
    <z unit=""> 0.0 </z>

    <ra> 22h03m10.9s </ra>
    <dec> +18d53m04.7s </dec>
</source>
```

The results will be like:



For the following, it is important to separate the target source, which is the source that we expect to have an astronomical signal associated (see *Astronomical signals*), from the others. This can be done by adding the *source_target* attribute to the source description:

```
<source> HD 209458
     <source_target>True</source_target>
 </source>
```

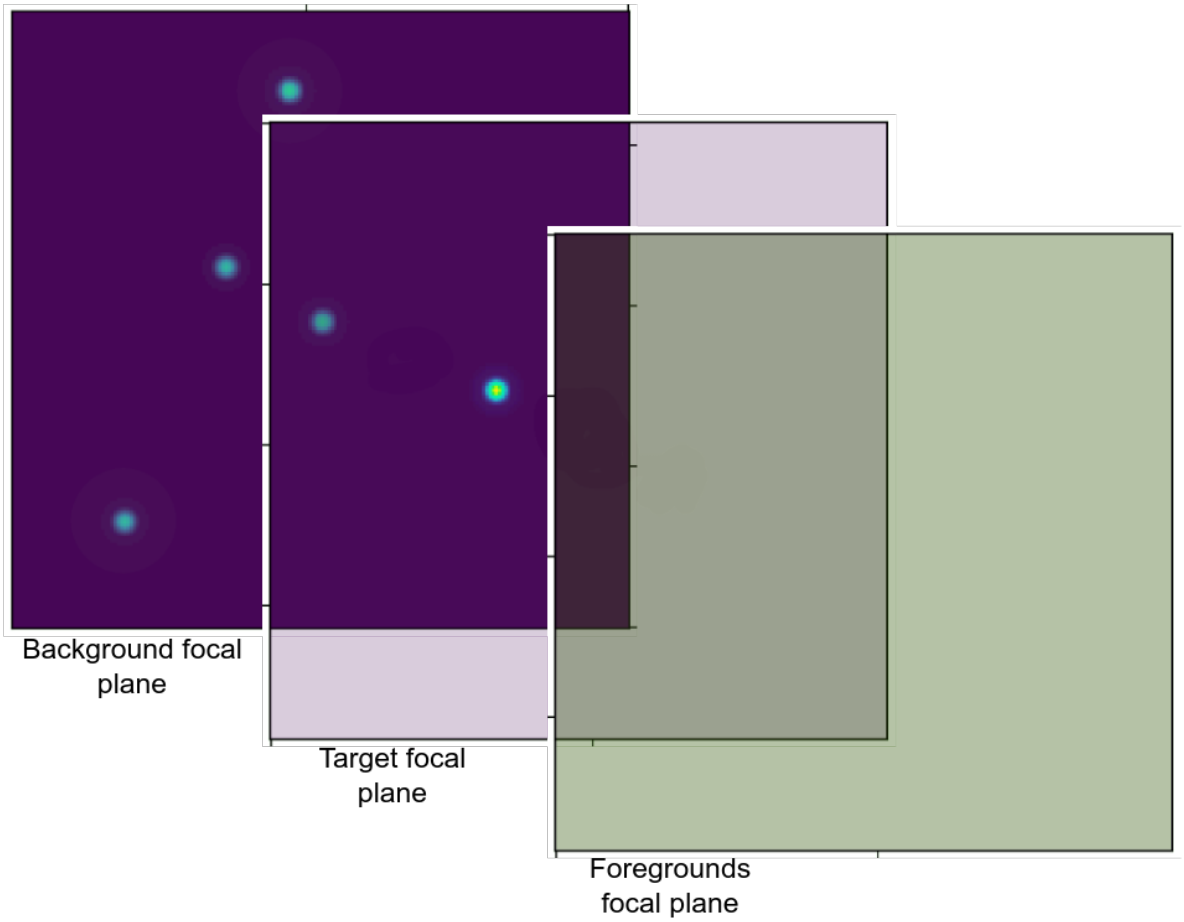
Then the target source will be treated differently from the others. In the focal plane data product, the target source will be stored under the `focal_plane` group. The other sources are considered *background sources* and will be stored under the `bkg_focal_plane` group.

2.2.8 Resulting focal planes

In the previous sections we saw how to produce the focal planes for different channels. Here we'll explain how this focal planes looks like.

Note that so far we have built three different focal planes, that can be considered are three layers:

- `focal_plane`: containing the signal from the target star
- `bkg_focal_plane`: containing the stars in the field of view
- `frg_focal_plane`: containing the signal from the foreground



The reason for this separation is cleared when the astronomical signal (see *Astronomical signals*) needs to be added to the target star.

We remember here that in the previous sections we mentioned the possibility to oversampling the pixel array. In that case the resulting focal plane will be oversampled.

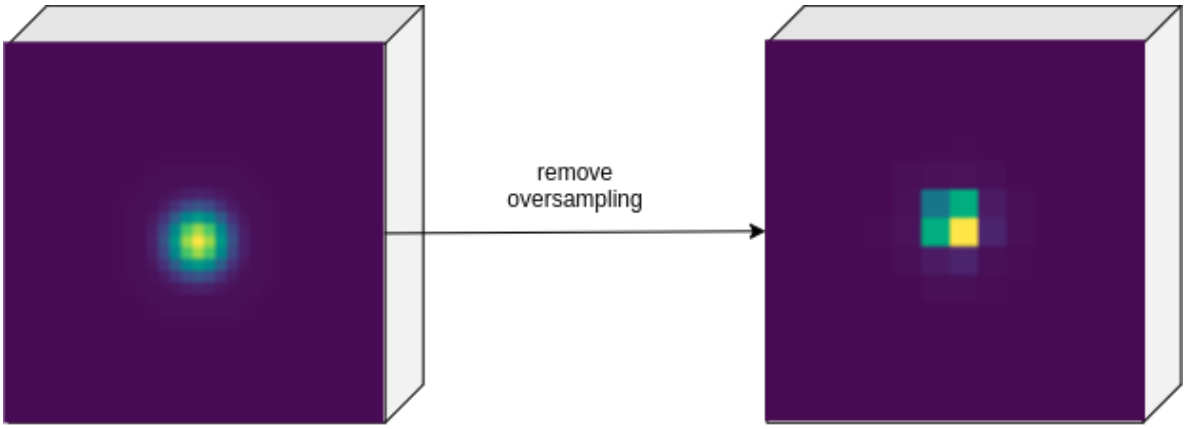
In the following we report examples for both the oversampled focal plane and the real focal plane. To move from the oversampled focal plane to the original one, one can use the following script

```
osf = 4
original = focal_plane.data[:, osf//2::osf, osf//2::osf]
```

where we assumed *focal_plane* to be the produced oversampled focal plane with an oversampling factor *osf=3*.

Photometers

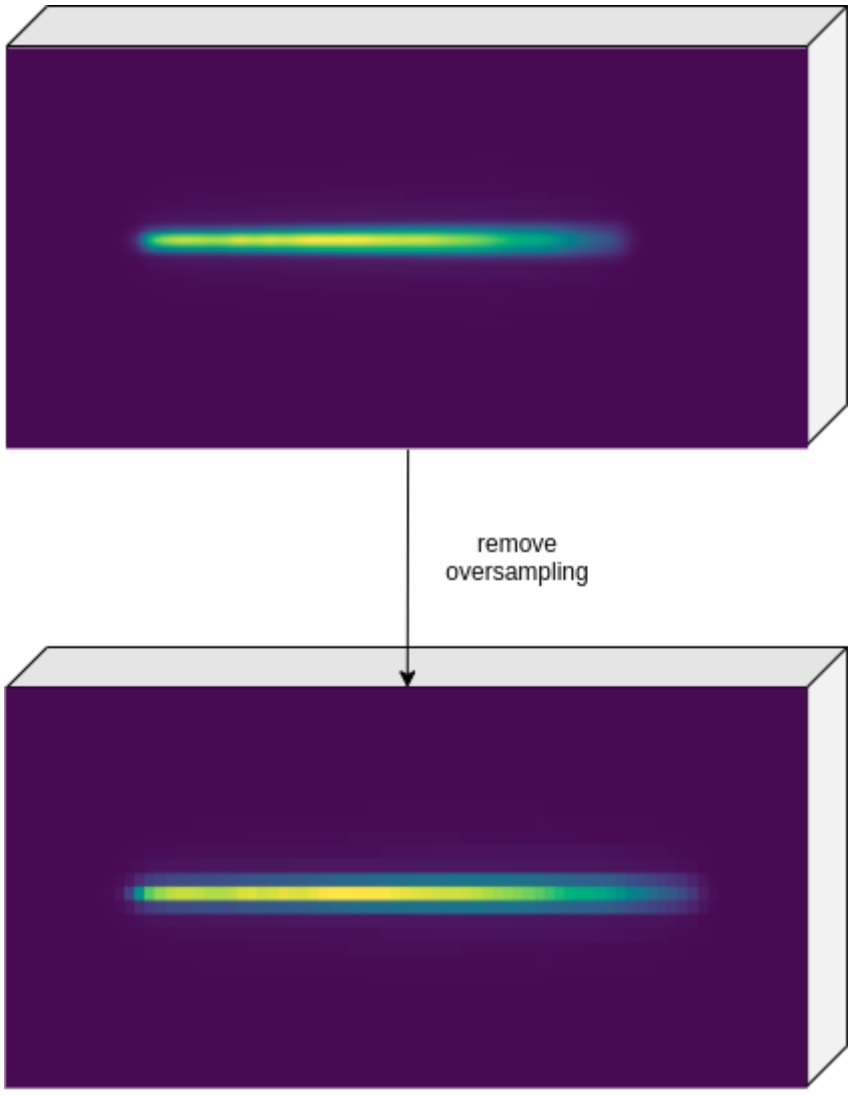
Here we report both the oversampled focal plane and the original one. In this example we used an oversampling factor of 4.



The focal planes looks like data cube because we recall here that the first axis is for time evolution.

Spectrometers

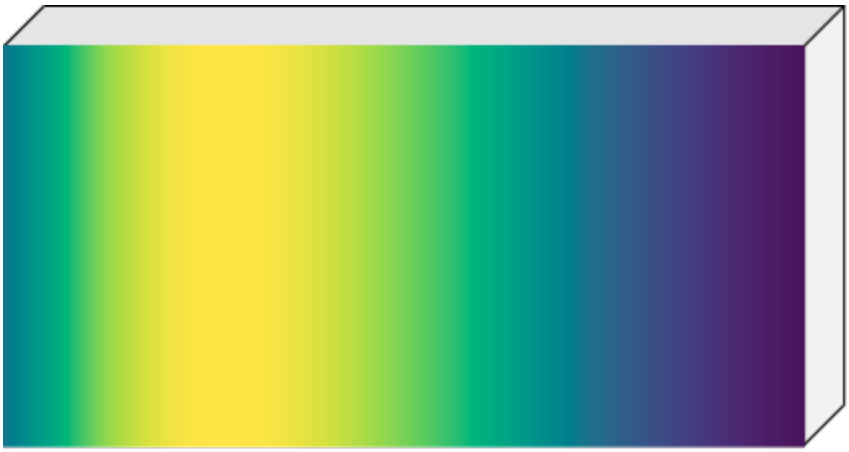
Here we report again both the oversampled focal plane and the original one. We still used an oversampling factor of 4.



Foregrounds

In this example, we report first the case of a not-dispersed foreground focal plane, which results in a constant value over the full oversampled array.

Then we include the example of a dispersed foreground focal plane:



Store and load the focal planes

To store the focal plane into the output file, the user can simply use the *write* method of the *Signal* class:

```
channel.focal_plane.write()
channel.frg_focal_plane.write()
```

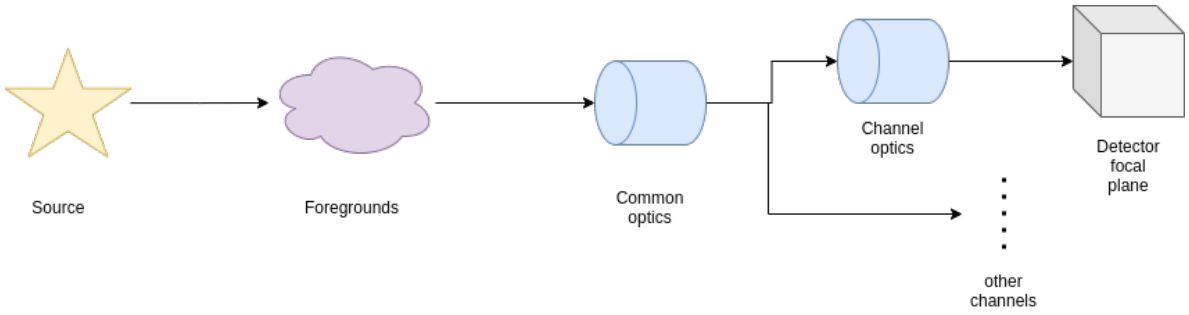
The sub focal planes, if generated, can be stored as

```
for key, value in channel.frg_sub_focal_planes.items():
    value.write()
```

If the output format is the default HDF5[20], refer to *How can I load HDF5 data into my code?* in the *FAQs* section for how to use the data, and see *Load signals and tables* in particular to cast the focal plane into a *Signal* class.

2.2.9 Focal plane automatic Recipe

By appending all the scripts shown previously we produce a pipeline for the production of the focal plane.



The scripts are already collected in a pre-made pipeline. This is under the *recipes* of *ExoSim*.

```
from exosim import recipes
recipes.CreateFocalPlane(options_file='your_config_file.xml',
                         output_file='output_file.h5')
```

[20] https://www.hdfgroup.org/solutions/hdf5/

The *CreateFocalPlane* can also be run from console as

```
exosim-focalplane -c your_config_file.xml -o output_file.h5
```

or

```
exosim-focalplane -c your_config_file.xml -o output_file.h5 -P
```

to also run ExoSim *FocalPlanePlotter*, which is documented in *Focal plane Plotter*.

Other useful capabilities of the focal plane creation process are documented in:

2.3 Radiometric Model

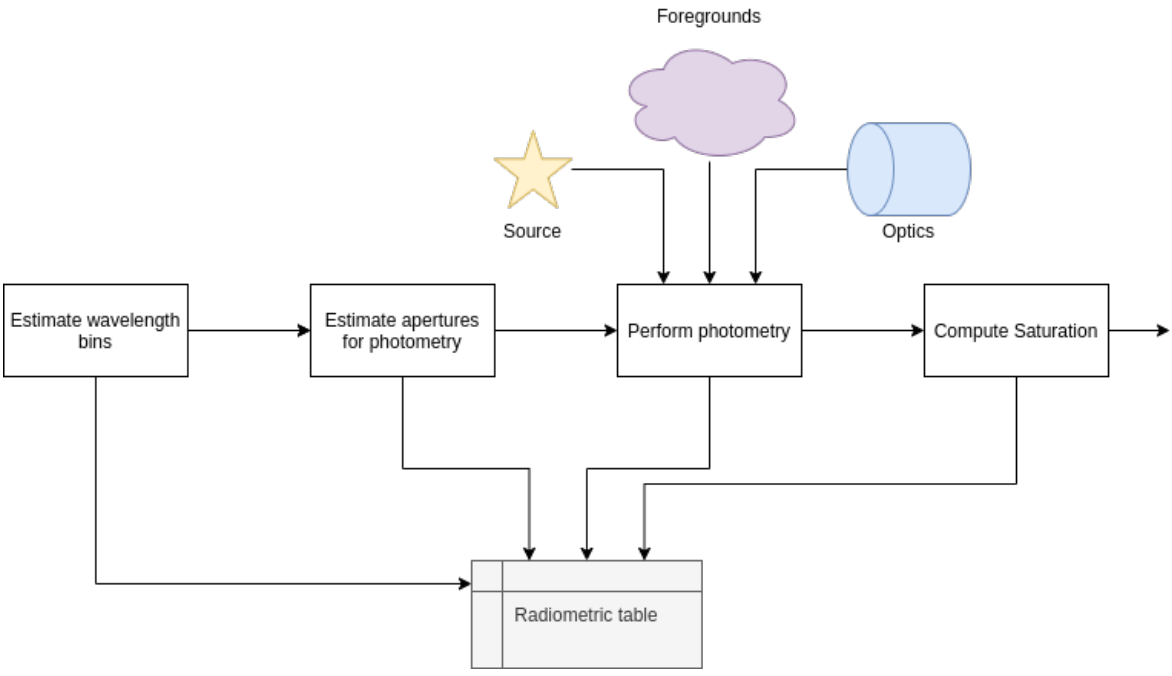
Once the focal plane has been created (see *Focal plane creation*), *ExoSim* can run a radiometric model for a fast estimate of channel parameters. The Radiometric model itself is handled as a recipe: *RadiometricModel*

For a complete description of the steps involve, we will refer in this guide to the pipeline encapsulated in the *RadiometricModel* class, listing its method and discussing the involved *Task*.

To initialise the *RadiometricModel* we assume for now that the user has already a focal plane produced following *Focal plane creation* or by using *CreateFocalPlane*.

2.3.1 Prepare and populate the radiometric table

Starting from a focal plane, the first steps are to create and populate the radiometric table. This includes the steps reported in the following figure:



To summarise, the first step is to create the wavelength grid for the table, then we can move to estimate the signal. We can estimate the signal using aperture photometry on every focal plane stored in the input file. From the source and background focal plane, we can then estimate the saturation time of the detector.

Each step is extensively described in the following pages:

Wavelength binning

The first step is to produce the starting radiometric table with the spectral bins and their edges. This can be done for each channel by default *EstimateSpectralBinning*. The user can specify a dedicated *Task* in the channel description as described later in this section.

Inside *RadiometricModel* this tasks is handled by the *create_table* method. To use the default task in a script on a channel parsed in a dictionary, the user can write:

```
import exosim.tasks.radiometric as radiometric

estimateSpectralBinning = radiometric.EstimateSpectralBinning()
table = estimateSpectralBinning(parameters=channel_dict)
```

Caution: If the user doesn't include the *spectral_binning_task* keyword in the channel description, the default *EstimateSpectralBinning* task is used. To develop a custom *Task*, please refer to *Custom Tasks*.

This *Task* returns an `astropy.table.QTable`[21] for each channel. The table has only the following keywords:

keyword	content
ch_name	channel name
Wavelength	central bin wavelength in μm
bandwidth	band width of the spectral bin in μm
left_bin_edge	left edge of the spectral bin
right_bin_edge	right edge of the spectral bin

The *EstimateSpectralBinning* task include different methods to estimate the spectral binning, that can be tuned in the channel description document.

Photometer

For a photometer the description *xml* file should look like this:

```
<channel> channel_name
    <type> photometer </type>
    ...
</channel>
```

In this case the radiometric table is estimated as the central wavelength of the photometer with a bin width equal to the wavelength band. Therefore the maximum and minimum wavelengths must be indicated along with the units in the *xml* file:

```
<channel> channel_name
    <type> photometer </type>

    <spectral_binning_task> EstimateSpectralBinning </spectral_binning_task>
    <wl_min unit="micron"> 0.5 </wl_min>
    <wl_max unit="micron"> 0.6 </wl_max>
```

(continues on next page)

[21] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

(continued from previous page)

```
    ...
</channel>
```

Spectrometer

For a spectrometer the description *xml* file should look like this:

```
<channel> channel_name
    <type> spectrometer </type>

    ...
</channel>
```

The wavelength grid can be estimated in 2 modes:

- *native* mode. If *targetR* is set to *native* the wavelength grid computed is the pixel level wavelength grid, where each bin is of the size of a pixel;
- *fixed R* mode. If targetR` is set to a constant value, the wavelength grid is estimated using *wl_grid*.

The modes must be indicated in the configuration *xml* file along with the maximum and minimum wavelengths. The *native* configuration will look like this

```
<channel> channel_name
    <type> spectrometer </type>

    <spectral_binning_task> EstimateSpectralBinning </spectral_binning_task>
    <wl_min unit="micron"> 2 </wl_min>
    <wl_max unit="micron"> 6 </wl_max>
    <targetR> native </targetR>

    ...
</channel>
```

The *fixed R* configuration will be like

```
<channel> channel_name
    <type> spectrometer </type>
    <spectral_binning_task> EstimateSpectralBinning </spectral_binning_task>
    <wl_min unit="micron"> 2 </wl_min>
    <wl_max unit="micron"> 6 </wl_max>
    <targetR> 50 </targetR>

    ...
</channel>
```

Estimate apertures

Once the wavelength table is ready, is it time to estimate the aperture sizes for the photometry. By default this is handled by *EstimateApertures*. This Task has lot of options inside, but is still possible to define a custom task to replace this one. To develop a custom *Task*, please refer to *Custom Tasks*. The aperture task must be specified under the *radiometric* keyword:

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```

Inside *RadiometricModel* this tasks is handled by the *compute_apertures* method. To use the default task in a script on a channel, the user can write:

```
import exosim.tasks.radiometric as radiometric

estimateApertures = radiometric.EstimateApertures()
aperture_table = estimateApertures(table=table,
                                   focal_plane=focal_plane,
                                   description=description['radiometric']['aperture_
↪photometry'],
                                   wl_grid=wl_grid)
```

Where *table* is the wavelength radiometric table, *focal_plane* is the channel source focal plane array, *description* is the dictionary containing the aperture photometry information from the *xml* file, and *wl_grid* is the focal plane wavelength grid.

Caution: If the user doesn't include the *apertures_task* keyword in the channel description, the default *EstimateApertures* task is used. To develop a custom *Task*, please refer to *Custom Tasks*.

The results of this *Task* is a `QTable`[22] with the centers, sizes and shapes of the apertures for the channel.

keyword	content
spectral_center	center of the aperture in the spectral direction
spectral_size	size of the aperture in the spectral direction
spatial_center	center of the aperture in the spatial direction
spatial_size	size of the aperture in the spatial direction
aperture_shape	shape of the aperture (rectangular or elliptical)

In the following we will investigate and discuss the *Task* options.

[22] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

Spectral and Spatial modes

By specifying the spectral and spatial modes, the user can define the way the focal plane data are summed in the two directions. By combining different options the user can have control on the summing method.

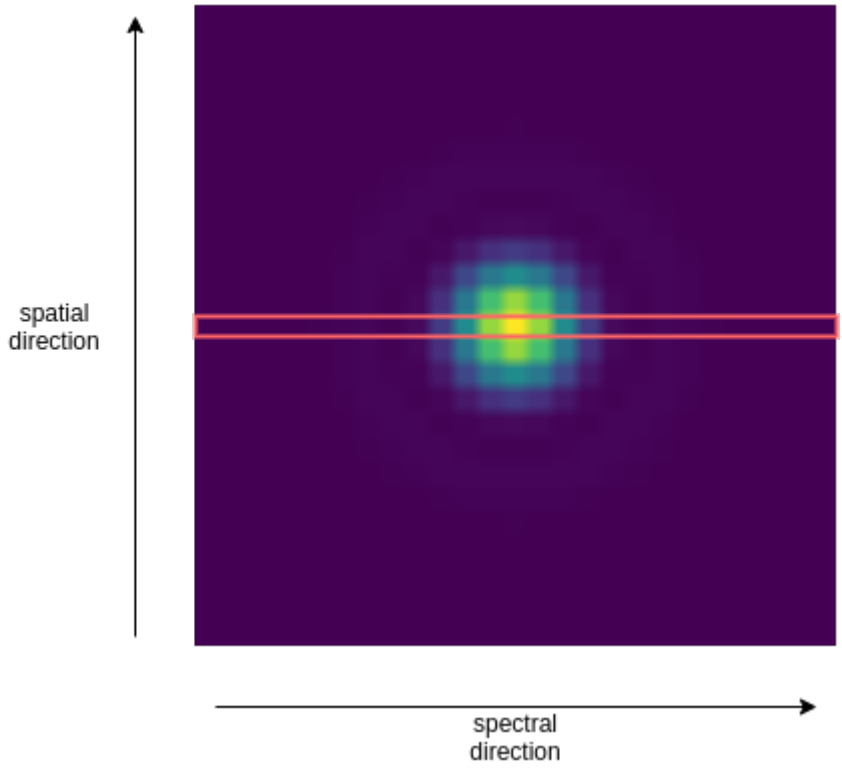
Spectral modes

Spectral modes specifies how the detector pixels counts are summed inm the two directions. These, are set with the *spectral_mode* keyword.

Rows

By setting the *spectral_mode* equal to *row*, the aperture sizes in the spectral direction are set such that the ful pixel row is summed together.

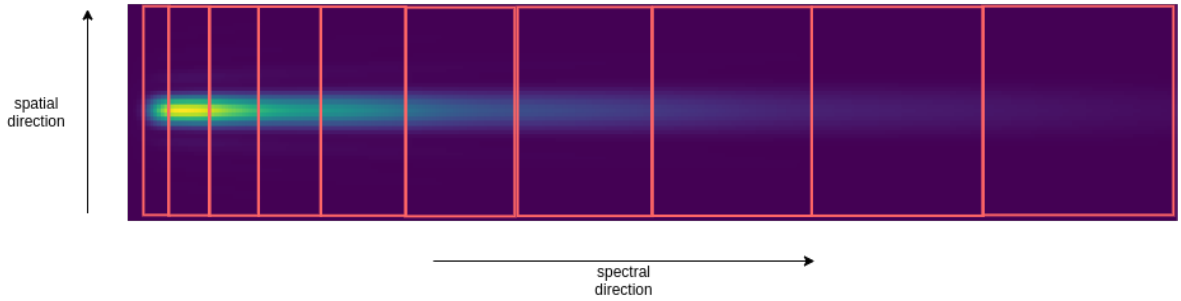
```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <spectral_mode> row </spectral_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Wavelength solution

By setting the *spectral_mode* equal to *wl_solution*, the aperture sizes in the spectral direction are estimated starting from the spectral bin size defined in the radiometric table (see *Wavelength binning*).

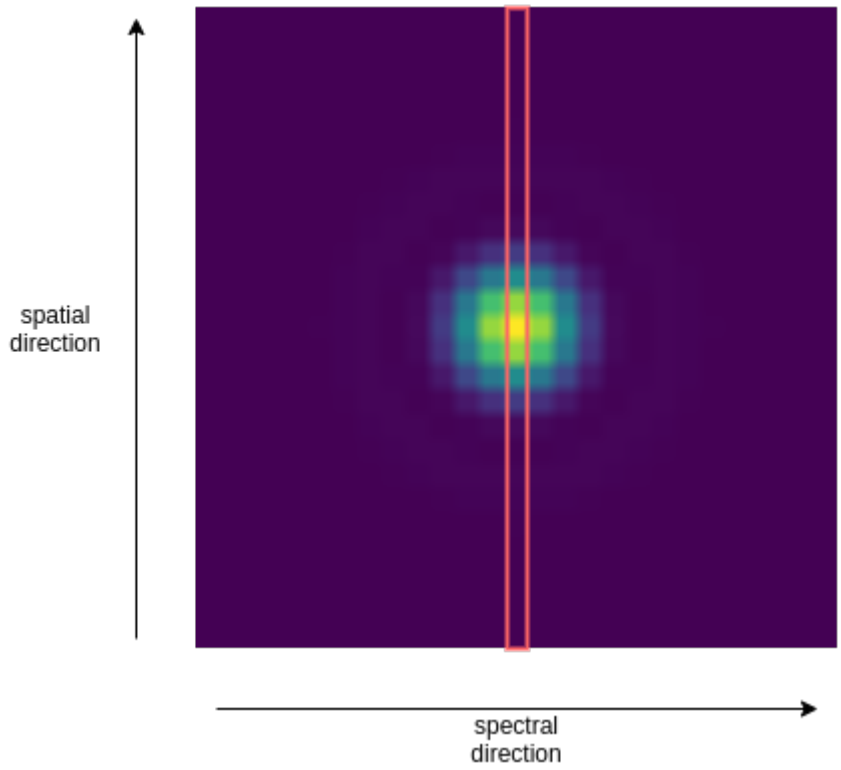
```
<channel> channel_name
    <type> spectrometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <spectral_mode> wl_solution </spectral_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Spatial modes

At the moment only a spatial mode is available. By setting the *spatial_mode* equal to *column*, the aperture sizes in the spatial direction are set such that the ful pixel column is summed together.

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <spatial_mode> column </spatial_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Use cases example

To summarise with a couple of examples, if the user wants to read a photometer by summing up all the pixel values, it can either use the automatic mode *full* (shown later)

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <auto_mode> full </auto_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```

or can specify the different methods in the two direction and ask to the *EstimateApertures* task to sum all the columns and rows:

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <spectral_mode> row </spectral_mode>
```

(continues on next page)

(continued from previous page)

```
            <spatial_mode> column </spatial_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```

If the user want to read a spectrometer by summing up all the pixel along the columns of a spectral bin, it can combine the *column* and *wl_solution* methods:

```
<channel> channel_name
    <type> spectrometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <spectral_mode> wl_solution </spectral_mode>
            <spatial_mode> column </spatial_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```

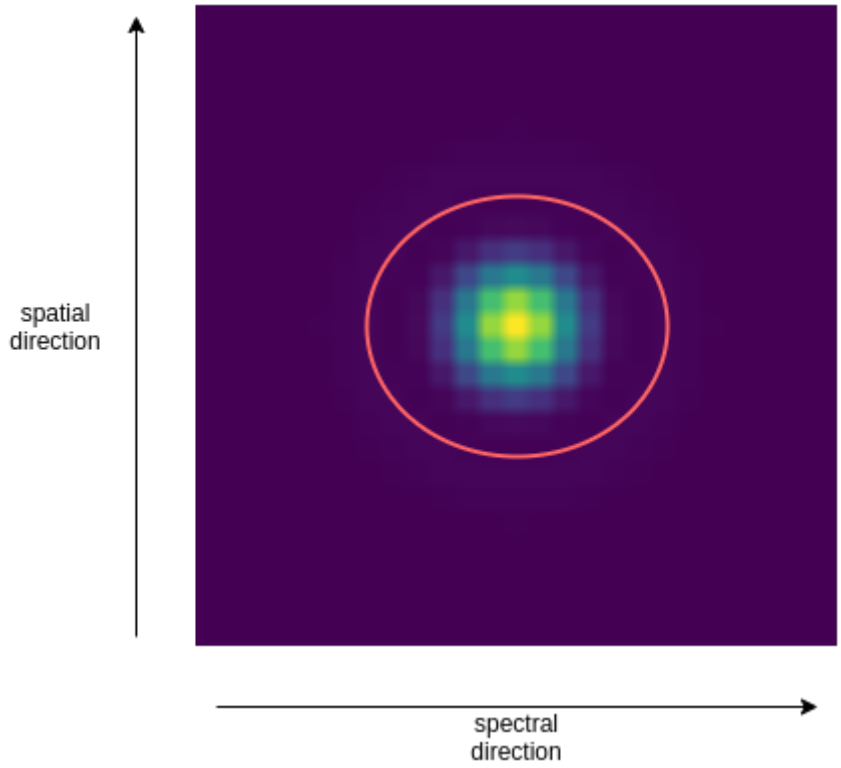
Automatic modes

The *EstimateApertures* task includes come automatic functionalities aimed to optimise the search for the right aperture.

Elliptical apertures

This mode can be set with *auto_mode* equal to *elliptical*. Using this method the *find_elliptical_aperture* is run. The function look for an elliptical aperture on the focal plane which enclose at least the Encircled Energy specify by the keyword *EnE*, while minimizing the number of pixel inside in the aperture area.

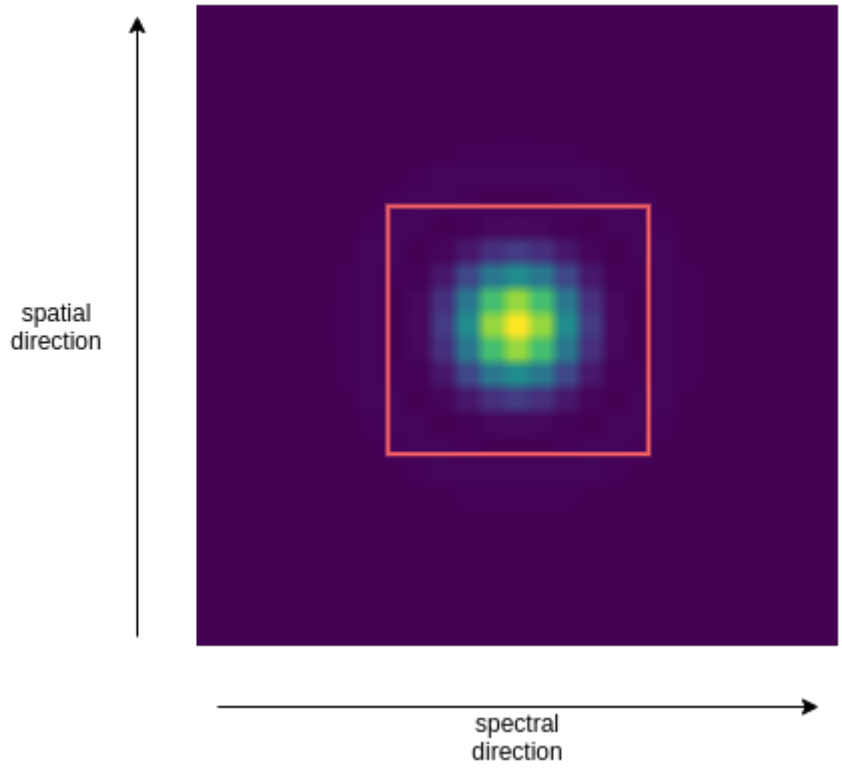
```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <auto_mode> elliptical </auto_mode>
            <EnE> 0.91 </EnE>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Rectangular apertures

This mode can be set with *auto_mode* equal to *rectangular*. Using this method the *find_rectangular_aperture* is run. The function look for a rectangular aperture on the focal plane which enclose at least the Encircled Energy specify by the keyword *EnE*, while minimizing the number of pixel inside in the aperture area.

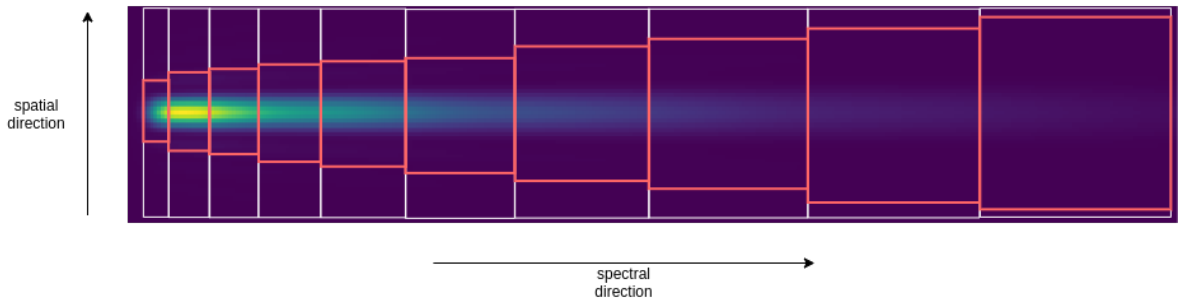
```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <auto_mode> rectangular </auto_mode>
            <EnE> 0.91 </EnE>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Spectral bins apertures

This mode can be set with *auto_mode* equal to *bin*. Using this method the *find_bin_aperture* is run. The function look for a rectangular aperture with fixed spectral size on the focal plane which enclose at least the Encircled Energy specify by the keyword *EnE*, while minimizing the number of pixel inside in the aperture area.

```
<channel> channel_name
    <type> spectrometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <auto_mode> bin </auto_mode>
            <EnE> 0.91 </EnE>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```



Full aperture

This mode can be set with *auto_mode* equal to *full* and create a rectangular aperture of the size of the focal plane.

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <aperture_photometry>
            <apertures_task> EstimateApertures </apertures_task>
            <auto_mode> full </auto_mode>
        </aperture_photometry>
        ...
    </radiometric>

</channel>
```

Estimate signals

There are different signals to estimate for the radiometric model, that depends on different focal planes:

- source
- foreground
- sub foregrounds

Source and Foreground signal

Source and foreground signal are estimated using aperture photometry in the same way and on the same apertures starting from their focal planes. They are estimated for each channel by using *ComputeSignalsChannel* task by default.

ComputeSignalsChannel needs a radiometric table with apertures listed and a focal plane, then it runs *AperturePhotometry* and returns its results.

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <signal_task> ComputeSignalsChannel </signal_task>
        ...
```

(continues on next page)

(continued from previous page)

```
    </radiometric>

</channel>
```

Inside *RadiometricModel* this tasks is handled by the *compute_source_signals* method for the source focal plane and by the *compute_foreground_signals* method for the foreground. To use the default task in a script on a channel the user can write:

```
import exosim.tasks.instrument as instrument

computeSignalsChannel = instrument.ComputeSignalsChannel()
photometry = computeSignalsChannel(table=table,
                                   focal_plane=focal_plane)
```

Where *table* is the wavelength radiometric table with apertures and *focal plane* is the channel source or foreground focal plane array.

Caution: If the user doesn't include the *signal_task* keyword in the channel description, the default *ComputeSignalsChannel* task is used. To develop a custom *Task*, please refer to *Custom Tasks*.

The default *ComputeSignalsChannel* task uses the apertures center, sizes and shapes in the radiometric table to perform aperture photometry with the appropriate apertures using photutils.aperture.aperture_photometry[23].

Foreground sub focal plane signals

If at least one of the foreground has the *isolate* option enable, there will be contributions to the focal plane to estimate for the radiometric table. As mentioned already in *Foreground sub focal planes*, these focal planes are stored in a dedicated directory. To estimate their contribution to the radiometric signal, a default *Task* has been developed: *ComputeSubFrgSignalsChannel*. As *ComputeSignalsChannel*, this task use *AperturePhotometry* to perform aperture photometry on the same apertures used for source and general foreground focal planes. This task should be indicated in the description document as

```
<channel> channel_name
    <type> photometer </type>

    <radiometric>
        <sub_frg_signal_task> ComputeSubFrgSignalsChannel </sub_frg_signal_task>
        ...
    </radiometric>

</channel>
```

Inside *RadiometricModel* this tasks is handled by the *compute_sub_foregrounds_signals* method. To use the default task in a script on a channel the user can write:

```
import exosim.tasks.radiometric as radiometric

computeFrgSignalsChannel = radiometric.ComputeSubFrgSignalsChannel()
```

(continues on next page)

[23] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry

(continued from previous page)

```
signal_table = computeFrgSignalsChannel(table=table,
                                        ch_name=ch,
                                        input_file=input,
                                        parameters=description)
```

Where *table* is th wavelength radiometric table with aperture, *ch_name* is the channel name, *input_file* is the input hdf5 file containing the focal planes, and *parameters* is the dictionary containing the aperture photometry information from the *xml* file.

Caution: If the user doesn't include the *sub_frg_signal_task* keyword in the channel description, the default *ComputeSubFrgSignalsChannel* task is used. To develop a custom *Task*, please refer to *Custom Tasks*.

Saturation time

The saturation time is estimated in each channel by *SaturationChannel*. This *Task* sums the source and the foreground focal plane and look for the maximum and minimum signals. From the detector details, it then estimates the saturation time. The channel description must include the well depth and the fraction of well depth to use:

```
<channel> channel_name

    <detector>
        <well_depth> 100000 </well_depth>
        <f_well_depth> 0.9 </f_well_depth>
    </detector>

</channel>
```

The saturation time is estimated as the well depth divided by the maximum signal in a pixel of the array. Then the integration time is the saturation time multiplied by the fraction of the well depth to use.

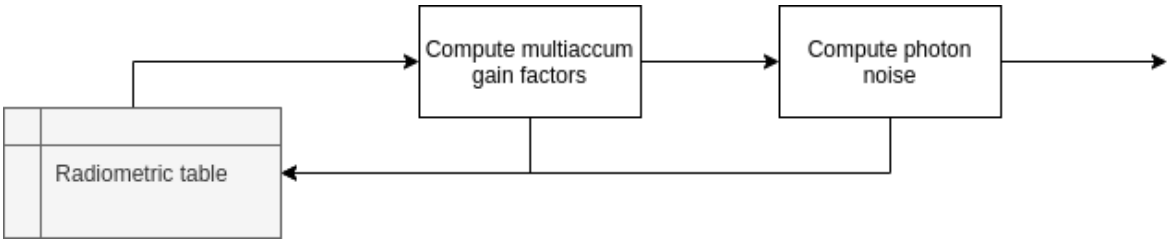
Inside *RadiometricModel* this tasks is handled by the *compute_saturation* method. To use the default task in a script on a channel the user can write:

```
import exosim.tasks.radiometric as radiometric

saturationChannel = radiometric.SaturationChannel()
sat, t_int, max_sig, min_sign = saturationChannel(table=table,
                                                  description=description
                                                  input_file=input)
```

Where *table* is the wavelength radiometric table with aperture, *ch_name* is the channel name, *description* is the dictionary containing the channel information from the *xml* file, and *input_file* is the input hdf5 file containing the focal planes.

2.3.2 Estimate the noise

Once the radiometric table is built, the next step is to estimate the noise. Numerous noise sources can be considered. In the following we describe ho to use the standard sources and how to include new or custom ones.

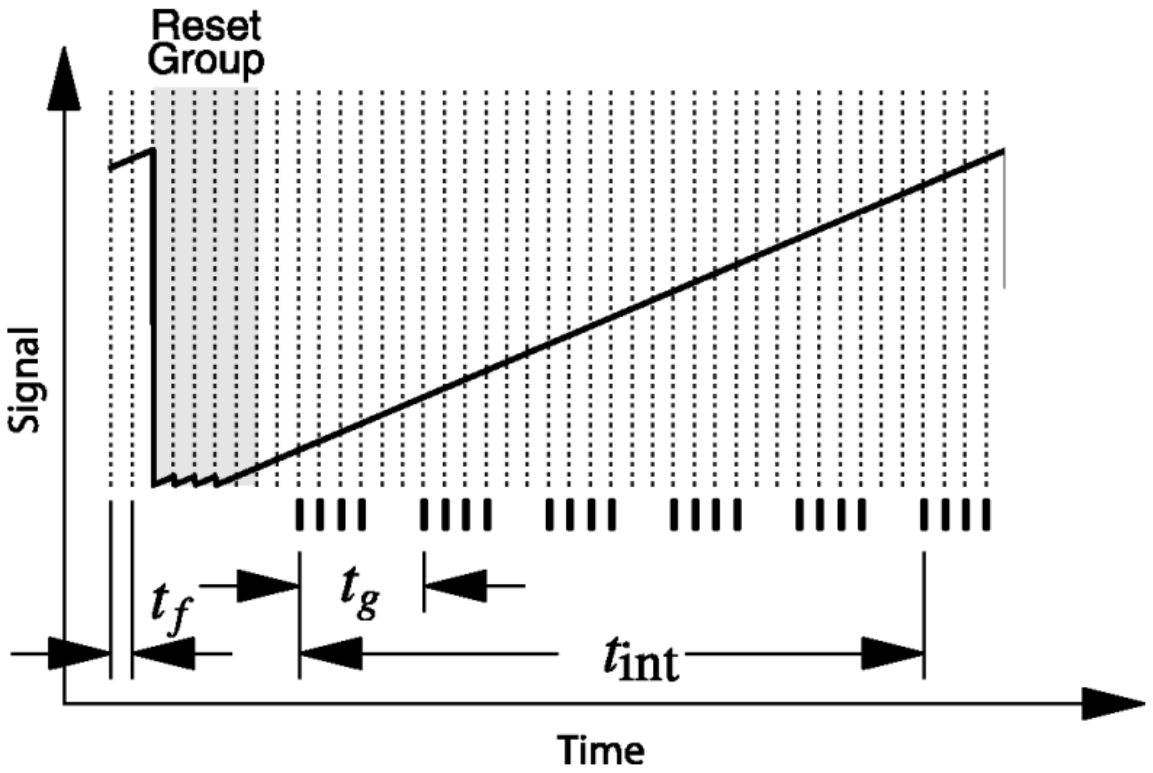


Multiaccum

Warning: Multiaccum factors are not validated yet.

When the detector is collecting light, the pixel are being filled with electrons. During time the number of electrons in the pixels increases until eh pixel saturates. This saturation ramp hold the information abut the incoming flux.

Different way to fit this ramp results in different noises. This analysis has been done in Rauscher and Fox et al. 2007 (http://iopscience.iop.org/article/10.1086/520887/pdf) with the definition of the MULTIACCUM equation. Such equation has been later corrected in Robberto 2009 (https://www.stsci.edu/files/live/sites/www/files/home/jwst/documentation/technical-documents/_documents/JWST-STScI-001853.pdf), and also reported in Batalha et al. 2017 (https://doi.org/10.1088/1538-3873/aa65b0).



The picture from Rauscher and Fox et al. 2007 (http://iopscience.iop.org/article/10.1086/520887/pdf) shows the detector readout scheme using MULTIACCUM sampling. The detector is read out at a constant cadence of t_f but not all the read frames are saved. Saved frames are averaged, resulting in one averaged group of data every t_g seconds.

The resulting total noise, as reported in Batalha et al. 2017 (https://doi.org/10.1088/1538-3873/aa65b0) is

$$\sigma_{tot}^2 = \frac{12(n-1)}{mn(n+1)} \cdot \sigma_{read}^2 + \frac{6(n^2+1)}{5n(n+1)}(n-1)t_g \cdot S + \frac{2(m^2-1)(n-1)}{mn(n+1)}t_f \cdot S$$

where m is the number of frames per group, n is the number of groups, σ_{read} is the read noise and S is the incoming photon flux.

To summarise, different fitting patters produce different gain factors for read noise and photon noise. We can then determine two gain factors: one for the read noise and one for the photon noise.

$$gain_{read} = \frac{12(n-1)}{mn(n+1)}$$

$$gain_{phot} = \frac{6(n^2+1)}{5n(n+1)}(n-1)t_g + \frac{2(m^2-1)(n-1)}{mn(n+1)}t_f$$

In *ExoSim* these gain factor are estimated by `Multiaccum`. Enable this option the user needs to specify the factor inside the *xml* file:

```
<channel> channel_name
    <radiometric>
        <multiaccum>
            <n>  </n>
            <m>  </m>
```

(continues on next page)

(continued from previous page)

```
            <tg unit='s'> </tg>
            <tf unit='s'> </tf>
        </multiaccum>
        ...
    </radiometric>
</channel>
```

And then the gain factors are estimated as

```
import exosim.tasks.radiometric as radiometric

estimateApertures = radiometric.Multiaccum()
gain_read, gain_shot = multiaccum(parameters=description['radiometric']['multiaccum
↪'])
```

Photon Noise

For each signal in the radiometric table is possible to compute the photon noise. The photon noise is computed by *ComputePhotonNoise*

Given the incoming signal S the resulting photon noise variance is $Var[S] = S$.

If photon gain factor $gain_{phot}$ has been computed with multiaccum equation (see *Multiaccum*), then $Var[S] = gain_{phot} \cdot Var[S]$.

The user can also add a margin to the photon noise as

```
<channel> channel_name
    <radiometric>
        <photon_margin> 0.4 </photon_margin>
    </radiometric>
</channel>
```

If photon noise margin, χ, is found in the description, then $Var[S] = (1 + \chi) \cdot Var[S]$. The noise returned is $\sigma = \sqrt{Var[S]}$

For each channel can be run in a script as

```
import exosim.tasks.radiometric as radiometric

computePhotonNoise = radiometric.ComputePhotonNoise()
phot_noise = computePhotonNoise(signal=table['signal_name'],
                                description=description,
                                multiaccum_gain=table['multiaccum_shot_gain'])
```

Total Noise

The total relative noise is estimated for a $1\,hr$ time scale observation. Therefore it has units of $\sqrt{hr}$.

The task dedicated to this job is *ComputeTotalNoise*.

We start from an empy array of variances :math:` Var_{1 , hr}(lambda)`.

This tasks iterates over the column of the radiometric table looking for noise sources. If a column name contains the word *noise* then is handled by this task. Assuming that the code founds a column name *X_noise*. If the column *X_noise* units are ct/s then

$$Var_{1\,hr}(\lambda) = Var_{1\,hr}(\lambda) + \frac{[\sigma_X(\lambda)]^2}{\Delta T_{int}}$$

where $\Delta T_{int} = 3600\,s$ for the $1\,hr$ integration time.

When the noise from all the columns that has such units has been added, the total variance is converted into relative noise.

Note: To avoid confusion, only the noise from the source and the cumulative foreground are added to the total noise.

Then the relative noise is

$$\sigma_{1\,hr}(\lambda) = \frac{Var_{1\,hr}(\lambda)}{S_{source}(\lambda)}$$

where S_{source} is the source signal in the radiometric table.

Assuming that the code founds also a column name *Y_noise* and that the column *Y_noise* has no units. This is a relative noise already. So, the code updates the total noise as

$$\sigma_{1\,hr}(\lambda) = \sqrt{[\sigma_{1\,hr}(\lambda)]^2 + [\sigma_Y(\lambda)]^2}$$

When also this check is concluded, the total relative noise is added to the radiometric table.

To run the task from script

```
import exosim.tasks.radiometric as radiometric

computeTotalNoise = radiometric.ComputeTotalNoise()
total_noise = computeTotalNoise(table)
```

2.3.3 Automatic recipies

Finally, an automatic pipeline has been developed to help the user. The pipeline is handeld by a *recipes* of *ExoSim* called *RadiometricModel*. In the following we described the different cases handled by the model.

Radiometric model automatic Recipe

As for the focal plane recipe described in *Focal plane automatic Recipe*, ExoSim include an automatic pipeline also for the radiometric model. This is under the `recipes` of *ExoSim*.

```
from exosim import recipes
recipes.RadiometricModel(options_file='your_config_file.xml',
                         input_file='output_file.h5')
```

The `RadiometricModel` can also be run from console as

```
exosim-radiometric -c your_config_file.xml -o output_file.h5
```

or

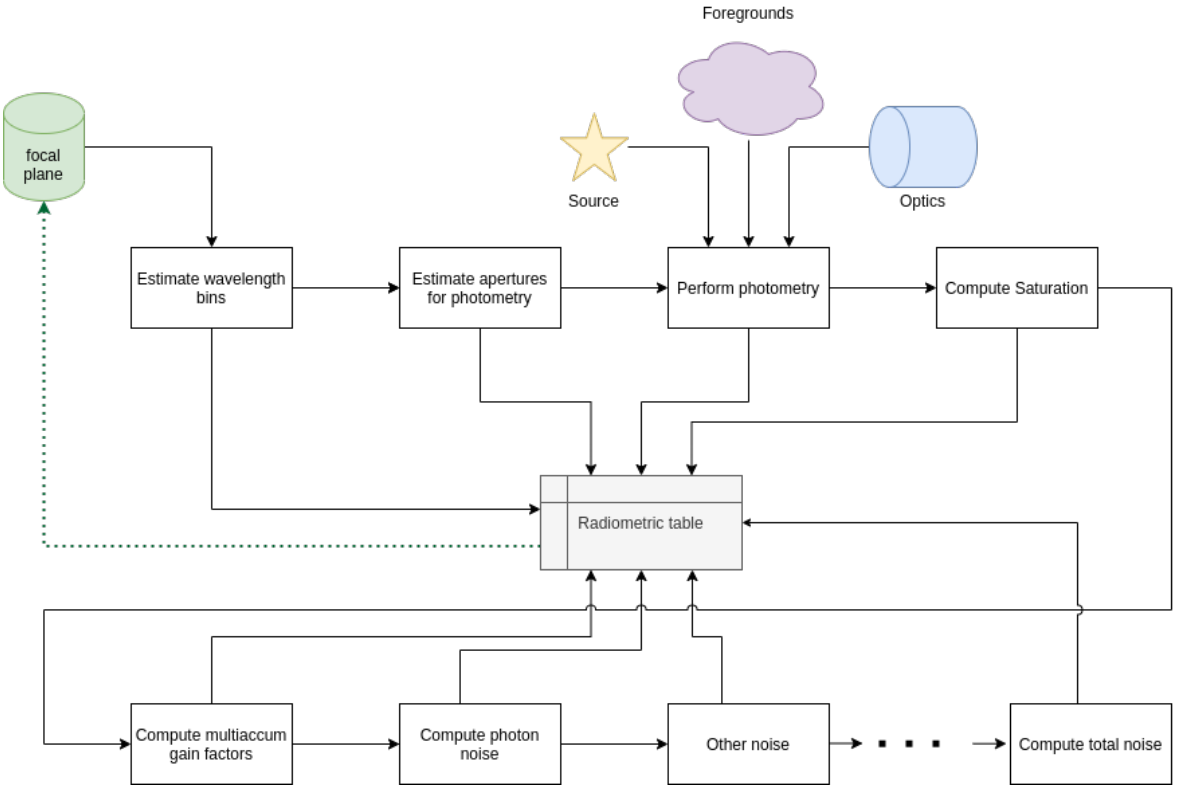
```
exosim-radiometric -c your_config_file.xml -o output_file.h5 -P
```

to also run ExoSim `RadiometricPlotter`, which is documented in *Radiometric Plotter*.

The radiometric pipeline stores the product in the output file by default using `exosim.recipes.radiometricModel.RadiometricModel.write`. If the output format is the default HDF5[24], refer to *How can I load HDF5 data into my code?* in the *FAQs* section for how to use the data, and see *Load signals and tables* in particular to cast the focal plane into a `Signal` class.

Existing focal plane

If a focal plane has been computed already and is available as input file, then the working scheme for the recipe is reported in the following figure:



[24] https://www.hdfgroup.org/solutions/hdf5/

Starting from the focal plane, all the steps are the ones reported previously in this documentation: The involved steps are:

1. creation of the wavelength table with `create_table` (see *Wavelength binning*);
2. estimation of the apertures sizes and number of pixels involved with `compute_apertures` (see *Estimate apertures*);
3. estimation of the signals in the apertures for the sub foregrounds, if any: `compute_sub_foregrounds_signals` (see *Estimate signals*);
4. estimation of the total foreground signal in the apertures: `compute_foreground_signals` (see *Estimate signals*);
5. estimation of the source focal plane signal in the aperture: `compute_source_signals` (see *Estimate signals*);
6. estimation of the saturation time in the channel: `compute_saturation` (see *Saturation time*);

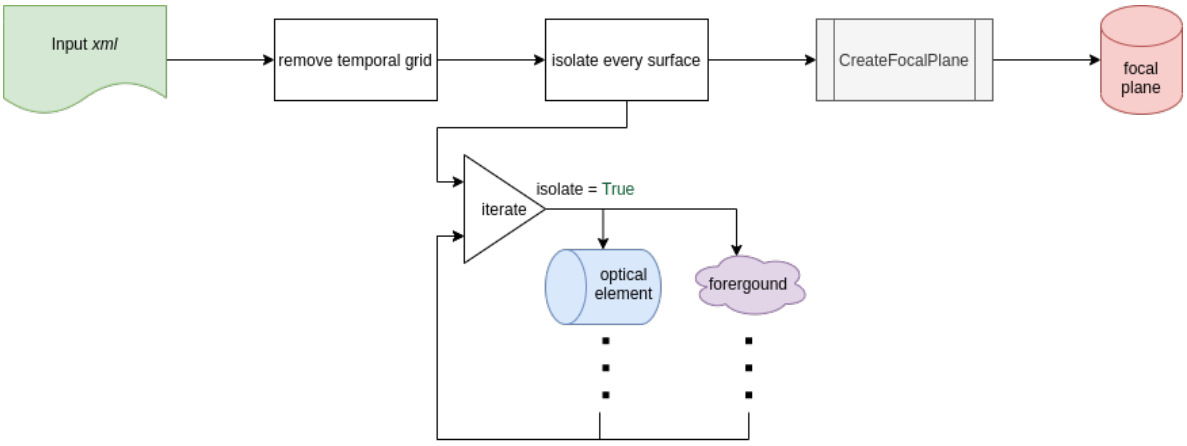
Then the noise estimation is run:

1. estimation of the multiaccum factors `compute_multiaccum` (see *Multiaccum*);
2. estimation shot noise `compute_photon_noise` (see *Photon Noise*);
3. update total noise `update_total_noise` (see *Total Noise*)

Then the radiometric table is stored into the input file.

Non existing focal plane

If a focal plane is not available as input, the `RadiometricModel` creates it.



Following the figure, the pipeline first loads the input configuration *xml* file. Then it removes the temporal dimension, as the radiometric model won't need it. It isolate every optical element, such that it can estimate their contributions, and finally creates the focal plane using *Focal plane automatic Recipe*.

Then, from the new focal plane the *Existing focal plane* pipeline is run.

Radiometric model over a target list

Not implemented yet

Where the following cases are addressed:

- *Existing focal plane*
- *Non existing focal plane*
- *Radiometric model over a target list*

2.4 Sub-Exposures

The second step af an *ExoSim* simulation is the creation of the *sub-exposures* starting from instrument focal planes. With *sub-exposures* here we are referring to the production of focal planes sampled at the same cadence of NDRs.

To better understand the idea of sub-exposures, we first need to discuss a little the detector ramp sampling. While the photons arrive to the focal plane, the detector pixels start converting them into electrons anc collecting them. We can think of each pixel as an accumulator of charge. So, during time, the collected charges increases. When the accumulator is full, a reset process empties it to collect charges again. The focal plane is read during the charge accumulation, producing a certain number of NDRs, according to the multiaccum reading scheme enabled. This procedure assume an instantaneous readout of the detector. Such read focal planes, are called here *sub-exposures*. The difference between *sub-exposures* and NDRs are the fact that the sub-exposures of the same ramp are not summed together, as will be for the NDRs production, and the detector effects, which are involved in the production of NDRs. These are added later, as a last step in *ExoSim 2.0*. So, each sub-exposures represent an integrated image of what happened between the previous detector action and the sub-exposure time.

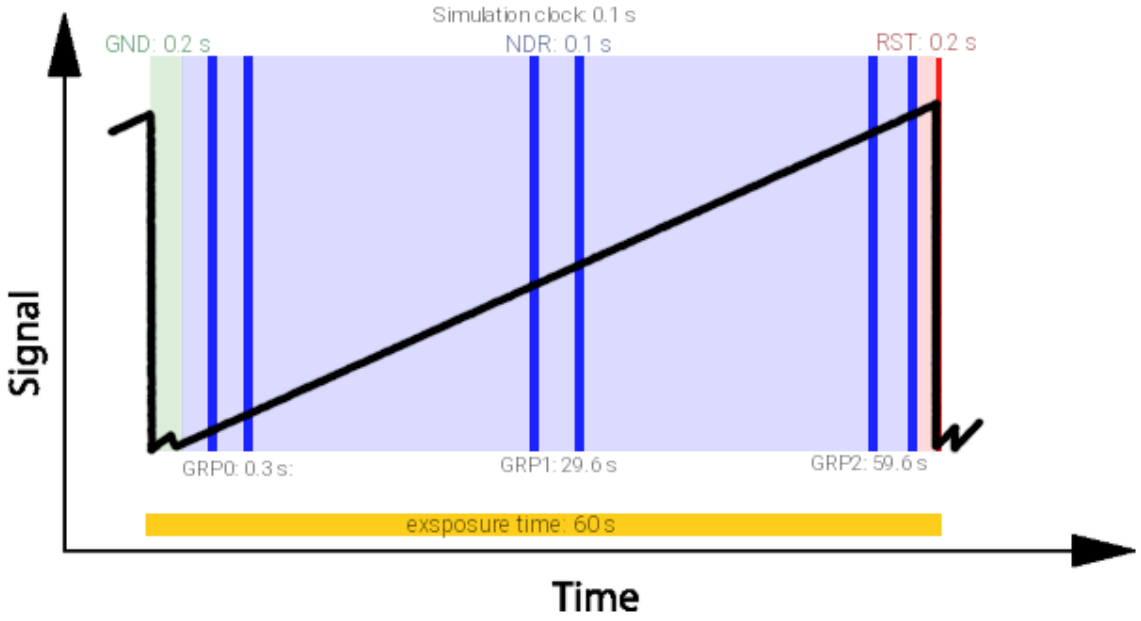
Note: In this model we are only considering *instantaneous read out* of the detector.

To produce these *sub-exposure* a simulation cadence is needed which refer to an higher frequency than the one used to sample the focal plane. This cadence can be set in the channel configuration file as *readout_frequency*:

```
<channel> channel name
    <readout>
        <readout_frequency unit ='s'> 0.1 </readout_frequency>
    </readout>
</channel>
```

The user can also set the *readout_frequency* in units of Hz instead of s.

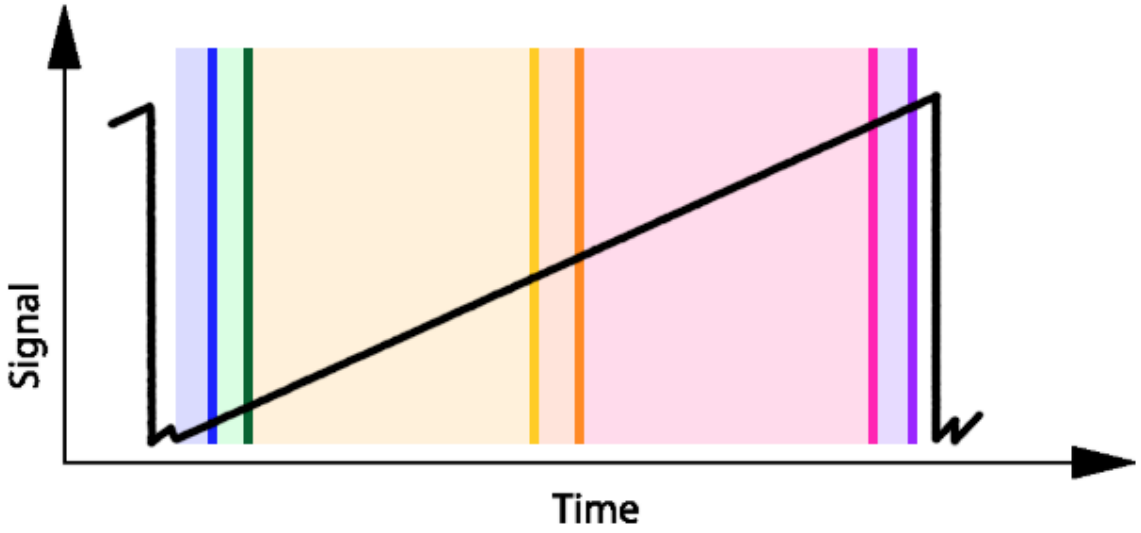
Let's seen an example. In the following figure is reported a reading scheme where we consider a mid frequencies resolution of 0.01 second, which here is called *simulation clock*. The other quantities in the figure are written as number of simulation clock units. The detector saturates in $60\,s$, which is the exposure time. This ramp is sampled using 3 groups of 2 NDRs each. For the first $0.2\,s$ the detector is kept in ground state (GND), these corresponds to 2 simulation clock units. We also know that for the last $0.2\,s$ (2 simulation clock units), the detector is in reset mode (RST). So, the available time to sample the ramp is $60 - 0.2 - 0.2 = 59.6\,s$, corresponding to 596 simulation clock units. Because the detector reading cadence is $0.1s$, after $0.1\,s$ (1 simulation clock unit), the first sub-exposure of the first group is read. After other $0.1\,s$ the second sub-exposure is read too. In every groups the two sub-exposures are separated by 1 simulation clock units. The groups are then separated by $29.6\,s$ (296 simulation clock units). This example is described in this picture, adapted from Rauscher and Fox et al. 2007 (http://iopscience.iop.org/article/10.1086/520887/pdf)



Where are reported the duration of states on the top, and the start of each group on the bottom. We'll describe later (*Reading Scheme*) how to design such reading scheme, but this numbers have been estimated using one of the *ExoSim Tools*: *Readout Scheme Calculator*.

Each of the simulation clock units corresponds to a different realisation of the focal plane, because of the pointing Jitter. All of these realisations are considered in the simulation. Schemes like this will be further investigated later, when we'll discuss how *ExoSim* handles reading schemes.

With reference to the previous figure, we can investigate the concept of *sub-exposures*, using the following image, where each color represents the area collected in a different sub-exposure.



The sub-exposures creation is automatised by a recipe: `CreateSubExposures`. In this section we explain each of the steps that lead to the sub-exposures creation.

2.4.1 Preparing the Pointing Jitter

Instrument Pointing Jitter

First, we need to simulate the instrument pointing jitter. As mentioned in *Sub-Exposures*, the jitter is sampled to the mid frequencies time scale, so the first step is to define this quantity:

```
<root>
    <time_grid>
        <start_time unit="hour">0.0</start_time>
        <end_time unit="hour">10.0</end_time>
        <low_frequencies_resolution unit="second">60.0</low_frequencies_resolution>
    </time_grid>
</root>
```

With this configuration we are simulating 10 hours of observation, with low frequencies variation sampled at 1 minute cadence, and mid frequencies effects sampled at 0.01 seconds cadence.

Then, still in the main configuration file, we need to describe the jitter under the *jitter* keyword:

```
<root>
    <jitter>
        ...
    </jitter>
</root>
```

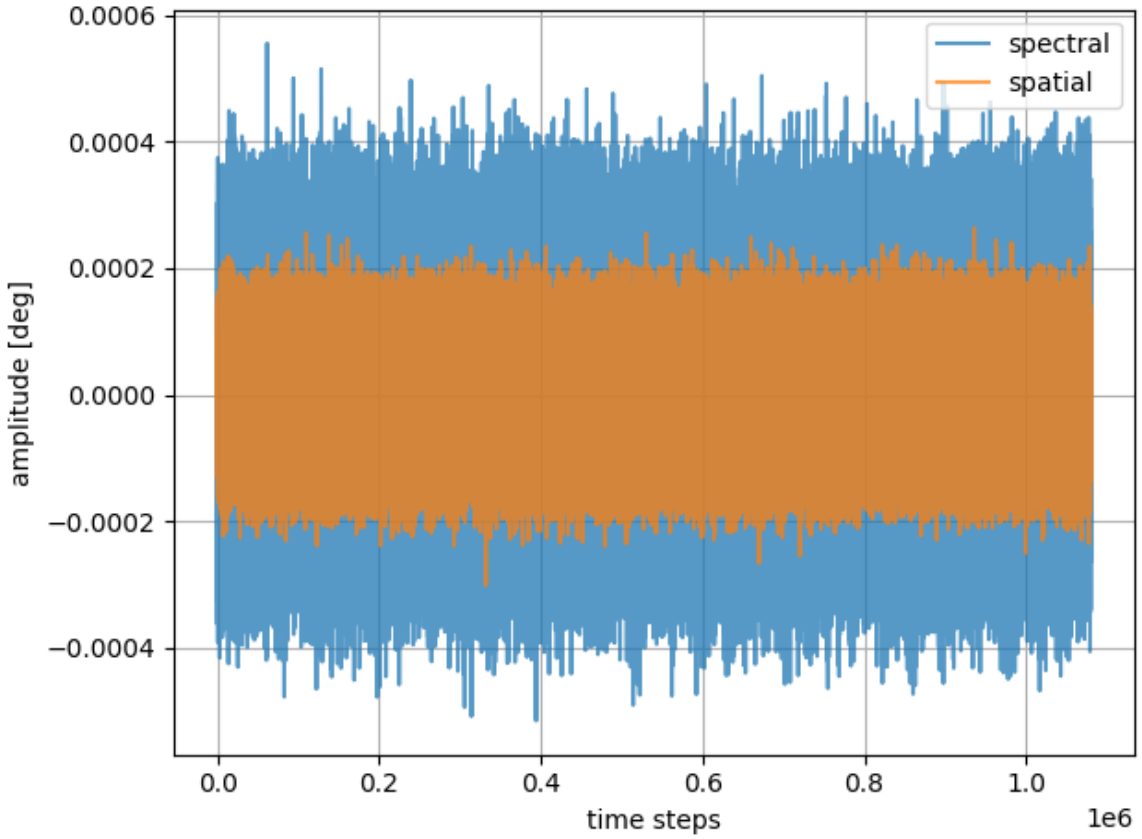
Here the first key to define is the *jitter_task*, which specify which jitter *Task* we want to use. To learn about how to customised *Task*, please refer to *Custom Tasks*. By default, we use *EstimatePointingJitter*. This class randomly build the jitter in the spectral and spatial directions expressed as *deg*, starting from the input standard deviations:

```
<jitter>
    <jitter_task> EstimatePointingJitter </jitter_task>
    <spatial unit="arcsec"> 0.2 </spatial>
    <spectral unit="arcsec"> 0.4 </spectral>
    <frequency_resolution unit="Hz"> 100 </frequency_resolution>
</jitter>
```

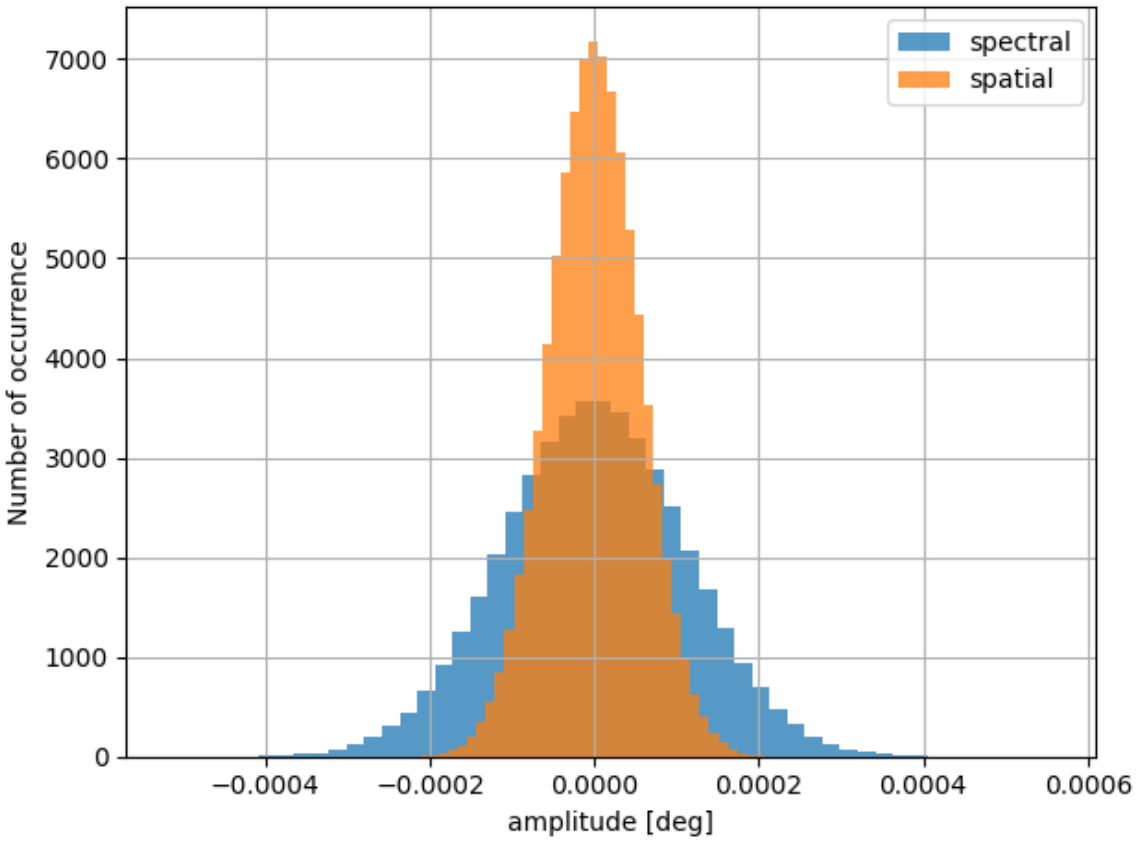
Using this configuration the resulting jittered positions will result as:

```
import exosim.tasks.subexposures as subexposures
estimatePointingJitter = subexposures.EstimatePointingJitter()
jitter_spa, jitter_spe, jitter_time = estimatePointingJitter(parameters=main_
↪parameters)
```

where *main_parameters* is the parameter dictionary from the main configuration file.



Which are distributed as



This is the same as run the class with the configuration

Channel Pointing Jitter

Once the instrument pointing jitter is computed, it is shared between all the channels. Because each channel has a different plate scale (see also *Telescope pointing and multiple sources*), we now need to rescale the pointing jitter to the channel pixel. This is handled by `EstimateChJitter`, which computes the angle of view of each sub-pixel of the focal plane and convert the instrument pointing jitter, expressed as deg, to units of sub-pixels.

Assuming the instrument jitter has been already computed, and the channels plate scales are in the parameter dictionary

```
<channel> Photometer
    <type> photometer </type>
    <detector>
        <plate_scale unit="arcsec/micron"> 0.01 </plate_scale>
    <detector>
    <readout>
        <readout_frequency unit="Hz">100</readout_frequency>
    </readout>
</channel>

<channel> Spectrometer
    <type> spectrometer </type>
    <detector>
```

(continues on next page)

(continued from previous page)

```
        <plate_scale>
            <spatial unit="arcsec/micron"> 0.01 </spatial>
            <spectral unit="arcsec/micron"> 0.05 </spectral>
        </plate_scale>
    <detector>
    <readout>
        <readout_frequency unit="Hz">100</readout_frequency>
    </readout>
</channel>
```

Than *EstimateChJitter* can be run as

```
import exosim.tasks.subexposures as subexposures
estimateChJitter = subexposures.EstimateChJitter()
jit_y, jit_x, jit_time = estimateChJitter(parameters = parameters,
                                          pointing_jitter=(jitter_spa,
                                                           jitter_spe,
                                                           jitter_time))
```

This will results in a list of jitter offsets in pixel units sampled at a multiple of the channel *readout_frequency* cadence, for the ful length of the observation.

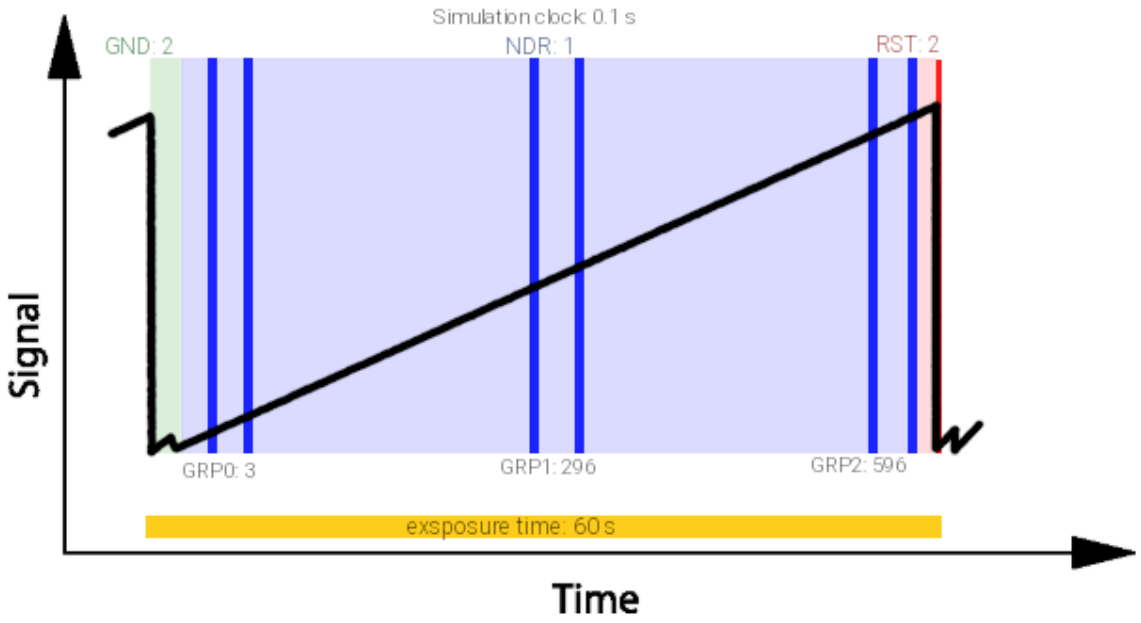
The new jitter time line, *jit_time* might be different from the previous *jitter_time*, and different from channel to channel. The new time line is estimated using the lowest multiple shared between the channel *readout_frequency* and the frequency used to sample the input jitter. This will result in *ExoSim* oversampling in frequency the detector readout to ensure that the input jitter is well represented and aligned to the detector readout scheme.

2.4.2 Reading Scheme

Once the jitter time lines are ready, we need to define the reading scheme for the detector.

Note: In this model we are only considering *instantaneous read out* of the detector.

Assuming we want to reproduce the following reading scheme, which is the same of *Sub-Exposures*, where a 60 s exposure time is sampled by 6 NDRs divided in 3 groups.



The ramp is sampled at *readout_frequency* cadence, defined in *Sub-Exposures*. In this example we want to spend $0.2\,s$ in ground (GND) state and $0.2\,s$ before the reset state (RST). The NDRs are read at a constant cadence of $0.1\,s$. Then we want to have 3 groups which divide the residual ramp into equal parts. Each group consists of 2 NDRs separated by 1 simulation steps, which is the time needed to read a NDR. We can translate all of this as in the following description in the channel configuration file:

```
<channel> channel name
    <readout>
        <readout_frequency unit="Hz">10</readout_frequency>
        <n_NRDs_per_group> 2 </n_NRDs_per_group>
        <n_groups> 3 </n_groups>
        <n_sim_clocks_Ground> 2 </n_sim_clocks_Ground>
        <n_sim_clocks_first_NDR> 1 </n_sim_clocks_first_NDR>
        <n_sim_clocks_Reset> 2 </n_sim_clocks_Reset>
        <n_sim_clocks_groups> 2996 </n_sim_clocks_groups>
    </readout>
</channel>
```

The user can also set the *readout_frequency* in units of Hz instead of s.

The reading scheme is computed by *ComputeReadingScheme*

```
import exosim.tasks.subexposures as subexposures
computeReadingScheme = subexposures.ComputeReadingScheme()
clock, base_mask, base_group_end, base_group_start, number_of_exposures =␣
↪computeReadingScheme(
    parameters=parameters,
    main_parameters=main_parameters,
    focal_plane=focal_plane,
    frg_focal_plane=frg_focal_plane)
```

The outputs of this `Task` can be confusing, because is written to optimise the next step in the sub-exposures procedure. In the following we discuss each of them.

- `clock`: this is the simulation frequency, which is the inverse of *high_frequencies_resolution* defined in

Sub-Exposures;

- `base_mask`: this is state machine for the reading operation on the ramp. In fact, a ramp is made of different states: ground state (GNS), reset state (RTS) and read states (NDR). This mask is a list of of 0 and 1, where 1 is for the steps indicating a read operation: Referring to the previous image, the base will look like [0, 1, 1, 1, 1, 1, 1, 0].
- `frame_sequence`: this is the full list of simulation stapes for each steps on the ramp repeated by the number of ramps. E.g. [2, 1, 1, 296, 1, 296, 1, 2].
- `number_of_exposures`: this is the number of exposures needed to sample the full observation using ramps of the exposure time size. To estimate this quantity, the *Task* compute the integration time using *ComputeSaturation*, which is why it need the focal planes.

For testing reasons, because sampling the full observation can be long and produce a lot of sub-exposure, the user can force the number of exposure to use by

```
<channel> channel name
    <type> channel type </type>
    <readout>
        <n_exposures> 2 </n_exposures>
    </readout>
</channel>
```

Note: To help the user in defining the detector reading scheme, *ExoSim* include a dedicated tool: *Readout Scheme Calculator*.

The readout scheme along with all the information needed for the instantaneous readout is computed by `PrepareInstantaneousReadOut`.

2.4.3 Instantaneous readout

We now have all the elements to simulate the instantaneous readout.

Focal plane Sub-Exposures

This is handled by *InstantaneousReadOut* task.

This *Task*, first calls *ComputeReadingScheme*, described in *Reading Scheme*, then it scales the jitter to the indicated channel focal plane, using *EstimateChJitter*, described in *Channel Pointing Jitter*.

Preparing the output datacube

Then, *InstantaneousReadOut* builds the output, which will have $counts$ units, and so it is a *Counts* class. Because the resulting datacube size can grow fast, *InstantaneousReadOut* initialises it as a cached *Signal* class, which is described in *Cached signals*. The time dimension of this *Signal* class contains the acquisition times of each sub-exposures. So, this datacube has for each time step the shape of the focal plane without considering the oversampling factor (see *Detector geometry*), and a time step for each sub-exposure expected, which corresponds to the total number of NDRs listed in the reading scheme. This means that if each ramp is sampled using 3 groups of 2 NDRs, we have 6 NDRs for each ramp, and therefore 6 sub-exposures. Saying that each ramp last 60 s, to sample 8 hr of observation requires 480 ramps, and this results in 2880 sub-exposures. As mentioned, this number can grow quite fast: for a bright target the saturation time can go down for example to 1 second. Assuming this is the size of the ramp, we have 172800 sub-exposure. Given that the sub-exposures are

stored using `float64` data format (64 bits = 8 Bytes memory size), the size of this datacube, for a focal plane of 64×64 pixels is $172800 \cdot 64 \cdot 64/8 = 88.473.600$ Bytes. The resulting datacube is around 84 MB. This justifies the use of cached data, which are stored stored and used in chunks. The chunk size is set to 2 MB by default, but the user can set its own value as.

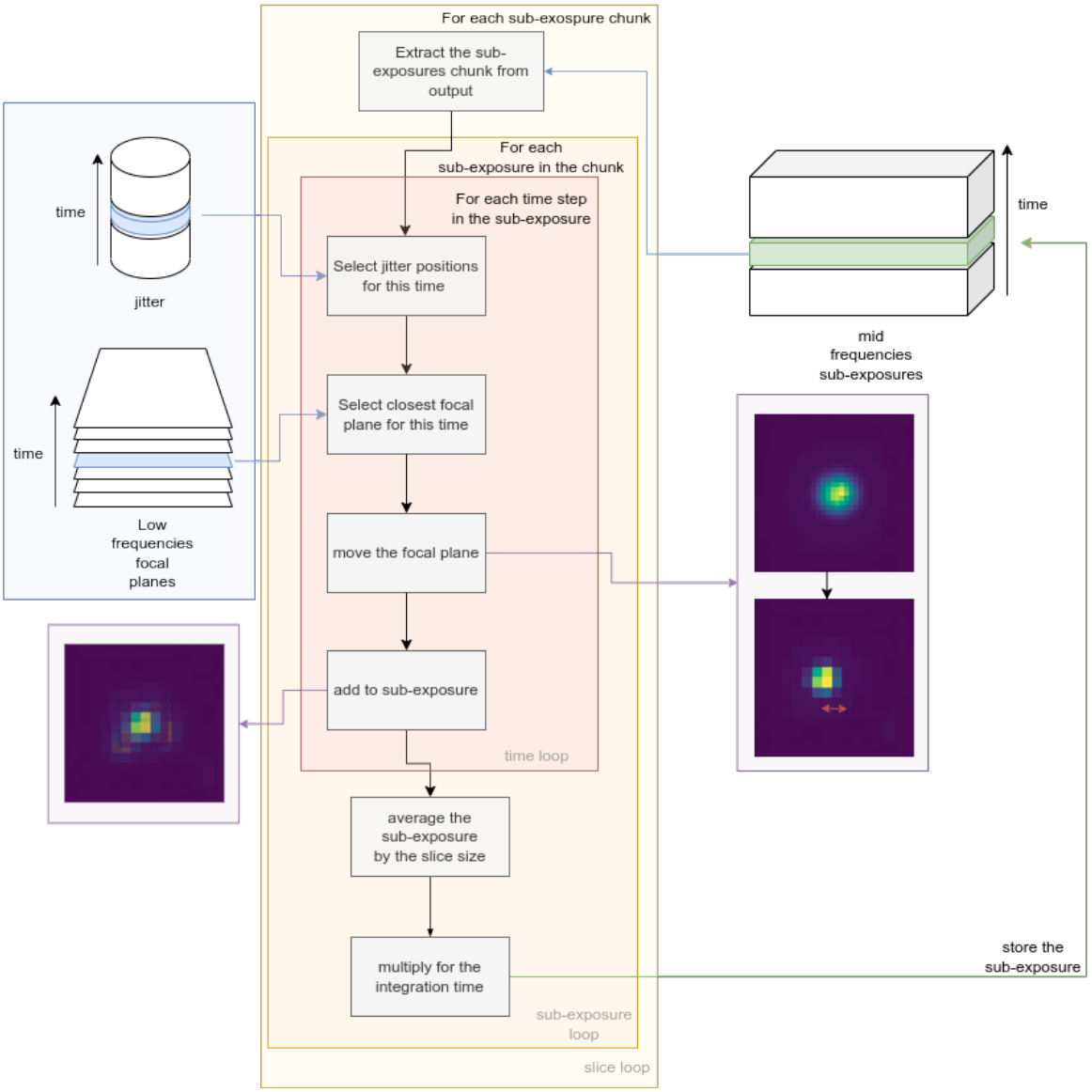
```
RunConfig.chunk_size = N
```

where *N* is the desired size expressed in Mbs.

Note: The use of `float64` data format, instead of `float32` is justified by the numerical precision required to handle the convoluted focal plane. Reducing the data format to `float32` would result in a loss of precision, which would be propagated to the final results as a non-conservation of the total incoming power.

Fill the output datacube

The main steps of the instantaneous readout process are summarised in the following figure.



Once the output is ready, *ExoSim* iterates over the chunks, thanks to the `h5py.Dataset`[25] methods (see also *Cached signals*), extracting a slice of sub-exposure. For each of the sub-exposures this slice there are a set of simulation time steps of *high_frequencies_resolution* unit associated. For each of this time step we recover the associated jitter offsets in the spectral and spatial directions. We select the low frequencies sampled focal plane corresponding to the time step considered. We remove the focal plane oversampling factor by shifting the focal plane by the offset quantity. Because the focal plane is sampled at a different cadence than the Sub-Exposures, in the Sub-Exposure signal is included a new key in the metadata: *focal_plane_time_indexes*. This array is as long as the sub-exposure temporal axis, and for each time step is reported the time index of the focal plane used for that sub-exposure.

Note: Here is where the oversampling factor comes into play. If the oversampling factor is smaller than jitter amplitude in pixel size, the jitter does not effect the final product of the simulation. It is important to calibrate the oversampling factor to the expected jitter amplitude in the channel.

Then, all the jittered focal planes associated to the same sub-exposure are averaged. The resulted sub-exposure

[25] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset

is then multiplied by its integration time, moving from the ct/s of the focal planet to the ct of the sub-exposure, and it is stored back to the output datacube.

This is performed with

```
import exosim.tasks.subexposures as subexposures

instantaneousReadOut = subexposures.InstantaneousReadOut()
se_out, integration_times = instantaneousReadOut(
                        main_parameters=main_parameters,
                        parameters=payload_parameters['channel'][ch],
                        focal_plane=focal_plane,
                        frg_focal_plane=frg_focal_plane,
                        pointing_jitter=(jitter_spa, jitter_spe),
                        output_file=output_file)
```

Where `main_parameters` is the the main configurations dictionary, `payload_parameters` is the payload configuration dictionary and `ch` is the channel name. `focal_plane` and `frg_focal_plane` are the focal plane and the foreground focal plane respectively. `jitter_spa` and `jitter_spe` are the jitter positions in deg in the spatial and spectral direction respectively. Finally, `output_file` is an output file, as described in *Cached signals*.

Note: Because of the phisycs of the problem, the total power collected on the focal plane is not always conserved. However, for debugging reasons, the user can force the conservation of the total power by setting the following parameter in the channel configuration file: **<force_power_conservation>** `True` **</force_power_conservation>**

Focal plane oversampling factor for small jitter effects

It may happens that the jitter effect is too small to be captured by the defined oversampling factor. In this case, the focal plane is resampled in order to capture the jitter rms in at least 3 sub-pixels. The user can specify the number of sub-pixels to caputure the jitter rms by setting

```
<channel> channel_name
    <detector>
    <jitter_rms_min_resolution> 10 </jitter_rms_min_resolution>
    </detector>
</channel>
```

In this case 10 subpixels are used. By default this quantity is set to 3. Small numbers of sub-pixels are suggested to sample random jitter effects (to sample a Normal distributed noise effect, 3 sub-pixel are more than enough), while larger numbers might be needed to sampled pointing drift. The use of an incorrect number of sub-pixel may results in a digitalisation effect on the photometry.

The magnification is computed by *PrepareInstantaneousReadOut*, but the resampling operation is performed by the *oversample*, method of *InstantaneousReadOut*, which is using the `scipy.interpolate.RectBivariateSpline`[26] class. If the focal plane is Nyquist sampled in the original oversampled focal plane, the signal information is conserved.

The ovsersampling can be imposed by the user by setting

[26] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RectBivariateSpline.html#scipy.interpolate.RectBivariateSpline

```
<channel> channel_name
    <detector>
    <jitter_resampler_mag> 2 </jitter_resampler_mag>
    </detector>
</channel>
```

In the example we are suggesting the code to use a magnification factor for the resampler of *2*. If the base oversampling factor is *4*, we are now resampling each pixel with an oversampling factor of $4 \times 2 = 8$. However, if the suggested magnification factor is not enough to sample the jitter rms with 3 sub-pixels, the code computes and apply the right factor.

Note: The magnificattion and the rms minimum resolution are two face of the same coin.

2.4.4 Astronomical signals

Forewords

Now we can introduce some astronomical signals. Signals are intended as relative variation on the target source signal. So, they are defined as a function of the target source signal in time.

Few forewords before we start. The astronomical signal are not a core part of the ExoSim framework. The ExoSim simulations are aimed to reproduce complex instrumental sistematic effects. In order to train data reduction pipeline to cope with these effects, there is no need to introduce some astronomical signals. Therefore, the the astronomical signals we are introducing here are not intended as a perfect copy of the expected one, but only representative, as order of magnitude effects. For the same reason, we introduce them after the jitter and not before. The jitter is the most important effect to be reproduced, and it is the one that is more difficult to model. introducing the astronomical signal before the jitter would make the jitter model more difficult to implement and the simulations far more slow. This simulation order prevent ExoSim from simulating the jitter spectral effect on the resulting signal, which is however a second-order effect, compared to other noise sources.

This order of magnitude simulation of the astronomical signal however is improved compared to the previous version of ExoSim: this new version of the code takes into account also the smoothing effect on the astronomical signal of the instrument line shape and the intra-pixel response function. More details are reported in the following section.

Estimate the signal

The astronomical signals are intriduced in the *sky configuration file*, along with the source description. The *Task* describing the astronomical signal is called *EstimateAstronomicalSignal*. This is an abstract tasks with no model implemented. A complete example is reported in *EstimatePlanetarySignal*.

Here we report as example a signal that is the primary transit light curve of an exoplanet modelled using *EstimatePlanetarySignal*.

```
<source> HD 209458
   <source_type> planck </source_type>
   <R unit="R_sun"> 1.18 </R>
   <T unit="K"> 6086 </T>
   <D unit="pc"> 47 </D>

   <planet> b
```

(continues on next page)

(continued from previous page)

```
      <signal_task>EstimatePlanetarySignal</signal_task>
      <t0 unit='hour'>4</t0>
      <period unit='day'>3.525</period>
      <sma>8.81</sma>
      <inc unit='deg'>86.71</inc>
      <w unit='deg'>0.0</w>
      <ecc>0.0</ecc>
      <rp> 0.12 </rp>
      <limb_darkening>linear</limb_darkening>
      <limb_darkening_coefficients>[0]</limb_darkening_coefficients>
   </planet>
</source>
```

In this example we point to *EstimatePlanetarySignal* task to model the primary transit light curve of an exoplanet thanks to the `signal_task`. Everything else under the `planet` tree is a parameter needed by the indicated Task. Note that `planet` is the keyword needed for *EstimatePlanetarySignal*, but it can be any other keyword, as long as the corresponding Task is able to parse it.

The user can define more astronomical signals for the same star. All of them are loaded and applied one per time by ExoSim2.

Warning: The current version of ExoSim applies the astronomical signals only to the target star. Please, be sure to define the astronomical signal for the target star only. If astronomical signals are needed for multiple stars, multiple simulations can be defined and the results can be combined later.

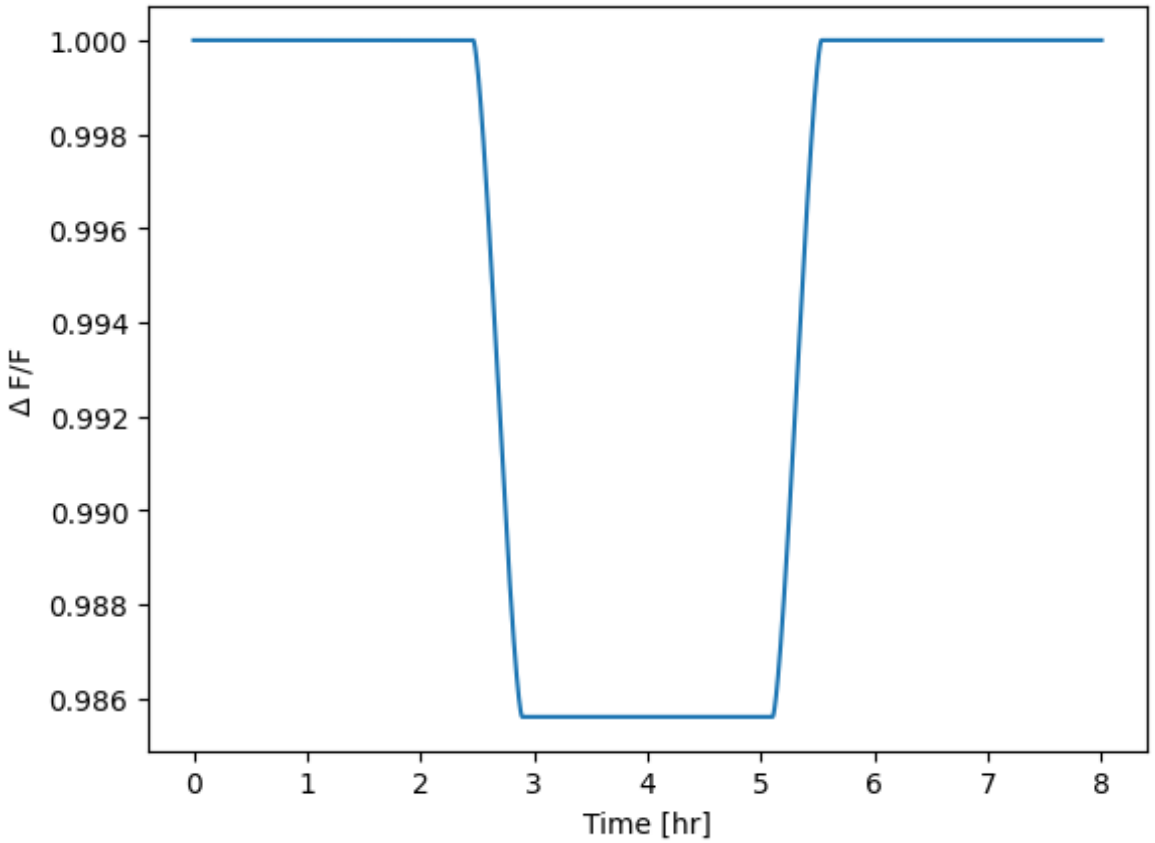
The astronomical signals are parsed by *FindAstronomicalSignals*, which looks for the `signal_task` keyword and instantiates the corresponding Task. The signal name is the parent tree keyword, in this case `planet`.

EstimatePlanetarySignal is based on the batman package[27] presented in Kreidberg 2015[28]. As usual, the User can replace the default Taks withg a custom one. The results of an *EstimateAstronomicalSignal* task shall be a 2D array with the first dimension being the wavelength and the second the time.

Warning: To run *EstimatePlanetarySignal* you need to have the batman package[29] installed. Because the `batman` package is not a core dependency of ExoSim, it is not installed by default.

In this example the planetary radius is constantly 0.12 times the stellar radius, as indicated under the `rp` keyword. For a single wavelength, the transit light curve is the following:

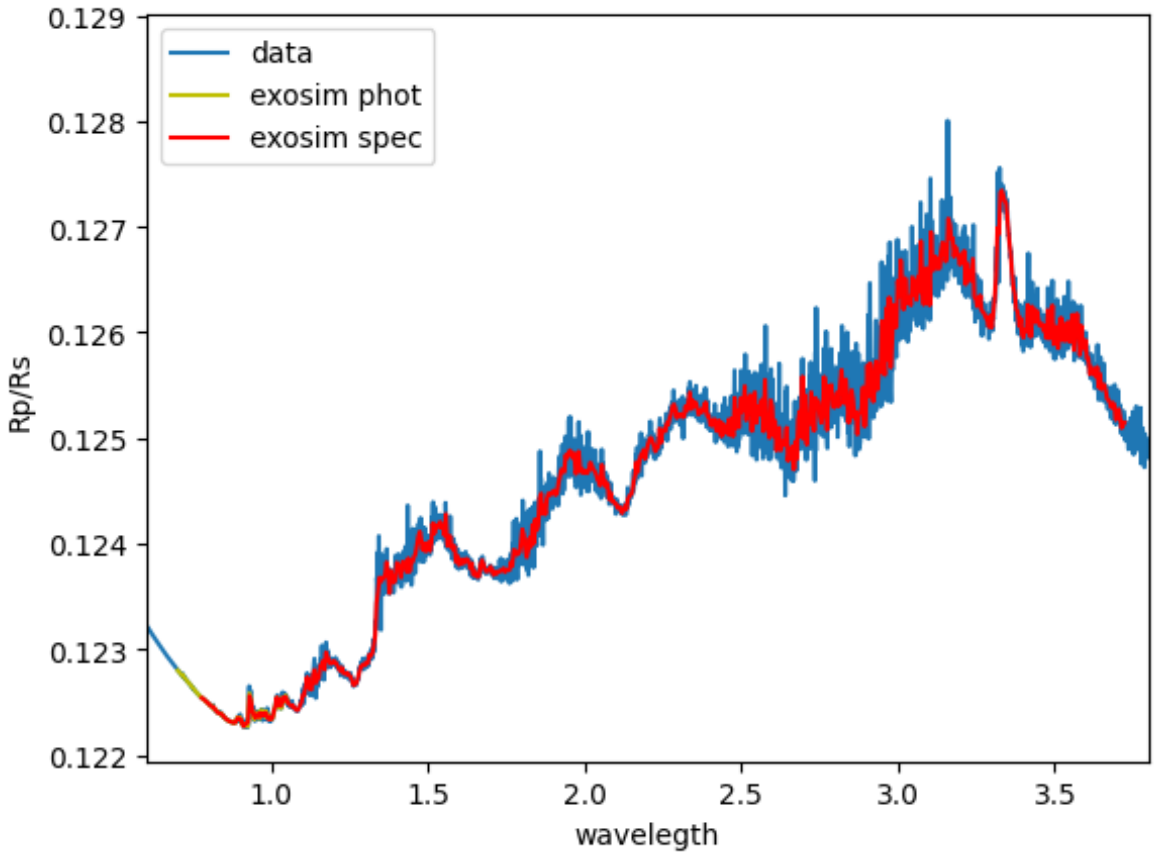
[27] http://lkreidberg.github.io/batman/docs/html/index.html
[28] https://ui.adsabs.harvard.edu/abs/2015PASP..127.1161K/abstract
[29] http://lkreidberg.github.io/batman/docs/html/installation.html



If we want to simulate a transit with a varying radius, we can use the `rp` keyword to indicate a csv file.

```
<source> HD 209458
   <planet> b
      <rp> radius_data.csv </rp>
   </planet>
</source>
```

where the `radius_data.csv` file is a csv file with two columns, the first being the wavelength and the second the radius in stellar radii. In this case, the input data are binned by the Task. To give an example, we use a simulated forward model for HD 209458 b produced with TauREx3 and the resulting spectrum is the following:



The used file is available in the `example/data` folder of the ExoSim package.

Multiple signals can be listed in the `sky` configuration file, and they will be parsed and applied one per time.

Apply the signal

Instrument line shape

To apply the signal, the Instrument line shapes are needed. These are loaded from the focal plane file by the *LoadILS* task. The Task returns a data cube of 1D psf for each wavelength. The data cube is a 3D array with the first dimension being the time, the second the wavelength and the third the shape response on the spectral direction. These line shapes are normalized to their maximum value, so that the maximum value of the line shape is 1.

The instrument line shapes produced by this task are not the same as the instrument line shapes as defined in the literature. To be used as the instrument line shapes as defined in the literature they need to be convolved with the intra-pixel response. This convolution is not part of this Task as it affects the way the ILS are sampled. The convolution with the intra-pixel response is done in the *ApplyAstronomicalSignal* Task, where the ILS are used to convolve the astronomical signal.

The *LoadILS* task is a default Task. If needed, it can be replaced with a custom Task.

```
<channel> channel_name
   <detector>
      <ils_task>LoadILS</ils_task>
```

(continues on next page)

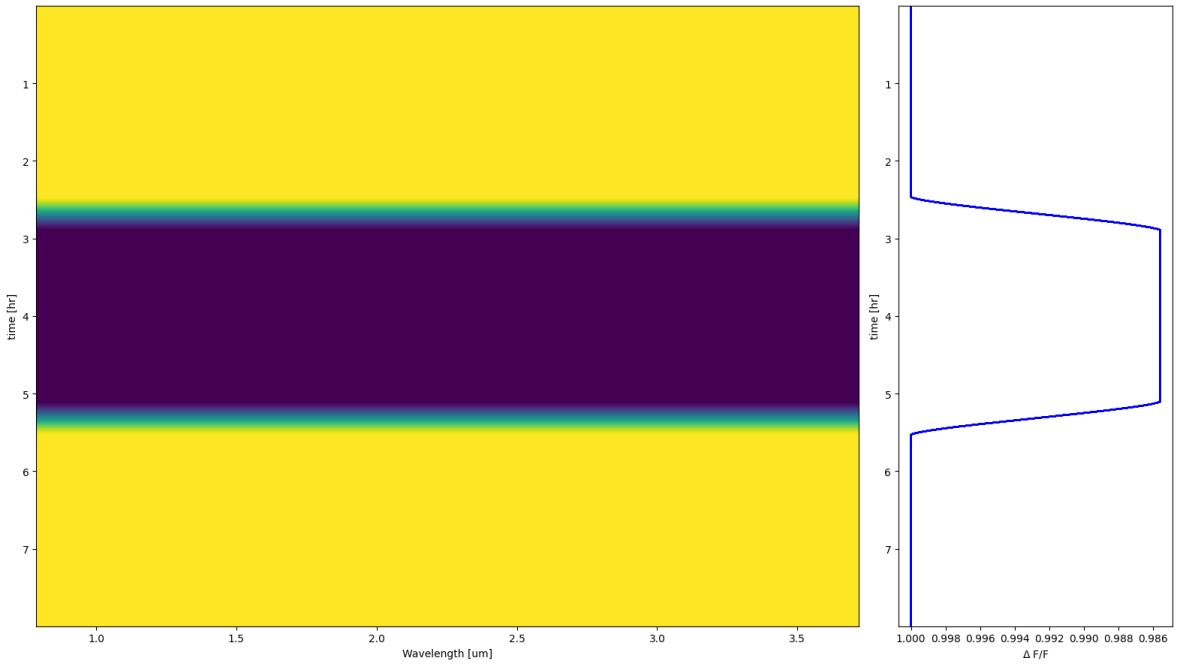
(continued from previous page)

```
    </detector>
</channel>
```

Signal application

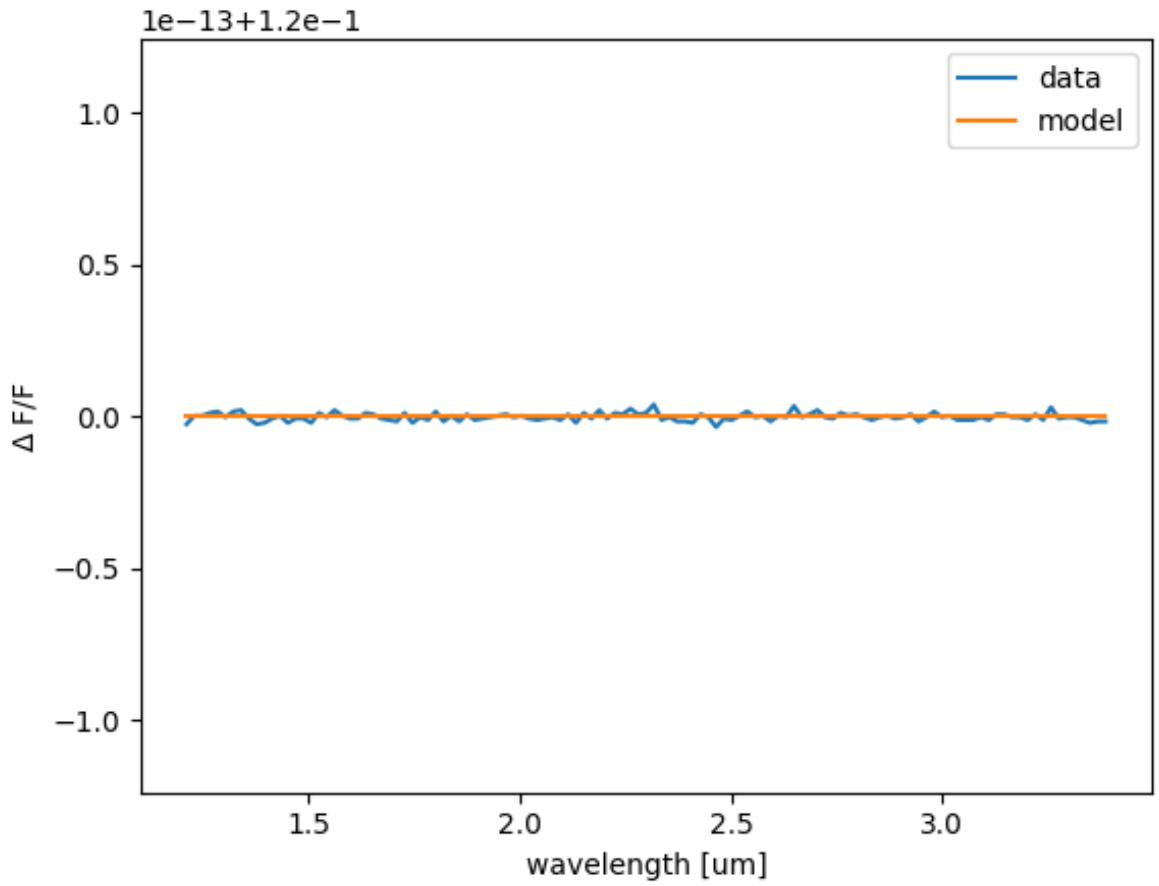
Once parsed, the astronomical signal is applied to Sub-Exposures by the *ApplyAstronomicalSignal* task. This task convolves the astronomical signal with the instrument line shape and the intra-pixel responde, adds it multiply it the Sub-Exposures.

Here we show some example results. In the following we are considering a spectrometer read using Correlated Double Sampling (CDS). We are considering a transit of HD 209458 b with a radius of 0.12 times the stellar radius. We select the second NDR for each ramp and we divide it by the second NDR of the first ramp. Then we sum the resulting images on the spectral direction. The following picture shows the results. On the left panel is reported the transit light curve for each pixel. So, the y axis is the time, while teh x axis is the pixel number, corresponding to a certain wavelength. On the right panel is reported the transit light curve for each wavelength.

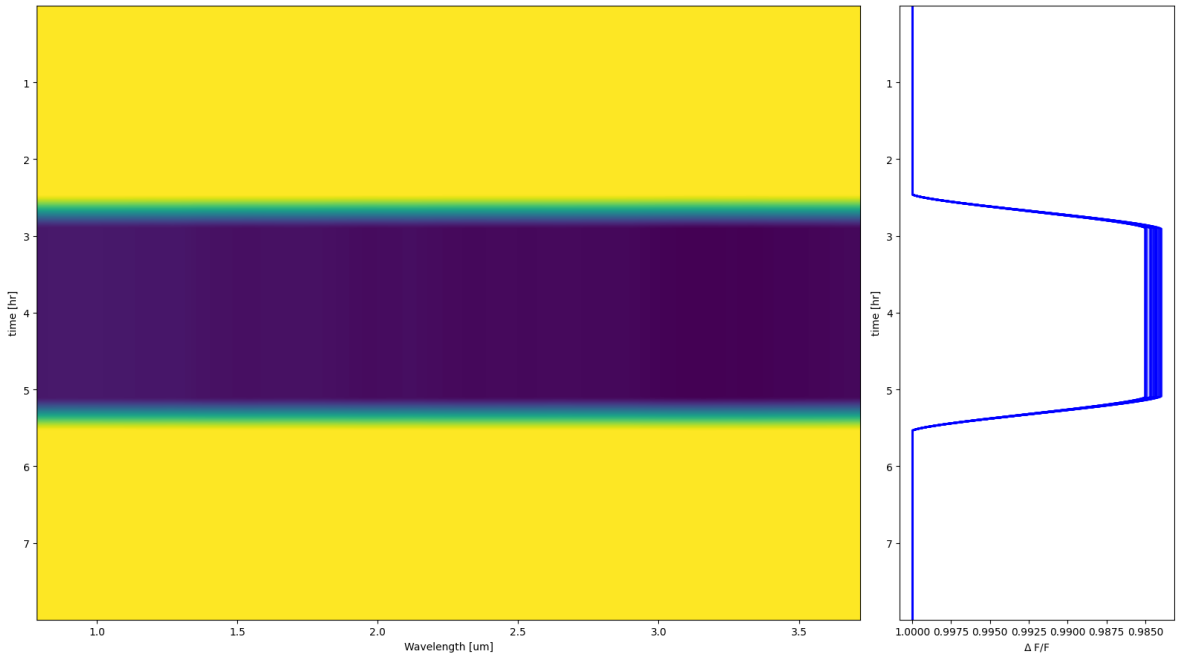


Because the transit depth is constant, the transit light curve for each spectral pixel is the same, and therefore the transit light curves on the right plot are aligned.

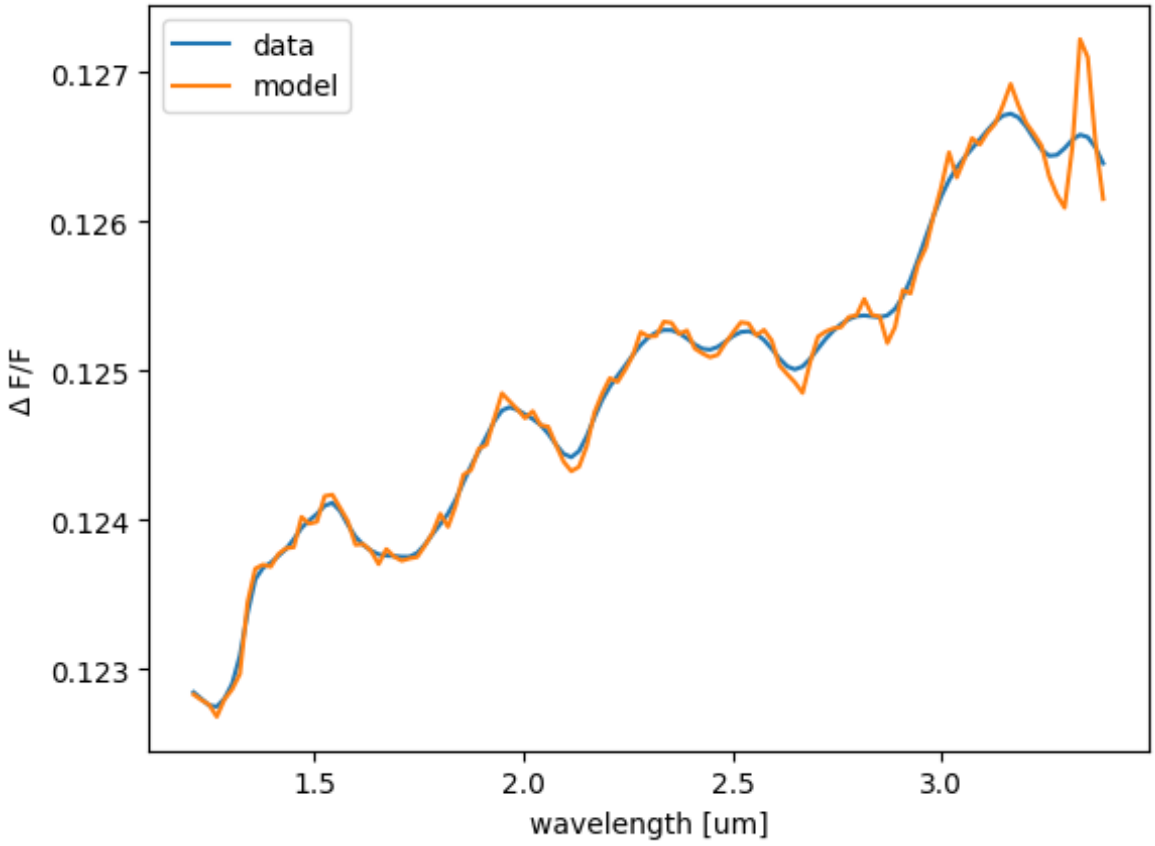
The following plot shows the transit depth at the center of the transit for each wavelength. We compare the transit depth with the input model, which is the result of constant radii ration of 0.12 between planet and star, and we see the the two are the same up to the numerical precision of 1e-15.



In the following example we use the same input parameters, but we use a varying radius. The input model is the same shown above for HD 209458 b. We can see now that in the first plot the transit light curves are not aligned anymore.



Also, we compare again the transit depth at the center of the transit for each wavelength, with the expected model. We can see that curve extracted from ExoSim data is smoother than the input model. This is the effect of applied ILS to the input model.



2.4.5 Finalising the Sub-Exposures

Add background sub-exposures

Because more sources can be in the field of view (see *Multiple sources in the field*), ExoSim allows to include the background in the sub-exposures. The background stars are read using the same procedure described in *Instantaneous readout*, with the same readout parameters used for the target star.

The resulting sub-exposures are added to the focal plane sub-exposures, and stored back in the output.

To include the background to the produced sub-exposures just enable it on the configuration file, setting the option to *True*.

```
<channel> channel_name
    <detector>
        <add_background_to_se> True </add_background_to_se>
    </detector>
</channel>
```

Add foregrounds sub-exposures

Similarly to what has been done for the focal planes, once the sub-exposure are completed, we operate on the diffused light foregrounds.

To include the foregrounds to the produced sub-exposures just enable it on the configuration file, setting the option to *True*.

```
<channel> channel_name
    <detector>
        <add_foregrounds_to_se> True </add_foregrounds_to_se>
    </detector>
</channel>
```

If the keyword is missing, the foregrounds are included by default.

For each of the sub-exposures in the output we select the foreground focal plane corresponding to the acquisition time, and we multiplied it by the integration time. The resulting foreground sub-exposure is added to the focal plane sub-exposure and stored back in the output.

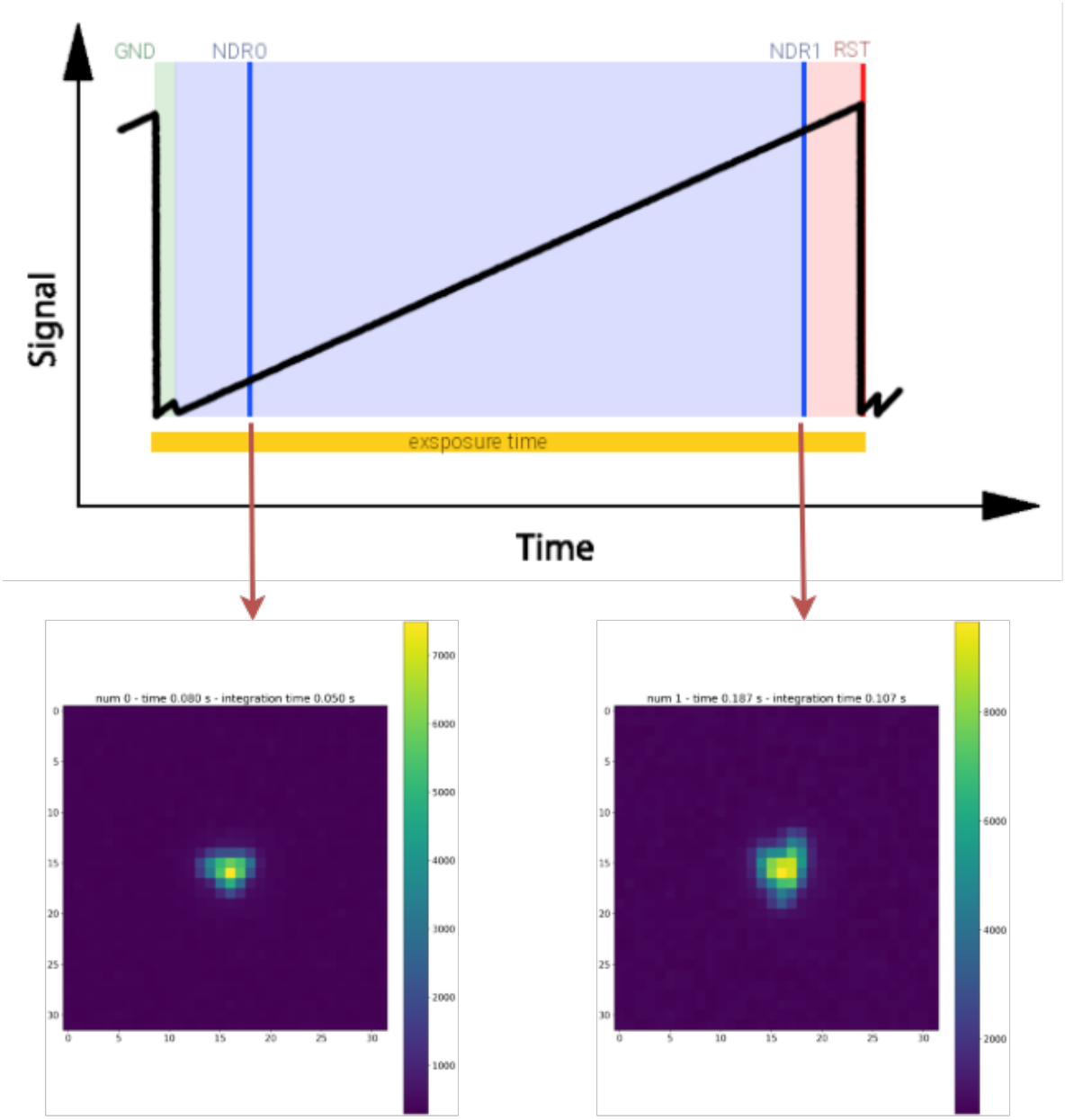
This is handled by *AddForegrounds* task.

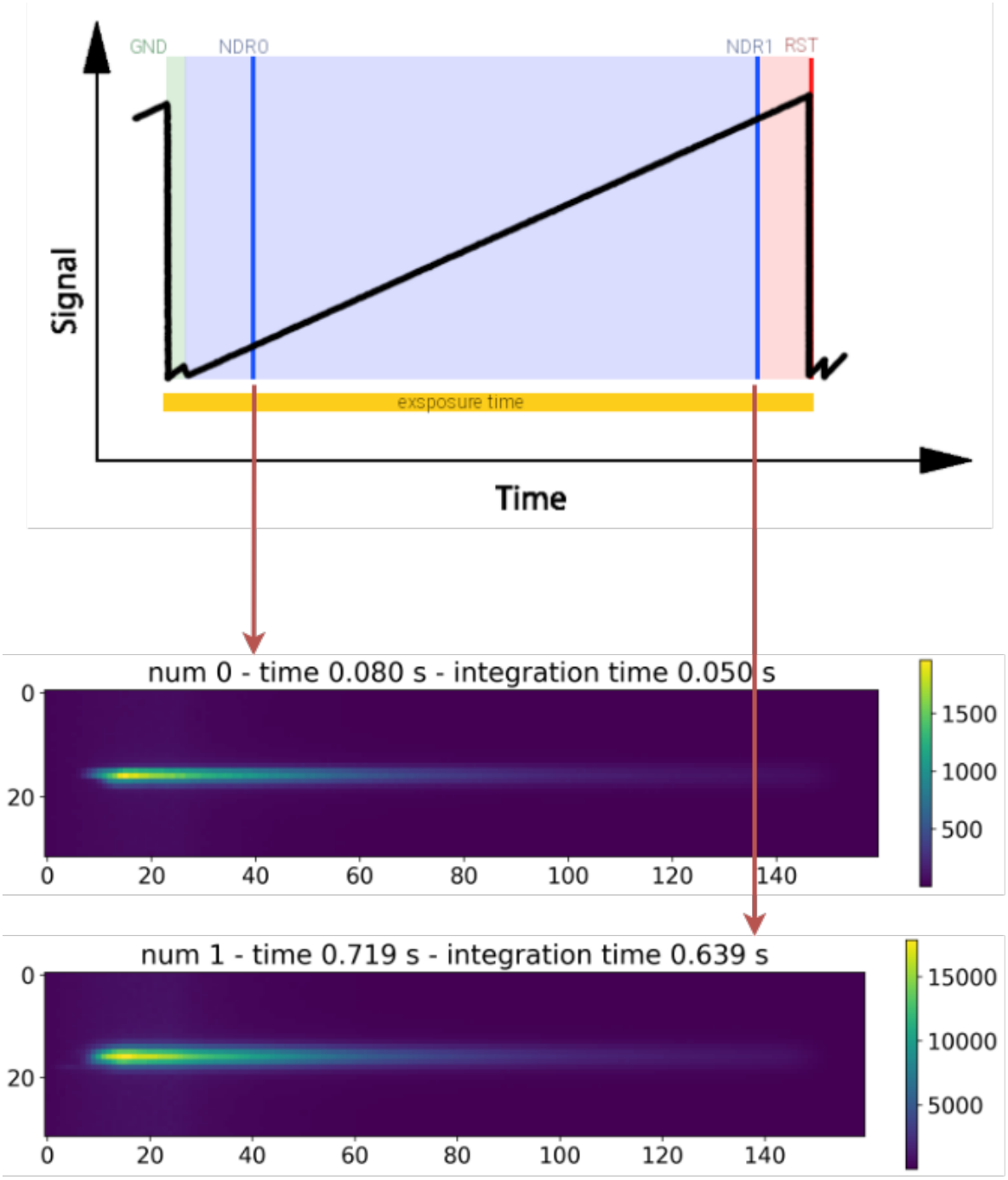
```
import exosim.tasks.subexposures as subexposures

addForegrounds = subexposures.AddForegrounds()
se_out = addForegrounds(subexposures=se_out, frg_focal_plane=frg_fp,
                        integration_time=integration_times)
```

With clear reference to the quantities defined above.

Resulting sub-exposure

The resulting sub-exposures will look similar to the followings:


GND
NDR0
NDR1
RST
Signal
exsposure time
Time
num 0 - time 0.080 s - integration time 0.050 s
num 1 - time 0.187 s - integration time 0.107 s



These examples have been produced with the procedure described in *Sub-Exposures Plotter*.

Quantum efficiency variation

Each pixel in the focal plane has a slightly different Quantum Efficiency (QE) from the others. This behaviour can be simulated in *ExoSim* by editing the normalisation of the pixel QEs

This can be set with a custom *Task*, as described in *Custom Tasks*, or by using the default *LoadQeMap* task. This task loads the QE variation map pre-computed from and *.h5* file. If the user has its own map, can write a custom task load this map into a *Signal* class. Otherwise, *ExoSim* includes a tool (*ExoSim Tools*) which allow the creation of a quantum efficiency variation map (see *Quantum efficiency variation map*), which can be stored and used in successive simulations.

The *Task* to use to laod the QE variation map should be indicated under the channel detector configuration using the *qe_map_task* keyword. In the following we report the example using the default *LoadQeMap* task:

```
<channel> channel_name
    <detector>
        <qe_map_task> LoadQeMap </qe_map_task>
        <qe_map_filename> __ConfigPath__/data/payload/qe_map.h5 </qe_map_filename>
    </detector>
</channel>
```

where the *qe_map_filename* keyword indicates the quantum efficiency variation map to use for every channel of the payload.

Alternatively, the map can be provided as a simple numpy array (see numpy documentation[30]) and parsed by the *LoadQeMapNumpy* task:

```
<channel> channel_name
    <detector>
        <qe_map_task> LoadQeMapNumpy </qe_map_task>
        <qe_map_filename> qe_map.npy </qe_map_filename>
    </detector>
</channel>
```

The resulting map is then applied to all the focal planes in the channel by the *ApplyQeMap* tasks. Because the quantum efficiency variation map can be time dependent, but sampled at a different cadence than the Sub-Exposure, in the Sub-Exposure signal is included a new key in the metadata: *qe_variation_map_index*. This array contains the indexes of the quantum efficiency realisation used for each of the sub-exposure: the array is as long as the sub-exposure temporal axis, and for each time step is reported the time index of the quantum efficiency variation map applied to that sub-exposure.

If no quantum efficiency variation map is provide, the code skip this step raising a Warning.

2.4.6 Sub-Exposures automatic Recipe

Al the steps needed to the production of sub-exposures are already collected in a pre-made pipeline. This is under the *recipes* of *ExoSim*.

```
from exosim import recipes
recipes.CreateSubExposures(input_file='./input_file.h5',
                           output_file='./output_file.h5',
                           options_file='your_config_file.xml')
```

The *CreateSubExposures* can also be run from console as

[30] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

```
exosim-sub-exposures -c your_config_file.xml -i input_file.h5 -o output_file.h5
```

or

```
exosim-sub-exposures -c your_config_file.xml -i input_file.h5 -o output_file.h5 -P
```

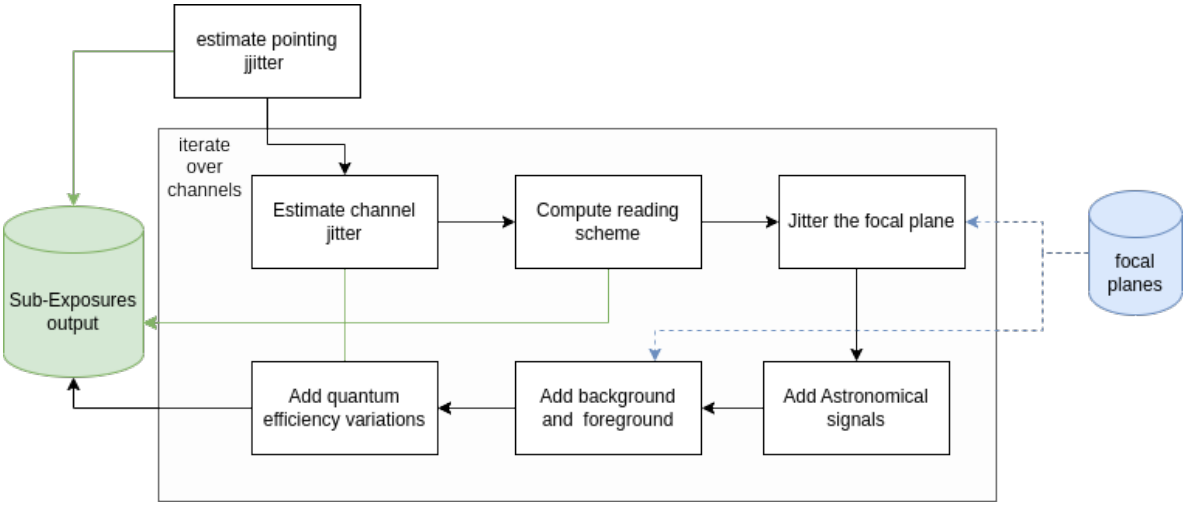
to also run ExoSim *SubExposuresPlotter*, which is documented in *Sub-Exposures Plotter*.

The user can also set the chunk size (see *Instantaneous readout*) using

```
exosim-sub-exposures -c your_config_file.xml -i input_file.h5 -o output_file.h5 --
↪chunk_size N
```

where *N* is the desired size expressed in Mbs.

The steps flow is summarised in the following figure:

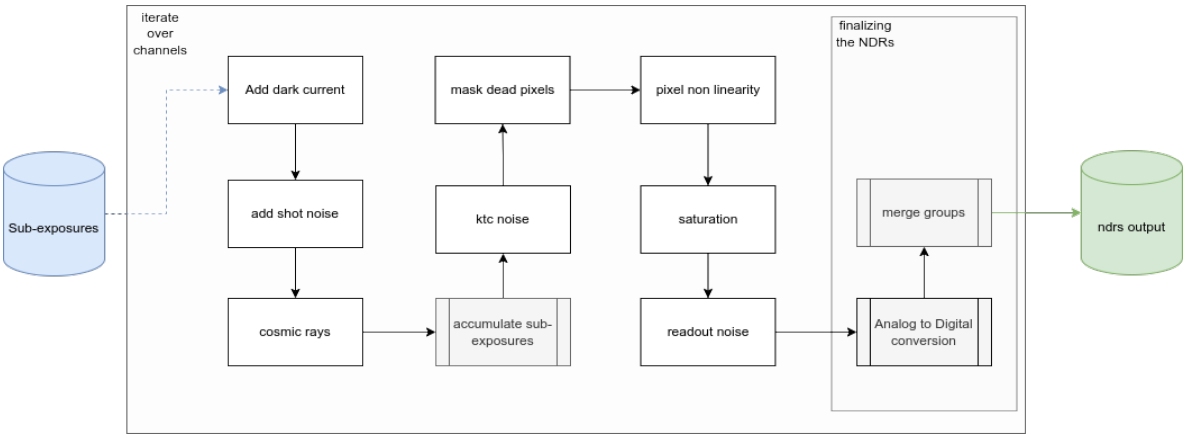


ExoSim also include a dedicated Plotter, called *SubExposuresPlotter*, which is described in *Sub-Exposures Plotter*

2.5 NDRS

Finally, we can produce the NDRs.

The steps flow is summarised in the following figure. The Tasks are divided here into blocks to help the reader.



In the following, we discuss each step, one per time.

Note: Be careful. All of these actions operate on cached data. The result of each step is not a new dataset, but the input one with overwritten values.

The NDRs creation is automatized by a recipe: *CreateNDRs*.

ExoSim also includes a dedicated Plotter, called *NDRsPlotter*, which is described in *NDRs Plotter*

2.5.1 Dark Current

The dark current signal can be added to each sub-exposure using the *dark_current* keyword

```
<channel> channel
    <detector>
        <dark_current> True </dark_current>
    <detector>
</channel>
```

or disabled by setting the *dark_current* keyword to *False*.

By default, this *Task* used to add the dark current is *AddConstantDarkCurrent*, which, as the name suggests, adds a constant flux to each pixel.

It can be set as

```
<channel> channel
    <detector>
        <dark_current> True </dark_current>
        <dc_task> AddConstantDarkCurrent </dc_task>
        <dc_mean unit="ct/s"> 5 </dc_mean>
    <detector>
</channel>
```

Using the configuration reported in the example, the codes adds to each pixel $5\,ct/s \times t_{s,\,int}$ where $t_{s,\,int}$ is the sub-exposure integration time.

It is always possible to replace this function with one using a dark current map, to add a different dark current to each pixel which can also evolve in time. A custom task can be used to replace *AddConstantDarkCurrent* (see *Custom Tasks*).

An implementation of dark current map has been prepared assuming numpy array (see numpy documentation[Page 101, 31])
as input (*AddDarkCurrentMapNumpy*). It can be used as

```
<channel> channel
    <detector>
        <dark_current> True </dark_current>
        <dc_task> AddDarkCurrentMapNumpy </dc_task>
        <dc_map_filename> dark_map.npy </dc_map_filename>
    <detector>
</channel>
```

Note: Other custom realizations of this Task can be developed by the user (see *Custom Tasks*).

2.5.2 Shot Noise

The addition of shot noise to each sub-exposures is managed by *AddShotNoise*.

This functionality can be enabled in the channel configuration file as

```
<channel> channel
    <detector>
        <shot_noise> True </shot_noise>
    <detector>
</channel>
```

or disabled by setting the *shot_noise* keyword to *False*

This *Task* replace the values of each pixel with random number distributed around its true value according to a Poisson distribution.

$$S_{meas} = \mathcal{P}(S_{true})$$

Where S_{meas} is the new value, which represents the measured value, and S_{true} is the true pixel count value, which also is the original one.

Note: For reproducibility, the seed for the random generator can be set as described in *Random seed and Random generators*. Remember that in the case of multiple chunks used, the random seed used in any chunk is stored in the output file for reproducibility.

2.5.3 Cosmic Rays

The *AddCosmicRays* task is part of the Exosim simulation package. It models the impact of cosmic rays on a detector during its exposure time. Cosmic rays are high-energy particles originating from space, which can introduce noise into the captured data. This class provides a simulation of this effect by adding cosmic ray events to the detector's sub-exposures.

[31] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

Phenomenon Overview

Cosmic rays can interact with the detector pixels in various predefined shapes such as crosses, rectangles, or isolated pixels. These interactions can saturate the affected pixels, setting their value to the detector's full well depth. The class provides flexibility in specifying these interaction shapes and their associated probabilities.

The number of cosmic ray events is calculated based on several parameters:

- Cosmic ray flux rate (in ct/s/cm^2)
- Pixel size
- Saturation rate due to cosmic rays
- Number of spatial and spectral pixels in the detector
- Integration times for the sub-exposures

These parameters should be specified in the configuration XML file as shown below:

```
<channel>
    <detector>
        <delta_pix unit="micron"> 18.0 </delta_pix>
        <spatial_pix> 64 </spatial_pix>
        <spectral_pix> 364 </spectral_pix>
        <well_depth unit="count"> 100000 </well_depth>

        <cosmic_rays> True </cosmic_rays>
        <cosmic_rays_task> AddCosmicRays </cosmic_rays_task>
        <cosmic_rays_rate unit="ct/cm^2/s"> 5 </cosmic_rays_rate>
        <saturation_rate> 0.03 </saturation_rate>
    </detector>
</channel>
```

So the number of pixels hit by a cosmic ray during a sub-exposures is

$$hits = rate_{cosmic\,rays} * delta_{pix}^2 * N_{pix\,spatial} * N_{pix\,spectral} * t_{int}$$

Where

- $rate_{cosmic\,rays}$ is the *cosmic_rays_rate*;
- $delta_{pix}$ is the *delta_pix*;
- $N_{pix\,spatial}$ is the *spatial_pix*;
- $N_{pix\,spectral}$ is the *spectral_pix*;
- t_{int} is the sub-exposure integration times.

Then we can estimate how many of these events can saturate our pixel using the pixel saturation rate:

$$saturated = hits * rate_{saturation}$$

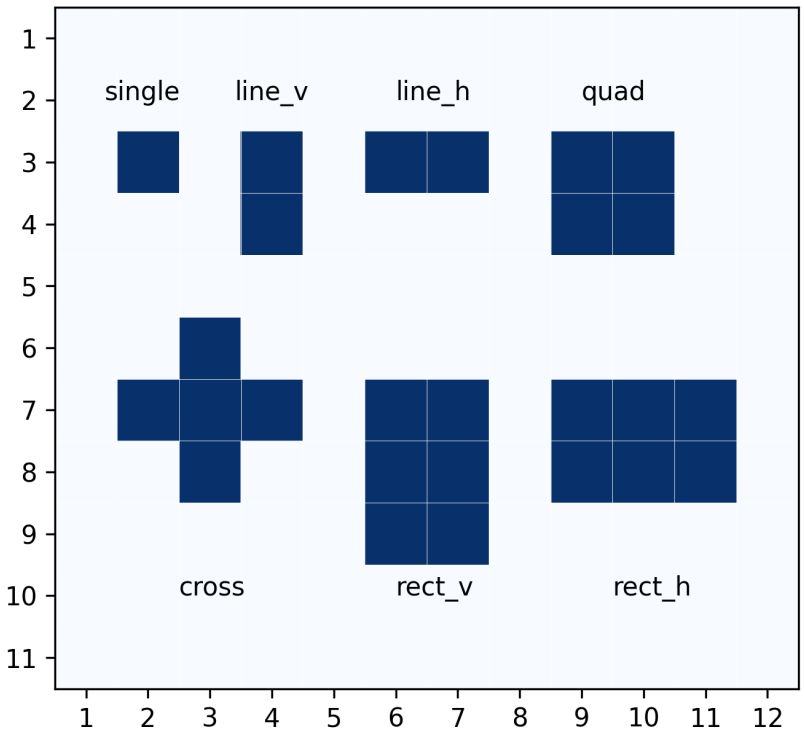
where $rate_{saturation}$ is the *saturation_rate*.

Then for each of these events, at least one pixel is saturated. If this value is smaller than unity, a random number is generated to estimate the odd of the event to saturate the pixel.

Interaction Shapes

The class includes predefined interaction shapes that describe the group of pixels saturated by each cosmic ray event. These are:

- Single pixel (`single`)
- Vertical line (`line_v`)
- Horizontal line (`line_h`)
- Square (`square`)
- Cross (`cross`)
- Vertical rectangle (`rect_v`)
- Horizontal rectangle (`rect_h`)



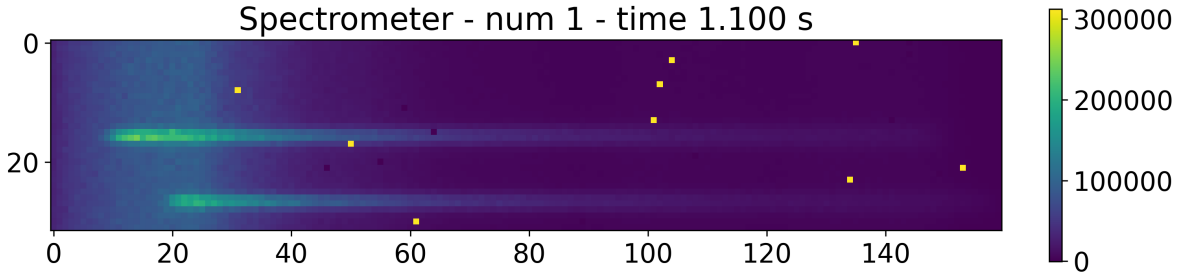
Specifying Probabilities

The user can specify the probability for each of these shapes in the XML configuration file:

```
<channel>
    <detector>
        <interaction_shapes>
            <line_v>0.5</line_v>
            <square>0.5</square>
        </interaction_shapes>
    </detector>
</channel>
```

If the sum of the specified probabilities is not equal to 1, the task will automatically use the `single` shape as a contingency to balance the probabilities.

Output

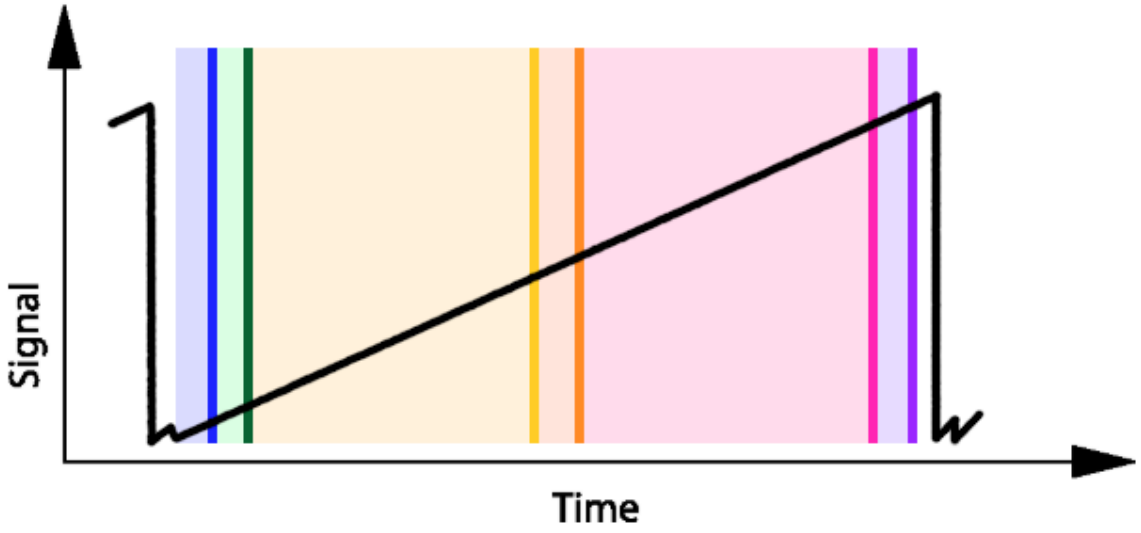


If an output group is provided, the default task will save all the pixels saturated by cosmic rays in a table, for reproducibility.

Note: Other custom realizations of this Task can be developed by the user (see *Custom Tasks*).

2.5.4 Accumulate NDRs

So far we have worked with sub-exposures. Each sub-exposures collects the signal incoming after the end of the collection of the previous sub-exposure, as indicated in the following figure from *Sub-Exposures*:



Now we want to accumulate the subsequent sub-exposures of the same exposure, building the ramp. This means that, starting from the first sub-exposure of the ramp, all the successive are equal of themselves plus the previous one. If an exposure is made of N sub-exposure, we have

$$Sub_0 = Sub_0$$

But then for every other sub-exposure on the same ramp

$$Sub_i = Sub_i + Sub_{i-1}$$

This is handled by *AccumulateSubExposures*, which overwrites the input cached dataset with the new one.

2.5.5 KTC noise

When the detector is reset, the offset signal in each pixel of each frame can be different. This is the kTC noise and can be included in the simulation as

```
<channel> channel
    <detector>
        <ktc_offset> True </ktc_offset>
    <detector>
</channel>
```

or disabled by setting the *ktc_offset* keyword to *False*.

By default, this *Task* used to add the reset bias is *AddKTC*, which adds a random number of counts to each pixel of the same ramp, normally distributed according to the given mean and standard deviation:

```
<channel> channel
    <detector>
        <ktc_offset> True </ktc_offset>
        <ktc_offset_task> AddKTC </ktc_offset_task>
        <ktc_sigma unit="ct"> 10 </ktc_sigma>
    <detector>
</channel>
```

$$S_{meas} = S_{meas} + \mathcal{N}(\mu = 0, \sigma = \sigma_{KTC})$$

Note: Other custom realizations of this Task can be developed by the user (see *Custom Tasks*).

2.5.6 Dead pixels map

Dead pixel map can be applied by default using `ApplyDeadPixelsMap`, as

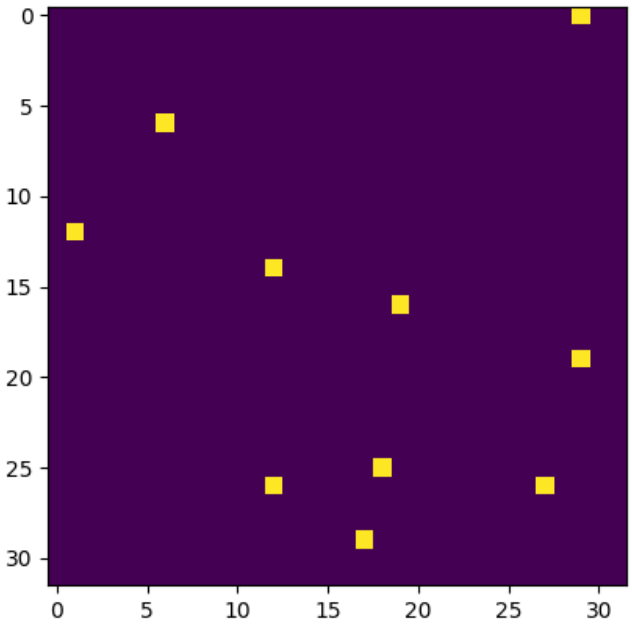
```
<channel> channel
    <detector>
        <dead_pixels> True </dead_pixels>
        <dp_map_task> ApplyDeadPixelsMap </dp_map_task>
        <dp_map> __ConfigPath__/data/payload/dead_pixel_map.csv </dp_map>
    <detector>
</channel>
```

As shown, the input is a *.csv* file. The file contains two columns with the spectral and spatial coordinates of the dead pixels: *spectral_coords* and *spatial_coords*.

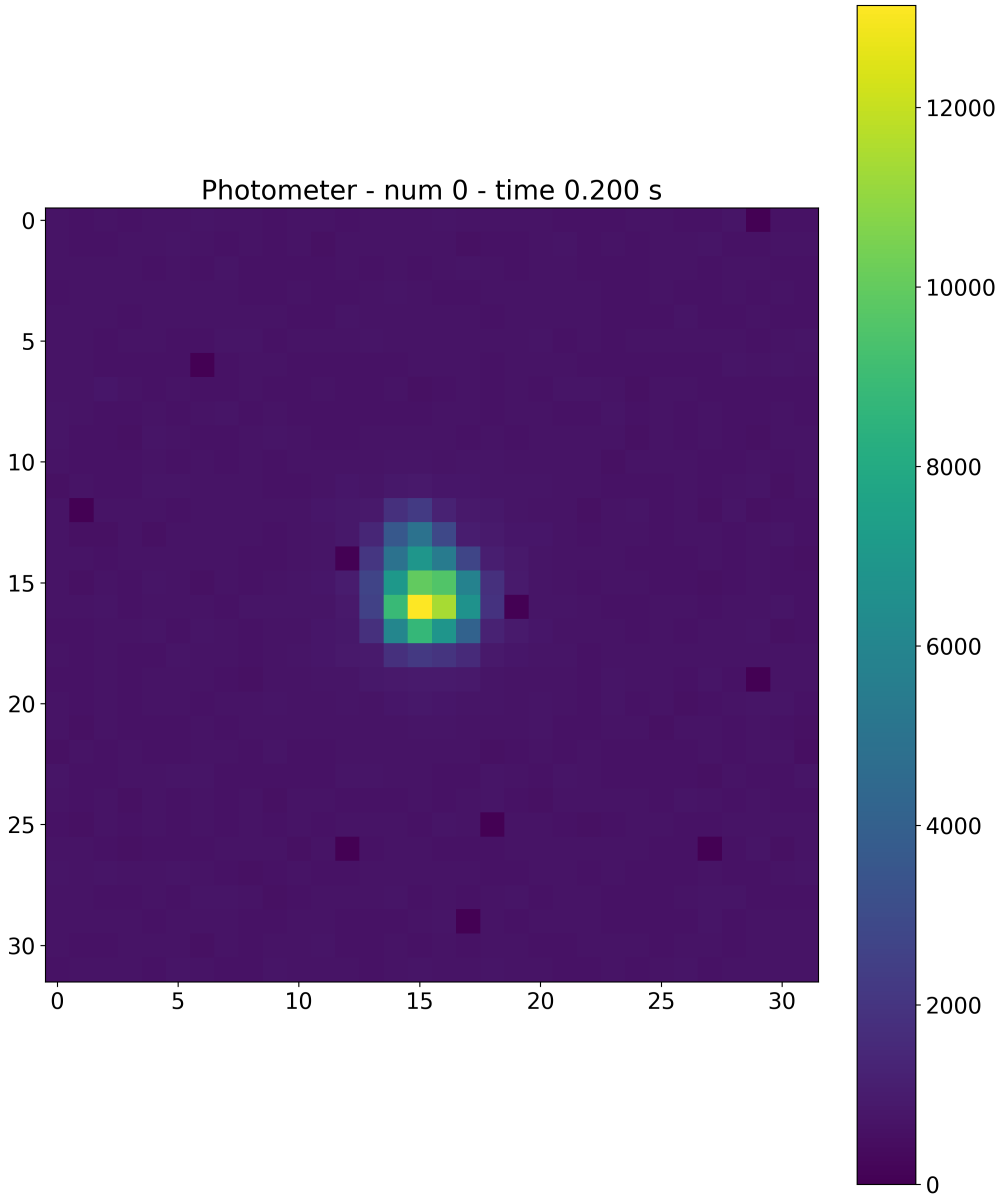
Alternatively, the dead pixel map can be provided as a numpy array (see numpy documentation[32]), and parsed with the `ApplyDeadPixelMapNumpy` task:

```
<channel> channel
    <detector>
        <dead_pixels> True </dead_pixels>
        <dp_map_task> AddReadNoiseMapNumpy </dp_map_task>
        <dp_map_filename> dead_pixel_map.npy </dp_map_filename>
    <detector>
</channel>
```

[32] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html



Applying these maps to the focal plane will result in



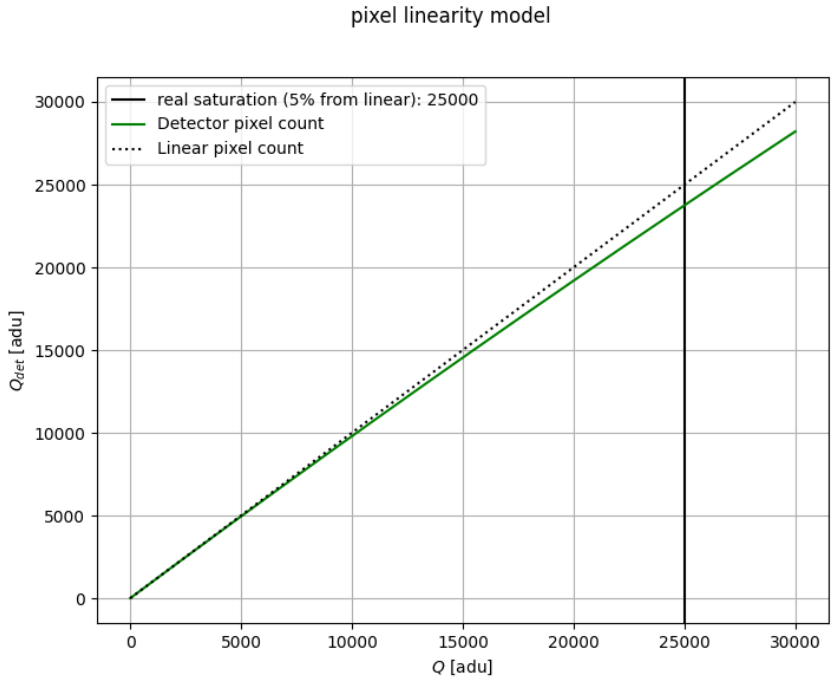
Note: Other custom realizations of this Task can be developed by the user (see *Custom Tasks*).

Note: If a dead pixel map is not available, *ExoSim* includes a dedicated tool to simulate one: *Dead pixels map*.

2.5.7 Pixels Non-Linearity and saturation

Pixels Non-Linearity

The detector's response to incoming light is not inherently linear. This non-linearity can be modeled as a function of the pixel value.



The detector non-linearity model is usually written as a polynomial such as

$$Q_{det} = Q \cdot (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi_t$, with ϕ being the number of electrons generated and t being the elapsed time.

To implement this non-linearity, the *ApplyPixelsNonLinearity* task is used. This Task needs as input a map of the coefficients of the polynomial for each pixel. The task requires a map of polynomial coefficients for each pixel. You can load this map using the *LoadPixelsNonLinearityMap* task and specify the file in your configuration..

As usual, the user can replace the default Task with a custom one. In this example, we use the *pnl_map.h5* file which is produced using one of the methods described in *Pixels Non-Linearity*.

```
<channel> channel
    <detector>
        <pixel_non_linearity> True </pixel_non_linearity>
        <pnl_task> ApplyPixelsNonLinearity </pnl_task>
        <pnl_map_task> LoadPixelsNonLinearityMap </pnl_map_task>
        <pnl_filename>__ConfigPath__/data/payload/pnl_map.h5</pnl_filename>
    <detector>
</channel>
```

Alternatively, the coefficients map can be provided as a numpy array (see numpy documentation[33]), and loaded

[33] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

using *LoadPixelsNonLinearityMapNumpy*:

```
<channel> channel
    <detector>
        <pixel_non_linearity> True </pixel_non_linearity>
        <pnl_task> ApplyPixelsNonLinearity </pnl_task>
        <pnl_map_task> LoadPixelsNonLinearityMapNumpy </pnl_map_task>
        <pnl_filename>__ConfigPath__/data/payload/pnl_map.npy</pnl_filename>
    <detector>
</channel>
```

Saturation

After undergoing non-linear adjustments, a pixel may reach its saturation point, or "full well capacity." The *ApplySimpleSaturation* handles pixel saturation. It sets the value of each pixel exceeding the full well capacity to the maximum allowable counts.

It needs to know the full well capacity and it can be set and used as

```
<channel> channel
    <detector>
        <well_depth unit="count"> 100000  </well_depth>
        <saturation> True </saturation>
        <sat_task> ApplySimpleSaturation </sat_task>
    <detector>
</channel>
```

2.5.8 Gain Drift

The *AddGainDrift* task, part of the ExoSim simulation package, is designed to model and apply gain drift to a detector simulator. The gain drift is constructed as a polynomial trend dependent on time and wavelength.

The polynomial coefficients are randomly generated within specified ranges. Finally, the resulting amplitude is rescaled according to the input parameters.

Usage and Parameters

To apply gain drift using the *AddGainDrift* task, the following parameters should be specified in the configuration file. Here we include also some example values. - `gain_coeff_order_t`: Order of the polynomial used for the time-dependent trend. - `gain_coeff_t_min` and `gain_coeff_t_max`: Minimum and maximum values for the randomly generated coefficients of the time-dependent polynomial trend. - `gain_coeff_order_w`: Order of the polynomial used for the wavelength-dependent trend. - `gain_coeff_w_min` and `gain_coeff_w_max`: Minimum and maximum values for the randomly generated coefficients of the wavelength-dependent polynomial trend. - `gain_drift_amplitude`: gain drift desired maximum amplitude relative to the signal.

These parameters control the characteristics of the gain noise, allowing for detailed modeling of the detector's response.

```
<channel>
    <detector>
        <gain_drift> True </gain_drift>
        <gain_drift_task> AddGainDrift </gain_drift_task>
```

(continues on next page)

(continued from previous page)

```
        <gain_drift_amplitude> 1e-2 </gain_drift_amplitude>

        <gain_coeff_order_t> 5 </gain_coeff_order_t>
        <gain_coeff_t_min> -1.0 </gain_coeff_t_min>
        <gain_coeff_t_max> 1.0 </gain_coeff_t_max>

        <gain_coeff_order_w> 5 </gain_coeff_order_w>
        <gain_coeff_w_min> -1.0 </gain_coeff_w_min>
        <gain_coeff_w_max> 1.0 </gain_coeff_w_max>
    </detector>
</channel>
```

Customization

The *AddGainDrift* task is designed for flexibility and can be customized or replaced by a user-defined implementation as needed.

Note: Users are encouraged to develop their own custom realizations of this task to fit specific simulation requirements (see *Custom Tasks*).

2.5.9 Read out noise

Every time a pixel is read by the electronic, an error is introduced. This is called *read noise*. This is the noise of the amplifier which converts the counts into a change in analog voltage for the ADC. This kind of uncertainty is represented by default by `AddNormalReadNoise`. This `Task` simulates the read noise as a normal distribution whose parameters can be defined in the configuration file.

```
<channel> channel
    <detector>
        <read_noise> True </read_noise>
        <read_noise_task> AddNormalReadNoise </read_noise_task>
        <read_noise_sigma unit="ct"> 10 </read_noise_sigma>
    <detector>
</channel>
```

A different realization of the same distribution is added to each pixel of each sub-exposure.

$$S_{meas} = S_{meas} + \mathcal{N}(\mu = 0, \sigma = \sigma_{RN})$$

Alternatively, a map of read noise measured for each pixel can be used. A default Task is provided for this scope assuming numpy array (see numpy documentation[34]) as input: `AddReadNoiseMapNumpy`

```
<channel> channel
    <detector>
        <read_noise> True </read_noise>
        <read_noise_task> AddReadNoiseMapNumpy </read_noise_task>
        <read_noise_filename> read_noise_map.npy </read_noise_filename>
```

(continues on next page)

[34] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

(continued from previous page)

```
    <detector>
</channel>
```

Note: Other custom realizations of this Task can be developed by the user (see *Custom Tasks*).

Note: For reproducibility, the seed for the random generator can be set as described in *Random seed and Random generators*. Remember that in the case of multiple chunks used, the random seed used in any chunk is stored in the output file for reproducibility.

2.5.10 Analog to Digital conversion

At this point the NDRs are stored as *float64*, however, we know that the detector output is reported in integers in adu units. Here we simulate the Analog to Digital Converter (ADC) thanks to `AnalogToDigital`, which converts the $counts$ units of sub-exposures into adu units of NDRs. This task needs two inputs from the channel configuration file:

- the number of bits of the output integer (e.g. 16 bits)
- ADC gain factor

These can be injected as

```
<channel> channel
    <detector>
        <ADC> True </ADC>
        <ADC_num_bit> 16 </ADC_num_bit>
        <ADC_gain> 0.5 </ADC_gain>
        <ADC_round_method>floor</ADC_round_method>
    <detector>
</channel>
```

To enable the conversion set *True* for the *ADC* keyword, as in the example. If *False*, this step will be skipped.

In this example, we want the ADC to convert the NDRs into 16 bits unsigned integers. Because integers can represent numbers up to a maximum value of $2^{16} - 1 = 65535$, we need a conversion factor to rescale our float NDRs to fit in the new data type range. This conversion factor is defined by *ADC_gain*, such that the float focal plane is multiplied by this gain (g_{ADC}) before the conversion. If this gain is not known, an estimate is provided by the *ADC gain estimator* tool.

$$S_{out} = [ADC_{gain} \cdot S_{meas}]_{int}$$

The user can input any integer number of bits up to 32. The `AnalogToDigital` chooses the minimum Python data type to store the desired output to minimize the size of the output product and to be more representative of the expected result.

ADC_round_method keyword indicates which method the ADC should use to cast the float into integers. Three options are available:

- *floor* which uses `numpy.floor`;
- *ceil* which uses `numpy.ceil`;
- *round* which uses `numpy.round`;

The default is *floor*.

2.5.11 Merge NDRs and Results

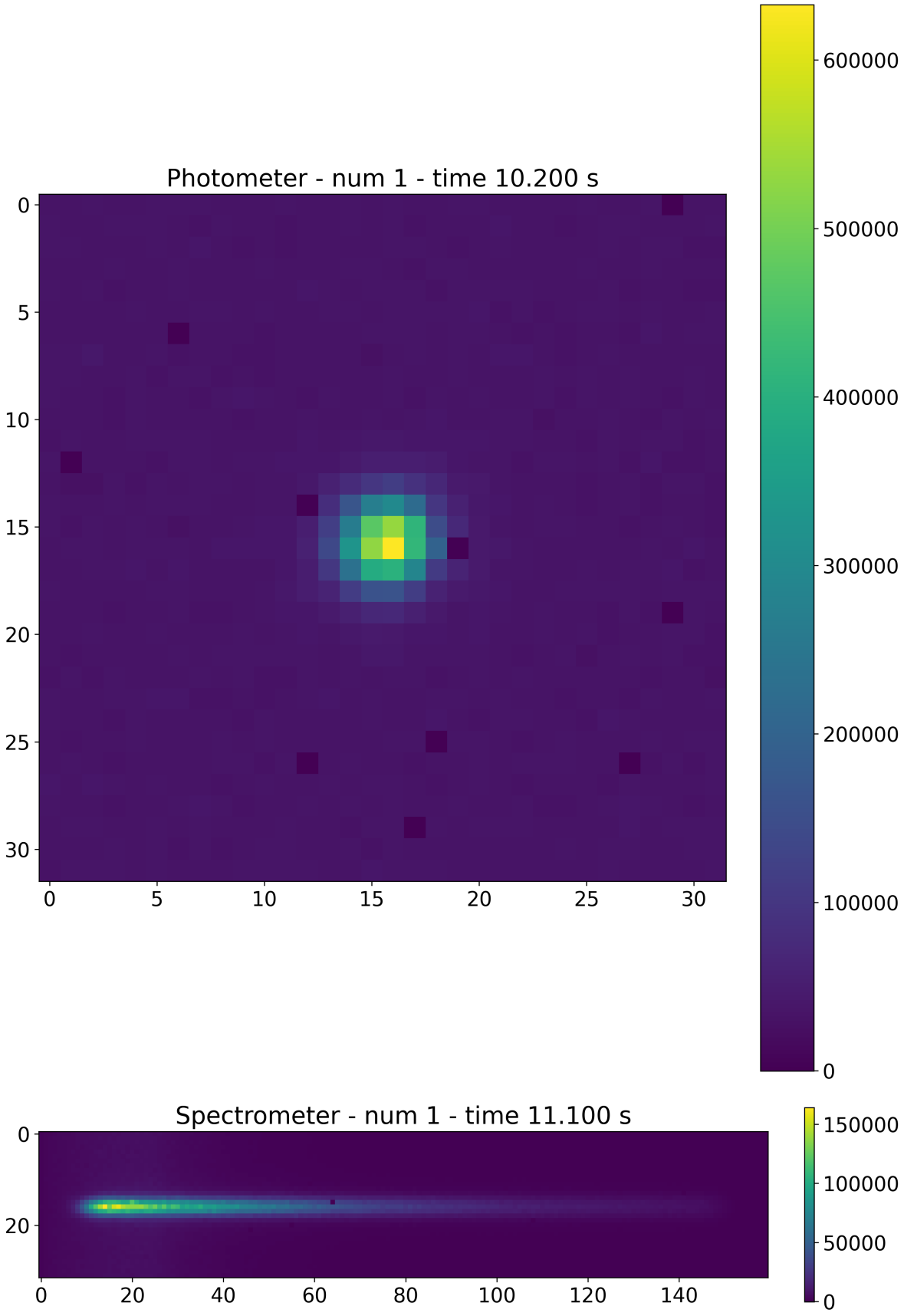
NDRs of the same group are to be merged. This is automatically handled by `MergeGroups`. This `Task` simply iterates over the exposures and finds all the NDRs belonging to the same group. These are then averaged together, returning a single NDR.

Assuming that the k-th NDR comes from the combination of N NDRs in the same group, we can write this as

$$NDR_k = \frac{1}{N}\sum_i^N S_{out,\,i}$$

Resulting NDRs

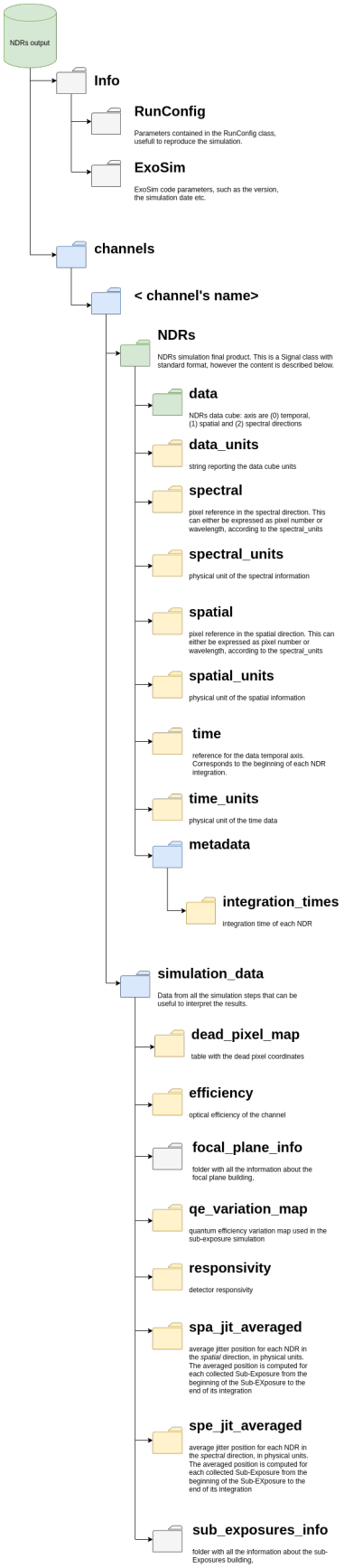
The resulting NDRs look like these.


Photometer - num 1 - time 10.200 s
0
5
10
15
20
25
30
600000
500000
400000
300000
200000
100000
0
Spectrometer - num 1 - time 11.100 s
0
20
40
60
80
100
120
140
150000
100000
50000
0

To produce such plots see also *NDRs Plotter*.

Output description

The following picture describes the structure of the NDRs data output.


NDRs output
Info
RunConfig
Parameters contained in the RunConfig class, usefull to reproduce the simulation.
ExoSim
ExoSim code parameters, such as the version, the simulation date etc.
channels
< channel's name>
NDRs
NDRs simulation final product. This is a Signal class with standard format, however the content is described below.
data
NDRs data cube: axis are (0) temporal, (1) spatial and (2) spectral directions
data_units
string reporting the data cube units
spectral
pixel reference in the spectral direction. This can either be expressed as pixel number or wavelength, according to the spectral_units
spectral_units
physical unit of the spectral information
spatial
pixel reference in the spatial direction. This can either be expressed as pixel number or wavelength, according to the spectral_units
spatial_units
physical unit of the spatial information
time
reference for the data temporal axis. Corresponds to the beginning of each NDR integration.
time_units
physical unit of the time data
metadata
integration_times
integration time of each NDR
simulation_data
Data from all the simulation steps that can be useful to interpret the results.
dead_pixel_map
table with the dead pixel coordinates
efficiency
optical efficiency of the channel
focal_plane_info
folder with all the information about the focal plane building,
qe_variation_map
quantum efficiency variation map used in the sub-exposure simulation
responsivity
detector responsivity
spa_jit_averaged
average jitter position for each NDR in the spatial direction, in physical units. The averaged position is computed for each collected Sub-Exposure from the beginning of the Sub-EXposure to the end of its integration
spe_jit_averaged
average jitter position for each NDR in the spectral direction, in physical units. The averaged position is computed for each collected Sub-Exposure from the beginning of the Sub-EXposure to the end of its integration
sub_exposures_info
folder with all the information about the sub-Exposures building,

2.5.12 NDRs automatic Recipe

Al the steps needed to the production of sub-exposures are already collected in a pre-made pipeline. This is under the *recipes* of *ExoSim*.

```
from exosim import recipes
recipes.CreateNDRs(input_file='./input_file.h5',
                   output_file='./output_file.h5',
                   options_file='your_config_file.xml')
```

The *CreateNDRs* can also be run from console as

```
exosim-ndrs -c your_config_file.xml -i input_file.h5 -o output_file.h5
```

or

```
exosim-ndrs -c your_config_file.xml -i input_file.h5 -o output_file.h5 -P
```

to also run ExoSim *NDRsPlotter*, which is documented in *NDRs Plotter*.

The user can also set the chunk size (see *Instantaneous readout*) using

```
exosim-ndrs -c your_config_file.xml -i input_file.h5 -o output_file.h5 --chunk_size␣
↪N
```

where *N* is the desired size expressed in Mbs.

2.6 Plotters

ExoSim 2 includes some plotters which allows a fast evaluation of the produced data. The default plotter can be run from console as `exosim-plot`.

The scipt includes two plotters: *FocalPlanePlotter* and *RadiometricPlotter*.

2.6.1 Focal plane Plotter

FocalPlanePlotter handles the methods to plot the focal planes produced by *exosim*.

It plots the focal planes of each channel at a specific time. For each channel it adds a `Axes`[35] to a figure. It returns a `Figure`[36] with two rows: on the first row are reported the oversampled focal planes. In the second row are reported the extracted focal plane, where the oversampling is removed. The focal plane plotted is the combination of the source focal plane plus the foreground focal plane.

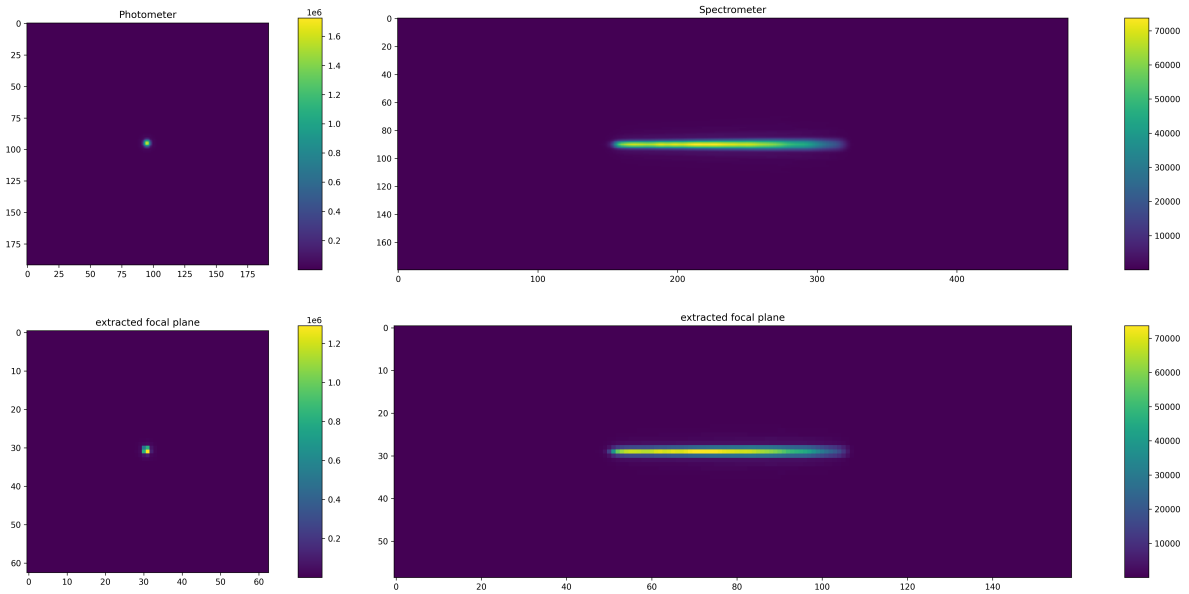
Given the *test_file.h5* produced by Exosim, to plot the focal plane at the first time step, the user can run from console

```
exosim-plot -i test_file.h5 -o plots/ -f -t 0 --plot-scale linear
```

where `-o` is the output directory, `-f` is to run the focal plane plotter (*FocalPlanePlotter*) and `-t` is to select the time step, `--plot-scale` indicate the image scale to use. By default the plot scale is *linear*, but another possible option is *dB*, and the image is plotted as $10 \cdot log_{10}\left(ima/max(ima)\right)$.

The result will be similar to

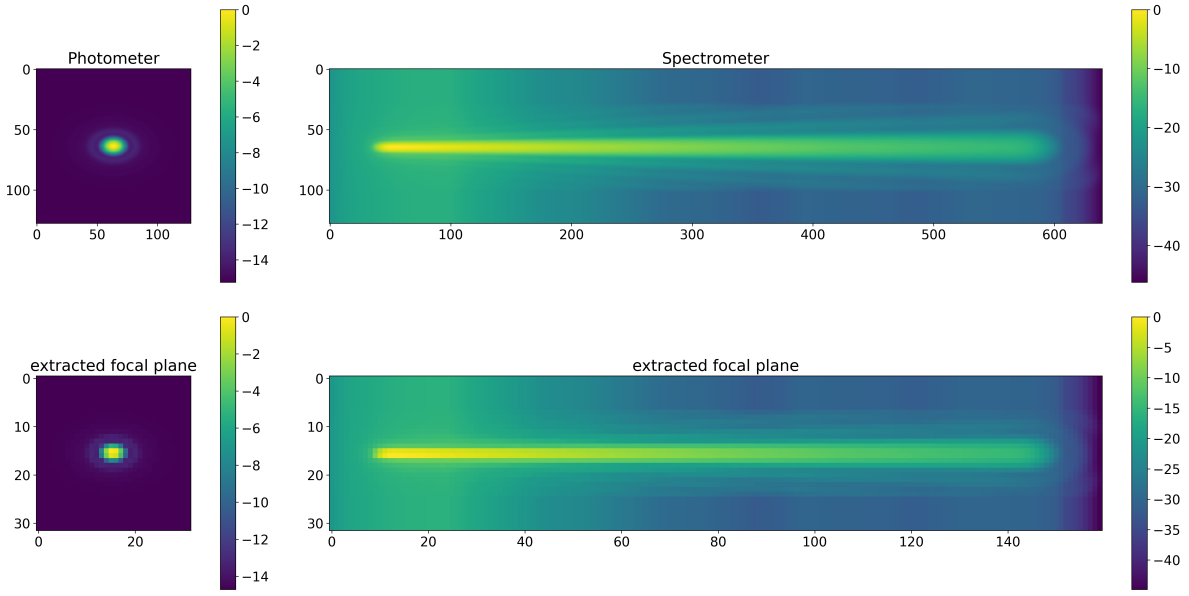
[35] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[36] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure



The same result can be obtained also by using the plotter in a python script:

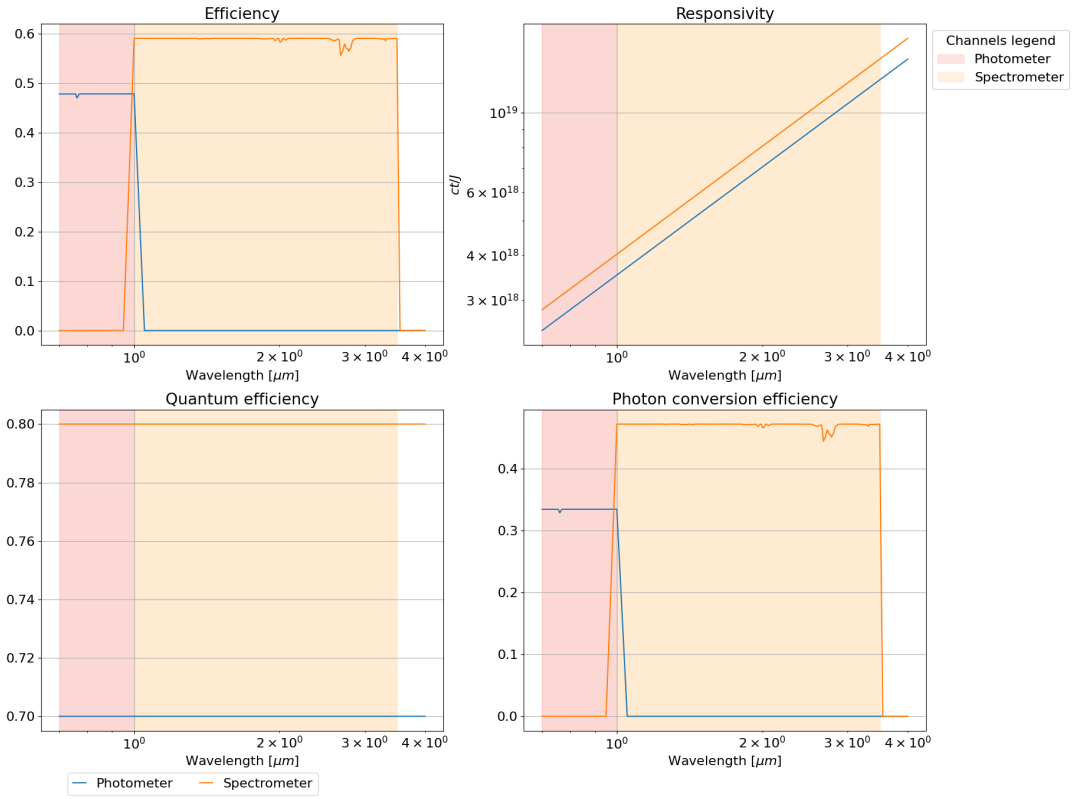
```
from exosim.plots import FocalPlanePlotter
focalPlanePlotter = FocalPlanePlotter(input='./test_file.h5')
focalPlanePlotter.plot_focal_plane(time_step=0, scale='linear')
focalPlanePlotter.save_fig('focal_plane.png')
```

if `-plot-scale` is set to *dB* the result will be



Inside the focal plane plotter a functionality to plot the total efficiency can be found:

```
from exosim.plots import FocalPlanePlotter
focalPlanePlotter = FocalPlanePlotter(input='./test_file.h5')
focalPlanePlotter.plot_efficiency()
focalPlanePlotter.save_fig('efficiency.png')
```



2.6.2 Radiometric Plotter

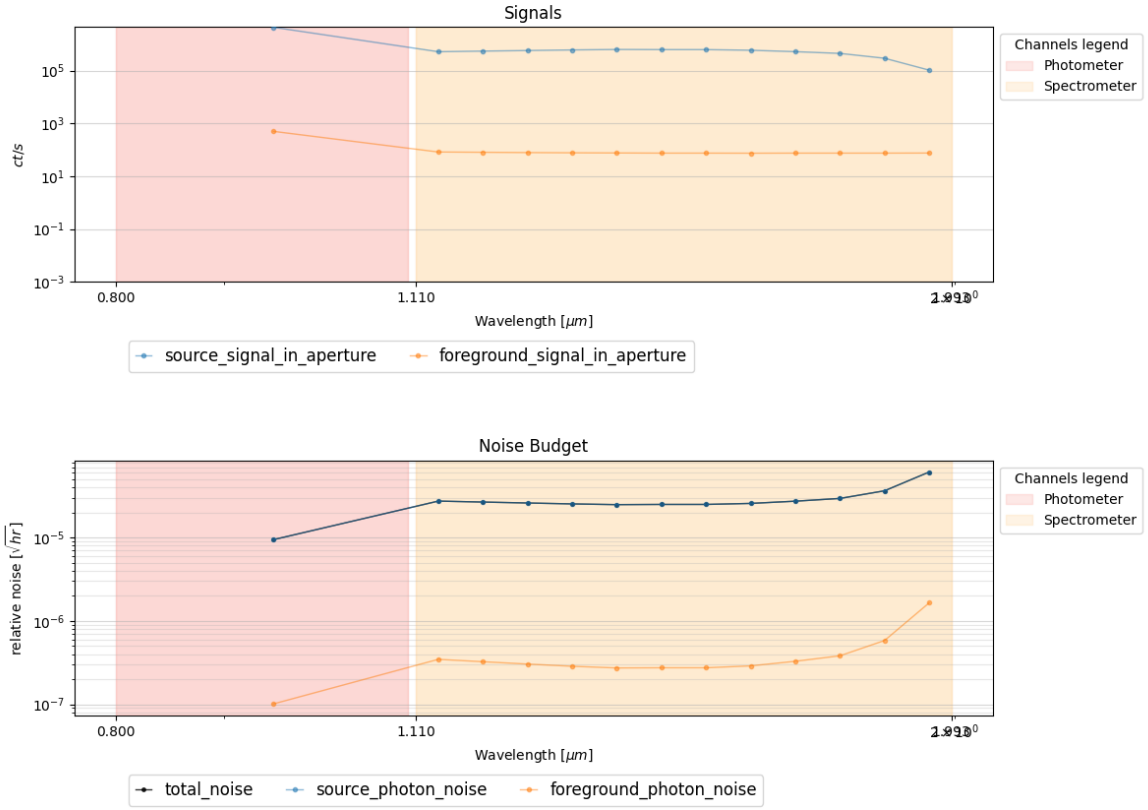
RadiometricPlotter handles the methods to plot the radiometric table produced by *exosim*.

Given the *test_file.h5* produced by Exosim and which includes a radiometric table, to plot the table the user can run from console

```
exosim-plot -i test_file.h5 -o plots/ -r
```

where `-o` is the output directory and `-r` is to run the radiometric plotter (*RadiometricPlotter*).

The result will be similar to

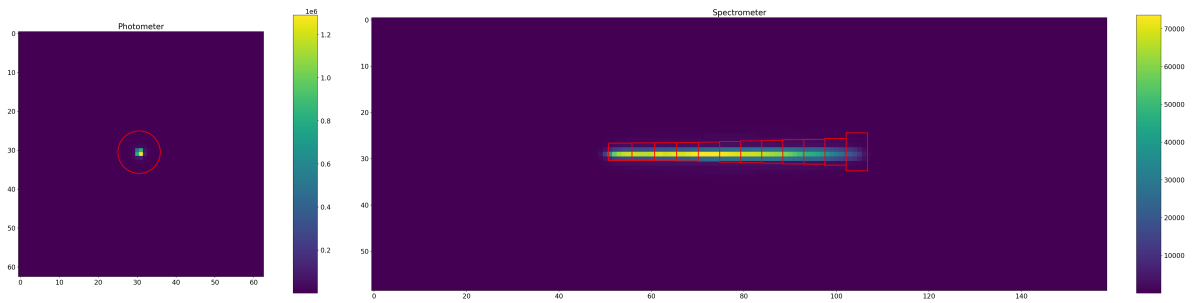


The same result can be obtained also by using the plotter in a python script:

```
from exosim.plots import RadiometricPlotter
radiometricPlotter = RadiometricPlotter(input='./test_file.h5')
radiometricPlotter.plot_table()
radiometricPlotter.save_fig('radiometric.png')
```

The radiometric plotter can also plot the apertures superimposed to the focal planes with

```
from exosim.plots import RadiometricPlotter
radiometricPlotter = RadiometricPlotter(input='./test_file.h5')
radiometricPlotter.plot_apertures()
radiometricPlotter.save_fig('apertures.png')
```



2.6.3 Sub-Exposures Plotter

SubExposuresPlotter handles the methods to plot the Sub-Exposures produced *CreateSubExposures*, as described in *Sub-Exposures*.

Given the *test_se.h5* produced by *ExoSim* and which includes the sub-exposures, to plot them, the user can run from console

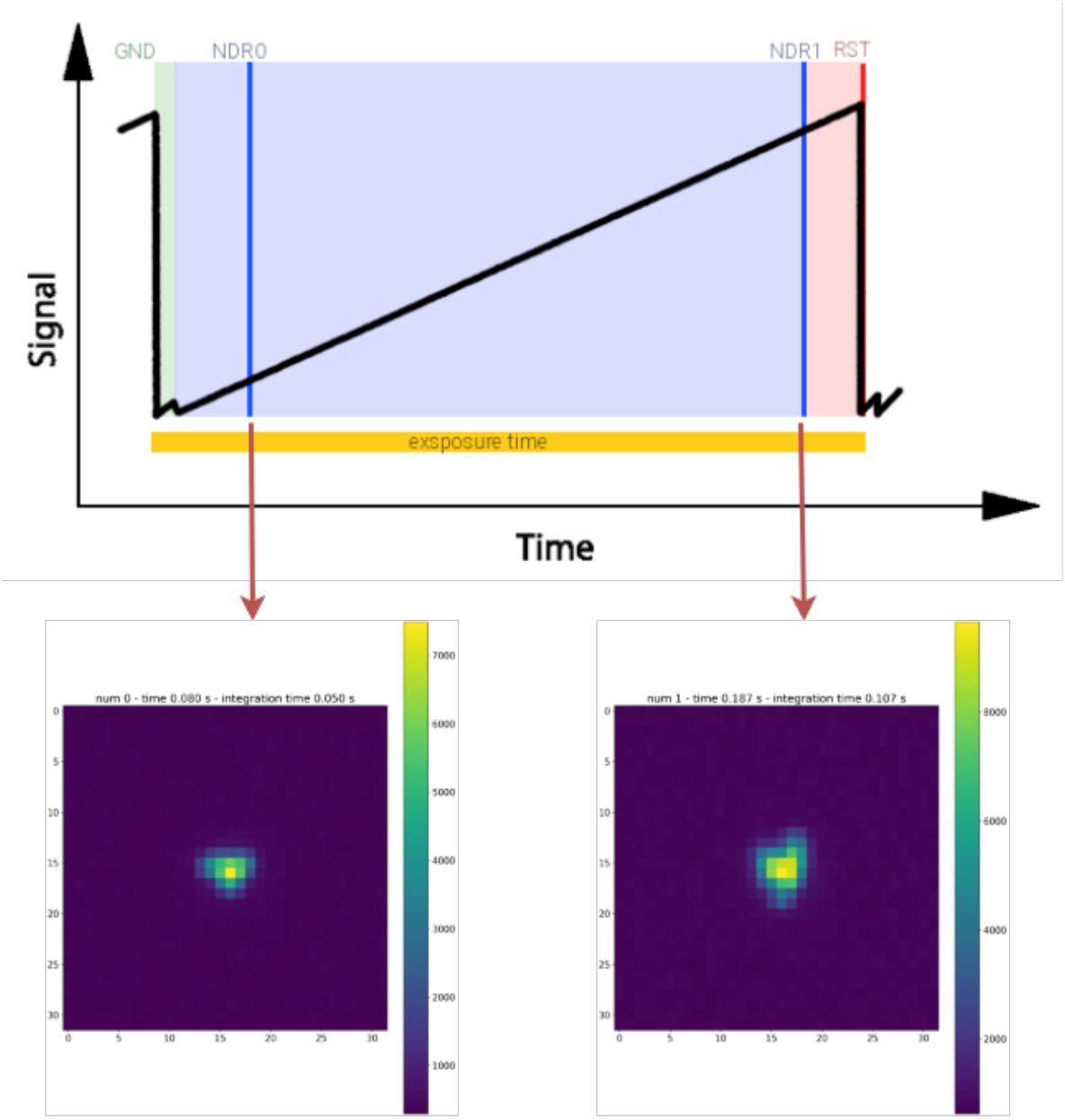
```
exosim-plot -i test_se.h5 -o plots/ --subexposures
```

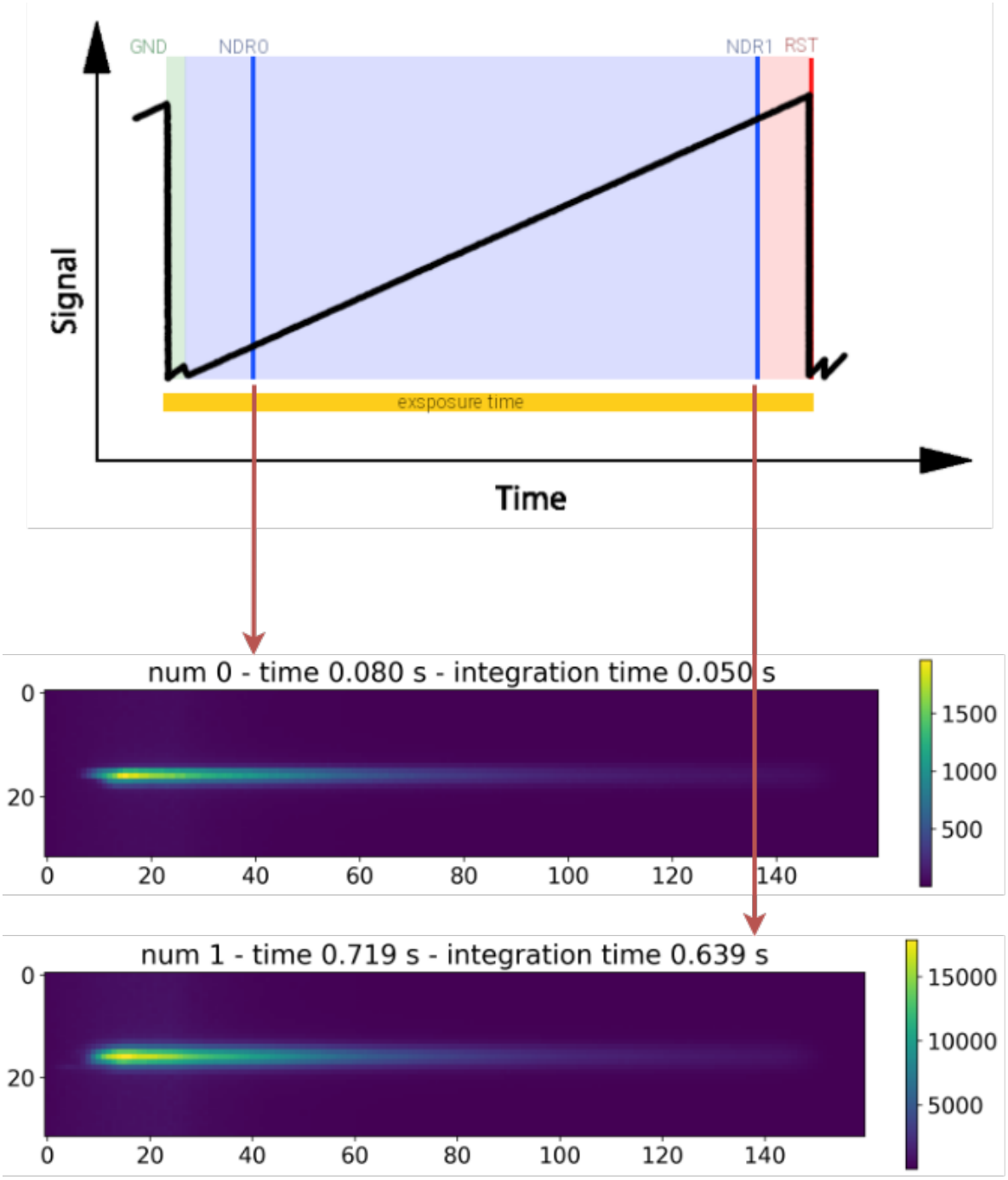
or

```
exosim-plot -i test_se.h5 -o plots/ -s
```

SubExposuresPlotter iteratively store the images of the sub-exposures in the output folder, along with the sub-exposure time (which is the time where the sub-exposure integration ends) and the integration time.

Here we report for example the first and the second sub-exposures, collected using a CDS reading scheme, for both the channels


GND
NDR0
NDR1
RST
Signal
exsposure time
Time
num 0 - time 0.080 s - integration time 0.050 s
num 1 - time 0.187 s - integration time 0.107 s



Note: Because the ExoSim output may contain a lot of sub-exposures, This plotter only produces images of the sub-exposures of the first exposure (the first ramp).

2.6.4 NDRs Plotter

NDRsPlotter handles the methods to plot the Sub-Exposures produced *CreateNDRs*, as described in *NDRS*.

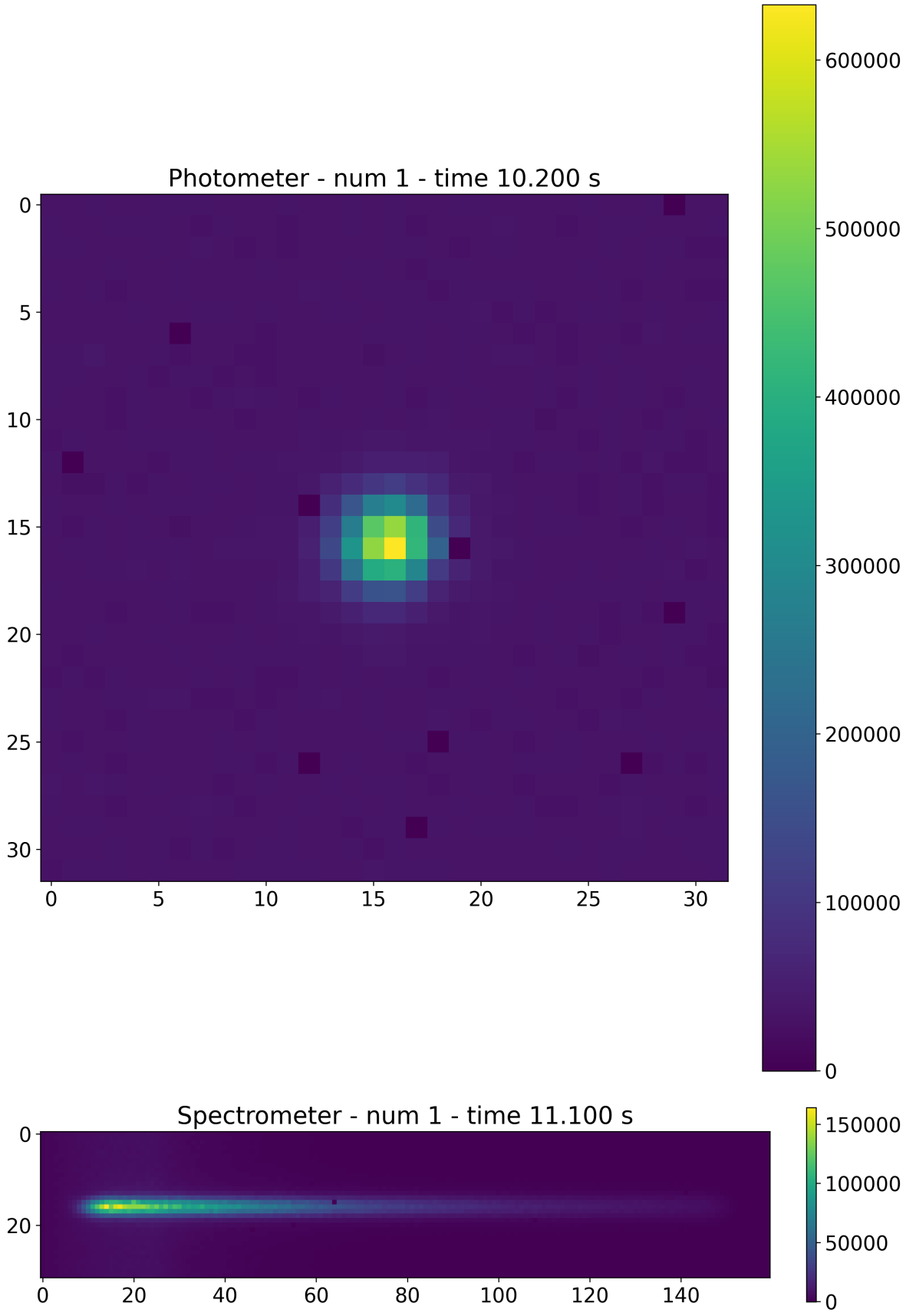
Given the *test_ndrs.h5* produced by *ExoSim* and which includes the NDRs, to plot them, the user can run from console

```
exosim-plot -i test_ndrs.h5 -o plots/ -ndrs
```

or

```
exosim-plot -i test_ndrs.h5 -o plots/ -n
```

NDRsPlotter iteratively stores the images of the NDRs in the output folder, along with the NDR exposure time.


Photometer - num 1 - time 10.200 s
0
5
10
15
20
25
30
600000
500000
400000
300000
200000
100000
0
Spectrometer - num 1 - time 11.100 s
0
20
40
60
80
100
120
140
150000
100000
50000
0

2.7 ExoSim Tools

In this section we present and describe a set of tool we included in ExoSim 2 to help the user to perform its simulations.

All the parameters to run the tools are parsed from and *.xml* file. In these examples, we assume the input file is called *tools_input_example.xml*. This file should mimic the general ExoSim input:

```
<root>

    <ConfigPath> path/to/your/configs </ConfigPath>

    <channel> channel 1
        ...
    </channel>

    <channel> channel 2
        ...
    </channel>

</root>
```

2.7.1 List of tools

Quantum efficiency variation map

The *QuantumEfficiencyMap* tool allows the creation of quantum efficiency variation maps that can be used in *ExoSim `(see :ref: `qe_map*).

The tool assumes that the QE has a normal distribution around the median (which is given by the *Estimate responsivity*) and it requires the standard deviation as input. Then it randomises the pixel QE according to the indicated distribution.

The following configuration are to be set into the tool input parameters

```
<channel> channel_name
    <detector>
        <qe_sigma> 0.1 </qe_sigma>
```

(continues on next page)

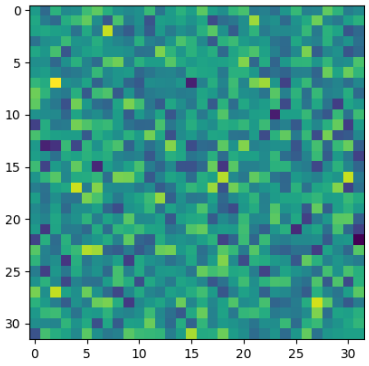
(continued from previous page)

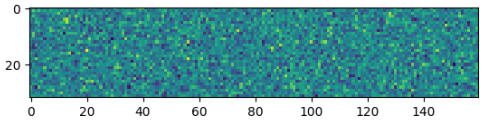
```
    </detector>
</channel>
```

Then the tool can be run as

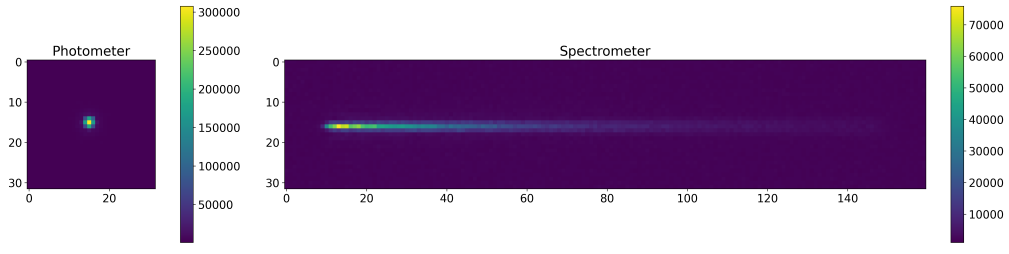
```
import exosim.tools as tools

tools.QuantumEfficiencyMap(options_file='tools_input_example.xml',
                           output='output_qe_map.h5')
```

The result will be like

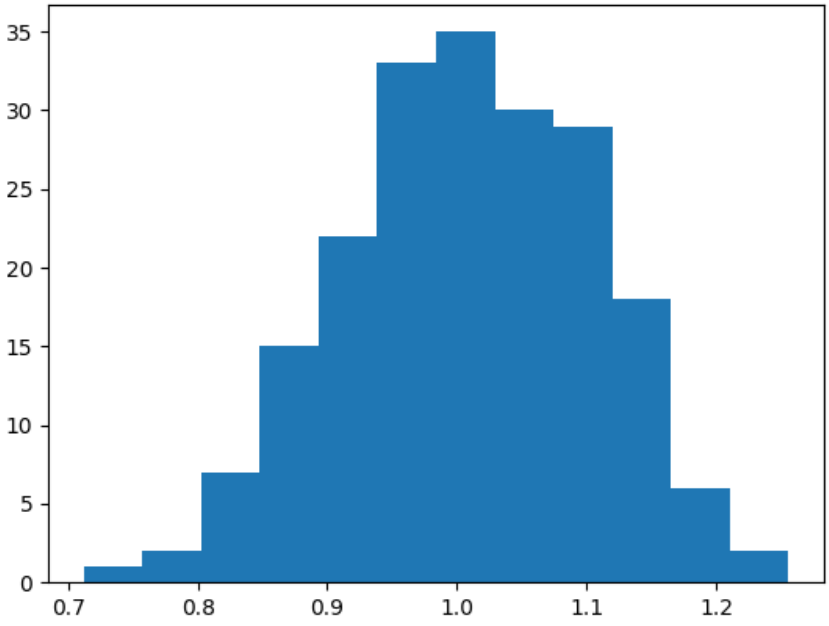




and then applied as

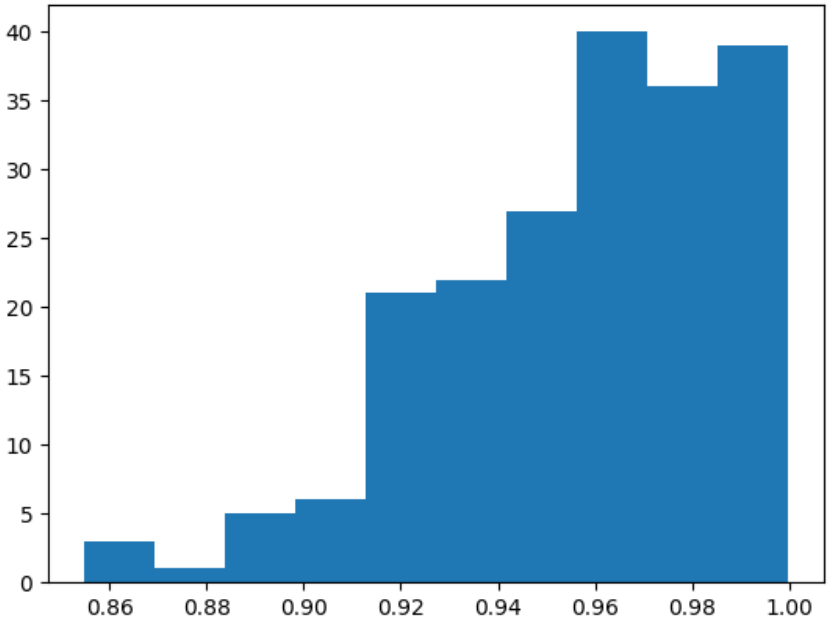


with the QE normalisation distributed as



The default *QuantumEfficiencyMap* also allow for QE degradation. Given the the amplitude of the degradation an the time scale, it creates a randomised aging factor for each pixel and interpolate the QE efficiency map in time to age accordingly.

```
<channel> channel_name
    <detector>
        <qe_sigma> 0.1 </qe_sigma>
        <qe_aging_factor> 0.01 </qe_aging_factor>
        <qe_aging_time_scale unit="hr"> 5 </qe_aging_time_scale>
    </detector>
</channel>
```



The resulting QE at 5 hr will be the product between the QE map computed with `<qe_sigma> 0.1 </qe_sigma>` and the aged map.

Pixels Non-Linearity

This tools helps the user to find the pixel non-linearity coefficients to as inputs for ExoSim, starting from the measurable pixel non-linearity correction.

In fact, the detector non linearity model, is usually written as polynomial such as

$$Q_{det} = Q \cdot (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi_t$, with ϕ being the number of electrons generated and t being the elapsed time.

Pixels Non-Linearity from scratch

The *PixelsNonLinearity* tool retrieves the a_i coefficients, starting from physical assumptions.

The detector non linearity model, is written as polynomial such as

$$Q_{det} = Q \cdot (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi_t$, with ϕ being the number of electrons generated and t being the elapsed time.

Considering the detector as a capacitor, the charge Q_{det} is given by

$$Q_{det} = \phi\tau \cdot \left(1 - e^{-Q/\phi\tau}\right)$$

where ϕ is the charge generated in the detector pixel, and τ is the capacitor time constant. In fact the product $\phi\tau$ is constant Q is the response of a linear detectror is given by $Q = \phi t$

The detector is considered saturated when the charge Q_{det} at the well depth $Q_{det,\,wd}$ differs from the ideal well depth Q_{wd} by 5%.

$$Q_{det} = (1 - 5\%)Q_{wd}$$

Then

$$\phi\tau \cdot \left(1 - e^{-Q_{wd}/\phi\tau}\right) = (1 - 5\%)Q_{wd}$$

This equation can be solved numerically and gives

$$\frac{Q_{wd}}{\phi\tau} \sim 0.103479$$

Therefore the detector collected charge is given by

$$Q_{det} = \frac{Q_{wd}}{0.103479} \cdot \left(1 - e^{\frac{-0.103479\,Q}{Q_{wd}}}\right)$$

Which can be approximated by a polynomial of order 4 as

$$\begin{aligned} Q_{det} = Q \Bigg[1 &- \frac{1}{2!}\frac{0.103479}{Q_{wd}}Q \\ &+ \frac{1}{3!}\left(\frac{0.103479}{Q_{wd}}\right)^2 Q^2 \\ &- \frac{1}{4!}\left(\frac{0.103479}{Q_{wd}}\right)^3 Q^3 \\ &+ \frac{1}{5!}\left(\frac{0.103479}{Q_{wd}}\right)^4 Q^4 \end{aligned}$$

The results are the coefficients for a 4-th order polynomial:

$$Q_{det} = Q \cdot (a_1 + a_2 \cdot Q + a_3 \cdot Q^2 + a_4 \cdot Q^3 + a_5 \cdot Q^4)$$

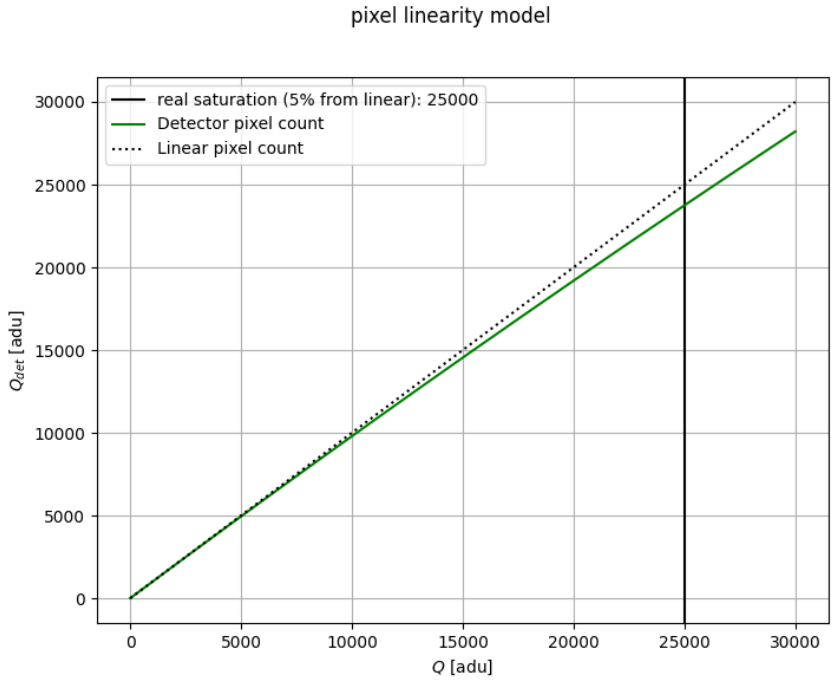
The user should indicate the expected non-linearity shape, by setting the saturation parameter, called *well_depth*:

```
<channel> channel_name
    <detector>
        <well_depth> 25000 </well_depth>
    </detector>
</channel>
```

Then the tool can be run as

```
import exosim.tools as tools

tools.PixelsNonLinearity(options_file='tools_input_example.xml',
                         output='pnl_map.h5')
```

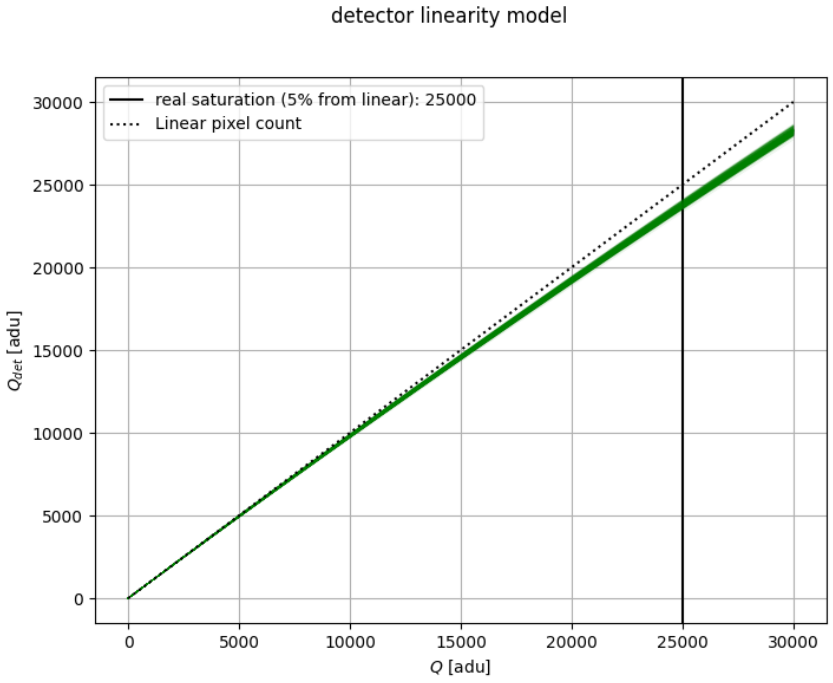
With the given example, we obtain the following expected non-linearity shape:



However, each pixel is different, and therefore, this class also produces a map of the coefficient for each pixel. Each coefficient is normally distributed around the mean value, with a standard deviation indicated in the configuration. If no standard deviation is indicated, the coefficients are assumed to be constant.

```
<channel> channel_name
    <detector>
        <spatial_pix> 200 </spatial_pix>
        <spectral_pix> 200 </spectral_pix>
        <pnl_coeff_std> 0.005 </pnl_coeff_std>
    </detector>
</channel>
```

To obtain the map we added to the configuration the detector sizes and the standard deviation of the coefficients.



The code output is a map of a_i coefficients for each pixel, which can be injected into *ApplyPixelsNonLinearity*.

Pixels Non-Linearity from correcting coefficients

Let's write again the detector non linearity model as

$$Q_{det} = Q \triangle (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi t$, with ϕ being the number of electrons generated and t being the elapsed time. In the equation above, $\triangle$ is the operator used to defined the relation between Q_{det} and Q, which depends on the definition of the coefficients a_i (see also equation below).

However, it is usually the inverse operation that is known, as it's coefficients are measurable empirically:

$$Q = Q_{det} \bigtriangledown (b_1 + \sum_{i=2} b_i \cdot Q_{det}^i)$$

Where $\bigtriangledown$ is the inverse operator of $\triangle$. Depending on the way the non linearity is estimated, the operator can either be a division ($\div$) or a multiplication ($\times$). If not specified, a division is assumed.

The *PixelsNonLinearityFromCorrection* can determine the coefficients b_i from the measured non linearity correction.

The b_i correction coefficients should be listed in the configuration file using the *pnl_coeff* keyword in increasing alphabetical order: *pnl_coeff_a* for b_1, *pnl_coeff_b* for b_2, *pnl_coeff_c* for b_3, *pnl_coeff_d* for b_4, *pnl_coeff_e* for b_5 and so on. The user can list any number of correction coefficients, and they will be automatically parsed. Please, note that using this notation, b_1 is not forced to be the unity.

```
<channel> channel_name
    <detector>
        <well_depth> 25000 </well_depth>
```

(continues on next page)

(continued from previous page)

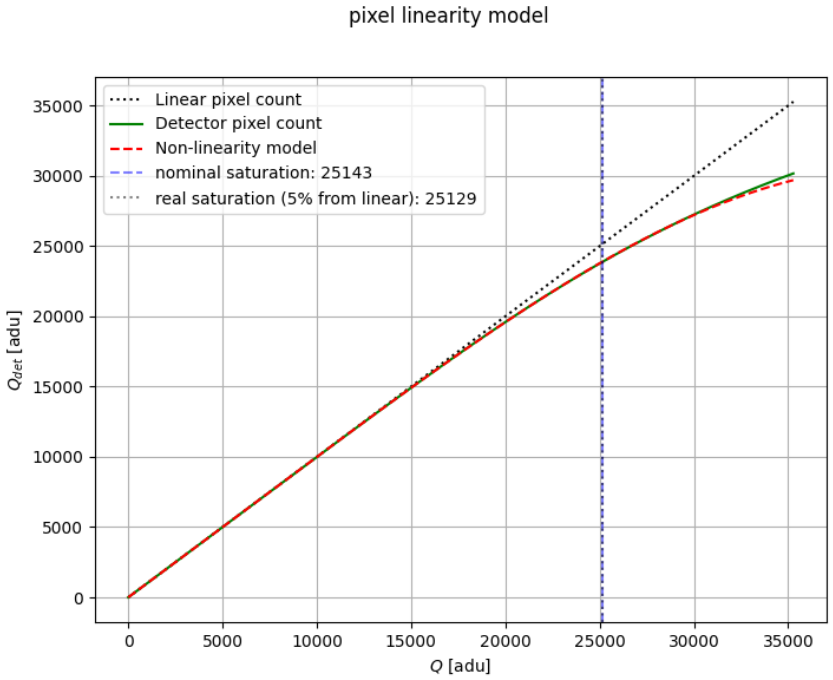
```
        <pnl_coeff_a>  1.00117667e+00 </pnl_coeff_a>
        <pnl_coeff_b> -5.41836850e-07 </pnl_coeff_b>
        <pnl_coeff_c> 4.57790820e-11 </pnl_coeff_c>
        <pnl_coeff_d> 7.66734616e-16 </pnl_coeff_d>
        <pnl_coeff_e> -2.32026578e-19 </pnl_coeff_e>
        <pnl_correction_operator> / </pnl_correction_operator>

        <pnl_coeff_std> 0.005 </pnl_coeff_std>
    </detector>
</channel>
```

as example of non linearity, we used the parameters from Hilbert 2009: "WFC3 TV3 Testing: IR Channel Nonlinearity Correction" (link[37]).

This class will restrieve the a_i coefficients, starting from the the indicated b_i. The results are the coefficients for a 4-th order polynomial:

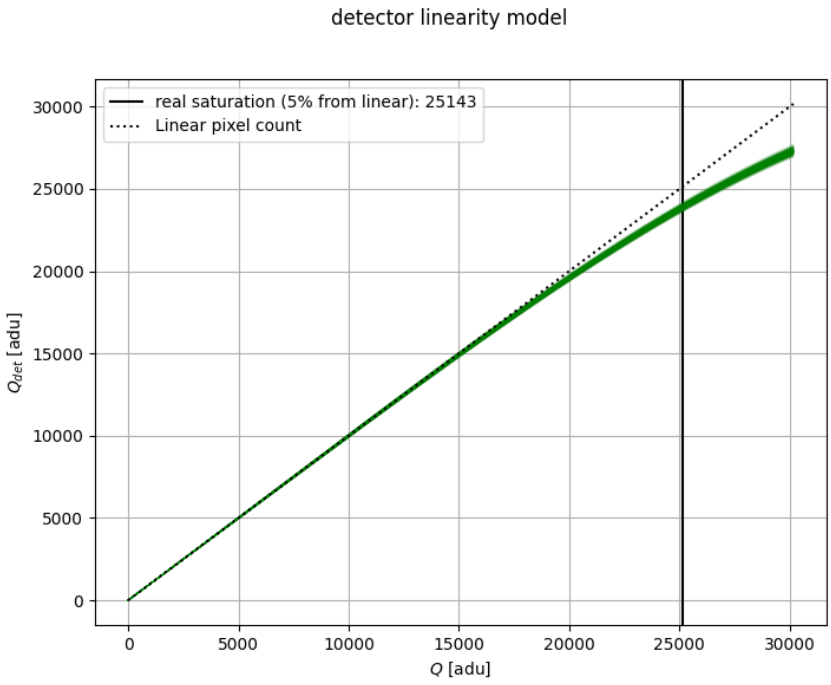
$$Q_{det} = Q \cdot (a_1 + a_2 \cdot Q + a_3 \cdot Q^2 + a_4 \cdot Q^3 + a_5 \cdot Q^4)$$

With the given example, we obtain the following expected non-linearity shape:



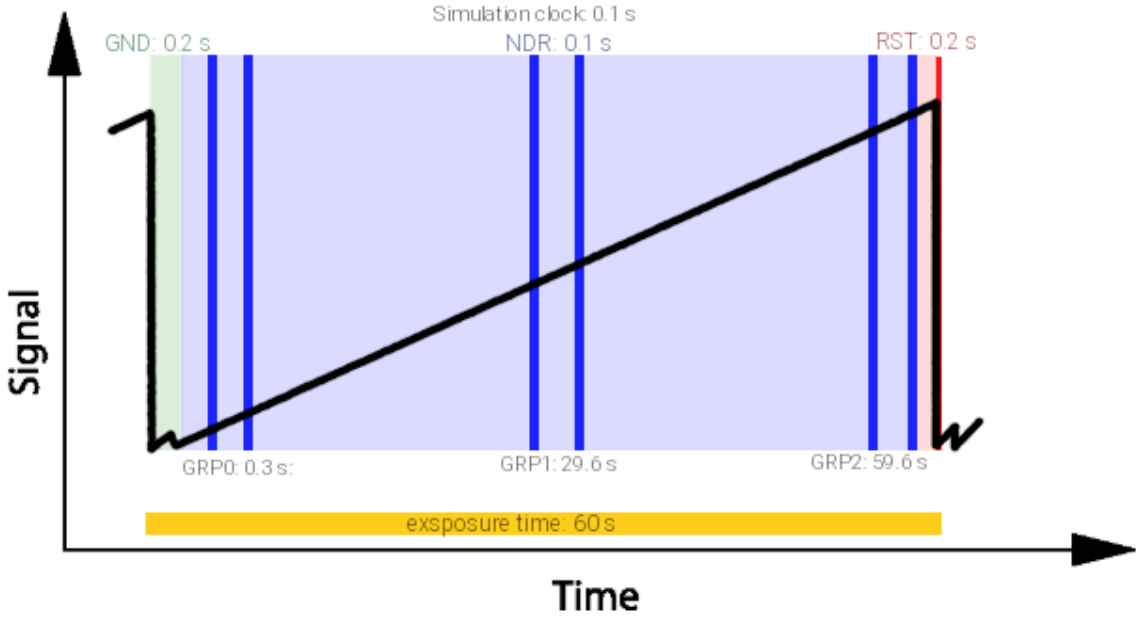
However, each pixel is different, and therefore, this class also produces a map of the coefficient for each pixel as before.

[37] https://www.stsci.edu/files/live/sites/www/files/home/hst/instrumentation/wfc3/documentation/instrument-science-reports-isrs/_documents/2008/WFC3-2008-39.pdf



Readout Scheme Calculator

Let's assume we want to produce the following readout scheme, but know only the following information.



The ramp is sampled at *readout_frequency* cadence, defined in *Sub-Exposures*. In this example we want to spend $0.2\,s$ seconds in ground (GND) state and $0.2\,s$ before the reset state (RST). The NDRs are read at a constant cadence of $0.1\,s$. Then we want to have 3 groups which divide the residual ramp into equal parts. Each group consists of 2 NDRs separated the time needed to read a NDR.

To produce the reading scheme, we need to know the time spent between groups (which is reported in the bottom

part of the figure expressed as time from the start of the simulation). But we also need to translate our human-readable units into the simulation clock unis needed for *ComputeReadingScheme*, as described in *Reading Scheme*. Finally, we want to make good use of the ramp sampling, and therefore we don't want to saturate the detector. All of this is handled by *ReadoutSchemeCalculator*.

First we need to translate all the known parameter in the following description in the channel section of the tool input file:

```
<channel> channel name
    <readout>
        <n_NRDs_per_group> 2 </n_NRDs_per_group>
        <n_groups>  3 </n_groups>
        <readout_frequency unit ='s'> 0.1 </readout_frequency>
        <Ground_time unit ='s'> 0.2 </Ground_time>
        <Reset_time unit ='s'> 0.2 </Reset_time>
    </readout>
</channel>
```

The user can also set the *readout_frequency* in units of Hz instead of s.

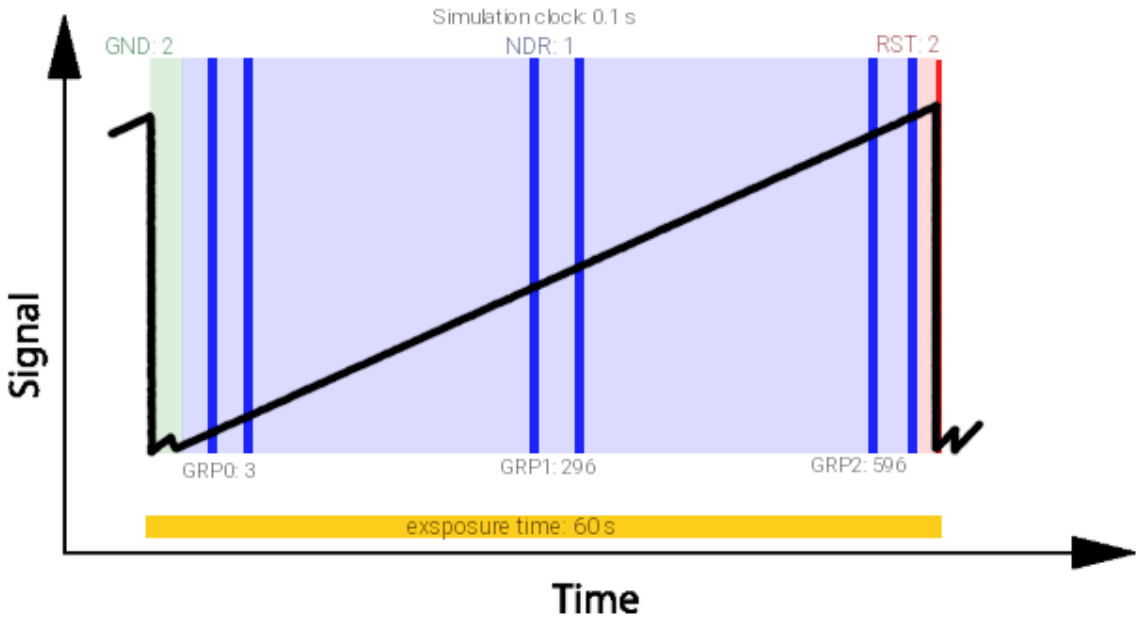
Obviously, to estimate the saturation time some other input is needed: the focal planes. We assume here that the focal plane are stored in *input_file.h5*:

```
import exosim.tools as tools

tools.ReadoutSchemeCalculator(options_file = 'tools_input_example.xml',
                              input_file='input_file.h5')
```

The code will then suggest the inputs to write on the payload configuration file. In this case, according to the figure, the results will be

```
<channel> channel name
    <readout>
        <readout_frequency unit="Hz">10</readout_frequency>
        <n_NRDs_per_group> 2 </n_NRDs_per_group>
        <n_groups> 3 </n_groups>
        <n_sim_clocks_Ground> 2 </n_sim_clocks_Ground>
        <n_sim_clocks_first_NDR> 1 </n_sim_clocks_first_NDR>
        <n_sim_clocks_Reset> 2 </n_sim_clocks_Reset>
        <n_sim_clocks_groups> 296 </n_sim_clocks_groups>
    </readout>
</channel>
```

Which will results in the following scheme



Also, the user can set a custom exposure time to use instead of the saturation time:

```
<channel> channel name
    <readout>
        <exposure_time unit="s"> 60 </exposure_time>
    </readout>
</channel>
```

Dead pixels map

The *DeadPixelsMap* tool allows the creation of dead pixels maps that can be used in *ExoSim*.

The tool requires the number of dead pixels in the channel to distribute them randomly over the focal plane.

The following configuration are to be set into the tool input parameters

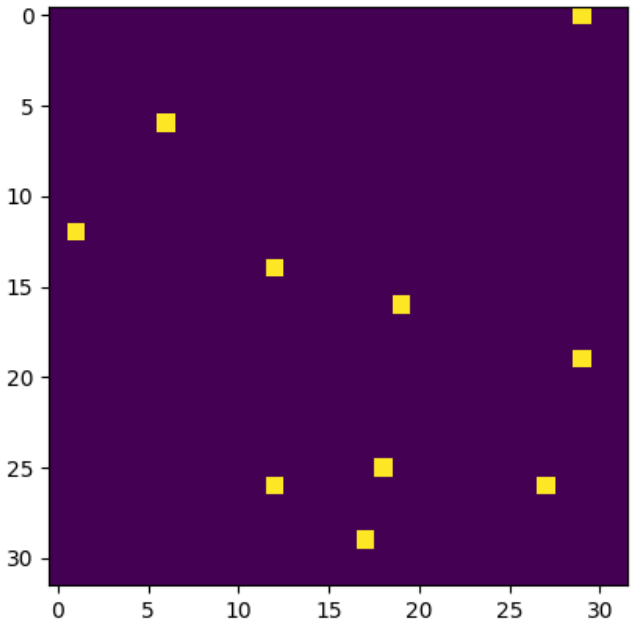
```
<channel> channel_name
    <detector>
        <dp_mean > 10 </dp_mean>
    </detector>
</channel>
```

Then the tool can be run as

```
import exosim.tools as tools

tools.DeadPixelsMap(options_file='tools_input_example.xml',
                            output='data/payload')
```

And it produces a *.csv* file for each channel with the coordinates of the dead pixels.

The result will be like



The tool can also be used in the case the exact number of the dead pixel is not known. In this case the inputs should include the uncertainty on this number.

```
<channel> channel_name
    <detector>
        <dp_mean > 10 </dp_mean>
        <dp_sigma > 1 </dp_sigma>
    </detector>
</channel>
```

The code will then select a random number of dead pixels centered in the *dp_meam* value and normally distributed with *dp_sigma* standard deviation.

The produced *.csv* files can be used as input for `ApplyDeadPixelsMap`, described in *Dead pixels map*.

ADC gain estimator

The sub-exposure measured signals are in counts units, however we know that the detector output is reported in adu units. ExoSim simulates the Analog to Digital Converter (ADC) thanks to `AnalogToDigital`, which converts the $counts$ units of sub-exposures into adu units of NDRs (see *Analog to Digital conversion*)

ADC output are unsigned integers defined in a certain number of bits. Because this objects can represent numbers up to a maximum value of $2^{n_{bits}} - 1$, we need a conversion factor to rescale the floats value NDRs to fit in the new data type range. This conversion factor is defined by *ADC_gain*. as:

$$g_{ADC} = \frac{2^{n_{bits}} - 1}{ADC_{max}}$$

where n_{bits} is the number of bits set for the ADC and ADC_{max} is the maximum value we want the ADC to handle. Assuming a 16 bits ADC, $2^{16} - 1 = 65535$ and a desired $ADC_{max} = 120000$, then we have $g_{ADC} = \frac{65535}{120000} = 0.546125$.

This condition can be expressed in the tools configuration files as:

```
<channel> channel_name
    <detector>
        <ADC_num_bit> 16 </ADC_num_bit>
        <ADC_max_value> 120000 </ADC_max_value>
    </detector>
</channel>
```

CHAPTER

THREE

DEVELOPER GUIDE

3.1 Contributing guidelines

3.1.1 Code bugs and issues

If you notice a bug or an issue, the best thing to do is to open an issue on the GitHub repository[38].

3.1.2 Coding conventions

The code has been developed following the PeP8[39] standard and the python Zen[40]. If you have any doubts, try

```
import this
```

3.1.3 Documentation

Every function or class should be documented using docstrings which follow numpydoc[41] structure. This web page is written using the reStructuredText[42] format, which is parsed by sphinx[43]. If you want to contribute to this documentation, please refer to sphinx[44] documentation first. You can improve this pages by digging into the *docs* directory in the source.

To help the contributor in writing the documentation, we have created two nox[45] sessions:

```
$ nox -s docs
$ nox -s docs-live
```

The first will build the documentation and the second will build the documentation and open a live server to see the changes in real time. The live server can be accessed at http://127.0.0.1:8000/

Note: To run a `nox` session, you need to install it first. You can do it by running:

```
$ pip install nox
```

[38] https://github.com/arielmission-space/ExoSim2-public/issues
[39] https://www.python.org/dev/peps/pep-0008/
[40] https://www.python.org/dev/peps/pep-0020/
[41] https://numpydoc.readthedocs.io/en/latest/
[42] https://docutils.sourceforge.io/rst.html
[43] https://www.sphinx-doc.org/en/master/
[44] https://www.sphinx-doc.org/en/master/
[45] https://nox.thea.codes/en/stable/

3.1.4 Testing

Unit-testing is very important for a code as big as *ExoSim 2*. At the moment *ExoSim* is tested using unittest[46]. If you add functionalities, please add also a dedicated test into the *tests* directory. All the tests can be run with

```
python -m unittest discover -s tests
```

3.1.5 Logging

Logging is important when coding, hence we include a *exosim.log.logger.Logger* class to inherit.

```
import exosim.log as log

class MyClass(log.Logger):
    ...
```

Now the new class has the following methods from the main *Logger* class.

```
self.info()
self.debug()
self.warning()
self.error()
self.error()
```

where the arguments shall be strings. The logger output will be printed on the run or stored in the log file, if the log file option is enabled. To enable the log file, the user can refer to *exosim.log.addLogFile*.

Note: The logger here produced is inspired by the logging classes in TauREx3[47] developed by Ahmed Al-Refaie.

The user can also set the level of the printed messaged using *exosim.log.setLogLevel*, or enable or disable the messaged with *exosim.log.enableLogging* or *exosim.log.disableLogging*

If the contributor wants to trace every time a function is called, the `exosim.log.logger.traced` decorator[48] comes handy:

```
import exosim.log as log

@log.traced
def my_func(args):
    ...
```

This will produce a log everytime the function is entered and exited with a *TRACE* logging level.

[46] https://docs.python.org/3/library/unittest.html
[47] https://taurex3-public.readthedocs.io/en/latest/
[48] https://realpython.com/primer-on-python-decorators/

3.1.6 Versioning conventions

The versioning convention used is the one described in Semantic Versioning (semver[49]) and is compliant to PEP440[50] standard. In the [Major].[minor].[patch] scheme, for each modification to the previous release we increase one of the numbers.

- *Major* is to increased only if the code in not compatible anymore with the previous version. This is considered a Major change.
- *minor* is to increase for minor changes. These are for the addition of new features that may change the results from previous versions. This are still hard edits, but not enough to justify the increase of an *Major*.
- *patch* are the patches. This number should increase for any big fixed, or minor addition or change to the code. It won't affect the user experience in any way.

Additional information can be added to the version number using the following scheme: [Major].[minor].[patch]-[Tag].[update]

- *Major* is to increased only if the code in not compatible anymore with the previous version. This is considered a Major change.
- *minor* is to increase for minor changes. These are for the addition of new features that may change the results from previous versions. This are still hard edits, but not enough to justify the increase of an *Major*.
- *patch* are the patches. This number should increase for any big fixed, or minor addition or change to the code. It won't affect the user experience in any way.
- *Tag* is a string that can be added to the version number. It can be used to indicate the type of release, or the type of change. For example, *alpha*, *beta*, *release* or *dev* can be used to indicate that the version is not stable yet.
- *updated* is a number to increase for all the changes that are not related to the code patch. This is usefull for development purposes, to keep track of the number of updates since the last release.

See also *What versioning system is used?*.

The version number is stored in the *version* keyword of the *setup.cfg* file.

Automatic update of the version number

The version number is automatically updated by bump2version[51]. The user can update the version number by running

```
bump2version --current-version Major.minor.patch --new-version  Major.minor.patch␣
↪patch
```

Note: To run `bump2version`, you need to install it first. You can do it by running:

```
$ pip install bump2version
```

The same can be done for *minor* and *major* changes.

However, the **best practice** is to use the routine included in the nox[52] configuration file as dedicated session.

[49] https://semver.org/spec/v2.0.0.html
[50] https://www.python.org/dev/peps/pep-0440/
[51] https://github.com/c4urself/bump2version
[52] https://nox.thea.codes/en/stable/

```
nox -s release -- Major.minor.patch -- "short description of the changes"
```

which will update the version number, commit the changes and tag the commit with the new version number. The short description of the changes will be added to the Changelog file as the new version title. It should be a couple of words describing the changes. The `nox` command will also update the Changelog files.

Note: The `nox` command will not create a new release on GitHub. This has to be done manually.

Note: To run a `nox` session, you need to install it first. You can do it by running:

```
$ pip install nox
```

Before running the `nox` command to update the version number, the user should run the linters included in the pre-commit routine to avoid errors. It can be run with

```
$ pre-commit run --all-files
```

if `pre-commit` is installed, or

```
nox -s lint
```

3.1.7 Source Control

The code is hosted on GitHub (https://github.com/arielmission-space/ExoSim2-public) and structured as follows.

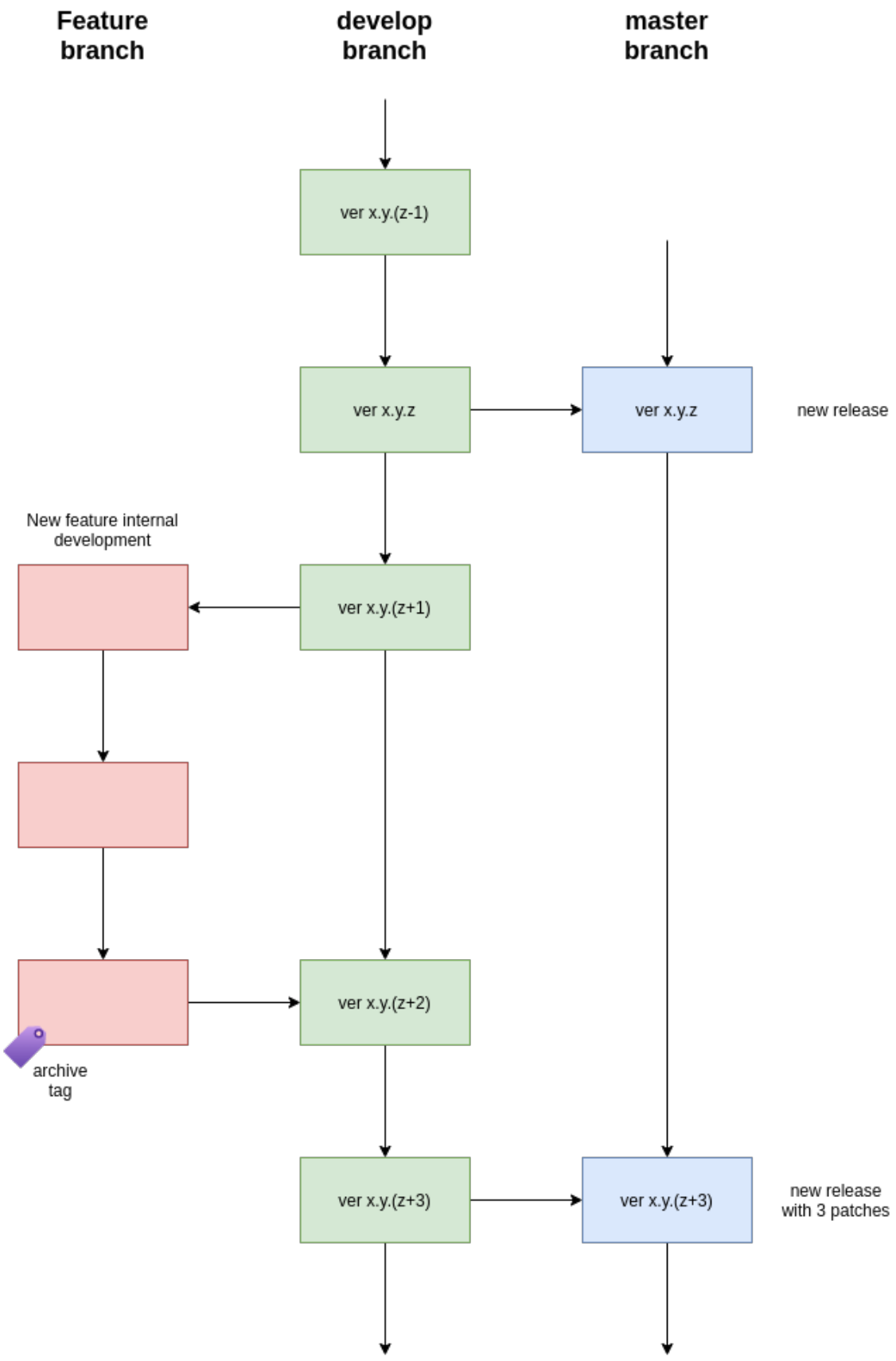
The source has two main branches:

- `master`: this is used for stable and releases. It is the public branch and should be handled carefully.
- `develop`: this is the working branch where the new features are tested before being moved on to the *master* branch and converted into releases.

Adding new features

New features can be added to the code following the schemes designed above.

If the contributor has writing rights to the repository, should create a new branch starting from the *develop* one. In the new *feature* branch the user should produce the new functionalities, according to the above guidelines. When the feature is ready, the branch can be merged into the official *develop* one.


Feature branch
develop branch
master branch
ver x.y.(z-1)
ver x.y.z
ver x.y.z
new release
New feature internal development
ver x.y.(z+1)
ver x.y.(z+2)
archive tag
ver x.y.(z+3)
ver x.y.(z+3)
new release with 3 patches

To create the new feature starting from the current develop version, the contributor should run

```
git checkout develop
git checkout -b feature/<branchname>
```

The completed feature can then be added to the develop. This can be done in two ways: by a merge or a pull[53] request.

Merge

A merge is a soft way to add a new feature to another branch. Performing a Merge mean that the change will be applied if according to gitHub there two branch are compatible.

```
git merge develop
git checkout develop
git merge feature/<branchname>
git push
```

Once a feature is completed and merged, the contributor should *archive* ` the branch and remove it, to keep the repository clean. The usual procedure is:

```
git tag archive/<branchname> feature/<branchname>
git push --tags
git branch -d feature/<branchname>
```

Remember to delete the branch also from the remote repository. If needed, the feature branch can be restored as

```
git checkout -b <branchname> archive/<branchname>
```

Fixing bug

The procedure to fix a bug is similar to the one for adding a new feature.

Create the new branch starting from the current develop versionn

```
git checkout develop
git checkout -b fix/<branchname>
```

Then, once the bug is fixed, the branch can be merged into the official *develop* one.

```
git merge develop
git checkout develop
git merge fix/<branchname>
git push

git tag archive/fix/<branchname> fix/<branchname>
git push --tags
git branch -d fix/<branchname>
```

[53] https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request

Pull request

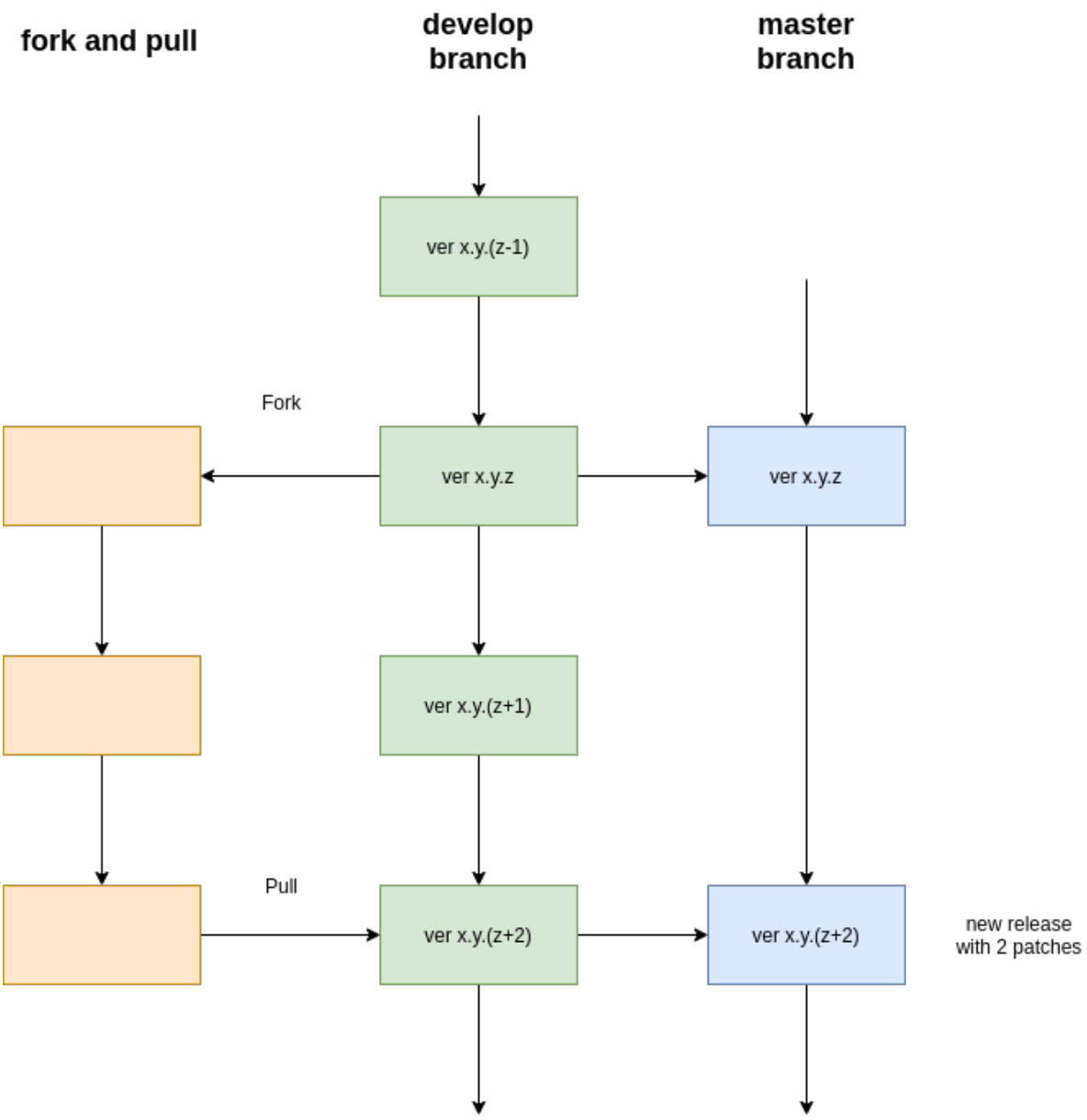
A similar result can be obtained by the gitHub web interface. When the feature is completed, the contributor can visit the branches[54] tab of the gitHub page. From there it is possible to advance a pull[55] request by clicking on the bottom on the right of the branch we want to merge. Then select *develop* as destination branch and confirm. GitHub will run all the python tests written for *ExoSim 2* and check for compatibility between the two branches. If everything is ok a merge can be confirmed.

Then, in the branches[56] page is possible to delete the new feature branch, if it is not useful anymore.

Fork and Pull

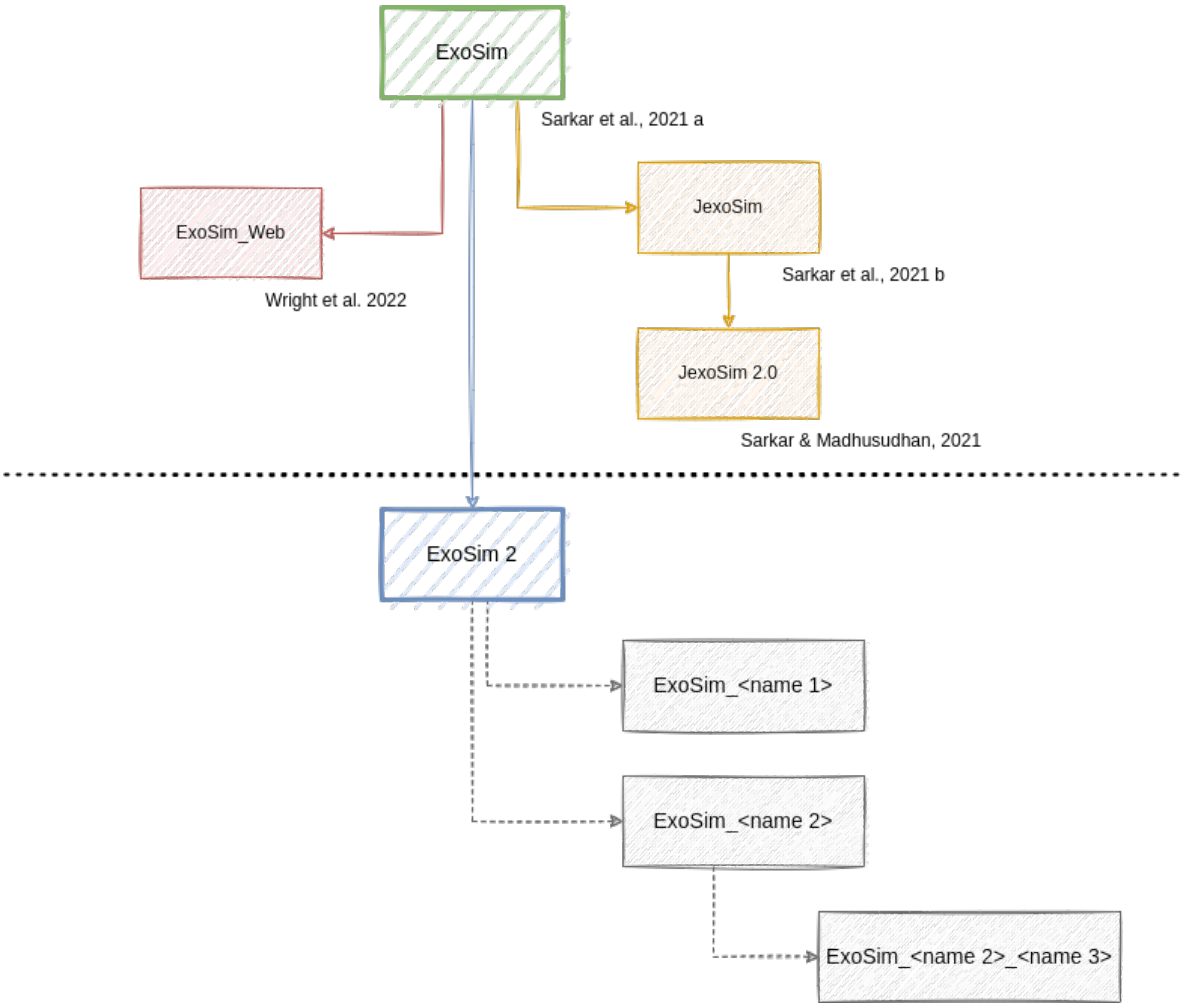
If the contributor does not have writing rights to the repository, should use the Fork-and-Pull[57] model. The contributor should fork[58] the main repository and clone it. Then the new features can be implemented. When the code is ready, a pull[59] request can be raised.

[54] https://github.com/arielmission-space/ExoSim2-public/branches
[55] https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request
[56] https://github.com/arielmission-space/ExoSim2-public/branches
[57] https://en.wikipedia.org/wiki/Fork_and_pull_model
[58] https://docs.github.com/en/get-started/quickstart/fork-a-repo
[59] https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request



Derived projects

If the contributor wants to maintain a custom forked repository or derived project, the following naming convention should be followed: the forked repository should be named after the original repository plus an identifying name. The following picture shows an example of a possible growth of the ExoSim family:



Automatic actions

Every time a commit is `pushed` into the *develop* or into the *main* branch, some automatic actions[60] are run by GitHub. The available action are stored into the *.github/workflows* directory in this repository. The basic actions are three:

- Linux OS (ci_linux.yml): this action run all the tests implemented in the repository for a Ubuntu virtual machine with Python 3.8 and 3.9.

If all the tests for each action are passed, a green badge will be added to the repository readme.

[60] https://github.com/features/actions

3.2 The Task structure

Instead of functions, *ExoSim* uses a tasks system. The *exosim.tasks.task.Task* is a class with logging properties (from *Logger*) which executes some operations.

3.2.1 Write a Task

To write a task, first we need to create a class that inherits from the *Task* class:

```
from exosim.tasks.task import Task

class ExampleTask(Task):
    """
    This is an example Task
    """
```

Then we need to define the inputs. This must be done in the class *__init__* using the *add_task_param* method:

```
def __init__(self):
    """
     Parameters
     __________
     parameters: dict
         dictionary containing the parameters. This is usually parsed from :class:`~
↪exosim.tasks.load.loadOptions.LoadOptions`
     wavelength: :class:`~astropy.units.Quantity`
         wavelength grid.
     output: :class:`~exosim.output.output.Output` (optional)
        output file
     """

    self.add_task_param('parameters', 'channel parameters dict')
    self.add_task_param('wavelength', 'wavelength grid')
    self.add_task_param('output', 'output file', None)
```

In this example we want the task to have 3 inputs: a dictionary, a wavelength grid and an output file. The latter is optional, in fact we have set `None` as a default value.

Then we can move to describe the operations that this Task is gonna do:

```
def execute(self):
    parameters = self.get_task_param('parameters')
    parameters = self.get_task_param('wavelength')
    parameters = self.get_task_param('output')

    ...

    variable = None
    self.set_output(variable)
```

The *get_task_param* method allow to retrieve the variable associated to the string used as argument. Then some operations are done and the output is set with the *set_output* method. If a list of variables are expected as output the code will be

```
variable1 = None
variable2 = None
self.set_output([variable1,variable2])
```

3.2.2 Logging

Logging is important when producing a new task, hence we include some logging options into the *Task* class. Here are some examples of how to use them, but you can have a better understanding by looking at the *Logger* class.

```
self.info("info message")
self.debug("debug message")
self.warning("warning message")
self.error("error message")
self.critical("critical message")
```

These lines can be include in every method inside the *Task* class.

3.2.3 Use a Task

To use a *Task* we first need to initialise it, and the call it with it's parameters:

```
exampleTask = ExampleTask()
variable = exampleTask(parameters = par_dic, wavelength=wl_grid)
```

3.3 Custom Tasks

ExoSim allow the user to replace some of the default *Task* with custom versions of the same process. Before writing a custom task, make you sure to read *The Task structure*.

To write a custom *Task* we need to create a class that first inherits from the default one, and then we can overwrite the *model* method.

Let's do an example. Suppose we want to write our own version of *LoadResponsivity*. This task estimate the detector responsivity and shall be indicated in the channel description, as described in *Estimate responsivity*.

The default task simply reads the right column from the file:

```
<channel> channel_name

    <qe>
        <responsivity_task>LoadResponsivity</responsivity_task>
        <datafile>__ConfigPath__/qe.ecsv</datafile>
    </qe>
```

Using the *model* method:

```
def model(self, parameters, wavelength, time):
    """
    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    parameters: dict
        dictionary contained the sources parameters.
    wavelength: :class:`~astropy.units.Quantity`
        wavelength grid.
    time: :class:`~astropy.units.Quantity`
        time grid.

    Returns
    --------
    :class:`~exosim.models.signal.Signal`
        channel responsivity

    """
    qe_data = parameters['qe']['data']
    wl_ = qe_data['Wavelength']
    qe_ = qe_data[parameters['value']]
    qe = signal.Dimensionless(data=qe_, spectral=wl_)
    qe.spectral_rebin(wavelength)
    qe.temporal_rebin(time)

    responsivity = signal.Signal(spectral=wavelength, time=time,
                                 data=qe.data * wavelength.to(
                                     u.m) / const.c / const.h * u.count)
    return responsivity
```

The input of the model is the *parameter* dictionary that contains the description of the full channel. So, let's imagine that instead of reading the data from a file, we want to estimate the quantum efficiency from a quadratic equation:

$$qe(\lambda) = A \cdot (\frac{\lambda}{\lambda_0})^2 + B \cdot \frac{\lambda}{\lambda_0} + C$$

where λ_0 is a reference wavelength. This equation for the quantum efficiency obviously has no physical justification. This example has been chosen specifically because it is not representative of any physical process, just to focus the attention on the code capabilities.

Then we need to include this model parameters in the channel description:

```
<channel> channel_name

    <qe>
        <A> 1 </A>
        <B> 2 </B>
        <C> 3 </C>
        <wl_0 unit=`micron`> 3.0 </wl_0>
    </qe>
```

and then we can write our own *Task* as

```
import exosim.tasks.load as load

class CustomResponsivity(load.LoadResponsivity):
    """
    Custom responsivity class
```

(continues on next page)

(continued from previous page)

```
    """

    def model(self, parameters, wavelength, time):
        """
        Parameters
        ----------
        parameters: dict
            dictionary contained the sources parameters.
        wavelength: :class:`~astropy.units.Quantity`
            wavelength grid.
        time: :class:`~astropy.units.Quantity`
            time grid.

        Returns
        --------
        :class:`~exosim.models.signal.Signal`
            channel responsivity

        """
        A = parameters['qe']['A']
        B = parameters['qe']['B']
        C = parameters['qe']['C']
        wl_0 = parameters['qe']['wl_0']
        qe_ = A * (wavelength/wl_0)**2 + B * (wavelength/wl_0) + c
        qe = signal.Dimensionless(data=qe_, spectral=wavelength)
        qe.temporal_rebin(time)

        responsivity = signal.Signal(spectral=wavelength, time=time,
                                     data=qe.data * wavelength.to(
                                         u.m) / const.c / const.h * u.count)
        return responsivity
```

It's important that the custom model returns an object of the same kind of the default one or an error will be raised.

Now we need to store this class in a dedicated file. Assume the file is *your/path/customResponsivity.py*, then you have to indicate it in the *.xml* description as

```
<channel> channel_name

    <qe>
        <responsivity_task> your/path/customResponsivity.py </responsivity_task>
        <A> 1 </A>
        <B> 2 </B>
        <C> 3 </C>
        <wl_0 unit=`micron`> 3.0 </wl_0>
    </qe>
```

Now *ExoSim* will run your task instead of the default one.

3.4 Signals

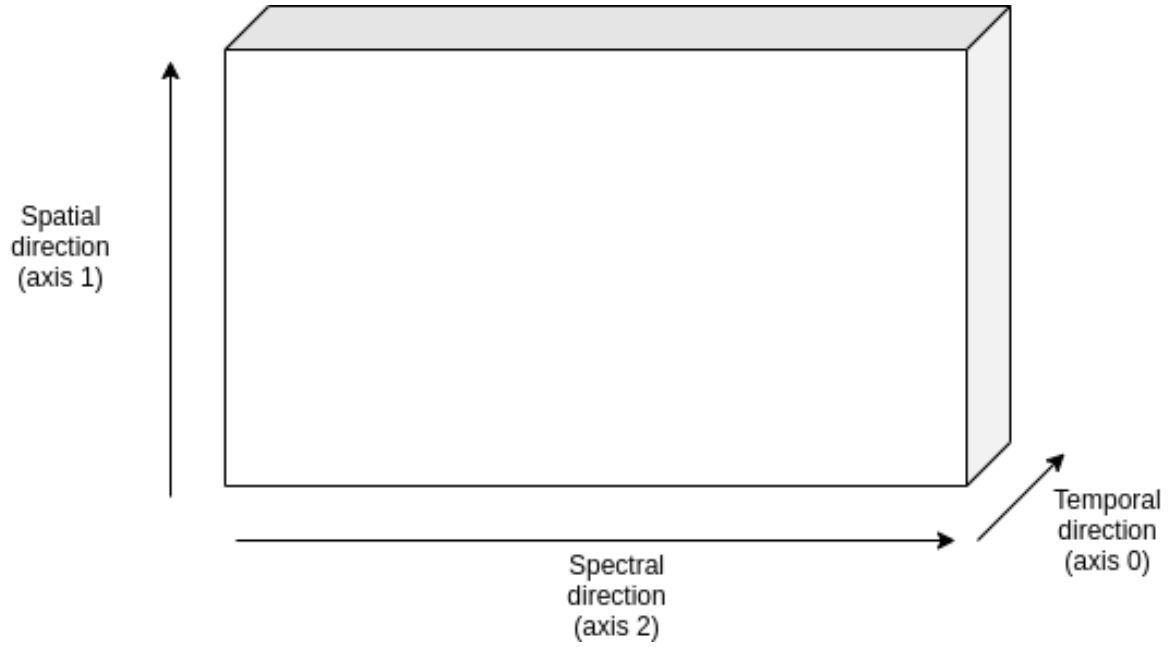
The data flow in the code is handled by the *exosim.models.signal.Signal* class. Signals are similar to array but with methods and a math specifically designed for the code goal.

To understand better how they work, let's produce a simple case:

```
import numpy as np
from exosim.models.signal import Signal

wl = np.linspace(0.1, 1, 10) * u.um
data = np.ones((10, 1, 10))
time_grid = np.linspace(1, 5, 10) * u.hr

signal = Signal(spectral=wl, data=data, time=time_grid)
```

The resulting `signal` variable now contains a *Signal* class. We can access the data stored in the `data` attribute. If units are attached to the data, are stored in the *data_units*.

The data are stored in a cube as described in the picture.



The grid used for the spectral direction (axis 2) is stored in the *spectral* attribute, with its units in *spectral_units*. Similarly, the grid used for the spatial direction (axis 1), if any, is stored in the *spatial* attribute, with its units in *spatial_units*, and the temporal grid (axis 0), if any, is stored in *time* with its units in *time_units*. If no information are provided for these grid the defaults values are $0\,\mu m$ for spectral and spatial axes and $0\,hr$ for temporal.

Also, *metadata* can be attach to a *Signal* class in the form a dictionary.

```
data = np.ones((10, 1, 10))
metadata = {'test': True}
signal = Signal(data=data, metadata=metadata)
```

or they can be attached later as

```
data = np.ones((10, 1, 10))
signal = Signal(data=data)
metadata = {'test': True}
signal.metadata = metadata
```

In both cases

```
>>> print(signal.metadata)
{'test': True}
```

3.4.1 Units

If any units is attached to the input data as in

```
data = np.ones(10)*u.m
signal = Signal(data=data)
```

Or they can be specified as:

```
data = np.ones(10)
signal = Signal(data=data, data_units=u.m)
```

Then, the data can be converted into a different units as

```
signal.to(u.cm)
```

3.4.2 Derived classes

Thanks to the units support, we can derive different derived classes:

- *exosim.models.signal.Sed*, which has units of $W\ m^{-2}\ \mu m^{-1}$
- *exosim.models.signal.Radiance*, which has units of $W\ m^{-2}\ \mu m^{-1}\ sr^{-1}$
- *exosim.models.signal.CountsPerSecond*, which has units of $counts\ s^{-1}$
- *exosim.models.signal.Counts*, which has units of $counts$
- *exosim.models.signal.Adu*, which has units of adu
- *exosim.models.signal.Dimensionless*, which has no units

The user can directly initialise one of these classes to specify the data units. Otherwise, if units are attached to the data, the main *Signal* class automatically detects the right derived class to use.

3.4.3 Mathematical operations

A set of mathematical operations is possible with the *Signal* class and it derived classes. Later here are listed the simplest examples to show the concept, however, the supported operation include:

- operations between *Signal* classes (as in the exaples);
- operations between a *Signal* and a `numpy.ndarray`[61] or a `Quantity`[62];

[61] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[62] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

- operations in reversed order (array + *Signal* instead of only *Signal* + array)

Also, the units are taken into account during the operation. In fact, multiplying a *Dimensionless* for a *Sed* results in a *Sed*, as multiply a *exosim.models.signal.Radiance* by a solid angle results in a *Sed*. It's not possible to sum or subtract a *Sed* class to a *Dimensionless*. This, again, is true not only between *Signal* classes, but also when operating *Signal* classes and Quantity[63].

Finally the operations involving *Signal* classes also work on cached classes. See *Cached signals* for more.

Sum

```
import numpy as np
import astropy.units as u
from exosim.models.signal import Signal

data = np.ones((3))
signal1 = Signal(data=data)

data = np.ones((3)) * 2
signal2 = Signal(data=data)

signal3 = signal1 + signal2
```

and hence

```
>>> print(signal3.data)
[[[3. 3. 3.]]]
```

Subtraction

```
import numpy as np
import astropy.units as u
from exosim.models.signal import Signal

data = np.ones((3))
signal1 = Signal(data=data)

data = np.ones((3)) * 2
signal2 = Signal(data=data)

signal3 = signal1 - signal2
```

and hence

```
>>> print(signal3.data)
[[[-1. -1. -1.]]]
```

[63] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Multiplication

```
import numpy as np
import astropy.units as u
from exosim.models.signal import Signal

data = np.ones((3))
signal1 = Signal(data=data)

data = np.ones((3)) * 2
signal2 = Signal(data=data)

signal3 = signal1 * signal2
```

and hence

```
>>> print(signal3.data)
[[[2. 2. 2.]]]
```

Division

```
import numpy as np
import astropy.units as u
from exosim.models.signal import Signal

data = np.ones((3))
signal1 = Signal(data=data)

data = np.ones((3)) * 2
signal2 = Signal(data=data)

signal3 = signal1 / signal2
```

and hence

```
>>> print(signal3.data)
[[[0.5 0.5 0.5]]]
```

Floor division

```
import numpy as np
import astropy.units as u
from exosim.models.signal import Signal

data = np.ones((3))
signal1 = Signal(data=data)

data = np.ones((3)) * 2
signal2 = Signal(data=data)

signal3 = signal1 // signal2
```

and hence

```
>>> print(signal3.data)
[[[0. 0. 0.]]]
```

3.4.4 Binning operation

Among the useful methods included in the *exosim.models.signal.Signal* class, it is worth mentioning the binning. There are two binning methods included in the class:

- *exosim.models.signal.Signal.spectral_rebin* to rebing the dataset in the spectral direction
- *exosim.models.signal.Signal.temporal_rebin* to rebing the dataset in the time direction

They are both based on *exosim.utils.binning.rebin*. The function resamples a function fp(xp) over the new grid x, rebinning if necessary, otherwise interpolates, but it does not perform extrapolation. The function is optimised to resample multidimensional array along a given axis.

Both *spectral_rebin* and *temporal_rebin* are described with examples in their documentation. Let's use as example here the case of a spectral binning. We first define the initial values:

```
>>> wavelength = np.linspace(0.1, 1, 10) * u.um
>>> data = np.ones((10, 1, 10))
>>> time_grid = np.linspace(1, 5, 10) * u.hr
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> print(signal.data.shape)
(10,1,10)
```

We can interpolates at a finer wavelength grid:

```
>>> new_wl = np.linspace(0.1, 1, 20) * u.um
>>> signal.spectral_rebin(new_wl)
>>> print(signal.data.shape)
(10,1,20)
```

or we can bin down the to a new wavelength grid:

```
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> new_wl = np.linspace(0.1, 1, 5) * u.um
>>> signal.spectral_rebin(new_wl)
>>> print(signal.data.shape)
(10,1,5)
```

3.4.5 Writing, copying and converting

Other useful methods are related to the capability to export the information content of a *exosim.models.signal.Signal* class.

A *exosim.models.signal.Signal* can be casted into a `dict`[64] object as

```
import numpy as np
from exosim.models.signal import Signal
```

(continues on next page)

[64] https://docs.python.org/dev/library/stdtypes.html#dict

(continued from previous page)

```
data = np.ones((3))
signal = Signal(data=data)

dict(signal)
```

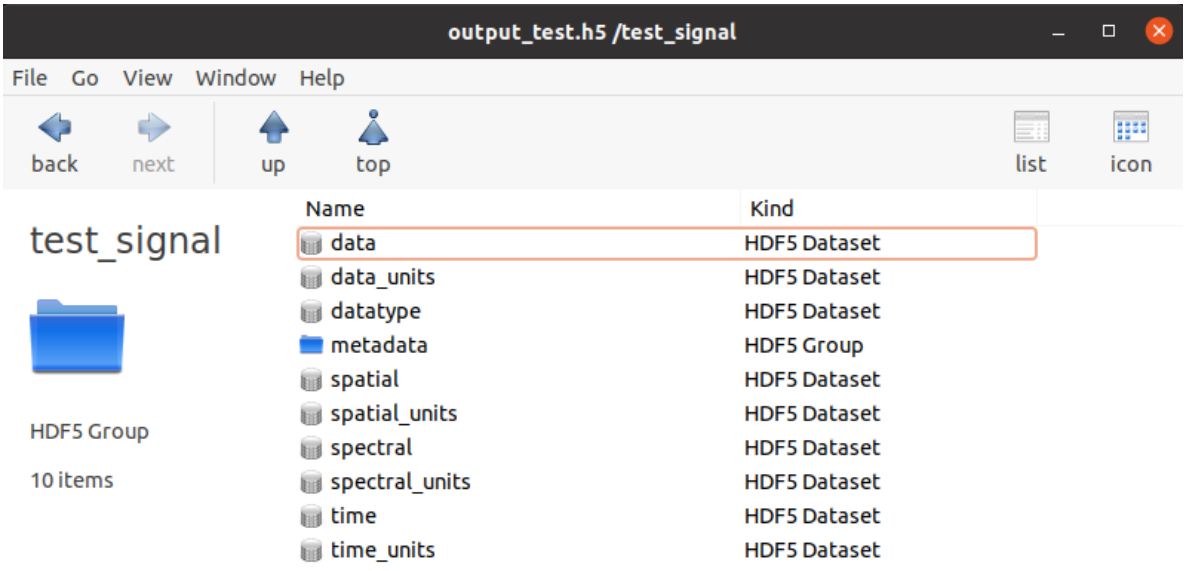
This will result in a dictionary with keys named after the class attributes with their content as value. The casting operation only conserves some of the *exosim.models.signal.Signal* class information. The attributes that are casted are: *data*, *time*, *spectral*, *spatial*, *metadata*, *data_units*, *time_units*, *spectral_units*, and *spatial_units*.

```
>>> print(dict(signal))
{'data': array([[[1., 1., 1.]]]),
 'time': array([0.]),
 'spectral': array([0.]),
 'spatial': array([0.]),
 'metadata': {},
 'data_units': '',
 'time_units': 'h',
 'spectral_units': 'um',
 'spatial_units': 'um'}
```

The *exosim.models.signal.Signal.write* method allow to store the content into and *Output* class. More commonly, into an HDF5 file. In the following example, we show how to store the signal class into an output file.

```
import os
from exosim.output.hdf5.hdf5 import HDF5Output

output = os.path.join("output_test.h5")
with HDF5Output(output) as o:
    signal.write(o, "test_signal")
```

Then, the output will contains the class information as:



The information stored are the same of `dict(signal)`

Also, an iterator has been implemented in the *Signal* class, such that the user can access the information as

```
>>> for k,v in signal1: print(k,v)
data [[[1. 1. 1.]]]
time [0.]
spectral [0.]
spatial [0.]
metadata {}
data_units
time_units h
spectral_units um
spatial_units um
```

Finally, a *Signal* class can be copied to a new class, thanks to the *exosim.models.signal.Signal.copy* method:

```
copied_signal = signal.copy()
```

3.4.6 Cached signals

Signal classes can be used in *cached* mode to handle huge datasets. This is enabled by chucked h5py.Dataset[65]. The *Signal* are supported by the The *CachedData* class. To produce a cached *Signal* the user must indicate a *HDF5OutputGroup* or *HDF5Output*, the data set shape, and a dataset name:

```
import numpy as np
import astropy.units as u

from exosim.models.signal import Signal
from exosim.output import SetOutput

output = SetOutput('test_file.h5')
with output.use(append=True, cache=True) as out:
    cached_signal = Signal(spectral = np.arange(0,100) * u.um,
                           data=None,
                           shape=(1000,100,100),
                           cached=True, output=out,
                           dataset_name='cached_dataset')
```

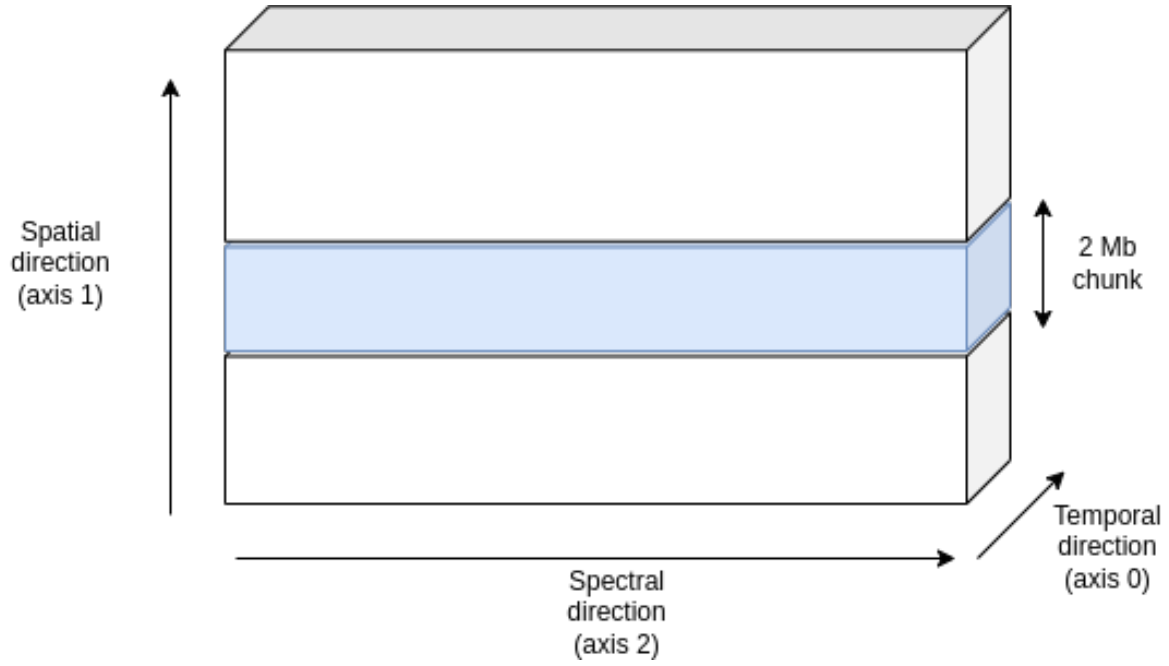
The dataset is stored in the indicated file un chunks of the user-defined size. By default the chunk size is set to 2 MB. Each chunk is a cube of the full spectral and spatial shapes and the number of time steps needed to weigh 2 MB.

Then, the chunk size can be set using the *RunConfig* class, as described in *Chunk size*:

```
from exosim.utils import RunConfig

RunConfig.chunk_size = N
```

where *N* is the desired size of chunk in MB, which will be set for the environment.

[65] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset



If no output file is indicated, the code produce a temporary file. Having a cached *Signal* is little different from a normal one. While for the normal one we usually access the datacube content using the `data` attribute, for a cached *Signal* is preferred to use the `dataset`: while the former forces the system to load all the datacube, which should be avoided for big dataset, the latter refers to the associate chunked `h5py.Dataset`[66] class.

To access the chunks and set the dataset values, one can use the `h5py.Dataset`[67] methods. In the following example, we iterate over the class chunks and set the values to 1.

```
for chunk in cached_signal.dataset.iter_chunks():
    dset = np.ones(cached_signal.dataset[chunk].shape)
    cached_signal.dataset[chunk] = dset
```

Otherwise, the data can be accessed as a normal numpy array

```
cached_signal.dataset[10,10,10] = 1
```

Note: A cached *Signal* allows the access to the associated `h5py.Dataset`[68] only as long as the *HDF5Output* is open.

To be sure to apply the edit to the dataset in the open file, remember to flush them:

```
cached_signal.output.flush()
```

Finally, if the user wants to loop over the chunks, a dedicate util is available in ExoSim: *iterate_over_chunks*

```
for chunk in iterate_over_chunks(cached_signal.dataset,
                                 desc="iterator description"):
    dset = np.ones(cached_signal.dataset[chunk].shape)
    cached_signal.dataset[chunk] = dset
    cached_signal.output.flush()
```

[66] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[67] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[68] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset

3.5 Run configuration

To handle shared information in the simulation, we can use the `RunConfig` class, which is a singleton initialized by *RunConfigInit*.

```
from exosim.utils import RunConfig
```

3.5.1 Parallel processing

Parallel processing is important for such demanding simulations. The number of parallel processes to use can be set using

```
from exosim.utils import RunConfig

RunConfig.n_job = N
```

This number is then set for both *joblib* and *numba* libraries.

3.5.2 Chunk size

The chunk size is the size of the chunk of the cached dataset (see *Cached signals*). This value can be set as

```
from exosim.utils import RunConfig

RunConfig.chunk_size = N
```

where *N* is the desired size of the chunk in MB, which will be set for the environment.

3.5.3 Random seed and Random generators

The initial random seed can be set as:

```
from exosim.utils import RunConfig

RunConfig.random_seed = N
```

where *N* is the desired seed number. By default, the seed is set to *None*, and therefore each simulation is unique.

ExoSim also provides a default random generator (`numpy.random.Generator`[69]) already initialized with the set random seed. The random generator can be accessed as:

```
from exosim.utils import RunConfig

rng = RunConfig.random_generator
```

and it can be used as any other random generator.

```
from exosim.utils import RunConfig

# uniform distribution:
```

(continues on next page)

[69] https://numpy.org/devdocs/reference/random/generator.html#numpy.random.Generator

(continued from previous page)

```
RunConfig.random_generator.uniform(-1,0,1000)

# normal distribution:
RunConfig.random_generator.normal(0,1,1000)

# Poisson distribution:
RunConfig.random_generator.poisson(5, 1000)
```

More examples are available in the numpy.random.Generator documentation[70].

Because ExoSim works with chunks of data and the generator may be used in loops, if the seed is not *None*, *random_generator* updates the seed at every call, by adding 1 to the given value.

[70] https://numpy.org/doc/stable/reference/random/generator.html

CHAPTER

FOUR

FAQS

4.1 What if I don't want to create a python virtual environment?

Sure you can avoid the use of virtual environment, if you want. In this case you have to follow all the step described *Installation & updates* except the environment sections.

Note that if you are installing ExoSim in your standard Python Environment, the system may ask you for administration privileges.

4.2 ExoSim is installed but not working: what can I do?

In our experience, mostly the errors raised after an failed run are of three different kinds:

1. Python rasie an `ImportError: No module named ####`, for it cannot import the module used.

   Generally you can fix it installing the missing dependency, as

   ```
   pip install ####
   ```

   if the installation return errors, you may need to use administration privileges. For a Unix system you can use:

   ```
   sudo pip install ####
   ```

2. Python raise an `ImportError: No module named ExoSim`.

   This means that ExoSim is not installed in the environment you are using. Try the installation procedure again or see the solution offered in *What if I don't want to create a python virtual environment?*

4.3 What versioning system is used?

We are using a versioning system compliant to PEP440[71] standards. Given a version number as X.Y.Z

- X is major modification identifier. Changing this number mean that we have refactored big part of the code. It's not compatible with previous versions.
- Y is minor modification. This number changes when we add functionalities or we change how part of the code work. A User will notice these variations and should check the documentation if some errors occur.

[71] https://www.python.org/dev/peps/pep-0440/

- Z is patches identification. We fixed some problem or optimized something. Probably the user won't notice the difference.

Other used versioning is in the form X.YbZ. The only difference is thar *b* stands for *beta*. You may also find *c*, meaning *release candidate*, or simply *r*, meaning *release* (this might be omitted) to indicate stable distributable versions.

4.4 How can I check which ExoSim version I'm using?

You can do this in multiple ways. The easiest way is to open the documentation (you already did) and look under the logo on the left panel or go to the *changelog*: there you will find all the versions listed. But this actually only refers to the directory you downloaded. To be sure that the version you are using is the same of the one you downloaded, run from the ExoSim Virtual Environment

```
pip show ExoSim
```

you will see all the installation details for ExoSim, including a *Version* line. Or you can check it from your python Virtual Environment

```
import ExoSim
ExoSim.__version__
```

or you can find it into the output files af an ExoSim module, looking for the `ExoSim version` metadata. Finally, you can even find that information inside the `ExoSim.log` file.

Be sure that version is the same reported in the documentation. If not, upgrade your installation with

```
pip install exosim --upgrade
```

if you installed *ExoSim* using pip (see install pip). If you used the source code from GitHub (see install git) go in your *ExoSim* directory, pull the last change and update your installation:

```
cd /your_path/ExoSim
git pull
pip install . --upgrade
```

and check again.

Tip: If you are using Anaconda Python, there must be a IDE listing all the installed package for your Virtual Environments and their versions.

4.5 How can I load HDF5 data into my code?

Once you have produced your dataset and it is stored into an *.h5* file, you can use the data using the python package h5py[72]. Assuming you data file is called *data_file.h5*, you can include it in your code as

```
import h5py

with h5py.File('data_file.h5', 'r+') as input_file:
    ...
```

[72] https://docs.h5py.org/en/stable/

Now the file can be navigated as a python dictionary. To read and use the data the user can refer to the documentation (https://docs.h5py.org/en/stable/high/dataset.html#reading-writing-data), but here is an example:

```
import h5py

with h5py.File('data_file.h5', 'r+') as input_file:
    data = input_file['first_level']['second_level']['dataset_name'][()]
```

This script navigates the file looking for the dataset called *dataset_name* that is under *first_level/second_level* and it loads all the dataset content into the *data* variable.

4.5.1 Load signals and tables

You can load the data stored into HDF5 file into their original python classes. In particular, you can cast a stored table into an `QTable`[73], using `astropy.io.misc.hdf5.read_table_hdf5`[74]:

```
import h5py
from astropy.io.misc.hdf5 import read_table_hdf5

with h5py.File('data_file.h5', 'r+') as input_file:
    table_data = input_file['first_level']['table_group']
    table = read_table_hdf5(table_data)
```

where *table_data* is a dictionary loaded from the hdf5 that contains both the table and the table metadata, stored in the fle as *.__table_column_meta__*.

In the case of *`Signal`* class, you can use the *`exosim.output.hdf5.utils.load_signal`*:

```
import h5py
from exosim.output.hdf5.utils import load_signal

with h5py.File('data_file.h5', 'r+') as input_file:
    signal_group = input_file['first_level']['stored_signal_name']
    signal = load_signal(signal_group)
```

[73] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[74] https://docs.astropy.org/en/latest/api/astropy.io.misc.hdf5.read_table_hdf5.html#astropy.io.misc.hdf5.read_table_hdf5

CHAPTER

FIVE

EXOSIM LICENSE

5.1 BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

CHAPTER

SIX

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog (keepachangelog[75]), and this project adheres to Semantic Versioning (semver[76]).

6.1 [v2.0.0-rc1[Page 169, 77]] - Scipy compatibility fix

6.1.1 Added

- .npy input support for pixel non-linearity coefficients (LoadPixelsNonLinearityMapNumpy)
- multiplicative noise simulator (AddGainNoise)

6.1.2 Fix

- replaced scipy.convolve with scipy.signal.convolve
- ADC now works with unsigned integers

6.2 [v2.0.0-rc0[78]] - Release Candidate

Cleaned repository

[75] https://keepachangelog.com/en/1.0.0/
[76] https://semver.org/spec/v2.0.0.html
[77] https://github.com/arielmission-space/ExoSim2-public/releases/tag/vv2.0.0-rc1
[78] https://github.com/arielmission-space/ExoSim2-public/releases/tag/v2.0.0-rc0

CHAPTER

SEVEN

API REFERENCE

This page contains auto-generated API reference documentation[762].

7.1 exosim

7.1.1 Subpackages

exosim.log

Submodules

exosim.log.logger

Module Contents

Classes

Logger	*Abstract class*

class Logger

Abstract class

Standard logging using logger library. It's an abtract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

set_log_name()

Produces the logger name and store it inside the class. The logger name is the name of the class that inherits this Logger class.

Return type
None

announce(*message*, **args*, ***kwargs*)

Produces ANNOUNCE level log See logging.Logger[79]

[762] Created with sphinx-autoapiPage 171, 763
[763] https://github.com/readthedocs/sphinx-autoapi

graphics(*message*, **args*, ***kwargs*)

Produces INFO level log See logging.Logger[80]

info(*message*, **args*, ***kwargs*)

Produces INFO level log See logging.Logger[81]

Parameters
message (*str*[82]) –

Return type
None

warning(*message*, **args*, ***kwargs*)

Produces WARNING level log See logging.Logger[83]

Return type
None

debug(*message*, **args*, ***kwargs*)

Produces DEBUG level log See logging.Logger[84]

Return type
None

trace(*message*, **args*, ***kwargs*)

Produces TRACE level log See logging.Logger[85]

Return type
None

error(*message*, **args*, ***kwargs*)

Produces ERROR level log See logging.Logger[86]

Return type
None

critical(*message*, **args*, ***kwargs*)

Produces CRITICAL level log See logging.Logger[87]

Return type
None

[79] https://docs.python.org/dev/library/logging.html#logging.Logger
[80] https://docs.python.org/dev/library/logging.html#logging.Logger
[81] https://docs.python.org/dev/library/logging.html#logging.Logger
[82] https://docs.python.org/dev/library/stdtypes.html#str
[83] https://docs.python.org/dev/library/logging.html#logging.Logger
[84] https://docs.python.org/dev/library/logging.html#logging.Logger
[85] https://docs.python.org/dev/library/logging.html#logging.Logger
[86] https://docs.python.org/dev/library/logging.html#logging.Logger
[87] https://docs.python.org/dev/library/logging.html#logging.Logger

Package Contents

Functions

trace(self, message, *args, **kws)	Log TRACE level.
announce(self, message, *args, **kws)	Log ANNOUNCE level.
graphics(self, message, *args, **kws)	Log GRAPHICS level.
setLogLevel(level[, log_id])	Simple function to set the logger level
disableLogging([log_id])	It disables the logging setting the log level to ERROR.
enableLogging([level, log_id])	It disables the logging setting the log level to ERROR.
addHandler(handler)	It adds a handler to the logging handlers list.
addLogFile([fname, reset, level])	It adds a log file to the handlers list.

Attributes

last_log

last_log

trace(*self*, *message*, **args*, ***kws*)

Log TRACE level. Trace level log should be produced anytime a function or a method is entered and exited.

Parameters
message (*str*[88]) –

Return type
None

announce(*self*, *message*, **args*, ***kws*)

Log ANNOUNCE level. This level log should be produced for huge announcements, as the starting of a long process.

graphics(*self*, *message*, **args*, ***kws*)

Log GRAPHICS level. This level log should be produced for graphical reasons only.

setLogLevel(*level*, *log_id=0*)

Simple function to set the logger level

Parameters

- **level** (*logging level*) –
- **log_id** (*int*[89]) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

Return type
None

[88] https://docs.python.org/dev/library/stdtypes.html#str
[89] https://docs.python.org/dev/library/functions.html#int

disableLogging(*log_id=0*)

It disables the logging setting the log level to ERROR.

Parameters
log_id (*int*[90]) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

Return type
None

enableLogging(*level=logging.INFO*, *log_id=0*)

It disables the logging setting the log level to ERROR.

Parameters

- **level** (*logging level*) – Default is logging.INFO.
- **log_id** (*int*[91]) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

Return type
None

addHandler(*handler*)

It adds a handler to the logging handlers list.

Parameters
handler (*logging handler*) –

Return type
None

addLogFile(*fname='{}.log'.format(__pkg_name__)*, *reset=False*, *level=logging.DEBUG*)

It adds a log file to the handlers list.

Parameters

- **fname** (*str*[92]) – name for the log file. Default is exosim.log.
- **reset** (*bool*[93]) – it reset the log file if it exists already. Default is False.
- **level** (*logging level*) – Default is logging.INFO.

Return type
None

exosim.models

Subpackages

exosim.models.utils

Submodules

exosim.models.utils.cachedData

[90] https://docs.python.org/dev/library/functions.html#int
[91] https://docs.python.org/dev/library/functions.html#int
[92] https://docs.python.org/dev/library/stdtypes.html#str
[93] https://docs.python.org/dev/library/functions.html#bool

Module Contents

Classes

CachedData	This class caches data cube into an h5 file. The cube data are chunked toward the first axis.

class CachedData(*axis0*, *axis1*, *axis2*, *output=None*, *output_path=None*, *dataset_name=None*, *dtype=np.float64*)

Bases: `exosim.log.Logger`

This class caches data cube into an h5 file. The cube data are chunked toward the first axis. In this class are also defined a set of operation to operate on the dataset using the chinks system.

Variables

- **axis0** (*int*[94]) – first axis size
- **axis1** (*int*[95]) – second axis size
- **axis2** (*int*[96]) – third axis size
- **output** (`h5py.File`[97] or *HDF5Output* or *HDF5OutputGroup*) – name of the file used for caching.
- **dataset_name** (*str*[98]) – name used to store the dataset into the h5 file.
- **output** – h5py open file used for caching
- **dataset_path** (*str*[99]) – path where is stored the dataset inside the output file.
- **chunked_dataset** (`h5py.Dataset`[100]) – h5py dataset used to store the data

Parameters

- **axis0** (*int*[101]) –
- **axis1** (*int*[102]) –
- **axis2** (*int*[103]) –
- **output** (*Union[str*[104]*, exosim.utils.types.HDF5OutputType]*) –
- **output_path** (*str*[105]) –
- **dataset_name** (*str*[106]) –
- **dtype** (*numpy.dtype*[107]) –

Notes

The cached data may be stored in a temporary file. To delete temporary files we included a garbage collector. Please, remember to delete the class when done as in the following example

```
>>> myClass = CachedData(1,1,1)
>>> del myClass
```

rename_dataset(*new_name*)

It renames the dataset in the HDF5 file.

Parameters
new_name (*str*[108]) – new name for the dataset

Return type
None

Submodules

exosim.models.channel

Module Contents

Classes

Channel	It handles the channel gven the description

class Channel(*parameters*, *wavelength*, *time*, *output=None*)

Bases: exosim.log.Logger

It handles the channel gven the description

Variables

- **ch_name** (*str*[109]) – channel name
- **path** (*dict*[110]) – dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the path.
- **responsivity** (*Signal*) – channel responsivity
- **sources** (*dict*[111]) – dictionary containing *Signal*
- **time** (Quantity[112]) – time grid.
- **parameters** (*dict*[113]) – dictionary contained the optical element parameters. This is usually parsed from *LoadOptions*
- **wavelength** (ndarray[114] or Quantity[115]) – wavelength grid. If no units are attached is considered as expressed in *um*.

[94] https://docs.python.org/dev/library/functions.html#int
[95] https://docs.python.org/dev/library/functions.html#int
[96] https://docs.python.org/dev/library/functions.html#int
[97] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[98] https://docs.python.org/dev/library/stdtypes.html#str
[99] https://docs.python.org/dev/library/stdtypes.html#str
[100] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[101] https://docs.python.org/dev/library/functions.html#int
[102] https://docs.python.org/dev/library/functions.html#int
[103] https://docs.python.org/dev/library/functions.html#int
[104] https://docs.python.org/dev/library/stdtypes.html#str
[105] https://docs.python.org/dev/library/stdtypes.html#str
[106] https://docs.python.org/dev/library/stdtypes.html#str
[107] https://numpy.org/devdocs/reference/generated/numpy.dtype.html#numpy.dtype
[108] https://docs.python.org/dev/library/stdtypes.html#str

- **focal_plane** (*Signal*) – source focal plane
- **frg_focal_plane** (*Signal*) – foreground focal plane
- **frg_sub_focal_planes** (*dict*[116]) – dictionary of *Signal*. It contains the sub focal planes produced by the radiances. This dictionary is produced only if at least one optical surface has `isolate=True`. The sum of the sub focal planes returns the frg_focal_plane. If not surface has `isolate=True`, the dictionary is empty.
- **output** (*Output*) – output file
- **target_source** (*str*[117]) – name of the target source

Parameters

- **parameters** (*dict*[118]) –
- **wavelength** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **output** (*exosim.utils.types.OutputType*) –

property target_source

parse_path(*light_path*)

It applies *ParsePath*

Parameters
light_path (*~collections.OrderedDict* (optional)) – dictionary of contributes

Returns
dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the path.

Return type
dict[119]

Note: The resulting information is also stored in the class under *path* attribute.

estimate_responsivity()

It estimates the responsivity using the indicated *LoadResponsivity*

Returns
channel responsivity

Return type
Signal

Note: The resulting information is also stored in the class under *responsivity* attribute.

propagate_foreground()

It multiplies each radiance in the path by the solid angle.

Returns
dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the path.

Return type
dict[120]

Note: it updates the *path* attribute of this class

propagate_sources(*sources*, *Atel*)

It propagates the sources though the channel, by applying *PropagateSources*

Parameters

- **sources** (*dict*[121]) – dictionary containing *Sed*
- **Atel** (Quantity[122]) – effective telescope Area

Returns
dictionary containing *Signal*

Return type
dict[123]

create_focal_planes()

It produces the empty focal planes

Returns
focal plane array (with time evolution)

Return type
Signal

rescale_contributions()

It updated the contributions (sources and path) by rebinning them to the wavelength solution grid and multipling them by the wl solution gradient

Return type
None

populate_focal_plane(*pointing=None*)

It populates the empty focal plane with monocromatic PSFs.

Parameters
pointing ((astropy.units.Quantity[124], astropy.units.Quantity[125]) (optional)) – telescope pointing direction, expressed ad a tuple of RA and DEC in degrees. Default is None

Returns
focal plane array populated

Return type
Signal

populate_bkg_focal_plane(*pointing=None*)

It populates the empty background focal plane with monocromatic PSFs for each of the background sources.

Parameters
pointing ((astropy.units.Quantity[126], astropy.units.Quantity[127]) (optional)) – telescope pointing direction, expressed ad a tuple of RA and DEC in degrees. Default is None

Returns
background focal plane array populated

Return type
Signal

apply_irf()

It applies the intra pixel response function (IRF) to the focal plane

Returns
focal plane array

Return type
Signal

populate_foreground_focal_plane()

It adds the foreground contribution to the foreground focal plane

Returns
focal plane array

Return type
Signal

exosim.models.signal

Module Contents

Classes

Signal	This class handles data cubes. The cube axes are time (0), spatial (1) and spectral (2) directions.
Sed	It's a Signal class with data having units of $[Wm^{-2}\mu m^{-1}]$
Radiance	It's a Signal class with data having units of $[Wm^{-2}\mu m^{-1}sr^{-1}]$
CountsPerSecond	It's a Signal class with data having units of $[ct/s]$
Counts	It's a Signal class with data having units of $[ct]$
Adu	It's a Signal class with data having units of $[adu/s]$
Dimensionless	It's a Signal class with data having no units

class Signal(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *data_units=None*, *spatial=[0] * u.um*, *shape=None*, *cached=False*, *output=None*, *output_path=None*, *dataset_name=None*, *metadata=None*, *dtype=np.float64*)

[109] https://docs.python.org/dev/library/stdtypes.html#str
[110] https://docs.python.org/dev/library/stdtypes.html#dict
[111] https://docs.python.org/dev/library/stdtypes.html#dict
[112] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[113] https://docs.python.org/dev/library/stdtypes.html#dict
[114] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[115] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[116] https://docs.python.org/dev/library/stdtypes.html#dict
[117] https://docs.python.org/dev/library/stdtypes.html#str
[118] https://docs.python.org/dev/library/stdtypes.html#dict
[119] https://docs.python.org/dev/library/stdtypes.html#dict
[120] https://docs.python.org/dev/library/stdtypes.html#dict
[121] https://docs.python.org/dev/library/stdtypes.html#dict
[122] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[123] https://docs.python.org/dev/library/stdtypes.html#dict
[124] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[125] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[126] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[127] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Bases: exosim.log.Logger

This class handles data cubes. The cube axes are time (0), spatial (1) and spectral (2) directions.

Variables

- **spectral** (ndarray[128]) – spectral direction grid.
- **data** (ndarray[129]) – data table. It has 3 axes: 0. time axis, 1. spatial axis, 2. spectral axis.
- **dataset** (h5py.Dataset[130]) – If cached mode is enabled, it contains the data table. It has 3 axes: 0. time axis, 1. spatial axis, 2. spectral axis.
- **time** (ndarray[131]) – time grid.
- **spatial** (ndarray[132]) – spatial direction grid.
- **spectral_units** (*str*[133]) – define the spectral direction units. Default is *um*
- **data_units** (*str*[134]) – define the data units.
- **time_units** (*str*[135]) – define the time direction units. Default is *hr*.
- **spatial_units** (*str*[136]) – define the spatial direction units. Default is *um*.
- **cached** (*bool*[137]) – it tells if cache mode is enabled.
- **output** (str or *HDF5Output*) – h5py open file used for caching
- **dataset_name** (h5py.Dataset[138]) – h5py dataset used to store the data
- **output_path** (*str*[139]) – path where is stored the dataset inside the output file.
- **metadata** (*dict*[140]) – signal metadata attached to the class

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **data_units** (*exosim.utils.types.UnitType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –
- **shape** (*tuple*[141] *[int*[142], *int*[143], *int*[144] *]*) –
- **cached** (*bool*[145]) –
- **output** (*str*[146]) –
- **output_path** (*str*[147]) –
- **dataset_name** (*str*[148]) –
- **metadata** (*dict*[149]) –
- **dtype** (*numpy.dtype*[150]) –

Notes

To understand caching mode, please look to *CachedData*

to(*units*)

It converts the Signal data into the desired units.

Parameters
units (Unit[151]) – desired unit

Return type
None

Examples

```
>>> import numpy as np
>>> import astropy.units as u
>>> from exosim.models.signal import Signal
>>> wl = np.linspace(0.1, 1, 10) * u.m
>>> time_grid = np.linspace(1, 5, 10) * u.s
>>> data = np.random.random_sample((10, 1, 10))*u.m**2
>>> signal = Signal(spectral=wl, data=data, time=time_grid)
>>> signal.to(u.cm**2)
```

spectral_rebin(*new_wavelength*, *fill_value=0.0*, ***kwargs*)

It bins the class data over the spectral direction and changes the wavelegth attributes. This method is based on *rebin*.

Parameters

- **new_wavelength** (ndarray[152] or Quantity[153]) – new wavelength grid. If no units are attached is considered expressed in 'um'
- **fill_value** (float or Quantity[154]) – filla value for empry bins. If no units are attached is considered expressed in 'um'.

Return type
None

Examples

```
>>> from exosim.models.signal import Signal
>>> import numpy as np
>>> import astropy.units as u
```

We first define the initial values:

```
>>> wavelength = np.linspace(0.1, 1, 10) * u.um
>>> data = np.ones((10, 1, 10))
>>> time_grid = np.linspace(1, 5, 10) * u.hr
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> print(signal.data.shape)
(10,1,10)
```

We now interpolates at a finer wavelength grid:

```
>>> new_wl = np.linspace(0.1, 1, 20) * u.um
>>> signal.spectral_rebin(new_wl)
>>> print(signal.data.shape)
(10,1,20)
```

We now bin down the to a new wavelength grid:

```
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> new_wl = np.linspace(0.1, 1, 5) * u.um
>>> signal.spectral_rebin(new_wl)
>>> print(signal.data.shape)
(10,1,5)
```

temporal_rebin(*new_time*, *fill_value='extrapolate'*, ***kwargs*)

It bins the class data over the temporal direction and changes the time attributes. This method is based on *rebin*.

Parameters

new_time (ndarray[155] or Quantity[156]) – new time grid. If no units are attached is considered expressed in 'hr'

Return type

None

Examples

```
>>> from exosim.models.signal import Signal
>>> import numpy as np
>>> import astropy.units as u
```

We first define the initial values:

```
>>> wavelength = np.linspace(0.1, 1, 10) * u.um
>>> data = np.ones((10, 1, 10))
>>> time_grid = np.linspace(1, 5, 10) * u.hr
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> print(signal.data.shape)
(10,1,10)
```

We now interpolates at a finer time grid:

```
>>> new_time = np.linspace(1, 5, 20) * u.hr
>>> signal.temporal_rebin(new_time)
>>> print(signal.data.shape)
(20,1,10)
```

We now bin down the to a new wavelength grid:

```
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
>>> new_time = np.linspace(1, 5, 5) * u.hr
>>> signal.temporal_rebin(new_time)
>>> print(signal.data.shape)
(5,1,10)
```

write(*output=None*, *name=None*)

It writes the Signal class into an *Output*. The signal class is stored as a dictionary.

Parameters

- **output** (*Output*) – container to use to write the class
- **name** (*str*[157]) – name to use to store

Return type
None

Examples

```
>>> from exosim.models.signal import Signal
>>> import numpy as np
>>> import astropy.units as u
```

We first define the initial values:

```
>>> wavelength = np.linspace(0.1, 1, 10) * u.um
>>> data = np.ones((10, 1, 10))
>>> time_grid = np.linspace(1, 5, 10) * u.hr
>>> signal = Signal(spectral=wavelength, data=data, time=time_grid)
```

Then we store it in a test output HDF5 file

```
>>> from exosim.output.hdf5.hdf5 import HDF5Output
>>> output = os.path.join(test_dir, 'output_test.h5')
>>> with HDF5Output(output) as o:
>>>     signal.write(o, 'test_signal')
```

Inside the file is now stored a dictionary like the one we can obtain by

```
>>> dict(signal)
```

get_slice(*start_time*, *end_time*)

It returnes the data relative to a time slice:

Parameters

- **start_time** (float or Quantity[158]) – if a float is given, it's assume to be expressed in hours.
- **end_time** (float or Quantity[159]) – if a float is given, it's assume to be expressed in hours.

Return type
ndarray[160]

set_slice(*start_time*, *end_time*, *data*)

It replaces the class data values in a specific a time slice:

Parameters

- **start_time** (float or Quantity[161]) – if a float is given, it's assume to be expressed in hours.

- **end_time** (float or Quantity[162]) – if a float is given, it's assume to be expressed in hours.
- **data** (ndarray[163]) – data to use as replacement for the existing ones.

Return type
None

copy(***kwargs*)

It returns a copy of the class.

Parameters
kwargs (*dict*[164]) – Dictionary of parameters to overwrite. The paramaters that can be included in the list are *cached*, *metadata*, *dataset_name*, 'output`, *output_path*.

Return type
Signal

Examples

```
>>> import numpy as np
>>> import astropy.units as u
>>> from exosim.models.signal import Signal
>>> wl = np.linspace(0.1, 1, 10) * u.um
>>> data = np.random.random_sample((10, 1, 10))
>>> time_grid = np.linspace(1, 5, 10) * u.hr
>>> signal = Signal(spectral=wl, data=data, time=time_grid)
>>> signal_new = signal.copy()
```

class Sed(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having units of $[Wm^{-2}\mu m^{-1}]$

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

class Radiance(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having units of $[Wm^{-2}\mu m^{-1}sr^{-1}]$

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

[128] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[129] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[130] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[131] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[132] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[133] https://docs.python.org/dev/library/stdtypes.html#str
[134] https://docs.python.org/dev/library/stdtypes.html#str
[135] https://docs.python.org/dev/library/stdtypes.html#str
[136] https://docs.python.org/dev/library/stdtypes.html#str
[137] https://docs.python.org/dev/library/functions.html#bool
[138] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[139] https://docs.python.org/dev/library/stdtypes.html#str
[140] https://docs.python.org/dev/library/stdtypes.html#dict
[141] https://docs.python.org/dev/library/stdtypes.html#tuple
[142] https://docs.python.org/dev/library/functions.html#int
[143] https://docs.python.org/dev/library/functions.html#int
[144] https://docs.python.org/dev/library/functions.html#int
[145] https://docs.python.org/dev/library/functions.html#bool
[146] https://docs.python.org/dev/library/stdtypes.html#str
[147] https://docs.python.org/dev/library/stdtypes.html#str
[148] https://docs.python.org/dev/library/stdtypes.html#str
[149] https://docs.python.org/dev/library/stdtypes.html#dict
[150] https://numpy.org/devdocs/reference/generated/numpy.dtype.html#numpy.dtype
[151] https://docs.astropy.org/en/latest/api/astropy.units.Unit.html#astropy.units.Unit
[152] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[153] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[154] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[155] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[156] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[157] https://docs.python.org/dev/library/stdtypes.html#str
[158] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[159] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[160] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[161] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[162] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[163] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[164] https://docs.python.org/dev/library/stdtypes.html#dict

class CountsPerSecond(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having units of $[ct/s]$

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

class Counts(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having units of $[ct]$

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

class Adu(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having units of $[adu/s]$

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

class Dimensionless(*spectral=[0] * u.um*, *data=None*, *time=[0] * u.hr*, *spatial=[0] * u.um*, **args*, ***kwargs*)

Bases: *Signal*

It's a Signal class with data having no units

Parameters

- **spectral** (*exosim.utils.types.ArrayType*) –
- **data** (*exosim.utils.types.ArrayType*) –
- **time** (*exosim.utils.types.ArrayType*) –
- **spatial** (*exosim.utils.types.ArrayType*) –

exosim.output

Subpackages

exosim.output.hdf5

Submodules

exosim.output.hdf5.hdf5

Module Contents

Classes

HDF5OutputGroup	*Abstract class*
HDF5Output	*Abstract class*

Attributes

META_KEY

META_KEY = '__table_column_meta__'

class HDF5OutputGroup(*entry*, *fd*, *cache=False*, *filename=None*)

Bases: *exosim.output.output.OutputGroup*

Abstract class

Standard logging using logger library. It's an abtract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

Parameters

- **entry** (*h5py.Group*[165]) –
- **fd** (*h5py.File*[166]) –
- **cache** (*bool*[167]) –
- **filename** (*str*[168]) –

write_array(*array_name*, *array*, *metadata=None*)

Method to store ndarray[169].

Parameters

- **array_name** (*str*[170]) – dataset name
- **array** (ndarray[171]) – data to store
- **metadata** (*dict*[172]) – metadata to attach

Return type
None

`write_table`(*table_name*, *table*, *metadata=None*, *replace=False*)

Method to store `Table`[173] or `QTable`[174].

Parameters

- **table_name** (*str*[175]) – dataset name
- **table** (`Table`[176] or `QTable`[177]) – data to store
- **metadata** (*dict*[178]) – metadata to attach
- **replace** (*bool*[179]) – if `True`, it replaces the table in the outpur file if already existing. Default is `False`

Return type
None

`write_scalar`(*scalar_name*, *scalar*, *metadata=None*)

Method to store scalars.

Parameters

- **scalar_name** (*str*[180]) – dataset name
- **scalar** (*int*[181] *or* *float*[182]) – data to store
- **metadata** (*dict*[183]) – metadata to attach

Return type
None

`write_string`(*string_name*, *string*, *metadata=None*)

Method to store strings.

Parameters

- **string_name** (*str*[184]) – dataset name
- **string** (*str*[185]) – data to store
- **metadata** (*dict*[186]) – metadata to attach

Return type
None

`create_group`(*group_name*)

it creates an *HDF5OutputGroup*.

Parameters
group_name (*str*[187]) – group name

Return type
OutputGroup

`delete_data`(*key*)

It deletes a specific key and its associated data from the data store.

Parameters
key (*str*[188]) – The key identifying the data to be deleted.

Raises
`KeyError`[189] – If the key is not found in the data store

flush()

Return type
None

write_string_array(*string_name*, *string_array*, *metadata=None*)

Method to store ndarray[190] of strings.

Parameters

- **string_name** (*str*[191]) – dataset name
- **string_array** (ndarray[192]) – data to store
- **metadata** (*dict*[193]) – metadata to attach

Return type
None

write_quantity(*quantity_name*, *quantity*, *metadata=None*)

Method to store Quantity[194].

Parameters

- **quantity_name** (*str*[195]) – dataset name
- **quantity** (Quantity[196]) – data to store
- **metadata** (*dict*[197]) – metadata to attach

Return type
None

[165] https://docs.h5py.org/en/latest/high/group.html#h5py.Group
[166] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[167] https://docs.python.org/dev/library/functions.html#bool
[168] https://docs.python.org/dev/library/stdtypes.html#str
[169] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[170] https://docs.python.org/dev/library/stdtypes.html#str
[171] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[172] https://docs.python.org/dev/library/stdtypes.html#dict
[173] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[174] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[175] https://docs.python.org/dev/library/stdtypes.html#str
[176] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[177] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[178] https://docs.python.org/dev/library/stdtypes.html#dict
[179] https://docs.python.org/dev/library/functions.html#bool
[180] https://docs.python.org/dev/library/stdtypes.html#str
[181] https://docs.python.org/dev/library/functions.html#int
[182] https://docs.python.org/dev/library/functions.html#float
[183] https://docs.python.org/dev/library/stdtypes.html#dict
[184] https://docs.python.org/dev/library/stdtypes.html#str
[185] https://docs.python.org/dev/library/stdtypes.html#str
[186] https://docs.python.org/dev/library/stdtypes.html#dict
[187] https://docs.python.org/dev/library/stdtypes.html#str
[188] https://docs.python.org/dev/library/stdtypes.html#str
[189] https://docs.python.org/dev/library/exceptions.html#KeyError
[190] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[191] https://docs.python.org/dev/library/stdtypes.html#str
[192] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[193] https://docs.python.org/dev/library/stdtypes.html#dict
[194] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[195] https://docs.python.org/dev/library/stdtypes.html#str
[196] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[197] https://docs.python.org/dev/library/stdtypes.html#dict

class HDF5Output(*filename*, *append=False*, *cache=False*)

Bases: *exosim.output.output.Output*

Abstract class

Standard logging using logger library. It's an abtract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

open()

It opens the output file

Return type
None

add_info(*attrs*, *name=None*)

Parameters

- **attrs** (*dict*[198]) –
- **name** (*str*[199]) –

Return type
None

flush()

Return type
None

create_group(*group_name*)

it creates an *HDF5OutputGroup*.

Parameters
group_name (*str*[200]) – group name

Return type
OutputGroup

store_dictionary(*dictionary*, *group_name=None*)

it stores a full dictionary inside the *HDF5Output*, forcing the flush.

Parameters

- **dictionary** (*dict*[201]) – data to store
- **group_name** (*str*[202] *(optional)*) – group name for the data

Return type
None

close()

It closes the output file

Return type
None

getsize()

It returns the output file size

Return type
astropy.units.Quantity[203]

delete_data(*key*)

exosim.output.hdf5.utils

Module Contents

Functions

load_signal(input)	It loads the appropriate *Signal* class from an opened hdf5 file.
recursively_read_dict_contents(input_dict)	Will recursive read a dictionary, initializing quantities and table from a dictionary read from an hdf5 file.
copy_file(in_object, out_object)	Recursively copy an HDF5 tree structure from one file to another.

load_signal(*input*)

It loads the appropriate *Signal* class from an opened hdf5 file.

Parameters
input (h5py.File[204]) – opened hdf5 file

Return type
Signal

Raises
IOError[205] – if the loaded Dataset is a Cached *Signal*.

recursively_read_dict_contents(*input_dict*)

Will recursive read a dictionary, initializing quantities and table from a dictionary read from an hdf5 file.

Parameters
input_dict (*dict*[206]) – dictionary read from hdf5

Returns
Dictionary we want to use

Return type
dict[207]

copy_file(*in_object*, *out_object*)

Recursively copy an HDF5 tree structure from one file to another.

This function traverses the hierarchy of the input HDF5 object (*in_object*) and replicates it in the output HDF5 object (*out_object*). It can copy both groups and datasets, and also replicates all attributes.

[198] https://docs.python.org/dev/library/stdtypes.html#dict
[199] https://docs.python.org/dev/library/stdtypes.html#str
[200] https://docs.python.org/dev/library/stdtypes.html#str
[201] https://docs.python.org/dev/library/stdtypes.html#dict
[202] https://docs.python.org/dev/library/stdtypes.html#str
[203] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[204] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[205] https://docs.python.org/dev/library/exceptions.html#IOError
[206] https://docs.python.org/dev/library/stdtypes.html#dict
[207] https://docs.python.org/dev/library/stdtypes.html#dict

Parameters

- **in_object** (*Union[h5py.Group*[208]*, h5py.Dataset*[209]*]*) – The input HDF5 object (either root, a subgroup, or a dataset).
- **out_object** (*Union[h5py.Group*[210]*, h5py.Dataset*[211]*]*) – The output HDF5 object (either root, a subgroup, or a dataset).

Raises

ValueError[212] – If an invalid object type is encountered.

Return type

None

Submodules

exosim.output.output

Module Contents

Classes

Output	*Abstract class*
OutputGroup	*Abstract class*

class Output

Bases: exosim.log.Logger

Abstract class

Standard logging using logger library. It's an abtract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

abstract open()

It opens the output file

abstract create_group(*group_name*)

it creates an *OutputGroup*.

Parameters

group_name (*str*[213]) – group name

Return type

OutputGroup

abstract close()

It closes the output file

[208] https://docs.h5py.org/en/latest/high/group.html#h5py.Group
[209] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[210] https://docs.h5py.org/en/latest/high/group.html#h5py.Group
[211] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset
[212] https://docs.python.org/dev/library/exceptions.html#ValueError

store_dictionary(*dictionary*, *group_name=None*)

it stores a full dictionary inside an *OutputGroup*.

Parameters

- **dictionary** (*dict*[214]) – data to store
- **group_name** (*str*[215]) – group name for the data

abstract getsize()

It returns the output file size

abstract delete_data(*key*)

class OutputGroup(*name*)

Bases: *Output*, abc.ABC[216]

Abstract class

Standard logging using logger library. It's an abtract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

abstract write_array(*array_name*, *array*, *metadata=None*)

Method to store ndarray[217].

Parameters

- **array_name** (*str*[218]) – dataset name
- **array** (ndarray[219]) – data to store
- **metadata** (*dict*[220]) – metadata to attach

write_list(*list_name*, *list_array*, *metadata=None*)

Method to store lists.

Parameters

- **list_name** (*str*[221]) – dataset name
- **list_array** (*list*[222]) – data to store
- **metadata** (*dict*[223]) – metadata to attach

abstract write_scalar(*scalar_name*, *scalar*, *metadata=None*)

Method to store scalars.

Parameters

- **scalar_name** (*str*[224]) – dataset name
- **scalar** (*int*[225] *or float*[226]) – data to store
- **metadata** (*dict*[227]) – metadata to attach

abstract write_string(*string_name*, *string*, *metadata=None*, *replace=False*)

Method to store strings.

Parameters

- **string_name** (*str*[228]) – dataset name
- **string** (*str*[229]) – data to store

[213] https://docs.python.org/dev/library/stdtypes.html#str
[214] https://docs.python.org/dev/library/stdtypes.html#dict
[215] https://docs.python.org/dev/library/stdtypes.html#str

- **metadata** (*dict*[230]) – metadata to attach

abstract write_string_array(*string_name*, *string_array*, *metadata=None*)

Method to store ndarray[231] of strings.

Parameters

- **string_name** (*str*[232]) – dataset name
- **string_array** (ndarray[233]) – data to store
- **metadata** (*dict*[234]) – metadata to attach

abstract write_table(*table_name*, *table*, *metadata=None*, *replace=False*)

Method to store Table[235] or QTable[236].

Parameters

- **table_name** (*str*[237]) – dataset name
- **table** (Table[238] or QTable[239]) – data to store
- **metadata** (*dict*[240]) – metadata to attach
- **replace** (*bool*[241]) – if True, it replaces the table in the outpur file if already existing. Default is False

abstract write_quantity(*quantity_name*, *quantity*, *metadata=None*)

Method to store Quantity[242].

Parameters

- **quantity_name** (*str*[243]) – dataset name
- **quantity** (Quantity[244]) – data to store
- **metadata** (*dict*[245]) – metadata to attach

abstract create_group(*group_name*)

it creates an *OutputGroup* as a subgroup

Parameters
group_name (*str*[246]) – group name

Return type
OutputGroup

abstract delete_data(*key*)

It deletes a specific key and its associated data from the data store.

Parameters
key (*str*[247]) – The key identifying the data to be deleted.

Raises
KeyError[248] – If the key is not found in the data store

exosim.output.setOutput

Module Contents

Classes

SetOutput	It sets the output for the code.

class SetOutput(*filename=None*, *replace=True*)

Bases: exosim.log.Logger

It sets the output for the code. This class created and initializes the output file. If a file name is provided, it loads the relative *Output* class is instatiated. Otherwise an *HDF5Output* is used by default for a temporary file.

Parameters

- **filename** (*str*[249]) –
- **replace** (*bool*[250]) –

use(*append=True*, *cache=False*)

It returns the *Output* with file opened and ready to write

Parameters

[216] https://docs.python.org/dev/library/abc.html#abc.ABC
[217] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[218] https://docs.python.org/dev/library/stdtypes.html#str
[219] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[220] https://docs.python.org/dev/library/stdtypes.html#dict
[221] https://docs.python.org/dev/library/stdtypes.html#str
[222] https://docs.python.org/dev/library/stdtypes.html#list
[223] https://docs.python.org/dev/library/stdtypes.html#dict
[224] https://docs.python.org/dev/library/stdtypes.html#str
[225] https://docs.python.org/dev/library/functions.html#int
[226] https://docs.python.org/dev/library/functions.html#float
[227] https://docs.python.org/dev/library/stdtypes.html#dict
[228] https://docs.python.org/dev/library/stdtypes.html#str
[229] https://docs.python.org/dev/library/stdtypes.html#str
[230] https://docs.python.org/dev/library/stdtypes.html#dict
[231] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[232] https://docs.python.org/dev/library/stdtypes.html#str
[233] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[234] https://docs.python.org/dev/library/stdtypes.html#dict
[235] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[236] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[237] https://docs.python.org/dev/library/stdtypes.html#str
[238] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[239] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[240] https://docs.python.org/dev/library/stdtypes.html#dict
[241] https://docs.python.org/dev/library/functions.html#bool
[242] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[243] https://docs.python.org/dev/library/stdtypes.html#str
[244] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[245] https://docs.python.org/dev/library/stdtypes.html#dict
[246] https://docs.python.org/dev/library/stdtypes.html#str
[247] https://docs.python.org/dev/library/stdtypes.html#str
[248] https://docs.python.org/dev/library/exceptions.html#KeyError

- **append** (*bool*[251] *(optional)*) – True to append data to already existing file. Default is True.
- **cache** (*bool*[252] *(optional)*) – True to write data in caching mode. Default is False.

Returns
output class instantiated.

Return type
Output

open()

It returns the *Output* with file opened and ready to read

Returns
output class instantiated.

Return type
Output

delete()

It deletes the output file created.

Return type
None

exosim.output.utils

Module Contents

Functions

recursively_save_dict_contents_to_output(output, dic)	Will recursive write a dictionary into output.
store_thing(output, key, item)	It stores one thing into the *Output*

recursively_save_dict_contents_to_output(*output*, *dic*)

Will recursive write a dictionary into output.

:param *Output*: Group (or root) in output file to write to :param dic: Dictionary we want to write :type dic: dict[253]

store_thing(*output*, *key*, *item*)

It stores one thing into the *Output*

:param *Output*: Group (or root) in output file to write to :param key: name for the stored item :type key: str :param item: item to store :type item: obj

[249] https://docs.python.org/dev/library/stdtypes.html#str
[250] https://docs.python.org/dev/library/functions.html#bool
[251] https://docs.python.org/dev/library/functions.html#bool
[252] https://docs.python.org/dev/library/functions.html#bool
[253] https://docs.python.org/dev/library/stdtypes.html#dict

exosim.plots

Submodules

exosim.plots.focalPlanePlotter

Module Contents

Classes

FocalPlanePlotter	Focal plane plotter.

class FocalPlanePlotter(*input*)

Bases: exosim.log.Logger

Focal plane plotter. This class handles the methods to plot the focal palnes produced by *exosim*.

Variables

- **input** (*str*[254]) – input file name
- **fig** (matplotlib.figure.Figure[255]) – produced figure

Parameters

input (*str*[256]) –

Examples

The following example, given the *test_file.h5* produced by *exosim*, plots the focal plane at the first time stamp and stores the figure as *focal_plane.png*.

```
>>> from exosim.plots import FocalPlanePlotter
>>> focalPlanePlotter = FocalPlanePlotter(input='./test_file.h5')
>>> focalPlanePlotter.plot_focal_plane(time_step=0)
>>> focalPlanePlotter.save_fig('focal_plane.png')
```

load_focal_plane(*ch*, *time_step*)

It loads the channel focal plane from the input file:

Parameters

- **ch** (*str*[257]) – channel name
- **time_step** (*int*[258]) – time step to plot

Returns

- numpy.ndarray[259] – focal plane
- *int* – over sampling factor

Return type

Tuple[numpy.ndarray[260], int[261]]

plot_focal_plane(*time_step=0*, *scale='linear'*)

It plots the focal planes at a specific time. For each channel it adds a Axes[262] to a figure. It returns a Figure[263] with two rows: on the first row are reported the oversampled focal planes. In the second row are reported the extracted focal plane, where the oversampling is removed. The focal plane plotted is the combination of the source focal plane plus the foreground focal plane.

Parameters

- **time_step** (*int*[264]) – time step identifier
- **scale** (*str*[265] *(optional)*) – scale to use for the plot. If 'dB' the plot is in dB. Default is 'linear'.

Returns
populated figure

Return type
matplotlib.figure.Figure[266]

plot_bands(*ax*, *scale='log'*, *channel_edges=True*, *add_legend=True*)

It plots the channels bands behind the indicated axes.

Parameters

- **ax** (matplotlib.axes.Axes[267]) – axes where to plot the bands
- **scale** (*str*[268]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[269]) – if True the x axes ticks are placed at the channel edges. Default is True.
- **add_legend** (*bool*[270]) –

Returns
axes with channel bands added

Return type
matplotlib.axes.Axes[271]

plot_efficiency(*scale='log'*, *channel_edges=False*, *ch_lengend=False*, *efficiency='all'*)

It produces a figure with efficiencies for the input table.

Parameters

- **scale** (*str*[272]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[273]) – if True the x axes ticks are placed at the channel edges. Default is True.
- **ch_lengend** (*bool*[274]) – if True add a legend for the channels color. Default is True.
- **efficiency** (*str*[275]) – what efficiency to plot. Options are "optical efficiency", "responsivity", "quantum efficiency", "photon conversion efficiency" and "all". Default is "all".

Returns

- matplotlib.figure.Figure[276] – plotted figure
- (matplotlib.axes.Axes[277], matplotlib.axes.Axes[278], matplotlib.axes.Axes[279], matplotlib.axes.Axes[280]) – tuple of axis. First axes is for optical efficiency, second is for responsivity, third is for quantum efficiency and fourth is for photon conversion efficiency.

Return type
Tuple[matplotlib.pyplot.Figure, List[matplotlib.axes.Axes[281]]]

save_fig(*name*)

It saves the produced figure.

Parameters
name (*str*[282]) – figure name

Return type
None

exosim.plots.ndrsPlotter

Module Contents

Classes

NDRsPlotter	Sub-Exposures plotter.

class NDRsPlotter(*input*)

Bases: exosim.log.Logger

Sub-Exposures plotter. This class handles the methods to plot all the sub-exposures produced by *ExoSim*.

[254] https://docs.python.org/dev/library/stdtypes.html#str
[255] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[256] https://docs.python.org/dev/library/stdtypes.html#str
[257] https://docs.python.org/dev/library/stdtypes.html#str
[258] https://docs.python.org/dev/library/functions.html#int
[259] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[260] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[261] https://docs.python.org/dev/library/functions.html#int
[262] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[263] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[264] https://docs.python.org/dev/library/functions.html#int
[265] https://docs.python.org/dev/library/stdtypes.html#str
[266] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[267] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[268] https://docs.python.org/dev/library/stdtypes.html#str
[269] https://docs.python.org/dev/library/functions.html#bool
[270] https://docs.python.org/dev/library/functions.html#bool
[271] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[272] https://docs.python.org/dev/library/stdtypes.html#str
[273] https://docs.python.org/dev/library/functions.html#bool
[274] https://docs.python.org/dev/library/functions.html#bool
[275] https://docs.python.org/dev/library/stdtypes.html#str
[276] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[277] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[278] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[279] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[280] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[281] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[282] https://docs.python.org/dev/library/stdtypes.html#str

Examples

The following example, given the *test_file.h5* preoduced by Exosim, plots the sub-exposures stores the figures in the indicated folder.

```
>>> from exosim.plots import SubExposuresPlotter
>>> subExposuresPlotter = SubExposuresPlotter(input='./test_ndr.h5')
>>> subExposuresPlotter.plot('plots/')
```

Parameters
input (*str*[283]) –

plot(*out_dir*)

It iterates over the channels and plot the ndrs.

Parameters
out_dir (*str*[284]) – output directory

Return type
None

plot_NDRs(*ndrs*, *time_line*, *i*, *ch*, *out_dir*)

It plots the ndrs for a given channel.

Parameters

- **ndrs** (*np.ndarray*) – ndrs array
- **time_line** (*u.Quantity*) – temporal array
- **i** (*int*[285]) – index of ndr to plot
- **ch** (*str*[286]) – channel name
- **out_dir** (*str*[287]) – output directory name

Return type
None

load_ndrs(*ch*, *f*)

It loads the channel NDRs from the input file:

Parameters

- **ch** (*str*[288]) – channel name
- **file** (h5py.File[289]) – input file
- **f** (*h5py.File*[290]) –

Returns

- numpy.ndarray[291] – NDR
- astropy.units.Quantity[292] – time line
- *int* – number of groups per ramp

Return type
Tuple[numpy.ndarray[293], astropy.units.Quantity[294], int[295]]

exosim.plots.plotter

Module Contents

Functions

main()	

Attributes

logger	

logger

main()

Return type
None

exosim.plots.radiometricPlotter

Module Contents

Classes

RadiometricPlotter	Radiometric plotter.

class RadiometricPlotter(*input*)

Bases: exosim.log.Logger

Radiometric plotter. This class handles the methods to plot the radiometric table produced by *exosim*.

Variables

- **input** (str or astropy.table.QTable[296]) – input data

[283] https://docs.python.org/dev/library/stdtypes.html#str
[284] https://docs.python.org/dev/library/stdtypes.html#str
[285] https://docs.python.org/dev/library/functions.html#int
[286] https://docs.python.org/dev/library/stdtypes.html#str
[287] https://docs.python.org/dev/library/stdtypes.html#str
[288] https://docs.python.org/dev/library/stdtypes.html#str
[289] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[290] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[291] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[292] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[293] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[294] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[295] https://docs.python.org/dev/library/functions.html#int

- **input_table** (astropy.table.QTable[297]) – input radiometric table
- **fig** (matplotlib.figure.Figure[298]) – produced figure

Parameters
input (*Union[str[299], astropy.table.Table[300]]*) –

Examples

The following example, given the *test_file.h5* preoduced by Exosim, plots the radiometric table and stores the figure as *radiometric.png*.

```
>>> from exosim.plots import RadiometricPlotter
>>> radiometricPlotter = RadiometricPlotter(input='./test_file.h5')
>>> radiometricPlotter.plot_table()
>>> radiometricPlotter.save_fig('radiometric.png')
```

load_table(*input_file*)

It loads the radiometric table from the input file:

Parameters
input_file (*str*[301]) – input file name

Returns
loaded radiometric table

Return type
astropy.table.QTable[302]

plot_bands(*ax*, *scale='log'*, *channel_edges=True*, *add_legend=True*)

It plots the channels bands behind the indicated axes.

Parameters

- **ax** (matplotlib.axes.Axes[303]) – axes where to plot the bands
- **scale** (*str*[304]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[305]) – if `True` the x axes ticks are placed at the channel edges. Default is `True`.
- **add_legend** (*bool*[306]) –

Returns
axes with channel bands added

Return type
matplotlib.axes.Axes[307]

plot_noise(*ax*, *scale='log'*, *channel_edges=True*, *contribs=False*, *ch_lengend=True*)

It plots the noise components found in the input table in the indicated axes.

Parameters

- **ax** (matplotlib.axes.Axes[308]) – axes where to plot the noises
- **scale** (*str*[309]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[310]) – if `True` the x axes ticks are placed at the channel edges. Default is `True`.
- **contribs** (*bool*[311]) – if `True` all the contributions are plotted. Default is `False`.

- **ch_lengend** (*bool*[312]) – if `True` add a legend for the channels color. Default is `True`.

Returns
axes with noises plotted

Return type
`matplotlib.axes.Axes`[313]

plot_signal(*ax*, *ylim=None*, *scale='log'*, *channel_edges=True*, *contribs=False*, *ch_lengend=True*)

It plots the signal components found in the input table in the indicated axes.

Parameters

- **ylim** (*float*[314] *or* (*float*[315], *float*[316])) – ylim for `matplotlib.axes.Axes`[317].
- **ax** (`matplotlib.axes.Axes`[318]) – axes where to plot the signals
- **scale** (*str*[319]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[320]) – if `True` the x axes ticks are placed at the channel edges. Default is `True`.
- **contribs** (*bool*[321]) – if `True` all the contributions are plotted. Default is `False`.
- **ch_lengend** (*bool*[322]) – if `True` add a legend for the channels color. Default is `True`.

Returns
axes with signals plotted

Return type
`matplotlib.axes.Axes`[323]

plot_table(*scale='log'*, *channel_edges=True*, *contribs=False*)

It produces a figure with signal and noise for the input table.

Parameters

- **scale** (*str*[324]) – x axes scale. Default is *log*.
- **channel_edges** (*bool*[325]) – if `True` the x axes ticks are placed at the channel edges. Default is `True`.
- **contribs** (*bool*[326]) – if `True` all the contributions are plotted. Default is `False`.

Returns

- `matplotlib.figure.Figure`[327] – plotted figure
- (`matplotlib.axes.Axes`[328], `matplotlib.axes.Axes`[329]) – tuple of axis. First axes is for signal, second is for noise.

Return type
Tuple[matplotlib.pyplot.Figure, Tuple[matplotlib.pyplot.Axes, matplotlib.pyplot.Axes]]

plot_efficiency(*scale='log'*, *channel_edges=False*, *ch_lengend=True*)

It produces a figure with efficiencies for the input table.

Parameters

- **scale** (*str*[330]) – x axes scale. Default is *log*.

- **channel_edges** (*bool*[331]) – if `True` the x axes ticks are placed at the channel edges. Default is `True`.
- **ch_lengend** (*bool*[332]) – if `True` add a legend for the channels color. Default is `True`.

Returns

- `matplotlib.figure.Figure`[333] – plotted figure
- (`matplotlib.axes.Axes`[334], `matplotlib.axes.Axes`[335]) – tuple of axis. First axes is for signal, second is for noise.

Return type
Tuple[matplotlib.pyplot.Figure, Tuple[matplotlib.pyplot.Axes, matplotlib.pyplot.Axes]]

plot_apertures()

It produces a figure with apertures superimposed to the focal plane.

Returns
plotted figure

Return type
`matplotlib.figure.Figure`[336]

save_fig(*name*)

It saves the produced figure.

Parameters
name (*str*[337]) – figure name

Return type
None

exosim.plots.subExposuresPlotter

Module Contents

Classes

SubExposuresPlotter	Sub-Exposures plotter.

class SubExposuresPlotter(*input*)

Bases: exosim.log.Logger

Sub-Exposures plotter. This class handles the methods to plot all the sub-exposures produced by *ExoSim*.

[296] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[297] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[298] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[299] https://docs.python.org/dev/library/stdtypes.html#str
[300] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[301] https://docs.python.org/dev/library/stdtypes.html#str
[302] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[303] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[304] https://docs.python.org/dev/library/stdtypes.html#str
[305] https://docs.python.org/dev/library/functions.html#bool
[306] https://docs.python.org/dev/library/functions.html#bool
[307] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[308] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[309] https://docs.python.org/dev/library/stdtypes.html#str
[310] https://docs.python.org/dev/library/functions.html#bool
[311] https://docs.python.org/dev/library/functions.html#bool
[312] https://docs.python.org/dev/library/functions.html#bool
[313] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[314] https://docs.python.org/dev/library/functions.html#float
[315] https://docs.python.org/dev/library/functions.html#float
[316] https://docs.python.org/dev/library/functions.html#float
[317] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[318] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[319] https://docs.python.org/dev/library/stdtypes.html#str
[320] https://docs.python.org/dev/library/functions.html#bool
[321] https://docs.python.org/dev/library/functions.html#bool
[322] https://docs.python.org/dev/library/functions.html#bool
[323] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[324] https://docs.python.org/dev/library/stdtypes.html#str
[325] https://docs.python.org/dev/library/functions.html#bool
[326] https://docs.python.org/dev/library/functions.html#bool
[327] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[328] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[329] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[330] https://docs.python.org/dev/library/stdtypes.html#str
[331] https://docs.python.org/dev/library/functions.html#bool
[332] https://docs.python.org/dev/library/functions.html#bool
[333] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[334] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[335] https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[336] https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.Figure
[337] https://docs.python.org/dev/library/stdtypes.html#str

Examples

The following example, given the *test_file.h5* preoduced by Exosim, plots the sub-exposures stores the figures in the indicated folder.

```
>>> from exosim.plots import SubExposuresPlotter
>>> subExposuresPlotter = SubExposuresPlotter(input='./test_se.h5')
>>> subExposuresPlotter.plot('plots/')
```

plot(*out_dir*)

It iterates over the channels and plot the sub-exposures.

Parameters
out_dir (*str*[338]) – output directory

plot_SubExposure(*exposures*, *time_line*, *integration_times*, *i*, *ch*, *out_dir*)

It plots the sub-exposures for a given channel.

Parameters

- **ndrs** (*np.ndarray*) – ndrs array
- **time_line** (*u.Quantity*) – temporal array
- **integration_times** (*u.Quantity*) – array for the integration times
- **i** (*int*[339]) – index of ndr to plot
- **ch** (*str*[340]) – channel name
- **out_dir** (*str*[341]) – output directory name
- **exposures** (*numpy.ndarray*[342]) –

Return type
None

load_ndrs(*ch*, *f*)

It loads the channel sub-exposures from the input file:

Parameters

- **ch** (*str*[343]) – channel name
- **f** (h5py.File[344]) – input file

Returns

- *np.ndarray* – sub-exposures array
- *u.Quantity* – temporal array
- *u.Quantity* – array for the integration times
- *int* – number of exposures per ramp

Return type
Tuple[numpy.ndarray[345], astropy.units.Quantity[346], astropy.units.Quantity[347], int[348]]

exosim.recipes

Submodules

exosim.recipes.createFocalPlane

Module Contents

Classes

CreateFocalPlane	Pipeline to create the instrument focal planes.

class CreateFocalPlane(*options_file*, *output_file*, *store_config=False*)

Bases: *exosim.utils.timed_class.TimedClass*, exosim.log.Logger

Pipeline to create the instrument focal planes. This pipeline loads the configuration file and produces an output, if indicated, where all the products are stored. It loads the source SED and the foregrounds and, after the optical chain production, it estimates the focal plane for the source and for the foregrounds.

Variables

- **mainConfig** (*dict*[349]) – This is parsed from *LoadOptions*
- **output** (*Output* (optional)) – output file
- **payloadConfig** (*dict*[350]) – payload configuration dictionary extracted from mainConfig`
- **time** (Quantity[351]) – time grid.
- **wl_grid** (ndarray[352] or Quantity[353]) – wavelength grid.

Parameters

- **options_file** (*Union[str*[354]*, dict*[355]*]*) –
- **output_file** (*str*[356]) –
- **store_config** (*bool*[357]) –

[338] https://docs.python.org/dev/library/stdtypes.html#str
[339] https://docs.python.org/dev/library/functions.html#int
[340] https://docs.python.org/dev/library/stdtypes.html#str
[341] https://docs.python.org/dev/library/stdtypes.html#str
[342] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[343] https://docs.python.org/dev/library/stdtypes.html#str
[344] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[345] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[346] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[347] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[348] https://docs.python.org/dev/library/functions.html#int

Examples

```
>>> import exosim.recipes as recipes
>>> recipes.CreateFocalPlane(options_file= 'main _configuration.xml',
>>>                          output_file = 'output_file.h5')
```

prepare_environment(*out*)

Il prepores the input data to build the instrument focal planes

Parameters
out (*OutputGroup*) – output group

Returns

- *dict* – soureces dict
- *~collections.OrderedDict* – common path dictionary

Return type
Tuple[collections.OrderedDict[358], collections.OrderedDict[359]]

run_channel(*description*, *common_path*, *sources*, *pointing=None*, *out=None*)

It instantiates and runs the *Channel* for the indicated channel`

Parameters

- **description** (*dict*[360]) – channel description
- **common_path** (*~collections.OrderedDict*) – dictionary of contributes
- **sources** (*dict*[361]) – dictionary containing *Sed*
- **pointing** ((astropy.units.Quantity[362], astropy.units.Quantity[363]) (optional)) – telescope pointing direction, expressed ad a tuple of RA and DEC in degrees. Default is None
- **out** (OutputGroup (optional)) – output group

Return type
None

[349] https://docs.python.org/dev/library/stdtypes.html#dict
[350] https://docs.python.org/dev/library/stdtypes.html#dict
[351] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[352] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[353] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[354] https://docs.python.org/dev/library/stdtypes.html#str
[355] https://docs.python.org/dev/library/stdtypes.html#dict
[356] https://docs.python.org/dev/library/stdtypes.html#str
[357] https://docs.python.org/dev/library/functions.html#bool
[358] https://docs.python.org/dev/library/collections.html#collections.OrderedDict
[359] https://docs.python.org/dev/library/collections.html#collections.OrderedDict
[360] https://docs.python.org/dev/library/stdtypes.html#dict
[361] https://docs.python.org/dev/library/stdtypes.html#dict
[362] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[363] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

exosim.recipes.createNDRs

Module Contents

Classes

CreateNDRs	Pipeline to create the observation NDRs.

class CreateNDRs(*input_file*, *output_file*, *options_file*)

Bases: *exosim.utils.timed_class.TimedClass*, exosim.log.Logger

Pipeline to create the observation NDRs. This pipeline loads the configuration file and the Sub-Exposures.

Examples

```
>>> import exosim.recipes as recipes
>>> recipes.CreateNDRs(input_file='./input_file.h5',
>>>                    output_file = 'ndrs_file.h5',
>>>                    options_file='main _configuration.xml')
```

Parameters

- **input_file** (*str*[364]) –
- **output_file** (*str*[365]) –
- **options_file** (*Union[str*[366], *dict*[367]*]*) –

load_subexposure_data(*ch*)

It loads the sub-exposures data from file

Parameters

ch (*str*[368]) – channel name

Returns

- astropy.units.Quantity[369] – spectral axes array
- astropy.units.Quantity[370] – spatial axes array
- astropy.units.Quantity[371] – temporal array
- astropy.units.Quantity[372] – integration times
- *int* – number of exposures
- *int* – number of NDRs per ramp
- *int* – number of NDRs per group

Return type

Tuple[astropy.units.Quantity[373], astropy.units.Quantity[374], astropy.units.Quantity[375], astropy.units.Quantity[376], int[377], int[378], int[379]]

prepare_output(*spectral*, *spatial*, *time_line*, *integration_times*, *ch*, *ch_grp*)

It produces the NDRs output per channel.

Parameters

- **spectral** (*u.Quantity*) – spectral axes array
- **spatial** (*u.Quantity*) – spatial axes array
- **time_line** (*u.Quantity*) – temporal array
- **integration_times** (*u.Quantity*) – integration times
- **ch** (*str*[380]) – channel name
- **ch_grp** (*h5py.Group*[381]) – output group

Returns

- *exosim.models.signal.Counts* – NDRs signal class
- exosim.output.hdf5.h5df.HDF5OutputGroup – output group

Return type

Tuple[*exosim.models.signal.Counts*, exosim.output.HDF5OutputGroup]

clean_output_tree(*out*, *key_list*)

Remove specified keys and their associated data from the output object.

This method iterates through a list of keys and deletes each key-value pair from the output object. Useful for cleaning up temporary or unnecessary data from the output.

Parameters

- **out** (HDF5OutputGroup) – The output object where key-value pairs are stored.
- **key_list** (*list*[382] *of str*[383]) – A list of keys that need to be removed from the output object.

Return type

None

Examples

```
>>> out = Output({'key1': 'value1', 'key2': 'value2', 'key3': 'value3'})
>>> clean_output(out, ['key1', 'key3'])
>>> out.data
{'key2': 'value2'}
```

Notes

The method assumes that the keys specified in the *key_list* are present in the output object. If a key is not found, the *delete_data* method should handle it gracefully.

refactor_output(*output_file*)

Renames the original output file and creates a new one with its contents.

This method takes the original output file, renames it by appending '_unrefactored' to its name, and then copies its content into a new file with the original name. The refactored (old version) file is deleted after the operation.

Parameters
output_file (*str*[384]) – The path of the original output file.

Return type
None

exosim.recipes.createSubExposures

Module Contents

Classes

CreateSubExposures	Pipeline to create the focal planes sub-exposures.

class CreateSubExposures(*input_file*, *output_file*, *options_file*)

Bases: *exosim.utils.timed_class.TimedClass*, exosim.log.Logger

Pipeline to create the focal planes sub-exposures. This pipeline loads the configuration file and the produced focal planes to produce the sub-exposures. It prepares the pointing jitter first, then it scales it to the instrument focal planes pixels. It estimates the detector reading scheme, considering instantaneous readout, and it produces the sub-exposures by jittering the focal planes.

Variables

- **input** (*str*[385]) – input file
- **mainConfig** (*dict*[386]) – This is parsed from *LoadOptions*
- **payloadConfig** (*dict*[387]) – payload configuration dictionary extracted from mainConfig
- **jitter_spa** (Quantity[388]) – pointing jitter in the spatial direction expressed in units of deg.
- **jitter_spe** (Quantity[389]) – pointing jitter in the spectral direction expressed in units of deg.

Parameters

[364] https://docs.python.org/dev/library/stdtypes.html#str
[365] https://docs.python.org/dev/library/stdtypes.html#str
[366] https://docs.python.org/dev/library/stdtypes.html#str
[367] https://docs.python.org/dev/library/stdtypes.html#dict
[368] https://docs.python.org/dev/library/stdtypes.html#str
[369] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[370] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[371] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[372] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[373] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[374] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[375] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[376] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[377] https://docs.python.org/dev/library/functions.html#int
[378] https://docs.python.org/dev/library/functions.html#int
[379] https://docs.python.org/dev/library/functions.html#int
[380] https://docs.python.org/dev/library/stdtypes.html#str
[381] https://docs.h5py.org/en/latest/high/group.html#h5py.Group
[382] https://docs.python.org/dev/library/stdtypes.html#list
[383] https://docs.python.org/dev/library/stdtypes.html#str
[384] https://docs.python.org/dev/library/stdtypes.html#str

- **input_file** (*str*[390]) –
- **output_file** (*str*[391]) –
- **options_file** (*Union[str*[392], *dict*[393]*]*) –

Examples

```
>>> import exosim.recipes as recipes
>>> recipes.CreateSubExposures(input_file='./input_file.h5',
>>>                            output_file = 'se_file.h5',
>>>                            options_file='main _configuration.xml')
```

load_focal_plane(*ch*)

It loads the channel focal plane from the input file:

Parameters
ch (*str*[394]) – channel name

Returns

- *CountsPerSecond* – focal plane
- *CountsPerSecond* – foreground focal plane
- *CountsPerSecond* – bkg focal plane

Return type
Tuple[*exosim.models.signal.CountsPerSecond*, *exosim.models.signal.CountsPerSecond*]

load_source(*ch*, *source*)

It loads the channel focal plane from the input file:

Parameters

- **ch** (*str*[395]) – channel name
- **source** (*str*[396]) – source name

Returns
source

Return type
CountsPerSecond

copy_simulation_data(*ch*, *ch_grp*)

It copies relevant data from the input file into the output one. The copied data are: *info*, *qe_map*, *efficiency* and *responsivity*.

Parameters

- **ch** (*str*[397]) – channel name
- **ch_grp** (*HDF5OutputGroup*) – output file

Returns
output group

Return type
HDF5OutputGroup

exosim.recipes.radiometricModel

Module Contents

Classes

RadiometricModel	Pipeline to create the radiometric model.

class RadiometricModel(*options_file*, *input_file*)

Bases: *exosim.utils.timed_class.TimedClass*, exosim.log.Logger

Pipeline to create the radiometric model. This pipeline has three working modes:

- it can load an already produced focal plane and use it to estimate a radiometric model;
- it can produce a single sourec focal planet and estimate the radiometric model;
- it can load a target list and produce the radiometric model for each target of the target list.

Variables

- **mainConfig** (*dict*[398]) – This is parsed from *LoadOptions*
- **input** (*Output*) – input/output file
- **payloadConfig** (*dict*[399]) – payload configuration dictionary extracted from mainConfig`
- **table** (QTable[400]) – table for the radiometric estimations

Parameters

- **options_file** (*Union[str*[401]*, dict*[402]*]*) –
- **input_file** (*str*[403]) –

[385] https://docs.python.org/dev/library/stdtypes.html#str
[386] https://docs.python.org/dev/library/stdtypes.html#dict
[387] https://docs.python.org/dev/library/stdtypes.html#dict
[388] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[389] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[390] https://docs.python.org/dev/library/stdtypes.html#str
[391] https://docs.python.org/dev/library/stdtypes.html#str
[392] https://docs.python.org/dev/library/stdtypes.html#str
[393] https://docs.python.org/dev/library/stdtypes.html#dict
[394] https://docs.python.org/dev/library/stdtypes.html#str
[395] https://docs.python.org/dev/library/stdtypes.html#str
[396] https://docs.python.org/dev/library/stdtypes.html#str
[397] https://docs.python.org/dev/library/stdtypes.html#str

Examples

If the user wants to estimate the radiometric model af an existing focal plane

```
>>> import exosim.recipes as recipes
>>> rm = recipes.RadiometricModel(options_file= 'main _configuration.xml',
>>>                               input_file = 'focal_plane.h5')
```

Otherwhise, if a focal planet has not been produced yet, this recipy can produce it, if a destination not existing file is provided:

```
>>> import exosim.recipes as recipes
>>> rm = recipes.RadiometricModel(options_file= 'main _configuration.xml',
>>>                               input_file = 'desired_output.h5')
```

In both cases, to store the produced table into the output file, the *write* is to be used:

```
>>> rm.write()
```

single_file_pipeline()

Radiometric pipeline to run for a single target with an already produced focal plane. The involved steps are:

1. creation of the wavelength table with *create_table*;
2. estimation of the apertures sizes and number of pixels involved with *compute_apertures*;
3. estimation of the signals in the apertures for the sub foregrounds, if any: *compute_sub_foregrounds_signals*;
4. estimation of the total foreground signal in the apertures: *compute_foreground_signals*;
5. estimation of the source focal plane signal in the aperture: *compute_source_signals*;
6. estimation of the saturation time in the channel: *compute_saturation*;

The pipeline will update the *table* attribute.

Return type
None

common_pipeline()

Radiometric pipeline to run starting from a radiometric table with already estimated signals. It computes the noise.

1. estimation of the multiaccum factors *compute_multiaccum*;
2. estimation shot noise *compute_photon_noise*;
3. update total noise *update_total_noise*

The pipeline will update the *table* attribute.

Return type
None

create_table()

Produces the starting radiometric table with the spectral bins and their edges. It is based on *EstimateSpectralBinning* by default.

Returns
table for the radiometric estimations with wavelength grid.

Return type
QTable[404]

compute_apertures()

Estimates the photometric aperture for each spectral bin using *EstimateApertures* by default.

Returns
table with the apertures for each channel and bin

Return type
QTable[405]

write(*output_file=None*)

It adds the radiometric table to the output. If the table exists already in the output file, it replaces it.

Parameters
output_file (*str*[406] *(optional)*) – output file. Default is *input*

Return type
None

compute_sub_foregrounds_signals()

It estimates the radiometric signals on the foreground sub focal planes for all the channels and returns a table with all the contributions.

It uses *ComputeSubFrgSignalsChannel* by default.

Returns
signal table

Return type
astropy.table.QTable[407]

compute_foreground_signals()

It estimates the radiometric signals on the foreground focal plane for all the channels and returns a table with all the contributions.

It uses *ComputeSignalsChannel* by default.

Returns
signal table

Return type
astropy.table.QTable[408]

compute_source_signals()

It estimates the radiometric signals on the source focal plane for all the channels and returns a table with all the contributions.

It uses *ComputeSignalsChannel* by default.

Returns
signal table

Return type
astropy.table.QTable[409]

compute_saturation()

It computes and adds the saturation time to the radiometric table

Returns
saturation table

Return type
astropy.table.QTable[410]

`compute_multiaccum`()

It estimates the multiaccum gain factors using *Multiaccum*. The

Returns
multiaccum factors

Return type
astropy.table.QTable[411]

`compute_photon_noise`()

It computes and adds the photon noise to the radiometric table using *ComputePhotonNoise*.

Returns
photon noise

Return type
astropy.table.QTable[412]

`update_total_noise`()

Updates the total noise column in the radiometric table.

Return type
astropy.table.QTable[413]

exosim.tasks

Subpackages

exosim.tasks.astrosignal

Submodules

exosim.tasks.astrosignal.applyAstronomicalSignal

Module Contents

[398] https://docs.python.org/dev/library/stdtypes.html#dict
[399] https://docs.python.org/dev/library/stdtypes.html#dict
[400] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[401] https://docs.python.org/dev/library/stdtypes.html#str
[402] https://docs.python.org/dev/library/stdtypes.html#dict
[403] https://docs.python.org/dev/library/stdtypes.html#str
[404] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[405] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[406] https://docs.python.org/dev/library/stdtypes.html#str
[407] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[408] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[409] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[410] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[411] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[412] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[413] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

Classes

ApplyAstronomicalSignal	This task applies the astronomical signal to the sub-exposures.

Functions

populate(j0, psf, psf_index, shape_spectral, model)	creates the variation of the source signal on the focal plane

class ApplyAstronomicalSignal

Bases: *exosim.tasks.task.Task*

This task applies the astronomical signal to the sub-exposures. To do so, it first convolve the astronomical signal with the instrument line shape (ILS) on the focal plane. This is done by populating the focal plane with the ILS and then weighting the contributions to each pixel. The resulting model is then convolved with the intrapixel response function (IPRF) and downsampled to the sub-exposure time resolution. Then is finally multiplied to the sub-exposure signal.

Returns
sub-exposure cached signal class

Return type
Counts

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

select_chunk_range(*chunk*, *start_t*, *end_t*)

Selects and adjusts the range of the chunk to be processed based on the given start and end times.

Parameters

- **chunk** (*slice*[414]) – The original slice object representing the chunk to be processed.
- **start_t** (*int*[415]) – The start time to consider for processing.
- **end_t** (*int*[416]) – The end time to consider for processing.

Returns
The adjusted slice object representing the new chunk to be processed. Returns None if the chunk is entirely outside the start_t and end_t range.

Return type
slice[417]

Examples

```
>>> select_chunk_range(slice(5, 15, 1), 7, 13)
slice(7, 13, 1)
>>> select_chunk_range(slice(5, 15, 1), 16, 20)
None
```

populate(*j0*, *psf*, *psf_index*, *shape_spectral*, *model*)

creates the variation of the source signal on the focal plane

Parameters

- **j0** (*numpy.array*) –
- **psf** (*numpy.array*) –
- **psf_index** (*numpy.array*) –
- **shape_spectral** (*int*[418]) –
- **model** (*numpy.array*) –

Return type

tuple[419]([np.array, np.array])

exosim.tasks.astrosignal.estimateAstronomicalSignal

Module Contents

Classes

EstimateAstronomicalSignal	It is a base class for all astronomical signal estimation tasks.

class EstimateAstronomicalSignal

Bases: *exosim.tasks.task.Task*

It is a base class for all astronomical signal estimation tasks. If an output file is provided, the signal model is stored in the output file.

Returns

returns the planetary signal in a 2D array (wavelength, time)

Return type

ArrayType

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[414] https://docs.python.org/dev/library/functions.html#slice
[415] https://docs.python.org/dev/library/functions.html#int
[416] https://docs.python.org/dev/library/functions.html#int
[417] https://docs.python.org/dev/library/functions.html#slice
[418] https://docs.python.org/dev/library/functions.html#int
[419] https://docs.python.org/dev/library/stdtypes.html#tuple

abstract model(*timeline*, *wl_grid*, *ch_parameters={}*, *source_parameters={}*)

Astronomical signal model to implement.

Parameters

- **timeline** (astropy.units.Quantity[420]) – timeline to compute the signal
- **wl_grid** (astropy.units.Quantity[421]) – wavelength grid
- **ch_parameters** (*dict*[422], *optional*) – channel parameters, by default {}
- **source_parameters** (*dict*[423]) – source parameters, by default {}

Returns

returns the planetary signal in a 2D array (wavelength, time)

Return type

ArrayType

Raises

NotImplementedError[424] – if the model is not implemented

exosim.tasks.astrosignal.estimatePlanetarySignal

Module Contents

Classes

EstimatePlanetarySignal	This task estimates the planetary signal.

class EstimatePlanetarySignal

Bases: *exosim.tasks.astrosignal.estimateAstronomicalSignal.EstimateAstronomicalSignal*

This task estimates the planetary signal. The signal parameters are extracted from the sky configuration file. This Task is a wrapper for the batman package: it parses the planet parameters and computes the transit model. The transit model is computed at the input wavelength grid. By default, the transit type is set to 'primary' (i.e. the planet transits in front of the star). The planetary sarius can be parsed as a string (path to a file) or as a float (in units of stellar radii). If a file is indicated, the planetary radius is binned or interpolated to the input wavelength grid.

Based on Batman package by L. Kreidberg (https://ui.adsabs.harvard.edu/abs/2015PASP..127.1161K/abstract).

model(*timeline*, *wl_grid*, *ch_parameters={}*, *source_parameters={}*)

Parameters

- **timeline** (astropy.units.Quantity[425]) – timeline to compute the signal
- **wl_grid** (astropy.units.Quantity[426]) – wavelength grid
- **ch_parameters** (*dict*[427], *optional*) – channel parameters, by default {}

[420] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[421] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[422] https://docs.python.org/dev/library/stdtypes.html#dict
[423] https://docs.python.org/dev/library/stdtypes.html#dict
[424] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

- **source_parameters** (*dict*[428]) – source parameters, by default {}

Returns
returns the planetary signal in a 2D array (wavelength, time)

Return type
ArrayType

get_t14(*inc*, *sma*, *period*, *rp*)

t14 Calculates the transit time based on the work of Seager, S., & Mallen-Ornelas, G. 2003, ApJ, 585, 1038

Parameters

- **inc** (astropy.units.Quantity[429]) – Planet oprbital inclination
- **sma** (*float*[430]) – Semimajor axis in stellar units
- **period** (astropy.units.Quantity[431]) – Orbital period
- **rp** (*np.ndarray*) – Planet radius in stellar units

Returns
transit duration – Returns the transit duration

Return type
astropy.units.Quantity[432]

exosim.tasks.astrosignal.findAstronomicalSignals

Module Contents

Classes

FindAstronomicalSignals	This tasks find astronomical signals in the sky parameters dictionary.

class FindAstronomicalSignals

Bases: *exosim.tasks.task.Task*

This tasks find astronomical signals in the sky parameters dictionary. The signals are identified by the presence of the key "signal_task" in the dictionary.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

Return type
None

[425] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[426] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[427] https://docs.python.org/dev/library/stdtypes.html#dict
[428] https://docs.python.org/dev/library/stdtypes.html#dict
[429] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[430] https://docs.python.org/dev/library/functions.html#float
[431] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[432] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

find_signals(*input_dict*, *source*, *effect*)

Finds and stores the signal estimation task and parameters for a given source and effect.

Parameters

- **input_dict** (*dict*[433]) – The dictionary containing the configuration parameters, including the optional key 'signal_task' which specifies the task responsible for signal estimation.
- **source** (*str*[434]) – The name of the source for which the signals are to be found.
- **effect** (*str*[435]) – The name of the effect to be considered in the signal estimation.

Return type
None

exosim.tasks.detector

Submodules

exosim.tasks.detector.accumulateSubExposures

Module Contents

Classes

AccumulateSubExposures	It accumulates sub-exposures of the same ramp.

class AccumulateSubExposures

Bases: *exosim.tasks.task.Task*

It accumulates sub-exposures of the same ramp.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

Return type
None

static sub_exposures_cumsum(*dset*, *state_machine*, *offset*)

Parameters

- **dset** (*numpy.ndarray*[436]) –
- **state_machine** (*List[int*[437]*]*) –
- **offset** (*List[int*[438]*]*) –

Return type
numpy.ndarray[439]

[433] https://docs.python.org/dev/library/stdtypes.html#dict
[434] https://docs.python.org/dev/library/stdtypes.html#str
[435] https://docs.python.org/dev/library/stdtypes.html#str

exosim.tasks.detector.addConstantDarkCurrent

Module Contents

Classes

AddConstantDarkCurrent	It adds constant dark current to all the pixel in the array.

class AddConstantDarkCurrent

Bases: *exosim.tasks.task.Task*

It adds constant dark current to all the pixel in the array. The dark current is loaded from the parameters

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*, *integration_times*, *output=None*)

Parameters

- **subexposures** (*Counts*) – sub-exposures cached signal
- **parameters** (*dict*[440]) – channel parameters dictionary
- **integration_times** (Quantity[441]) – sub-exposures integration times
- **outputs** (*Output* (optional)) – output file

Return type
None

exosim.tasks.detector.addCosmicRays

Module Contents

Classes

AddCosmicRays	Task to simulate the addition of cosmic rays to sub-exposures in a detector.

[436] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[437] https://docs.python.org/dev/library/functions.html#int
[438] https://docs.python.org/dev/library/functions.html#int
[439] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[440] https://docs.python.org/dev/library/stdtypes.html#dict
[441] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

class AddCosmicRays

Bases: *exosim.tasks.task.Task*

Task to simulate the addition of cosmic rays to sub-exposures in a detector.

This task models the impact of cosmic rays on a detector during the exposure time. The model assumes that the cosmic rays can interact with the detector pixels in various predefined shapes like a cross, horizontal rectangle, and vertical rectangle. The number of cosmic ray events and their impact on the detector pixels are calculated based on the given cosmic ray flux, detector characteristics, and integration times for the sub-exposures.

Notes

- This task assumes that the cosmic ray events saturate the affected pixels, setting their value to the full well depth.
- The probabilities for the interaction shapes are configurable. The task issues a warning if the sum of provided probabilities is not 1.
- The cosmic ray flux is specified in ct/s/cm^2 and is scaled based on the pixel size and detector dimensions.

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*, *integration_times*, *output=None*)

Default model to simulate the addition of cosmic rays to the sub-exposures.

This method saturates the hit pixels in the sub-exposure data based on the cosmic ray rate, detector properties, and the given integration times. The cosmic ray interactions could be in various shapes like single pixel, lines, squares, etc., which are defined in the configuration.

Parameters

- **subexposures** (Signal) – The sub-exposures' cached signal.
- **parameters** (*dict*[442]) – The channel parameters dictionary containing detector information.
- **integration_times** (*ArrayType*) – The integration times for the sub-exposures.
- **output** (Output, *optional*) – The output file where the cosmic ray events will be stored.

Return type
None

Notes

The method assumes multiple possible shapes for the interaction: - Single pixel - Vertical line: Saturates two pixels vertically aligned. - Horizontal line: Saturates two pixels horizontally aligned. - Square: Saturates four pixels in a square. - Cross: Saturates five pixels in a cross shape. - Horizontal rectangle: Saturates six pixels in a horizontal rectangle shape. - Vertical rectangle: Saturates six pixels in a vertical rectangle shape.

count_events(*rate*, *pixel_size*, *spatial_pix*, *spectral_pix*, *integration_times*, *saturation_rate*)

Calculate the number of cosmic ray events in each sub-exposure.

Parameters

- **rate** (*float*[443]) – Cosmic rays flux rate in ct/s/cm^2.
- **pixel_size** (*float*[444]) – Size of a detector pixel in cm^2.
- **spatial_pix** (*int*[445]) – Number of spatial pixels in the detector.
- **spectral_pix** (*int*[446]) – Number of spectral pixels in the detector.
- **integration_times** (Quantity[447]) – Sub-exposure integration times.
- **saturation_rate** (*float*[448]) – Saturation rate for the detector.

Returns

events_counter_i – An array containing the number of events for each sub-exposure, rounded to the nearest integer.

Return type

np.ndarray

Notes

The method first scales the rate based on the pixel size and the number of pixels in both spatial and spectral dimensions. It then multiplies the scaled rate by the integration times and saturation rate to get the number of events. The fractional part of the events is handled probabilistically.

shapes_and_probs(*parameters*)

Generate cosmic ray interaction shapes and their corresponding probabilities.

Parameters

parameters (*dict*[449]) – A dictionary containing optional 'interaction_shapes' which is a sub-dictionary specifying shapes and their probabilities.

Returns

- **shapes** (*list of tuples*) – A list of tuples representing shapes of cosmic ray interactions.
- **probs** (*list of float*) – A list of probabilities corresponding to the shapes.

Raises

ValueError[450] – If the sum of provided probabilities for all shapes is greater than 1.

Return type

Tuple[List, List]

Examples

```
>>> shapes, probs = AddCosmicRays.shapes_and_probs({"interaction_shapes": {
↪"single": 0.5, "line_h": 0.3, "line_v": 0.2}})
>>> print(shapes)
[([0], [0]), ([0, 1], [0, 0]), ([0, 0], [0, 1]), ([0, 0, 1, 1], [0, 1, 0,␣
↪1]), ([0, 0, 0, -1, 1], [0, 1, -1, 0, 0])]
>>> print(probs)
[0.5, 0.3, 0.2, 0.1, 0.1]
```

```
>>> shapes, probs = AddCosmicRays.shapes_and_probs({"interaction_shapes": {
↪"single": 0.5, "line_h": 0.3, "line_v": 0.2}})
>>> print(shapes)
```

Warning: A warning is issued if the sum of provided probabilities for all shapes is not 1.

exosim.tasks.detector.addDarkCurrentMapNumpy

Module Contents

Classes

AddDarkCurrentMapNumpy	It adds a dark current map to the array.

class AddDarkCurrentMapNumpy

Bases: *exosim.tasks.task.Task*

It adds a dark current map to the array. The map must be indicated under the *dc_map_filename* keyword. The dark current is loaded from a NPY format file (see numpy documentation[451]).

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*, *integration_times*, *output=None*)

Parameters

- **subexposures** (*Counts*) – sub-exposures cached signal
- **parameters** (*dict*[452]) – channel parameters dictionary
- **integration_times** (Quantity[453]) – sub-exposures integration times

[442] https://docs.python.org/dev/library/stdtypes.html#dict
[443] https://docs.python.org/dev/library/functions.html#float
[444] https://docs.python.org/dev/library/functions.html#float
[445] https://docs.python.org/dev/library/functions.html#int
[446] https://docs.python.org/dev/library/functions.html#int
[447] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[448] https://docs.python.org/dev/library/functions.html#float
[449] https://docs.python.org/dev/library/stdtypes.html#dict
[450] https://docs.python.org/dev/library/exceptions.html#ValueError

- **outputs** (*Output* (optional)) – output file

Return type
None

exosim.tasks.detector.addGainDrift

Module Contents

Classes

AddGainDrift	It adds a gain noise map to the array.

class AddGainDrift

Bases: *exosim.tasks.task.Task*

It adds a gain noise map to the array.

The gain Drift is modeled as a polynomial model for a spectral and time dependent modulation.

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*, *output=None*)

Apply a gain drift model to the subexposures.

This method models the gain drift as a polynomial trend based on time and wavelength.

Parameters

- **subexposures** (Signal) – Sub-exposures cached signal.
- **parameters** (*dict*[454]) – A dictionary containing parameters for the noise model. Expected keys and sub-keys include: - 'readout': dict with 'readout_frequency' - 'detector': dict with keys 'gain_w0', 'gain_f0', 'gain_coeff_order_t', 'gain_coeff_t_min', 'gain_coeff_t_max', 'gain_coeff_order_w', 'gain_coeff_w_min', and 'gain_coeff_w_max'.
- **integration_times** (*np.ndarray*) – Sub-exposures integration times.
- **output** (*Optional[Output]*) – Output file to write information, defaults to None.

Returns
This method does not return a value but updates the subexposures dataset with applied gain noise.

[451] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html
[452] https://docs.python.org/dev/library/stdtypes.html#dict
[453] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Return type
None

Notes

The method computes Brownian noise based on the frequency parameters in 'parameters' and applies a polynomial trend to it. The trend's coefficients are randomly generated within specified ranges. The resulting gain noise is then rebinned and applied to the subexposures.

exosim.tasks.detector.addKTC

Module Contents

Classes

AddKTC	It adds the ktc bias to the sub-exposures.

class AddKTC

Bases: *exosim.tasks.task.Task*

It adds the ktc bias to the sub-exposures. This Task produces a new random offset map for each ramp, and it adds it to the sub-exposures of the same ramp. If an output group is provided, it saves all the random seeds used.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

static add_offset(*dset*, *state_machine*, *offset*, *bias_std*)

exosim.tasks.detector.addReadNoise

Module Contents

Classes

AddNormalReadNoise	This Task simulates the read noise as a normal distribution which parameters can be defined in the configuration file.

class AddNormalReadNoise

Bases: *exosim.tasks.task.Task*

This Task simulates the read noise as a normal distribution which parameters can be defined in the configuration file.

[454] https://docs.python.org/dev/library/stdtypes.html#dict

If it is not explicitly stated, the distribution mean is set to 0. A different realisation of the same distribution is added to each pixel of each sub-exposure. If an output group is provided, it saves all the random seeds used.

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.detector.addReadNoiseMapNumpy

Module Contents

Classes

AddReadNoiseMapNumpy	This Task simulates the read noise as a normal distribution which parameters can be defined with a map indicated in the configuration file under *read_noise_filename* keyword.

class AddReadNoiseMapNumpy

Bases: *exosim.tasks.task.Task*

This Task simulates the read noise as a normal distribution which parameters can be defined with a map indicated in the configuration file under *read_noise_filename* keyword. The input must be a NPY format file (see numpy documentation[455]) containing the map of the distribution standard deviation for each pixel.

A different realisations of the same distribution is added to each pixel of each sub-exposure. If an output group is provided, it saves all the random seeds used.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.detector.addShotNoise

Module Contents

Classes

AddShotNoise	It adds the shot noise to the sub-exposures.

[455] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

class AddShotNoise

Bases: *exosim.tasks.task.Task*

It adds the shot noise to the sub-exposures. The shot noise is added as a Poisson noise to the sub-exposures. If an output group is provided, it saves all the random seeds used.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.detector.analogToDigital

Module Contents

Classes

AnalogToDigital	It converts the $counts$ units of sub-exposures into adu units of NDRs.

class AnalogToDigital

Bases: *exosim.tasks.task.Task*

It converts the $counts$ units of sub-exposures into adu units of NDRs. The conversion is related to the user defined number of bits of the ADC. In the number of bits is not defined, 32 is the default value. This Task select the smallest `dtype` to represent the desired data type:

- if the input number of bits is smaller than 16, a `int16` data type is used,
- if the input number of bits is smaller than 8, a `int8` data type is used.

Otherwise, a `int32` data type is used.

The user should also specify the ADC gain factor to apply to the focal plane before the conversion.

Finally, the user should specify the rounding method to use to cast the float into integers. The *ADC_round_method* keyword indicates which method the ADC should use to cast the float into integers. Three options are available:

- *floor* which uses `numpy.floor`;
- *ceil* which uses `numpy.ceil`;
- *round* which uses `numpy.round`;

Default is *floor*.

Returns
sub-exposures converted into NDRs adu units

Return type
Adu

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.detector.applyDeadPixelMap

Module Contents

Classes

ApplyDeadPixelsMap	It masks the dead pixel in the array given their coordinates.

class ApplyDeadPixelsMap

Bases: *exosim.tasks.task.Task*

It masks the dead pixel in the array given their coordinates.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*, *output*)

static add_dead_pixels(*ndrs*, *dead_pixels_map*)

exosim.tasks.detector.applyDeadPixelMapNumpy

Module Contents

Classes

ApplyDeadPixelMapNumpy	It masks the dead pixel in the array given a numpy map.

class ApplyDeadPixelMapNumpy

Bases: exosim.tasks.detector.ApplyDeadPixelsMap

It masks the dead pixel in the array given a numpy map. The input must be a NPY format file (see numpy documentation[456]) containing a boolean map marking with True the dead pixels. The map should be indicated under dp_map_filename keyword.

model(*subexposures*, *parameters*, *output*)

static add_dead_pixels(*ndrs*, *dead_pixels_map*)

[456] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

exosim.tasks.detector.applyPixelsNonLinearity

Module Contents

Classes

ApplyPixelsNonLinearity	Given the pixel non-linearity parameters,

class ApplyPixelsNonLinearity

Bases: *exosim.tasks.task.Task*

Given the pixel non-linearity parameters, this Task correct the ideal measured signal rate to the pixel linearity.

$$Q_{det} = Q \cdot (a + b \cdot Q + c \cdot Q^2 + d \cdot Q^3 + e \cdot Q^4)$$

The input is a dictionary with a *map* keyword containg an array with the coefficients for each pixel. The map shape is (n_pixels_x, n_pixels_y, coefficient order).

The user can list any number of coefficients, that will be parsed in the following model

$$Q_{det} = Q \cdot (a_0 + \sum_i a_i \cdot Q^i)$$

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*)

Parameters

- **subexposures** (exosim.models.signal.Counts) –
- **parameters** (*dict*[457]) –

Return type
None

exosim.tasks.detector.applySimpleSaturation

Module Contents

Classes

ApplySimpleSaturation	This Task applies a simple model of saturation to pixel counts.

[457] https://docs.python.org/dev/library/stdtypes.html#dict

class ApplySimpleSaturation

Bases: *exosim.tasks.task.Task*

This Task applies a simple model of saturation to pixel counts. If the counts in a pixel are higher than the well depth, the counts are set to the well capacity.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*subexposures*, *parameters*)

Applies the saturation model to the subexposures.

Parameters

- **subexposures** (Counts) – The subexposures to be saturated.
- **parameters** (*dict*[458]) – Dictionary containing saturation parameters like 'detector' and 'well_depth'.

Return type
None

exosim.tasks.detector.loadPixelsNonLinearityMap

Module Contents

Classes

LoadPixelsNonLinearityMap	Loads the pixels non-linearuty map

class LoadPixelsNonLinearityMap

Bases: *exosim.tasks.task.Task*

Loads the pixels non-linearuty map

Returns
channel non linearity map

Return type
dict[459]

Raises
KeyError: – if the output do not have the *map* key

[458] https://docs.python.org/dev/library/stdtypes.html#dict

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*)

Parameters
parameters (*dict*[460]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*

Returns
channel pixel non-linearity map

Return type
dict[461]

exosim.tasks.detector.loadPixelsNonLinearityMapNumpy

Module Contents

Classes

LoadPixelsNonLinearityMapNumpy	Loads the pixels non-linearity map given a numpy map.

class LoadPixelsNonLinearityMapNumpy

Bases: exosim.tasks.detector.LoadPixelsNonLinearityMap

Loads the pixels non-linearity map given a numpy map. The input must be a NPY format file (see numpy documentation[462]) containing the coefficients for each pixel. The map should have dimensions (n, y, x) where n is the number of coefficients for the polynomial approximation and y and x are the number of pixels in spatial and spectral directions. The map should be indicated under pnl_filename keyword.

Returns
channel non linearity map

Return type
dict[463]

Raises
KeyError: – if the output do not have the *map* key

model(*parameters*)

Parameters
parameters (*dict*[464]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*

[459] https://docs.python.org/dev/library/stdtypes.html#dict
[460] https://docs.python.org/dev/library/stdtypes.html#dict
[461] https://docs.python.org/dev/library/stdtypes.html#dict

Returns
channel pixel non-linearity map

Return type
dict[465]

exosim.tasks.detector.mergeGroups

Module Contents

Classes

MergeGroups	It averages the NDRs of the same group.

class MergeGroups

Bases: *exosim.tasks.task.Task*

It averages the NDRs of the same group.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.foregrounds

Submodules

exosim.tasks.foregrounds.estimateZodi

Module Contents

Classes

EstimateZodi	It estimate the zodiacal radiance in the target direction for a specific wavelength range

class EstimateZodi

Bases: *exosim.tasks.task.Task*

It estimate the zodiacal radiance in the target direction for a specific wavelength range

Returns
zodiacal radiance

Return type
Radiance

[462] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html
[463] https://docs.python.org/dev/library/stdtypes.html#dict
[464] https://docs.python.org/dev/library/stdtypes.html#dict
[465] https://docs.python.org/dev/library/stdtypes.html#dict

Examples

```
>>> estimateZodi = EstimateZodi()
>>> wavelength = np.logspace(np.log10(0.45), np.log10(2.2), 6000) * u.um
>>> zodi = estimateZodi(wavelength=wavelength, zodiacal_factor=1)
```

or, given the pointing direction

```
>>> zodi = self.estimateZodi(wavelength=wavelength,  coordinates=(90 * u.deg, -
↪66 * u.deg))
```

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*a*, *wl*)

The used zodiacal model is based on the zodiacal light model presented in Glasse et al. 2010

$$I_{zodi}(\lambda) = a\left(3.5 \cdot 10^{-14} BB(\lambda, 5500\,K) + 3.52 \cdot 10^{-8} BB(\lambda, 270\,K)\right)$$

where $BB(\lambda, T)$ is the Planck black body law and a is the fitted coefficient.

Parameters

- **a** (*float*[466]) – zodiacal multiplicative factor. Default is 0.
- **wl** (ndarray[467] or Quantity[468]) – wavelength grid. If no units are attached is considered as expressed in *um*

Returns
zodiacal radiance

Return type
Radiance

zodiacal_fit_direction(*coordinates*, *zodi_map=None*)

In this case the A coefficient is selected by a precompiled grid. The grid has been estimated by fitting our model with Kelsall et al. (1998) data.

A custom map can be provided, to replace the default one, as long as it matches the format.

Parameters

- **coordinates** ((*float*[469], *float*[470])) – pointing coordinates as (ra, dec)
- **zodi_map** (*str*[471] *(optional)*) – map file containing the zodiacal coefficients per sky positions. A default map is inclluded in ExoSim, that contains the coefficients fitted over Kelsall et al. 1998 model.

Returns
zodiacal factor for Glasse et al. 2010 zodiacal model

Return type
float[472]

[466] https://docs.python.org/dev/library/functions.html#float
[467] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[468] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[469] https://docs.python.org/dev/library/functions.html#float
[470] https://docs.python.org/dev/library/functions.html#float
[471] https://docs.python.org/dev/library/stdtypes.html#str
[472] https://docs.python.org/dev/library/functions.html#float

exosim.tasks.instrument

Submodules

exosim.tasks.instrument.applyIntraPixelResponseFunction

Module Contents

Classes

ApplyIntraPixelResponseFunction	It applies the intra pixel response function to the plane

class ApplyIntraPixelResponseFunction

Bases: *exosim.tasks.task.Task*

It applies the intra pixel response function to the plane

Returns
focal plane array (irf applied)

Return type
Signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

select_convoltion_func(*method*, *irf*, *irf_delta*, *img_delta*)

exosim.tasks.instrument.computeSaturation

Module Contents

Classes

ComputeSaturation	It computes the saturation time given the focal plane and the detector parameters.

class ComputeSaturation

Bases: *exosim.tasks.task.Task*

It computes the saturation time given the focal plane and the detector parameters.

Returns

- Quantity[473] – saturation time
- Quantity[474] – frame time
- Quantity[475] – maximum signal
- Quantity[476] – minimum signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.computeSolidAngle

Module Contents

Classes

ComputeSolidAngle	It computes the solid angle given the system parameters.

class ComputeSolidAngle

Bases: *exosim.tasks.task.Task*

It computes the solid angle given the system parameters.

Returns
solid angle expressed as $sr \cdot m^2$

Return type
Quantity[477]

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.computeSourcesPointingOffset

Module Contents

Classes

ComputeSourcesPointingOffset	It computes the source offset on the focal plane respect to the pointing direction.

[473] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[474] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[475] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[476] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[477] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Functions

angle_of_view(plate_scale, delta_pix, ovs)	Computes the Angle of View for a single pixel

class ComputeSourcesPointingOffset

Bases: *exosim.tasks.task.Task*

It computes the source offset on the focal plane respect to the pointing direction. The offset is in units of subpixels.

Returns

- *int* – offset in the spatial direction
- *int* – offset in the spectral direction

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

angle_of_view(*plate_scale*, *delta_pix*, *ovs*)

Computes the Angle of View for a single pixel

Parameters

- **plate_scale** (astropy.units.Quantity[478]) – plate scale
- **delta_pix** (astropy.units.Quantity[479]) – size of a pixel

Returns

- astropy.units.Quantity[480] – angle of view in deg of each subpixel in the spatial direction
- astropy.units.Quantity[481] – angle of view in deg of each subpixel in the spectral direction

exosim.tasks.instrument.createFocalPlane

Module Contents

Classes

CreateFocalPlane	It produces the empty focal plane

class CreateFocalPlane

Bases: *exosim.tasks.task.Task*

It produces the empty focal plane

[478] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[479] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[480] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[481] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Returns
focal plane array (with time evolution)

Return type
Signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.createFocalPlaneArray

Module Contents

Classes

CreateFocalPlaneArray	It produces the focal plane array

class CreateFocalPlaneArray

Bases: *exosim.tasks.task.Task*

It produces the focal plane array

Returns
focal plane array (no time evolution)

Return type
Signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.createIntrapixelResponseFunction

Module Contents

Classes

CreateIntrapixelResponseFunction	It creates the intrapixel response function to be used in the convolution with the focal plane array.

class CreateIntrapixelResponseFunction

Bases: *exosim.tasks.task.Task*

It creates the intrapixel response function to be used in the convolution with the focal plane array.

Returns

- **kernel** (*2D array*) – the kernel image
- **kernel_delta** (*scalar*) – the kernel sampling interval in microns

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*)

This model creates the intrapixel response function to be used in the convolution with the focal plane array. Thsi intrapixel response function is compatible with the Scipy convolution functions.

Estimate the detector pixel response function with the prescription of Barron et al., PASP, 119, 466-475 (2007).

Parameters

- **oversampling** (*int*[482]) – number of samples in each resolving element. The final shape of the response function would be shape*osf
- **delta_pix** (astropy.units.Quantity[483]) – Phisical size of the detector pixel in microns
- **diffusion_length** (astropy.units.Quantity[484]) – diffusion length in microns
- **intra_pix_distance** (astropy.units.Quantity[485]) – distance between two adjacent detector pixels in microns

Returns

- **kernel** (*2D array*) – the kernel image
- **kernel_delta** (astropy.units.Quantity[486]) – the kernel sampling interval in microns

exosim.tasks.instrument.createOversampledIntrapixelResponseFunction

Module Contents

Classes

CreateOversampledIntrapixelResponseFuncti	It creates the intrapixel response function to be used in the convolution with the focal plane array.

class CreateOversampledIntrapixelResponseFunction

Bases: *exosim.tasks.instrument.createIntrapixelResponseFunction.CreateIntrapixelResponseFunction*

It creates the intrapixel response function to be used in the convolution with the focal plane array.

[482] https://docs.python.org/dev/library/functions.html#int
[483] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[484] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[485] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[486] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Returns

- **kernel** (*2D array*) – the kernel image
- **kernel_delta** (*scalar*) – the kernel sampling interval in microns

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

model(*parameters*)

This class produces an oversampled version of the intrapixel response function. This kernel is zero-padded to the size of the PSF. This version is compatible with *exosim.utils.convolution.fast_convolution* convolution function.

Estimate the detector pixel response function with the prescription of Barron et al., PASP, 119, 466-475 (2007).

Parameters

- **oversampling** (*int*[487]) – number of samples in each resolving element. The final shape of the response function would be shape*osf
- **delta_pix** (astropy.units.Quantity[488]) – Phisical size of the detector pixel in microns
- **diffusion_length** (astropy.units.Quantity[489]) – diffusion length in microns
- **intra_pix_distance** (astropy.units.Quantity[490]) – distance between two adjacent detector pixels in microns

Returns

- **kernel** (*2D array*) – the kernel image
- **kernel_delta** (astropy.units.Quantity[491]) – the kernel sampling interval in microns

exosim.tasks.instrument.foregroundsToFocalPlane

Module Contents

Classes

ForegroundsToFocalPlane	It adds the foreground contribution to the focal plane

class ForegroundsToFocalPlane

Bases: *exosim.tasks.task.Task*

It adds the foreground contribution to the focal plane

[487] https://docs.python.org/dev/library/functions.html#int
[488] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[489] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[490] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[491] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Returns
focal plane array

Return type
Signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.loadPsf

Module Contents

Classes

LoadPsf	It loads the PSFs from files.

class LoadPsf

Bases: *exosim.tasks.task.Task*

It loads the PSFs from files.

Returns

- ndarray[492] – cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.
- ndarray[493] – cube normalization factors. It contains the volume of each psf. axis=0 is time, axis=1 is wavelength.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

abstract model(*filename*, *parameters*, *wavelength*, *time*)

Parameters

- **filename** (*str*[494]) – PSF input file
- **parameters** (*dict*[495]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[496]) – wavelength grid.

Returns
cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.

Return type
ndarray[497]

[492] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[493] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[494] https://docs.python.org/dev/library/stdtypes.html#str
[495] https://docs.python.org/dev/library/stdtypes.html#dict
[496] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[497] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

exosim.tasks.instrument.loadPsfPaos

Module Contents

Classes

LoadPsfPaos	It loads the PSFs from a PAOS file.

class LoadPsfPaos

Bases: *exosim.tasks.instrument.loadPsf.LoadPsf*

It loads the PSFs from a PAOS file. Each PAOS file contains a PSF vs wavelength. The task loads the PSF cube provided by the *filename* parameter. The PSF are then interpolated over a grid matching the one used to produce the focal planes, to convert them into the physical units. Then the total volume of the interpolated PSF is rescaled to the total volume of the original one. This allow to take into account for loss in the transmission due to the optical path. The PSF are then interpolated over a wavelength grid matching the one used to for the focal plane, producing the cube. Then, the PSF cube is repeated on the temporal axis, because no temporal variation is considered in this Task.

Returns
cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.

Return type
ndarray[498]

model(*filename*, *parameters*, *wavelength*, *time*)

Parameters

- **filename** (*str*[499]) – PSF input file
- **parameters** (*dict*[500]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[501]) – wavelength grid.

Returns
cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.

Return type
ndarray[502]

static load_wl_sampled(*wl*, *data*)

extract the wavelength from the data stored in the surfaces

Parameters

- **wl** (*str*[503]) – wavelength
- **data** (h5py.File[504]) – opened HDF5 file

Returns
wavelength

Return type
float[505]

load_imac(*wl*, *data*, *parameters*, *oversampling*, *delta_pix*)

Returns the PSF interpolated to the detector pixel size. The PSF is normalized to the initial volume after the interpolation.

Parameters

- **wl** (*str*[506]) – wavelength
- **data** (h5py.File[507]) – opened HDF5 file
- **parameters** (*dict*[508]) – detector description
- **oversampling** (*int*[509]) – oversampling factor
- **delta_pix** (float or astropy.units.Quantity[510]) – sub-pixel size

Returns
PSF scaled to physical size

Return type
numpy.ndarray[511]

crop_image_stack(*psf_out*, *ene=0.99*, *time_iteration=False*)

It crops the image stack. It takes the PSF for the longest wavelength, as it is expected to be the largest PSF, and then selects the smallest aperture which collect at least the desired ene, using `find_rectangular_aperture`. Then it remove these area from all the image in the psf data cube.

Parameters

- **psf_out** (numpy.ndarray[512]) – output PSF cube
- **ene** (*float*[513]) – encircled energy desired. Default is 99%.
- **time_iteration** (*bool*[514]) – if time iteration is True then it perform

Returns
PSF cube cropped

Return type
numpy.ndarray[515]

static wl_interpolate(*psf_cube*, *wl*, *wl_sampled*)

This function interpolates the PSF to the desired wavelength grid

Parameters

- **psf_out** (numpy.ndarray[516]) – output PSF cube
- **psf_cube** (numpy.ndarray[517]) – input PSF cube
- **wl** (numpy.ndarray[518]) – desired wavelength grid
- **wl_sampled** (numpy.ndarray[519]) – input wavelength grid

Returns
output PSF cube

Return type
numpy.ndarray[520]

exosim.tasks.instrument.loadPsfPaosTimeInterp

Module Contents

Classes

LoadPsfPaosTimeInterp	It loads the PSFs from two PAOS files and interpolates over time between them according to the given parameters.

class LoadPsfPaosTimeInterp

Bases: *exosim.tasks.instrument.loadPsfPaos.LoadPsfPaos*

It loads the PSFs from two PAOS files and interpolates over time between them according to the given parameters. Each PAOS file contains a PSF vs wavelength. The task loads the PSF cubes provided by the *filename* parameter. The PSFs are then interpolated over a grid matching the one used to produce the focal planes, to convert them into the physical units. Then the total volume of the interpolated PSF is rescaled to the total volume of the original one. This allow to take into account for loss in the transmission due to the optical path. The PSF are then interpolated over a wavelength grid matching the one used to for the focal plane, producing the cube.

Then an example of time dependency is implemented in the model method.

The PSF cube is then cropped to removed unused edges, where the signal is less than 1/100 of the PSF peak. This would fasten up the successive *ExoSim* step.

Returns
cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.

Return type
ndarray[521]

Note: *filename* must be a string of filenames separated by a comma and a space. To write your version,

[498] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[499] https://docs.python.org/dev/library/stdtypes.html#str
[500] https://docs.python.org/dev/library/stdtypes.html#dict
[501] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[502] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[503] https://docs.python.org/dev/library/stdtypes.html#str
[504] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[505] https://docs.python.org/dev/library/functions.html#float
[506] https://docs.python.org/dev/library/stdtypes.html#str
[507] https://docs.h5py.org/en/latest/high/file.html#h5py.File
[508] https://docs.python.org/dev/library/stdtypes.html#dict
[509] https://docs.python.org/dev/library/functions.html#int
[510] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[511] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[512] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[513] https://docs.python.org/dev/library/functions.html#float
[514] https://docs.python.org/dev/library/functions.html#bool
[515] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[516] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[517] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[518] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[519] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[520] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

define a new class that inherits from this one in a dedicated python file. Then copy the *model* method and replace the indicated section (where *REPLACE THIS WITH YOUR MODE* is written).

---

model(*filename*, *parameters*, *wavelength*, *time*)

Parameters

- **filename** (*str*[522]) – PSF input file
- **parameters** (*dict*[523]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[524]) – wavelength grid.

Returns
cube of psfs. axis=0 is time, axis=1 is wavelength, axis=2 is spatial direction, axis=3 is spectral direction.

Return type
ndarray[525]

static normalise(*psf_out*, *psf_out_cropped*)

load_psfs(*filename*, *wavelength*, *parameters*)

Parameters

- **parameters** (*dict*[526]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[527]) – wavelength grid.
- **filename** (*str*[528]) – string containing PSF input files separated by a comma

Returns
list of psf cubes. axis=0 is wavelength, axis=1 is spatial direction, axis=2 is spectral direction.

Return type
list[529]

exosim.tasks.instrument.loadResponsivity

Module Contents

Classes

LoadResponsivity	Loads the channel responsivity expressed as counts/Joule

[521] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[522] https://docs.python.org/dev/library/stdtypes.html#str
[523] https://docs.python.org/dev/library/stdtypes.html#dict
[524] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[525] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[526] https://docs.python.org/dev/library/stdtypes.html#dict
[527] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[528] https://docs.python.org/dev/library/stdtypes.html#str
[529] https://docs.python.org/dev/library/stdtypes.html#list

class LoadResponsivity

Bases: *exosim.tasks.task.Task*

Loads the channel responsivity expressed as counts/Joule

Returns
channel responsivity

Return type
Signal

Raises

- **TypeError:** – if the output is not a *Signal* class
- **UnitConversionError** – if the output has not the right units (counts/Joule)

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*, *wavelength*, *time*)

Parameters

- **parameters** (*dict*[530]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[531]) – wavelength grid.
- **time** (Quantity[532]) – time grid.

Returns
channel responsivity

Return type
Signal

exosim.tasks.instrument.loadWavelengthSolution

Module Contents

Classes

LoadWavelengthSolution	It loads the wavelength solution expressed as a table with wavelength position on spectral and spatial directcions

[530] https://docs.python.org/dev/library/stdtypes.html#dict
[531] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[532] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

class LoadWavelengthSolution

Bases: *exosim.tasks.task.Task*

It loads the wavelength solution expressed as a table with wavelength position on spectral and spatial directcions

Returns
wavelength solution table

Return type
Qtable

Raises

- **TypeError:** – if the output is not a QTable[533] class
- **UnitsError** – if the output has not the right units (micron) in every column

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*)

It loads the Wavelength, X and Y columns from a file and it stores them into a class:*~astropy.table.Qtable* class

Parameters
parameters (*dict*[534]) – channel parameter dictionary. This is usually parsed from *LoadOptions*

Returns
wavelength solution table

Return type
Qtable

exosim.tasks.instrument.populateFocalPlane

Module Contents

Classes

PopulateFocalPlane	It populates the empty focal plane with monocromatic PSFs.

[533] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[534] https://docs.python.org/dev/library/stdtypes.html#dict

Functions

populate(i0, j0, psf, focal_plane, source)	it populates the focal plane adding the pfs

class PopulateFocalPlane

Bases: *exosim.tasks.task.Task*

It populates the empty focal plane with monocromatic PSFs.

Returns
focal plane array populated

Return type
Signal

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

static load_psf(*parameters*, *focal_plane*, *output*)

Loads the PSF as indicated in the configuration file or it creates them.

Parameters

- **parameters** (*dict*[535]) – channel parameter dictionary. This is usually parsed from *LoadOptions*
- **focal_plane** (*Signal*) – focal plane array (with time evolution)
- **output** (*Output* (optional)) – output file

Returns
four dimensional array: axis 0 is time, axis 1 is wavelength, axis 2 is spatial, axis 3 is spectral.

Return type
ndarray[536]

populate(*i0*, *j0*, *psf*, *focal_plane*, *source*)

it populates the focal plane adding the pfs

Parameters

- **i0** (*numpy.array*) –
- **j0** (*numpy.array*) –
- **psf** (*numpy.array*) –
- **focal_plane** (*numpy.array*) –
- **source** (*numpy.array*) –

Return type
numpy.array

[535] https://docs.python.org/dev/library/stdtypes.html#dict
[536] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

exosim.tasks.instrument.propagateForegrounds

Module Contents

Classes

PropagateForegrounds	it propagates the foreground though the channel.

class PropagateForegrounds

Bases: *exosim.tasks.task.Task*

it propagates the foreground though the channel.

Returns
dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the path.

Return type
~collections.OrderedDict

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.instrument.propagateSources

Module Contents

Classes

PropagateSources	it propagates the sources though the channel.

class PropagateSources

Bases: *exosim.tasks.task.Task*

it propagates the sources though the channel. It multiplies the stellar SED by the effective telescope area, the channel efficiency and the channel responsivity:

$$S_\nu = S^*_\nu \times A_{tel} \times \eta \times R_\nu$$

The result is in units of $ct/s/\mu m$

Returns
dictionary containing *Signal*

Return type
dict[537]

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[537] https://docs.python.org/dev/library/stdtypes.html#dict

exosim.tasks.load

Submodules

exosim.tasks.load.loadOpticalElement

Module Contents

Classes

LoadOpticalElement	Abstract class to load an optical element and returns the element self emission

class LoadOpticalElement

Bases: *exosim.tasks.task.Task*

Abstract class to load an optical element and returns the element self emission and it optical efficiency.

Returns

- *Radiance* – optical element radiance
- *Dimensionless* – optical element efficiency

Raises

TypeError: – if the outputs are not the right *Signal* subclasses

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*, *wavelength*, *time*)

It loads the indicated columns from the data file.

If emissivity and temperature are provided the radiance is estimated as

$$I_{surf}(\lambda) = \epsilon \cdot BB(\lambda, T)$$

where ϵ is the indicated emissivity and $BB(\lambda, T)$ is the Planck black body law.

Parameters

- **parameters** (*dict*[538]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[539]) – wavelength grid.
- **time** (Quantity[540]) – time grid.

Returns

- *Radiance* – optical element radiance

- *Dimensionless* – optical element efficiency

exosim.tasks.load.loadOptions

Module Contents

Classes

LoadOptions	Reads the xml file with payload parameters and return an object with attributes related to the input data

class LoadOptions

Bases: *exosim.tasks.task.Task*

Reads the xml file with payload parameters and return an object with attributes related to the input data

Variables
configPath (*str*[541]) – configuration path

Returns
parsed xml input file

Return type
dict[542]

Raises
IOError[543] – if the indicated file is not found or the format is not supported

Examples

```
>>> loadOptions = LoadOptions()
>>> options = loadOptions(filename = 'path/to/file.xml')
```

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[538] https://docs.python.org/dev/library/stdtypes.html#dict
[539] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[540] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[541] https://docs.python.org/dev/library/stdtypes.html#str
[542] https://docs.python.org/dev/library/stdtypes.html#dict
[543] https://docs.python.org/dev/library/exceptions.html#IOError

exosim.tasks.parse

Submodules

exosim.tasks.parse.parseOpticalElement

Module Contents

Classes

ParseOpticalElement	Given the element parameters, it parses the optical element and returns a dictionary.

class ParseOpticalElement

Bases: *exosim.tasks.task.Task*

Given the element parameters, it parses the optical element and returns a dictionary. It also applyes the time variation if provided.

Returns
dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the optical element.

Return type
dict[544]

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.parse.parsePath

Module Contents

Classes

ParsePath	Given the optical path descrition, it parses the optical elements and return an ordered dictionary.

class ParsePath

Bases: *exosim.tasks.task.Task*

Given the optical path descrition, it parses the optical elements and return an ordered dictionary.

Returns
dictionary of *Radiance* and *Dimensionless*, represeting the radiance and efficiency of the path.

[544] https://docs.python.org/dev/library/stdtypes.html#dict

Return type
dict[545]

Note: The user can force the parser to isolate contribution by addind to the description dictionary the key 'isolate' set to `True`.

property radiance_keys_list

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.parse.parseSource

Module Contents

Classes

ParseSources	Given the source descrition, it parses the sources elements and return a dictionary.
ParseSource	Given the source parameters, it parses the source element and returns a dictionary.

class ParseSources

Bases: *exosim.tasks.task.Task*

Given the source descrition, it parses the sources elements and return a dictionary. It also applyes the time variation if provided.

Returns
dictionary containing *Sed*

Return type
dict[546]

Examples

```
>>> import astropy.units as u
>>> import numpy as np
>>> from exosim.tasks.parse import ParseSources
>>> from collections import OrderedDict
>>>
>>> wl = np.linspace(0.5, 7.8, 10000) * u.um
>>> tt = np.linspace(0.5, 1, 10) * u.hr
>>>
>>> sources_in = OrderedDict({'HD 209458': {'value': 'HD 209458',
>>>                                         'source_type': 'planck',
>>>                                         'R': 1.18 * u.R_sun,
```

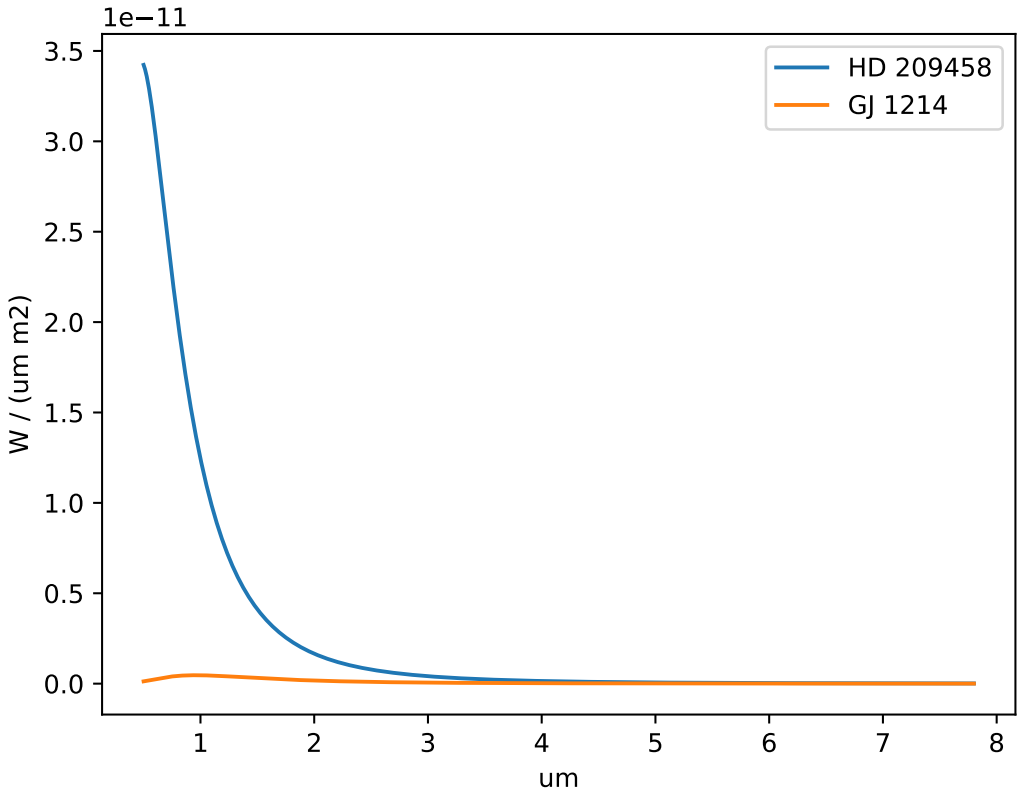
(continues on next page)

[545] https://docs.python.org/dev/library/stdtypes.html#dict

(continued from previous page)

```
>>>                                         'D': 47 * u.pc,
>>>                                         'T': 6086 * u.K,
>>>                                         },
>>>                             'GJ 1214': {'value': 'GJ 1214',
>>>                                         'source_type': 'planck',
>>>                                         'R': 0.218 * u.R_sun,
>>>                                         'D': 13 * u.pc,
>>>                                         'T': 3026 * u.K,
>>>                                         }, })
>>>
>>> parseSources = ParseSources()
>>> sources_out = parseSources(parameters=sources_in,
>>>                            wavelength=wl,
>>>                            time=tt)
```

```
>>> import matplotlib.pyplot as plt
>>>
>>> plt.plot(source_out['HD 209458'].spectral, source_out['HD 209458'].data[0,
↪0])
>>> plt.ylabel(source_out['HD 209458'].data_units)
>>> plt.xlabel(source_out['HD 209458'].spectral_units)
>>> plt.show()
```



execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

class ParseSource

Bases: *exosim.tasks.task.Task*

Given the source parameters, it parses the source element and returns a dictionary. It also applyes the time variation if provided.

Returns
dictionary containing *Sed*

Return type
dict[547]

Examples

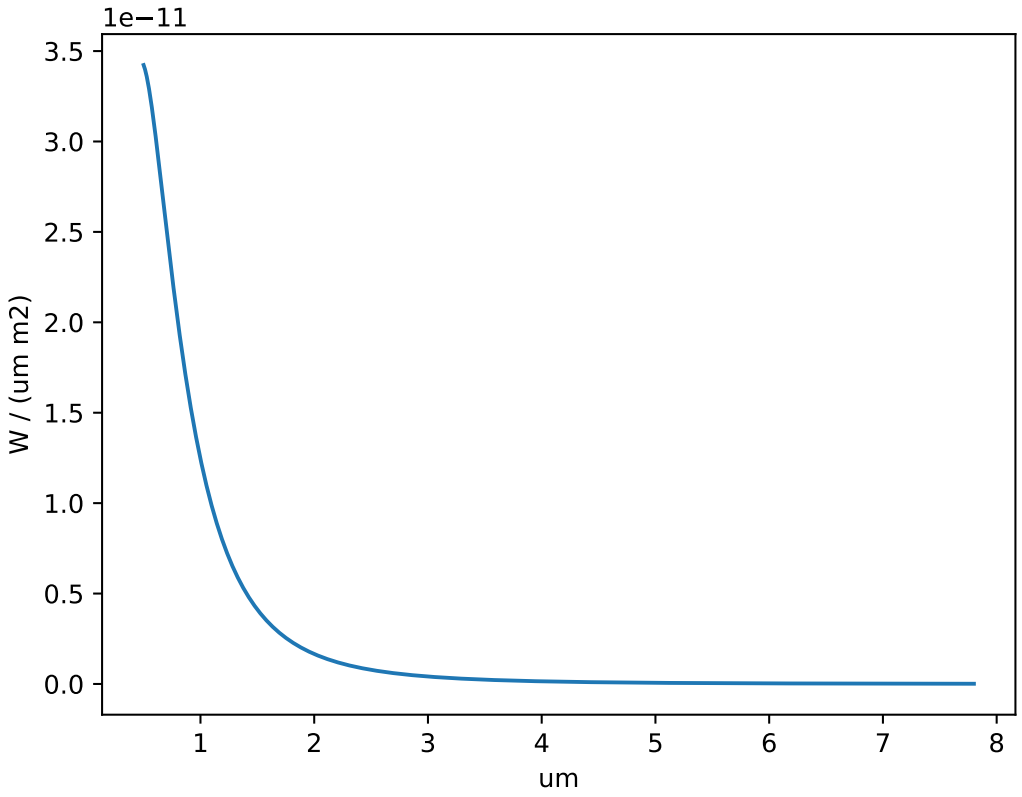
```
>>> from exosim.tasks.parse import ParseSource
>>> import astropy.units as u
>>> import numpy as np
>>> parseSource = ParseSource()
>>> wl = np.linspace(0.5, 7.8, 10000) * u.um
>>> tt = np.linspace(0.5, 1, 10) * u.hr
>>> source_in = {
>>>     'value': 'HD 209458',
>>>     'source_type': 'planck',
>>>     'R': 1.18 * u.R_sun,
>>>     'D': 47 * u.pc,
>>>     'T': 6086 * u.K,
>>>     }
>>> source_out = parseSource(parameters=source_in,
>>>                          wavelength=wl,
>>>                          time=tt)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(source_out['HD 209458'].spectral, source_out['HD 209458'].data[0,
↪0])
>>> plt.ylabel(source_out['HD 209458'].data_units)
>>> plt.xlabel(source_out['HD 209458'].spectral_units)
>>> plt.show()
```

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[546] https://docs.python.org/dev/library/stdtypes.html#dict
[547] https://docs.python.org/dev/library/stdtypes.html#dict


1e−11
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
W / (um m2)
1
2
3
4
5
6
7
8
um

exosim.tasks.parse.parseZodi

Module Contents

Classes

ParseZodi	This tasks parses the zodiacal foreground.

class ParseZodi

Bases: *exosim.tasks.task.Task*

This tasks parses the zodiacal foreground.

Returns
dictionary of *Radiance* and *Dimensionless*, representing the radiance and efficiency of the optical element.

Return type
dict[548]

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.radiometric

Submodules

exosim.tasks.radiometric.aperturePhotometry

Module Contents

Classes

AperturePhotometry	It performs the aperture photometry using photutils.aperture.aperture_photometry[549].

class AperturePhotometry

Bases: *exosim.tasks.task.Task*

It performs the aperture photometry using photutils.aperture.aperture_photometry[550].

The details of the aperture strongly depends on the configurations set by the user.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[548] https://docs.python.org/dev/library/stdtypes.html#dict
[549] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[550] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry

exosim.tasks.radiometric.computePhotonNoise

Module Contents

Classes

ComputePhotonNoise	Computes the photon noise.

class ComputePhotonNoise

Bases: *exosim.tasks.task.Task*

Computes the photon noise. Given the incoming signal S the resulting photon noise variance is $Var[S] = S$. If photon gain factor $gain_{phot}$ is given, then $Var[S] = gain_{phot} \cdot Var[S]$. If photon noise margin χ is found in the description, then $Var[S] = (1 + \chi) \cdot Var[S]$. The noise returned is $\sigma = \sqrt{Var[S]}$

Returns
photon noise

Return type
astropy.table.QTable[551]

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.radiometric.computeSignalsChannel

Module Contents

Classes

ComputeSignalsChannel	It estimates the radiometric signals on the input focal plane..

class ComputeSignalsChannel

Bases: *exosim.tasks.task.Task*

It estimates the radiometric signals on the input focal plane..

Returns
photometry

Return type
astropy.units.Quantity[552]

Raises

- **TypeError:** – if the output is not Quantity[553]
- **UnitsError:** – wrong output units

[551] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*table*, *focal_plane*, *parameters*)

It estimates the radiometric signals on the input focal plane.. It uses photutils.aperture.aperture_photometry[554] with the apertures from *EstimateApertures*.

Parameters

- **table** (astropy.table.QTable[555]) – apertures table
- **plane** (*focal*) – focal plane in the HDF5 file
- **parameters** (*dict*[556]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*

Returns
photometry

Return type
astropy.units.Quantity[557]

exosim.tasks.radiometric.computeSubFrgSignalsChannel

Module Contents

Classes

ComputeSubFrgSignalsChannel	It iteratively estimates the radiometric signals on the foregrounds sub focal planes for a channel

class ComputeSubFrgSignalsChannel

Bases: *exosim.tasks.task.Task*

It iteratively estimates the radiometric signals on the foregrounds sub focal planes for a channel and returns a table with all the contributions.

Returns
signal table

Return type
astropy.table.QTable[558]

Raises
TypeError: – if the output is not QTable[559]

[552] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[553] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[554] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[555] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[556] https://docs.python.org/dev/library/stdtypes.html#dict
[557] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*ch*, *table*, *input_file*, *parameters*)

It iteratively estimates the radiometric signals on the foregrounds sub focal plane for a channel and returns a table with all the contributions. It uses photutils.aperture.aperture_photometry[560] with the apertures from *EstimateApertures*.

Parameters

- **table** (astropy.table.QTable[561]) – apertures table
- **ch_name** (*str*[562]) – channel name
- **input_file** (*Output*) – input HDF5 file
- **parameters** (*dict*[563]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*

Returns
signal table

Return type
astropy.table.QTable[564]

exosim.tasks.radiometric.computeTotalNoise

Module Contents

Classes

ComputeTotalNoise	It computes the total noise from a radiometric table

class ComputeTotalNoise

Bases: *exosim.tasks.task.Task*

It computes the total noise from a radiometric table

Returns
table – total noise column

Return type
astropy.table.QTable[565]

[558] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[559] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[560] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[561] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[562] https://docs.python.org/dev/library/stdtypes.html#str
[563] https://docs.python.org/dev/library/stdtypes.html#dict
[564] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.radiometric.estimateApertures

Module Contents

Classes

EstimateApertures	It returns the sizes of the apertures to perform photometry for photutils.aperture.aperture_photometry[566].

class EstimateApertures

Bases: *exosim.tasks.task.Task*

It returns the sizes of the apertures to perform photometry for photutils.aperture.aperture_photometry[567]. The details of the apertures depends on the configurations set by the user.

Return type
astropy.table.QTable[568]

Raises
TypeError: – if the output is not QTable[569]

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*table*, *focal_plane*, *description*, *wl_grid*)

It returns the sizes of the apertures to perform photometry for photutils.aperture.aperture_photometry[570]. The default methods to estimate the aperture are the following and these are also the keywords required for the *description* dictionary.

- **spatial_mode: str**
  aperture in spatial direction. If *column* the aperture spatial length is the full pixel column in the array. Default is *column*.
- **spectral_mode: str**
  aperture in spectral direction. If *row* the aperture is spectral length is the full pixel row of the array. If *wl_solution* the spectral size is the same of the spectral bin in the input table. Default is 'wl_solution`.

[565] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[566] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry

- **auto_mode: str**
  automatic aperture mode. The dictionary must contains two keywords. The *mode* keyword should be a str: if *rectangular* then `find_rectangular_aperture` is used; if *elliptical* then `find_elliptical_aperture` is used; if *bin* then `find_bin_aperture` is used; if *full* then the full array is integrated.
- **Ene: float**
  The *EnE* keyword should be a float and represents the Encircled Energy to include in the aperture.

Parameters

- **table** (astropy.table.QTable[571]) – wavelength table with bin edge
- **focal_plane** – focal plane
- **description** (*dict*[572]) – dictionary containing the aperture photometry description

Return type
astropy.table.QTable[573]

exosim.tasks.radiometric.estimateSpectralBinning

Module Contents

Classes

EstimateSpectralBinning	It computes spectral binning useful to produce the radiometric tables

class EstimateSpectralBinning

Bases: *exosim.tasks.task.Task*

It computes spectral binning useful to produce the radiometric tables

Returns
wavelength bins table

Return type
astropy.table.QTable[574]

Raises

- **TypeError:** – if the output is not QTable[575]
- **KeyError:** – if a column is missing in the output QTable[576]

[567] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[568] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[569] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[570] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[571] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[572] https://docs.python.org/dev/library/stdtypes.html#dict
[573] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*wl_grid*, *parameters*)

It runs a dedicated method if the channel is a spectrometer or a photometer.

Parameters

- **wl_grid** (Quantity[577] (optional)) – focal plane wavelength grid. Useful for spectrometers. Default is None.
- **parameters** (*dict*[578]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*

Returns
wavelength bins table

Return type
astropy.table.QTable[579]

Raises
AttributeError: – if the channel type is not *spectrometer* or *photometer*

wavelength_table_spectrometer(*description*, *wl_grid*)

It returns the wavelength table for a spectrometer. The wavelength grid can be estimated in 2 modes:

- *native* mode. If *targetR* is set to *native* the wavelength grid computed is the pixel level wavelength grid, where each bin is of the size of a pixel;
- *fixed R* mode. If targetR` is set to a constant value, the wavelength grid is estimated using *wl_grid*.

Parameters

- **description** (*dict*[580]) – dictionary contained the channel parameters.
- **wl_grid** (Quantity[581]) – wavelength grid.

Returns
wavelength bins table

Return type
astropy.table.QTable[582]

Raises
KeyError[583] – Channel targetR format unsupported

wavelength_table_photometer(*description*)

It returns the wavelength table for a photometer. It is estimated as the central wavelength of the photometer with a bin width equal to the wavelength band.

Parameters
description (*dict*[584]) – dictionary contained the channel parameters.

Returns
wavelength bins table

Return type
astropy.table.QTable[585]

exosim.tasks.radiometric.multiaccum

Module Contents

Classes

Multiaccum	It computes the MULTIACCUM gain factor from Rauscher and Fox 2007 (http://iopscience.iop.org/article/10.1086/520887/pdf)

class Multiaccum

Bases: *exosim.tasks.task.Task*

It computes the MULTIACCUM gain factor from Rauscher and Fox 2007 (http://iopscience.iop.org/article/10.1086/520887/pdf) with the correction from Robberto 2009, also reported in Batalha 2017 (https://doi.org/10.1088/1538-3873/aa65b0)

Returns

- *float* – read noise gain factor
- *float* – shot noise gain factor

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.radiometric.saturationChannel

Module Contents

Classes

SaturationChannel	It computes and adds the saturation time to the radiometric table using *ComputeSaturation*.

[574] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[575] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[576] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[577] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[578] https://docs.python.org/dev/library/stdtypes.html#dict
[579] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[580] https://docs.python.org/dev/library/stdtypes.html#dict
[581] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[582] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable
[583] https://docs.python.org/dev/library/exceptions.html#KeyError
[584] https://docs.python.org/dev/library/stdtypes.html#dict
[585] https://docs.astropy.org/en/latest/api/astropy.table.QTable.html#astropy.table.QTable

class SaturationChannel

Bases: *exosim.tasks.task.Task*

It computes and adds the saturation time to the radiometric table using *ComputeSaturation*.

Returns

- *astropy.units.Quantity* – saturation time
- *astropy.units.Quantity* – maximum signal in focal plane
- *astropy.units.Quantity* – minimum signal in focal plane

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.sed

Submodules

exosim.tasks.sed.createCustomSource

Module Contents

Classes

CreateCustomSource	Creates a custom SED from input parameters.

class CreateCustomSource

Bases: *exosim.tasks.task.Task*

Creates a custom SED from input parameters.

Returns
Star Sed

Return type
Sed

Raises
TypeError: – if the output is not a *Sed* class

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*)

Parameters
parameters (*dict*[586]) – dictionary contained the source parameters. This is usually parsed from *LoadOptions*.

Returns
source sed

Return type
Sed

exosim.tasks.sed.createPlanckStar

Module Contents

Classes

CreatePlanckStar	Create a star SED using the Planck function.

class CreatePlanckStar

Bases: *exosim.tasks.task.Task*

Create a star SED using the Planck function.

The star emission is simulated by astropy.modeling.physical_models.BlackBody[587]. The resulting sed is then converted into $W/m^2/sr/\mu m$ and scaled by the solid angle $\pi \left(\frac{R}{D}\right)^2$.

Returns
Star Sed

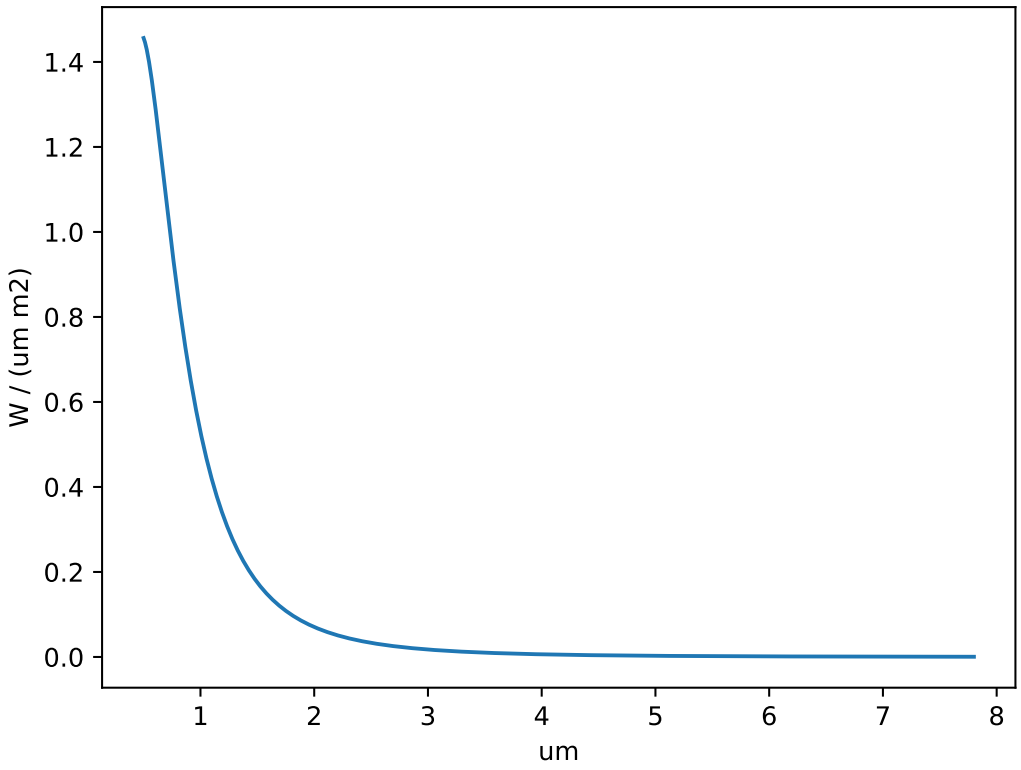
Return type
Sed

Examples

```
>>> from exosim.tasks.sed import CreatePlanckStar
>>> import astropy.units as u
>>> import numpy as np
>>> createPlanckStar = CreatePlanckStar()
>>> wl = np.linspace(0.5, 7.8, 10000) * u.um
>>> T = 6086 * u.K
>>> R = 1.18 * u.R_sun
>>> D = 47 * u.au
>>> sed = createPlanckStar(wavelength=wl, T=T, R=R, D=D)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(sed.spectral, sed.data[0,0])
>>> plt.ylabel(sed.data_units)
>>> plt.xlabel(sed.spectral_units)
>>> plt.show()
```

[586] https://docs.python.org/dev/library/stdtypes.html#dict


1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
W / (um m2)
1
2
3
4
5
6
7
8
um

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.sed.loadCustom

Module Contents

Classes

LoadCustom	Loads a custom SED from a file and scaled it by the solid angle $\pi \left(\frac{R}{D}\right)^2$.

class LoadCustom

Bases: *exosim.tasks.task.Task*

Loads a custom SED from a file and scaled it by the solid angle $\pi \left(\frac{R}{D}\right)^2$.

Returns
Star Sed

Return type
Sed

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.sed.loadPhoenix

Module Contents

Classes

LoadPhoenix	Loads a star SED from a grid of Phoenix spectra or from a specific Phoenix file.

class LoadPhoenix

Bases: *exosim.tasks.task.Task*

Loads a star SED from a grid of Phoenix spectra or from a specific Phoenix file.

Returns
Star Sed

Return type
Sed

[587] https://docs.astropy.org/en/latest/api/astropy.modeling.physical_models.BlackBody.html#astropy.modeling.physical_models.BlackBody

Examples

```
>>> from exosim.tasks.sed import LoadPhoenix
>>> loadPhoenix = LoadPhoenix()
```

Prepare the star

```
>>> from astropy import constants as cc
>>> import astropy.units as u
>>> import numpy as np
>>> D= 12.975 * u.pc
>>> T= 3016 * u.K
>>> M= 0.15 * u.Msun
>>> R= 0.218 * u.Rsun
>>> z= 0.0
>>> g = (cc.G * M.si / R.si ** 2).to(u.cm / u.s ** 2)
>>> logg = np.log10(g.value)
```

Load the sed from a directory

```
>>> sed = loadPhoenix(path = phoenix_directory,  T=T, D=D, R=R, z=z, logg=logg)
```

or load the sed from a file

```
>>> sed = loadPhoenix(filename = phoenix_file, D=D, R=R)
```

Notes

This class can either load the SED from a file or select the most suitable file from the Phoenix spectra path, given the proper information on the star. In the former case the *filename* keyword is needed, in the latter are required the keywords *path*, *T*, *z*, *logg*. Not providing the right keywords would result in an error.

Raises

- **InputError** – if neither *filename* or *path* are given.
- **KeyError**[588] – if a needed parameter is missing in the user inputs.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

load(*ph_file*)

Returs a sed given a Phoenix filename :param ph_file: filename :type ph_file: str

Returns
Star Sed

Return type
Sed

get_phoenix_model_filename(*path*, *T*, *logg*, *z*)

It returns the name of the phoenix file that best matches the input star parameters.

Parameters

- **path** (*str*[589]) – Phoenix directory path

- **T** (Quantity[590]) – star temperature
- **logg** (*float*[591]) – star logG
- **z** (*float*[592]) – star metallicity

Returns
Phoenix file name

Return type
str[593]

exosim.tasks.sed.prepareSed

Module Contents

Classes

PrepareSed	Returns a source SED.

class PrepareSed

Bases: *exosim.tasks.task.Task*

Returns a source SED. The SED depends on the source type selected. This Task analyse the inputs and return the desired SED.

Returns
Star Sed

Return type
Sed

Examples

We first define all the inputs and then we show how to produce the star sed:

```
>>> from exosim.tasks.sed import LoadPhoenix
>>> wavelength = np.linspace(0.5, 7.8, 10000) * u.um
>>> T = 5778 * u.K
>>> R = 1 * u.R_sun
>>> D = 1 * u.au
>>> M = 1 * u.Msun
>>> z = 0.0
>>> g = (cc.G * M.si / R.si ** 2).to(u.cm / u.s ** 2)
>>> logg = np.log10(g.value)
```

To produce a planck star:

[588] https://docs.python.org/dev/library/exceptions.html#KeyError
[589] https://docs.python.org/dev/library/stdtypes.html#str
[590] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[591] https://docs.python.org/dev/library/functions.html#float
[592] https://docs.python.org/dev/library/functions.html#float
[593] https://docs.python.org/dev/library/stdtypes.html#str

```
>>> prepareSed = LoadPhoenix()
>>> sed = PrepareSed(source_type='planck', wavelength=wavelength, T=T, R=R,␣
↪D=D)
```

To load a Phoenix star: >>> sed_1 = prepareSed(source_type='phoenix', path=phoenix_stellar_model, T=T, R=R, D=D, logg=logg, z=z)

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.subexposures

Submodules

exosim.tasks.subexposures.addForegrounds

Module Contents

Classes

AddForegrounds	It adds the foregrounds the sub exposures.

class AddForegrounds

Bases: *exosim.tasks.task.Task*

It adds the foregrounds the sub exposures.

Returns
sub-exposure cached signal class

Return type
Counts

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

static add_frg(*ndrs*, *frg*, *index*, *integration_time*)

exosim.tasks.subexposures.applyQeMap

Module Contents

Classes

ApplyQeMap	It applies the quantum efficiency variation map to the sub exposures.

class ApplyQeMap

Bases: *exosim.tasks.task.Task*

It applies the quantum efficiency variation map to the sub exposures.

Returns
sub-exposure cached signal class

Return type
Counts

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

static apply_qe(*se*, *qe*, *index*)

exosim.tasks.subexposures.computeReadingScheme

Module Contents

Classes

ComputeReadingScheme	It computes the reading scheme for the sub-exposures given the instrument parameters and the focal planes.

class ComputeReadingScheme

Bases: *exosim.tasks.task.Task*

It computes the reading scheme for the sub-exposures given the instrument parameters and the focal planes.

Returns

- astropy.units.Quantity[594] – simulation frequency.
- numpy.ndarray[595] – state machine for the reading operation on the ramp.
- numpy.ndarray[596] – state machine of the group ends sampling the ramp.
- numpy.ndarray[597] – state machine of the group starts sampling the ramp.
- numpy.ndarray[598] – full list of simulation stapes for each steps on the ramp repeated by the number of ramps.
- *int* – number of exposures needed to sample the full observation using ramps of the exposure time size.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

[594] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[595] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[596] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[597] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[598] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

exosim.tasks.subexposures.estimateChJitter

Module Contents

Classes

EstimateChJitter	It scales the pointing jitter expressed as deg into the pixel unit according to the channel plate scale

class EstimateChJitter

Bases: *exosim.tasks.task.Task*

It scales the pointing jitter expressed as deg into the pixel unit according to the channel plate scale and interpolates it to a time grid which is aligned to the detector readout time grid.

Returns

Tuple containing the pointing jitter in the spatial and spectral direction expressed in units of pixels.

Return type

(ndarray[599], ndarray[600])

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

exosim.tasks.subexposures.estimatePointingJitter

Module Contents

Classes

EstimatePointingJitter	Produces the telescope pointing jitter expressed as deg on the line of sight.

class EstimatePointingJitter

Bases: *exosim.tasks.task.Task*

Produces the telescope pointing jitter expressed as deg on the line of sight.

Returns

- Quantity[601], – pointing jitter in the spatial and spectral direction expressed in units of deg.
- Quantity[602] – pointing jitter in the spatial and spectral direction expressed in units of deg.
- Quantity[603] – pointing jitter in the spatial and spectral direction expressed in units of deg.

[599] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[600] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

Raises

TypeError: – if the output is not a Quantity[604] class

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*)

This default model builds the pointing jitter using random values. Starting from the spectral and spatial standard deviations, expressed as angles, it computes the pointing position as normally distributed around zero in the two direction. The input dictionary, under the *jitter* keyword, must contain *spatial* and *spectral* keyword for the standard deviation. The time grid is built using the *high_frequencies_resolution* under the *time_grid* keyword.

Parameters

parameters (*dict*[605]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*

Returns

pointing jitter in the spatial direction expressed in units of deg. Quantity[606])

pointing jitter in the spectral direction expressed in units of deg.

Quantity[Page 275, 607])

pointing jitter timeline expressed in units of s.

Return type

Quantity[608]

exosim.tasks.subexposures.instantaneousReadOut

Module Contents

Classes

InstantaneousReadOut	This task implements the instantaneous read out.

[601] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[602] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[603] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[604] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[605] https://docs.python.org/dev/library/stdtypes.html#dict
[606] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[607] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[608] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

class InstantaneousReadOut

Bases: *exosim.tasks.task.Task*

This task implements the instantaneous read out. It loads the readout configuration from a dictionary that is produced by *PrepareInstantaneousReadOut*. Then it creates the sub-exposure datacube, with sub-exposure for each NDRs in the ramp sampling scheme. Each of these sub-exposures collects more simulation steps, which have their own jitter offset. This Task iterates over the time steps to jitter the focal plane and pile it to the appropriate sub-exposure. The jittering is based on the focal plane oversampling factor. The contributions to each sub-exposures are averaged and the final product is multiplied by its integration time.

Returns
sub-exposure cached signal class

Return type
Counts

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

force_power_conservation(*out*, *parameters*, *focal_plane*, *fp_time*, *osf*)

static oversample(*fp*, *ad_osf*)

It increases the oversampling factor of the focal plane.

Parameters

- **fp** (ndarray[609]) – 2D focal plane
- **ad_osf** (*int*[610]) – magnification factor

Returns
2D focal plane sampled with the new oversampling factor

Return type
ndarray[611]

static jittering_the_focalplane(*fp*, *osf*, *start_index*, *end_index*, *x_jit*, *y_jit*, *fp_time*)

Parameters

- **fp** (*numpy.array*) –
- **osf** (*int*[612]) –
- **start_index** (*numpy.array*) –
- **end_index** (*numpy.array*) –
- **x_jit** (*numpy.array*) –
- **y_jit** (*numpy.array*) –
- **fp_time** (*numpy.array*) –

Return type
numpy.array

static replicating_the_focalplane(*fp*, *index*, *fp_time*)

[609] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[610] https://docs.python.org/dev/library/functions.html#int
[611] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[612] https://docs.python.org/dev/library/functions.html#int

exosim.tasks.subexposures.loadILS

Module Contents

Classes

LoadILS	This Task loads the instrument line shapes (ILS) from the input file.

class LoadILS

Bases: *exosim.tasks.task.Task*

This Task loads the instrument line shapes (ILS) from the input file. The loaded ILS are then used to convolve the astronomical signal. The ILS here are intended as the PSF of the instrument. The ILS shapes are normalized to the maximum value.

Returns
instrument line shape. The shape is (time, wavelength, spectral direction)

Return type
numpy.ndarray[613]

Note: The instrument line shapes produced by this task are not the same as the instrument line shapes as defined in the literature. The ILS produced by this task are the PSF of the instrument. To be used as the instrument line shapes as defined in the literature they need to be convolved with the intra-pixel response. This convolution is not part of this Task as it affects the way the ILS are sampled. The convolution with the intra-pixel response is done in the *exosim.tasks.astrosignal.applyAstronomicalSignal.ApplyAstronomicalSignal* Task, where the ILS are used to convolve the astronomical signal.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*input_file*, *parameters*, *wl_grid*)

It loads the channel instrument line shapes from the input file.

Parameters

- **input_file** (*str*[614]) – focal plane input file
- **ch_param** (*dict*[615]) – channel parameter
- **wl_grid** (astropy.units.Quantity[616]) – output wavelength grid
- **parameters** (*dict*[617]) –

Returns
instrument line shape. The shape is (time, wavelength, spectral direction)

Return type
numpy.ndarray[618]

exosim.tasks.subexposures.loadQeMap

Module Contents

Classes

LoadQeMap	Loads the Quantum efficiency map

class LoadQeMap

Bases: *exosim.tasks.task.Task*

Loads the Quantum efficiency map

Returns
channel responsivity variation map

Return type
Signal

Raises
TypeError: – if the output is not a *Signal* class

Notes

This is a default class with standardised inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*, *time*)

Parameters

- **parameters** (*dict*[619]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*
- **time** (Quantity[620]) – time grid.

Returns
channel responsivity

Return type
Signal

[613] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[614] https://docs.python.org/dev/library/stdtypes.html#str
[615] https://docs.python.org/dev/library/stdtypes.html#dict
[616] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[617] https://docs.python.org/dev/library/stdtypes.html#dict
[618] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[619] https://docs.python.org/dev/library/stdtypes.html#dict
[620] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

exosim.tasks.subexposures.loadQeMapNumpy

Module Contents

Classes

LoadQeMapNumpy	Loads the Quantum efficiency map from a NPY file (see numpy documentation[621]).

class LoadQeMapNumpy

Bases: exosim.tasks.subexposures.LoadQeMap

Loads the Quantum efficiency map from a NPY file (see numpy documentation[622]).

Returns
channel responsivity variation map

Return type
Signal

Raises
TypeError: – if the output is not a *Signal* class

model(*parameters*, *time*)

Parameters

- **parameters** (*dict*[623]) – dictionary contained the channel parameters. This is usually parsed from *LoadOptions*
- **time** (Quantity[624]) – time grid.

Returns
channel responsivity

Return type
Signal

exosim.tasks.subexposures.prepareInstantaneousReadOut

Module Contents

Classes

PrepareInstantaneousReadOut	This task prepares the instantaneous read out.

[621] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html
[622] https://numpy.org/devdocs/reference/generated/numpy.lib.format.html
[623] https://docs.python.org/dev/library/stdtypes.html#dict
[624] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

class PrepareInstantaneousReadOut

Bases: *exosim.tasks.task.Task*

This task prepares the instantaneous read out. It calls *ComputeReadingScheme* to compute the ramp sampling scheme, and *EstimateChJitter* to scale the input pointing jitter to the focal planet pixel units. The jittering is based on the focal plane oversampling factor.

Returns

- *dict* – readout_parameters dict
- Quantity[625] – sub-exposures integration times

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

force_power_conservation(*out*, *parameters*, *focal_plane*, *fp_time*, *osf*)

Submodules

exosim.tasks.chainTask

Module Contents

Classes

ChainTask	Abstract class to operate on a *Signal* and return a *Signal*.

class ChainTask

Bases: *exosim.tasks.task.Task*

Abstract class to operate on a *Signal* and return a *Signal*.

Returns
optical element radiance

Return type
Radiance

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

abstract model(*parameters*, *wavelength*, *time*)

Parameters

- **signal** (*Signal*) – input signal
- **parameters** (*dict*[626]) – dictionary containing the parameters. This is usually parsed from *LoadOptions*
- **wavelength** (Quantity[627]) – wavelength grid.

[625] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

- **time** (Quantity[628]) – time grid.

Returns
output signal c

Return type
Signal

exosim.tasks.task

Module Contents

Classes

Task	*Abstract class*

class Task

Bases: exosim.log.Logger, *exosim.utils.timed_class.TimedClass*

Abstract class

Base class for tasks.

abstract execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

Return type
None

get_output()

Returns the output values.

Return type
Any

set_output(*product*)

It sets the values to return.

Parameters
product (*Any*) –

Return type
None

get_task_param(*paramName*)

It get the value from the task parameter.

Parameters
paramName (*str*[629]) –

Return type
Any

[626] https://docs.python.org/dev/library/stdtypes.html#dict
[627] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[628] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

add_task_param(*param_name*, *param_description*, *default=None*)

It adds a parameter for the task.

Parameters

- **param_name** (*str*[630]) –
- **param_description** (*str*[631]) –
- **default** (*Any*) –

Return type
None

exosim.tools

Submodules

exosim.tools.adcGainEstimator

Module Contents

Classes

ADCGainEstimator	It computes the desired ADC gain, given the number of bits used by the ADC

class ADCGainEstimator(*options_file*, *output=None*)

Bases: *exosim.tools.exosimTool.ExoSimTool*

It computes the desired ADC gain, given the number of bits used by the ADC and the maximum number of counts in the pixel to represent.

Returns
ADC gain factors

Return type
dict[632]

Examples

```
>>> import exosim.tools as tools
>>>
>>> tools.ADCGainEstimator(options_file='tools_input_example.xml')
```

model(*parameters*)

Parameters
parameters (*dict*[633]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*

[629] https://docs.python.org/dev/library/stdtypes.html#str
[630] https://docs.python.org/dev/library/stdtypes.html#str
[631] https://docs.python.org/dev/library/stdtypes.html#str

Returns

- *float* – adc gain factor
- *int* – max adc value
- *str* – data type used in ExoSim

exosim.tools.darkCurrentMap

Module Contents

Classes

DarkCurrentMap	Produces the channel dark current map

class DarkCurrentMap

Bases: *exosim.tasks.task.Task*

Produces the channel dark current map

Returns
channel dark current map

Return type
Signal

Raises
TypeError: – if the output is not a *Signal* class

Notes

This is a default class with standardized inputs and outputs. The user can load this class and overwrite the "model" method to implement a custom Task to replace this.

execute()

Class execution. It runs on call and executes all the task actions returning the outputs. It requires the input with correct keywords

model(*parameters*, *time*)

Parameters

- **parameters** (*dict*[634]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **time** (Quantity[635]) – time grid.

Returns
dark current efficiency map

Return type
Signal

[632] https://docs.python.org/dev/library/stdtypes.html#dict
[633] https://docs.python.org/dev/library/stdtypes.html#dict

compute_dc_mean(*detector*)

Computes the mean of the dark current (dc_mean) from the log-normal distributon.

The probability density function for the log-normal distributon is:

$$pdf(x) =$$

rac{1}{sigma x sqrt{2pi}}
expleft(-

rac{(log(x) - mu)^2}{2 sigma^2} ight)

The mean of the pdf can be computed as:

$$mean = \exp(\mu +$$

rac{s^2}{2})

where s is the pdf standard deviation, computed by taking the sqrt of the variance, defined as:

$$var = \left(\exp(\sigma^2) - 1\right.$$

ight) exp(2 mu + sigma^2)

detector: dict
Dictionary for the detector. This is usually parsed from *LoadOptions*

None
Updates the detector dictionary with the value of dc_mean

exosim.tools.deadPixelsMap

Module Contents

Classes

DeadPixelsMap	Produces the channel dead pixel map

class DeadPixelsMap(*options_file*, *output=None*)

Bases: *exosim.tools.exosimTool.ExoSimTool*

Produces the channel dead pixel map

Returns
channels' dead pixels maps

Return type
dict[636]

Raises
TypeError: – if the output is not a Table[637] class

[634] https://docs.python.org/dev/library/stdtypes.html#dict
[635] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Examples

```
>>> import exosim.tools as tools
>>>
>>> tools.DeadPixelsMap(options_file='tools_input_example.xml',
>>>                     output='./')
```

model(*parameters*)

Parameters
parameters (*dict*[638]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*

Returns
channel dead pixel coordinates

Return type
Table[639]

exosim.tools.exosimTool

Module Contents

Classes

ExoSimTool	*Abstract class*

class ExoSimTool(*options_file*)

Bases: exosim.log.Logger

Abstract class

Base class for exosim tools.

Variables

- **ch_param** (*dict*[640]) – dictionary for the channels' configurations
- **options** (*dict*[641]) – configurations dictionary
- **results** (*dict*[642]) – dictionary for the results output

Parameters
options_file (*Union[str*[643], *dict*[644]*]*) –

property ch_list: List[str[645]**]**

list of channel names

Return type
List[str[646]]

[636] https://docs.python.org/dev/library/stdtypes.html#dict
[637] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table
[638] https://docs.python.org/dev/library/stdtypes.html#dict
[639] https://docs.astropy.org/en/latest/api/astropy.table.Table.html#astropy.table.Table

exosim.tools.pixelsNonLinearity

Module Contents

Classes

PixelsNonLinearity	This tools helps the user to find the pixel non-linearity coefficients to as inputs for ExoSim,

class PixelsNonLinearity(*options_file*, *output=None*, *show_results=True*)

Bases: *exosim.tools.exosimTool.ExoSimTool*

This tools helps the user to find the pixel non-linearity coefficients to as inputs for ExoSim, starting from the an estimate of pixel non-linearity correction.

This class will retrieve the a_i coefficients, starting from physical assumptions.

The detector non linearity model, is written as polynomial such as

$$Q_{det} = Q \cdot (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi_t$, with ϕ being the number of electrons generated and t being the elapsed time.

Considering the detector as a capacitor, the charge Q_{det} is given by

$$Q_{det} = \phi\tau \cdot \left(1 - e^{-Q/\phi\tau}\right)$$

where ϕ is the charge generated in the detector pixel, and τ is the capacitor time constant. In fact the product $\phi\tau$ is constant Q is the response of a linear detectror is given by $Q = \phi t$

The detector is considered saturated when the charge Q_{det} at the well depth $Q_{det,\,wd}$ differs from the ideal well depth Q_{wd} by 5%.

$$Q_{det} = (1 - 5\%)Q_{wd}$$

Then

$$\phi\tau \cdot \left(1 - e^{-Q_{wd}/\phi\tau}\right) = (1 - 5\%)Q_{wd}$$

This equation can be solved numerically and gives

$$\frac{Q_{wd}}{\phi\tau} \sim 0.103479$$

Therefore the detector collected charge is given by

$$Q_{det} = \frac{Q_{wd}}{0.103479} \cdot \left(1 - e^{\frac{-0.103479\,Q}{Q_{wd}}}\right)$$

[640] https://docs.python.org/dev/library/stdtypes.html#dict
[641] https://docs.python.org/dev/library/stdtypes.html#dict
[642] https://docs.python.org/dev/library/stdtypes.html#dict
[643] https://docs.python.org/dev/library/stdtypes.html#str
[644] https://docs.python.org/dev/library/stdtypes.html#dict
[645] https://docs.python.org/dev/library/stdtypes.html#str
[646] https://docs.python.org/dev/library/stdtypes.html#str

Which can be approximated by a polynomial of order 4 as

$$
\begin{aligned}
Q_{det} = Q \Bigg[1 &- \frac{1}{2!}\frac{0.103479}{Q_{wd}}Q \\
&+ \frac{1}{3!}\left(\frac{0.103479}{Q_{wd}}\right)^2 Q^2 \\
&- \frac{1}{4!}\left(\frac{0.103479}{Q_{wd}}\right)^3 Q^3 \\
&+ \frac{1}{5!}\left(\frac{0.103479}{Q_{wd}}\right)^4 Q^4
\end{aligned}
$$

The results are the coefficients for a 4-th order polynomial:

$$Q_{det} = Q \cdot (a_1 + a_2 \cdot Q + a_3 \cdot Q^2 + a_4 \cdot Q^3 + a_5 \cdot Q^4)$$

However, each pixel is different, and therefore, this class also produces a map of the coefficient for each pixel. Each coefficient is normally distributed around the mean value, with a standard deviation indicated in the configuration. If no standard deviation is indicated, the coefficients are assumed to be constant.

The code output is a map of a_i coefficients for each pixel, which can be injected into *ApplyPixelsNonLinearity*.

Examples

```
>>> import exosim.tools as tools
>>>
>>> results = tools.PixelsNonLinearity(options_file='tools_input_example.xml',
>>>                                    output='output_pnl_map.h5')
```

Parameters

- **options_file** (*Union[str[647], dict[648]]*) –
- **output** (*str[649]*) –
- **show_results** (*bool[650]*) –

compute_coefficients(*parameters*, *show_results=True*)

It computes the non linearity coefficients.

Parameters

- **parameters** (*dict[651]*) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **show_results** (*bool[652]*) – it tells the code if showing the results in a plot. Default is *True*.

Return type

Tuple[List[float[653]], float[654]]

create_map(*parameters*, *input_dict*, *show_results=True*)

Create a map of the pixel non-linearity correction coefficients. To create a non linearity map of the detector, we randomize the coefficients. If not specified, the standard deviation is set to 0 and the coefficients are assumed to be standard. of the mean value of the polynomial coefficients

Parameters

- **parameters** (*dict*[655]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **input_dict** (*dict*[656]) – dictionary produced by *compute_coefficients*. It contains the coefficients (*coeff*), the saturation ('saturation') and the estimated well depth (*well_depth*)
- **show_results** (*bool*[657]) – it tells the code if showing the results in a plot. Default is *True*.

Returns
map of the coefficients. The shape is (4, nrow, ncol) The first axes refers to the coefficients (a, b, c, d, e)

Return type
np.ndarray

exosim.tools.pixelsNonLinearityFromCorrection

Module Contents

Classes

PixelsNonLinearityFromCorrection	This tools helps the user to find the pixel non-linearity coefficients to as inputs for ExoSim,

class PixelsNonLinearityFromCorrection(*options_file*, *output=None*, *show_results=True*)

Bases: *exosim.tools.pixelsNonLinearity.PixelsNonLinearity*

This tools helps the user to find the pixel non-linearity coefficients to as inputs for ExoSim, starting from the measurable pixel non-linearity correction.

In fact, the detector non linearity model, is usually written as polynomial such as

$$Q_{det} = Q \triangle (1 + \sum_i a_i \cdot Q^i)$$

where Q_{det} is the charge read by the detector, and Q is the ideal count, as $Q = \phi_t$, with ϕ being the number of electrons generated and t being the elapsed time. In the equation above, $\triangle$ is the operator used to defined the relation between Q_{det} and Q, which depends on the definition of the coefficients a_i (see also equation below).

[647] https://docs.python.org/dev/library/stdtypes.html#str
[648] https://docs.python.org/dev/library/stdtypes.html#dict
[649] https://docs.python.org/dev/library/stdtypes.html#str
[650] https://docs.python.org/dev/library/functions.html#bool
[651] https://docs.python.org/dev/library/stdtypes.html#dict
[652] https://docs.python.org/dev/library/functions.html#bool
[653] https://docs.python.org/dev/library/functions.html#float
[654] https://docs.python.org/dev/library/functions.html#float
[655] https://docs.python.org/dev/library/stdtypes.html#dict
[656] https://docs.python.org/dev/library/stdtypes.html#dict
[657] https://docs.python.org/dev/library/functions.html#bool

However, it is usually the inverse operation that is known, as it's coefficients are measurable empirically:

$$Q = Q_{det} \bigtriangledown (b_1 + \sum_{i=2} b_i \cdot Q_{det}^i)$$

Where $\bigtriangledown$ is the inverse operator of $\bigtriangleup$. Depending on the way the non linearity is estimated, the operator can either be a division ($\div$) or a multiplication ($\times$). If not specified, a division is assumed.

The b_i correction coefficients should be listed in the configuration file using the *pnl_coeff* keyword in increasing alphabetical order: *pnl_coeff_a* for b_1, *pnl_coeff_b* for b_2, *pnl_coeff_c* for b_3, *pnl_coeff_d* for b_4, *pnl_coeff_e* for b_5 and so on. The user can list any number of correction coefficients, and they will be automatically parsed. Please, note that using this notation, b_1 is not forced to be the unity.

This class will restrieve the a_i coefficients, starting from the the indicated b_i. The results are the coefficients for a 4-th order polynomial:

$$Q_{det} = Q \cdot (a_1 + a_2 \cdot Q + a_3 \cdot Q^2 + a_4 \cdot Q^3 + a_5 \cdot Q^4)$$

However, each pixel is different, and therefore, this class also produces a map of the coefficient for each pixel. Each coefficient is normally distributed around the mean value, with a standard deviation indicated in the configuration. If no standard deviation is indicated, the coefficients are assumed to be constant.

The code output is a map of a_i coefficients for each pixel, which can be injected into *ApplyPixelsNonLinearity*.

Variables

- **operator_dict** (*dict*[658]) – dictionary of operators to use to estimates Q_{det}
- **Npt** (*int*[659]) – number of points used to estimate the coefficients

Parameters

- **options_file** (*Union[str*[660], *dict*[661]*]*) –
- **output** (*str*[662]) –
- **show_results** (*bool*[663]) –

Examples

```
>>> import exosim.tools as tools
>>>
>>> results = tools.PixelsNonLinearityFromCorrection(options_file='tools_input_
↪example.xml',
>>>                                                  output='output_pnl_map.h5')
```

operator_dict

Npt

compute_coefficients(*parameters*, *show_results=True*)

It computes the non linearity coefficients.

Parameters

- **parameters** (*dict*[664]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **show_results** (*bool*[665]) – it tells the code if showing the results in a plot. Default is *True*.

Return type
Tuple[List[float[666]], float[667], float[668]]

exosim.tools.quantumEfficiencyMap

Module Contents

Classes

QuantumEfficiencyMap	Produces the channels quantum efficiency variation map

class QuantumEfficiencyMap(*options_file*, *output=None*)

Bases: *exosim.tools.exosimTool.ExoSimTool*

Produces the channels quantum efficiency variation map

Returns
channels' quantum efficiency map

Return type
dict[669]

Raises
TypeError: – if the output is not a *Signal* class

Examples

```
>>> import exosim.tools as tools
>>>
>>> results = tools.QuantumEfficiencyMap(options_file='tools_input_example.xml
↪',
>>>                                      output='output_qe_map.h5')
```

model(*parameters*, *time*)

Parameters

- **parameters** (*dict*[670]) – dictionary contained the sources parameters. This is usually parsed from *LoadOptions*
- **time** (Quantity[671]) – time grid.

[658] https://docs.python.org/dev/library/stdtypes.html#dict
[659] https://docs.python.org/dev/library/functions.html#int
[660] https://docs.python.org/dev/library/stdtypes.html#str
[661] https://docs.python.org/dev/library/stdtypes.html#dict
[662] https://docs.python.org/dev/library/stdtypes.html#str
[663] https://docs.python.org/dev/library/functions.html#bool
[664] https://docs.python.org/dev/library/stdtypes.html#dict
[665] https://docs.python.org/dev/library/functions.html#bool
[666] https://docs.python.org/dev/library/functions.html#float
[667] https://docs.python.org/dev/library/functions.html#float
[668] https://docs.python.org/dev/library/functions.html#float

Returns
channel quantum efficiency map

Return type
Signal

exosim.tools.readoutSchemeCalculator

Module Contents

Classes

ReadoutSchemeCalculator	This tool helps in the definition of the channels' readout schemes.

class ReadoutSchemeCalculator(*options_file*, *input_file*)

Bases: *exosim.tools.exosimTool.ExoSimTool*

This tool helps in the definition of the channels' readout schemes. Starting from the desired readout format, this tool estimates the best parameters to set in the channel description to best fit the ramp.

Variables

- **input** (*str*[672]) – input file
- **options** (*dict*[673]) – This is parsed from *LoadOptions*

Examples

```
>>> import exosim.tools as tools
>>>
>>> tools.ReadoutSchemeCalculator(options_file = 'tools_input_example.xml',
>>>                               input_file='input_file.h5')
```

load_focal_plane(*ch*)

It loads the channel focal plane from the input file:

Parameters
ch (*str*[674]) – channel name

Returns

- *CountsPerSecond* – focal plane
- *CountsPerSecond* – foreground focal plane

669 https://docs.python.org/dev/library/stdtypes.html#dict
670 https://docs.python.org/dev/library/stdtypes.html#dict
671 https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
672 https://docs.python.org/dev/library/stdtypes.html#str
673 https://docs.python.org/dev/library/stdtypes.html#dict
674 https://docs.python.org/dev/library/stdtypes.html#str

exosim.utils

Submodules

exosim.utils.aperture

Module Contents

Functions

find_rectangular_aperture(ima, desired_ene[, center, ...])	Estimates the smallest rectangular aperture to collect the desired Encircled Energy from a PSF.
find_elliptical_aperture(ima, desired_ene[, center])	Estimates the smallest elliptical aperture to collect the desired Encircled Energy from a PSF.
find_bin_aperture(ima, desired_ene, spatial_with[, center])	Estimates the smallest rectangular aperture for spectral bin of already fixed spatial size

Attributes

logger

logger

find_rectangular_aperture(*ima*, *desired_ene*, *center=None*, *start_h=None*, *start_w=None*)

Estimates the smallest rectangular aperture to collect the desired Encircled Energy from a PSF. It uses photutils.aperture.aperture_photometry[675].

Parameters

- **ima** (ndarray[676]) – two-dimensional array.
- **desired_ene** (*float*[677]) – desired Encircled Energy
- **center** ((*float*[678], *float*[679])) – spectral and spatial coordinates of the aperture center in pixels. If None the center of the array is used. Default is *None*
- **start_h** (*int*[680]) – starting aperture height
- **start_w** (*int*[681]) – starting aperture width

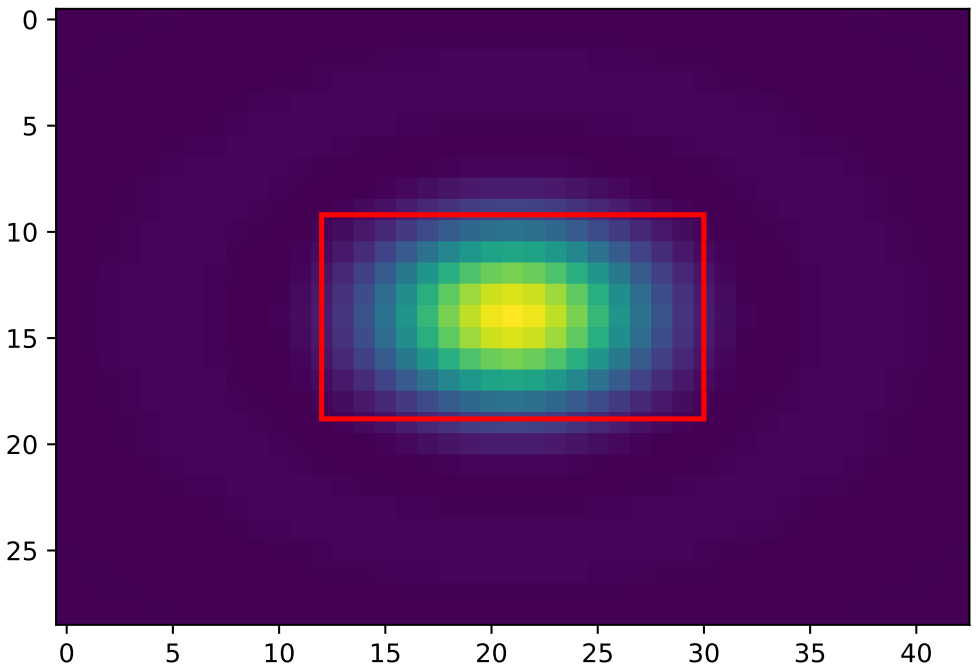
Returns

- *(float, float)* – sizes of the rectangular aperture (h,w)
- *float* – number of pixels in the aperture
- *float* – encircled energy collected by the aperture

Example

In this example we find the optimal aperture containing the 85% of the energy of a PSF.

```
>>> import matplotlib.pyplot as plt
>>> import astropy.units as u
>>> import photutils
>>> from exosim.utils.aperture import find_rectangular_aperture
>>> from exosim.utils.psf import create_psf
>>>
>>> img = create_psf(1*u.um, (60,40), 6*u.um)
>>> size, area, ene = find_rectangular_aperture(img, 0.85)
>>> positions = [(img.shape[1]//2,img.shape[0]//2)]
>>> aperture = photutils.aperture.RectangularAperture(positions, size[0],␣
↪size[1])
>>>
>>> plt.imshow(img)
>>> aperture.plot(color='r', lw=2,)
>>> plt.show()
```



675 https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
676 https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
677 https://docs.python.org/dev/library/functions.html#float
678 https://docs.python.org/dev/library/functions.html#float
679 https://docs.python.org/dev/library/functions.html#float
680 https://docs.python.org/dev/library/functions.html#int
681 https://docs.python.org/dev/library/functions.html#int

find_elliptical_aperture(*ima*, *desired_ene*, *center=None*)

Estimates the smallest elliptical aperture to collect the desired Encircled Energy from a PSF. It uses photutils.aperture.aperture_photometry[682].

Parameters

- **ima** (ndarray[683]) – two-dimensional array.
- **desired_ene** (*float*[684]) – desired Encircled Energy
- **center** ((*float*[685], *float*[686])) – spectral and spatial coordinates of the aperture center in pixels. If None the center of the array is used. Default is *None*

Returns

- *(float, float)* – sizes of the elliptical aperture (h,w)
- *float* – number of pixels in the aperture
- *float* – encircled energy collected by the aperture

Example

In this example we find the optimal aperture containing the 85% of the energy of a PSF.

```
>>> import matplotlib.pyplot as plt
>>> import astropy.units as u
>>> import photutils
>>> from exosim.utils.psf import find_elliptical_aperture
>>> from exosim.utils.psf import create_psf
>>>
>>> img = create_psf(1*u.um, (60,40), 6*u.um)
>>> size, area, ene = find_elliptical_aperture(img, 0.85)
>>> positions = [(img.shape[1]//2,img.shape[0]//2)]
>>> aperture = photutils.aperture.EllipticalAperture(positions, size[0],␣
↪size[1])
>>>
>>> plt.imshow(img)
>>> aperture.plot(color='r', lw=2,)
>>> plt.show()
```

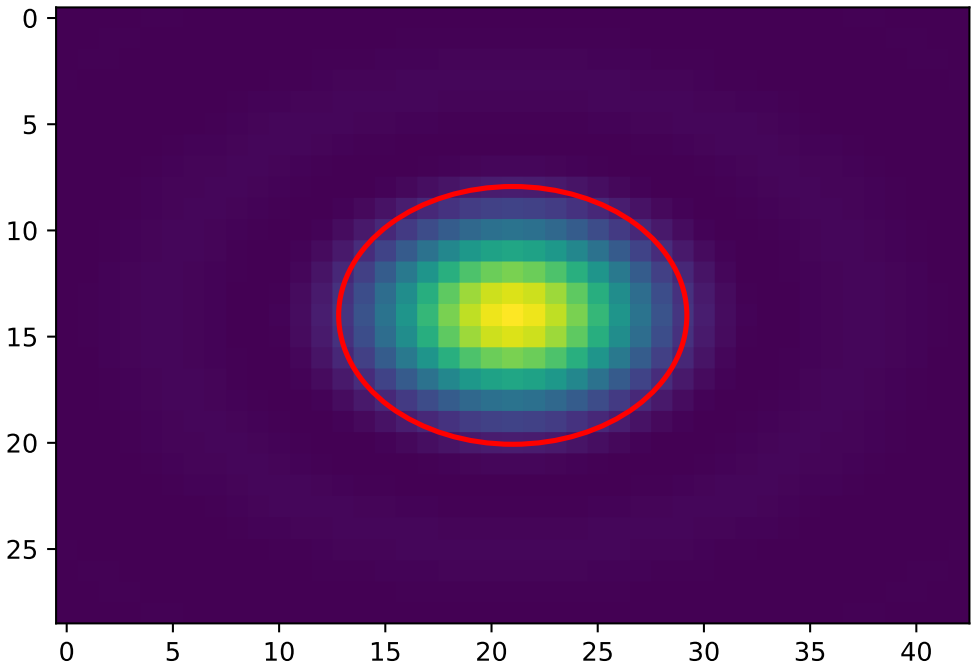
find_bin_aperture(*ima*, *desired_ene*, *spatial_with*, *center=None*)

Estimates the smallest rectangular aperture for spectral bin of already fixed spatial size to collect the desired Encircled Energy from a PSF. It uses photutils.aperture.aperture_photometry[687].

Parameters

- **ima** (ndarray[688]) – two-dimensional array.
- **spatial_with** (*float*[689]) – fixed bin spatial size
- **desired_ene** (*float*[690]) – desired Encircled Energy
- **center** ((*float*[691], *float*[692])) – spectral and spatial coordinates of the aperture center in pixels. If None the center of the array is used. Default is *None*

[682] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[683] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[684] https://docs.python.org/dev/library/functions.html#float
[685] https://docs.python.org/dev/library/functions.html#float
[686] https://docs.python.org/dev/library/functions.html#float


0
5
10
15
20
25
0
5
10
15
20
25
30
35
40

Returns

- *float* – spectral size of the rectangular aperture
- *float* – number of pixels in the aperture
- *float* – encircled energy collected by the aperture

exosim.utils.ascii_arts

Module Contents

astronomer1

astronomer2

astronomer3

astronomer4

observatory

exosim.utils.binning

Module Contents

Functions

rebin(x, xp, fp[, axis, mode, fill_value])	This function can resample multidimensional array along a given axis.

Attributes

logger

rebin(*x*, *xp*, *fp*, *axis=0*, *mode='mean'*, *fill_value=0.0*)

This function can resample multidimensional array along a given axis. Resample a function fp(xp) over the new grid x, rebinning if necessary, otherwise interpolates. Interpolation is done using 'linear' method. This function doesn't perform extrapolation: unsample coordinates will be filled with filled value.

Parameters

- **x** (ndarray[693]) – New coordinates

[687] https://photutils.readthedocs.io/en/stable/api/photutils.aperture.aperture_photometry.html#photutils.aperture.aperture_photometry
[688] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[689] https://docs.python.org/dev/library/functions.html#float
[690] https://docs.python.org/dev/library/functions.html#float
[691] https://docs.python.org/dev/library/functions.html#float
[692] https://docs.python.org/dev/library/functions.html#float

- **fp** (ndarray[694]) – y-coordinates to be resampled
- **xp** (ndarray[695]) – x-coordinates at which fp are sampled
- **axis** (*int*[696] *(optional)*) – fp axis to resample. Default is 0.
- **mode** (*str*[697] *(optional)*) – the mode indicates the statistc to use for binning by scipy.stats.binned_statistic[698]. Default is 'mean'.
- **fill_value** (*float*[699] *(optional)*) – fill value for unsampled coordinates. Default is 0.0.

Returns
re-sampled fp

Return type
ndarray[700]

Raises
NotImplementedError[701] – If the mode is not implemented.

Examples

```
>>> import numpy as np
>>> from exosim.utils.binning import rebin
```

We define the original function, sampled in data 50 points:

```
>>> xp = np.linspace(1,10, 50)
>>> fp = np.sin(xp)
```

We bin it down, sampling it at 10 points

```
>>> x_bin = np.linspace(1,10, 10)
>>> f_bin = rebin(x_bin, xp, fp)
```

We wl_interpolate the original function, sampling it at 100 points

```
>>> x_inter = np.linspace(1,10, 100)
>>> f_inter = rebin(x_inter, xp, fp)
```

To compare the outcome of our interpolation, we produce a plot.

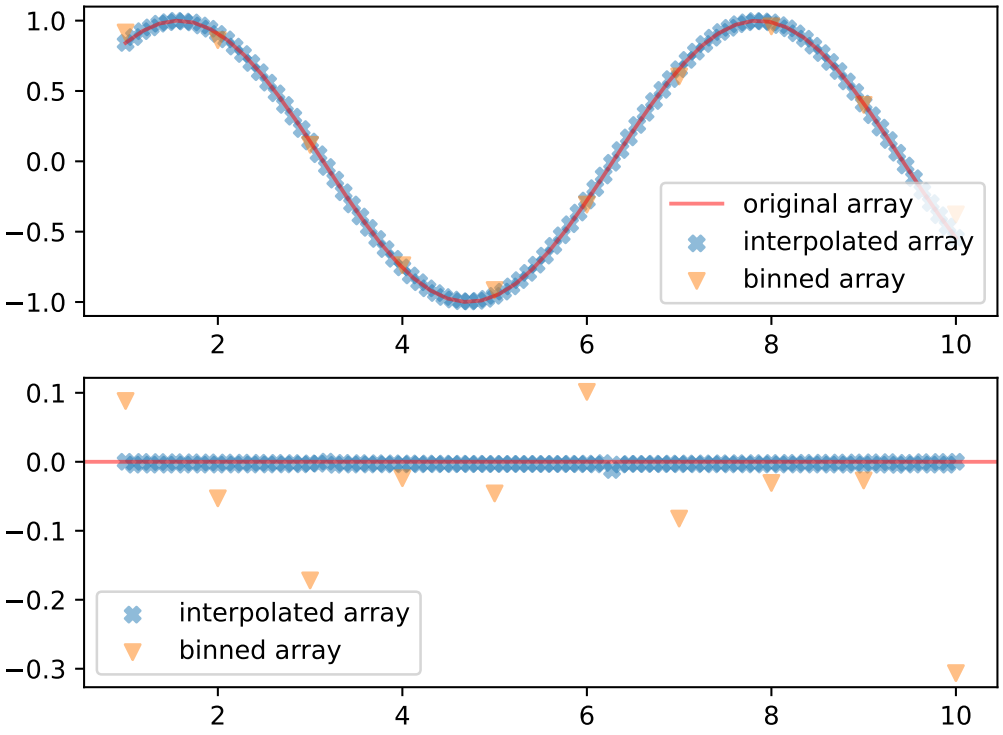
```
>>> import matplotlib.pyplot as plt
>>> fig, (ax0, ax1) = plt.subplots(2,1)
```

In the top panel we want to see the original function compared to the binned one and the interpolated one.

```
>>> ax0.plot(xp, fp, label='original array', alpha=0.5, c='r')
>>> ax0.scatter(x_inter, f_inter, marker='X', label='interpolated array',␣
↪alpha=0.5)
>>> ax0.scatter(x_bin, f_bin, marker='V', label='binned array', alpha=0.5)
>>> ax0.legend()
```

In the bottom panel we compare each function with the true value and we divide this quantity by the true value.

```
>>> ax1.axhline(0, c='r', alpha=0.5)
>>> ax1.scatter(x_inter, (f_inter- np.sin(x_inter))/np.sin(x_inter),
...             marker='X', label='interpolated array', alpha=0.5)
>>> ax1.scatter(x_bin, (f_bin - np.sin(x_bin))/np.sin(x_bin),
...             marker='v', label='binned array', alpha=0.5)
>>> ax1.legend()
>>> plt.show()
```



It's important to notice here that when we perform the binning, we are not simply resampling the input function, but we are using the mean value inside each of the bins.

logger

[693] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[694] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[695] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[696] https://docs.python.org/dev/library/functions.html#int
[697] https://docs.python.org/dev/library/stdtypes.html#str
[698] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.binned_statistic.html#scipy.stats.binned_statistic
[699] https://docs.python.org/dev/library/functions.html#float
[700] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[701] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

exosim.utils.checks

Module Contents

Functions

check_units(input_data, desired_units[, ...])	It checks the units of the inputs and returns the quantity rescaled to the desired units.
find_key(input_class_keys, key_list[, calling_class])	It finds the which key from a list is contained in an input class.
look_for_key(input_dict, key, value[, foo])	Returns `True` if a certain key is in the dictionary and has a certain value.

check_units(*input_data*, *desired_units*, *calling_class=None*, *force=False*)

It checks the units of the inputs and returns the quantity rescaled to the desired units.

Parameters

- **input_data** (ndarray[702] or Quantity[703]) – input class to rescale.
- **desired_units** (Quantity[704]) – desired unit for the input qquantity.
- **calling_class** (*Logger* (optional)) – calling class. This is needed to print the eventual debug message inside the calling class.
- **force** (*bool*[705] *(optional)*) – if True, if the input data has no units, it assumes is expressed in the desired units. Default is False.

Returns

scaled input quantity.

Return type

Quantity[706]

Raises

UnitConversionError – if it cannot convert the original units into the desired ones

find_key(*input_class_keys*, *key_list*, *calling_class=None*)

It finds the which key from a list is contained in an input class.

Parameters

- **input_class_keys** (*list*[707]) – list of the input class keys.
- **key_list:** – list: list of the desired keys.
- **calling_class** (*Logger* (optional)) – calling class. This is needed to print the eventual debug message inside the calling class.

Returns

found key

Return type

str[708]

[702] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[703] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[704] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[705] https://docs.python.org/dev/library/functions.html#bool
[706] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

Raises
KeyError[709] – if no useful key is found.

look_for_key(*input_dict*, *key*, *value*, *foo=False*)

Returns True if a certain key is in the dictionary and has a certain value.

Parameters

- **input_dict** (*dict*[710]) – input dictionary
- **key** (*str*[711]) – key to search
- **value** – key content to check

Return type
bool[712]

exosim.utils.convolution

Module Contents

Functions

fast_convolution(im, delta_im, ker, delta_ker)	fast_convolution.

fast_convolution(*im*, *delta_im*, *ker*, *delta_ker*)

fast_convolution. Convolve an image with a kernel. Image and kernel can be sampled on different grids defined.

Parameters

- **im** (ndarray[713]) – the image to be convolved
- **delta_im** (*float*[714]) – image sampling interval
- **ker** (ndarray[715]) – the convolution kernel
- **delta_ker** (*float*[716]) – Kernel sampling interval

Returns
the image convolved with the kernel.

Return type
ndarray[717]

[707] https://docs.python.org/dev/library/stdtypes.html#list
[708] https://docs.python.org/dev/library/stdtypes.html#str
[709] https://docs.python.org/dev/library/exceptions.html#KeyError
[710] https://docs.python.org/dev/library/stdtypes.html#dict
[711] https://docs.python.org/dev/library/stdtypes.html#str
[712] https://docs.python.org/dev/library/functions.html#bool
[713] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[714] https://docs.python.org/dev/library/functions.html#float
[715] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray
[716] https://docs.python.org/dev/library/functions.html#float
[717] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray

exosim.utils.focal_plane_locations

Module Contents

Functions

locate_wavelength_windows(psf, focal_plane, parameters)	

Attributes

logger

logger

locate_wavelength_windows(*psf*, *focal_plane*, *parameters*)

Parameters

- **psf** (*numpy.array*) –
- **focal_plane** (exosim.models.signal.Signal) –
- **parameters** (*dict*[718]) –

Return type
Tuple[numpy.array, numpy.array]

exosim.utils.grids

Module Contents

Functions

wl_grid(wl_min, wl_max, R[, return_bin_width])	It returns the wavelength log-grid in microns
time_grid(time_min, time_max, low_frequencies_resolution)	It returns the time grid in hours

wl_grid(*wl_min*, *wl_max*, *R*, *return_bin_width=False*)

It returns the wavelength log-grid in microns

The wavelength at the center of the spectral bins is defined as

$$\lambda_c = \frac{1}{2}(\lambda_j + \lambda_{j+1})$$

[718] https://docs.python.org/dev/library/stdtypes.html#dict

where λ_j is the wavelength at the bin edge defined by the recursive relation, and R is the *logbin_resolution* defined by the user.

$$\lambda_{j+1} = \lambda_j \left(1 + \frac{1}{R}\right)$$

And, given the maximum and minimum wavelengths, provided by the user, the number of bins is

$$n_{bins} = \frac{\log\left(\frac{\lambda_{max}}{\lambda_{min}}\right)}{\log\left(1 + \frac{1}{R}\right)} + 1$$

Parameters

- **wl_min** (Quantity[719] or float.) – minimum wavelength sampled. If no units are attached is considered as expressed in *um*
- **wl_max** (Quantity[720] or float.) – maximum wavelength sampled. If no units are attached is considered as expressed in *um*
- **R** (*int*[721]) – spectral resolving power
- **return_bin_width** (*bool*[722]) – if True returns also the bin width. Default is False.

Returns

- Quantity[723] – wavelength grid
- Quantity[724] – bin wavelength width

time_grid(*time_min*, *time_max*, *low_frequencies_resolution*)

It returns the time grid in hours

Parameters

- **time_min** (Quantity[725] or float.) – minimum time sampled. If no units are attached is considered as expressed in hours.
- **time_max** (Quantity[726] or float.) – maximum time sampled. If no units are attached is considered as expressed in hours.
- **low_frequencies_resolution** (Quantity[727] or float.) – time sampling interval. If no units are attached is considered as expressed in hours.

Returns

time grid

Return type

Quantity[728]

[719] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[720] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[721] https://docs.python.org/dev/library/functions.html#int
[722] https://docs.python.org/dev/library/functions.html#bool
[723] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[724] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[725] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[726] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[727] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[728] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity

exosim.utils.iterators

Module Contents

Functions

iterate_over_opticalElements(input, key, last_key, val)	It iterates over the optical element of a given class and returns the edited dictionary
iterate_over_chunks(dataset, **kwargs)	Iterates over the dataset chunks using tqdm to produce an adaptive progress bass
searchsorted(known_array, test_array)	

iterate_over_opticalElements(*input*, *key*, *last_key*, *val*)

It iterates over the optical element of a given class and returns the edited dictionary

Parameters

- **input** (*dict*[729]) – input dictionaru
- **key** – optical element class
- **last_key** – optical element key to edit
- **val** – value to edit

Return type
dict[730]

iterate_over_chunks(*dataset*, ***kwargs*)

Iterates over the dataset chunks using tqdm to produce an adaptive progress bass :param dataset: h5py chunked dataset used to store the data :type dataset: h5py.Dataset[731]

Return type
tqdm.tqdm

searchsorted(*known_array*, *test_array*)

exosim.utils.klass_factory

Module Contents

[729] https://docs.python.org/dev/library/stdtypes.html#dict
[730] https://docs.python.org/dev/library/stdtypes.html#dict
[731] https://docs.h5py.org/en/latest/high/dataset.html#h5py.Dataset

Functions

find_klass_in_file(python_file, baseclass)	It finds in the indicated python file a class that is a sublcass of the given one.
load_klass(input, baseclass)	It returns a class that is a subclass of the given base class.
find_task(input, baseclass)	It looks for a class that is a subclass of the base class indicated.
find_and_run_task(parameters, key, baseclass)	It looks in the input parameters for a class that is a subclass of the base class indicated, and it initialises it.

find_klass_in_file(*python_file*, *baseclass*)

It finds in the indicated python file a class that is a sublcass of the given one.

Parameters

- **python_file** (*str*[732]) – python file name
- **baseclass** (*class*) – reference class to search for

Returns
class found in the python file.

Return type
class

load_klass(*input*, *baseclass*)

It returns a class that is a subclass of the given base class.

Parameters

- **input** (*str*[733] *or class*) – if is a string, *find_klass_in_file* is used to return the right class. If is a class, it checks whether it is an eligible class or not.
- **baseclass** (*class*) – reference class to search for

Returns
subclass of baseclass

Return type
class

find_task(*input*, *baseclass*)

It looks for a class that is a subclass of the base class indicated.

Parameters

- **input** (*str*[734] *or object*[735]) – can either be a string indicating a class name, a python file, or it can be a class.
- **baseclass** (*object*[736]) – reference class

Return type
object[737]

[732] https://docs.python.org/dev/library/stdtypes.html#str
[733] https://docs.python.org/dev/library/stdtypes.html#str

find_and_run_task(*parameters*, *key*, *baseclass*)

It looks in the input parameters for a class that is a subclass of the base class indicated, and it initialises it.

Parameters

- **parameters** (*dict*[738]) – input dictionaty
- **key** (*str*[739]) – string indicating the keyword for the class name
- **baseclass** (*object*[740]) – reference class

Return type
callable

exosim.utils.operations

Module Contents

Functions

operate_over_axis(matrix, vector, axis[, operation])	This function is used to perform an operation over a given axis of a matrix.

Attributes

ops

ops

operate_over_axis(*matrix*, *vector*, *axis*, *operation='*'*)

This function is used to perform an operation over a given axis of a matrix.

Parameters

- **matrix** (*np.ndarray*) – N dimensional array
- **vector** (*np.ndarray*) – 1 dimensional array
- **axis** (*int*[741]) – axis to operate over
- **operation** (*str*[742], *optional*) – operator symbol, by default "*"

Returns
operated matrix

Return type
np.ndarray

[734] https://docs.python.org/dev/library/stdtypes.html#str
[735] https://docs.python.org/dev/library/functions.html#object
[736] https://docs.python.org/dev/library/functions.html#object
[737] https://docs.python.org/dev/library/functions.html#object
[738] https://docs.python.org/dev/library/stdtypes.html#dict
[739] https://docs.python.org/dev/library/stdtypes.html#str
[740] https://docs.python.org/dev/library/functions.html#object

exosim.utils.prepare_recipes

Module Contents

Functions

load_options(options_file)	It loads the configuration files into dictionaries.
copy_input_files(output_dir)	It copied the input configuration xml file to the output folder, if they are not there already.

Attributes

logger

load_options(*options_file*)

It loads the configuration files into dictionaries.

Parameters

options_file (*str*[743] *or* *dict*[744]) – configuration data to load

Returns

- *dict* – main configuration dictionary
- *dict* – payload configuration dictionary

copy_input_files(*output_dir*)

It copied the input configuration xml file to the output folder, if they are not there already.

Parameters

output_dir (*str*[745]) – output folder

logger

exosim.utils.psf

Module Contents

Functions

create_psf(wl, fnum, delta[, nzero, shape, ...])	Calculates an Airy Point Spread Function arranged as a data-cube.

[741] https://docs.python.org/dev/library/functions.html#int
[742] https://docs.python.org/dev/library/stdtypes.html#str
[743] https://docs.python.org/dev/library/stdtypes.html#str
[744] https://docs.python.org/dev/library/stdtypes.html#dict
[745] https://docs.python.org/dev/library/stdtypes.html#str

Attributes

logger

logger

create_psf(*wl*, *fnum*, *delta*, *nzero=4*, *shape='airy'*, *max_array_size=None*, *array_size=None*)

Calculates an Airy Point Spread Function arranged as a data-cube. The spatial axes are 0 and 1. The wavelength axis is 2. Each PSF volume is normalised to unity.

Parameters

- **wl** (Quantity[746]) – array of wavelengths at which to calculate the PSF
- **fnum** (*float*[747] *or* (*float*[748], *float*[749])) – Instrument f/number. It can be a tuple of two values, in which case the first value is the f/number for the x axis and the second value is the f/number for the y axis.
- **delta** (Quantity[750]) – the increment to use [physical unit of length]
- **nzero** (*float*[751]) – number of Airy zeros. The PSF kernel will be this big. Calculated at wl.max()
- **shape** (*str*[752] *(optional)*) – Set to 'airy' for a Airy function,to 'gauss' for a Gaussian
- **max_array_size** ((*int*[753],*int*[754]) *(optional)*) – Maximum size of the PSF array. If None, the size is calculated from the f/number and the wavelength range.
- **array_size** ((*int*[755] *or* *str*[756],*int*[757] *or* *str*[758]) *(optional)*) – Size of the PSF array. If 'full' then the *max_array_size* are used. If None, the size is calculated from the f/number and the wavelength range.

Returns

three-dimensional array. Each PSF normalised to unity

Return type

ndarray[759]

Examples

```
>>> import astropy.units as u
>>> from exosim.utils.psf import create_psf
```

We produce and plot an Airy PSF:

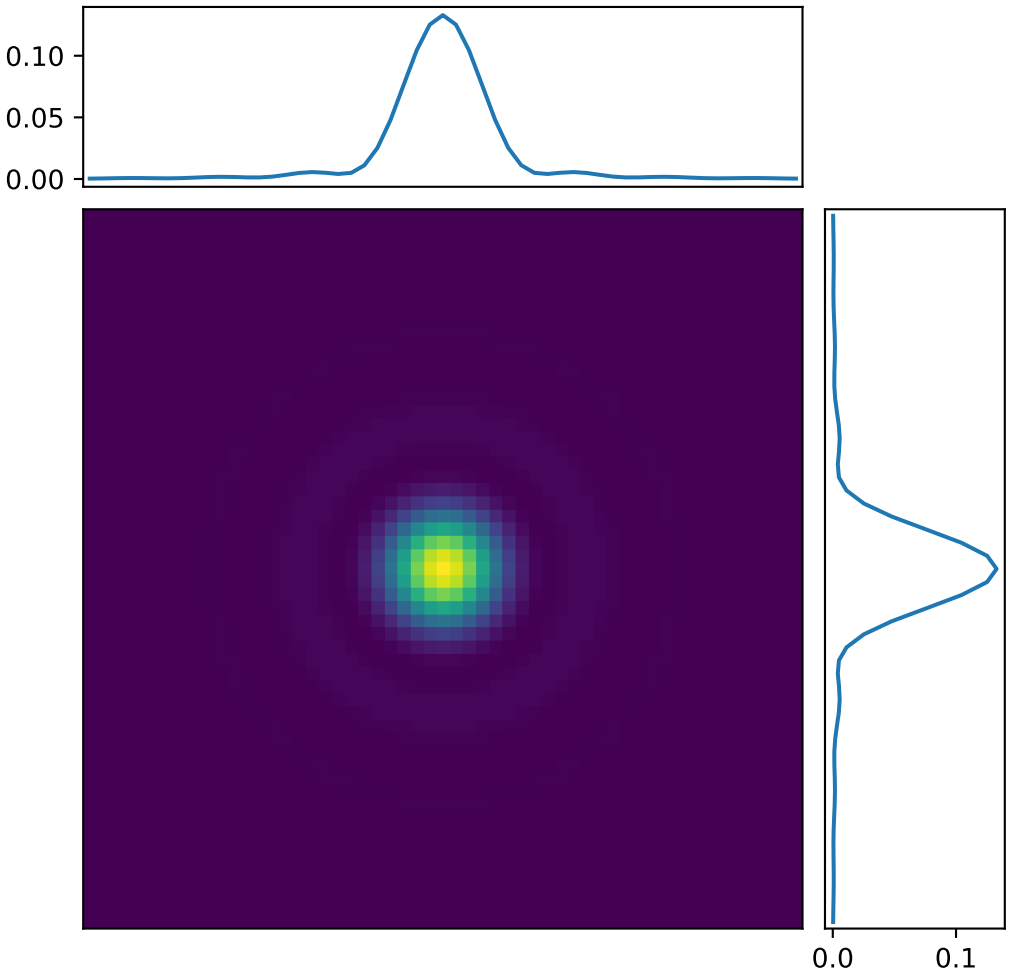
```
>>> img = create_psf(1*u.um, 40, 6*u.um, nzero=8, shape='airy')
```

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from matplotlib.gridspec import GridSpec
>>> fig = plt.figure(figsize=(6, 6))
>>> gs = fig.add_gridspec(2, 2,  width_ratios=(4, 1), height_ratios=(1, 4),
>>>                       left=0.1, right=0.9, bottom=0.1, top=0.9,
>>>                       wspace=0.05, hspace=0.05)
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_subplot(gs[1, 0])
>>> ax.imshow(img)
>>> ax_x = fig.add_subplot(gs[0, 0], sharex=ax)
>>> ax_y = fig.add_subplot(gs[1, 1], sharey=ax)
>>> axis_x = np.arange(0, img.shape[1])
>>> ax_x.plot(axis_x, img.sum(axis=0))
>>> ax_x.set_xticks([], [])
>>> axis_y = np.arange(0, img.shape[0])
>>> ax_y.plot(img.sum(axis=1), axis_y)
>>> ax_y.set_yticks([], [])
>>> plt.show()
```



Similarly, we can produce and plot a Gaussian PSF:

```
>>> img = create_psf(1*u.um, (40,40), 6*u.um, shape='gauss')
```

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from matplotlib.gridspec import GridSpec
>>> fig = plt.figure(figsize=(6, 6))
>>> gs = fig.add_gridspec(2, 2,  width_ratios=(4, 1), height_ratios=(1, 4),
>>>                       left=0.1, right=0.9, bottom=0.1, top=0.9,
>>>                       wspace=0.05, hspace=0.05)
>>> ax = fig.add_subplot(gs[1, 0])
>>> ax.imshow(img)
>>> ax_x = fig.add_subplot(gs[0, 0], sharex=ax)
>>> ax_y = fig.add_subplot(gs[1, 1], sharey=ax)
>>> axis_x = np.arange(0, img.shape[1])
>>> ax_x.plot(axis_x, img.sum(axis=0))
>>> ax_x.set_xticks([], [])
>>> axis_y = np.arange(0, img.shape[0])
>>> ax_y.plot(img.sum(axis=1), axis_y)
>>> ax_y.set_yticks([], [])
>>> plt.show()
```

We can also create a PSF with different F-numbers:

```
>>> img = create_psf(1*u.um, (60,40), 6*u.um, shape='gauss')
```

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(img, aspect='equal',)
>>> plt.show()
```

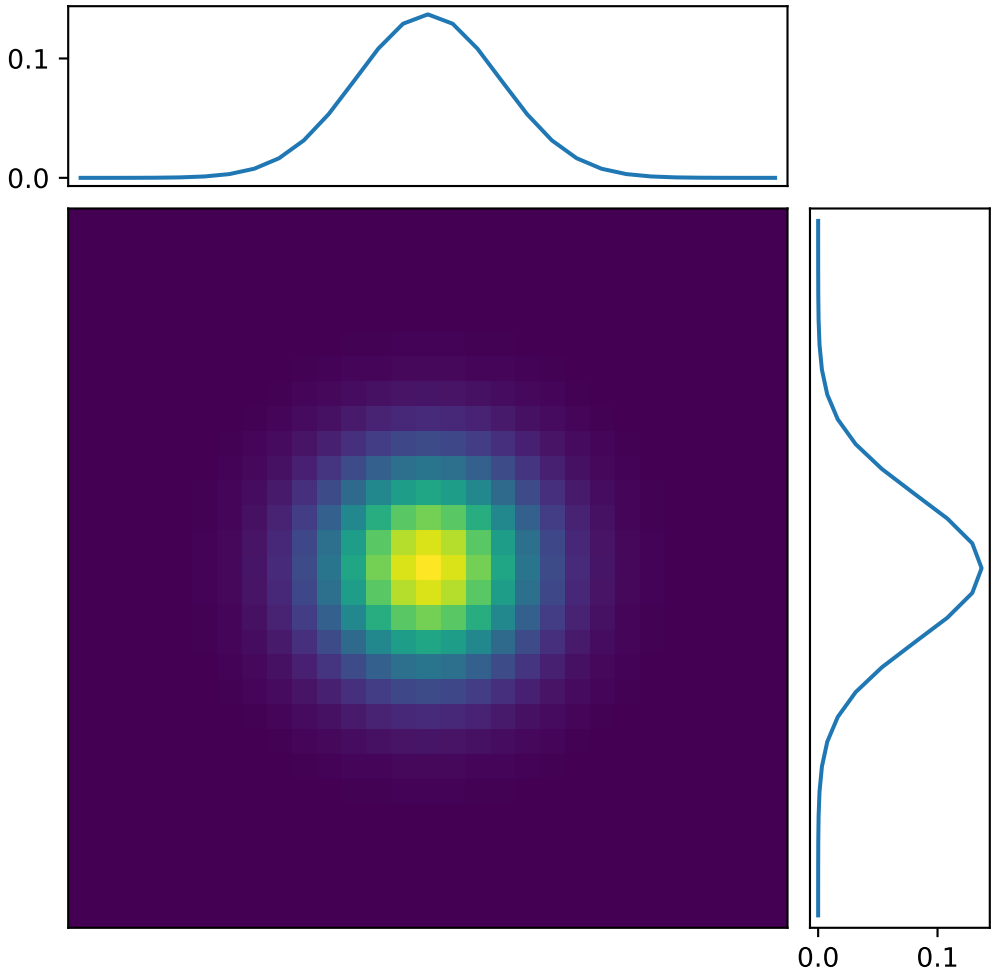
exosim.utils.runConfig

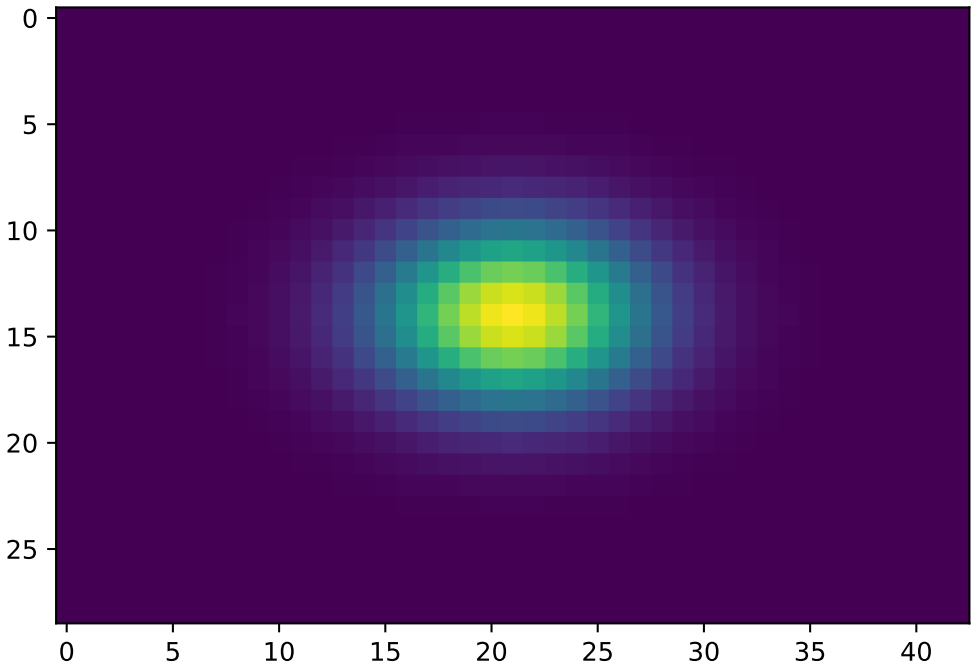
Module Contents

Classes

SingletonMeta	The Singleton class can be implemented in different ways in Python. Some
RunConfigInit	Class used to propagate values through the code.

[746] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[747] https://docs.python.org/dev/library/functions.html#float
[748] https://docs.python.org/dev/library/functions.html#float
[749] https://docs.python.org/dev/library/functions.html#float
[750] https://docs.astropy.org/en/latest/api/astropy.units.Quantity.html#astropy.units.Quantity
[751] https://docs.python.org/dev/library/functions.html#float
[752] https://docs.python.org/dev/library/stdtypes.html#str
[753] https://docs.python.org/dev/library/functions.html#int
[754] https://docs.python.org/dev/library/functions.html#int
[755] https://docs.python.org/dev/library/functions.html#int
[756] https://docs.python.org/dev/library/stdtypes.html#str
[757] https://docs.python.org/dev/library/functions.html#int
[758] https://docs.python.org/dev/library/stdtypes.html#str
[759] https://numpy.org/devdocs/reference/generated/numpy.ndarray.html#numpy.ndarray


0.1
0.0
0.0
0.1


0
5
10
15
20
25
0
5
10
15
20
25
30
35
40

Attributes

total_cpus	
RunConfig	

total_cpus

class SingletonMeta

Bases: type[760]

The Singleton class can be implemented in different ways in Python. Some possible methods include: base class, decorator, metaclass. We will use the metaclass because it is best suited for this purpose.

class RunConfigInit

Bases: exosim.log.Logger

Class used to propagate values through the code.

property random_seed

property n_job

property random_generator

Returns a random generator with the current random seed updated by one if the random seed is set

chunk_size = 2

config_file_list = []

random_seed()

stats(*log=True*)

It returns a dictionary with the updated run configurations.

Parameters

log (*bool*[761] *(optional)*) – if True, it prints the run configuration stats. Default is True.

RunConfig

exosim.utils.timed_class

Module Contents

Classes

TimedClass	This class adds methods to log the elapsed time

[760] https://docs.python.org/dev/library/functions.html#type
[761] https://docs.python.org/dev/library/functions.html#bool

Attributes

logger

class TimedClass

This class adds methods to log the elapsed time

log_runtime(*message*, *level='info'*)

log_runtime_complete(*message*, *level='info'*)

logger

exosim.utils.types

Module Contents

ArrayType

ValueType

UnitType

HDF5OutputType

OutputType

7.1.2 Submodules

exosim.exosim

Module Contents

Functions

help()
focalplane()
radiometric()
subexposures()
ndrs()

Attributes

logger
code_name_and_version
configuration_flags
configuration
output_file_flags
output_file
input_file_flags
input_file
n_threads_flags
n_threads
debug_conf_flags
debug_conf
logger_conf_flags
logger_conf
plot_options_flags
plot_options
chunk_options_flags
chunk_options

logger

code_name_and_version

configuration_flags = ['-c', '--configuration']

configuration

output_file_flags = ['-o', '--output']

output_file

input_file_flags = ['-i', '--input']

input_file

n_threads_flags = ['--nThreads']

n_threads

debug_conf_flags = ['-d', '--debug']

debug_conf

logger_conf_flags = ['-l', '--logger']

logger_conf

plot_options_flags = ['-P', '--plot']

plot_options

chunk_options_flags = ['--chunk_size']

chunk_options

help()

focalplane()

radiometric()

subexposures()

ndrs()

7.1.3 Package Contents

git_folder

logger

Warning: This documentation is not completed yet. If you find any issue or difficulty, please contact the developers for help.

CHAPTER

EIGHT

CITE

A dedicated publication has been submitted and the relative information will be published soon. In the meantime, please, send an email to the developers.

CHAPTER

NINE

ACKNOWLEDGMENTS

ExoSim 2 has been developed under the umbrella of Ariel Space Mission[764], with the support of the Ariel Consortium and the members of Simulator Software, Management and Documentation (S2MD) Working Group.

During the development of the first alpha and beta versions of this software, L. V. Mugnai[765] was affiliated to Sapienza University of Rome[766] and supported by ASI[767].

We thank Ahmed Al-Refaie for his support during the development and the inspiration provided by his code: TauREx3[768] .

[764] https://arielmission.space/
[765] https://www.lorenzomugnai.com/
[766] https://www.phys.uniroma1.it/fisica/en-welcome
[767] https://www.asi.it/en/
[768] https://arxiv.org/abs/1912.07759

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z