# PySpi Documentation

**Bjoern Biltzinger, Dr. J. Michael Burgess**
**Thomas Siegert**

**Mar 18, 2022**

# CONTENTS

PySPI is pure python interface to analyze Gamma-Ray Burst (GRB) data from the spectrometer (SPI) onboard the International Gamma-Ray Astrophysics Laboratory (INTEGRAL). The INTEGRAL satellite is a gamma-ray observatory hosting four instruments that operate in the energy range between 3 keV and 10 MeV. It was launched in 2002 and is still working today. The main goals of PySPI are to provide an easy to install and develop analysis software for SPI, which includes improvements on the statistical analysis of GRB data. At the moment PySPI is designed for transient sources, like Gamma Ray Bursts (GRBs). In the future we plan to add support for other types of sources, such as persistent point sources as well as extended emission.

CHAPTER

ONE

# COMPARISON TO OSA

The main analysis tool to analyze SPI data up to now is the "Off-line Scientific Analysis" (OSA) Chernyakova et al., 2020), which is maintained by the INTEGRAL Science Data Centre (ISDC). While it is comprehensive in its capabilities for manipulating data obtained from all instrument on-board INTEGRAL, it exists as an IDL interface to a variety of low-level C++ libraries and is very difficult to install on modern computers. While there are containerized versions of OSA now available, the modern workflow of simply installing the software from a package manager and running on a local workstation is not possible and often students rely on a centralized installation which must be maintained by a seasoned expert. Moreover, adding more sophisticated and/or correct data analysis methods to the software requires an expertise that is not immediately accessible to junior researchers or non-experts in the installation of OSA. Also due to the increased computational power that is available today compared to that of 20 years ago, many of the analysis methods can be improved. PySPI addresses both these problems: It is providing an easy to install software, that can be developed further by everyone who wants to contribute. It also allows Bayesian fits of the data with true forward folding of the physical spectra into the data space via the response. This improves the sensitivity and the scientific output of GRB analyses with INTEGRAL/SPI.

CHAPTER

TWO

# MULTI MISSION ANALYSIS

PySPI provides a plugin for 3ML. This makes multi missions analysis with other instruments possible. Also all the spectral models from astromodels are available for the fits. Check out these two software packages for more information.

## 2.1 Installation

### 2.1.1 Pip

To install PySPI via pip just use

```
pip install py-spi
```

### 2.1.2 Conda/Mamba

If you have problems installing PySPI within a Conda environment try to create your environment with this command

```
conda create -n pyspi -c conda-forge python=3.9 numpy scipy ipython numba astropy␣
↪matplotlib h5py pandas pytables
```

or for Mamba

```
mamba create -n pyspi -c conda-forge python=3.9 numpy scipy ipython numba astropy␣
↪matplotlib h5py pandas pytables
```

and then run

```
pip install py-spi
```

with the environment activated.

### 2.1.3 Github

To install the latest release from Github run

```
git clone https://github.com/BjoernBiltzinger/pyspi.git
```

After that first install the packages from the requirement.txt file with

```
cd pyspi
pip install -r requirements.txt
```

Now you can install PySPI with

```
python setup.py install
```

### 2.1.4 Additional Data Files

There are a few large data files for the background model and the response that are not included in the Github repository. To get these data files run the following commands. Here the data folder is downloaded and is moved to a user defined path where this data folder should be stored on your local machine. Here you have to change the /path/to/internal/data to the path you want to use on your local computer. This only needs to be downloaded once and will not change afterwards.

```
wget https://grb.mpe.mpg.de/pyspi_datafolder && unzip pyspi_datafolder
mv data /path/to/internal/data && rm -f pyspi_datafolder
```

### 2.1.5 Environment Variables

Next you have to set two environment variable. One to define the path to the folder of the external data like the different SPI data files that will be downloaded by PySPI and one to define the path to the internal data folder we downloaded earlier.

```
export PYSPI=/path/to/external/datafolder
export PYSPI_PACKAGE_DATA=/path/to/internal/data
```

Here /path/to/external/datafolder is the path to a folder on your local machine, where PySPI should save all the downloaded data needed for the analysis. The data that will be saved into this folder are the SPI data files as well as one housekeeping data file of SPI and one housekeeping data file of INTEGRAL per analyzed GRB. In total this adds up to roughly 30-70 MB per analyzed GRB. It is not recommended to use the same path for both environment variables.

You should also add these two line to your bashrc (or similar) file to automatically set these variables in every new terminal.

Now we are ready to go.

### 2.1.6 Run Unit Test Locally

PySPI includes unit test to check that non of its functionality break in new versions. These run automatically for every push on GitHub via GitHub Actions. But you can also run the tests locally. To run the test you need to install pytest and pytest-cov.

```
pip install pytest pytest-cov
```

After this run

```
pytest -v
```

in the top level directory.

## 2.2 Light Curves

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

Gamma-Ray Bursts are transient sources with a typical duration between milliseconds and a few tens of seconds. Therefore they are nicely visible in light curves. In the following we will see how we can get the light curve of a real GRB as seen by an INTEGRAL/SPI detector.

First we have to define the rough time of the GRB.

[2]:
```
from astropy.time import Time
grbtime = Time("2012-07-11T02:44:53", format='isot', scale='utc')
```

Next we need to define the bounds of the energy bins we want to use.

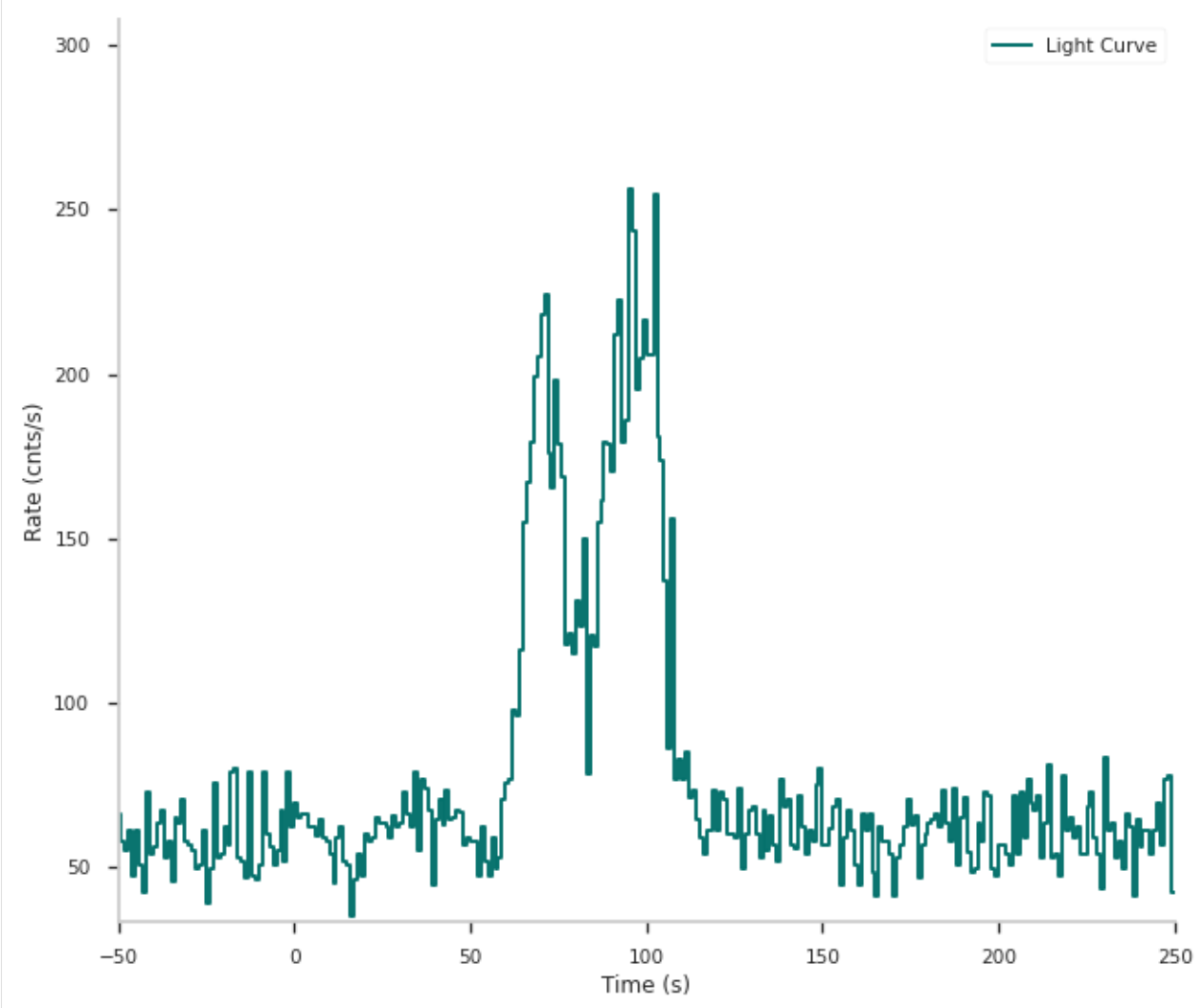
[3]:
```
import numpy as np
ebounds = np.geomspace(20,8000,100)
```

Now we can construct the time series.

[4]:
```
from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
det = 0
tsb = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
                                        det,
                                        grbtime,
                                        ebounds=ebounds,
                                        sgl_type="both",
                                        )
```

We can now plot the light curves for visualization, in which we can clearly see a transient source in this case.

[5]:
```
fig = tsb.view_lightcurve(-50,250)
```




## 2.3 Response

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

For every analysis of SPI data we need the correct response for the observation, which is the connection between physical spectra and detected counts. Normally the response is a function of the position of the source in the satellite

frame, the input energies of the physical spectrum and the output energy bins of the experiment. For SPI, there is also a time dependency, because a few detectors failed during the mission time and this changed the response of the surrounding detectors.

In PySPI we construct the response from the official IRF and RMF files, which we interpolate for a given source position and user chosen input and output energy bins.

We start by defining a time, for which we want to construct the response, to get the pointing information of the satellite at this time and the version number of the response.

[2]:
```
from astropy.time import Time
rsp_time = Time("2012-07-11T02:42:00", format='isot', scale='utc')
```

Next we define the input and output energy bins for the response.

[3]:
```
import numpy as np
ein = np.geomspace(20,8000,1000)
ebounds = np.geomspace(20,8000,100)
```

Get the response version and construct the rsp base, which is an object holding all the information of the IRF and RMF for this response version. We use this, because if we want to combine many observations later, we don't want to read in this for every observation independently, because this would use a lot of memory. Therefore all the observations with the same response version can share this rsp_base object.

[4]:
```
from pyspi.utils.function_utils import find_response_version
from pyspi.utils.response.spi_response_data import ResponseDataRMF
version = find_response_version(rsp_time)
print(version)
rsp_base = ResponseDataRMF.from_version(version)
```

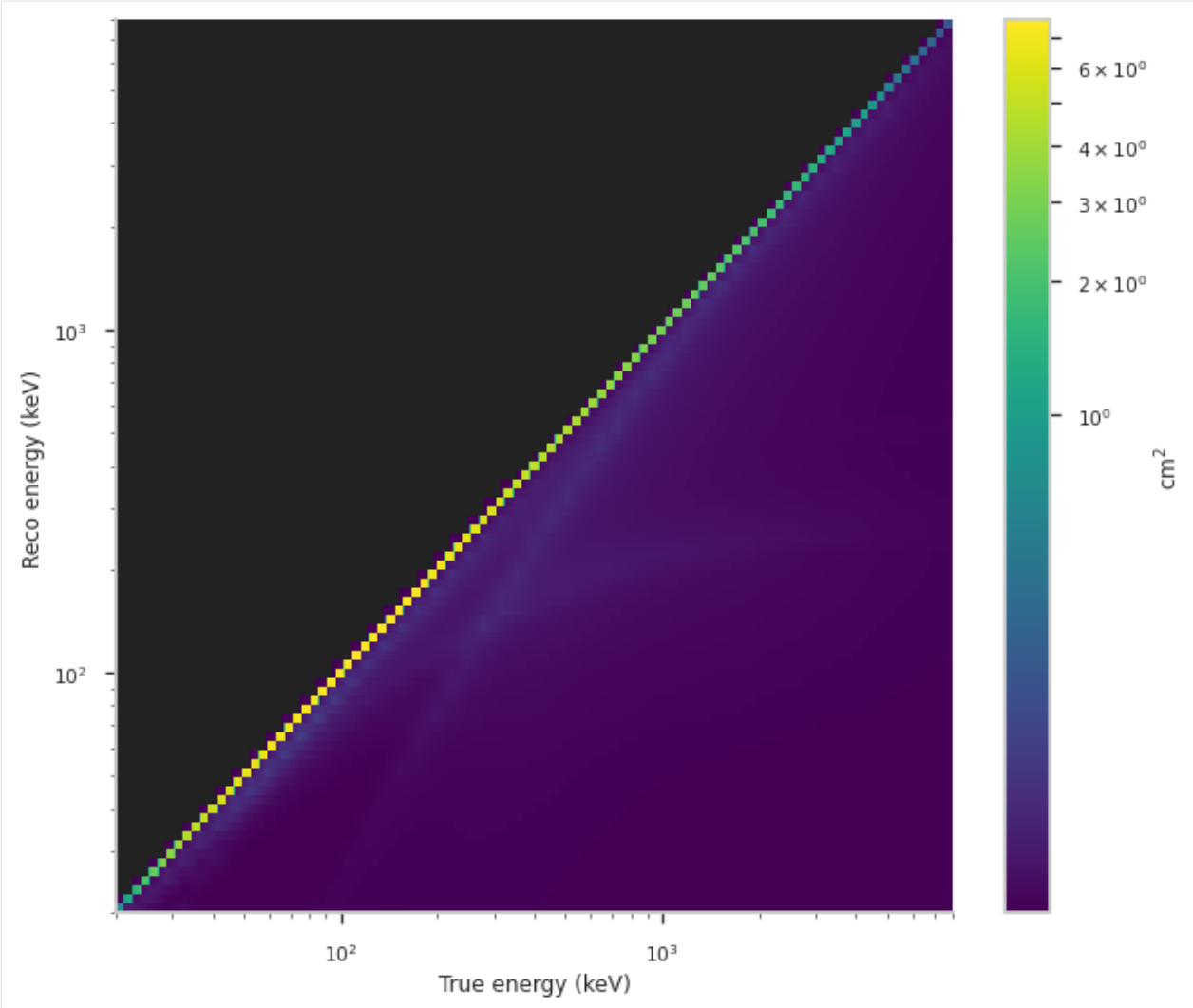
```
4
Using the irfs that are valid between 10/05/27 12:45:00 and present (YY/MM/DD HH:MM:SS)
```

Now we can construct the response for a given detector and source position (in ICRS coordinates)

[5]:
```
from pyspi.utils.response.spi_response import ResponseRMFGenerator
from pyspi.utils.response.spi_drm import SPIDRM
det = 0
ra = 94.6783
dec = -70.99905
drm_generator = ResponseRMFGenerator.from_time(rsp_time,
                                               det,
                                               ebounds,
                                               ein,
                                               rsp_base)
sd = SPIDRM(drm_generator, ra, dec)
```

SPIDRM is a child class of InstrumentResponse from threeML, therefore we can use the plotting functions from 3ML.

[6]:
```
fig = sd.plot_matrix()
```




## 2.4 Electronic Noise Region

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

Since shortly after the launch of INTEGRAL it is known that there are spurious events in the SPI data around ~1.5 MeV. A paper from Roques & Jourdain gives an explanation for this problem. Luckily this problem exists only in the events that only triggered the analog front-end electronics (AFEE). The events that trigger in addition the pulse shape

discrimination electronics (PSD) do not show this problem. According to Roques & Jourdain, one should therefore use the PSD events whenever this is possible, which is for events between ~500 and 2500 keV (the precise boundaries were changed during the mission a few times). In the following the events that trigger both the AFEE and PSD are called "PSD events" and the other normal "single events" or "Non-PSD events", even thought the PSD events are of course also single events.

To account for this problem in our analysis we can construct plugins for the "PSD events" and the for the "Non-PSD events" and use only the events with the correct flags, when we construct the time series.

Let's check the difference between the PSD and the Non-PSD events, to see the effect in real SPI data.

First we define the time and the energy bins we want to use. Then we construct the time series for the three cases:

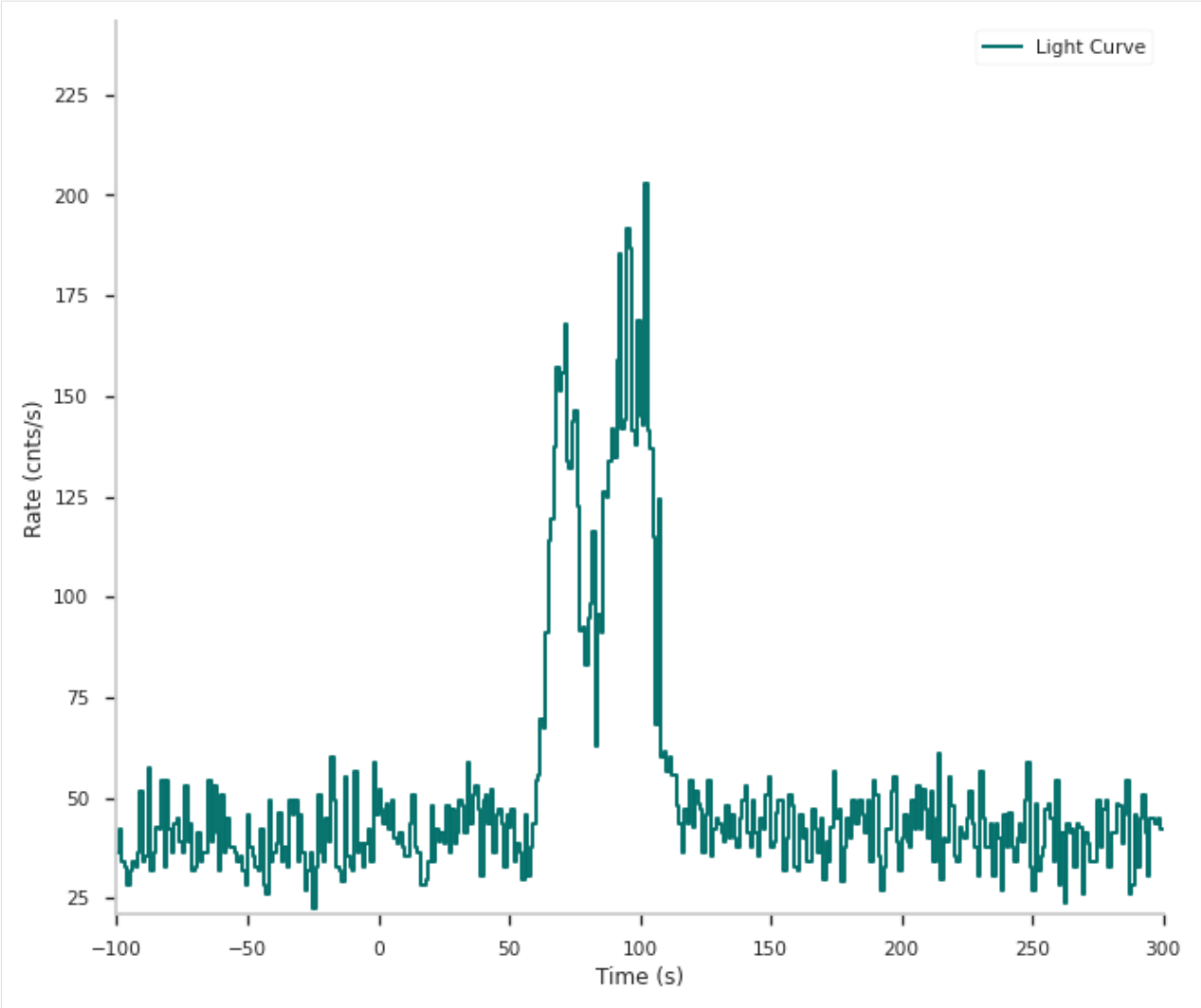
1. Only the events that trigger AFEE and not PSD
2. Only the events that trigger AFEE and PSD
3. All the single events

[2]:
```
from astropy.time import Time
import numpy as np
from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
grbtime = Time("2012-07-11T02:44:53", format='isot', scale='utc')
ebounds = np.geomspace(20,8000,300)
det = 0

from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
tsb_sgl = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
    det,
    grbtime,
    ebounds=ebounds,
    sgl_type="sgl",
    )

tsb_psd = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
    det,
    grbtime,
    ebounds=ebounds,
    sgl_type="psd",
    )

tsb_both = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
    det,
    grbtime,
    ebounds=ebounds,
    sgl_type="both",
    )
```

We can check the light curves for all three cases.

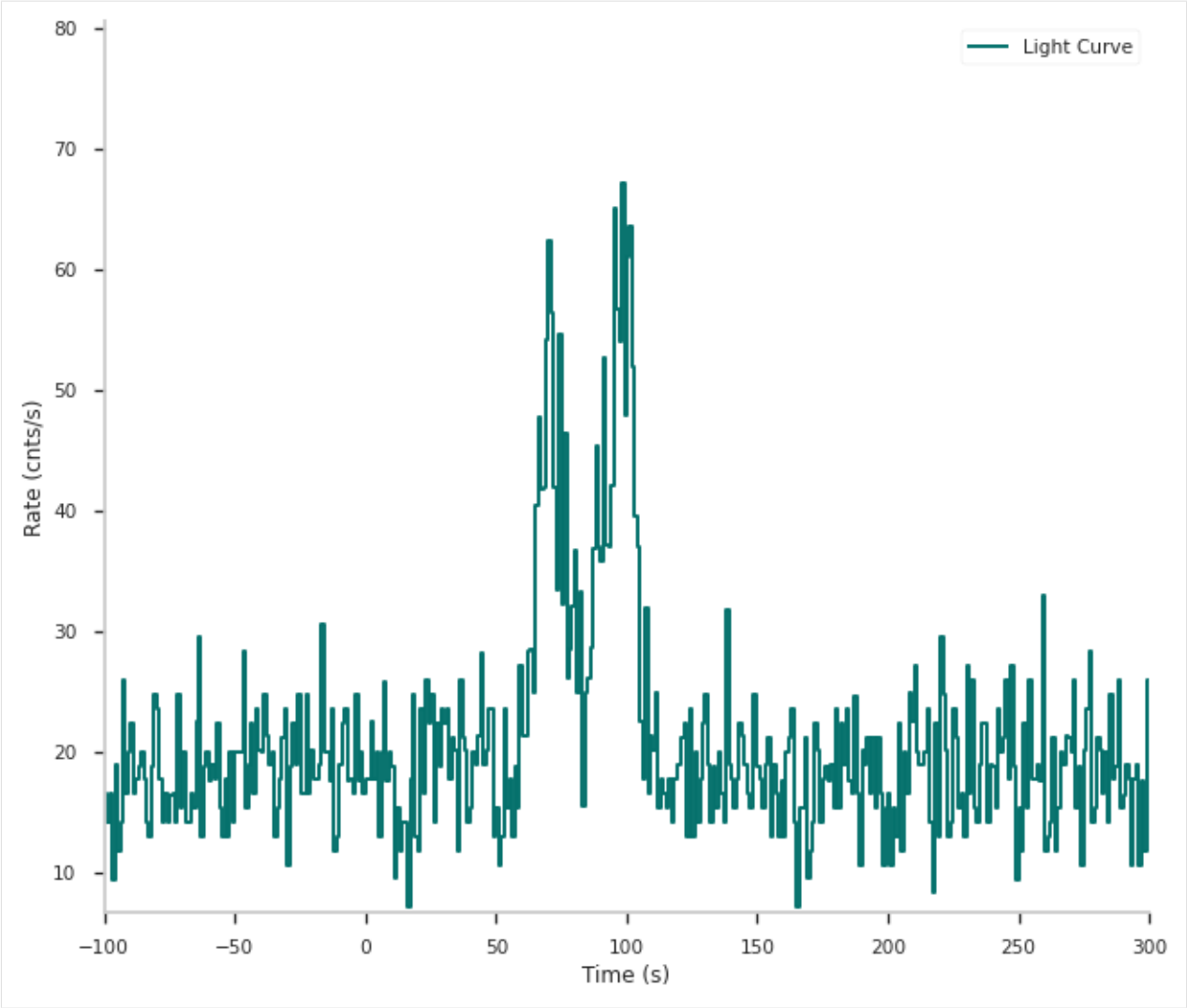
[3]:
```
print("Only AFEE:")
fig = tsb_sgl.view_lightcurve(-100,300)
```

```
Only AFEE:
```



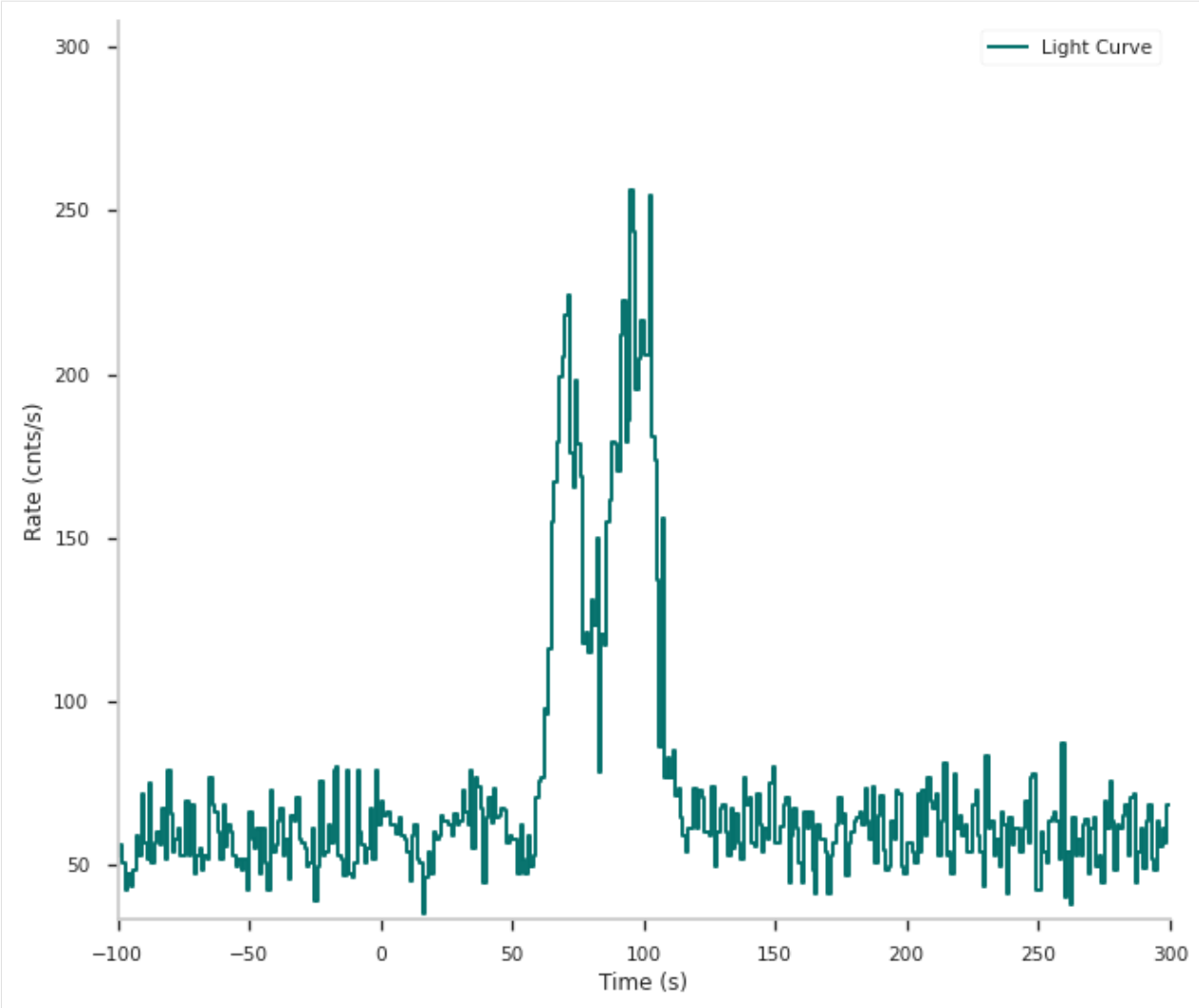

[4]:
```
print("AFFE and PSD trigger:")
fig = tsb_psd.view_lightcurve(-100,300)
```

```
AFFE and PSD trigger:
```




[5]:
```
print("Both Combined:")
fig = tsb_both.view_lightcurve(-100,300)
```

```
Both Combined:
```




We can see that the PSD event light curve has way less counts. This is due to the fact, that the PSD trigger only starts detecting photons with energies >~ 400 keV.

Next we can get the time integrated counts per energy channel.

[6]:
```
tstart = -500
tstop = 1000
counts_sgl = tsb_sgl.time_series.count_per_channel_over_interval(tstart, tstop)
counts_psd = tsb_psd.time_series.count_per_channel_over_interval(tstart, tstop)
counts_both = tsb_both.time_series.count_per_channel_over_interval(tstart, tstop)
```

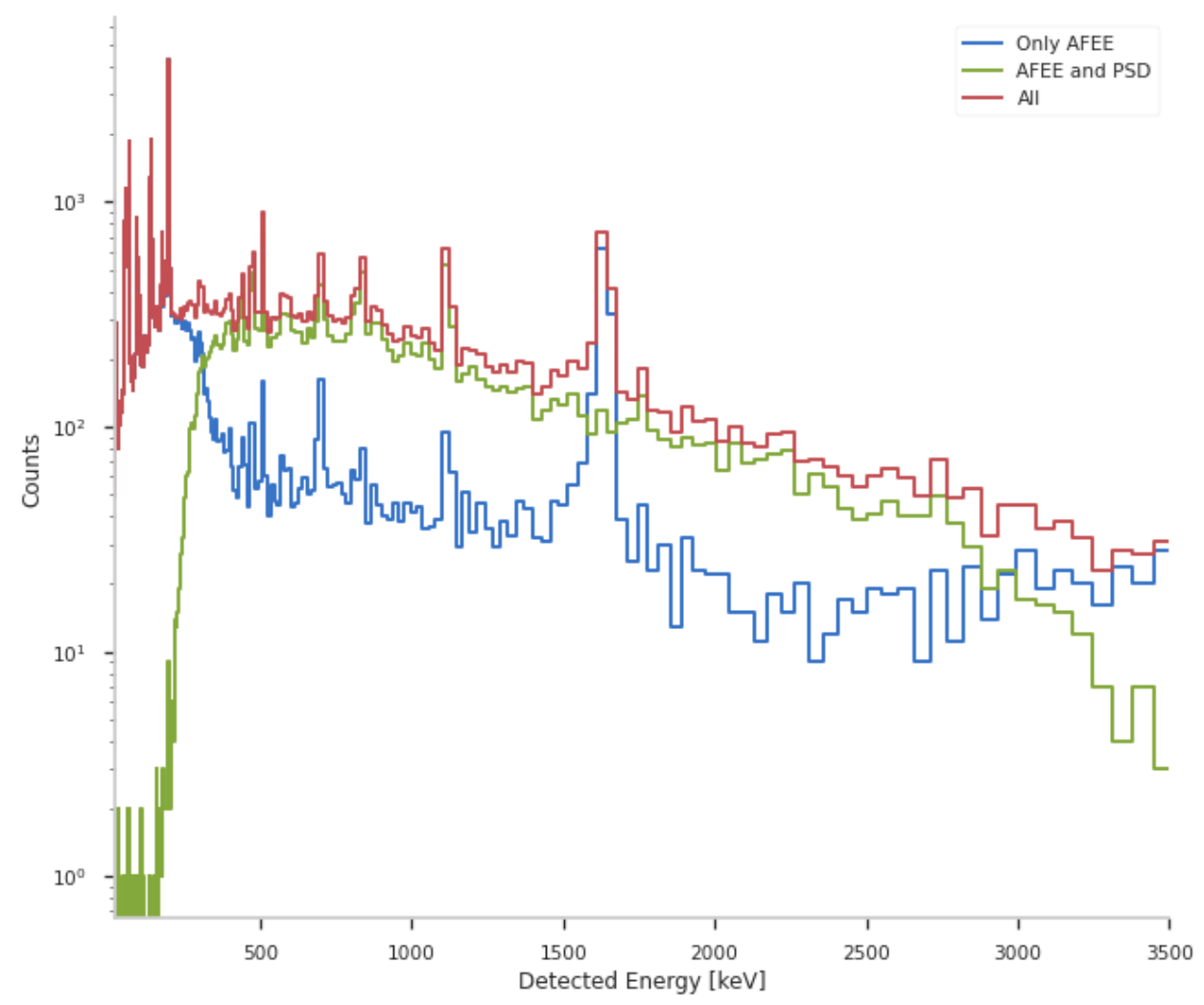
We can now plot the counts as a function of the energy channel energies

[7]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1,1)
ax.step(ebounds[1:], counts_sgl, label="Only AFEE")
ax.step(ebounds[1:], counts_psd, label="AFEE and PSD")
ax.step(ebounds[1:], counts_both, label="All")
ax.set_xlabel("Detected Energy [keV]")
ax.set_ylabel("Counts")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim(20,3500)
ax.set_yscale("log")
ax.legend();
```




Several features are visible.

1. A sharp cutoff at small energies for the PSD events, which is due to the low energy threshold in the PSD electronics.
2. For energies>~2700 keV the PSD events decrease again faster than the other events.
3. In the Non-PSD events we see a peak at ~ 1600 keV that is not visible in the PSD events. This is the so called electronic noise, which consists of spurious events.
4. The fraction of PSD events to all single events between ~500 and ~2700 keV is very stable and can be explained by an additional dead time for the PSD electronics.

## 2.5 Active Detectors

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

During the life of INTEGRAL/SPI several detectors stopped working correctly and were therefore disabled. In our analysis we need to take this into account, to not include a detector with 0 counts all the time and because the response for the surrounding detectors change when a detector is deactivated.

With PySPI you can calculate for a given time, which detectors are active and which response version is valid at that time.

[2]:
```
time_string = "051212 205010" #"YYMMDD HHMMSS"; astropy time object also possible
```

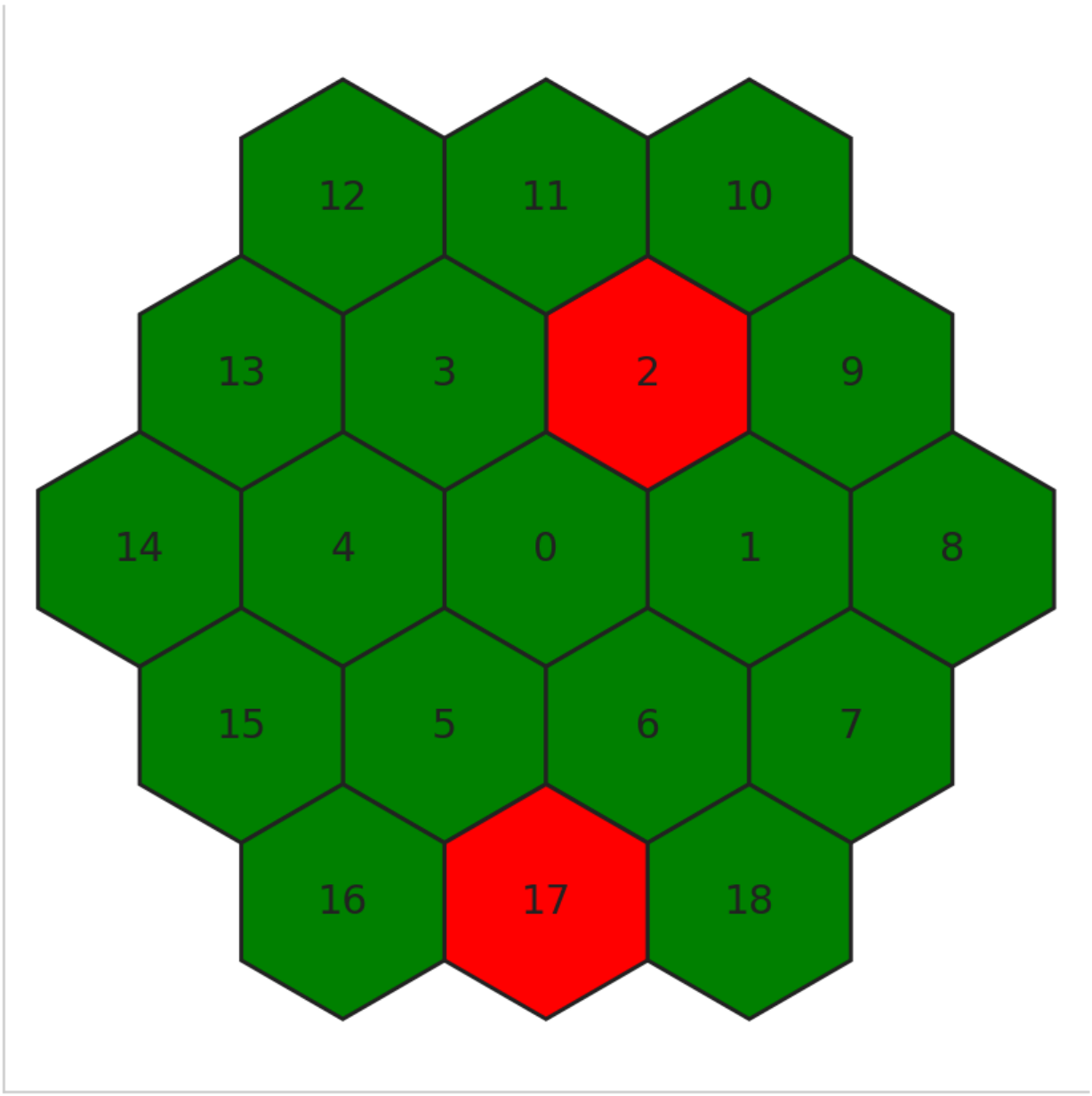
To get the active single detectors for this time use:

[3]:
```
from pyspi.utils.livedets import get_live_dets
ld = get_live_dets(time_string, event_types="single")
print(f"Active detectors: {ld}")
```

```
Active detectors: [ 0  1  3  4  5  6  7  8  9 10 11 12 13 14 15 16 18]
```

It is also possible to plot the same information visually. This shows the detector plane, and all the inactive detectors at the given time are colored red.

[4]:
```
from pyspi.io.plotting.spi_display import SPI
s = SPI(time=time_string)
fig = s.plot_spi_working_dets()
```




Also the response version at that time can be calculated.

[5]:
```
from pyspi.utils.function_utils import find_response_version
v = find_response_version(time_string)
print(f"Response version number: {v}")
```

```
Response version number: 2
```

## 2.6 Access the Underlying Data

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

Sometime you maybe want to access the underlying data of the analysis to do your own analysis or tests with this data. This section shows how to access some basic quantities, like for example the detected counts per energy channel and the response matrix. First we have to initialize the usual objects in PySPI.

[2]:
```
from astropy.time import Time
import numpy as np
from pyspi.utils.function_utils import find_response_version
from pyspi.utils.response.spi_response_data import ResponseDataRMF
from pyspi.utils.response.spi_response import ResponseRMFGenerator
from pyspi.utils.response.spi_drm import SPIDRM
from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
from pyspi.SPILike import SPILikeGRB

grbtime = Time("2012-07-11T02:44:53", format='isot', scale='utc')
ein = np.geomspace(20,800,300)
ebounds = np.geomspace(20,400,30)
version = find_response_version(grbtime)
rsp_base = ResponseDataRMF.from_version(version)
det=0
ra = 94.6783
dec = -70.99905
drm_generator = ResponseRMFGenerator.from_time(grbtime,
                                               det,
                                               ebounds,
                                               ein,
                                               rsp_base)
sd = SPIDRM(drm_generator, ra, dec)
tsb = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
    det,
    grbtime,
    response=sd,
    sgl_type="both",
    )
active_time = "65-75"
bkg_time1 = "-500--10"
bkg_time2 = "150-1000"
tsb.set_active_time_interval(active_time)
tsb.set_background_interval(bkg_time1, bkg_time2)
sl = tsb.to_spectrumlike()
```

(continues on next page)

(continued from previous page)

```
plugin = SPILikeGRB.from_spectrumlike(sl,free_position=False)
```

```
Using the irfs that are valid between 10/05/27 12:45:00 and present (YY/MM/DD HH:MM:SS)
```

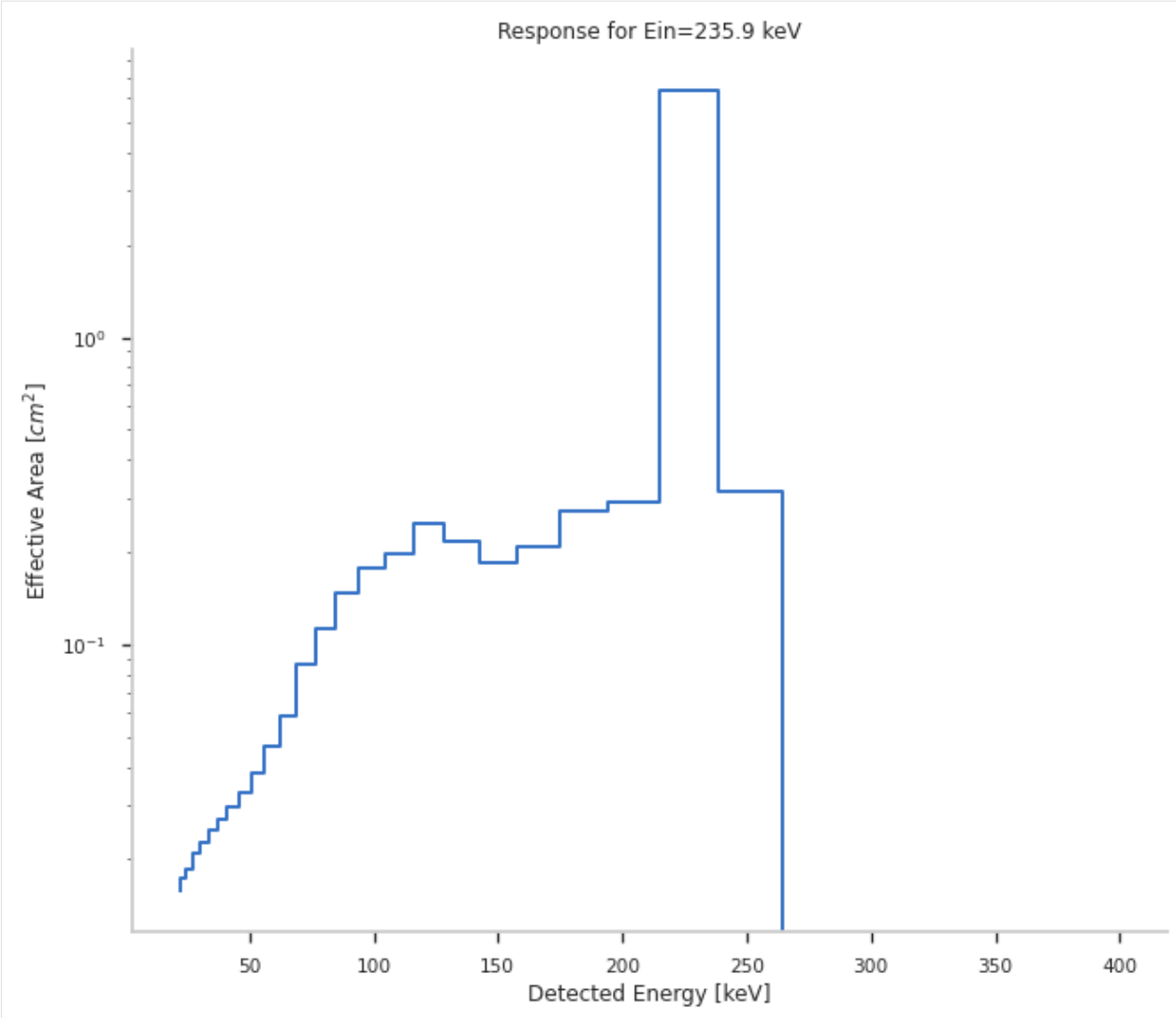
```
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
```

In the following it is listed how you can access some of the basic underlying data.

### 2.6.1 Response Matrix

Get response matrix and plot the response for one incoming energy.

[3]:
```
import matplotlib.pyplot as plt
ein_id = 200
matrix = sd.matrix

fig, ax = plt.subplots(1,1)
ax.step(ebounds[1:], matrix[:,ein_id])
ax.set_title(f"Response for Ein={round(ein[ein_id], 1)} keV")
ax.set_xlabel("Detected Energy [keV]")
ax.set_ylabel("Effective Area [$cm^2$]")
ax.set_yscale("log");
```




### 2.6.2 Event Data

The data is saved as time tagged events. You can access the arrival time and reconstructed energy bin of every photons. It is important to keep in mind that the reconstructed energy is not the true energy, it is just the energy assigned to one of the energy channels.

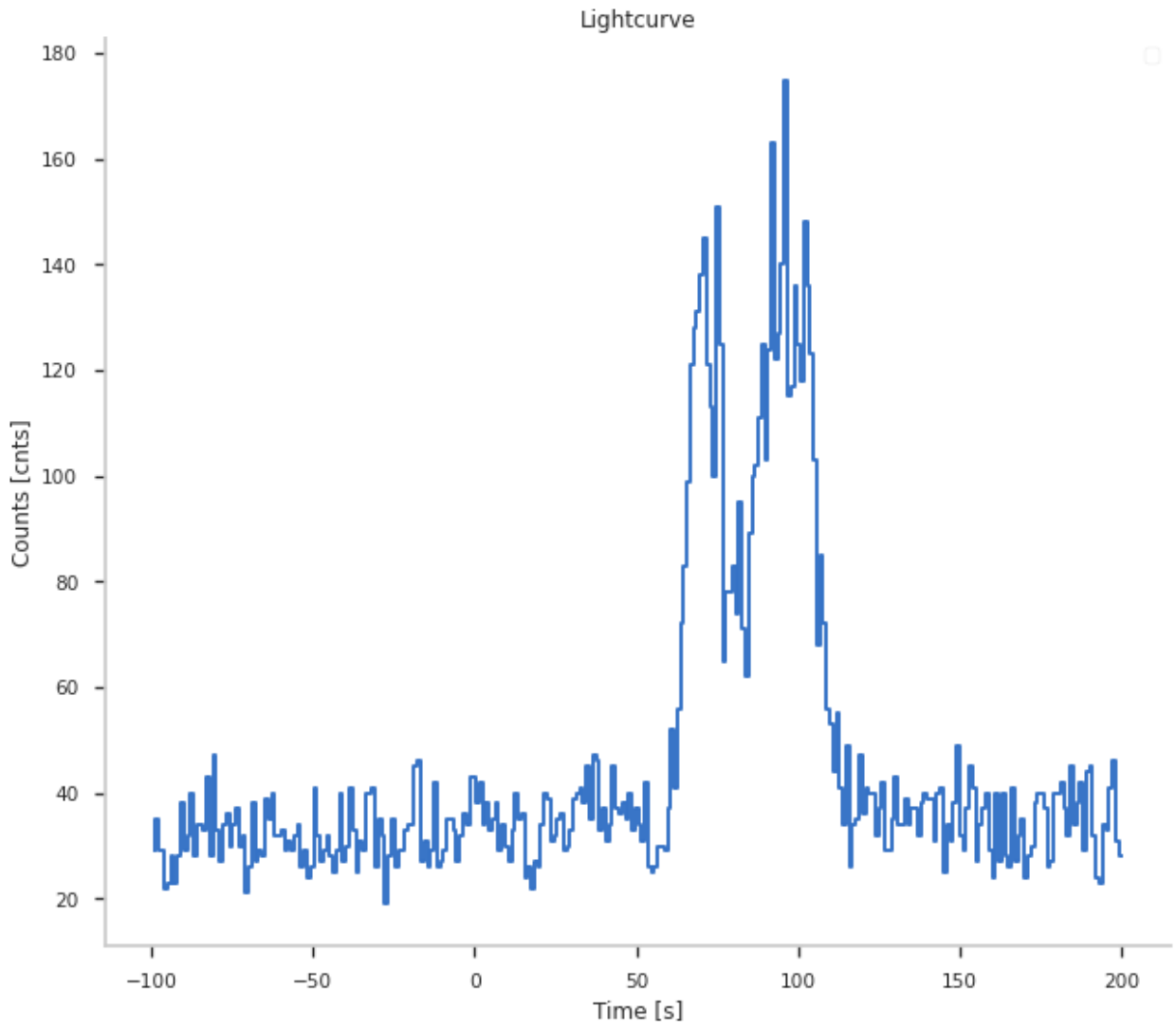
[4]:
```
#arrival times (time in seconds relative to given trigger time)
arrival_times = tsb.time_series.arrival_times

#energy bin of the events
energy_bin = tsb.time_series.measurement
```

### 2.6.3 Lightcurve Data

With the event data you can create the lightcurves manually

[5]:
```
# plot lightcurves for all echans summed together
bins = np.linspace(-100,200,300)
cnts, bins = np.histogram(arrival_times, bins=bins)

fig, ax = plt.subplots(1,1)
ax.step(bins[1:], cnts)
ax.set_xlabel("Time [s]")
ax.set_ylabel("Counts [cnts]")
ax.set_title("Lightcurve")
ax.legend();
```

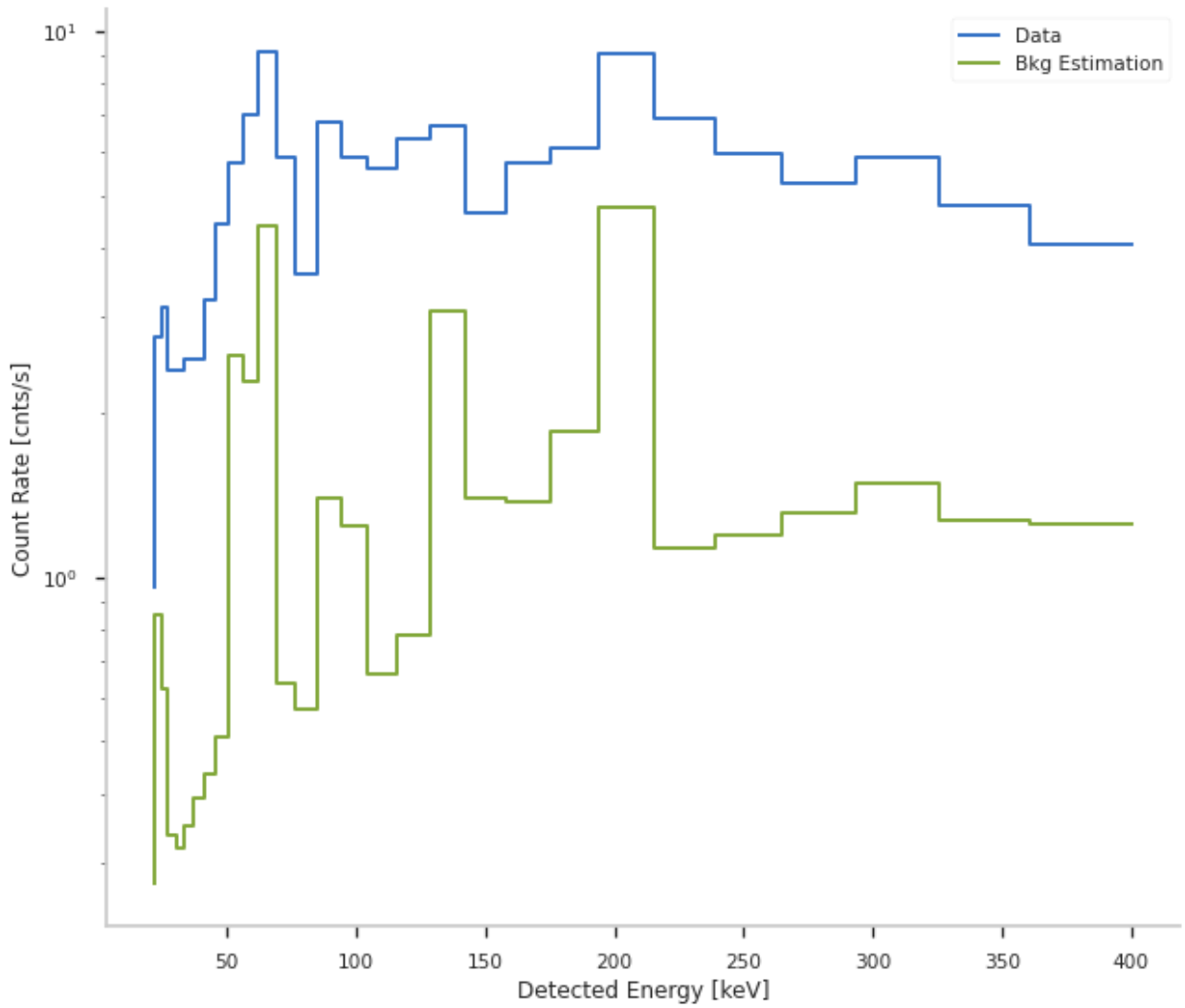
```
No handles with labels found to put in legend.
```



### 2.6.4 Observed Data Active Time

Get the observed data of the active time and background time selection

[6]:
```
# counts
active_time_counts = plugin.observed_counts
estimated_background_counts = plugin.background_counts

# exposure
exposure = plugin.exposure

fig, ax = plt.subplots(1,1)
ax.step(ebounds[1:], active_time_counts/exposure, label="Data")
ax.step(ebounds[1:], estimated_background_counts/exposure, label="Bkg Estimation")

ax.set_xlabel("Detected Energy [keV]")
ax.set_ylabel("Count Rate [cnts/s]")
ax.set_yscale("log")
ax.legend();
```



## 2.7 Contributing

Contributions to PySPI are always welcome. They can come in the form of:

### 2.7.1 Issues

Please use the Github issue tracking system for any bugs, for questions, bug reports and or feature requests.

### 2.7.2 Add to Source Code

To directly contribute to the source code of PySPI, please fork the Github repository, add the changes to one of the branches in your forked repository and then create a pull request to the master of the main repository from this branch. Code contribution is welcome for different topics:

#### 2.7.2.1 Add Functionality

If PySPI is missing some functionality that you need, you can either create an issue in the Github repository or add it to the code and create a pull request. Always make sure that the old tests do not break and adjust them if needed. Also please add tests and documentation for the new functionality in the pyspi/test folder. This ensures that the functionality will not get broken by future changes to the code and other people will know that this feature exists.

#### 2.7.2.2 Code Improvement

You can also contribute code improvements, like making calculations faster or improve the style of the code. Please make sure that the results of the software do not change in this case.

#### 2.7.2.3 Bug Fixes

Fixing bugs that you found or that are mentioned in one of the issues is also a good way to contribute to PySPI. Please also make sure to add tests for your changes to check that the bug is gone and that the bug will not recur in future versions of the code.

#### 2.7.2.4 Documentation

Additions or examples, tutorials, or better explanations are always welcome. To ensure that the documentation builds with the current version of the software, we are using jupytext to write the documentation in Markdown. These are automatically converted to and executed as jupyter notebooks when changes are pushed to Github.

## 2.8 API

Here you can find the documentation of all classes and methods:

## 2.9 pyspi

### 2.9.1 pyspi package

#### 2.9.1.1 Subpackages

##### 2.9.1.1.1 pyspi.io package

##### Subpackages

##### pyspi.io.plotting package

##### Submodules

##### pyspi.io.plotting.spi_display module

**class** pyspi.io.plotting.spi_display.**DetectorContents**(*detector_array*)

Bases: object

**__init__**(*detector_array*)

**classmethod from_spi_data**(*spi_data*)

**classmethod from_total_effective_area**(*spi_response*, *azimuth*, *zenith*)

**class** pyspi.io.plotting.spi_display.**DoubleEventDetector**(*detector_number*, *origin*, *detector1*, *detector2*)

Bases: *pyspi.io.plotting.spi_display.SPIDetector*

**__init__**(*detector_number*, *origin*, *detector1*, *detector2*)

**Parameters**

- **detector_number** –
- **origin** –
- **detector1** –
- **detector2** –

**class** pyspi.io.plotting.spi_display.**SPI**(*bad_detectors=[]*, *time=None*)

Bases: object

**__init__**(*bad_detectors=[]*, *time=None*)

**plot_spi_working_dets**(*with_pseudo_detectors=True*, *show_detector_number=True*)

Plot the SPI Detectors and mark the detectors that are not working red :param with_pseudo_detectors: Plot pseudo detectors? :param show_detector_number: Show the det numbers in the plot? :return:

**class** pyspi.io.plotting.spi_display.**SPIDetector**(*detector_number*, *origin*, *is_pseudo_detector=False*)

Bases: object

**__init__**(*detector_number*, *origin*, *is_pseudo_detector=False*)

A SPI detector is defined by its number, origin and type :param detector_number: the detector number :param origin: the detector origin :param is_pseudo_detector: if this is a real detector or not

**property bad**

**property detector_number**

**property is_pseudo_detector**

**property origin**

**set_bad**(*flag*)
Set the flag if this is a bad detector :param flag: Bad detector? :return:

**class** pyspi.io.plotting.spi_display.**TripleEventDetector**(*detector_number*, *origin*, *is_pseudo_detector=False*)
Bases: *pyspi.io.plotting.spi_display.SPIDetector*

### Module contents

### Submodules

### pyspi.io.file_utils module

pyspi.io.file_utils.**file_existing_and_readable**(*filename*)
Check if a file exists :param filename: Filename to check :return: True or False

pyspi.io.file_utils.**path_exists_and_is_directory**(*path*)
Check if a path exists and is a directory :param path: Path to check :return: True or False

pyspi.io.file_utils.**sanitize_filename**(*filename*, *abspath=False*)
Sanitize filename :param filename: name of file :param abspath: Get the absolute path? :return: sanitized filename

### pyspi.io.get_files module

pyspi.io.get_files.**create_file_structure**(*pointing_id*)
Create the file structure to save the datafiles :param pointing_id: Id of pointing e.g. '180100610010' as string! :return:

pyspi.io.get_files.**get_and_save_file**(*extension*, *pointing_id*, *access='isdc'*)
Function to get and save a file located at file_path to file_save_path :param extension: File name you want to download :param pointing_id: The id of the pointing :param access: How to get the data. Possible are "isdc" and "afs" :return:

pyspi.io.get_files.**get_files**(*pointing_id*, *access='isdc'*)
Function to get the needed files for a certain pointing_id and save them in the correct folders. :param pointing_id: Id of pointing e.g. '180100610010' as string or int :param access: How to get the data. Possible are "isdc" and "afs" :return:

#### pyspi.io.package_data module

pyspi.io.package_data.**get_path_of_external_data_dir**()
Get path to the external data directory (mostly to store data there)

pyspi.io.package_data.**get_path_of_internal_data_dir**()
Get path to the external data directory (mostly to store data there)

#### Module contents

#### 2.9.1.1.2 pyspi.test package

#### Submodules

#### pyspi.test.test_active_dets module

pyspi.test.test_active_dets.**test_active_dets_and_response_version**()

pyspi.test.test_active_dets.**test_plotting**()

#### pyspi.test.test_download module

pyspi.test.test_download.**test_download**()

#### pyspi.test.test_grb_fit module

pyspi.test.test_grb_fit.**test_grb_fit**()

#### pyspi.test.test_packagedata module

pyspi.test.test_packagedata.**test_packagedata**()

#### pyspi.test.test_response module

pyspi.test.test_response.**test_response**()

#### pyspi.test.test_spipointing module

pyspi.test.test_spipointing.**test_spipointing**()

### pyspi.test.test_time_series module

`pyspi.test.test_time_series.test_time_series_with_response()`

`pyspi.test.test_time_series.test_time_series_without_response()`

### Module contents

#### 2.9.1.1.3 pyspi.utils package

### Subpackages

### pyspi.utils.data_builder package

### Submodules

### pyspi.utils.data_builder.time_series_builder module

**class** `pyspi.utils.data_builder.time_series_builder.SPISWFile(det, pointing_id, ebounds)`

Bases: `object`

**__init__**(*det*, *pointing_id*, *ebounds*)

Class to read in all the data needed from a SCW file for a given config file

**Parameters**

- **config** – Config yml filename, Config object or dict
- **det** – For which detector?

**property deadtime_bin_starts**

Start time of time bins which have the deadtime information

**Type** returns

**property deadtime_bin_stops**

Stop time of time bins which have the deadtime information

**Type** returns

**property deadtimes_per_interval**

Deadtime per time bin which have the deadtime information

**Type** returns

**property det**

detector ID

**Type** returns

**property det_name**

Name det

**Type** returns

**property ebounds**

ebounds of analysis

**Type** returns

**property energies**
energies of detected events

**Type** returns

**property energy_bins**
energy bin number of every event

**Type** returns

**property geometry_file_path**
Path to the spacecraft geometry file

**Type** returns

**property livetimes_per_interval**
Livetime per time bin which have the deadtime information

**Type** returns

**property mission**
Name Mission

**Type** returns

**property n_channels**
number energy channels

**Type** returns

**property time_start**
start time of lightcurve

**Type** returns

**property time_stop**
stop time of lightcurve

**Type** returns

**property times**
times of detected events

**Type** returns

**class** pyspi.utils.data_builder.time_series_builder.**SPISWFileGRB**(*det*, *ebounds*, *time_of_grb*, *sgl_type=None*)

Bases: object

**__init__**(*det*, *ebounds*, *time_of_grb*, *sgl_type=None*)
Class to read in all the data needed from a SCW file for a given grbtime

**Parameters**

- **det** – For which detector?
- **ebounds** – Ebounds for the Analysis.
- **time_of_grb** – Time of the GRB as "YYMMDD HHMMSS"
- **sgl_type** – Which type of single events?

Only normal sgl, psd or both?

**Returns** Object

**property deadtime_bin_starts**
Start time of time bins which have the deadtime information

**Type** returns

**property deadtime_bin_stops**
Stop time of time bins which have the deadtime information

**Type** returns

**property deadtimes_per_interval**
Deadtime per time bin which have the deadtime information

**Type** returns

**property det**
detector ID

**Type** returns

**property det_name**
Name det

**Type** returns

**property ebounds**
ebounds of analysis

**Type** returns

**property energies**
energies of detected events

**Type** returns

**property energy_bins**
energy bin number of every event

**Type** returns

**property geometry_file_path**
Path to the spacecraft geometry file

**Type** returns

**property livetimes_per_interval**
Livetime per time bin which have the deadtime information

**Type** returns

**property mission**
Name Mission

**Type** returns

**property n_channels**
number energy channels

**Type** returns

**property time_start**
start time of lightcurve

**Type** returns

**property time_stop**
stop time of lightcurve

**Type** returns

**property times**

times of detected events

**Type** returns

**class** pyspi.utils.data_builder.time_series_builder.**TimeSeriesBuilderSPI**(*name*, *time_series*, *response=None*, *poly_order=-1*, *verbose=True*, *restore_poly_fit=None*, *container_type=<class 'threeML.utils.spectrum.binned_spectrum*, ***kwargs*)

Bases: threeML.utils.data_builders.time_series_builder.TimeSeriesBuilder

**__init__**(*name*, *time_series*, *response=None*, *poly_order=-1*, *verbose=True*, *restore_poly_fit=None*, *container_type=<class 'threeML.utils.spectrum.binned_spectrum.BinnedSpectrumWithDispersion'>*, ***kwargs*)

Class to build the time_series for SPI. Inherited from the 3ML TimeSeriesBuilder with added class methods to build the object for SPI datafiles. :param name: Name of the tsb :param time_series: Timeseries with the data :param response: Response object :param poly_order: poly order for the polynominal fitting :param verbose: Verbose? :param restore_poly_fit: Path to a file with the poly bkg fits :param containter_type: ContainerType for spectrum :returns: Object

**classmethod from_spi_constant_pointing**(*det=0*, *pointing_id='118900570010'*, *ebounds=None*, *response=None*)

Class method to build the time_series_builder for a given pointing id

**Parameters**

- **det** – Which det?
- **ebounds** – Output ebounds for analysis.
- **pointing_id** – Pointing ID
- **response** – InstrumenResponse Object

**Returns** Initalized TimeSeriesBuilder object

**classmethod from_spi_grb**(*name*, *det*, *time_of_grb*, *ebounds=None*, *response=None*, *sgl_type=None*, *restore_background=None*, *poly_order=0*, *verbose=True*)

Class method to build the time_series_builder for a given GRB time

**Parameters**

- **name** – Name of object
- **det** – Which det?
- **ebounds** – Output ebounds for analysis.
- **time_of_grb** – Astropy time object with the time of the GRB (t0)
- **response** – InstrumenResponse Object
- **sgl_type** – What kind of sinlge events? Standard single events? PSD events? Or both?
- **restore_background** – File to restore bkg
- **poly_order** – Which poly_order? -1 gives automatic determination

- **verbose** – Verbose?

**Returns** Initalized TimeSeriesBuilder object

### Module contents

**class** pyspi.utils.data_builder.**SPISWFileGRB**(*det*, *ebounds*, *time_of_grb*, *sgl_type=None*)

Bases: object

**__init__**(*det*, *ebounds*, *time_of_grb*, *sgl_type=None*)

Class to read in all the data needed from a SCW file for a given grbtime

**Parameters**

- **det** – For which detector?
- **ebounds** – Ebounds for the Analysis.
- **time_of_grb** – Time of the GRB as "YYMMDD HHMMSS"
- **sgl_type** – Which type of single events?

Only normal sgl, psd or both?

**Returns** Object

**property deadtime_bin_starts**

Start time of time bins which have the deadtime information

**Type** returns

**property deadtime_bin_stops**

Stop time of time bins which have the deadtime information

**Type** returns

**property deadtimes_per_interval**

Deadtime per time bin which have the deadtime information

**Type** returns

**property det**

detector ID

**Type** returns

**property det_name**

Name det

**Type** returns

**property ebounds**

ebounds of analysis

**Type** returns

**property energies**

energies of detected events

**Type** returns

**property energy_bins**

energy bin number of every event

**Type** returns

**property geometry_file_path**
Path to the spacecraft geometry file

**Type** returns

**property livetimes_per_interval**
Livetime per time bin which have the deadtime information

**Type** returns

**property mission**
Name Mission

**Type** returns

**property n_channels**
number energy channels

**Type** returns

**property time_start**
start time of lightcurve

**Type** returns

**property time_stop**
stop time of lightcurve

**Type** returns

**property times**
times of detected events

**Type** returns

**class** pyspi.utils.data_builder.**TimeSeriesBuilderSPI**(*name*, *time_series*, *response=None*, *poly_order=-1*, *verbose=True*, *restore_poly_fit=None*, *container_type=<class 'threeML.utils.spectrum.binned_spectrum.BinnedSpectrumWithDisp*
***kwargs*)

Bases: threeML.utils.data_builders.time_series_builder.TimeSeriesBuilder

**__init__**(*name*, *time_series*, *response=None*, *poly_order=-1*, *verbose=True*, *restore_poly_fit=None*, *container_type=<class 'threeML.utils.spectrum.binned_spectrum.BinnedSpectrumWithDispersion'>*, ***kwargs*)

Class to build the time_series for SPI. Inherited from the 3ML TimeSeriesBuilder with added class methods to build the object for SPI datafiles. :param name: Name of the tsb :param time_series: Timeseries with the data :param response: Response object :param poly_order: poly order for the polynominal fitting :param verbose: Verbose? :param restore_poly_fit: Path to a file with the poly bkg fits :param containter_type: ContainerType for spectrum :returns: Object

**classmethod from_spi_constant_pointing**(*det=0*, *pointing_id='118900570010'*, *ebounds=None*, *response=None*)

Class method to build the time_series_builder for a given pointing id

**Parameters**

- **det** – Which det?
- **ebounds** – Output ebounds for analysis.
- **pointing_id** – Pointing ID
- **response** – InstrumenResponse Object

**Returns** Initalized TimeSeriesBuilder object

**classmethod from_spi_grb**(*name*, *det*, *time_of_grb*, *ebounds=None*, *response=None*, *sgl_type=None*, *restore_background=None*, *poly_order=0*, *verbose=True*)

Class method to build the time_series_builder for a given GRB time

**Parameters**

- **name** – Name of object
- **det** – Which det?
- **ebounds** – Output ebounds for analysis.
- **time_of_grb** – Astropy time object with the time of the GRB (t0)
- **response** – InstrumenResponse Object
- **sgl_type** – What kind of sinlge events? Standard single events? PSD events? Or both?
- **restore_background** – File to restore bkg
- **poly_order** – Which poly_order? -1 gives automatic determination
- **verbose** – Verbose?

**Returns** Initalized TimeSeriesBuilder object

## pyspi.utils.response package

### Submodules

### pyspi.utils.response.spi_drm module

**class** pyspi.utils.response.spi_drm.**SPIDRM**(*drm_generator*, *ra*, *dec*)

Bases: threeML.utils.OGIP.response.InstrumentResponse

**__init__**(*drm_generator*, *ra*, *dec*)

Init a SPIDRM object which is based on the InstrumenResponse class from 3ML. Contains everything that is necessary for 3ML to recognize it as a response.

**Parameters**

- **drm_generator** – DRM generator for the SPI Response
- **ra** – ra of source (in ICRS)
- **dec** – dec of source (in ICRS)

**clone**() → *pyspi.utils.response.spi_drm.SPIDRM*

Get clone of this response object

**Returns** new cloned response

**set_location**(*ra*, *dec*, *cache=False*)

Set the source location

**Parameters**

- **ra** – ra of source (in ICRS)
- **dec** – dec of source (in ICRS)

**Returns**

**set_location_direct_sat_coord**(*azimuth*, *zenith*, *cache=False*)
Set the source location

**Parameters**

- **azimuth** – az poisition in the sat. frame
- **zenith** – zenith poisition in the sat. frame

**Returns**

### pyspi.utils.response.spi_frame module

**class** pyspi.utils.response.spi_frame.**SPIFrame**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

Bases: astropy.coordinates.baseframe.BaseCoordinateFrame

INTEGRAL SPI Frame :Parameters: **representation** (*BaseRepresentation* or None) – A representation object or None to have no data (or use the other keywords)

**property default_differential**

**property default_representation**

**frame_attributes = {'scx_dec': <astropy.coordinates.attributes.Attribute object>, 'scx_ra': <astropy.coordinates.attributes.Attribute object>, 'scy_dec': <astropy.coordinates.attributes.Attribute object>, 'scy_ra': <astropy.coordinates.attributes.Attribute object>, 'scz_dec': <astropy.coordinates.attributes.Attribute object>, 'scz_ra': <astropy.coordinates.attributes.Attribute object>}**

**property frame_specific_representation_info**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

**dict(iterable) -> new dictionary initialized as if via:** d = {} for k, v in iterable:

d[k] = v

**dict(**kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

**name = 'spiframe'**

**scx_dec = None**

**scx_ra = None**

**scy_dec = None**

**scy_ra = None**

**scz_dec = None**

**scz_ra = None**

pyspi.utils.response.spi_frame.**j2000_to_spi**(*j2000_frame*, *spi_coord*)
Transform icrs frame to SPI frame

pyspi.utils.response.spi_frame.**spi_to_j2000**(*spi_coord*, *j2000_frame*)
Transform spi fram to ICRS frame

`pyspi.utils.response.spi_frame.transform_icrs_to_spi(ra_icrs, dec_icrs, sc_matrix)`

Calculates lon, lat in spi frame for given ra, dec in ICRS frame and given sc_matrix (sc_matrix pointing dependent)

**Parameters**

- **ra_icrs** – Ra in ICRS in degree
- **dec_icrs** – Dec in ICRS in degree
- **sc_matrix** – sc Matrix that gives orientation of SPI in ICRS frame

**Returns** lon, lat in spi frame

`pyspi.utils.response.spi_frame.transform_spi_to_icrs(az_spi, zen_spi, sc_matrix)`

Calculates lon, lat in spi frame for given ra, dec in ICRS frame and given sc_matrix (sc_matrix pointing dependent)

**Parameters**

- **az_spi** – azimuth in SPI coord system in degree
- **zen_spi** – zenit in SPI coord system in degree
- **sc_matrix** – sc Matrix that gives orientation of SPI in ICRS frame

**Returns** ra, dex in ICRS in deg

### pyspi.utils.response.spi_pointing module

**class** `pyspi.utils.response.spi_pointing.SPIPointing(sc_pointing_file)`

Bases: `object`

**__init__**(*sc_pointing_file*)

This class handles the **current** SPI pointings based of the input SPI pointing file

**Parameters** **sc_pointing_file** – An INTEGRAL/SPI spacecraft pointing file

**Returns**

**property sc_matrix**

get the sc_matrics of all the times in this pointing :returns: array of sc_matrices

**property sc_points**

ra, dec coordinates of the SPI x,y and z axis in the ICRS frame for all the times in this pointing

**Returns** ra, dec coordinates of the SPI x,y and z axis in the ICRS frame for all the pointings

`pyspi.utils.response.spi_pointing.construct_sc_matrix(scx_ra, scx_dec, scy_ra, scy_dec, scz_ra, scz_dec)`

Construct the sc_matrix, with which we can transform ICRS <-> Sat. Frame

**Parameters**

- **scx_ra** – ra coordinate of satellite x-axis in ICRS
- **scx_dec** – dec coordinate of satellite x-axis in ICRS
- **scy_ra** – ra coordinate of satellite y-axis in ICRS
- **scy_dec** – dec coordinate of satellite y-axis in ICRS
- **scz_ra** – ra coordinate of satellite z-axis in ICRS
- **scz_dec** – dec coordinate of satellite z-axis in ICRS

**Returns** sc_matrix (3x3)

`pyspi.utils.response.spi_pointing.construct_scy(scx_ra, scx_dec, scz_ra, scz_dec)`

Construct the vector of the y-axis of the Integral coord system in the ICRS frame

**Parameters**

- **scx_ra** – ra coordinate of satellite x-axis in ICRS
- **scx_dec** – dec coordinate of satellite x-axis in ICRS
- **scz_ra** – ra coordinate of satellite z-axis in ICRS
- **scz_dec** – dec coordinate of satellite z-axis in ICRS

**Returns** vector of the y-axis of the Integral coord system

in the ICRS frame

### pyspi.utils.response.spi_response module

`class pyspi.utils.response.spi_response.ResponseGenerator(pointing_id=None, ebounds=None, response_irf_read_object=None, det=None)`

Bases: `object`

`__init__(pointing_id=None, ebounds=None, response_irf_read_object=None, det=None)`

Base Response Class - Here we have everything that stays the same for GRB and Constant Pointsource Reponses

**Parameters**

- **ebounds** – User defined ebins for binned effective area
- **response_irf_read_object** – Object that holds the read in irf values
- **sc_matrix** – Matrix to convert SPI coordinate system <-> ICRS
- **det** – Which detector

`property det`

detector number

**Type** returns

`property ebounds`

Ebounds of the analysis

**Type** returns

`property ene_max`

End of Ebounds

**Type** returns

`property ene_min`

Start of ebounds

**Type** returns

`get_xy_pos(azimuth, zenith)`

Get xy position (in SPI simulation) for given azimuth and zenith

**Parameters**

- **azimuth** – Azimuth in Sat. coordinates [rad]
- **zenith** – Zenith in Sat. coordinates [rad]

**Returns** grid position in (x,y) coordinates

**property irf_ob**

the irf_read object with the information from the response simulation

**Type** returns

**property rod**

Ensure that you know what you are doing.

**Returns** Roland

**set_binned_data_energy_bounds**(*ebounds*)

Change the energy bins for the binned effective_area

**Parameters** **ebounds** – New ebinedges: ebounds[:-1] start of ebins,

ebounds[1:] end of ebins

**Returns**

**set_location**(*ra*, *dec*)

Calculate the weighted irfs for the three event types for a given position

**Parameters**

- **azimuth** – Azimuth position in sat frame
- **zenith** – Zenith position in sat frame

**Returns**

**set_location_direct_sat_coord**(*azimuth*, *zenith*)

Calculate the weighted irfs for the three event types for a given position

**Parameters**

- **azimuth** – Azimuth position in sat frame
- **zenith** – Zenith position in sat frame

**Returns** ra and dec value

**class** pyspi.utils.response.spi_response.**ResponsePhotopeakGenerator**(*pointing_id=None*, *ebounds=None*, *response_irf_read_object=None*, *det=None*)

Bases: *pyspi.utils.response.spi_response.ResponseGenerator*

**__init__**(*pointing_id=None*, *ebounds=None*, *response_irf_read_object=None*, *det=None*)

Init Response object with photopeak only

**Parameters** **pointing_id** – The pointing ID for which the

response should be valid :param ebound: Ebounds of Ebins :param response_irf_read_object: Object that holds the read in irf values :param det: Detector ID

**property effective_area**

vector with photopeak effective area

**Type** returns

**classmethod from_time**(*time*, *det*, *ebounds*, *rsp_read_obj*)

Init Response object with photopeak only

**Parameters**

- **time** – The time for which the response should be valid
- **ebound** – Ebounds of Ebins
- **response_irf_read_object** – Object that holds the read in irf values
- **det** – Detector ID

**Returns** Object

**class** pyspi.utils.response.spi_response.**ResponseRMFGenerator**(*pointing_id=None*, *monte_carlo_energies=None*, *ebounds=None*, *response_irf_read_object=None*, *det=None*, *fixed_rsp_matrix=None*)

Bases: *pyspi.utils.response.spi_response.ResponseGenerator*

**__init__**(*pointing_id=None*, *monte_carlo_energies=None*, *ebounds=None*, *response_irf_read_object=None*, *det=None*, *fixed_rsp_matrix=None*)

Init Response object with total RMF used

**Parameters**

- **pointing_id** – The pointing ID for which the response should be valid
- **ebound** – Ebounds of Ebins
- **monte_carlo_energies** – Input energy bin edges
- **response_irf_read_object** – Object that holds the read in irf values
- **det** – Detector ID
- **fixed_rsp_matrix** – A fixed response matrix to overload the normal matrix

**Returns** Object

**clone**()

Clone this response object

**Returns** cloned response

**classmethod from_time**(*time*, *det*, *ebounds*, *monte_carlo_energies*, *rsp_read_obj*, *fixed_rsp_matrix=None*)

Init Response object with total RMF used from a time

**Parameters**

- **time** – Time for which to construct the response object
- **ebound** – Ebounds of Ebins
- **monte_carlo_energies** – Input energy bin edges
- **response_irf_read_object** – Object that holds

the read in irf values :param det: Detector ID :param fixed_rsp_matrix: A fixed response matrix to overload the normal matrix

**Returns** Object

**property matrix**

response matrix

**Type** returns

**property monte_carlo_energies**
Input energies for response

**Type** returns

**property transpose_matrix**
transposed response matrix

**Type** returns

pyspi.utils.response.spi_response.**add_frac**(*ph_matrix*, *i*, *idx*, *ebounds*, *einlow*, *einhigh*)
Recursive Funktion to get the fraction of einlow…

pyspi.utils.response.spi_response.**log_interp1d**(*x_new*, *x_old*, *y_old*)
Linear interpolation in log space for base value pairs (x_old, y_old) for new x_values x_new

**Parameters**

- **x_old** – Old x values used for interpolation
- **y_old** – Old y values used for interpolation
- **x_new** – New x values

**Returns** y_new from liner interpolation in log space

pyspi.utils.response.spi_response.**multi_response_irf_read_objects**(*times*, *detector*, *drm='Photopeak'*)
TODO: This is very ugly. Come up with a better way. Function to initalize the needed responses for the given times. Only initalize every needed response version once! Because of memory. One response object needs about 1 GB of RAM… TODO: Not needed at the moment. We need this when we want to analyse many pointings together.

**Parameters** **times** – Times of the different sw used

**Returns** list with correct response version object of the times

pyspi.utils.response.spi_response.**trapz**(*y*, *x*)
Fast trapz integration with numba

**Parameters**

- **x** – x values
- **y** – y values

**Returns** Trapz integrated

### pyspi.utils.response.spi_response_data module

**class** pyspi.utils.response.spi_response_data.**ResponseData**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*)

Bases: `object`

Base Dataclass to hold the IRF data

**__init__**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*) → None

**ebounds_rmf_2_base: numpy.array**

**ebounds_rmf_3_base: numpy.array**

**energies_database: numpy.array**

**get_data**(*version*)

Read in the data we need from the irf hdf5 file

**Parameters** **version** – Version of irf file

**Returns** all the infomation we need as a list

**irf_nx: int**

**irf_ny: int**

**irf_xbin: float**

**irf_xmin: float**

**irf_ybin: float**

**irf_ymin: float**

**n_dets: int**

**rmf_2_base: numpy.array**

**rmf_3_base: numpy.array**

**class** pyspi.utils.response.spi_response_data.**ResponseDataPhotopeak**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*, *irfs_photopeak: numpy.array*)

Bases: *pyspi.utils.response.spi_response_data.ResponseData*

Dataclass to hold the IRF data if we only need the photopeak irf

**__init__**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*, *irfs_photopeak: numpy.array*) → None

**classmethod from_version**(*version*)

Construct the dataclass object

**Parameters** **version** – Which IRF version?

**Returns** ResponseDataPhotopeak object

**irfs_photopeak: numpy.array**

**class** pyspi.utils.response.spi_response_data.**ResponseDataRMF**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*, *irfs_photopeak: numpy.array*, *irfs_nonphoto_1: numpy.array*, *irfs_nonphoto_2: numpy.array*)

Bases: *pyspi.utils.response.spi_response_data.ResponseData*

Dataclass to hold the IRF data if we only need all three irfs

**__init__**(*energies_database: numpy.array*, *irf_xmin: float*, *irf_ymin: float*, *irf_xbin: float*, *irf_ybin: float*, *irf_nx: int*, *irf_ny: int*, *n_dets: int*, *ebounds_rmf_2_base: numpy.array*, *rmf_2_base: numpy.array*, *ebounds_rmf_3_base: numpy.array*, *rmf_3_base: numpy.array*, *irfs_photopeak: numpy.array*, *irfs_nonphoto_1: numpy.array*, *irfs_nonphoto_2: numpy.array*) → None

**classmethod from_version**(*version*)

Construct the dataclass object

**Parameters** **version** – Which IRF version?

**Returns** ResponseDataPhotopeak object

**irfs_nonphoto_1: numpy.array**

**irfs_nonphoto_2: numpy.array**

**irfs_photopeak: numpy.array**

pyspi.utils.response.spi_response_data.**load_rmf_non_ph_1**()

Load the RMF for the non-photopeak events that first interact in the det

**Returns** ebounds of RMF and rmf matrix for the non-photopeak events that first interact in the det

pyspi.utils.response.spi_response_data.**load_rmf_non_ph_2**()

Load the RMF for the non-photopeak events that first interact in the dead material

**Returns** ebounds of RMF and rmf matrix for the non-photopeak events that first interact in the dead material

### Module contents

### Submodules

### pyspi.utils.function_utils module

pyspi.utils.function_utils.**ISDC_MJD**(*time_object*)

Get INTEGRAL MJD time from a given time object

**Parameters** **time_object** – Astropy time object of grb time

**Returns** Time in Integral MJD time

pyspi.utils.function_utils.**ISDC_MJD_to_cxcsec**(*ISDC_MJD_time*)

Convert ISDC_MJD to UTC

**Parameters** **ISDC_MJD_time** – time in ISDC_MJD time format

**Returns** time in cxcsec format (seconds since 1998-01-01 00:00:00)

`pyspi.utils.function_utils.find_needed_ids(time)`
Get the pointing id of the needed data to cover the GRB time

**Returns** Needed pointing id

`pyspi.utils.function_utils.find_response_version(time)`
Find the correct response version number for a given time

**Parameters** **time** – time of interest

**Returns** response version number

`pyspi.utils.function_utils.get_time_object(time)`
Transform the input into a time object. Input can either be a time object or a string with the format "YYMMDD HHMMSS"

**Parameters** **time** – time object or a string with the format "YYMMDD HHMMSS"

**Returns** time object

`pyspi.utils.function_utils.leapseconds(time_object)`
Hard coded leap seconds from start of INTEGRAL to time of time_object

**Parameters** **time_object** – Time object to which the number of

leapseconds should be detemined

**Returns** TimeDelta object of the needed leap seconds

### pyspi.utils.geometry module

`pyspi.utils.geometry.cart2polar(vector)`
Convert cartesian to ra, dec

**Parameters** **vector** – cartesian coord vector

**Returns** ra and dec

`pyspi.utils.geometry.polar2cart(ra, dec)`
Convert ra, dec to cartesian

**Parameters**

- **ra** – ra coord
- **dec** – dec coord

**Returns** cartesian coord vector

### pyspi.utils.livedets module

`pyspi.utils.livedets.get_live_dets(time, event_types=['single', 'double', 'triple'])`

Get the live dets for a given time

**Parameters**

- **time** – Live dets at a given time. Either "YYMMDD HHMMSS" or as astropy time object
- **event_types** – which event types? List with single, double and/or triple

**Returns** array of live dets

`pyspi.utils.livedets.get_live_dets_pointing(pointing, event_types=['single', 'double', 'triple'])`

Get livedets for a given pointing id

**Parameters**

- **pointing** – pointing id
- **event_types** – which event types? List with single, double and/or triple

**Returns**

### Module contents

## 2.9.1.2 Submodules

### 2.9.1.2.1 pyspi.SPILike module

**class** `pyspi.SPILike.SPILike(name: str, observation, background, bkg_base_array, free_position: bool, verbose: bool = True, **kwargs)`

Bases: `threeML.plugins.DispersionSpectrumLike.DispersionSpectrumLike`

Plugin for the data of SPI, based on PySPI

**__init__**`(name: str, observation, background, bkg_base_array, free_position: bool, verbose: bool = True, **kwargs)`

Init the plugin for a constant source analysis with PySPI

**Parameters**

- **name** – Name of plugin
- **observation** – observed spectrum
- **background** – background spectrum
- **bkg_base_array** – Base array for background model
- **free_position** – Free the position in the fit?
- **verbose** – Verbose?

**Returns** Object

**classmethod from_spectrumlike**`(spectrum_like, bkg_base_array, free_position=False)`

Generate SPILikeGRB from an existing SpectrumLike child

**Parameters**

- **spectrum_like** – SpectrumLike child
- **rsp_object** – Response object

**Free_position** Free the position? boolean

**Returns** Initialized Object

**get_model**(*precalc_fluxes: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get the model

**Parameters** **precalc_fluxes** – Precaclulated flux of spectrum

**Returns** model counts

**set_free_position**(*flag*)
Set the free position flag

**Parameters** **flag** – True or False

**Returns**

**set_model**(*likelihood_model: astromodels.core.model.Model*) → None
Set the model to be used in the joint minimization.

**Parameters** **likelihood_model** – likelihood model instance

**Returns**

**class** pyspi.SPILike.**SPILikeGRB**(*name*, *observation*, *background=None*, *free_position=False*, *verbose=True*, ***kwargs*)

Bases: threeML.plugins.DispersionSpectrumLike.DispersionSpectrumLike

Plugin for the data of SPI, based on PySPI

**__init__**(*name*, *observation*, *background=None*, *free_position=False*, *verbose=True*, ***kwargs*)
Init the plugin for a GRB analysis with PySPI

**Parameters**

- **name** – Name of plugin
- **observation** – observed spectrum
- **background** – background spectrum
- **free_position** – Free the position in the fit?
- **verbose** – Verbose?

**classmethod from_spectrumlike**(*spectrum_like*, *free_position=False*)
Generate SPILikeGRB from an existing SpectrumLike child

**Parameters**

- **spectrum_like** – SpectrumLike child
- **rsp_object** – Response object

**Free_position** Free the position? boolean

**Returns** Initialized Object

**get_model**(*precalc_fluxes=None*)
Get the model

**Parameters** **precalc_fluxes** – Precaclulated flux of spectrum

**Returns** model counts

**set_free_position**(*flag*)
Set the free position flag

**Parameters** **flag** – True or False

**Returns**

**set_model**(*likelihood_model*)
Set the model to be used in the joint minimization.

**Parameters** **likelihood_model** – likelihood model instance

**Returns**

#### 2.9.1.3 Module contents

## 2.10 Analyse GRB data

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

The first thing we need to do, is to specify the time of the GRB. We do this by specifying a astropy time object or a string in the format YYMMDD HHMMSS.

[2]:
```
from astropy.time import Time
grbtime = Time("2012-07-11T02:44:53", format='isot', scale='utc')
#grbtime = "120711 024453" # works also
```

Next, we need to specify the output and input energy bins we want to use.

[3]:
```
import numpy as np
ein = np.geomspace(20,800,300)
ebounds = np.geomspace(20,400,30)
```

Due to detector failures there are several versions of the response for SPI. Therefore we have to find the version number for the time of the GRB and construct the base response object for this version.

[4]:
```
from pyspi.utils.function_utils import find_response_version
from pyspi.utils.response.spi_response_data import ResponseDataRMF
version = find_response_version(grbtime)
print(version)
rsp_base = ResponseDataRMF.from_version(version)
```

```
4
Using the irfs that are valid between 10/05/27 12:45:00 and present (YY/MM/DD HH:MM:SS)
```

Now we can create the response object for detector 0 and set the position of the GRB, which we already know.

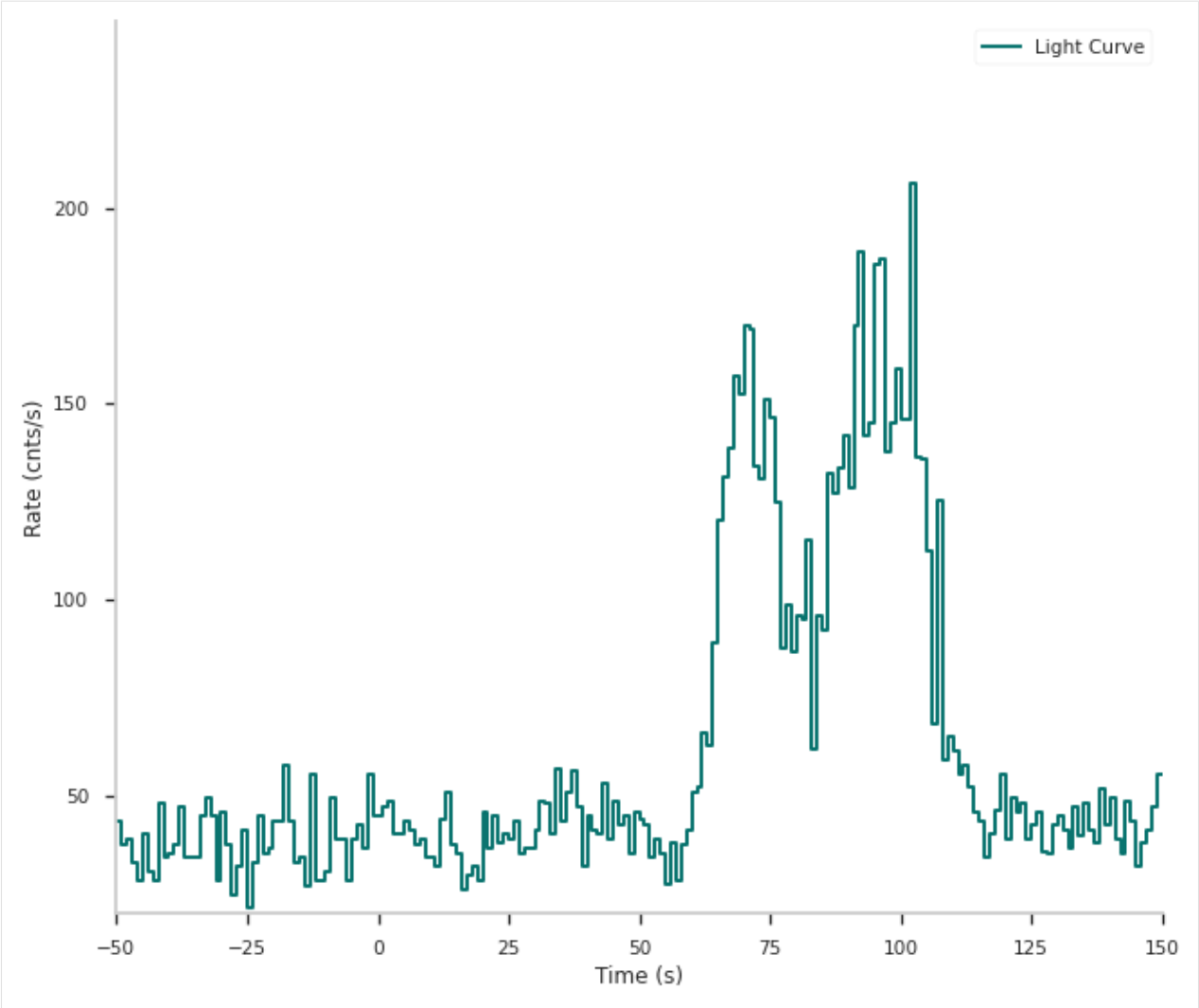
[5]:
```
from pyspi.utils.response.spi_response import ResponseRMFGenerator
from pyspi.utils.response.spi_drm import SPIDRM
det=0
ra = 94.6783
dec = -70.99905
drm_generator = ResponseRMFGenerator.from_time(grbtime,
                                               det,
                                               ebounds,
                                               ein,
                                               rsp_base)
sd = SPIDRM(drm_generator, ra, dec)
```

With this we can build a time series and we use all the single events in this case (PSD + non PSD; see section about electronic noise). To be able to convert the time series into 3ML plugins later, we need to assign them a response object.

[6]:
```
from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
tsb = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
    det,
    grbtime,
    response=sd,
    sgl_type="both",
    )
```

Now we can have a look at the light curves from -50 to 150 seconds around the specified GRB time.

[7]:
```
fig = tsb.view_lightcurve(-50,150)
```



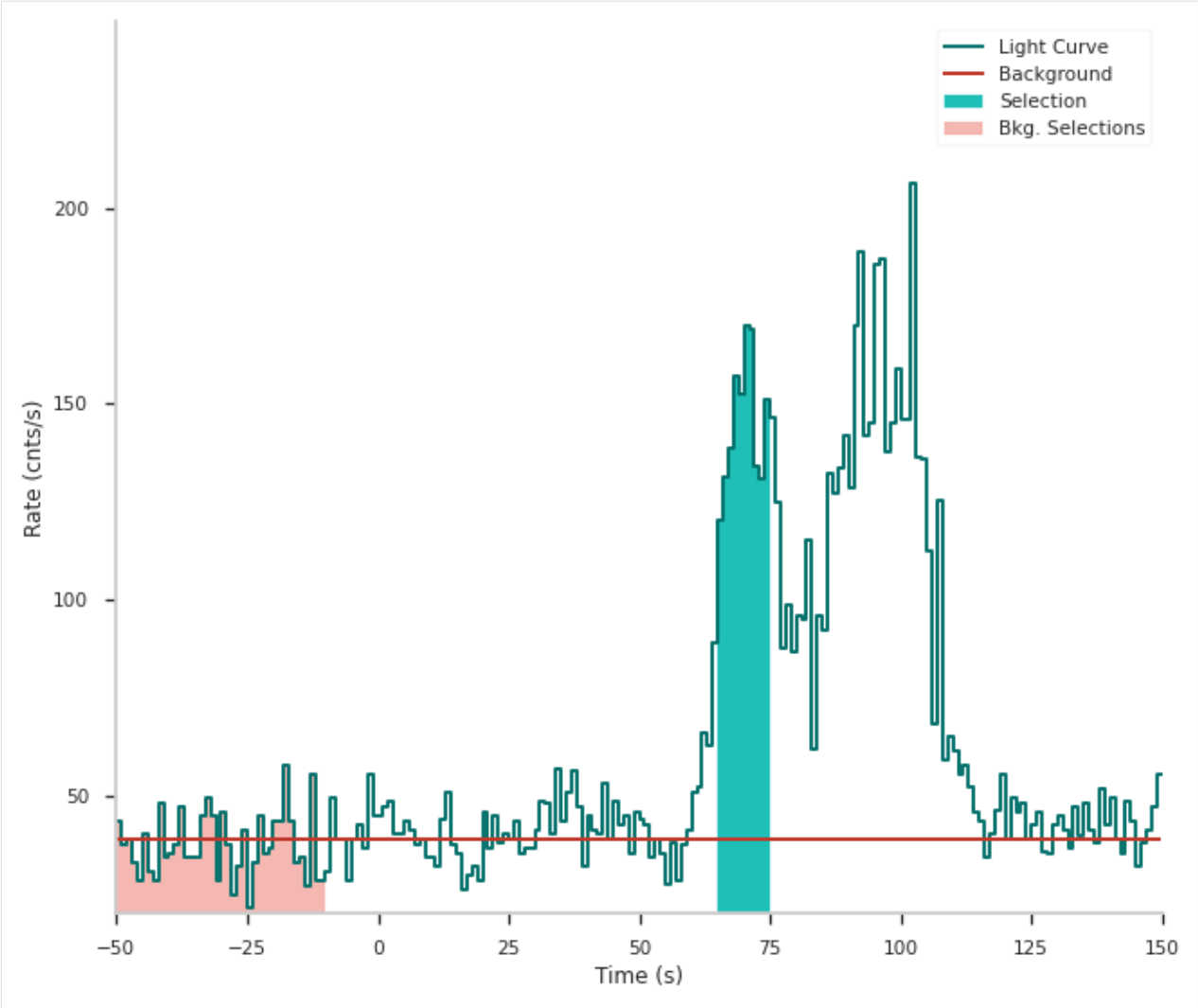

With this we can select the active time and some background time intervals.

[8]:
```
active_time = "65-75"
bkg_time1 = "-500--10"
bkg_time2 = "150-1000"
tsb.set_active_time_interval(active_time)
tsb.set_background_interval(bkg_time1, bkg_time2)
```

```
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
```

We can check if the selection and background fitting worked by looking again at the light curve

[9]:
```
fig = tsb.view_lightcurve(-50,150)
```




For the fit we of course want to use all the available detectors. So we first check which detectors were still working at that time.

[10]:
```
from pyspi.utils.livedets import get_live_dets
active_dets = get_live_dets(time=grbtime, event_types=["single"])
print(active_dets)
```

```
[ 0  3  4  6  7  8  9 10 11 12 13 14 15 16 18]
```

Now we loop over these detectors, build the times series, fit the background and construct the SPILikeGRB plugins which we can use in 3ML.

[11]:
```
from pyspi.SPILike import SPILikeGRB
from threeML import DataList
spilikes = []
for d in active_dets:
    drm_generator = ResponseRMFGenerator.from_time(grbtime,
                                                   d,
                                                   ebounds,
                                                   ein,
```

(continues on next page)

(continued from previous page)

```
                                                    rsp_base)
    sd = SPIDRM(drm_generator, ra, dec)
    tsb = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{d}",
                                            d,
                                            grbtime,
                                            response=sd,
                                            sgl_type="both",
                                            )
    tsb.set_active_time_interval(active_time)
    tsb.set_background_interval(bkg_time1, bkg_time2)

    sl = tsb.to_spectrumlike()
    spilikes.append(SPILikeGRB.from_spectrumlike(sl,
                                            free_position=False))
datalist = DataList(*spilikes)
```

```
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 3 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 4 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 6 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 7 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 8 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 9 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 10 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 11 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 12 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 13 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 14 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 15 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 16 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 18 background:   0%|          | 0/29 [00:00<?, ?it/s]
```

Now we have to specify a model for the fit. We use astromodels for this.

[12]:
```
from astromodels import *
pl = Powerlaw()
pl.K.prior = Log_uniform_prior(lower_bound=1e-6, upper_bound=1e4)
pl.K.bounds = (1e-6, 1e4)
pl.index.set_uninformative_prior(Uniform_prior)
pl.piv.value = 200
ps = PointSource('GRB',ra=ra, dec=dec, spectral_shape=pl)

model = Model(ps)
```

Everything is ready to fit now! We make a Bayesian fit here with emcee

[13]:
```
from threeML import BayesianAnalysis
ba_spi = BayesianAnalysis(model, datalist)
ba_spi.set_sampler("emcee", share_spectrum=True)
ba_spi.sampler.setup(n_walkers=20, n_iterations=500)
ba_spi.sample()
```

```
  0%|          | 0/125 [00:00<?, ?it/s]

  0%|          | 0/500 [00:00<?, ?it/s]

Maximum a posteriori probability (MAP) point:


                                                   result             unit
parameter
GRB.spectrum.main.Powerlaw.K      (2.25 +/- 0.04) x 10^-2  1 / (cm2 keV s)
GRB.spectrum.main.Powerlaw.index     -1.014 -0.018 +0.017


Values of -log(posterior) at the minimum:


          -log(posterior)
SPIDet0        -79.708463
SPIDet10       -64.725855
SPIDet11       -68.904660
SPIDet12       -68.001808
SPIDet13       -83.540126
SPIDet14       -73.634753
SPIDet15       -62.837192
SPIDet16       -62.263356
SPIDet18       -75.559807
SPIDet3        -72.583142
SPIDet4        -73.741573
SPIDet6        -59.864307
SPIDet7        -78.342385
SPIDet8        -60.375355
SPIDet9        -59.532740
total        -1043.615524


Values of statistical measures:


      statistical measures
AIC            2091.258825
BIC            2099.381739
DIC            2137.264487
PDIC              1.935718
```

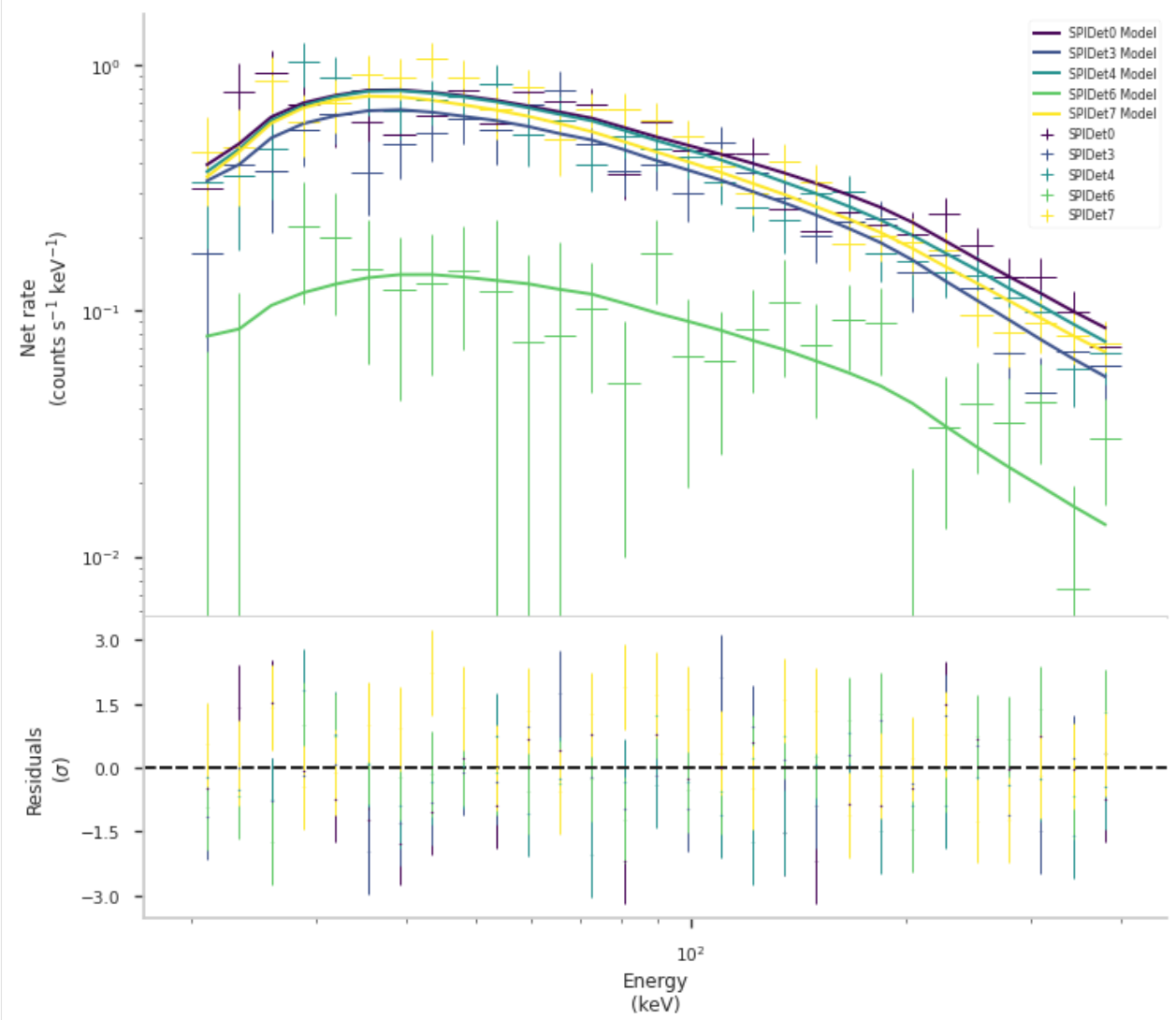
We can inspect the fits with residual plots

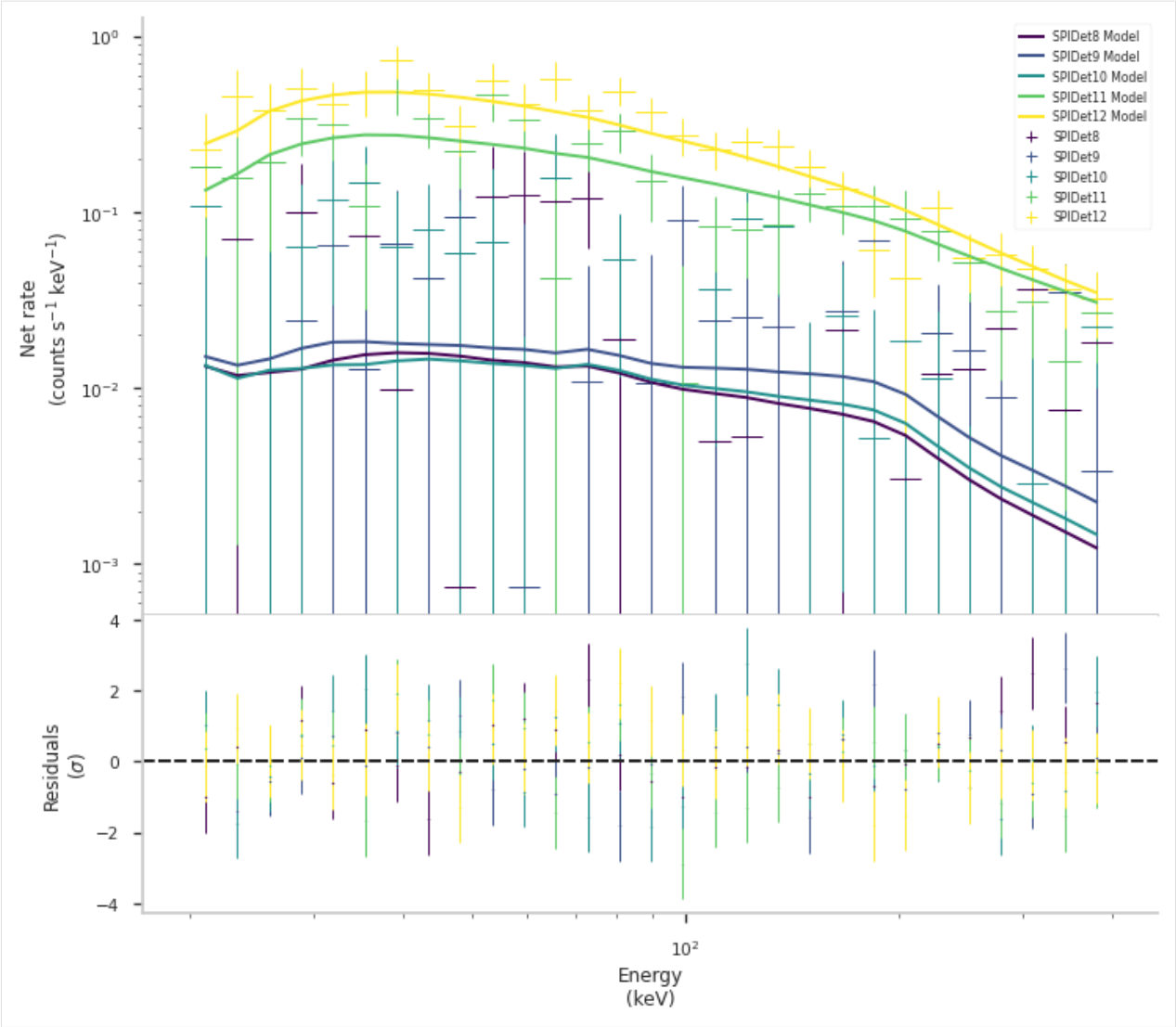
[14]:
```
from threeML import display_spectrum_model_counts
fig = display_spectrum_model_counts(ba_spi,
                                    data_per_plot=5,
                                    source_only=True,
                                    show_background=False,
```

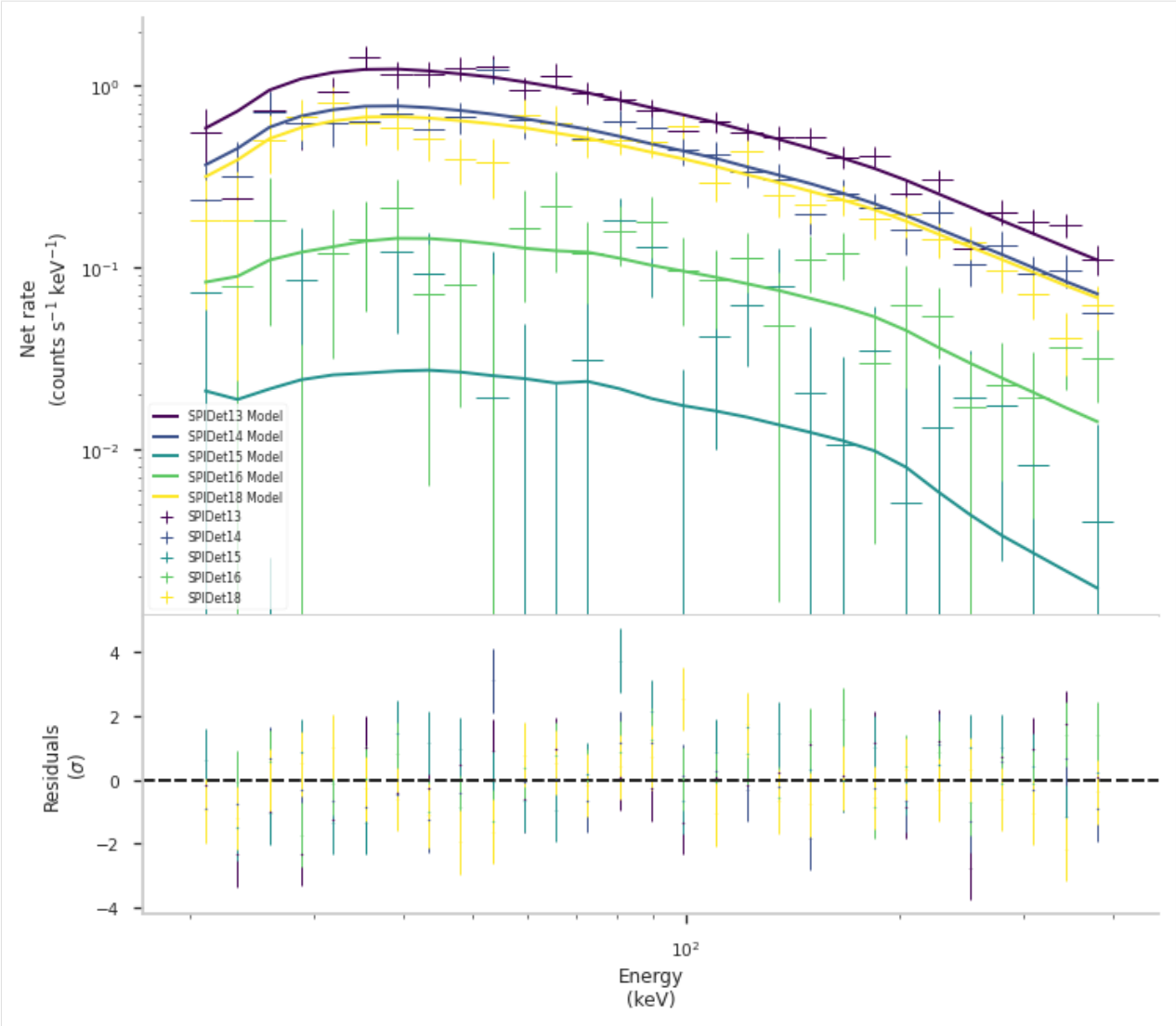
(continues on next page)

(continued from previous page)

```
                    model_cmap="viridis",
                    data_cmap="viridis",
                    background_cmap="viridis")
```




SPIDet8 Model
SPIDet9 Model
SPIDet10 Model
SPIDet11 Model
SPIDet12 Model
SPIDet8
SPIDet9
SPIDet10
SPIDet11
SPIDet12
Net rate
(counts s$^{-1}$ keV$^{-1}$)
$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
Residuals
($\sigma$)
4
2
0
−2
−4
$10^2$
Energy
(keV)

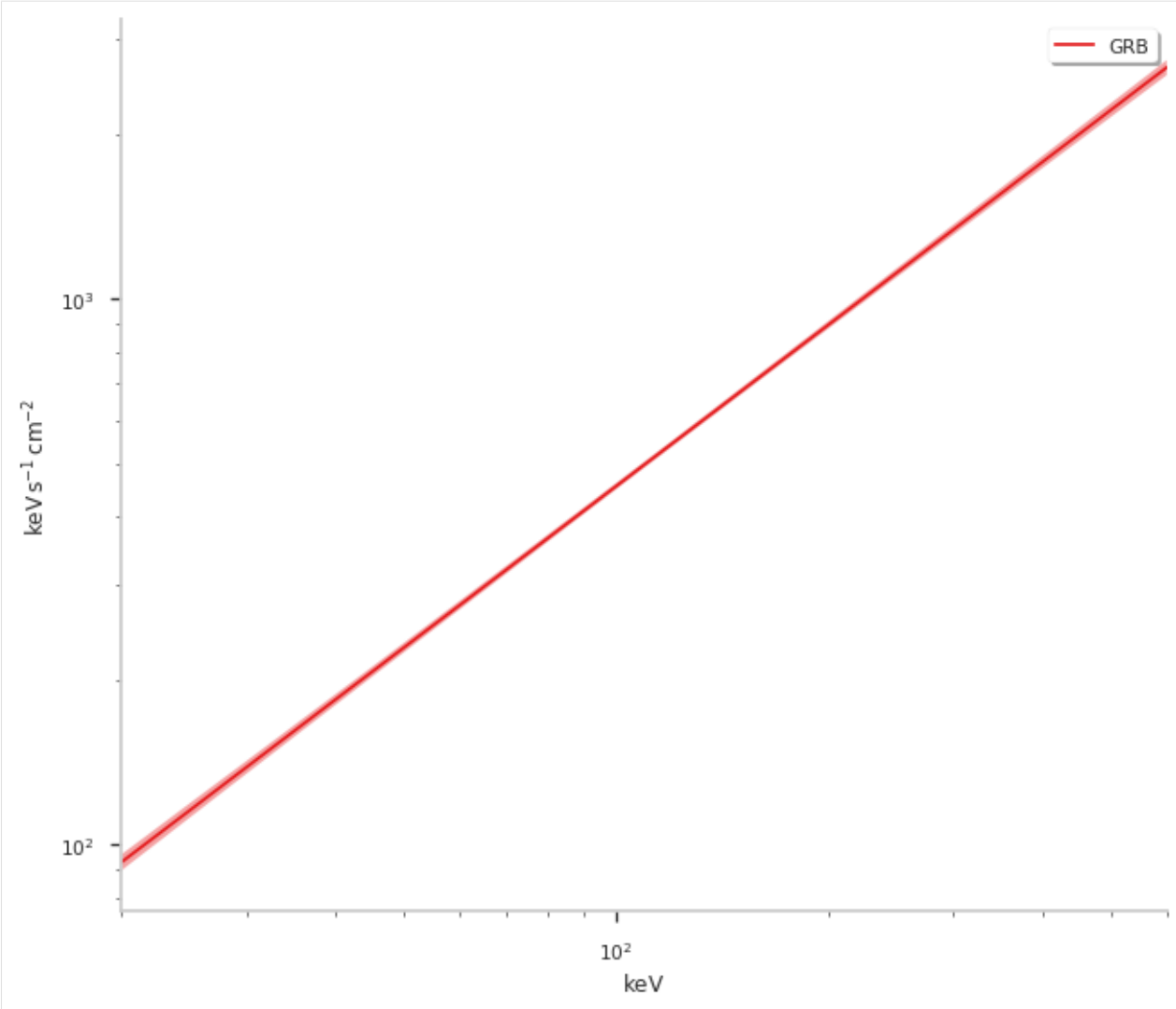



and have a look at the spectrum

[15]:
```
from threeML import plot_spectra
fig = plot_spectra(ba_spi.results, flux_unit="keV/(s cm2)", ene_min=20, ene_max=600)
```

```
processing Bayesian analyses:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
Propagating errors:   0%|          | 0/100 [00:00<?, ?it/s]
```




We can also get a summary of the fit and write the results to disk (see 3ML documentation).

It is also possible to localize GRBs with PySPI, to this we simply free the position of point source with:

[16]:
```
for s in spilikes:
    s.set_free_position(True)
datalist = DataList(*spilikes)
```

Initialize the Bayesian Analysis and start the sampling with MultiNest. To use MultiNest you need to install pymultinest according to its documentation.

[17]:
```
import os
os.mkdir("./chains_grb_example")
ba_spi = BayesianAnalysis(model, datalist)
ba_spi.set_sampler("multinest")
ba_spi.sampler.setup(500,
                     chain_name='./chains_grb_example/docsfit1_',
                     resume=False,
                     verbose=False)
```

(continues on next page)

(continued from previous page)

```
ba_spi.sample()
```

```
Freeing the position of SPIDet0 and setting priors
Freeing the position of SPIDet3 and setting priors
Freeing the position of SPIDet4 and setting priors
Freeing the position of SPIDet6 and setting priors
Freeing the position of SPIDet7 and setting priors
Freeing the position of SPIDet8 and setting priors
Freeing the position of SPIDet9 and setting priors
Freeing the position of SPIDet10 and setting priors
Freeing the position of SPIDet11 and setting priors
Freeing the position of SPIDet12 and setting priors
Freeing the position of SPIDet13 and setting priors
Freeing the position of SPIDet14 and setting priors
Freeing the position of SPIDet15 and setting priors
Freeing the position of SPIDet16 and setting priors
Freeing the position of SPIDet18 and setting priors
 ****************************************************
 MultiNest v3.12
 Copyright Farhan Feroz & Mike Hobson
 Release Nov 2019

 no. of live points =  500
 dimensionality =    4
 ****************************************************
  analysing data from chains_grb_example/docsfit1_.txt ln(ev)=  -1088.6098789639104
↪+/-  0.21907028375800258

 Total Likelihood Evaluations:       291325
 Sampling finished. Exiting MultiNest
Maximum a posteriori probability (MAP) point:
```

```
                                                        result  \
parameter
GRB.position.ra                          (9.466 +/- 0.007) x 10
GRB.position.dec                  (-7.1053 -0.0017 +0.0016) x 10
GRB.spectrum.main.Powerlaw.K             (2.27 +/- 0.04) x 10^-2
GRB.spectrum.main.Powerlaw.index            -1.014 -0.017 +0.018

                                            unit
parameter
GRB.position.ra                              deg
GRB.position.dec                             deg
GRB.spectrum.main.Powerlaw.K      1 / (cm2 keV s)
GRB.spectrum.main.Powerlaw.index
```

```
Values of -log(posterior) at the minimum:
```

```
          -log(posterior)
SPIDet0        -81.578248
SPIDet10       -67.160529
```

(continues on next page)

(continued from previous page)

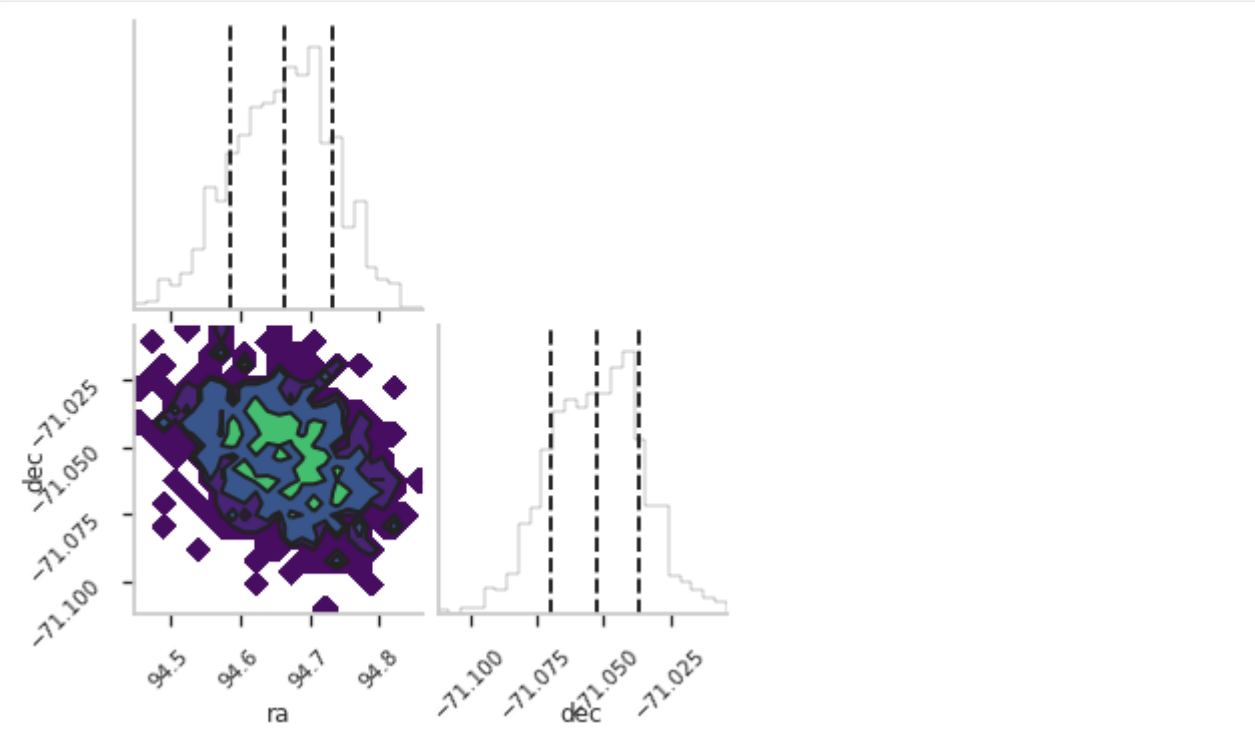
```
SPIDet11       -71.383255
SPIDet12       -69.645227
SPIDet13       -86.448622
SPIDet14       -76.474985
SPIDet15       -65.363899
SPIDet16       -64.161539
SPIDet18       -77.792754
SPIDet3        -76.674979
SPIDet4        -73.979425
SPIDet6        -62.198945
SPIDet7        -78.372394
SPIDet8        -62.911150
SPIDet9        -62.034177
total        -1076.180129
```

```
Values of statistical measures:
```

```
        statistical measures
AIC              2160.453280
BIC              2176.661641
DIC              2134.789897
PDIC                3.876941
log(Z)           -472.777263
```

We can use the 3ML features to create a corner plot for this fit:

[18]:
```
from threeML.config.config import threeML_config
threeML_config.bayesian.corner_style.show_titles = False
fig = ba_spi.results.corner_plot(components=["GRB.position.ra", "GRB.position.dec"])
```




When we compare the results for ra and dec, we can see that this matches with the position from Swift-XRT for the same GRB (RA, Dec = 94.67830, -70.99905)

## 2.11 Fit for the PSD Efficiency

Setup to make the output clean for the docs:

[1]:
```
%%capture
from threeML import silence_logs
import warnings
warnings.filterwarnings("ignore")
silence_logs()
import matplotlib.pyplot as plt
%matplotlib inline
from jupyterthemes import jtplot
jtplot.style(context="talk", fscale=1, ticks=True, grid=False)
```

The first thing we need to do, is to specify the time of the GRB. We do this by specifying a astropy time object or a string in the format YYMMDD HHMMSS.

[2]:
```
from astropy.time import Time
grbtime = Time("2012-07-11T02:44:53", format='isot', scale='utc')
#grbtime = "120711 024453" # works also
```

Now we want to analyze in total the energy between 20 and 2000 keV. So we have to take into account the spurious events in the Non-PSD events (see electronic noise section). For the energy bins up to 500 keV we will use all the single events and from 500 to 2000 keV, we will only use the PSD events.

[3]:
```
import numpy as np
ein = np.geomspace(20,3000,300)
ebounds_sgl = np.geomspace(20,500,30)
ebounds_psd = np.geomspace(500,2000,30)
```

Due to detector failures there are several versions of the response for SPI. Therefore we have to find the version number for the time of the GRB and construct the base response object for this version.

[4]:
```
from pyspi.utils.function_utils import find_response_version
from pyspi.utils.response.spi_response_data import ResponseDataRMF
version = find_response_version(grbtime)
print(version)
rsp_base = ResponseDataRMF.from_version(version)
```

```
4
Using the irfs that are valid between 10/05/27 12:45:00 and present (YY/MM/DD HH:MM:SS)
```

Now we can create the response object for detector 0 and set the position of the GRB, which we already know.

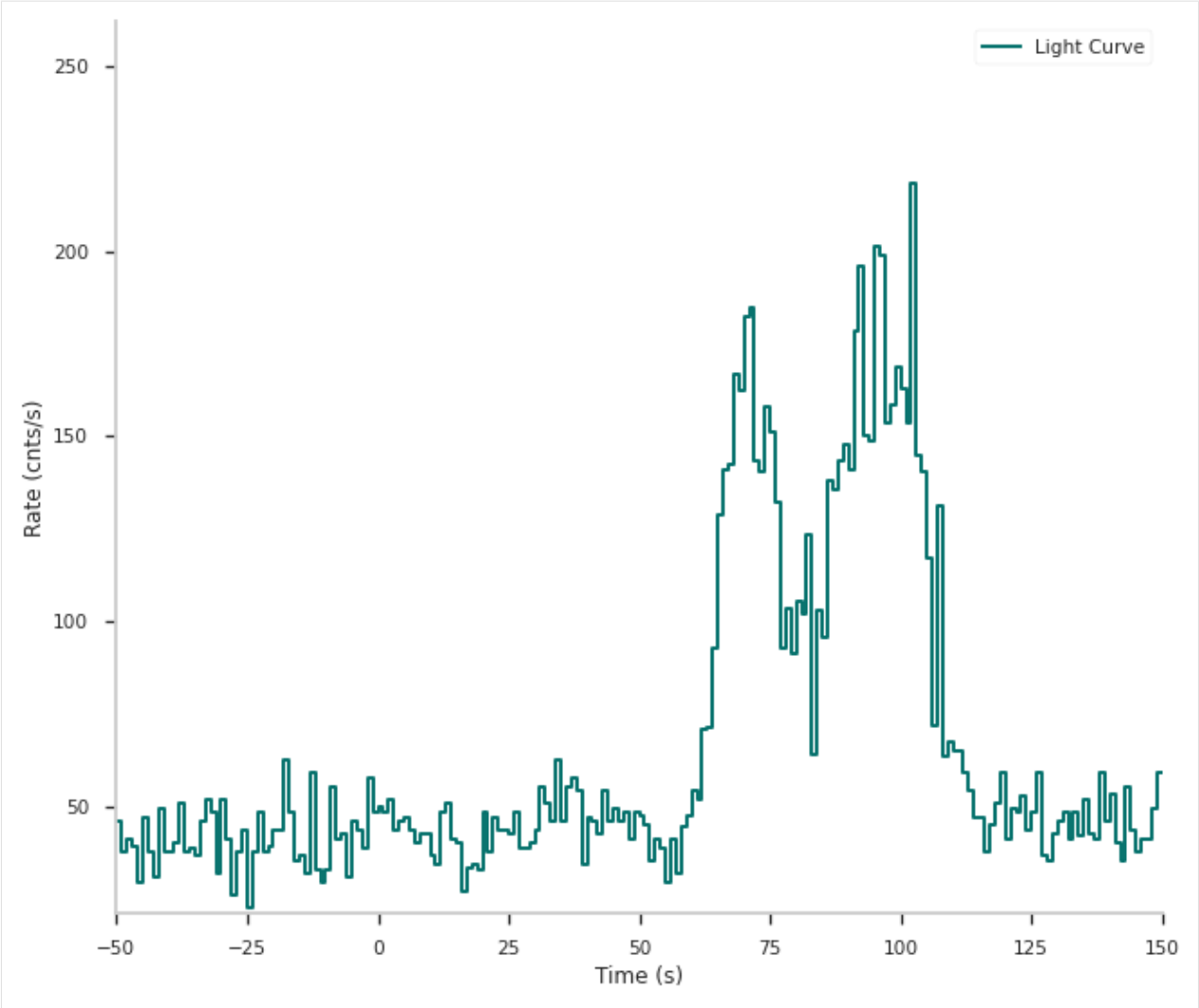
[5]:
```
from pyspi.utils.response.spi_response import ResponseRMFGenerator
from pyspi.utils.response.spi_drm import SPIDRM
det=0
ra = 94.6783
dec = -70.99905
drm_generator_sgl = ResponseRMFGenerator.from_time(grbtime,
                                                    det,
                                                    ebounds_sgl,
                                                    ein,
                                                    rsp_base)
sd_sgl = SPIDRM(drm_generator_sgl, ra, dec)
```

With this we can build a time series and we use all the single events in this case (PSD + non PSD; see section about electronic noise). To be able to convert the time series into 3ML plugins later, we need to assign them a response object.

[6]:
```
from pyspi.utils.data_builder.time_series_builder import TimeSeriesBuilderSPI
tsb_sgl = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{det}",
                                            det,
                                            grbtime,
                                            response=sd_sgl,
                                            sgl_type="both",
                                            )
```

Now we can have a look at the light curves from -50 to 150 seconds around the specified GRB time.

[7]:
```
fig = tsb_sgl.view_lightcurve(-50,150)
```



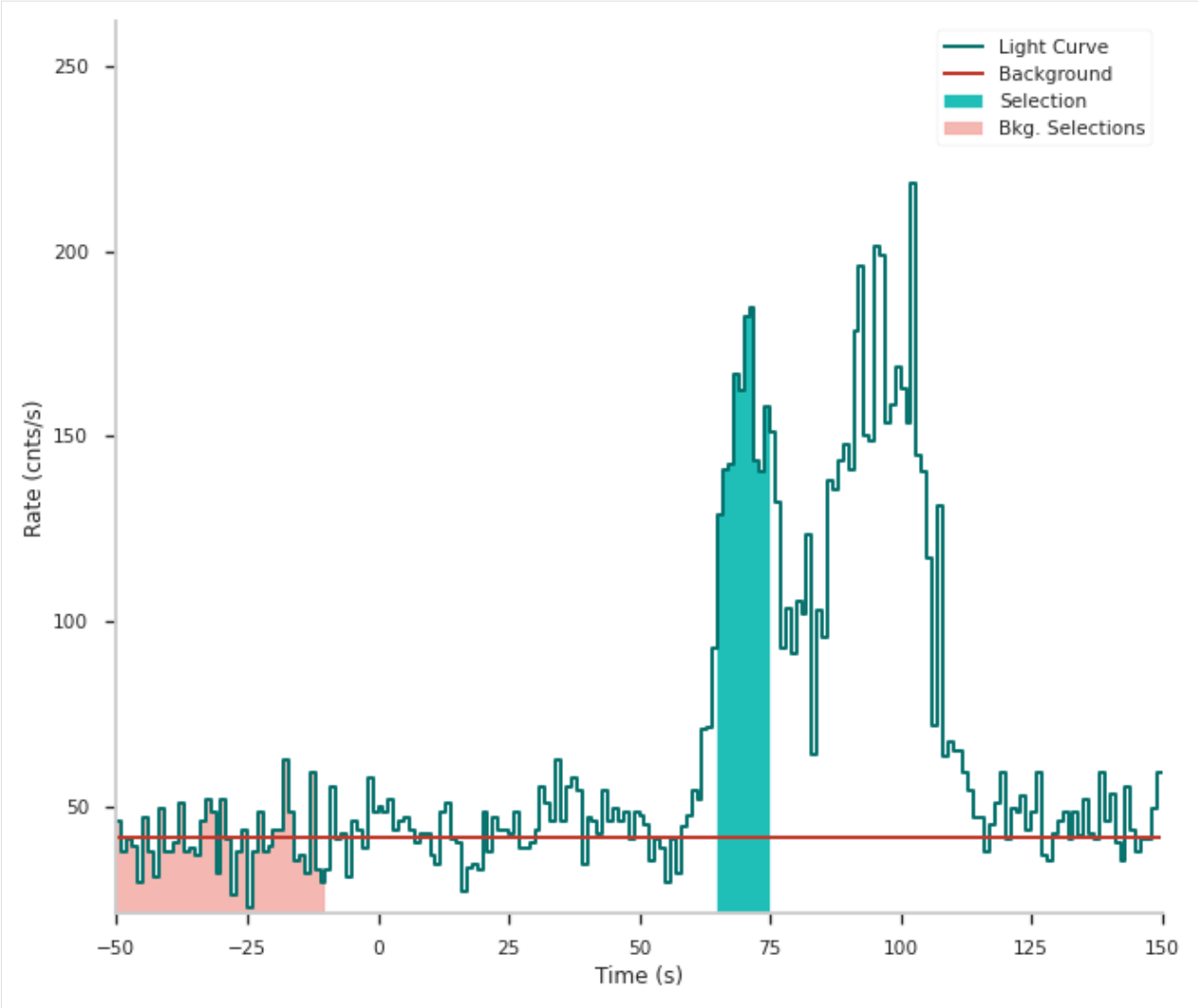

With this we can select the active time and some background time intervals.

[8]:
```
active_time = "65-75"
bkg_time1 = "-500--10"
bkg_time2 = "150-1000"
tsb_sgl.set_active_time_interval(active_time)
tsb_sgl.set_background_interval(bkg_time1, bkg_time2)
```

```
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
```

We can check if the selection and background fitting worked by looking again at the light curve

[9]:
```
fig = tsb_sgl.view_lightcurve(-50,150)
```




In this example we use three detectors (IDs: 0, 3 and 4). For these three detectors we build the times series, fit the background and construct the SPILikeGRB plugins which we can use in 3ML.

[10]:
```
from pyspi.SPILike import SPILikeGRB
from threeML import DataList
spilikes_sgl = []
spilikes_psd = []
for d in [0,3,4]:
    drm_generator_sgl = ResponseRMFGenerator.from_time(grbtime,
                                                        d,
                                                        ebounds_sgl,
                                                        ein,
                                                        rsp_base)
    sd_sgl = SPIDRM(drm_generator_sgl, ra, dec)
    tsb_sgl = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDet{d}",
                                                d,
                                                grbtime,
                                                response=sd_sgl,
                                                sgl_type="both",
```

(continues on next page)

(continued from previous page)

```
                                                )
    tsb_sgl.set_active_time_interval(active_time)
    tsb_sgl.set_background_interval(bkg_time1, bkg_time2)

    sl_sgl = tsb_sgl.to_spectrumlike()
    spilikes_sgl.append(SPILikeGRB.from_spectrumlike(sl_sgl,
                                                     free_position=False))

    drm_generator_psd = ResponseRMFGenerator.from_time(grbtime,
                                                       d,
                                                       ebounds_psd,
                                                       ein,
                                                       rsp_base)
    sd_psd = SPIDRM(drm_generator_psd, ra, dec)
    tsb_psd = TimeSeriesBuilderSPI.from_spi_grb(f"SPIDetPSD{d}",
                                                d,
                                                grbtime,
                                                response=sd_psd,
                                                sgl_type="both",
                                                )
    tsb_psd.set_active_time_interval(active_time)
    tsb_psd.set_background_interval(bkg_time1, bkg_time2)

    sl_psd = tsb_psd.to_spectrumlike()
    spilikes_psd.append(SPILikeGRB.from_spectrumlike(sl_psd,
                                                     free_position=False))

datalist = DataList(*spilikes_sgl, *spilikes_psd)
```

```
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 0 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 3 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 3 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 4 background:   0%|          | 0/29 [00:00<?, ?it/s]
Fitting Detector 4 background:   0%|          | 0/29 [00:00<?, ?it/s]
```

Now we set a nuisance parameter for the 3ML fit. Nuisance parameter are parameters that only affect one plugin. In this case it is the PSD efficiency for every plugin that uses only PSD events. We do not link the PSD efficiencies in this case, so we determine the PSD efficiency per detector.

[11]:
```
for i, s in enumerate(spilikes_psd):
    s.use_effective_area_correction(0,1)
```

Now we have to specify a model for the fit. We use astromodels for this.

[12]:
```
from astromodels import *
band = Band()
band.K.prior = Log_uniform_prior(lower_bound=1e-6, upper_bound=1e4)
band.K.bounds = (1e-6, 1e4)
band.alpha.set_uninformative_prior(Uniform_prior)
```

(continues on next page)

(continued from previous page)

```
band.beta.set_uninformative_prior(Uniform_prior)
band.xp.prior = Uniform_prior(lower_bound=10,upper_bound=8000)
band.piv.value = 200
ps = PointSource('GRB',ra=ra, dec=dec, spectral_shape=band)

model = Model(ps)
```

Everything is ready to fit now! We make a Bayesian fit here with MultiNest. To use MultiNest you need to install pymultinest according to its documentation.

[13]:
```
from threeML import BayesianAnalysis
import os
os.mkdir("./chains_psd_eff")
ba_spi = BayesianAnalysis(model, datalist)
for i, s in enumerate(spilikes_psd):
    s.use_effective_area_correction(0,1)
ba_spi.set_sampler("multinest")

ba_spi.sampler.setup(500,
                     chain_name='./chains_psd_eff/docsfit1_',
                     resume=False,
                     verbose=False)
ba_spi.sample()
```

```
 ****************************************************
 MultiNest v3.12
 Copyright Farhan Feroz & Mike Hobson
 Release Nov 2019

 no. of live points =  500
 dimensionality =    7
 ****************************************************
  analysing data from chains_psd_eff/docsfit1_.txt ln(ev)=  -382.17785337379030      +/-␣
↪ 0.15332793358616456
 Total Likelihood Evaluations:        19921
 Sampling finished. Exiting MultiNest

Maximum a posteriori probability (MAP) point:
```

```
                                          result             unit
parameter
GRB.spectrum.main.Band.K      (2.02 +/- 0.08) x 10^-2  1 / (cm2 keV s)
GRB.spectrum.main.Band.alpha     -1.052 -0.034 +0.033
GRB.spectrum.main.Band.xp        (5.1 +/- 1.9) x 10^3              keV
GRB.spectrum.main.Band.beta              -3.3 +/- 1.2
cons_SPIDetPSD0                 (5.4 +/- 0.7) x 10^-1
cons_SPIDetPSD3                 (6.5 +/- 1.0) x 10^-1
cons_SPIDetPSD4                 (6.1 +/- 0.9) x 10^-1
```

```
Values of -log(posterior) at the minimum:
```

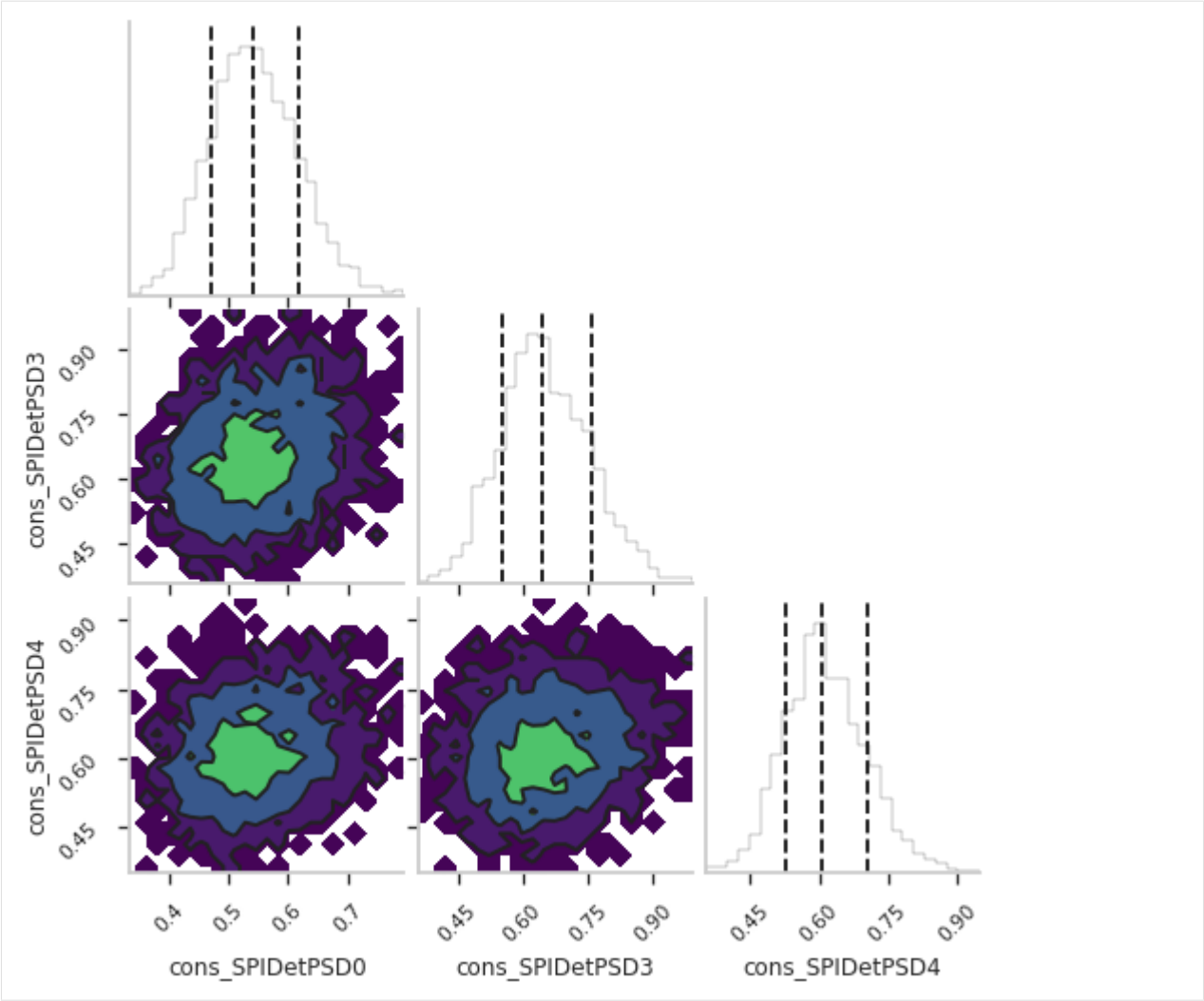
```
            -log(posterior)
SPIDet0          -80.155176
SPIDet3          -77.061524
SPIDet4          -72.049594
SPIDetPSD0       -48.358257
SPIDetPSD3       -39.201423
SPIDetPSD4       -40.868095
total           -357.694068
```

```
Values of statistical measures:
```

```
        statistical measures
AIC               730.062835
BIC               751.501524
DIC               742.371506
PDIC                4.881625
log(Z)           -165.977733
```

We can use the 3ML features to create a corner plot for this fit:

[14]:
```
from threeML.config.config import threeML_config
threeML_config.bayesian.corner_style.show_titles = False
fig = ba_spi.results.corner_plot(components=["cons_SPIDetPSD0", "cons_SPIDetPSD3", "cons_
↪SPIDetPSD4"])
```




So we see we have a PSD efficiency of ~60 +/- 10 % in this case.

# PYTHON MODULE INDEX

## p

# INDEX

## Symbols



## A



## B



## C



## D

## E



## F



## G

## I



## J



## L



## M

## N



## O



## P



## R

## S



## T