XMM: Generate and Analyse (XGA)

David J Turner

Aug 20, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO XGA

XMM: Generate and Analyse (XGA) is a Python module designed to make it easy to analyse X-ray sources that have been observed by the XMM-Newton Space telescope. It is based around declaring different types of source and sample objects which correspond to real X-ray sources, finding all available data, and then insulating the user from the tedious generation and basic analysis of X-ray data products (though with the option to get stuck into the data directly if required).

XGA will generate photometric products and spectra for individual sources, or whole samples, with just a few lines of code. It is not a pipeline in itself, as it can be used for interactive analyses in Jupyter Notebooks, however it is quite possible to build pipeline's using XGA's features and methods. XGA provides an easy to use Python interface with XMM's Science Analysis System (SAS) and XSPEC, where all generation and fitting procedures have been parallelised as much as is possible. A major goal of this module is that you shouldn't need to leave a Python environment at any point during your analysis, as all XMM products and fit results are read into an XGA source storage structure.

This module also supports more complex analyses for specific object types; the easy generation of scaling relations, the measurement of gas masses for galaxy clusters, and the PSF correction of images for instance. It is also possible, for extended sources (such as galaxy clusters), to generate and fit sets of annular spectra. This allows you to investigate how properties change radially with distance from the centre, and enables the measurement of hydrostatic masses of clusters.

While XGA is a piece of open source software, I would appreciate it if any work that makes use of it would cite the Journal of Open Source Software paper that will accompany this package. It is in preparation, but has not yet been submitted, so if you are going to use this module in a piece of work please get in touch with me so we can work something out.

If wish to contribute to XGA, have feature suggestions, or any comments at all, then please go to the "Getting Support" section and submit an issue on GitHub/send me an email, I'll be happy to hear from you!

CHAPTER

TWO

INSTALLING AND CONFIGURING XGA

This is a slightly more complex installation than many Python modules, but shouldn't be too difficult. If you're having issues feel free to contact me.

2.1 Data Required For Using This Module

This is very important - Currently, to make use of this module, you **must** have access to cleaned XMM-Newton event lists, as XGA is not yet capable of producing them itself.

2.2 Region Files

It will be beneficial if you have region files available, as it will allow XGA to remove interloper sources. If you wish to use existing region files, then they must be in a DS9 compatible format, **point sources** must be **red** and **extended sources** must be **green**.

2.3 The Module

XGA is available on PyPi, so you can simply run:

```
pip install xga
```

Alternatively you can fetch the current working version from the git repository, which (as XGA is still in a fairly early stage of development) may have more up-to-date features than the PyPi release:

```
git clone https://github.com/DavidT3/XGA
cd XGA
python setup.py install
```

2.4 Required Dependencies

XGA depends on two non-Python pieces of software:

- XMM's Science Analysis System (SAS) - Version 17.0.0, but other versions should be largely compatible with the software. SAS version 14.0.0 however, does not support features that PSF correction of images depends on.
- HEASoft's XSPEC - Version 12.10.1 - I can't guarantee later versions will work.

All required Python modules can be found in requirements.txt, and should be added to your system during the installation of XGA.

Excellent installation guides for SAS and HEASoft already exist, so I won't go into that here. XGA will not run without detecting these pieces of software installed on your system.

2.5 Optional Dependencies

XGA can also make use of external software for some limited tasks, but they are not required to use the module as a whole:

- The R interpreter.
- Rpy2 - A Python module that provides an interface with the R language in Python.
- LIRA - An R fitting package.

The R interpreter, Rpy2, and LIRA are all necessary only if you wish to use the LIRA scaling relation fitting function.

2.6 Configuring XGA

Before XGA can be used you must fill out a configuration file (a completed example can be found here).

Follow these steps to fill out the configuration file:

1. Import XGA to generate the initial, incomplete, configuration file.
2. Navigate to ~/.config/xga and open xga.cfg in a text editor. The .config directory is usually hidden, so it is probably easier to navigate via the terminal.
3. Take note of the entries that currently have /this/is/required at the beginning, without these entries the module will not function.
4. Set the directory in which XGA will save the products and files it generates. I just set it to xga_output, so wherever I run a script that imports XGA it will create a folder called xga_output there. You could choose to use an absolute path and have a global XGA folder however, it would make a lot of sense.
5. You may also set an optional parameter in the [XGA_SETUP] section, 'num_cores'. If you wish to manually limit the number of cores that XGA is allowed to use, then set this to an integer value, e.g. num_cores = 10. You can also set this at runtime, by importing NUM_CORES from xga and setting that to a value.
6. The root_xmm_dir entry is the path of the parent folder containing all of your observation data.
7. Most of the other entries tell XGA how different files are named. clean_pn_evts, for instance, gives the naming convention for the cleaned PN events files that XGA generates products from.
8. Bear in mind when filling in the file fields that XGA uses the Python string formatting convention, so **anywhere you see {obs_id} will be filled formatted with the ObsID of interest when XGA is actually running**.

9. The lo_en and hi_en entries can be used to tell XGA what images and exposure maps you may already have. For instance, if you already had 0.50-2.00keV and 2.00-10.00keV images and exposure maps, you could set lo_en = ['0.50', '2.00'] and hi_en = ['2.00', '10.00'].
10. Finally, the region_file entry tells XGA where region files for each observation are stored (if they exist).

Disclaimer: If region files are supplied, XGA also expects at least one image per instrument per observation, for WCS information.

I have tried to make this section as general as possible, but I am biased by how my research group generates and stores our data products. If you are an X-ray astronomer who wishes to use this module, but it seems to be incompatible with your setup, please get in touch or raise an issue.

Remote Data Access: If your data lives on a remote server, and you want to use XGA on a local machine, I recommend setting up an SFTP connection and mounting the server as an external volume. Then you can fill out the configuration file with paths going through the mount folder - its how I use it a lot of the time.

2.7 XGA's First Run After Configuration

The first time you import any part of XGA, it will create an 'observation census', where it will search through all the observations it can find (based on your entries in the configuration file), check that there are events lists present, and record the pointing RA and DEC. *This can take a while*, but will only take that long on the first run. The module will check the census against your observation directory and see if it needs to be updated on every run.

2.8 Blacklisting ObsIDs

If you don't wish your analyses to include certain ObsIDs, then you can 'blacklist' them and remove them from all consideration, you simply need to add the ObsID to 'blacklist.csv', which is located in the same directory as the configuration file. If you need to know where this configuration file is located, import CONFIG_FILE from xga.utils.

It is possible that you might want to do this so that ObsIDs with significant problems (flaring, for instance), don't contribute to and spoil your current analysis.

CHAPTER

THREE

TUTORIALS

3.1 Introducing Sources and Samples

In this tutorial, I will explain the basic concepts around which XGA is designed, and why its been set up this way. There won't be much detail on how to use the module for any kind of analysis, but by the end you should have an understanding of how to get started defining sources, then in the next tutorials we can actually start to analyse them.

[1]:
```
from astropy.units import Quantity
import numpy as np
import pandas as pd

# Here we import various types of source class from XGA
from xga.sources import BaseSource, NullSource, ExtendedSource, PointSource,␣
↪GalaxyCluster
# And here we import different of sample class
from xga.samples.extended import ClusterSample
```

3.1.1 What are sources?

XGA revolves around 'source' objects, which are representative of X-ray sources in real life. These are at the heart of any analysis performed with XGA, and just as there are different types of objects that emit X-rays, there are different types of source object built into XGA. We make distinctions between the different types of source due to the different information they can require for their analysis (clusters need overdensity radii for instance, whereas that isn't a useful concept for an AGN). Different source classes also have some different procedures and methods built into them, as we often wish to measure different things for different types of source.

At their most basic, all that is required to define a source is a position on the sky. The first time XGA is run on a new system, it makes an 'observation census' of the data that you have pointed it to, and finds what observations are available and what their pointing coordinates are; when a new source is defined XGA searches through the census to check whether there are data available for the given coordinates, and if there are relevant observations then they are 'associated' with the source object. An observation will be associated with a source if the aimpoint of the observation is within 30 arcminutes of the source coordinates, though some source classes support further cleaning steps to remove observations that don't cover the entire object.

This approach means that the user doesn't have to directly deal with data if they don't want to, XGA will fetch all available data by itself. When it comes to actually analysing and measuring quantities from the source, all the data will be used, not just single observations.

As we ask you to supply region files to XGA in the configuration file, the module can be aware of where other detected sources are in the data its chosen, that allows it to define any 'interloper' sources that have to be excluded from spectrum generation and photometric analysis.

3.1.2 What types of source are there?

- **BaseSource** - The superclass for all the other source classes, and the simplest of them all, there are very few circumstances where this class should be initialised by users. BaseSource only needs an RA and Dec to be initialised, and if a name is not supplied by the user then one will be generated from those coordinates.
- **NullSource** - This class of source is an exception to XGA's design philosophy that an XGA source represents a real X-ray emitting object. By default NullSource associated every available ObsID with itself (though you may specify which ObsIDs to associate with it), and as such shouldn't be used for astrophysical analysis. This class of source should only be used for bulk generation of products such as images and exposure maps.
- **ExtendedSource** - This is a general class for extended X-ray sources, it is also the superclass of the GalaxyCluster class. XGA will attempt to find a matching extended source from the supplied region files, and if it does then that region will be used for any analysis. The user may also supply a custom circular region in which to analyse the object. Unless it is told not to, XGA will also attempt to find the X-ray peak of this extended object.
- **GalaxyCluster** - This class is specifically for the analysis of Galaxy Clusters, and is a subclass of ExtendedSource. Defining an instance of this class **requires** a redshift to be passed, as well as **at least** one overdensity radius (R_{200}, R_{500}, and R_{2500} are supported). Also supports passing weak lensing mass and richness values, for use in multi-wavelength analyses. Point sources close to the centre of the cluster will not be removed, as they could be a misidentified cool core, please see the *_source_type_match* method of the GalaxyCluster class for more information.
- **PointSource** - Similar to the ExtendedSource class in that this is a superclass for more specific point source classes. There are no methods in this class to produce radial plots for instance, as for point-like sources the ideal of a radial profile has very little meaning. When a PointSouce is declared, an attempt will be made to match to a point source in region files, if they are supplied.

If you would like a more specific class implemented for the type of object you're working on, please get in contact with me and I will see what I can do.

3.1.3 What are samples?

An XGA sample is a group of the same type of object that we wish to analyse as a population; for instance you might want to analyse multiple Galaxy Clusters and derive a scaling relation from them.

There is a secondary benefit to using a sample object rather than multiple Source objects, a sample can be passed into any function that will accept a source, and the function will perform its job on every source in it. This not only makes writing your code easier and cleaner, but can also be more efficient when running SAS and XSPEC, as XGA will run any such jobs in parallel, rather than you having to run them sequentially in a loop for instance.

When a sample is declared (other than the BaseSample class), it will make sure that images and exposure maps for all associated observations are generated for all constinuent sources exist, and if they don't then they will be generated.

3.1.4 What types of sample are there?

These mostly mirror the types of source that are present in XGA. Specific sample types can have properties or methods unique to that type of astrophysical object.

- **BaseSample** - The superclass for all the other sample classes, there are very few circumstances where this class should be initialised by users. All a BaseSample requires to be instantiated are two numpy arrays, containing RA and Dec values. Arrays of names and redshifts may also be supplied (though names supplied to sample definitions **must be unique**).
- **PointSample** - For a population of some generic type of point source. Again only RA and Dec values have to be supplied, though redshift information can be provided if available. As this is a general point source class, no methods to generate scaling relations have been provided.

- **ClusterSample** - For a population of Galaxy Clusters. Just as with the GalaxyCluster source class, here we **require** that redshift and overdensity radius information be provided on declaration. Many convenient features have been added to this sample class, for instance you can retrieve temperatures of all clusters in the sample (if measured) using a ClusterSample method. You can also easily generate common scaling relations by calling methods of the ClusterSample class, using several different fitting methods.

3.1.5 Defining your first source

Here I demonstrate just how simple it is to define a PointSource object, all I've done is to supply the Right Ascension and Declination of Castor, a famous sextuple star system that emits in X-ray. **All coordinates used with XGA must be passed as decimal degrees, sexagesimal coordinates are not supported by this module.**

PointSource also accepts various other keyword arguments that you may wish to change from defaults, please see *this documentation* for a full list. A particularly useful keyword argument is cosmology, which allows you to pass an Astropy cosmology object, which will then be used in all aspects of analysis; the default cosmology is currently Planck15 - this ability to set the cosmology is present in all source and sample objects, and is used throughout any analysis done with that source/sample.

[2]:
```
demo_src = PointSource(113.65833, 31.87083, name='Castor')
```
```
Generating products of type(s) ccf: 100%|| 3/3 [00:18<00:00,  6.05s/it]
```

We can see from the progress bar above that XGA has detected that no appropriate XMM calibration files were available for the data associated with Castor, and so it used its SAS interface to automatically create the necessary files. It will also have automatically generated combined images and exposure maps, using all available data.

We can now use the info() method to see a summary of the information we have about this particular source. You will notice that XGA has used the input coordinates to find an nH value (using a HEASoft tool). You may also notice that a custom region radius of 0.01 degrees has been used to calculate a SNR value, this is the default region radius for the PointSource class, and may be changed using a keyword argument when the PointSource is defined.

[3]:
```
demo_src.info()
```
```
------------------------------------------------------
Source Name - Castor
User Coordinates - (113.65833, 31.87083) degrees
X-ray Peak - (113.65833, 31.87083) degrees
nH - 0.0446 1e+22 / cm2
XMM ObsIDs - 3
PN Observations - 3
MOS1 Observations - 3
MOS2 Observations - 3
On-Axis - 3
With regions - 3
Total regions - 200
Obs with one match - 3
Obs with >1 matches - 0
Images associated - 18
Exposure maps associated - 18
Combined Ratemaps associated - 1
Spectra associated - 0
------------------------------------------------------
```

A lot of information is stored in all source objects, and I would advise you look at the BaseSource API documentation (or use the dir() command on any source object) to explore what it can tell you.

Here I demonstrate how easy it is to retrieve simple information such as the hydrogen column density at the source coordinates, which ObsIDs are associated with the source, and which ObsIDs are considered 'on-axis' observations. The `instruments` property provides a dictionary with the associated ObsIDs as keys, and the instruments associated with them as values; this is necessary because we cannot take for granted that all observations have data from all cameras.

[4]:
```
# This property returns an astropy quantity with the hydrogen column density
print(demo_src.nH)

# This property is just the ObsIDs associated with the source
print(demo_src.obs_ids)

# However this property returns a dictionary of ObsIDs and which of their instruments␣
↪are valid
print(demo_src.instruments)

# And finally we can easily see which observations are considered on-axis
print(demo_src.on_axis_obs_ids)
```

```
0.0446 1e+22 / cm2
['0123710201', '0123710101', '0112880801']
{'0123710201': ['pn', 'mos1', 'mos2'], '0123710101': ['pn', 'mos1', 'mos2'],
↪'0112880801': ['pn', 'mos1', 'mos2']}
['0123710201', '0123710101', '0112880801']
```

3.1.6 Defining your first sample

This is a simple demonstration of how you can define a sample of GalaxyClusters, with four clusters from the XCS-SDSS sample (Giles et al. (in prep)).

First I create a Pandas dataframe, simply because its a convenient way to store the initial sample data (you don't have to use it), and because I often read in samples using Pandas. Then it is as simple as passing the different columns into the ClusterSample class. Note that the radii are supplied as Astropy quantities - I use quantities throughout XGA, most values that have a unit will be one.

[5]:
```
column_names = ['name', 'ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']
cluster_data = np.array([['XCSSDSS-124', 0.80057775, -6.0918182, 0.251, 1220.11, 1777.
↪06, 109.55, 4.49],
                         ['XCSSDSS-2789', 0.95553986, 2.068019, 0.11, 1039.14, 1519.
↪79, 38.90, 2.83],
                         ['XCSSDSS-290', 2.7226392, 29.161021, 0.338, 935.58, 1359.37,
↪ 105.10, 5.99],
                         ['XCSSDSS-134', 4.9083898, 3.6098177, 0.273, 1157.04, 1684.
↪15, 108.60, 4.79]])

sample_df = pd.DataFrame(data=cluster_data, columns=column_names)
sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']] = \
    sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']].
↪astype(float)

sample_df
```

[5]:

	name	ra	dec	z	r500	r200	richness	\
0	XCSSDSS-124	0.800578	-6.091818	0.251	1220.11	1777.06	109.55	
1	XCSSDSS-2789	0.955540	2.068019	0.110	1039.14	1519.79	38.90	
2	XCSSDSS-290	2.722639	29.161021	0.338	935.58	1359.37	105.10	
3	XCSSDSS-134	4.908390	3.609818	0.273	1157.04	1684.15	108.60	

(continues on next page)

(continued from previous page)

```
   richness_err
0          4.49
1          2.83
2          5.99
3          4.79
```

Just as with the definition of the PointSource object, there are many keyword arguments that can be supplied here, and I recommend examining the *documentation* to see whether you need any of the other options.

It is not necessary to supply two different overdensity radii (as I have done here), but can be very useful. The richness information I passed in is also not needed to define the sample. Remember that you **must** pass redshift information to define GalaxyCluster objects, and as such redshift information is required for ClusterSample objects as well.

Note that this ClusterSample definition generates some images and exposure maps - those are combined images and exposure maps, and they have to exist for the individual GalaxyCluster objects to perform peak finding on the data.

[6]:
```
demo_smp = ClusterSample(sample_df["ra"].values, sample_df["dec"].values, sample_df["z
↪"].values,
                         sample_df["name"].values, r200=Quantity(sample_df["r200"].
↪values, "kpc"),
                         r500=Quantity(sample_df["r500"].values, 'kpc'),␣
↪richness=sample_df['richness'].values,
                         richness_err=sample_df['richness_err'].values)
```

```
Declaring BaseSource Sample: 100%|| 4/4 [00:02<00:00,  1.93it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:18<00:00,  4.54s/it]
Generating products of type(s) image: 100%|| 4/4 [00:02<00:00,  1.61it/s]
Generating products of type(s) expmap: 100%|| 4/4 [00:00<00:00,  6.38it/s]
Setting up Galaxy Clusters: 100%|| 4/4 [00:06<00:00,  1.58s/it]
```

All sample classes have an `info()` method, just like sources, though not as much information is included as in the source `info()` methods. It is simple to retrieve properties for all sources, such as name and redshift.

[7]:
```
demo_smp.info()

print(demo_smp.names)
print(demo_smp.redshifts)
```

```
------------------------------------------------------
Number of Sources - 4
Redshift Information - True
------------------------------------------------------

['XCSSDSS-124' 'XCSSDSS-2789' 'XCSSDSS-290' 'XCSSDSS-134']
[0.251 0.11  0.338 0.273]
```

3.1.7 Interacting with a source object in a sample

Just as with many Python objects (lists, dictionaries, etc.), a sample can be indexed to retrieve individual elements from the whole (in this case individual source objects). What is slightly different about XGA sample objects is that you may use an integer value or a name to retrieve the specific source object you want:

[8]:
```
# Looking at the first source stored in the sample
chosen_src = demo_smp[0]
# And printing its name
print(chosen_src.name)

# Now showing that the name of a source can also be used to retrieve the object
chosen_src = demo_smp['XCSSDSS-124']
print(chosen_src.name)
```

```
XCSSDSS-124
XCSSDSS-124
```

3.1.8 Removing an ObsID from a source object

It is possible that you may want to remove an ObsID from a source object, and as such throw away the data associated with that ObsID. In XGA this is called disassociating, and you can choose to remove as many ObsIDs as you like, or even individual instruments from an ObsID.

You may pass a string if you just wish to remove one ObsID, or a dictionary if you wish to be more precise.

[9]:
```
# This removes all reference to observation 0123710201 from the source object
demo_src.disassociate_obs('0123710201')

# This, however, removes only the MOS1 and MOS2 data from observation 0112880801
demo_src.disassociate_obs({'0112880801': ['mos1', 'mos2']})
```

Looking at the source summary and instruments property again to confirm that we have removed data.

[10]:
```
demo_src.info()

print(demo_src.instruments)
```

```
------------------------------------------------------
Source Name - Castor
User Coordinates - (113.65833, 31.87083) degrees
X-ray Peak - (113.65833, 31.87083) degrees
nH - 0.0446 1e+22 / cm2
XMM ObsIDs - 2
PN Observations - 2
MOS1 Observations - 1
MOS2 Observations - 1
On-Axis - 2
With regions - 2
Total regions - 170
Obs with one match - 2
Obs with >1 matches - 0
Images associated - 8
Exposure maps associated - 8
Combined Ratemaps associated - 0
Spectra associated - 0
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------

{'0123710101': ['pn', 'mos1', 'mos2'], '0112880801': ['pn']}
```

3.1.9 Removing a source from a sample

You may also wish to remove a source from a sample, which is even easier than removing observations from sources. We can simply use the Python 'del' operator, identifying the source to be removed either with its name, or with its index in the sample object.

[11]:
```
# Here we remove a source using its name
del demo_smp['XCSSDSS-124']

# But we can also remove a source just using an index
del demo_smp[2]
```

And if we look again at the sources included in this sample, we can see two have been removed

[12]:
```
demo_smp.info()

print(demo_smp.names)
print(demo_smp.redshifts)
```

```
------------------------------------------------------
Number of Sources - 2
Redshift Information - True
------------------------------------------------------

['XCSSDSS-2789' 'XCSSDSS-290']
[0.11  0.338]
```

3.2 Introducing XGA Products

In this tutorial I will briefly touch on XGA's product classes, which are a key part of the internal makeup of the module. They act as an interface between XGA's functions and the data in different types of X-ray data (images, exposure maps, spectra etc.), and contain many useful and convenient methods for analysis and the extraction of information.

I will not be demonstrating specific abilities of different product types in this tutorial, there are too many to go into here, but I will give an overview of the purpose and important abilities of all the product classes built into XGA.

[1]:
```
from astropy.units import Quantity

from xga.sources import GalaxyCluster
```

3.2.1 What are products?

Most XGA products are what we use to wrap various types of data produced by XMM's SAS routines, as well as providing various product specific functionality. Generally, the user is unlikely to ever define a product instance themselves, internal methods in a source object and the SAS wrapper functions will define, check, and store the products. Those products that are generated by XGA's SAS interface have an added ability to parse the stderr output of the SAS routines used to generate them, then flag any recognised errors (by comparing to an archive of known SAS errors).

There are some XGA products that are not wrappers for SAS generated data products, but are purely for storing and providing access to XGA generated data. These are not based on the same superclass as the other products, but are stored in the same way.

Products instantiated by XGA are immediately 'associated' with a source object, though beyond storing the name of the source in the product's internal structure, they have no knowledge or awareness of the XGA source and its properties. This was by design, to make sure that any functionality built into a product would work regardless of the type of XGA source, and even if it was defined independently of any source at all.

3.2.2 What types of product are there?

- **BaseProduct** - The superclass for many of the standard XGA product classes, there isn't really any reason for a BaseProduct to be declared.
  - **EventList** - A very simple product class which differs only slightly from the BaseProduct class, its only used to store path and header information for the XMM event lists that the configuration file points XGA at.
  - **Image** - An extremely useful product with many extra features, it is used to wrap fits images. The data and header information are read into memory (when required), and can be accessed with properties and attributes of an Image instance. A view method (to look at the image and overlay extra information), and a coordinate transformation method are examples of the built-in functionality.
    - **ExpMap** - A subclass of the XGA Image class, this is a very simple extension to Image that adds a method to easily retrieve an exposure time at a given angular or pixel coordinate.
    - **RateMap** - The class that most photometric analyses are based around, also a subclass of Image. Instantiating this class requires the user pass matching Image and ExpMap instances (same ObsID, instrument etc.) A count rate map is then calculated from this information. This has several added methods, including the ability to retrieve a count rate at a given coordinate, and different peak finding methods.
    - **PSF** - This wraps two-dimensional PSF images generated by a routine such as psfgen, and can be used to PSF correct images and ratemaps. Few methods have been added, and it is unlikely a user will ever need to interact directly with this. The added functionality here is the ability to resample the PSF at a scale provided by a passed in Image object.
  - **Spectrum** - A complicated sub-class of BaseProduct, it wraps and stores a spectrum generated by a SAS routine, including storing paths to all of the other files necessary to analyse it (e.g. RMF, ARF, background spectrum). When a source that has had spectra associated with it is passed to an XGA XSPEC function, the fit results are added to the spectrum objects. This has methods to retrieve fit results, as well as view the fitted Spectrum.
- **BaseAggregateProduct** - Another base product class, this one is designed to store a group of related XGA products.
  - **PSFGrid** - An XGA object used during the PSF correction of images and ratemaps, it stores a grid of PSF objects generated at different spatial positions on the XMM detectors. It provides access to those PSFs on request from the PSF correction function, and context as to which PSF was generated at which position.

  - **AnnularSpectra** - This holds sets of XGA Spectrum products generated in concentric annuli, and includes various methods for accessing the individual spectra, retrieving fit information (if a fit has been run), and viewing the spectra (both for individual annuli, and the whole set). This class is crucial for the measurement of galaxy cluster temperature profiles, and by extension the measurement of hydrostatic mass profiles.

- **BaseProfile1D** - Here we move to the special products that don't wrap existing X-ray data products, these are entirely generated by XGA. This class is designed to store, fit, and generate plots of different 1D profiles. The user should never declare an instance of this class, only the specific subclass that they need for their analysis.
  - **SurfaceBrightness1D** - Mostly meant for Galaxy Clusters, this class will store a 1D surface brightness profile, and enable the fitting of valid models such as a beta profile, or double beta profile.
  - **GasDensity1D** - This is meant to store a gas density profile as calculated by XGA, and includes methods to calculate a total gas mass within a given radius, as well as to generate a gas mass profile.
  - **GasMass1D** - A class for gas mass 1D profiles, which currently has no extra functionality over BaseProfile1D.
  - **ProjectedGasTemperature1D** - A class for the projected temperature profiles which are measured by fitting plasma emission models to annular spectra. They are 'projected' because they are a combination of temperatures of the 3D shells which are intersected along the line of sight by the annulus.
  - **APECNormalisation1D** - A class for storing profiles of the normalisation of the APEC plasma emission model, which is extracted from the same fitting process (run on an AnnularSpectra) that produces the projected temperature profiles. This profile can be used to measure the 3D density profile, and (when converted to an emission measure profile and combined with knowledge of the projected temperature profile) allows us to infer the 3D temperature profile.
  - **EmissionMeasure1D** - Calculated from an APECNormalisation1D, and knowledge of the cosmology and the redshift of the source. The emission measure profile can be used to help infer the 3D temperature profile of a cluster, when combined with the projected temperature profile and assumptions about the source geometry.
  - **ProjectedGasMetallicity1D** - Another profile that *can* be measured from the fitting of AnnularSpectra, though only if metallicity is allowed to vary as a free parameter. Again it is 'projected' because the metallicities are a combination of the metallicities of the 3D shells intersected along the line of sight by the annuli.
  - **GasTemperature3D** - A three-dimensional radial map of the plasma temperature of the intra-cluster medium of a galaxy cluster. This can be used, in combination with knowledge of the 3D gas density, to measure a mass profile for a galaxy cluster.
  - **HydrostaticMass** - Defined with a gas density profile and a 3D temperature profile, this type of profile describes the change of the total mass contained within a radius, and has methods to measure a mass at whatever radius the user wants to.
  - **BaryonFraction** - Can be generated by a HydrostaticMass profile, this shows the change in total baryon fraction within a radius, with radius. Again the value at a specific point can be calculated using a method implemented in this class.

- **BaseAggregateProfile1D** - This is unlikely ever to actually have any specific subclasses, as awareness of the type of profile is not really necessary for its only job, which is to display multiple profiles on one axis. These can be generated just by adding multiple profile products together with the standard Python + operator

- **ScalingRelation** - The scaling relation products are unique in XGA, in that they are the only products that cannot be stored within a source, as they concern multiple sources (or no sources at all if declared from the literature), it would not make sense. These are produced by the scaling relation generation functions built into XGA, and in that case would contain both data and a fitted model. There are also several scaling relations from

literature defined in XGA, these only contain the fitted model. A view method capable of producing publication quality plots is provided, along with many other convenient methods.

- **AggregateScalingRelation** - Just as the BaseAggregateProfile1D class was created purely to view multiple profiles on the same axes, this class is designed to enable scaling relations to be easily combined and viewed together. This class is instantiated by adding multiple ScalingRelation objects together.

3.2.3 Generating Products

This tutorial will not touch on how to generate the product objects, this will be left to more subject specific tutorials such as "Photometry with XGA" and "Spectroscopy with XGA".

However, I do wish to say that any SAS data products generated by XGA are stored in the directory pointed to by the **xga_save_path** entry in the configuration file (which can be either an absolute or relative path). By default a source will load in any product previously generated for it, to save the computational expense of re-generating products that already exist, this behaviour can be turned off if the user sets the **load_products** keyword argument to False when defining a source object.

3.2.4 How products are stored in source objects

Here I will demonstrate how XGA stores product objects within a source, and how you can retrieve them if you wish to interact with the products directly. I will be demonstrate how to associate a product with a source object, as this is an internal mechanism with various checks and validation steps that the user should never have to interact with.

I'll define a GalaxyCluster source object corresponding to my favourite cluster, Abell 907 (**please note that the overdensity radii and the redshift that I've used here are approximate and should not be used for a scientific analysis**). We can see that this cluster is quite well observed, with three ObsIDs associated, with all instruments valid for each observation.

[2]:
```
src = GalaxyCluster(149.59209, -11.05972, 0.16, r500=Quantity(1200, 'kpc'),␣
↪r200=Quantity(1700, 'kpc'),
                    name="A907")
src.info()
```

```
--------------------------------------------------
Source Name - A907
User Coordinates - (149.59209, -11.05972) degrees
X-ray Peak - (149.59251340970866, -11.063958320861634) degrees
nH - 0.0534 1e+22 / cm2
Redshift - 0.16
XMM ObsIDs - 3
PN Observations - 3
MOS1 Observations - 3
MOS2 Observations - 3
On-Axis - 3
With regions - 3
Total regions - 69
Obs with one match - 3
Obs with >1 matches - 0
Images associated - 18
Exposure maps associated - 18
Combined Ratemaps associated - 1
Spectra associated - 0
R200 - 1700.0 kpc
R200 SNR - 230.23
```

(continues on next page)

(continued from previous page)

```
R500 - 1200.0 kpc
R500 SNR - 251.61
------------------------------------------------------
```

Products associatied with a source are stored in a protected attribute of the source, src._products. This essentially consists of a series of nested dictionaries, with the top level keys being the associated ObsIDs (any products that are a combination of multiple ObsIDs are stored under the 'combined' key). The user should **never** interact directly with this protected attribute, but I display below the current contents of this source's storage structure - this will expand as further analyses are performed on the source, with things like spectra and PSF corrected images also being stored here.

You will notice that ratemap objects are already present, that is because XGA checks for matching image-expmap pairs every time an image or expmap is assigned to it; if it finds a matching pair with no corresponding ratemap, it generates one automatically.

[3]:
```
src._products
```

[3]:
```
{'0404910601': {'pn': {'events': <xga.products.misc.EventList at 0x7f309d948220>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d948ca0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d948160>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d948cd0>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d948d60>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d948d00>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d948820>}},
  'mos1': {'events': <xga.products.misc.EventList at 0x7f309d948bb0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d948e80>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d948e50>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d9487c0>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d948f70>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d948f40>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d948e20>}},
  'mos2': {'events': <xga.products.misc.EventList at 0x7f309d948af0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d948c10>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e10d0>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1070>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e11c0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1190>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e10a0>}}},
 '0201901401': {'pn': {'events': <xga.products.misc.EventList at 0x7f309d948be0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e12e0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e12b0>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1130>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e13d0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e13a0>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1280>}},
  'mos1': {'events': <xga.products.misc.EventList at 0x7f309d948fa0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e14f0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e14c0>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d9481c0>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e1580>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1490>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1340>}},
  'mos2': {'events': <xga.products.misc.EventList at 0x7f309d948a60>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e16a0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1670>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1460>},
```

(continues on next page)

(continued from previous page)

```
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e1790>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1760>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1640>}}},
 '0201903501': {'pn': {'events': <xga.products.misc.EventList at 0x7f309de2a0a0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e18b0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1880>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1610>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e19a0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1970>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1850>}},
  'mos1': {'events': <xga.products.misc.EventList at 0x7f309d936fa0>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e1ac0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1a90>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1820>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e1bb0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1b80>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1a60>}},
  'mos2': {'events': <xga.products.misc.EventList at 0x7f309d8e1700>,
   'bound_0.5-2.0': {'image': <xga.products.phot.Image at 0x7f309d8e1cd0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1ca0>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1250>},
   'bound_2.0-10.0': {'image': <xga.products.phot.Image at 0x7f309d8e1dc0>,
    'expmap': <xga.products.phot.ExpMap at 0x7f309d8e1d90>,
    'ratemap': <xga.products.phot.RateMap at 0x7f309d8e1c70>}}},
 'combined': {'bound_0.5-2.0': {'combined_image': <xga.products.phot.Image at␣
↪0x7f309d7bbd90>,
   'combined_expmap': <xga.products.phot.ExpMap at 0x7f309d7c6a00>,
   'combined_ratemap': <xga.products.phot.RateMap at 0x7f309d7b6670>}}}
```

Once we get below the top level of the storage structure, things are slightly different - any entry stored under an ObsID will be another dictionary, with keys corresponding to the available instruments. As you might expect I store data products from the different XMM instruments under these different keys.

Below that level (in a specific ObsID-Instrument dictionary) we start to find keys for specific types of product. Products that are generated within a specific energy range are stored under 'bound_lower-upper' keys, where lower is the lower energy limit in keV and upper is the upper energy limit in keV. These products will include images, expmaps, and ratemaps, which are all stored under keys that match the name of their product classes.

The top level combined key is structured nearly identically, but skips out the dictionary layer that uses instruments as key names. Once you get into an energy bound sub-dictionary however, you will notice that combined products have a prefix on their class keys, 'combined_' is added to the type of the object, e.g. 'combined_image'.

[4]:
```
# Showing the instrument dictionary layer for observation 0404910601
print(src._products['0404910601'].keys(), '\n')

# And the layer where energy bound and non-energy bound products are stored␣
↪separately.
print(src._products['0404910601']['pn'].keys(), '\n')

# Here we can see how images, expmaps, and ratemaps are stored in an energy bound sub-
↪dictionary
print(src._products['0404910601']['pn']['bound_0.5-2.0'].keys(), '\n')

print(src._products['combined']['bound_0.5-2.0'])
```

```
dict_keys(['pn', 'mos1', 'mos2'])
```

(continues on next page)

(continued from previous page)

```
dict_keys(['events', 'bound_0.5-2.0', 'bound_2.0-10.0'])

dict_keys(['image', 'expmap', 'ratemap'])

{'combined_image': <xga.products.phot.Image object at 0x7f309d7bbd90>, 'combined_
↪expmap': <xga.products.phot.ExpMap object at 0x7f309d7c6a00>, 'combined_ratemap':
↪<xga.products.phot.RateMap object at 0x7f309d7b6670>}
```

3.2.5 A general method for retrieving products

To actually retrieve the product objects I have implemented a *get_products()* method. To use this method you need to specify the type of object to retrieve, (e.g. 'image', o), the specific ObsID and instrument (if these options are not specified then the method will retrieve any product that matches the other criteria you provide).

You can also pass an 'extra key', which would be something like a specific bound energy key; this, and knowing the key that corresponds to the type of product you wish to retrieve, are the hardest parts of using this method, and the reason I will be introducing separate methods for specific product types.

The final keyword argument taken by get_products is 'just_obj', which tells the method if you would just like a list of product objects, or if you'd like a list of lists which contained the key path a particular product was stored under.

If you just wanted to retrieve all images then you could call the method like this:

[5]:
```
src.get_products('image')
```

[5]:
```
[<xga.products.phot.Image at 0x7f309d948ca0>,
 <xga.products.phot.Image at 0x7f309d948d60>,
 <xga.products.phot.Image at 0x7f309d948e80>,
 <xga.products.phot.Image at 0x7f309d948f70>,
 <xga.products.phot.Image at 0x7f309d948c10>,
 <xga.products.phot.Image at 0x7f309d8e11c0>,
 <xga.products.phot.Image at 0x7f309d8e12e0>,
 <xga.products.phot.Image at 0x7f309d8e13d0>,
 <xga.products.phot.Image at 0x7f309d8e14f0>,
 <xga.products.phot.Image at 0x7f309d8e1580>,
 <xga.products.phot.Image at 0x7f309d8e16a0>,
 <xga.products.phot.Image at 0x7f309d8e1790>,
 <xga.products.phot.Image at 0x7f309d8e18b0>,
 <xga.products.phot.Image at 0x7f309d8e19a0>,
 <xga.products.phot.Image at 0x7f309d8e1ac0>,
 <xga.products.phot.Image at 0x7f309d8e1bb0>,
 <xga.products.phot.Image at 0x7f309d8e1cd0>,
 <xga.products.phot.Image at 0x7f309d8e1dc0>]
```

Though by itself its not very useful, as you can't see which image is which. The image class has properties that will tell you the energy band, ObsID, and instrument (*energy_bounds*, *obs_id*, *instrument*), but perhaps you want that information in the returned list?

[6]:
```
src.get_products('image', just_obj=False)
```

[6]:
```
[['0404910601',
  'pn',
  'bound_0.5-2.0',
  <xga.products.phot.Image at 0x7f309d948ca0>],
 ['0404910601',
  'pn',
```

(continues on next page)

(continued from previous page)

```
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d948d60>],
['0404910601',
 'mos1',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d948e80>],
['0404910601',
 'mos1',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d948f70>],
['0404910601',
 'mos2',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d948c10>],
['0404910601',
 'mos2',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e11c0>],
['0201901401',
 'pn',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e12e0>],
['0201901401',
 'pn',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e13d0>],
['0201901401',
 'mos1',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e14f0>],
['0201901401',
 'mos1',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e1580>],
['0201901401',
 'mos2',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e16a0>],
['0201901401',
 'mos2',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e1790>],
['0201903501',
 'pn',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e18b0>],
['0201903501',
 'pn',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e19a0>],
['0201903501',
 'mos1',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e1ac0>],
['0201903501',
 'mos1',
 'bound_2.0-10.0',
```

(continues on next page)

(continued from previous page)

```
 <xga.products.phot.Image at 0x7f309d8e1bb0>],
['0201903501',
 'mos2',
 'bound_0.5-2.0',
 <xga.products.phot.Image at 0x7f309d8e1cd0>],
['0201903501',
 'mos2',
 'bound_2.0-10.0',
 <xga.products.phot.Image at 0x7f309d8e1dc0>]]
```

If you wanted a very specific image, perhaps a 0.5-2.0keV EPIC-PN image from observation 0201903501, you could call the get_products method like this:

```
[7]: src.get_products('image', '0201903501', 'pn', extra_key='bound_0.5-2.0')
```

```
[7]: [<xga.products.phot.Image at 0x7f309d8e18b0>]
```

And finally, if you were actually interested in the combined image (all observations, all instruments - XGA does not generate combined data products for individual observations), then you would call get_products with a different product name:

```
[8]: src.get_products('combined_image', extra_key='bound_0.5-2.0')
```

```
[8]: [<xga.products.phot.Image at 0x7f309d7bbd90>]
```

3.2.6 Methods for retrieving specific types of product

Above I have detailed the general method for retrieving any type of product, but I have also implemented methods that will return specific types on request, though they are essentially all wrappers of that original, general function. Many have been implemented, and I advise you to look at the API documentation for the specific source class you're interested in, but I shall list and demonstrate a few here.

Firstly, we can retrieve single camera images and exposure maps with their own get methods, calling them like this will retrieve the individual images and exposure maps (in all energy bands), though please note that to retrieve PSF corrected images you must pass `psf_corr=True`:

```
[9]: specific_ims = src.get_images()
     specific_ims
```

```
[9]: [<xga.products.phot.Image at 0x7f309d948ca0>,
 <xga.products.phot.Image at 0x7f309d948d60>,
 <xga.products.phot.Image at 0x7f309d948e80>,
 <xga.products.phot.Image at 0x7f309d948f70>,
 <xga.products.phot.Image at 0x7f309d948c10>,
 <xga.products.phot.Image at 0x7f309d8e11c0>,
 <xga.products.phot.Image at 0x7f309d8e12e0>,
 <xga.products.phot.Image at 0x7f309d8e13d0>,
 <xga.products.phot.Image at 0x7f309d8e14f0>,
 <xga.products.phot.Image at 0x7f309d8e1580>,
 <xga.products.phot.Image at 0x7f309d8e16a0>,
 <xga.products.phot.Image at 0x7f309d8e1790>,
 <xga.products.phot.Image at 0x7f309d8e18b0>,
 <xga.products.phot.Image at 0x7f309d8e19a0>,
 <xga.products.phot.Image at 0x7f309d8e1ac0>,
 <xga.products.phot.Image at 0x7f309d8e1bb0>,
```

(continues on next page)

(continued from previous page)

```
 <xga.products.phot.Image at 0x7f309d8e1cd0>,
 <xga.products.phot.Image at 0x7f309d8e1dc0>]
```

If I cycle through the list of image products and look at their energy bounds, ObsIDs, and instruments we can see that we are indeed retrieving the images for all ObsID-Instrument combinations, in all energy bands:

[10]:
```
for im in specific_ims:
    print(im.energy_bounds, im.obs_id, im.instrument)
```

```
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0404910601 pn
(<Quantity 2. keV>, <Quantity 10. keV>) 0404910601 pn
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0404910601 mos1
(<Quantity 2. keV>, <Quantity 10. keV>) 0404910601 mos1
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0404910601 mos2
(<Quantity 2. keV>, <Quantity 10. keV>) 0404910601 mos2
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201901401 pn
(<Quantity 2. keV>, <Quantity 10. keV>) 0201901401 pn
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201901401 mos1
(<Quantity 2. keV>, <Quantity 10. keV>) 0201901401 mos1
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201901401 mos2
(<Quantity 2. keV>, <Quantity 10. keV>) 0201901401 mos2
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201903501 pn
(<Quantity 2. keV>, <Quantity 10. keV>) 0201903501 pn
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201903501 mos1
(<Quantity 2. keV>, <Quantity 10. keV>) 0201903501 mos1
(<Quantity 0.5 keV>, <Quantity 2. keV>) 0201903501 mos2
(<Quantity 2. keV>, <Quantity 10. keV>) 0201903501 mos2
```

Then, if we wanted to be more even more specific, we could decide that we only wanted to retrieve the 0.5-2.0keV exposure map for the PN camera of observation 0201903501. When we specify exactly what exposure map we want, the method returns a single ExpMap instance, rather than a list:

[11]:
```
specific_exs = src.get_expmaps('0201903501', 'pn', Quantity(0.5, 'keV'), Quantity(2.0,
↪ 'keV'))
specific_exs
```

[11]:
```
<xga.products.phot.ExpMap at 0x7f309d8e1880>
```

Then I might perhaps want to retrieve the 0.5-2.0keV **merged** image and ratemap generated for this source, I would use different methods again, specifically for combined images and combined ratemaps (again the `psf_corr=True` argument must be passed to either of these if you want to retrieve a PSF corrected image/ratemap):

[12]:
```
comb_rt = src.get_combined_ratemaps(Quantity(0.5, 'keV'), Quantity(2.0, 'keV'))
comb_im = src.get_combined_images(Quantity(0.5, 'keV'), Quantity(2.0, 'keV'))
```

There are many other specific get methods for products, including for individual and annular spectra, combined_exposure maps, and many different types of profiles (dependant on the particular source class you're using).

3.2.7 NoProductAvailableError

This exception is triggered if you try and use one of the specific get methods to retrieve a product that does not exist. Here, for instance, I am trying to retrieve a merged RateMap in the 0.5-4.2keV energy range, an energy range for which I have not even generated individual Images/ExpMaps, let alone a merged RateMap:

[13]:
```
src.get_combined_ratemaps(Quantity(0.5, 'keV'), Quantity(4.2, 'keV'))
```

```
---------------------------------------------------------------------------
NoProductAvailableError                   Traceback (most recent call last)
<ipython-input-13-b512f325b934> in <module>
----> 1 src.get_combined_ratemaps(Quantity(0.5, 'keV'), Quantity(4.2, 'keV'))

~/code/PycharmProjects/XGA/xga/sources/base.py in get_combined_ratemaps(self, lo_en,␣
↪hi_en, psf_corr, psf_model, psf_bins, psf_algo, psf_iter)
   2902             matched_prods = matched_prods[0]
   2903         elif len(matched_prods) == 0:
-> 2904             raise NoProductAvailableError("Cannot find any combined ratemaps␣
↪matching your input.")
   2905
   2906         return matched_prods

NoProductAvailableError: Cannot find any combined ratemaps matching your input.
```

3.3 Photometry with XGA

This tutorial will show you how to perform basic photometric analyses on XGA sources, starting with how you use the built in SAS interface to generate the necessary XMM images, exposure maps, and the XGA ratemap objects. Using the SAS interface requires that a version of SAS be installed on your system, and the module will check for the presence of this software before trying to run any SAS routines.

I will take you through the basics of interacting with the XGA photometric products, including using source masks and the built in peak finding techniques.

[1]:
```
from astropy.units import Quantity
from astropy.visualization import LinearStretch
import numpy as np
import pandas as pd

from xga.sources import GalaxyCluster, NullSource
from xga.samples import ClusterSample
from xga.sas import evselect_image, eexpmap, emosaic
from xga.utils import xmm_sky
```

First of all, I will declare an individual galaxy cluster source object, and a galaxy cluster sample object. I am going to demonstrate that you can use the SAS interface with individual sources and samples of sources in exactly the same way. The sample of four clusters is taken from the XCS-SDSS sample (Giles et al. (in prep)), and the individual cluster is Abell 907.

[2]:
```
# Setting up the column names and numpy array that go into the Pandas dataframe
column_names = ['name', 'ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']
cluster_data = np.array([['XCSSDSS-124', 0.80057775, -6.0918182, 0.251, 1220.11, 1777.
↪06, 109.55, 4.49],
                         ['XCSSDSS-2789', 0.95553986, 2.068019, 0.11, 1039.14, 1519.
↪79, 38.90, 2.83],
```

(continues on next page)

(continued from previous page)

```
                          ['XCSSDSS-290', 2.7226392, 29.161021, 0.338, 935.58, 1359.37,
↪ 105.10, 5.99],
                          ['XCSSDSS-134', 4.9083898, 3.6098177, 0.273, 1157.04, 1684.
↪15, 108.60, 4.79]])

# Possibly I'm overcomplicating this by making it into a dataframe, but it is an␣
↪excellent data structure,
#  and one that is very commonly used in my own analyses.
sample_df = pd.DataFrame(data=cluster_data, columns=column_names)
sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']] = \
    sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']].
↪astype(float)

# Defining the sample of four XCS-SDSS galaxy clusters
demo_smp = ClusterSample(sample_df["ra"].values, sample_df["dec"].values, sample_df["z
↪"].values,
                         sample_df["name"].values, r200=Quantity(sample_df["r200"].
↪values, "kpc"),
                         r500=Quantity(sample_df["r500"].values, 'kpc'),␣
↪richness=sample_df['richness'].values,
                         richness_err=sample_df['richness_err'].values)

# And defining an individual source object for Abell 907
demo_src = GalaxyCluster(149.59209, -11.05972, 0.16, r500=Quantity(1200, 'kpc'),␣
↪r200=Quantity(1700, 'kpc'),
                         name="A907")
```

```
Declaring BaseSource Sample: 100%|| 4/4 [00:01<00:00,  2.77it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:10<00:00,  2.70s/it]
Generating products of type(s) image: 100%|| 4/4 [00:03<00:00,  1.32it/s]
Generating products of type(s) expmap: 100%|| 4/4 [00:01<00:00,  2.24it/s]
Setting up Galaxy Clusters: 100%|| 4/4 [00:03<00:00,  1.23it/s]
```

3.3.1 A note on XGA's Parallelism

In every SAS function built into XGA (and indeed many functions in other parts of the module), you will find a `num_cores` keyword argument which tells the function how many cores on your local machine it is allowed to use. The default value is 90% of the available cores on your system, though you are of course free to set your own value when you call the functions.

To see the number of cores which have automatically allocated to XGA, you can import the NUM_CORES constant from the base xga module (this tutorial for instance was run on a laptop with an Intel Core i5-8300H CPU, with four physical cores and four logical cores):

[3]:
```
from xga import NUM_CORES
NUM_CORES
```

[3]:
```
7
```

You can also manually set this value globally, before running anything. Either set the `num_cores` option in the [XGA_SETUP] section of the configuration file, or simply set the NUM_CORES constant imported from xga.

3.3.2 Generating XMM Images with XGA

Please note that the XCS data reduction pipeline produces images and exposure maps in the 0.5-2.0keV and 2.0-10.0keV energy ranges, and as the configuration file points XGA towards existing data, these sources will already have those images and exposure maps loaded in.

Please also note that ClusterSample class automatically generates images and exposure maps in the energy range used for peak finding (set by the keyword arguments `peak_lo_en` and `peak_hi_en`, by default 0.5keV and 2.0keV respectively). An individual GalaxyCluster source object will also automatically generate images and exposure maps in this range, but only if the `use_peak` keyword argument is set to True (which it is by default).

As such, I'm going to demonstrate how you can generate images and exposure maps in another (probably not very useful in a real analysis) energy range, 0.5-2.0keV. When telling SAS what energy range to generate in, we use Astropy quantities in units of keV, though you could supply limits in any energy unit:

[4]:
```
# Setting up the lower and upper energy bounds for our products
lo_en = Quantity(0.5, 'keV')
hi_en = Quantity(10.0, 'keV')
```

[5]:
```
# This will generate images in the 0.5-10keV range, for every instrument in
#  every ObsID in every source in this sample
demo_smp = evselect_image(demo_smp, lo_en=lo_en, hi_en=hi_en)

# The function call is identical when generating images for an individual source, but␣
↪here
#  I have limited XGA to using only four cores
demo_src = evselect_image(demo_src, lo_en=lo_en, hi_en=hi_en, num_cores=4)
```

```
Generating products of type(s) image: 100%|| 12/12 [00:15<00:00,  1.31s/it]
Generating products of type(s) image: 100%|| 9/9 [00:04<00:00,  1.97it/s]
```

These images have now been generated and associated with the appropriate source.

3.3.3 Generating XMM Exposure Maps with XGA

The process for generating exposure maps is almost identical to the image generation process, we simply call a different function, though XMM calibration files (CCFs) are required to make exposure maps, so they will be generated if they do not already exist:

[6]:
```
# This will generate expmaps in the 0.5-10keV range, for every instrument in
#  every ObsID in every source in this sample
demo_smp = eexpmap(demo_smp, lo_en=lo_en, hi_en=hi_en)

# The function call is identical when generating expmaps for an individual source
demo_src = eexpmap(demo_src, lo_en=lo_en, hi_en=hi_en)
```

```
Generating products of type(s) expmap: 100%|| 12/12 [01:21<00:00,  6.78s/it]
Generating products of type(s) expmap: 100%|| 9/9 [00:46<00:00,  5.21s/it]
```

3.3.4 XGA Ratemaps

Now that we've generated images and exposure maps for our weird, custom energy range, we can see that XGA has automatically made RateMaps from those products, I use the individual source as a demonstration here. There are nine ratemaps present because there are three observations of Abell 907, each with three valid instruments:

[7]:
```
rts = demo_src.get_ratemaps(lo_en=lo_en, hi_en=hi_en)
rts
```

[7]:
```
[<xga.products.phot.RateMap at 0x7f8bb5477130>,
 <xga.products.phot.RateMap at 0x7f8bae83efa0>,
 <xga.products.phot.RateMap at 0x7f8bb5477dc0>,
 <xga.products.phot.RateMap at 0x7f8bae7f8430>,
 <xga.products.phot.RateMap at 0x7f8bae7f8eb0>,
 <xga.products.phot.RateMap at 0x7f8bae83e040>,
 <xga.products.phot.RateMap at 0x7f8bae8477c0>,
 <xga.products.phot.RateMap at 0x7f8bb54774f0>,
 <xga.products.phot.RateMap at 0x7f8bae8384c0>]
```

3.3.5 Basic properties of ANY XGA Product

Here I just demonstrate some of the most basic (but still useful) properties of XGA's photometric products. I am going to use one of the ratemaps generated for Abell 907, but the properties I demonstrate in this section will be present in **any** XGA photometric product. These first properties should actually be present in every XGA product, photometric or otherwise:

[8]:
```
chosen_ratemap = rts[0]

# If generated using XGA's SAS interface, the product stores the name
#  of the source it is associated with
print(chosen_ratemap.src_name, '\n')

# Its also aware of what ObsID and instrument it is from
print(chosen_ratemap.obs_id, chosen_ratemap.instrument, '\n')

# And where the base file is stored
print(chosen_ratemap.path)
```

```
A907

0404910601 pn

/home/dt237/code/PycharmProjects/XGA/docs/source/notebooks/tutorials/xga_output/
↪0404910601/0404910601_pn_0.5-10.0keVimg.fits
```

3.3.6 Basic properties of XGA Photometric products

Here I demonstrate how to access the data and fits header information of an XGA photometric product. I also show off some of the useful functions that are built into photometric objects to make our lives easier.

Accessing an image's (or in this case a ratemap's) data is extremely simple, with the added benefit that data is not loaded into memory from the disk until the user actually wants to access it. Its possible to have hundreds of images associated with a single source, so avoiding having data in memory unless its needed helps stop us running out of RAM.

You shouldn't be concerned that the data shown here is are all zeros, the outskirts of the ratemap are off of the XMM detector and as such have a countrate of zero:

[9]:
```
# Retrieving the data as a numpy array is as simple as
#  calling the data property
my_ratemap_data = chosen_ratemap.data
my_ratemap_data
```

[9]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

You can also retrieve the whole header of the fits image by using the header property:

[10]:
```
chosen_ratemap.header
```

[10]:
```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                  512 / length of data axis 1
NAXIS2  =                  512 / length of data axis 2
EXTEND  =                    T / FITS dataset may contain extensions
XPROC0  = 'evselect table=/home/dt237/apollo_mnt/xmm_obs/data/0404910601/eclean'
XDAL0   = '0404910601_pn_0.5-10.0keVimg.fits 2021-06-09T16:25:26.000 Create evs'
CREATOR = 'evselect (evselect-3.68) [xmmsas_20190531_1155-18.0.0]' / name of cod
DATE    = '2021-06-09T16:25:26.000' / creation date
ORIGIN  = 'User site'           /
ODSVER  = '16.911'              / ODS version
ODSCHAIN= 'S2KO'                / Processing Chain
FILTER  = 'Thin1'               / Filter ID
FRMTIME =                  199 / [ms] Frame Time, nearest integer
TELESCOP= 'XMM'                 / Telescope (mission) name
INSTRUME= 'EPN'                 / Instrument name
REVOLUT =                 1184 / XMM revolution number
OBS_MODE= 'POINTING'            / Observation mode (pointing or slew)
OBS_ID  = '0404910601'          / Observation identifier
EXP_ID  = '0404910601003'       / Exposure identifier
EXPIDSTR= 'S003'                / Exposure identifier (Xnnn)
CONTENT = 'EPIC PN IMAGING MODE EVENT LIST' /
DATAMODE= 'IMAGING'             / Instrument mode (IMAGING, TIMING, BURST, etc.)
SUBMODE = 'PrimeFullWindowExtended' / Guest Observer mode
OBJECT  = 'RXCJ0958-1103'       / Name of observed object
DATE-OBS= '2006-05-27T10:58:40' / Start Time (UTC) of exposure
DATE-END= '2006-05-27T15:34:21' / End Time (UTC) of exposure
DATE_OBS= '2006-05-27T09:57:27.000' / Date observations were made
DATE_END= '2006-05-27T15:42:37.000' / Date observations ended
DURATION=              20710.0 / [s] Duration of observation
TIMESYS = 'TT'                  / XMM time will be TT (Terrestial Time)
MJDREF  =              50814.0 / [d] 1998-01-01T00:00:00 (TT) expressed in MJD (
TIMEZERO=                    0 / Clock correction (if not zero)
TIMEUNIT= 's'                   / All times in s unless specified otherwise
CLOCKAPP=                    T / Clock correction applied
TIMEREF = 'LOCAL'               / Reference location of photon arrival times
TASSIGN = 'SATELLITE'           / Location of time assignment
```

(continues on next page)

(continued from previous page)

```
OBSERVER= 'Prof Hans Boehringer' / Name of PI
RA_OBJ  =           149.592084 / [deg] Ra of the observed object
DEC_OBJ =          -11.0597222 / [deg] Dec of the observed object
RA_NOM  =           149.592084 / [deg] Ra of boresight
DEC_NOM =          -11.0597222 / [deg] Dec of boresight
SRCPOS  =                  190 / [pixel] Assumed RAWY source position for fast m
SRCPOSX =     36.6997234597762 / [pixel] real-valued RAWX source position
SRCPOSY =     190.216005661132 / [pixel] real-valued RAWY source position
EXPSTART= '2006-05-27T10:58:40' / Start Time (UTC) of exposure
EXPSTOP = '2006-05-27T15:34:21' / End Time (UTC) of exposure
OBSSTART= '2006-05-27T09:57:27.000' / Date observation was started
OBSSTOP = '2006-05-27T15:42:37.000' / Date observation was stopped
ATT_SRC = 'AHF'                / Source of attitude data (AHF|RAF|THF)
ORB_RCNS=                    T / Reconstructed orbit data used?
TFIT_RPD=                    F / Recalculated signal propagation delays?
TFIT_DEG=                    4 / Degree of OBT-MET fit polynomial
TFIT_RMS= 5.43513446215138e-06 / RMS value of OBT-MET polynomial fit
TFIT_PFR=                  0.0 / Fraction of disregarded TCS data points
TFIT_IGH=                    T / Ignored GS handover in TC data?
LONGSTRN= 'OGIP 1.0'           /
EQUINOX =               2000.0 / Equinox for sky coordinate x/y axes
RADECSYS= 'FK5'                / World coord. system for this file
REFXCTYP= 'RA---TAN'           / WCS Coord. type: RA tangent plane projection
REFXCRPX=                25921 / WCS axis reference pixel of projected image
REFXCRVL=           149.591375 / [deg] WCS coord. at X axis ref. pixel
REFXCDLT= -1.38888888888889e-05 / [deg/pix] WCS X increment at ref. pixel
REFXLMIN=                    1 / WCS minimum legal QPOE projected image X axis v
REFXLMAX=                51840 / WCS maximum legal QPOE projected image X axis v
REFXDMIN=                 1254 / WCS minimum projected image X axis data value
REFXDMAX=                14799 / WCS maximum projected image X axis data value
REFXCUNI= 'deg'                / WCS Physical units of X axis
REFYCTYP= 'DEC--TAN'           / WCS Coord. type: DEC tangent plane projection
REFYCRPX=                25921 / WCS axis reference pixel of projected image
REFYCRVL=    -11.0887222222222 / [deg] WCS coord. at Y axis ref. pixel
REFYCDLT= 1.38888888888889e-05 / [deg/pix] WCS Y increment at ref. pixel
REFYLMIN=                    1 / WCS minimum legal QPOE projected image Y axis v
REFYLMAX=                51840 / WCS maximum legal QPOE projected image Y axis v
REFYDMIN=                17644 / WCS minimum projected image Y axis data value
REFYDMAX=                33229 / WCS maximum projected image Y axis data value
REFYCUNI= 'deg'                / WCS Physical units of Y axis
AVRG_PNT= 'MEDIAN'             / Meaning of PNT values (mean or median)
RA_PNT  =           149.591375 / [deg] Actual (mean) pointing RA of the optical
DEC_PNT =    -11.0887222222222 / [deg] Actual (mean) pointing Dec of the optica
PA_PNT  =     306.720916748047 / [deg] Actual (mean) measured position angle of
HISTORY   1-06-09T16:25:26
HISTORY   Created by evselect (evselect-3.68) [xmmsas_20190531_1155-18.0.0] at '
XMMSEP  =                   83 / 83 primary image keywords follow
SLCTEXPR= '(PI in [500:10000])' / Filtering expression used by evselect
HDUCLASS= 'OGIP'               / Format conforms to OGIP/GSFC conventions
HDUCLAS1= 'IMAGE'              / File contains an Image
HDUCLAS2= 'TOTAL'              / Gross Image
HDUVERS1= '1.1.0'              / Version of format
CTYPE1  = 'RA---TAN'           / Coord. type: RA tangent plane projection
CRPIX1  =     256.505747126437 / WCS reference pixel
CRVAL1  =           149.591375 / [deg] coord. at X axis ref. pixel
CUNIT1  = 'deg'                / Physical units of X axis
CDELT1  = -0.00120833333333333 / WCS pixel size
```

(continues on next page)

(continued from previous page)

```
CTYPE1L = 'X'                  / WCS coordinate name
CRPIX1L =                    1 / WCS reference pixel
CRVAL1L =               3692.0 / WCS reference pixel value
CDELT1L =                 87.0 / WCS pixel size
LTV1    =    -41.4367816091954 /
LTM1_1  =   0.0114942528735632 /
CTYPE2  = 'DEC--TAN'           / Coord. type: DEC tangent plane projection
CRPIX2  =     256.505747126437 / WCS reference pixel
CRVAL2  =    -11.0887222222222 / [deg] coord. at Y axis ref. pixel
CUNIT2  = 'deg'                / Physical units of Y axis
CDELT2  =  0.00120833333333333 / WCS pixel size
CTYPE2L = 'Y'                  / WCS coordinate name
CRPIX2L =                    1 / WCS reference pixel
CRVAL2L =               3692.0 / WCS reference pixel value
CDELT2L =                 87.0 / WCS pixel size
LTV2    =    -41.4367816091954 /
LTM2_2  =   0.0114942528735632 /
WCSNAMEL= 'PHYSICAL'           / WCS L name
WCSAXESL=                    2 / No. of axes for WCS L
LTM1_2  =                    0 /
LTM2_1  =                    0 /
TIMEDEL =           0.19453132 / [s] Length of exposure entry interval
DSTYP1  = 'CCDNR'              / data subspace descriptor: name
DSTYP2  = 'FLAG'               / data subspace descriptor: name
DSTYP3  = 'PATTERN'            / data subspace descriptor: name
DSTYP4  = 'FLAG'               / data subspace descriptor: name
DSTYP5  = 'TIME'               / data subspace descriptor: name
DSUNI5  = 's'                  / data subspace descriptor: units
DSTYP6  = 'PI'                 / data subspace descriptor: name
DSUNI6  = 'eV'                 / data subspace descriptor: units
DSVAL1  = '1'                  / data subspace descriptor: value
DSFORM2 = 'X'                  / data subspace descriptor: data type
DSVAL2  = 'b00xx00000x0xxxxxxxxxxxxxxxx' / data subspace descriptor: value
DSVAL3  = ':4'                 / data subspace descriptor: value
DSVAL4  = '0'                  / data subspace descriptor: value
DSVAL5  = 'TABLE'              / data subspace descriptor: value
DSREF5  = ':GTI00005'          / data subspace descriptor: reference
DSVAL6  = '500:10000'          / data subspace descriptor: value
2DSVAL1 = '2'                  / data subspace descriptor: value
2DSREF5 = ':GTI00105'          / data subspace descriptor: reference
3DSVAL1 = '3'                  / data subspace descriptor: value
3DSREF5 = ':GTI00205'          / data subspace descriptor: reference
4DSVAL1 = '4'                  / data subspace descriptor: value
4DSREF5 = ':GTI00305'          / data subspace descriptor: reference
5DSVAL1 = '5'                  / data subspace descriptor: value
5DSREF5 = ':GTI00405'          / data subspace descriptor: reference
6DSVAL1 = '6'                  / data subspace descriptor: value
6DSREF5 = ':GTI00505'          / data subspace descriptor: reference
7DSVAL1 = '7'                  / data subspace descriptor: value
7DSREF5 = ':GTI00605'          / data subspace descriptor: reference
8DSVAL1 = '8'                  / data subspace descriptor: value
8DSREF5 = ':GTI00705'          / data subspace descriptor: reference
9DSVAL1 = '9'                  / data subspace descriptor: value
9DSREF5 = ':GTI00805'          / data subspace descriptor: reference
10DSVAL1= '10'                 / data subspace descriptor: value
10DSREF5= ':GTI00905'          / data subspace descriptor: reference
11DSVAL1= '11'                 / data subspace descriptor: value
```

(continues on next page)

(continued from previous page)

```
11DSREF5= ':GTI01005'          / data subspace descriptor: reference
12DSVAL1= '12'                 / data subspace descriptor: value
12DSREF5= ':GTI01105'          / data subspace descriptor: reference
ONTIME01=               6050.0 / Sum of GTIs for a particular CCD
ONTIME02=               6050.0 / Sum of GTIs for a particular CCD
ONTIME03=               6050.0 / Sum of GTIs for a particular CCD
ONTIME04=               6050.0 / Sum of GTIs for a particular CCD
ONTIME05=               6050.0 / Sum of GTIs for a particular CCD
ONTIME06=               6050.0 / Sum of GTIs for a particular CCD
ONTIME07=               6050.0 / Sum of GTIs for a particular CCD
ONTIME08=               6050.0 / Sum of GTIs for a particular CCD
ONTIME09=               6050.0 / Sum of GTIs for a particular CCD
ONTIME10=               6050.0 / Sum of GTIs for a particular CCD
ONTIME11=               6050.0 / Sum of GTIs for a particular CCD
ONTIME12=               6050.0 / Sum of GTIs for a particular CCD
EXPOSURE=               6050.0 / max of ONTIMEnn values
```

And a specific entry by indexing the header object using the entries name:

[11]:
```
chosen_ratemap.header['OBSERVER']
```

[11]:
```
'Prof Hans Boehringer'
```

3.3.7 Converting coordinates with XGA Photometric products

Reading in the header means that XGA has access to the World Coordinate System (WCS) information of the fits image, and using this I wrote a function that will convert between different coordinate systems for a given photometric product. Again, any coordinates must be passed in as Astropy quantity objects:

[12]:
```
# Defining a position to convert in pixel coordinates
pix_coord = Quantity([100, 100], 'pix')

# The pixel coordinate can be converted to degrees like this
print(chosen_ratemap.coord_conv(pix_coord, 'deg'), '\n')

# If the product fits file had a WCS entry for the XMM sky coordinates system,
#  then we can convert to that as well
print(chosen_ratemap.coord_conv(pix_coord, 'xmm_sky'))
```

```
[149.782975   -11.27656289] deg

[12392. 12392.] xmm_sky
```

We can also actually define an Astropy quantity in XMM's sky units, by dint of a custom unit defined in the XGA *utils* file (I imported it at the top of this tutorial). Please be aware that the Astropy module is not aware of this custom unit, so defining a quantity with a string representation of the unit will not work:

[13]:
```
# Defining an XMM Sky position
sky_coord = Quantity([15000, 15000], xmm_sky)

# Converting to pixels
print(chosen_ratemap.coord_conv(sky_coord, 'pix'), '\n')

# Converting to an RA-Dec position
print(chosen_ratemap.coord_conv(sky_coord, 'deg'))
```

```
[130 130] pix

[149.74602109 -11.24036252] deg
```

We can also easily convert between any combination of these units, and here I demonstrate how to convert the user supplied coordinates (used when the cluster was defined), as well as the peak coordinates found during declaration:

[14]:
```
# This is an easy way to access the coordinates the user passed in when declaring the source
print('The initial coordinates were', demo_src.ra_dec)
print(chosen_ratemap.coord_conv(demo_src.ra_dec, 'pix'), '\n')

# And an equally easy way to access the peak coordinates
print('The peak coordinates are', demo_src.peak)
print(chosen_ratemap.coord_conv(demo_src.peak, 'pix'), '\n')
```

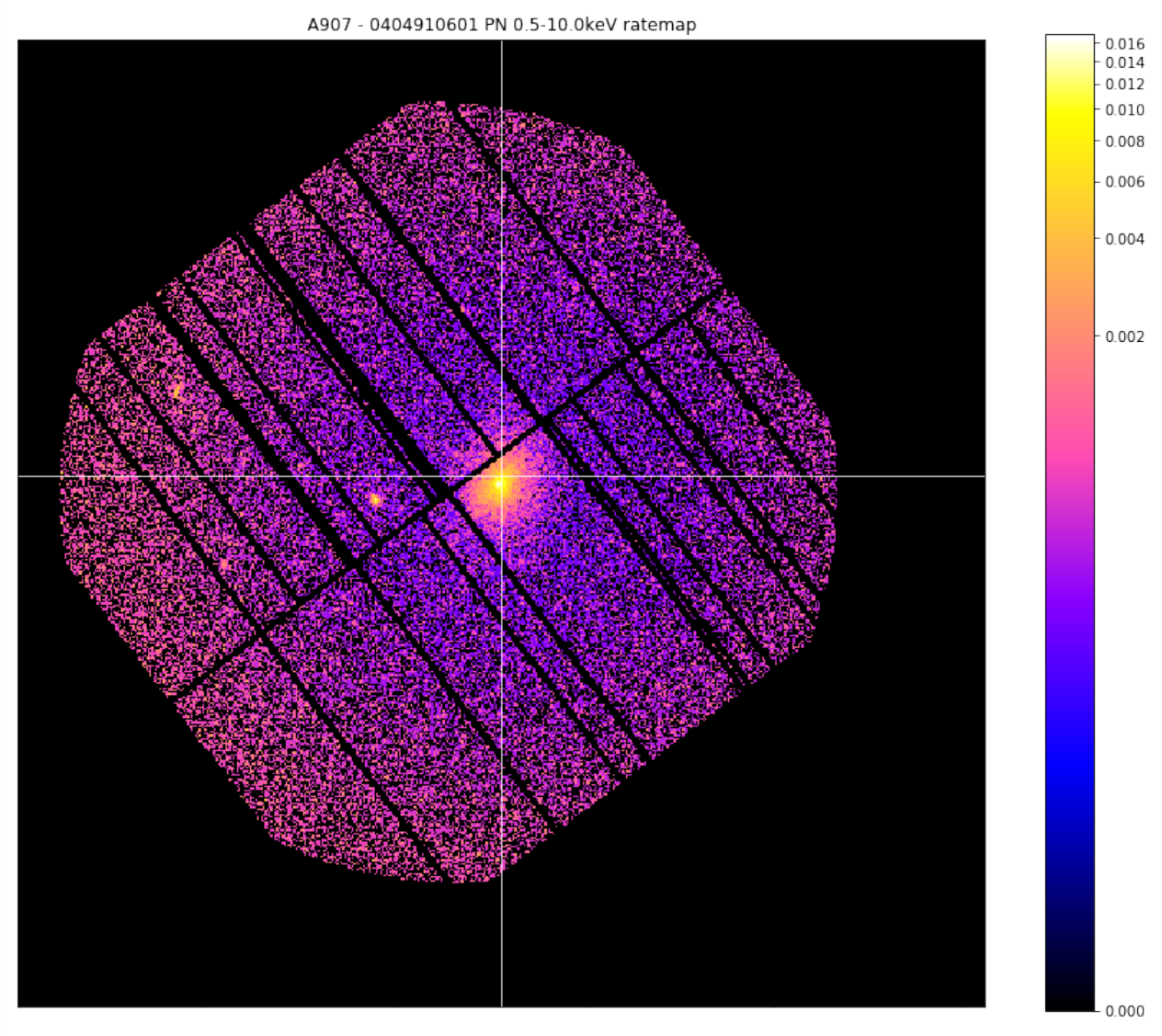
```
The initial coordinates were [149.59209 -11.05972] deg
[255 280] pix

The peak coordinates are [149.59251341 -11.06395832] deg
[255 276] pix
```

3.3.8 Viewing XGA Photometric products

One of the most useful features built into XGA's photometric products is the ability to easily visualise them, with multiple ways to modify the image produced. This includes changing the size of the image, altering the scaling applied (log-scaling is used by default), applying masks, zooming in specific parts of the image, and adding crosshairs to indicate a coordinate.

Again, we're using a ratemap taken from Abell 907, and with one simple line of code we can produce a useful visualisation. I've increased the size of the image so that it looks better in the documentation, and used the crosshairs to highlight the coordinate of the user supplied position. Only one crosshair can be placed on an image, but the coordinate can be in degrees, pixels, or XMM sky coordinates:

[15]:
```
# Displaying the ratemap, with a crosshair for a particular coordinate
chosen_ratemap.view(cross_hair=demo_src.ra_dec, figsize=(12, 10))
```



3.3.9 Retrieving a value at a certain coordinate

At some point you may wish to know the value of a photometric product at a certain coordinate, a ratemap or an exposure map for instance. There are built in methods to help you retrieve those value which will accept coordinates in any valid unit (degrees, pixels, or XMM sky). The value is returned as an Astropy quantity, with the correct units for the type of product.

I'll also show you a useful feature of ratemap, that it stores links to the image and expmap that were used to create it, so accessing those constituent products is as easy as using the 'image' or 'expmap' properties of the ratemap:

[16]:
```
# For a ratemap, we use the method 'get_rate'
print(chosen_ratemap.get_rate(demo_src.ra_dec), '\n')

# If we wished to know the exposure time at that position, we could access the ratemap
↪'s
#  expmap and use 'get_exp'
print(chosen_ratemap.expmap.get_exp(demo_src.ra_dec), '\n')
```

```
0.006222707412880577 ct s

4981.75439453125 s
```

3.3.10 Combined images and exposure maps

Now that we have generated images and exposure maps for all the ObsIDs and instruments associated with all these sources that we're interested in analysing, we will want to bring all that information together by creating combined images and exposure maps. Most XGA analyses take place on combined products, because that ability to find all relevant data for a particular source is one of XGA's key strengths. Currently I use the SAS routine emosaic to generate the combined data products, and it is nearly as simple as generating images and exposure maps, the main difference is that you need to tell emosaic what type of product you want to combine:

[17]:
```
# ------------IMAGES------------
# Starting off with making combined images for the sample
demo_smp = emosaic(demo_smp, 'image', lo_en=lo_en, hi_en=hi_en)

# And for the single source
demo_src = emosaic(demo_src, 'image', lo_en=lo_en, hi_en=hi_en)

# -----------EXPMAPS------------
# The same again but telling emosaic that we want combined exposure maps this time
demo_smp = emosaic(demo_smp, 'expmap', lo_en=lo_en, hi_en=hi_en)

demo_src = emosaic(demo_src, 'expmap', lo_en=lo_en, hi_en=hi_en)
```

```
Generating products of type(s) image: 100%|| 4/4 [00:00<00:00,  9.67it/s]
Generating products of type(s) image: 100%|| 1/1 [00:00<00:00,  2.51it/s]
Generating products of type(s) expmap: 100%|| 4/4 [00:00<00:00, 10.10it/s]
Generating products of type(s) expmap: 100%|| 1/1 [00:00<00:00,  2.52it/s]
```

I'm also going to read out the combined ratemap into a variable for easy use in other parts of this tutorial:

[18]:
```
chosen_comb_rt = demo_src.get_products('combined_ratemap', extra_key='bound_0.5-10.0
↪')[0]
```

3.3.11 Retrieving and using masks from a source

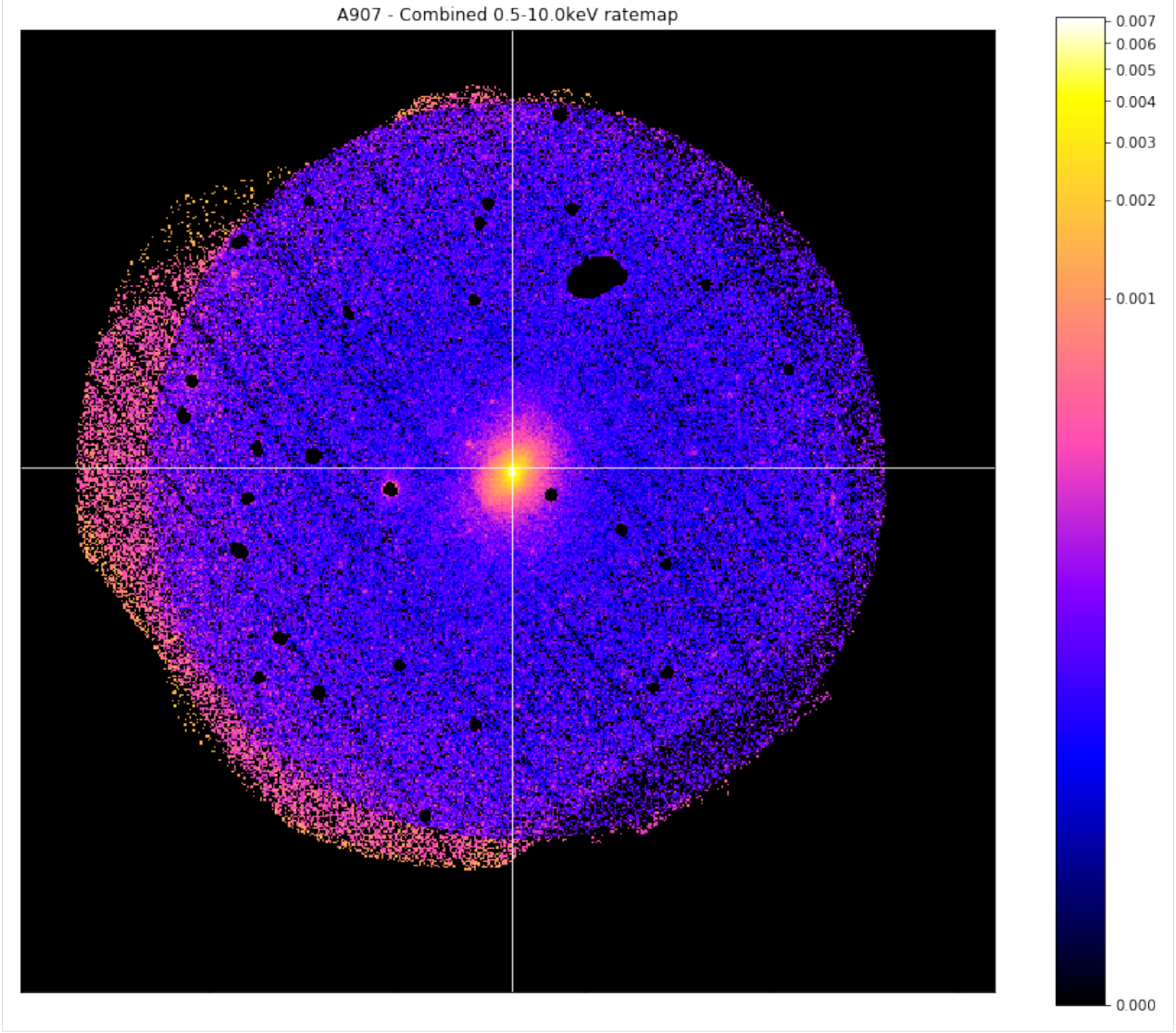
It is common practise in photometric analyses to mask out any sources that might bias or interfere with the source that you are investigating, and as such each XGA source object is able to produce masking arrays. These masking arrays are filled with ones and zeros, where any zero elements are to be considered 'masked', and masked image/expmap/ratemap data can be calculated by multiplying the image/expmap/ratemap data by the masking array.

XGA is capable of producing several types of mask; interloper masks (where non-relevant sources are masked), source masks (where any part of the product not within a certain source region is masked out), and standard masks (which are a combination of the last two types).

An XGA source can generate masks for any of the ObsIDs associated with it, which is important because the ObsIDs are all likely to have different pointing coordinates and roll angles, and as such the pixel coordinates will not correspond to the same RA-Dec coordinates. If no ObsID information is supplied then XGA will assume that the mask should be generated for a combined photometric product.

I'm also going to demonstrate the masking functionality of the view() method for photometric XGA products by showing the effects of the masks I'm generating on actual ratemaps:

[19]:
```
# This generates a mask that removes interloper sources (any non-relevant source). As we
#  didn't provide an ObsID it is only valid for the combined products
interloper_mask = demo_src.get_interloper_mask()

# It is very easy to apply the mask to remove any interlopers from the ratemap
chosen_comb_rt.view(cross_hair=demo_src.ra_dec, mask=interloper_mask, figsize=(12, 10))
```



Its just as easy to produce an interloper mask for a specific ObsID, and though the ratemaps look extremely similar, please note that this is because the three observations associated with this source all have near identical pointing coordinates. That means that the shape of this image is *very* similar to the 512x512 of the individual images. This may not be the case for other sources, where separate observations added together can result in quite different image shapes.

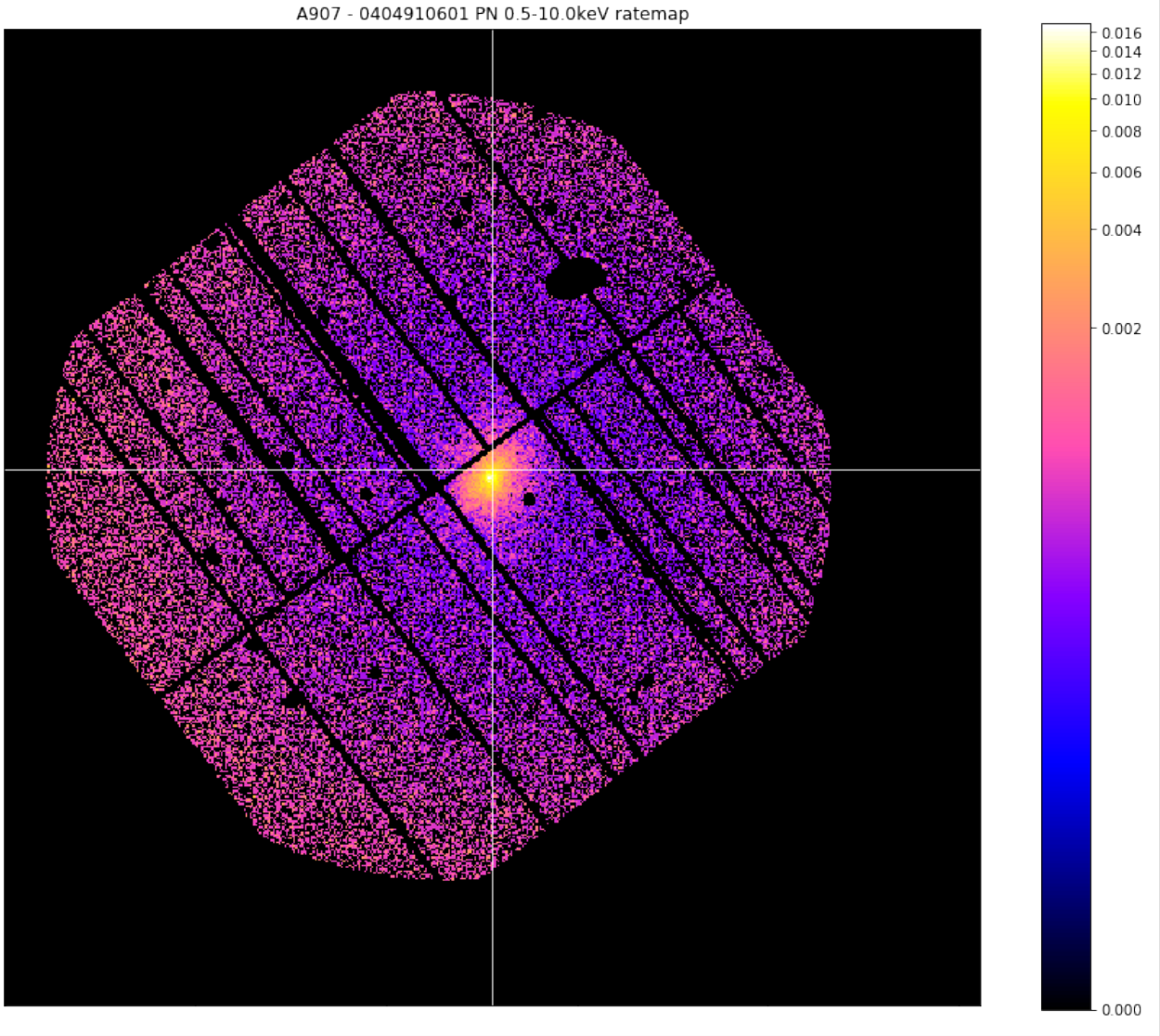
Here we produce a mask for the Abell 907 ratemap we retrieved earlier:

[20]:
```
# Much the same process as to generate the mask for the combined ratemap, but we also supply an ObsID
```

(continues on next page)

(continued from previous page)

```
individual_interloper_mask = demo_src.get_interloper_mask(obs_id=chosen_ratemap.obs_
↪id)

# It is very easy to apply the mask to remove any interlopers from the ratemap
chosen_ratemap.view(cross_hair=demo_src.ra_dec, mask=individual_interloper_mask,␣
↪figsize=(12, 10))
```

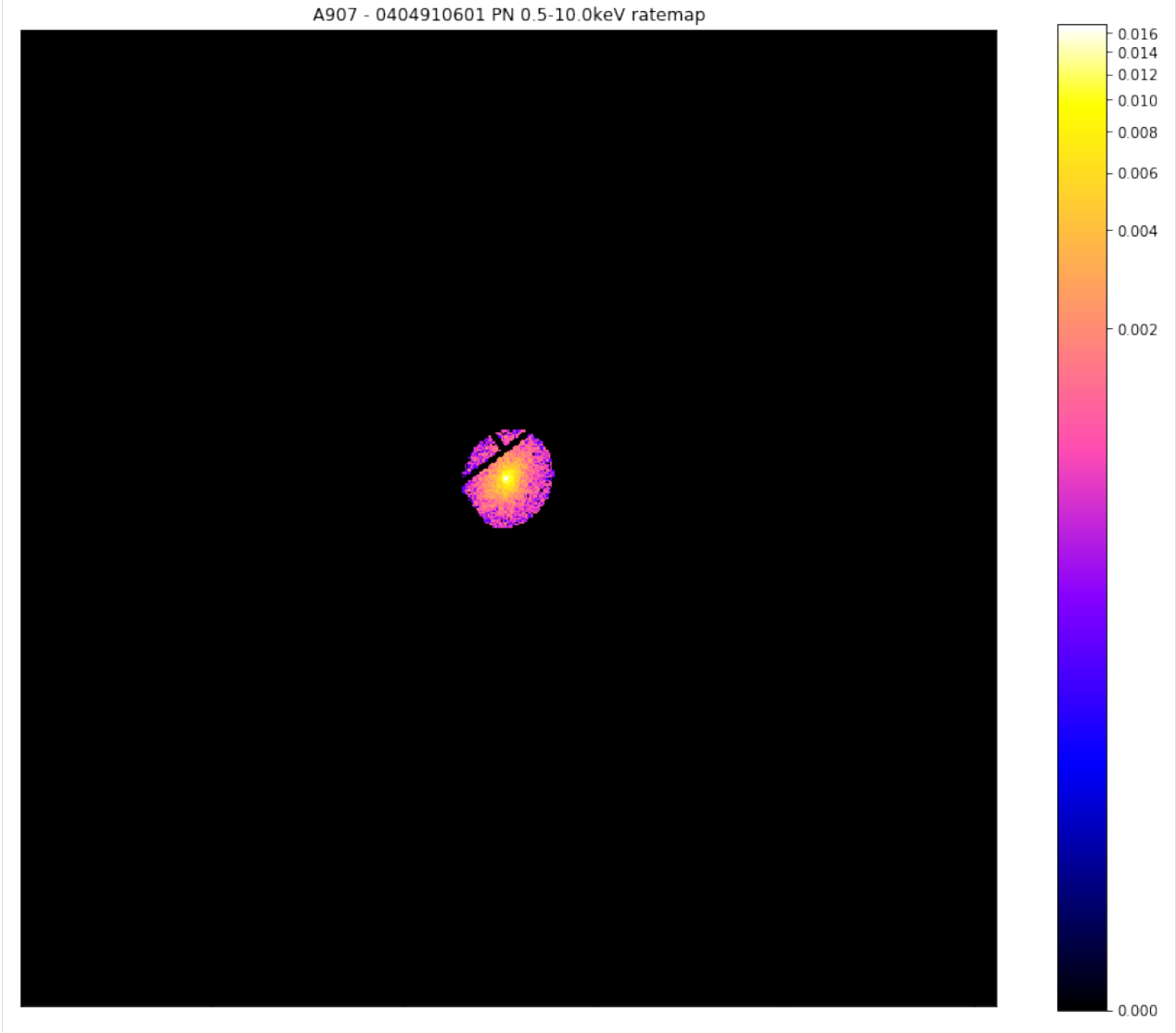


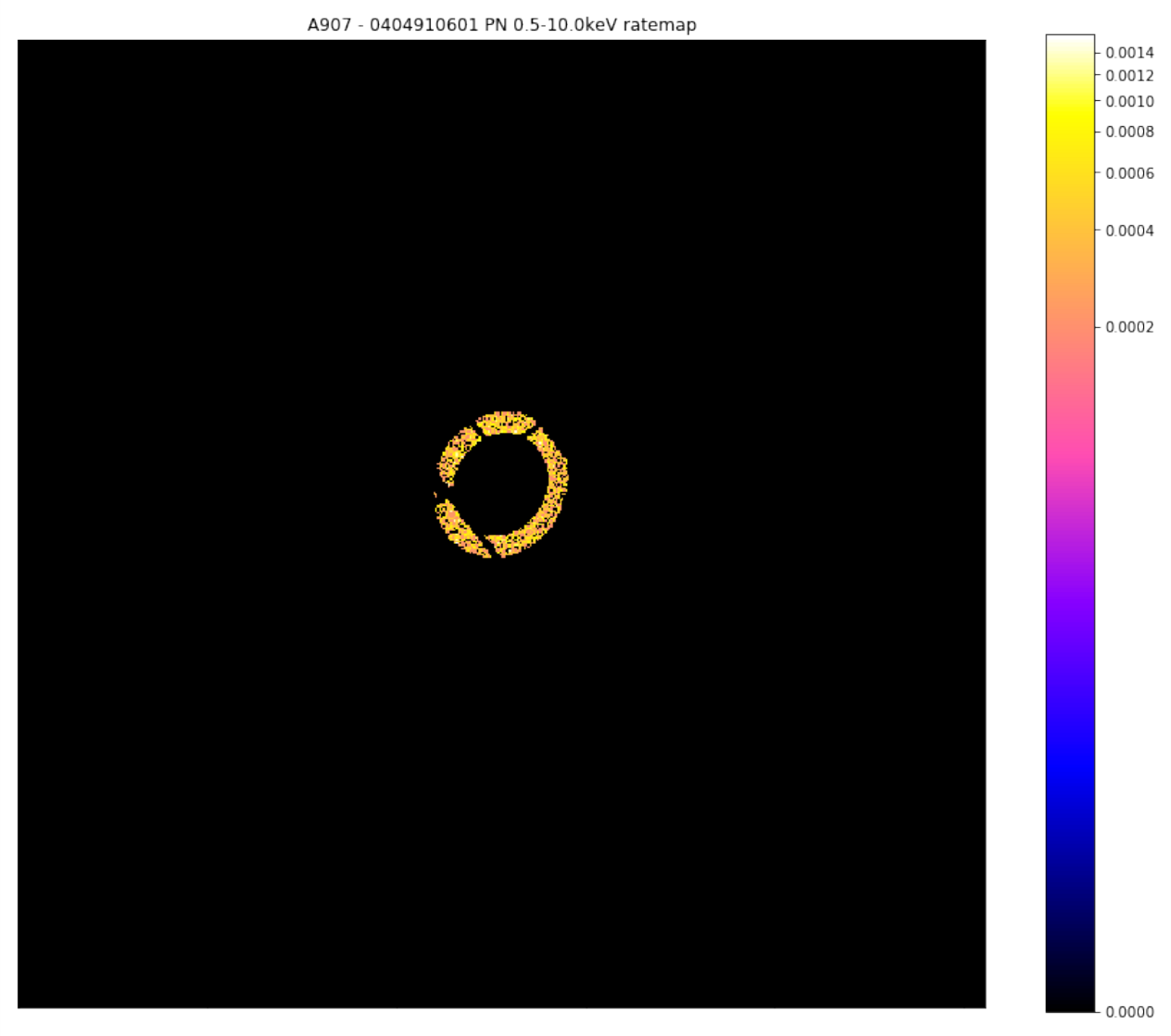
Now we are going to learn how to generate masks that lets us select the source region of interest. At this point we have to make a choice as to which source region we're interested in; most source objects will allow us to generate a mask from the region files which were supplied to XGA in the configuration file, but if you have a point source then you could also choose 'point', and if you have a Galaxy Cluster then you could also select one of the overdensity regions (such as 'r200').

When you generate a source region mask, you will also be supplied with a matching background region mask, which is a region extending between `back_inn_rad_factor` - `back_out_rad_factor` multiplied by the radius of interest, where these keyword arguments are set when defining a source object.

[21]:
```
# A source and background mask generated for the selected source from our region file␣
↪for
#  this particular ObsID
region_masks = demo_src.get_source_mask('region', chosen_ratemap.obs_id)

# Here I will show the source mask applied to the 0404910601-PN ratemap in the 0.5-10.
↪0keV range
chosen_ratemap.view(mask=region_masks[0], figsize=(12, 10))
# I will also show the background mask applied to this ratemap
chosen_ratemap.view(mask=region_masks[1], figsize=(12, 10))
```

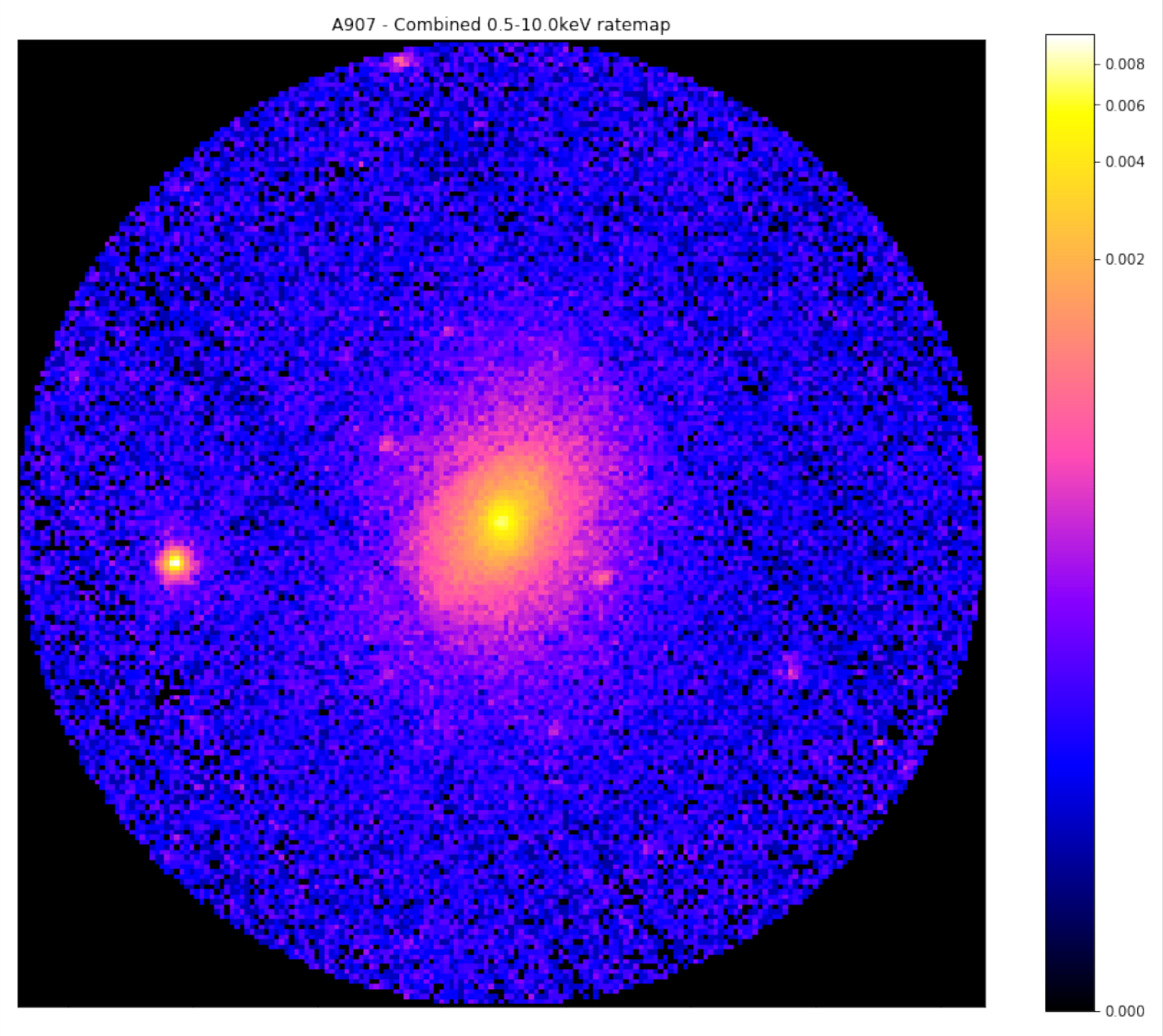
```
/home/dt237/code/PycharmProjects/XGA/xga/sources/base.py:1334: UserWarning: You␣
↪cannot use custom central coordinates with a region from supplied region files
  warnings.warn("You cannot use custom central coordinates with a region from␣
↪supplied region files")
```





Now we've seen how to access a region file region, we're going to look at an R_{500} region, this also seems like a good time to demonstrate XGA's automatic zoom functionality when viewing photometric products:

[22]:
```
# A source and background mask generated for the combined ratemap for Abell 907 at R_
↪500
r500_masks = demo_src.get_source_mask('r500')

# Here I will show the source mask applied to the combined ratemap in the 0.5-10.0keV␣
↪range
chosen_comb_rt.view(mask=r500_masks[0], figsize=(12, 10), zoom_in=True)
```

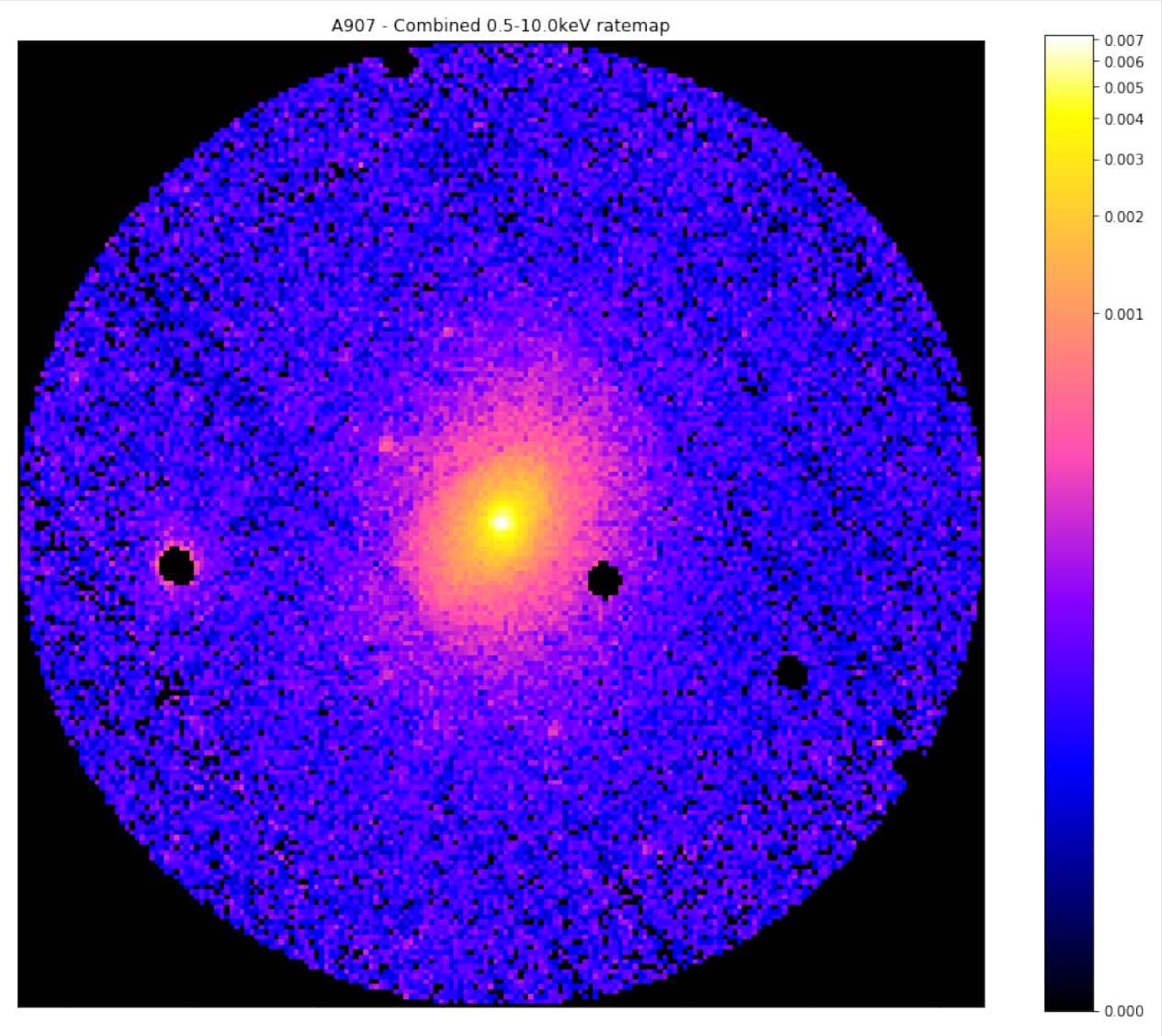


Finally, we can still see that there are some obvious point sources remaining in this source region, and for a real analysis we would want to be able to remove them. As such we use the source method 'get_mask', and apply it to the same combined ratemap.

I also wish to note that the user does not have to generate these masks centered at the default central coordinates for the source object in question, you can also specify another central coordinate using **central_coord** in a mask generation method.

When we see the image below we can quite clearly see examples of a few point sources that obviously have to be removed before a meaningful analysis of the galaxy cluster can begin:

[23]:
```
# A source and background mask generated for the combined ratemap for Abell 907 at R_
↪500
r500_interloper_masks = demo_src.get_mask('r500',)

# Here I will show the source mask applied to the combined ratemap in the 0.5-10.0keV␣
↪range
chosen_comb_rt.view(mask=r500_interloper_masks[0], figsize=(12, 10), zoom_in=True)
```



3.3.12 Creating a custom mask

You can also create a mask with whatever custom radius you desire, suitable to be applied to either the merged images or one of the individual observation images. You may also define custom coordinates to centre the new mask on, as well as defining both inner and outer radii (if you wish to make an annulus). I shall start by making a mask with an outer radius of 0.01 degrees (though if the source in question has redshift information you can also use a proper distance unit) for the combined image of the demo source:

[24]:
```
custom_mask = demo_src.get_custom_mask(Quantity(0.01, 'deg'))
custom_mask
```

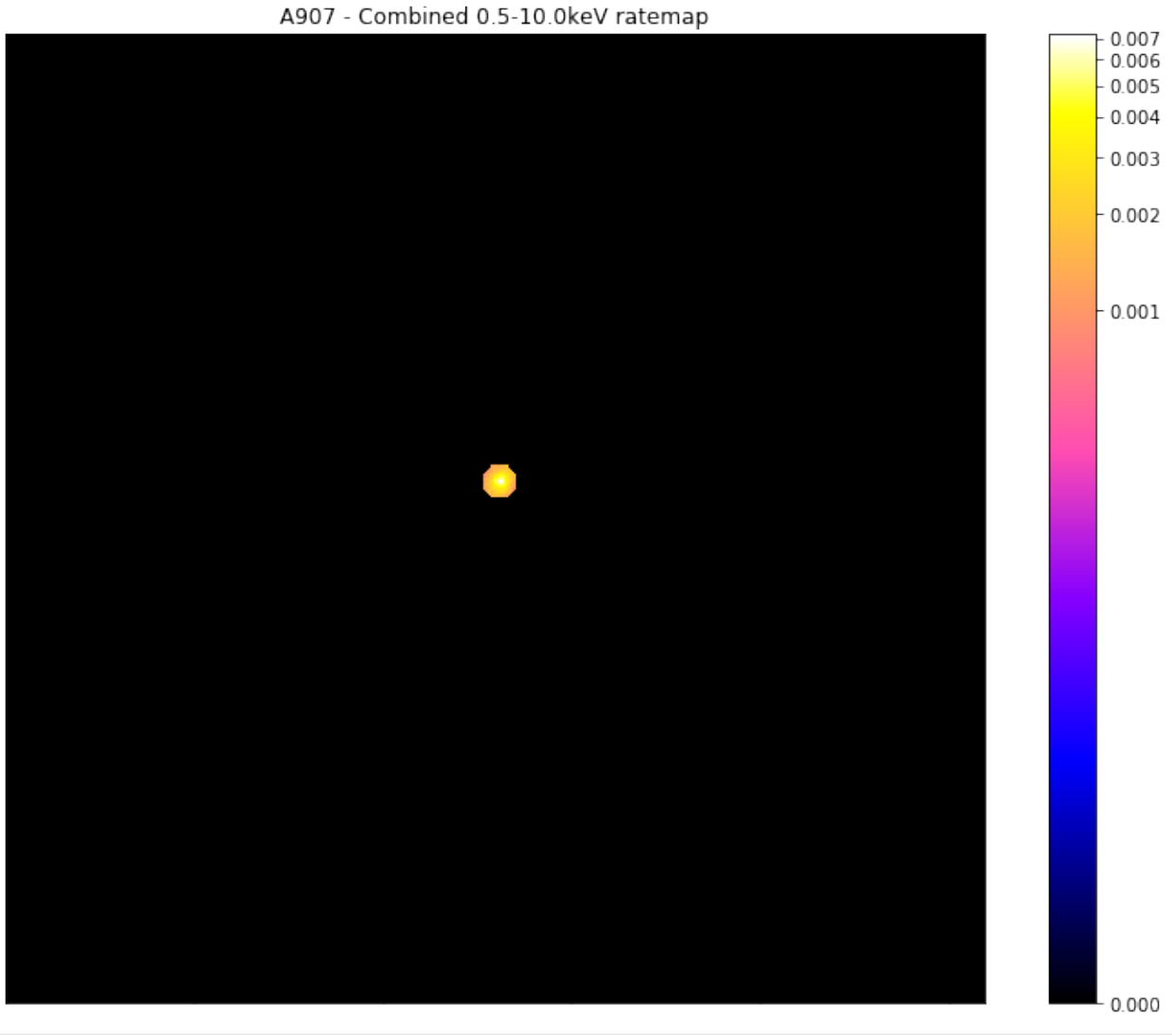
[24]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

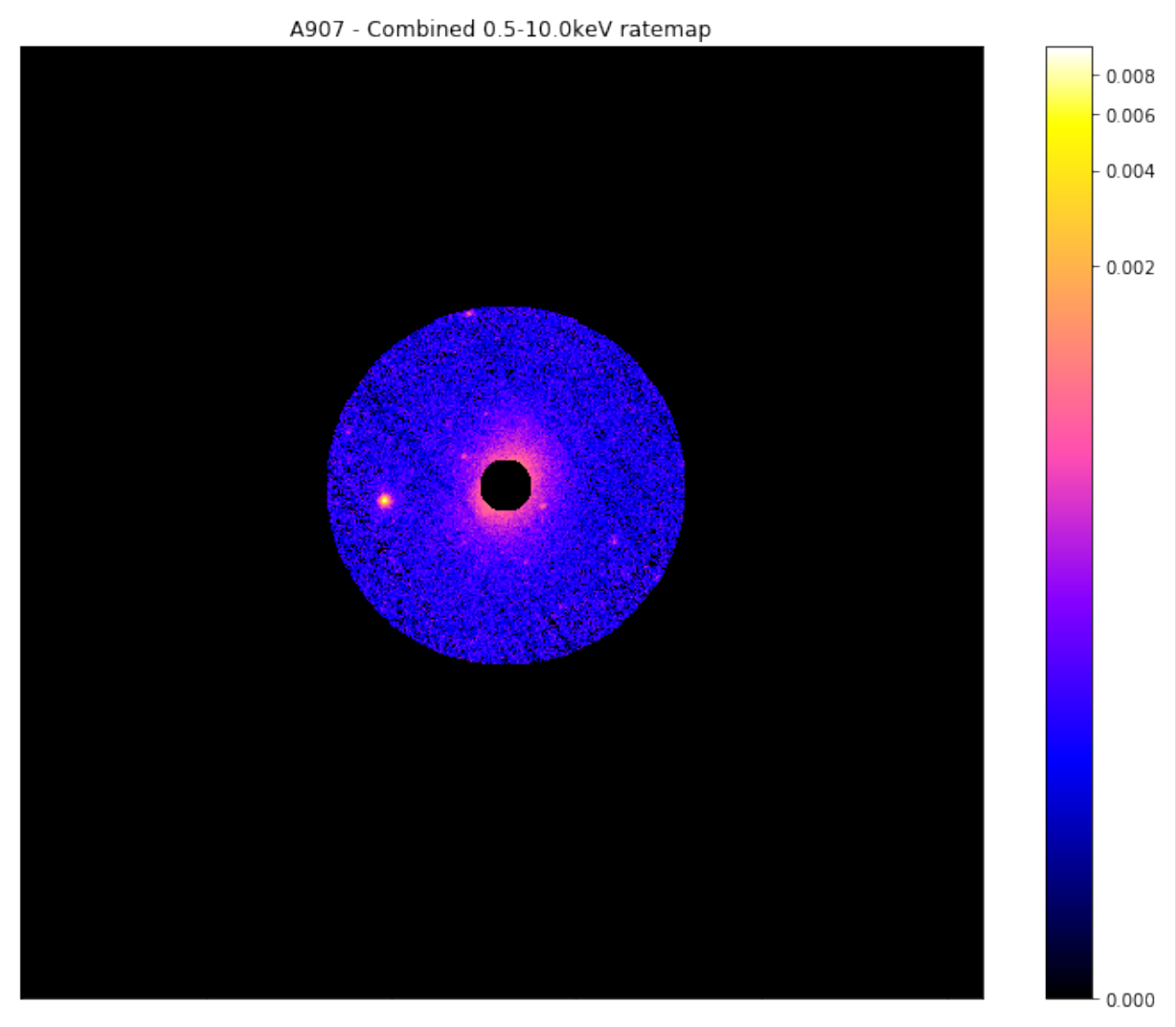
```
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

[25]:
```
chosen_comb_rt.view(mask=custom_mask)
```



Now I shall make a mask with the core of this galaxy cluster excised (the mask shall allow everything from 0.15-1R_{500}), but also I shall decide that I no longer want to remove interloper sources:

[26]:
```
custom_mask = demo_src.get_custom_mask(demo_src.r500, demo_src.r500*0.15, remove_
↪interlopers=False)
chosen_comb_rt.view(mask=custom_mask)
```



3.3.13 Measuring signal-to-noise

Measuring the signal to noise of a given source is a fairly common task in astronomy, and an ability which has been implemented and added to the photometric product classes in XGA. It is a simple matter to quickly find the signal to noise of a galaxy cluster within an overdensity radius for instance, you need only call the `info()` method. There you will find that XGA has automatically measured the signal-to-noise within the supplied overdensity radii:

[27]:
```
demo_src.info()
```

```
------------------------------------------------------
Source Name - A907
User Coordinates - (149.59209, -11.05972) degrees
X-ray Peak - (149.59251340970866, -11.063958320861634) degrees
nH - 0.0534 1e+22 / cm2
Redshift - 0.16
XMM ObsIDs - 3
PN Observations - 3
```

(continues on next page)

(continued from previous page)

```
MOS1 Observations - 3
MOS2 Observations - 3
On-Axis - 3
With regions - 3
Total regions - 69
Obs with one match - 3
Obs with >1 matches - 0
Images associated - 27
Exposure maps associated - 27
Combined Ratemaps associated - 2
Spectra associated - 0
R200 - 1700.0 kpc
R200 SNR - 230.23
R500 - 1200.0 kpc
R500 SNR - 251.61
------------------------------------------------------
```

If, however, you wish to retrieve the signal to noise programmatically (or measure it within a custom radius), you can simply use the `get_snr` method, which I hear use to retrieve the signal to noise of this cluster within the R_{500} and within a custom radius of 300kpc:

[28]:
```
r500_snr = demo_src.get_snr('r500')
print(r500_snr)
cust_snr = demo_src.get_snr(Quantity(300, 'kpc'))
print(cust_snr)
```

```
251.61136957214998
206.61225573609303
```

You are also able to perform a simple assessment of which observation associated with a given source is 'the best', by ranking them by signal to noise of the source. Call the `snr_ranking` method, supply it with a radius within which to measure the SNR, and it will return an array of ObsID-instrument combinations, and an array of SNR values, going from worst to best:

[29]:
```
oi_combs, snr_vals = demo_src.snr_ranking('r500')
```

[30]:
```
oi_combs
```

[30]:
```
array([['0201901401', 'mos1'],
       ['0201901401', 'mos2'],
       ['0201901401', 'pn'],
       ['0201903501', 'mos1'],
       ['0201903501', 'mos2'],
       ['0404910601', 'mos2'],
       ['0404910601', 'mos1'],
       ['0404910601', 'pn'],
       ['0201903501', 'pn']], dtype='<U10')
```

[31]:
```
snr_vals
```

[31]:
```
array([ 71.88691358,  71.90096549,  88.79940544,  94.18597822,
        96.22761135, 108.10871705, 108.73900404, 114.58252691,
       124.73613614])
```

3.3.14 Bulk generation of photometric products

If you wished to generate images and exposure maps in a particular energy range for a large number of unrelated observations, then you would use a NullSource object.

Normally, when defining a NullSource object, it would automatically associate every available ObsID with itself, but it is possible to pass a list of specific ObsIDs of interest (which I do in this demo so I don't destroy my laptop):

[32]:
```
# Four randomly select observation that will make up the NullSource
chosen_obs = ['0677630133', '0301900401', '0770581201', '0827201501']
# And defining the source with four unrelated observations associated with it
bulk_src = NullSource(chosen_obs)
```

[33]:
```
# If we run the standard info() method then we can say a very little information␣
↪about the source
bulk_src.info()
```

```
------------------------------------------------------
Source Name - 4Observations
XMM ObsIDs - 4
PN Observations - 4
MOS1 Observations - 4
MOS2 Observations - 4
------------------------------------------------------
```

Now we can generate images and exposure maps for this bulk sample, though we cannot create combined data products through the emosaic tool because it makes no sense - these are four random observations and have no common coordinate on which they could all be combined:

[34]:
```
# And of course we can generate images and exposure maps in the same way as with the␣
↪other sources
evselect_image(bulk_src, lo_en=lo_en, hi_en=hi_en)
eexpmap(bulk_src, lo_en=lo_en, hi_en=hi_en)
```

```
Generating products of type(s) image: 100%|| 12/12 [00:07<00:00,  1.60it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:12<00:00,  3.08s/it]
Generating products of type(s) expmap: 100%|| 12/12 [00:59<00:00,  4.96s/it]
```

[34]:
```
<xga.sources.base.NullSource at 0x7f8bae415880>
```

3.4 Spectroscopy with XGA

This tutorial will show you how to perform basic spectroscopic analyses on XGA sources, starting with how to generate spectra for a source using the SAS interface. Once I have demonstrated how to generate the spectra for a source, I will show how we can use the XSPEC interface to fit a model to the data, and then how we can retrieve the fit results.

I will also demonstrate the use of the spectrum product view method, to produce visualisations of spectra roughly equivalent to those that XSPEC produces.

[1]:
```
from astropy.units import Quantity
import numpy as np
import pandas as pd

from xga.samples import ClusterSample
```

(continues on next page)

(continued from previous page)

```
from xga.sources import PointSource
from xga.sas import evselect_spectrum
from xga.xspec import single_temp_apec, power_law
```

I need to define some sources to help demonstrate, and I've chosen to define both a sample of galaxy clusters (to demonstrate fitting a standard absorbed plasma emission model), and a point source (representing a quasar) to demonstrate the fitting of a redshift dependant absorbed power law).

Again I've chosen four clusters from the XCS-SDSS sample:

[2]:
```
# Setting up the column names and numpy array that go into the Pandas dataframe
column_names = ['name', 'ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']
cluster_data = np.array([['XCSSDSS-124', 0.80057775, -6.0918182, 0.251, 1220.11, 1777.
↪06, 109.55, 4.49],
                         ['XCSSDSS-2789', 0.95553986, 2.068019, 0.11, 1039.14, 1519.
↪79, 38.90, 2.83],
                         ['XCSSDSS-290', 2.7226392, 29.161021, 0.338, 935.58, 1359.37,
↪ 105.10, 5.99],
                         ['XCSSDSS-134', 4.9083898, 3.6098177, 0.273, 1157.04, 1684.
↪15, 108.60, 4.79]])

# Possibly I'm overcomplicating this by making it into a dataframe, but it is an␣
↪excellent data structure,
#  and one that is very commonly used in my own analyses.
sample_df = pd.DataFrame(data=cluster_data, columns=column_names)
sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']] = \
    sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']].
↪astype(float)

# Defining the sample of four XCS-SDSS galaxy clusters
demo_smp = ClusterSample(sample_df["ra"].values, sample_df["dec"].values, sample_df["z
↪"].values,
                         sample_df["name"].values, r200=Quantity(sample_df["r200"].
↪values, "kpc"),
                         r500=Quantity(sample_df["r500"].values, 'kpc'),␣
↪richness=sample_df['richness'].values,
                         richness_err=sample_df['richness_err'].values)
```

```
Declaring BaseSource Sample: 100%|| 4/4 [00:01<00:00,  2.89it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:13<00:00,  3.27s/it]
Generating products of type(s) image: 100%|| 4/4 [00:00<00:00,  7.78it/s]
Generating products of type(s) expmap: 100%|| 4/4 [00:00<00:00,  8.28it/s]
Setting up Galaxy Clusters: 100%|| 4/4 [00:03<00:00,  1.26it/s]
```

Here I'm defining a point source object for a quasar at z~0.9, this object has no particular interest or significance (at least not to me), but it will be used to demonstrate the fitting of a powerlaw by XSPEC:

[3]:
```
demo_src = PointSource(182.2416, 45.67667, 0.9254, name='PG1206+459')
```

```
Generating products of type(s) ccf: 100%|| 1/1 [00:08<00:00,  8.86s/it]
```

3.4.1 Generating spectra for our sources

It is just as simple to generate spectra (both for sample objects and source objects) as it is to generate the photometric products that we used in the last tutorial. The primary extra information required by evselect_spectrum is a region in which to generate the spectra, for a galaxy cluster it might be an overdensity radius (like R_{200} for instance), for a point source we would use the 'point' region, as defined on point source initilisation with the `point_radius` keyword argument.

You can also set the level of grouping that you want to be applied to the spectrum (by the XMM SAS routine specgroup), using the `min_counts`, `min_sn`, and `over_sample` keyword arguments.

If the `one_rmf` keyword argument is set to True (as is the default) then only one RMF per ObsID-instrument combo will be generated, and will be used by any future spectrum generation for the ObsID-instrument combo. RMFs do change slightly with position on detector, but are also very computationally expensive to produce (relative to the other spectrum generation tasks), as such this is an acceptable compromise.

[4]:
```
# For the quasar, we want to use the point region type
demo_src = evselect_spectrum(demo_src, 'point')

# Whereas for my sample of clusters I've decided I want to use the R500 regions
demo_smp = evselect_spectrum(demo_smp, 'r500')
```

```
Generating products of type(s) spectrum: 100%|| 3/3 [01:46<00:00, 35.44s/it]
Generating products of type(s) spectrum: 100%|| 12/12 [54:54<00:00, 274.52s/it]
```

3.4.2 Generating core-excised spectra for our galaxy clusters

It is not unusual to want to measure quantities from spectra where the core of the galaxy cluster in question has been removed, though I will not go into the reasons for that here. It is not difficult to generate spectra like this in XGA, you simply pass an inner radius (that by default is set to zero), when calling evselect_spectrum:

[5]:
```
demo_smp = evselect_spectrum(demo_smp, demo_smp.r500, demo_smp.r500*0.15)
```

```
Generating products of type(s) spectrum: 100%|| 12/12 [38:22<00:00, 191.84s/it]
```

3.4.3 A note on ARF generation

The Auxiliary Response File (ARF), is an extremely important product which describes the change of the effective area of the telescope with energy, something that must be understood to try and recover the spectrum that was actually emitted by the object rather than the one that was recieved by the telescope. Calculating the ARF is a fairly complex process which, for XMM, is fully described in the SAS documentation, and as such the SAS *arfgen* procedure has numerous configuration options for the user to set.

XGA chooses to generate ARFs in two different ways, depending on whether the source being analysed is extended or point-like. For extended sources we generate a 'detector map', which is essentially an image of the source in detector coordinates, and acts to weight the calculation of the ARF curve by the emission of the source. For point sources this is not valid or necessary, so we do not use a detector map, and we set the `extendedsource` argument to False.

The use of the detector map to weight ARF generation for extended sources does extend the runtime, so do not be surprised if generating many spectra takes hours.

3.4.4 How do the XSPEC fits work?

As XGA finds all available data relevant to a given source, we would ideally perform a model fit using every single spectrum that we generate, in order to get the best constraints. Unfortunately, we find that it is not always wise to use every single spectrum available, as if one or more are low quality, they can drag the whole fit down with them.

As such we implemented a cleaning stage, almost identical to that used in the existing XCS luminosity-temperature pipeline, where (optionally) XSPEC fits can be performed on all of the spectra individually, and if certain quality requirements aren't met then that spectrum won't be included in the final, simultaneous, fit. More information about the default quality checks for the single_temp_apec function can be found in the documentation, and the checks can be disabled by setting `spectrum_checking=False`. The power_law function does not currently have any spectrum checking mechanism.

I have written a general XSPEC fitting script that in theory can deal with any model setup, and which is populated by XGA before being run for a specific source. Then the fit, luminosity, and plotting values are extracted and read into XGA. We simultaneously fit all valid spectra with the same model, making sure to link the key parameters so that the values are the result of minimising the difference of a single model to all the data.

Just as when we generate SAS products, this process runs in parallel, with four different source fits all being executed on different cores (for example), this is especially useful when analysing large samples of objects.

The default models implemented in XGA (an absorbed plasma model, and an absorbed powerlaw) both have an extra multiplicative constant, which helps us to deal with the different normalisations of spectra from different instruments and observations. The normalisations of all the spectra are linked, then the constant of the first spectrum in the dataset is frozen at 1, and all other constants are allowed to vary freely.

3.4.5 Fitting models to our sources

For the quasar I've chosen a powerlaw, rather than any more complex model; there are two types of power law available to fit in the XGA XSPEC interface, an absorbed redshift dependant powerlaw (constant*tbabs*zpowerlw), and an absorbed redshift independent powerlaw (constant*tbabs*powerlaw). You can choose which model to fit using the `redshifted` keyword argument when power_law (imported from xga.xspec) is called, though of course only sources with redshift information are allowed to use the zpowerlw version.

For those sources which have redshift information, luminosity values will always be calculated after a fit is performed. A call to an XGA XSPEC function allows you to choose which energy band to calculate the luminosity in (the defaults are between 0.5-2.0keV and between 0.01-100keV).

The user may also choose a photon index to use as the start value for the XSPEC fit, using the `start_pho_index` parameter. The energy range of the data that is to be fit to can be selected with the `lo_en` and `hi_en` parameters, but passing Astropy quantities (the default range is 0.3-7.9keV). The hydrogen column value is taken from the source object and is (by default) frozen during the fit, this behaviour can be changed with the `freeze_nh` keyword argument.

[6]:
```
demo_src = power_law(demo_src, 'point', redshifted=True)
```
```
Running XSPEC Fits: 100%|| 1/1 [00:00<00:00,  1.28it/s]
```

The model I have chosen to fit my clusters with is an absorbed APEC plasma emission model (constant*tbabs*apec), and the options you can pass into the function call are very similar to those of power_law. Start values for the temperature and metallicity of the plasma may be passed, along with parameters which will cause the nH and metallicity values to be frozen or unfrozen during the fit process:

[7]:
```
demo_smp = single_temp_apec(demo_smp, 'r500')
```
```
Running XSPEC Fits: 100%|| 4/4 [00:16<00:00,  4.16s/it]
```

Seeing as I generated core-excised spectra for these galaxy clusters earlier, I will also fit the same model to those. It is worth noting here that if I hadn't already generated those core-excised spectra then this convenience function would have noticed that and generated them for me:

[8]:
```
demo_smp = single_temp_apec(demo_smp, demo_smp.r500, demo_smp.r500*0.15)
```

```
Running XSPEC Fits: 100%|| 4/4 [00:20<00:00,  5.11s/it]
```

3.4.6 Saving fits to disk and loading them in again

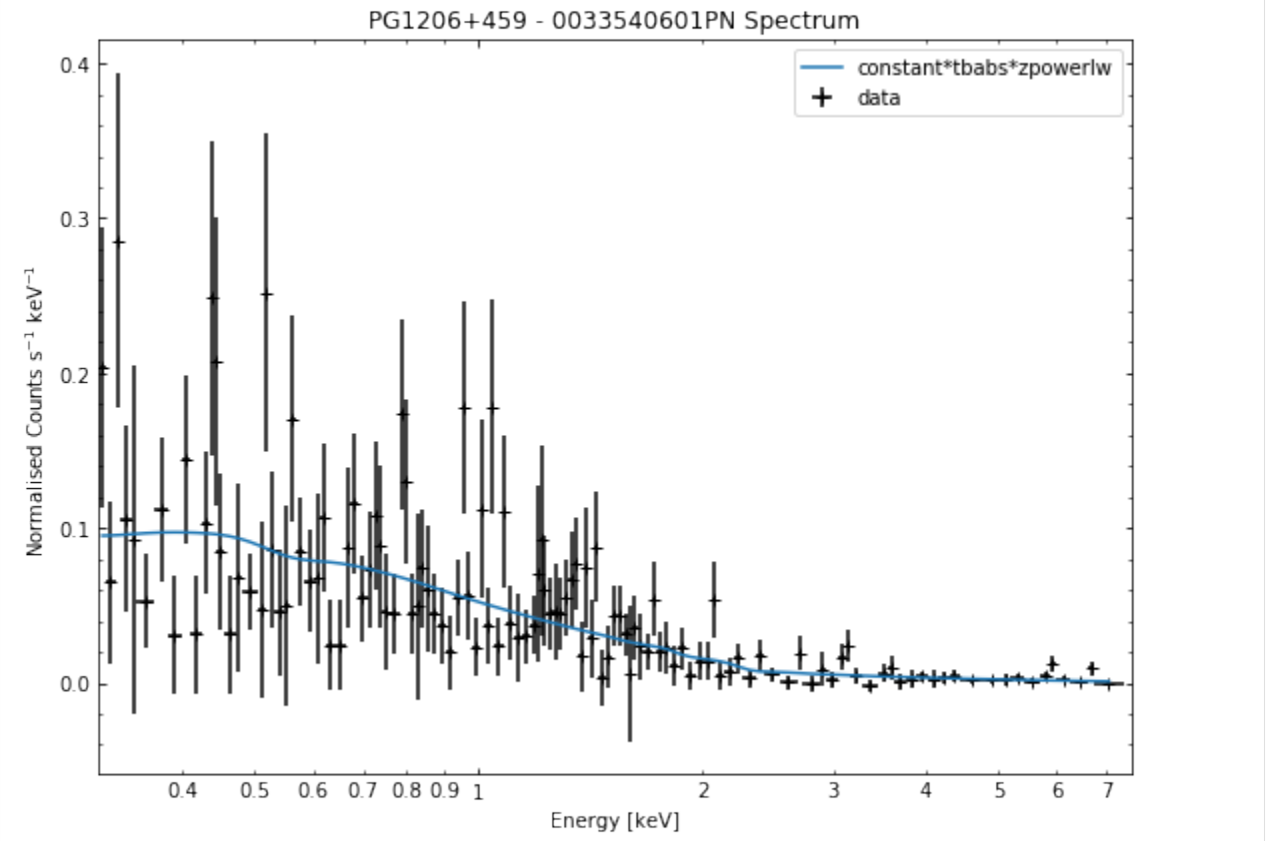
Fit results are always saved to disk, and reside in the 'XSPEC' sub-directory of the 'xga_save_path' that is set in the configuration file. There they will be saved in another sub-directory with the name of the source in question.

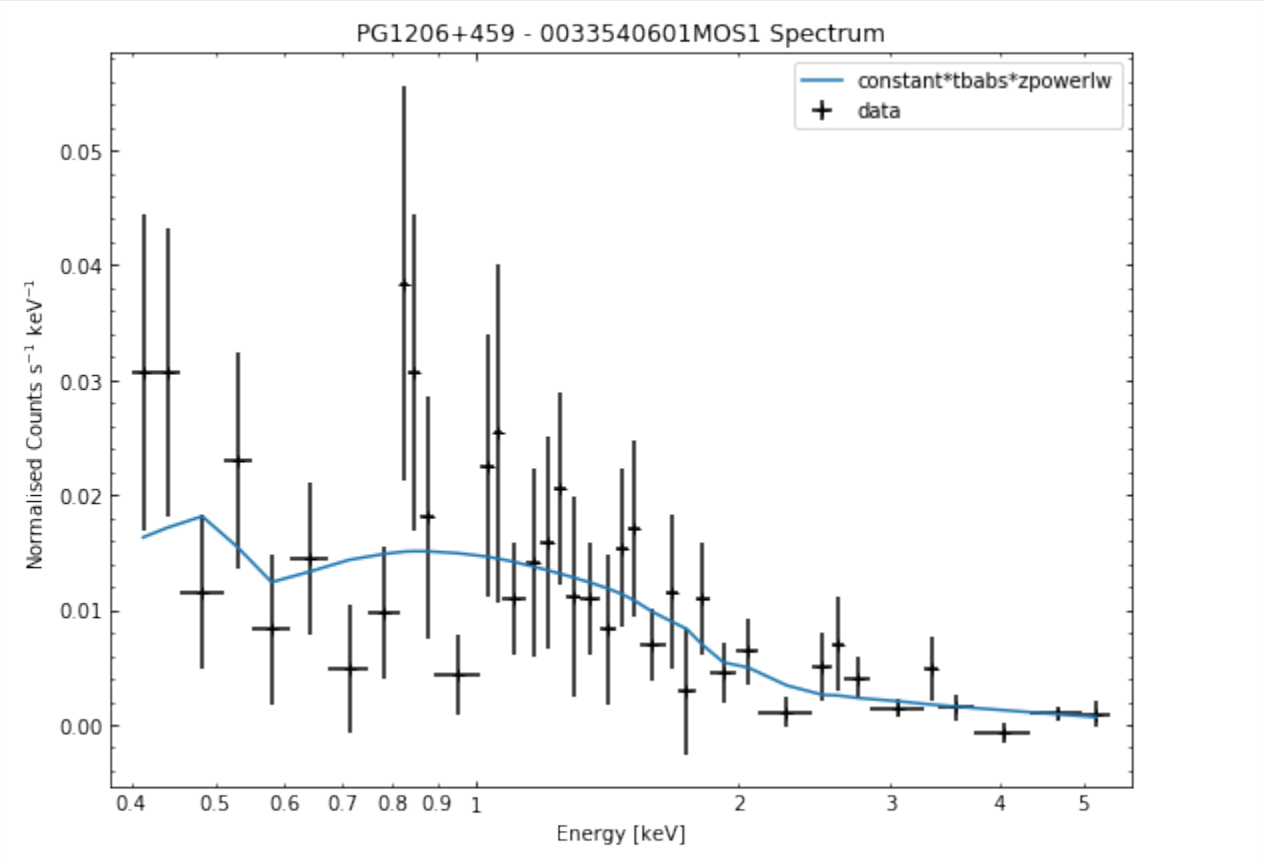
If you decide to declare the same source again after you have already run a fit on it, you can choose to load the old fit in by setting the `load_fits` keyword argument to True on declaration of the source object. By default this is False however.

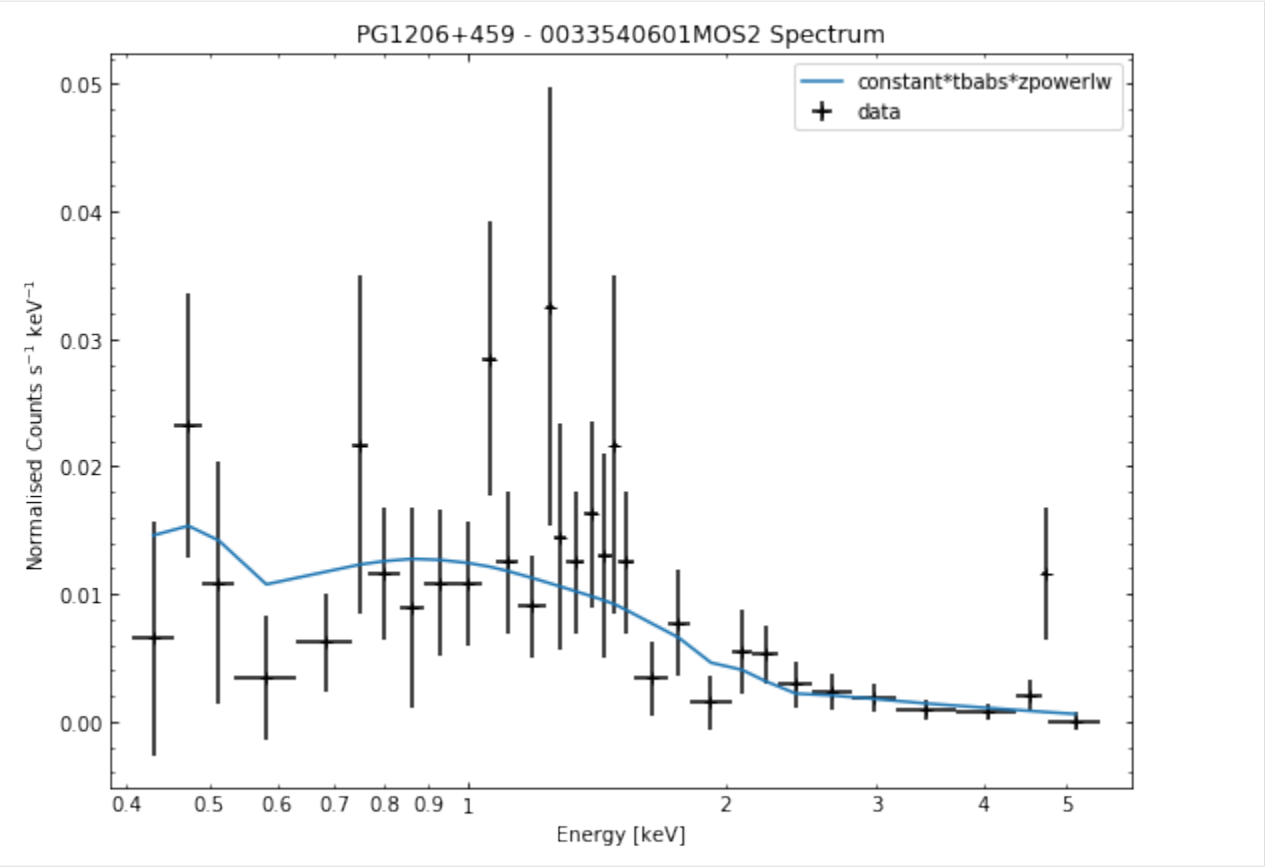
3.4.7 Viewing fitted spectra

Its just as easy to visualise a spectrum as it is to visualise the photometric products we explored in the last tutorial. We simply use the view() method on a spectrum object and a data + model plot will be shown:

[9]:
```
# Cycle through the spectra for the quasar and call their view methods
for sp in demo_src.get_spectra('point'):
    sp.view()
```

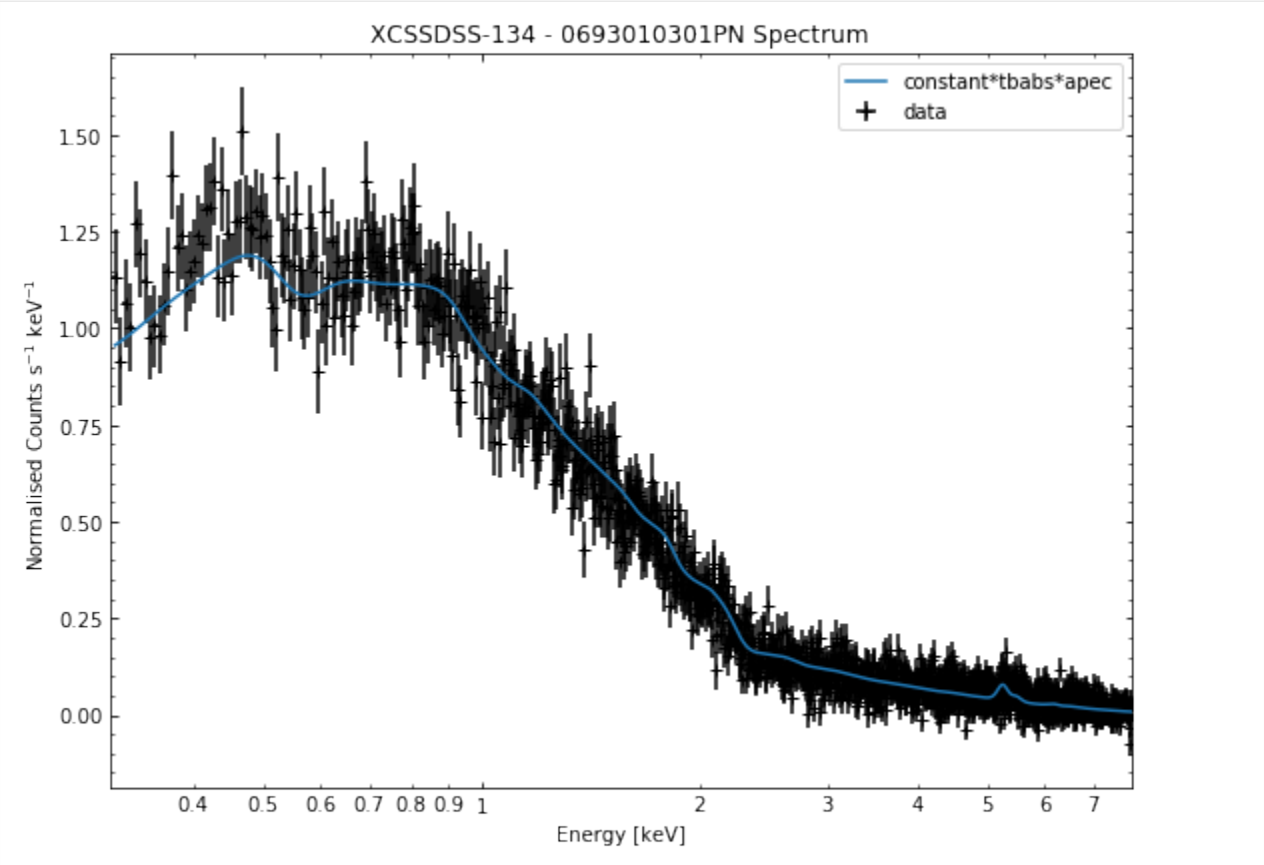


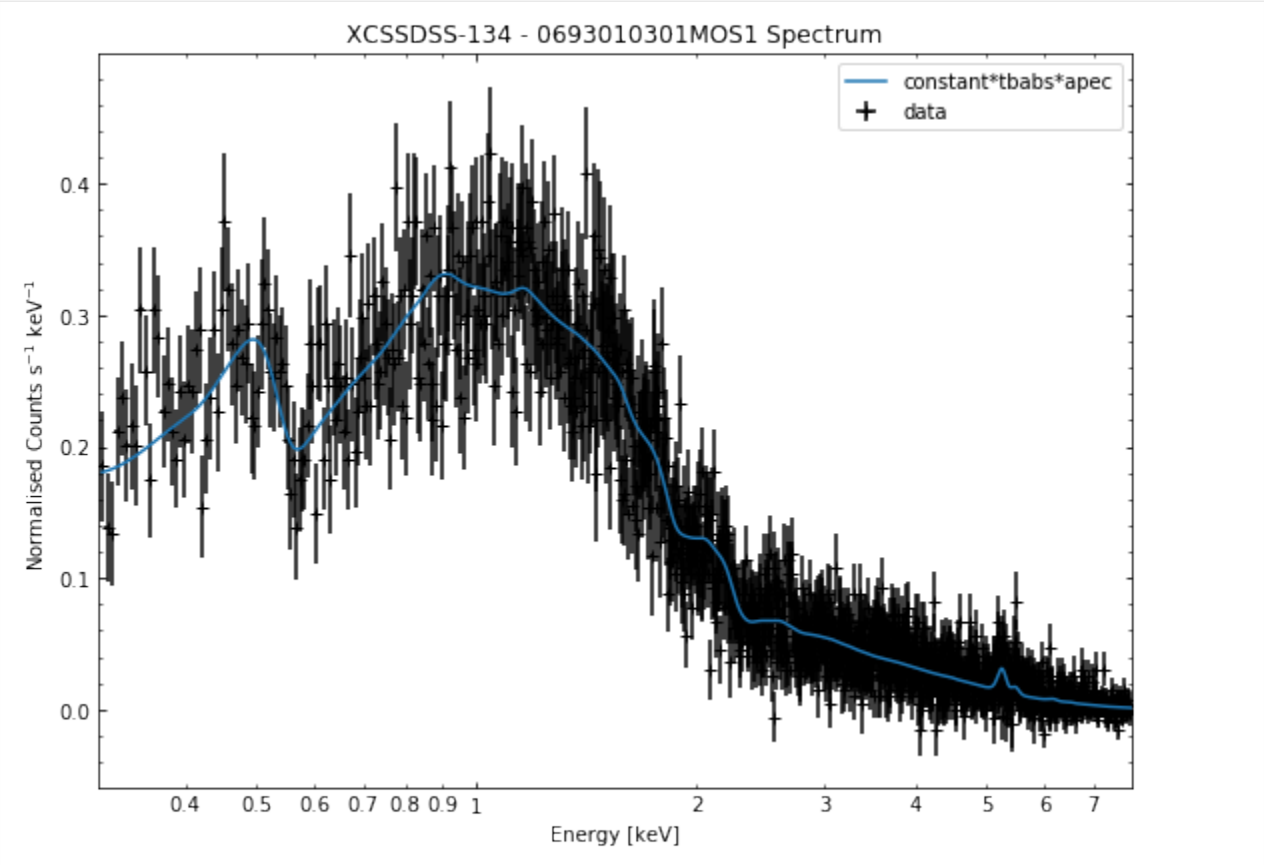

PG1206+459 - 0033540601MOS1 Spectrum
constant*tbabs*zpowerlw
data
Normalised Counts s^{-1} keV^{-1}
0.05
0.04
0.03
0.02
0.01
0.00
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
Energy [keV]

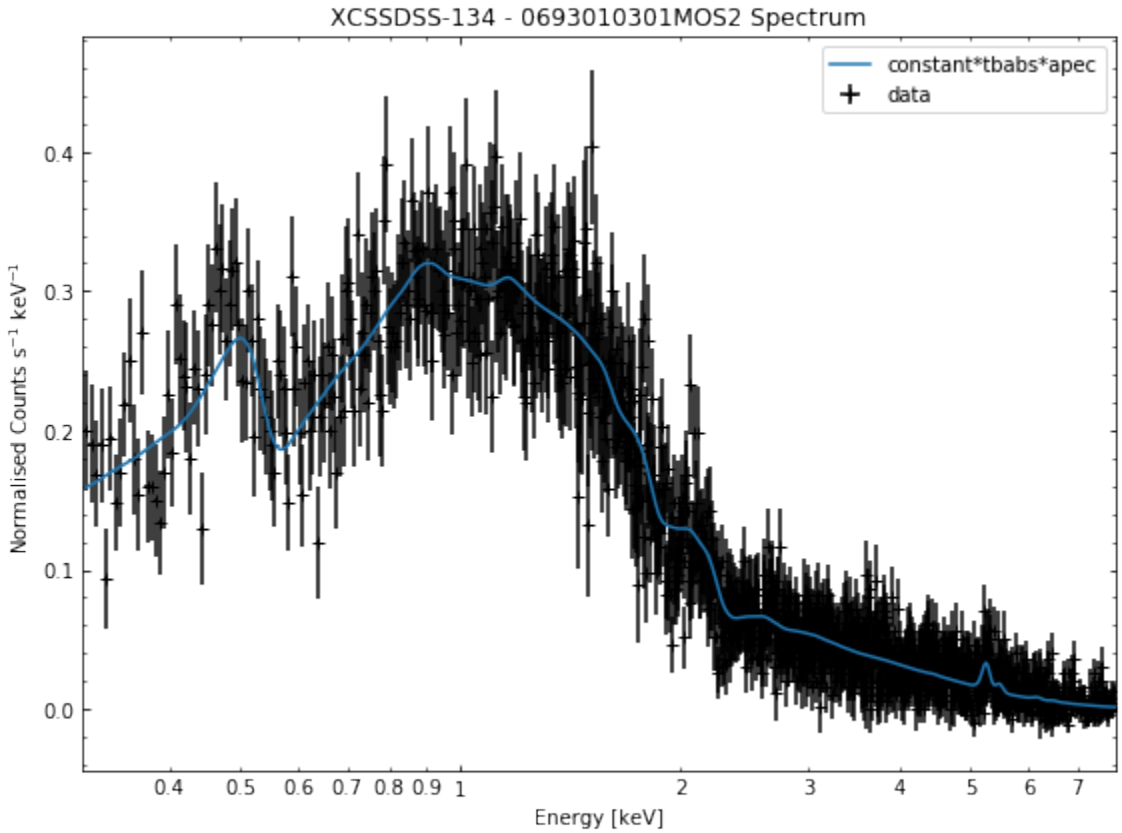


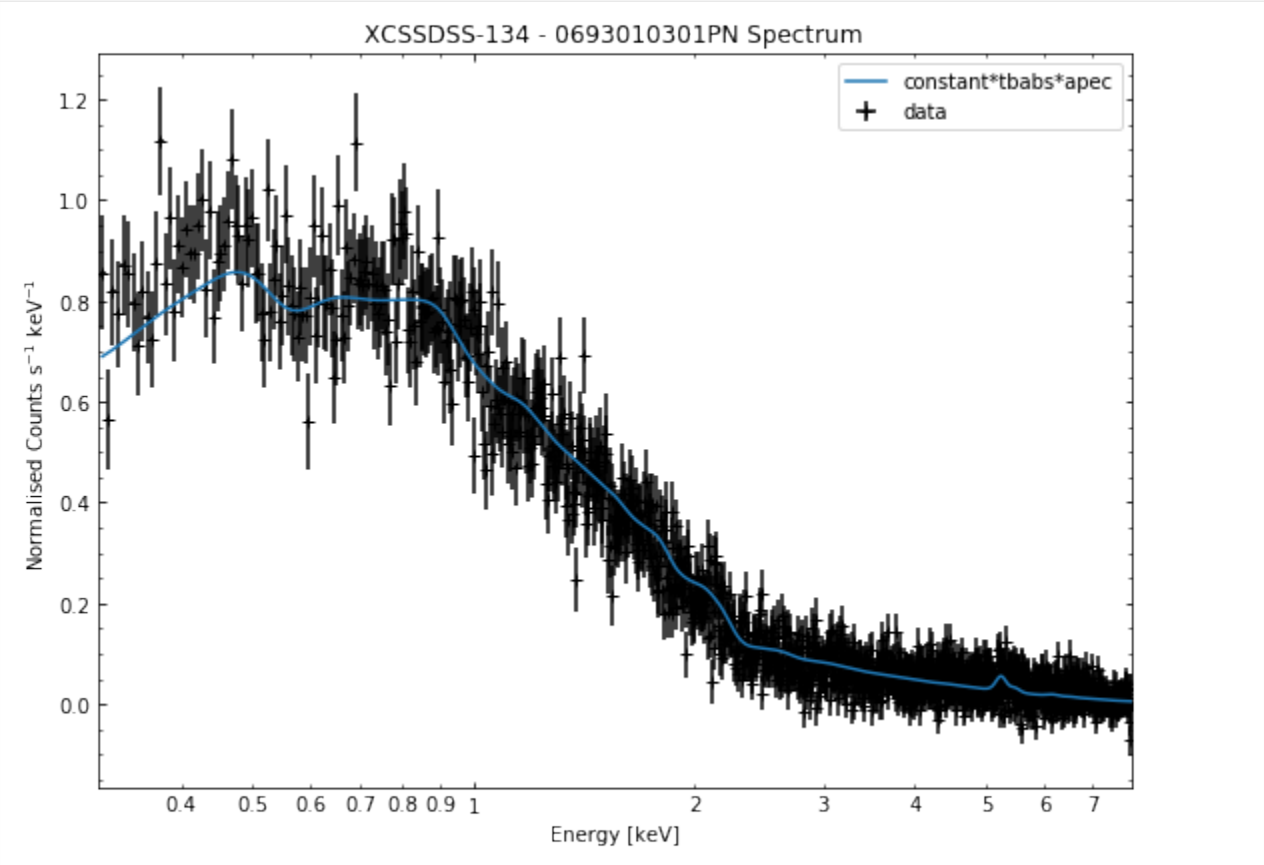
We do the same thing for the last cluster in our sample (XCSSDSS-134), both for the normal R_{500} and core-excised spectra:

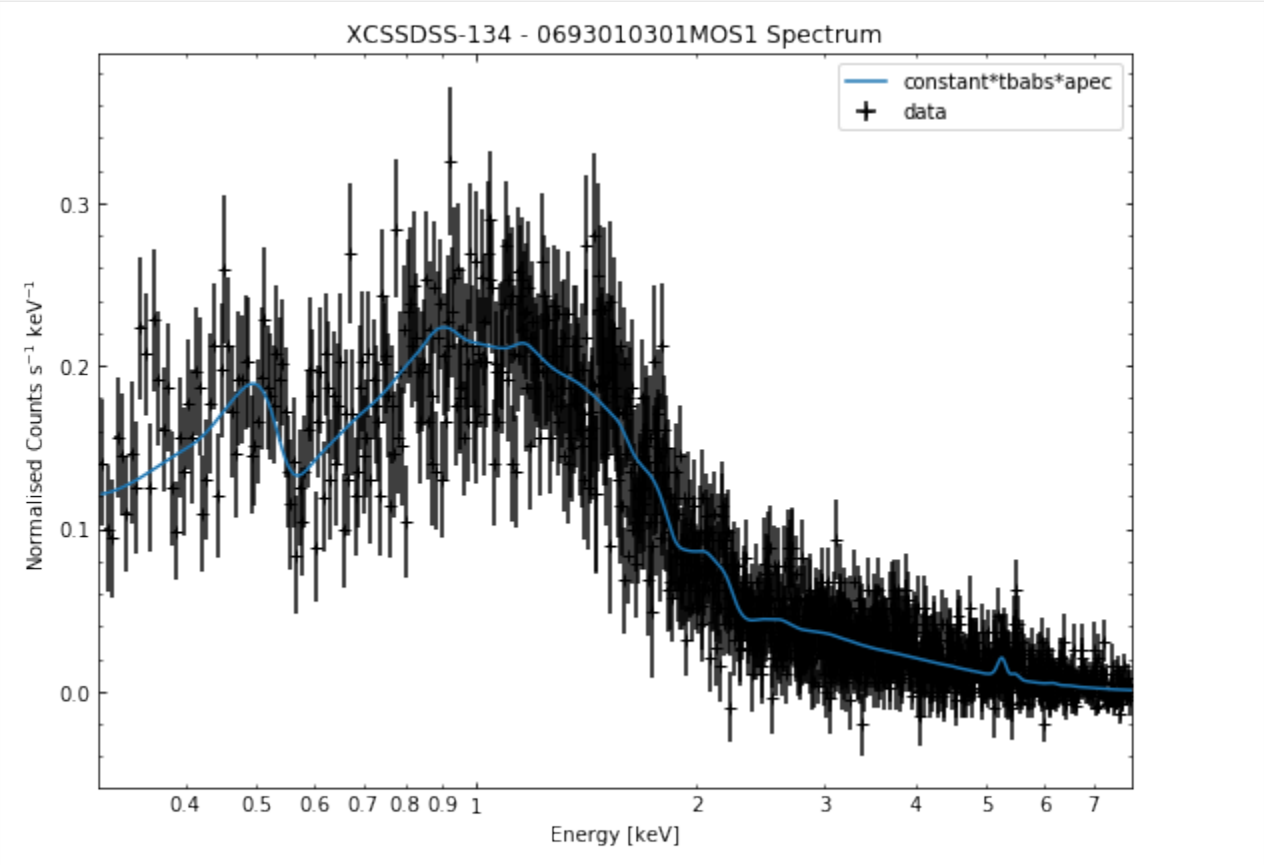
[10]:
```
# Going through the R500 spectra
for sp in demo_smp[3].get_spectra('r500'):
    sp.view()
print('\n\n\n')

# Going through the core-excised spectra
for sp in demo_smp[3].get_spectra(demo_smp[3].r500, inner_radius=demo_smp[3].r500*0.
↪15):
    sp.view()
```

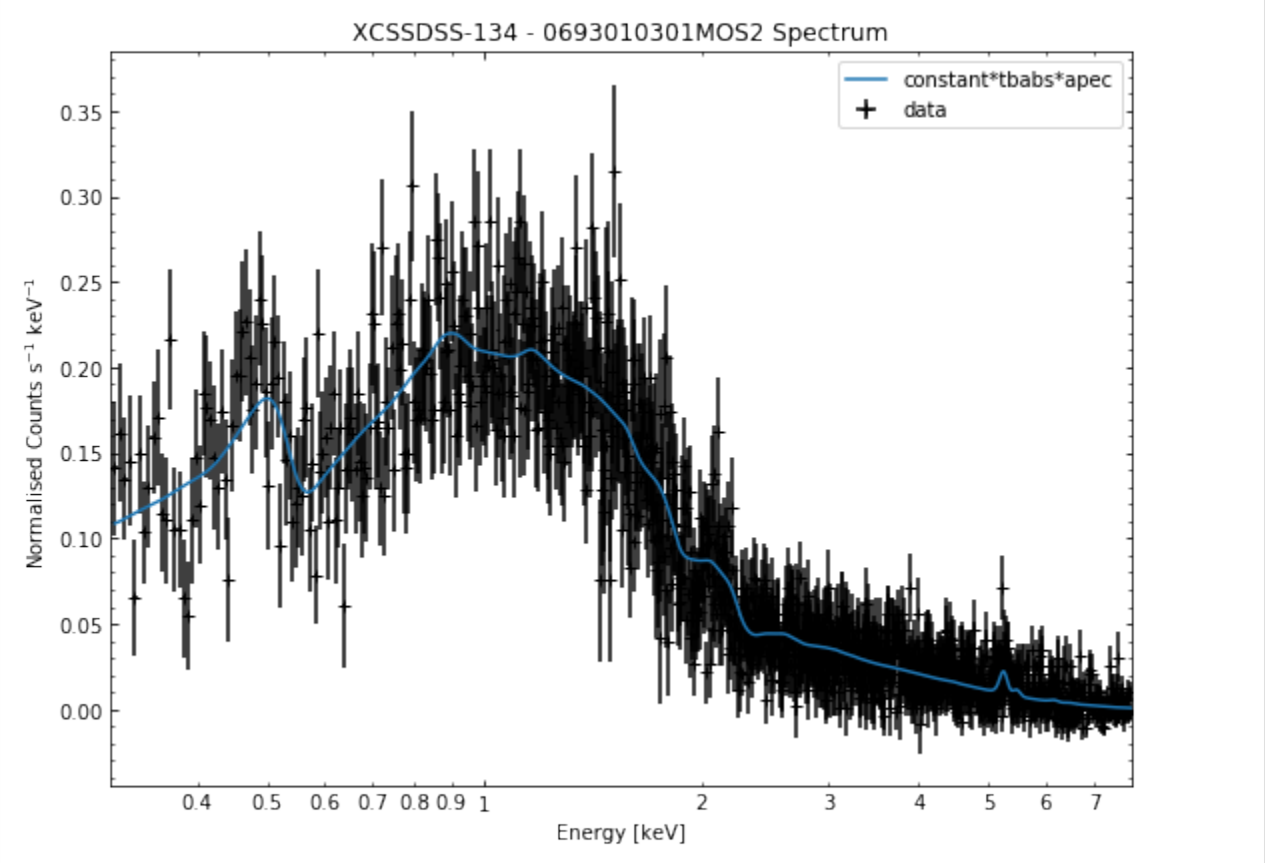

XCSSDSS-134 - 0693010301PN Spectrum
constant*tbabs*apec
data
Normalised Counts s^{-1} keV^{-1}
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7
Energy [keV]


XCSSDSS-134 - 0693010301MOS1 Spectrum
constant*tbabs*apec
data
Normalised Counts s^{-1} keV^{-1}
0.0
0.1
0.2
0.3
0.4
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7
Energy [keV]


XCSSDSS-134 - 0693010301MOS2 Spectrum
constant*tbabs*apec
data
Normalised Counts s^{-1} keV^{-1}
0.0
0.1
0.2
0.3
0.4
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7
Energy [keV]


XCSSDSS-134 - 0693010301PN Spectrum
constant*tbabs*apec
data
Normalised Counts s^{-1} keV^{-1}
Energy [keV]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7


XCSSDSS-134 - 0693010301MOS1 Spectrum
constant*tbabs*apec
data
Normalised Counts s^{-1} keV^{-1}
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7
Energy [keV]



3.4.8 General retrieval of fit parameters

Once we have performed a fit, we will almost certainly want to know what the values of the fit parameters (as well as the uncertainties on those parameters). It is a simple matter to retrieve these fit values, you only need to supply the name of the model and the region the fit was run on.

This information is passed into the 'get_results' method of a source object, and a dictionary of parameter values + uncertainties will be returned. For the quasar we have analysed as an example, the 'PhoIndex' entry is a numpy array with three values, the first column is the parameter value, the second column is the -error on the value, and the third column is the +error.

The 'factor' entry in the dictionary is the multiplicative constant we use to deal with differing normalisations of the separate spectra, there are only two entries (rather than the three you might expect due to there being three spectra) because one of them was set to one and frozen:

[11]:
```
demo_src.get_results('point', 'constant*tbabs*zpowerlw')
```

[11]:
```
{'PhoIndex': array([1.78984   , 0.07400441, 0.07558034]),
 'norm': array([1.68883000e-04, 1.27526738e-05, 1.31476429e-05]),
 'factor': array([[0.988555  , 0.10202081, 0.11219239],
        [0.821484  , 0.09475646, 0.10438363]])}
```

We can also use the exact same method on a GalaxyCluster source we get from our ClusterSample object. It is important to note that only parameters that are allowed to vary during the fit are saved in the fit paramater dictionaries. If, when we called 'single_temp_apec', we had allowed the nH and metalicity to vary, they would also have entries in the dictionary returned below:

[12]:
```
demo_smp[3].get_results('r500', 'constant*tbabs*apec')
```

[12]:
```
{'kT': array([6.66601   , 0.10205891, 0.10217485]),
 'norm': array([5.7889900e-03, 3.0805578e-05, 3.0887788e-05]),
 'factor': array([[1.0043    , 0.00974371, 0.00984066],
        [1.0067    , 0.00969691, 0.00980753]])}
```

Though do note that for a GalaxyCluster source object the model is set by default to constant*tbabs*apec, for convenience' sake, so you could just call `get_results` like this (I've added the inner_radius information so we're retrieving the results for the core-excised spectra):

[13]:
```
demo_smp[3].get_results('r500', inner_radius=demo_smp[3].r500*0.15)
```

[13]:
```
{'kT': array([6.33982   , 0.13287747, 0.13300562]),
 'norm': array([4.0435000e-03, 2.6871529e-05, 2.6958326e-05]),
 'factor': array([[0.970026  , 0.01215592, 0.01231316],
        [0.983651  , 0.01210961, 0.01226908]])}
```

3.4.9 Parameters from GalaxyCluster and ClusterSample objects

Galaxy Clusters are my particular area of study, and the source objects that I most commonly analyse with XGA. As such I have built in certain methods to make the retrieval of things like the intra-cluster medium temperature as simple as possible, both from sources directly and samples of clusters.

Again the only information we need to provide are the name of the model and the region that the fit was run on. The method to use is `get_temperature`, and the default model it tries to get values for is 'constant*tbabs*apec':

[14]:
```
# We take an individual GalaxyCluster source and retrieve its temperature and -+␣
↪temperature errors
print(demo_smp[3].get_temperature('r500'))
```

```
[6.66601    0.10205891 0.10217485] keV
```

When dealing with a sample of Galaxy Clusters however, we often wish to know the properties of the entire population, in which case we can use the 'Tx' method of the ClusterSample. Again the analysis radius must be passed, but the model is set by default to 'constant*tbabs*apec':

[15]:
```
# This retrieves the T500 values (measured with constant*tbabs*apec) for the whole␣
↪sample
print(demo_smp.Tx('r500'))
```

```
[[7.02059    0.12973252 0.12976439]
 [4.67254    0.08191512 0.08205011]
 [5.07272    0.27327653 0.27598913]
 [6.66601    0.10205891 0.10217485]] keV
```

3.4.10 Retrieving L_x values from sources and samples

As I mentioned earlier, if a source has a redshift associated with it then L_X values will automatically be measured as part of the fitting process. The use of X-ray luminosities in analysis is extremely common in X-ray astronomy, so I have introduced methods to retrieve luminosities from any source or sample object, in much the same way as we retrieve T_x values from GalaxyCluster and ClusterSample objects.

Seeing as L_X values are measured within an energy band, I am defining the lower and upper limits of the measurement that I want to retrieve (as Astropy quantities), in this case the bolometric luminosity. If I didn't supply specific energy

limits then the 'get_luminosities' method would supply all L_X measurements for the region-model combination in question.

Just as with the parameter dictionaries, the first column is the L_X value, the second is the -error, and the third is the +error:

[16]:
```
lo_en = Quantity(0.01, 'keV')
hi_en = Quantity(100.0, 'keV')
```

[17]:
```
# Grabbing the bolometric luminosity of the quasar we have investigated
print(demo_src.get_luminosities('point', 'constant*tbabs*zpowerlw', lo_en=lo_en, hi_
↪en=hi_en), '\n')

# And all the r500 luminosities of the first cluster in our sample
print(demo_smp[0].get_luminosities('r500', 'constant*tbabs*apec'))
```

```
[2.28052016e+45 2.30134509e+44 1.62657738e+44] erg / s

{'bound_0.5-2.0': <Quantity [5.78680953e+44, 2.94296973e+42, 2.74997150e+42] erg / s>,
↪ 'bound_0.01-100.0': <Quantity [2.23554673e+45, 1.63491236e+43, 1.60168515e+43] erg␣
↪/ s>}
```

In much the same way we can retrieve L_X values for an entire sample, and please note that this method can be used with any type of sample, not just ClusterSample objects:

[18]:
```
# Bolometric luminosities for our sample of clusters
print(demo_smp.Lx('r500', 'constant*tbabs*apec', lo_en=lo_en, hi_en=hi_en))
```

```
[[2.23554673e+45 1.63491236e+43 1.60168515e+43]
 [3.74006375e+44 3.52762889e+42 3.06791738e+42]
 [9.55726922e+44 2.83318399e+43 2.93291919e+43]
 [1.87930718e+45 1.40294002e+43 1.49164499e+43]] erg / s
```

A final note on XGA's L_X values. When XSPEC performs a simultaneous fit to multiple spectra, and then a luminosity is measured, it actually measures luminosities individually for all the spectra. As such, I had to make a decision about which L_X values I wanted XGA to report when requested from a source or sample object.

I chose to preferentially report a PN luminosity (because of the greater sensitivity compared to the MOS cameras). If a PN spectra was not available, then a MOS2 luminosity is taken instead, and if MOS2 wasn't available then a MOS1 luminosity will be used.

3.4.11 Generating annular spectra

The generation and use of annular spectra is dealt with in an advanced tutorial.

3.5 Generating Scaling Relations with XGA

This tutorial will explain how to generate XGA scaling relations objects, and go through some of their key useful features. Generating scaling relations that map difficult-to-measure parameters onto values that are easier to determine is very common in cluster science, with mass-temperature, luminosity-temperature, and luminosity-richness relations being just a few that spring to mind.

As such I created this class of product, ScalingRelation, as well as several independent fitting method to generate them from arbitrary data. This part of XGA can almost be viewed as a submodule within the main module, as you do not have to use the source and sample structure to generate relations, the data can be supplied from an external source, and the fitting process will work exactly the same.

[1]:
```
import pandas as pd
from astropy.units import Quantity, pix
from astropy.cosmology import Planck15
import numpy as np

from xga.samples.extended import ClusterSample
from xga.sas import evselect_spectrum, emosaic
from xga.xspec import single_temp_apec
from xga.models.misc import power_law
from xga.relations.fit import scaling_relation_lira, scaling_relation_odr
from xga.relations.clusters import LT, L, MT
from xga.products.relation import ScalingRelation
```

3.5.1 Relation quality disclaimer

All of the relations we generate here are for demonstration purposes only, they **should not** be used for any real scientific purposes.

3.5.2 Defining our sample

I am again using clusters from the XCS-SDSS sample, they will be used to demonstrate the fitting and visualisation of scaling relations from an XGA ClusterSample object. I have read in their information from a file as I did not want to include a large table of irrelevant information in one of the cells of this tutorial. This time I will be using fourty randomly chosen clusters rather than four, as we could not reasonably hope to get a good fit with such a small sample.

[2]:
```
sample = pd.read_csv("modified_xcssdss_sample.csv", header="infer").sample(40)

srcs = ClusterSample(sample["RA_xcs"].values, sample["DEC_xcs"].values, sample["z"].
↪values,
                     sample["name"].values, r500=Quantity(sample["r500"].values,
↪'arcmin'),
                     clean_obs=True, clean_obs_reg="r500", load_fits=False, psf_
↪corr=False,
                     richness=sample['richness'].values, richness_err=sample[
↪'richness_err'].values)
```

```
Declaring BaseSource Sample: 100%|| 40/40 [00:35<00:00,  1.13it/s]
Generating products of type(s) image: 100%|| 3/3 [00:03<00:00,  1.19s/it]
Generating products of type(s) ccf: 100%|| 95/95 [02:43<00:00,  1.72s/it]
Generating products of type(s) expmap: 100%|| 3/3 [00:38<00:00, 12.93s/it]
Generating products of type(s) image: 100%|| 40/40 [01:03<00:00,  1.59s/it]
Generating products of type(s) expmap: 100%|| 40/40 [00:40<00:00,  1.01s/it]
Setting up Galaxy Clusters:   0%|          | 0/40 [00:00<?, ?it/s]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-31144 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:   2%|         | 1/40 [00:10<07:02, 10.83s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/extended.py:180: UserWarning: A point source␣
↪has been detected in 0728170101 and is very close to the user supplied coordinates␣
↪of XCSSDSS-165. It will not be excluded from analysis due to the possibility of a␣
↪mis-identified cool core
  warnings.warn("A point source has been detected in {o} and is very close to the␣
↪user supplied "
```

(continues on next page)

(continued from previous page)

```
Setting up Galaxy Clusters:   5%|         | 2/40 [00:12<05:06,  8.06s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-2 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  10%|         | 4/40 [00:16<02:56,  4.91s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/extended.py:180: UserWarning: A point source␣
↪has been detected in 0601260201 and is very close to the user supplied coordinates␣
↪of XCSSDSS-10401. It will not be excluded from analysis due to the possibility of a␣
↪mis-identified cool core
  warnings.warn("A point source has been detected in {o} and is very close to the␣
↪user supplied "
Setting up Galaxy Clusters:  22%|       | 9/40 [00:23<00:58,  1.87s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-1174 has␣
↪not been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  28%|       | 11/40 [00:27<00:59,  2.06s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:84: UserWarning: There are 1␣
↪alternative matches for observation 0109080301, associated with source XCSSDSS-14715
  warnings.warn("There are {0} alternative matches for observation {1}, associated␣
↪with "
/home/dt237/code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-
↪14715 has not been detected in all region files, so generating and fitting products␣
↪with the 'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  30%|       | 12/40 [00:30<01:00,  2.16s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-8060 has␣
↪not been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  32%|      | 13/40 [00:33<01:03,  2.36s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:84: UserWarning: There are 1␣
↪alternative matches for observation 0404967101, associated with source XCSSDSS-8070
  warnings.warn("There are {0} alternative matches for observation {1}, associated␣
↪with "
/home/dt237/code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-
↪8070 has not been detected in all region files, so generating and fitting products␣
↪with the 'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  38%|      | 15/40 [00:51<02:08,  5.14s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-487 has␣
↪not been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  48%|     | 19/40 [01:00<01:01,  2.93s/it]/home/dt237/
↪code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-7190 has␣
↪not been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
```

(continues on next page)

(continued from previous page)

```
Setting up Galaxy Clusters:  52%|    | 21/40 [01:10<01:09,  3.68s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-4003 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  55%|    | 22/40 [01:13<01:02,  3.48s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-23 has not been␣
↪detected in all region files, so generating and fitting products with the 'region'␣
↪reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  65%|   | 26/40 [01:24<00:35,  2.56s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-382 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  68%|   | 27/40 [01:27<00:36,  2.79s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-2984 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  70%|   | 28/40 [01:47<01:32,  7.74s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/extended.py:180: UserWarning: A point source has␣
↪been detected in 0404190201 and is very close to the user supplied coordinates of␣
↪XCSSDSS-10223. It will not be excluded from analysis due to the possibility of a␣
↪mis-identified cool core
  warnings.warn("A point source has been detected in {o} and is very close to the␣
↪user supplied "
/home/dt237/code/PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-
↪10223 has not been detected in all region files, so generating and fitting products␣
↪with the 'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  75%|  | 30/40 [02:04<01:14,  7.44s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/extended.py:180: UserWarning: A point source has␣
↪been detected in 0760230301 and is very close to the user supplied coordinates of␣
↪XCSSDSS-137. It will not be excluded from analysis due to the possibility of a mis-
↪identified cool core
  warnings.warn("A point source has been detected in {o} and is very close to the␣
↪user supplied "
Setting up Galaxy Clusters:  78%|  | 31/40 [02:07<00:55,  6.13s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-5077 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters:  98%|| 39/40 [02:24<00:02,  2.35s/it]/home/dt237/code/
↪PycharmProjects/XGA/xga/sources/general.py:99: UserWarning: XCSSDSS-30950 has not␣
↪been detected in all region files, so generating and fitting products with the
↪'region' reg_type will not use all available data
  warnings.warn("{n} has not been detected in all region files, so generating and␣
↪fitting products"
Setting up Galaxy Clusters: 100%|| 40/40 [02:28<00:00,  3.70s/it]
Generating products of type(s) image: 100%|| 15/15 [00:03<00:00,  4.06it/s]
```

(continues on next page)

(continued from previous page)

```
Generating products of type(s) expmap: 100%|| 15/15 [00:01<00:00,  7.70it/s]
```

3.5.3 Measuring cluster properties

Just as in our last tutorial 'Spectroscopy with XGA', we're going to generate spectra in the R_{500} region for all the sources, and then fit them with a single temperature absorbed APEC model. This will give us the temperature and luminosity of all the clusters in the sample, which will allow us to build a scaling relation as a demonstration of XGA's capabilities:

[3]:
```
# This command generates spectra for all the clusters in our sample, in their R500␣
↪regions
srcs = evselect_spectrum(srcs, 'r500')
```

```
Generating products of type(s) spectrum: 100%|| 164/164 [2:56:28<00:00, 64.57s/it]
```

[4]:
```
# And this fits a single temperature APEC model to them
srcs = single_temp_apec(srcs, 'r500')
```

```
Running XSPEC Fits: 100%|| 40/40 [03:50<00:00,  5.76s/it]
```

3.5.4 A scaling relation from a ClusterSample

I've built convenience functions into the ClusterSample class, which make it very easy to generate certain types of common scaling relation. The only information that the method typically needs is the region for which spectra were generated and parameters measured, though plenty of other information (such as the model used to fit the data) can be supplied. The methods assume that, for a ClusterSample, we would have fit an absorbed APEC model, so that is the default.

Once the method call is complete, an XGA scaling relation product is returned to the user. Note that it is not stored inside the sample object, it goes directly back to the user.

The user is allowed to select the x and y normalisation values that should be used during the fit, and can easily change them from their standard values by passing them into keyword arguments. We have found that, in general, the best results are given when the normalisation is similar to the median value of the dataset.

[5]:
```
# Making an Lx-Tx relation
lx_tx_relation = srcs.Lx_Tx('r500')

# Making an Lx-richness relation
lx__relation = srcs.Lx_richness('r500')
```

```
/home/dt237/code/PycharmProjects/XGA/xga/samples/base.py:242: UserWarning: There are␣
↪no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:425: UserWarning: One of␣
↪the temperature uncertainty values for XCSSDSS-10401 is more than three times␣
↪larger than the other, this means the fit quality is suspect.
  warn("One of the temperature uncertainty values for {s} is more than three times␣
↪larger than "
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:436: UserWarning: There␣
↪are no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 2 sources␣
↪have NaN values and have been excluded
```

(continues on next page)

(continued from previous page)

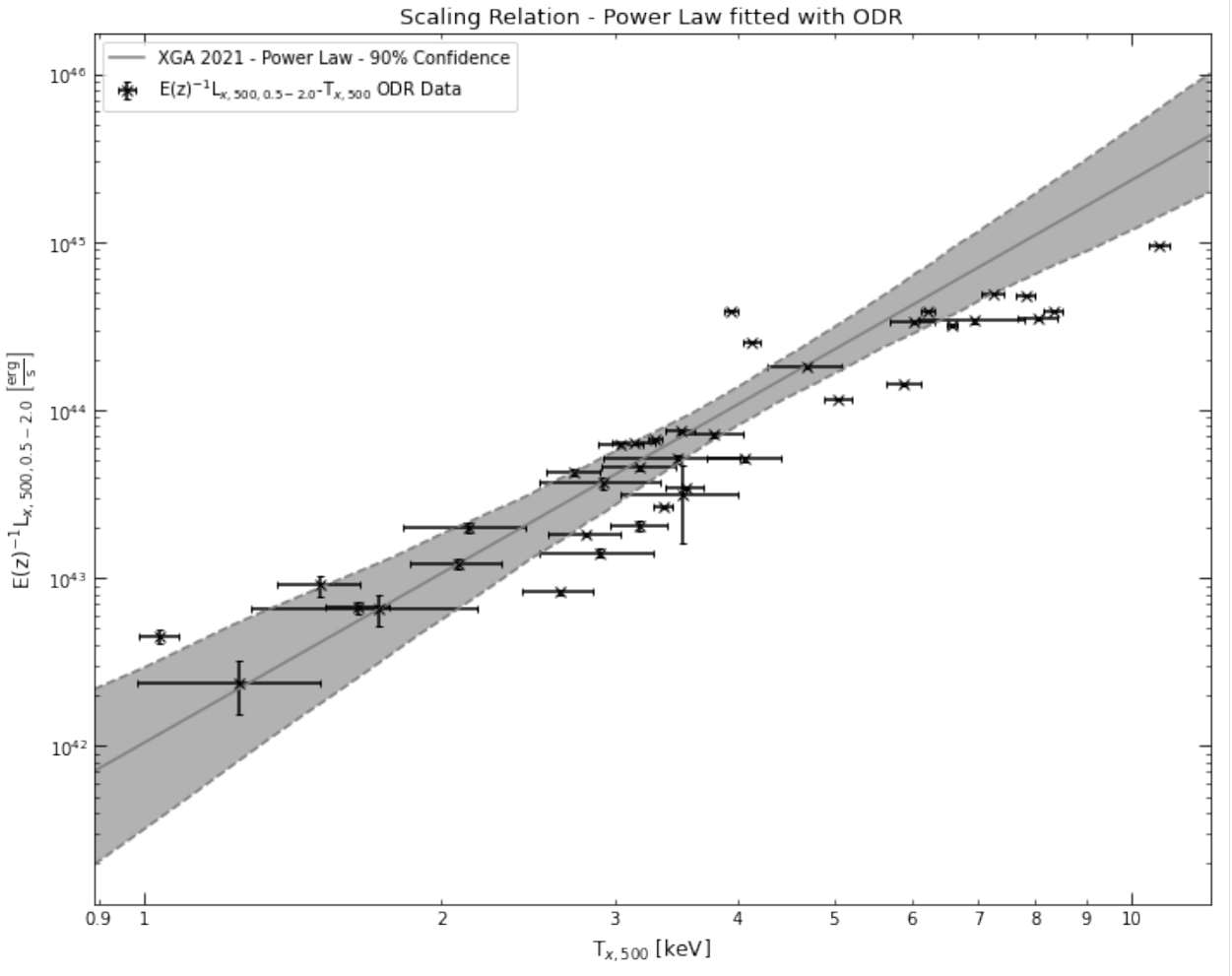
```
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 1 sources␣
↪have NaN values and have been excluded
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
<string>:12: RuntimeWarning: invalid value encountered in power
```

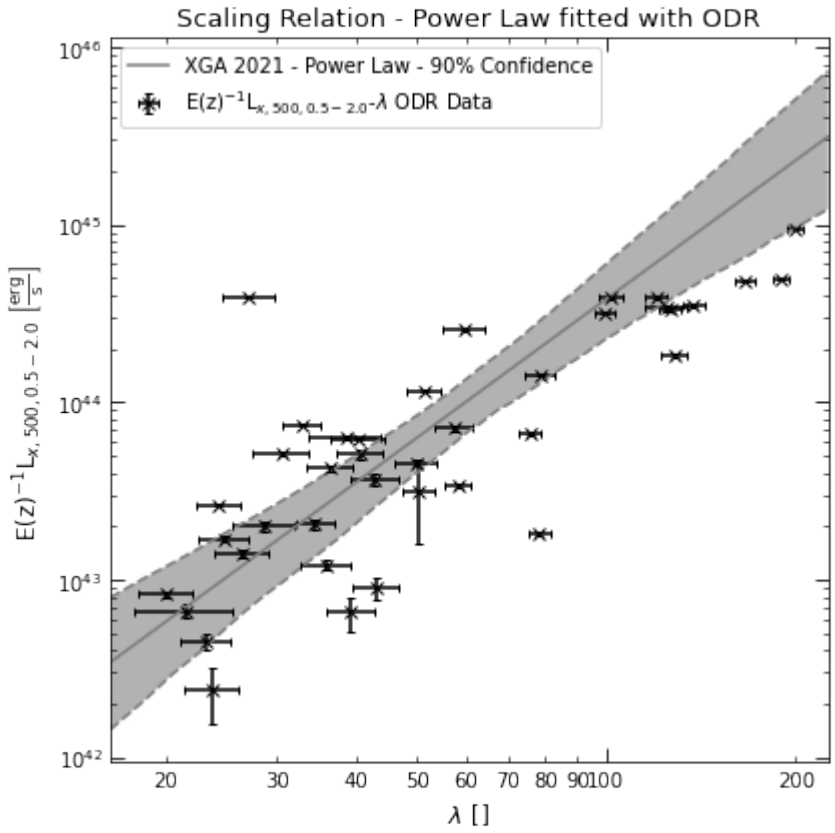
3.5.5 Viewing scaling relations

Just as with other XGA products, there is a `view()` method built into the class, to easily create high quality visualisations of the relation. As we generated these relations ourselves, they also have data point associated with them. That is not the case if the scaling relation comes from literature, whether it is one of the ones built into XGA, or one you have defined yourself.

There are several options that change what the output looks like, but one of the main options is the ability to change the figure size, which can be set using the `figsize` keyword argument:

[6]:
```
lx_tx_relation.view()

lx__relation.view(figsize=(6, 6))
```





3.5.6 Different fitting methods

I have implemented three different ways to fit these scaling relations, primarily to give the user some choice, though I would **highly recommend** that you use the implementation of the LIRA fitting package for R (Sereno (2015)) - LIRA is an extremely capable fitting package, and produces excellent results. The downside is that it requires the installation of some optional external dependencies.

The other methods available are scipy's simple non-linear least squares implementation (curve_fit), and scipy's orthogonal distance regression (ODR) implementation. ODR is the default method for ClusterSample scaling relation methods, as it does not require external dependencies, and seems to work quite well.

I'm going to use a ClusterSample Lx_Tx relation as an example, and show how you can change the fitting method when you call the function - all this does is call the different fitting functions in xga.relations.fit:

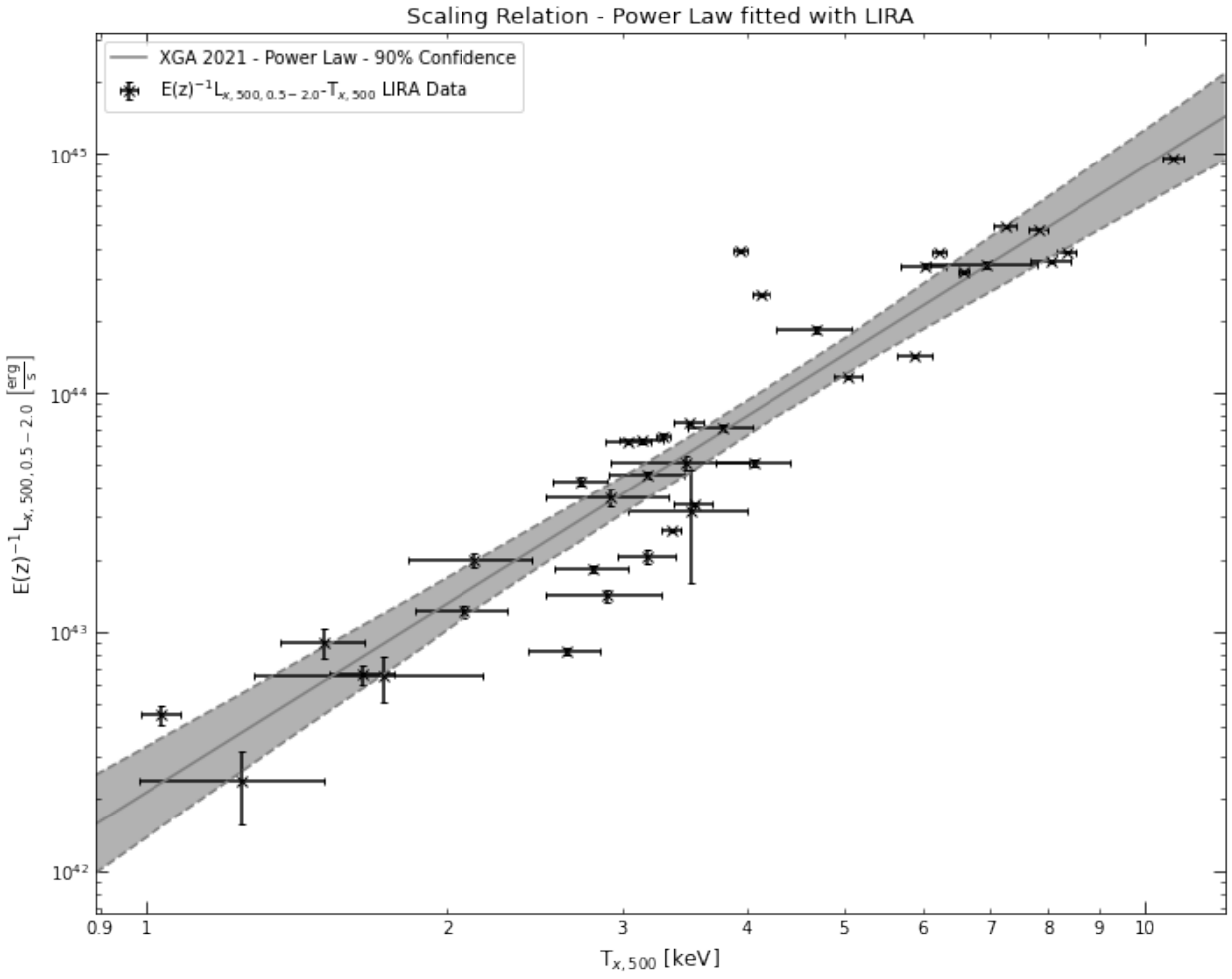
[7]:
```
# By setting the fit method to LIRA, we choose to use that instead of the default ODR
lx_tx_relation_lira = srcs.Lx_Tx('r500', fit_method='lira')
# And quickly viewing the relation
lx_tx_relation_lira.view()

# We repeat this process but with scipy's curve_fit
lx_tx_relation_curvefit = srcs.Lx_Tx('r500', fit_method='curve_fit')
lx_tx_relation_curvefit.view()
```

```
/home/dt237/code/PycharmProjects/XGA/xga/samples/base.py:242: UserWarning: There are␣
↪no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
```

(continues on next page)

(continued from previous page)

```
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:425: UserWarning: One of␣
↪the temperature uncertainty values for XCSSDSS-10401 is more than three times␣
↪larger than the other, this means the fit quality is suspect.
  warn("One of the temperature uncertainty values for {s} is more than three times␣
↪larger than "
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:436: UserWarning: There␣
↪are no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 2 sources␣
↪have NaN values and have been excluded
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
R[write to console]: module mix loaded

  |************************************************| 100%
```

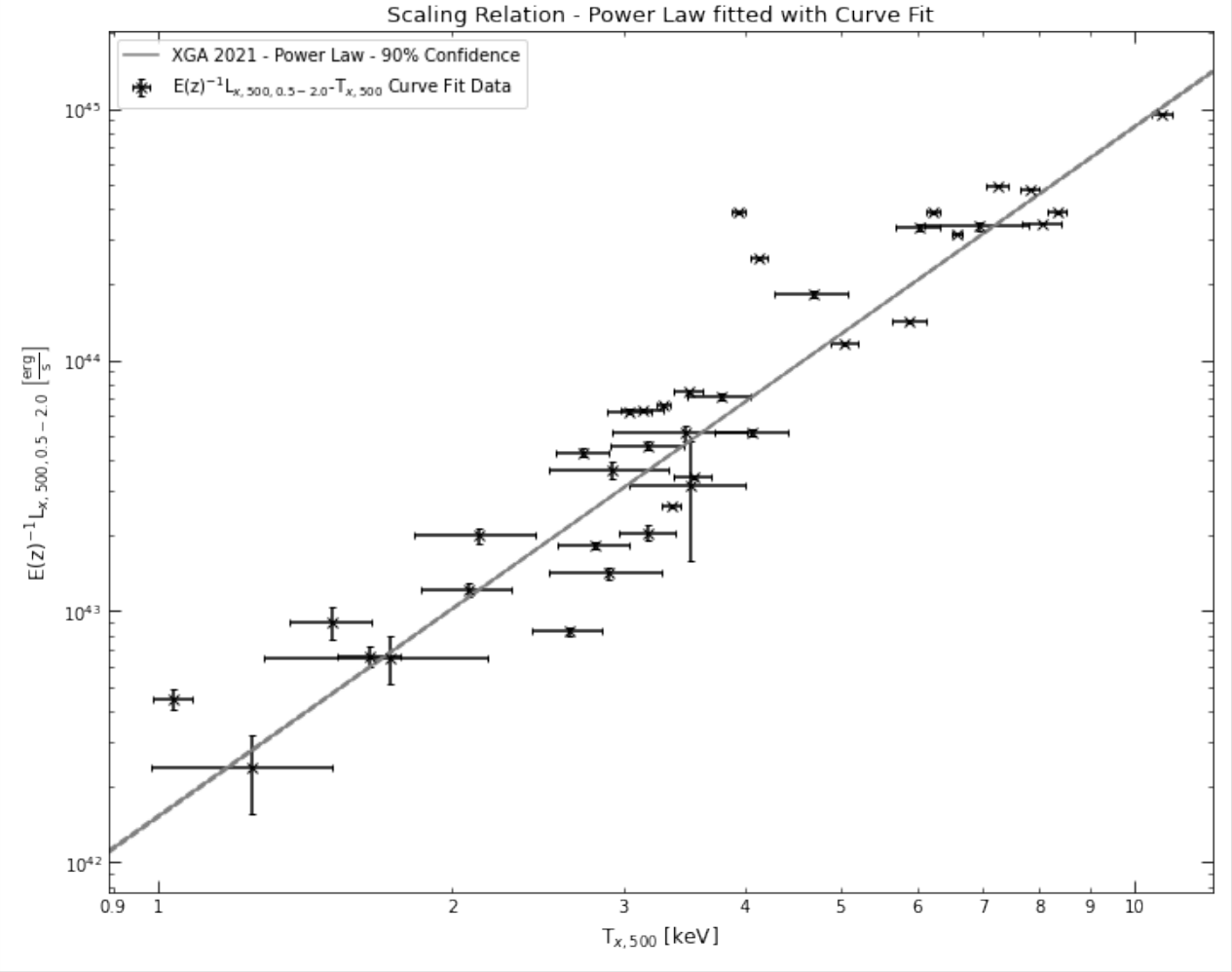



```
/home/dt237/code/PycharmProjects/XGA/xga/samples/base.py:242: UserWarning: There are␣
↪no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:425: UserWarning: One of␣
↪the temperature uncertainty values for XCSSDSS-10401 is more than three times␣
↪larger than the other, this means the fit quality is suspect.
  warn("One of the temperature uncertainty values for {s} is more than three times␣
↪larger than "
```

(continues on next page)

(continued from previous page)

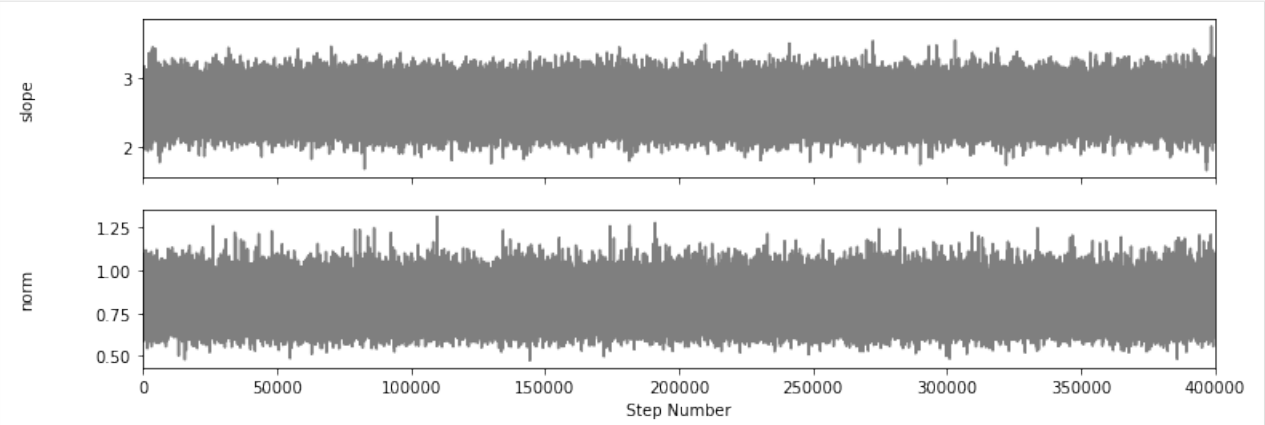
```
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:436: UserWarning: There␣
↪are no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 2 sources␣
↪have NaN values and have been excluded
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
```

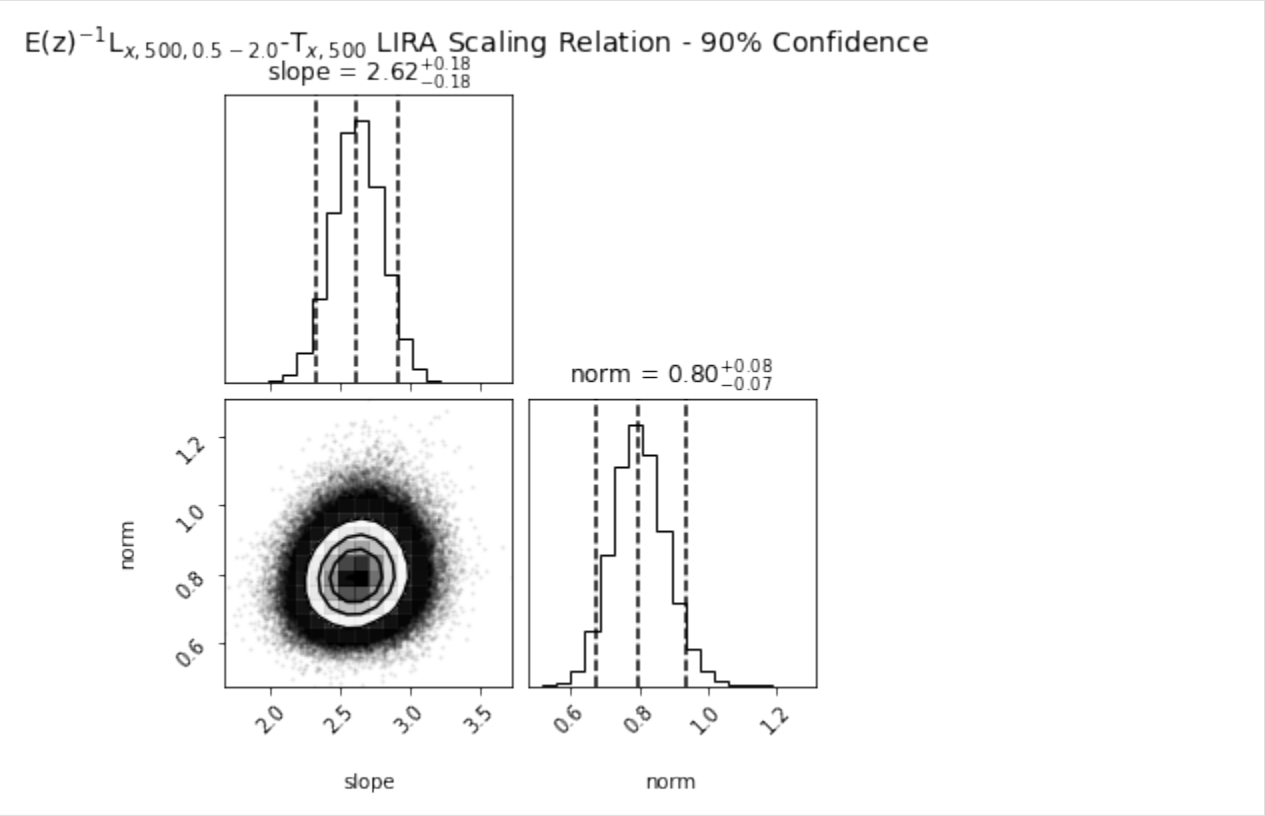


ScalingRelation objects created using the LIRA fitting method actually have some extra functionality compared to other ScalingRelations. As LIRA is based around the Just Another Gibbs Sampler (JAGS) package, it is an MCMC fitting process, and as such returns parameter chains which it can be useful to visualise.

The output chains are stored in the ScalingRelation on its declaration, and two methods were implemented to generated chain and corner plots:

[8]:
```
# Here we visualise the parameter chains
lx_tx_relation_lira.view_chains()

# And this shows us a useful corner plot of the parameter space
lx_tx_relation_lira.view_corner()
```





3.5.7 ScalingRelation product properties and making predictions

These product objects are quite simple compared to some of the other products in XGA, but still contain a lot of useful information. The information can be broadly split into two groups; administrative information concerning when the relation was generated, who by etc. (especially useful for relations from literature), and that information concerning the actual properties of the relation, units, the normalisations applied etc.

I will demonstrate a few of the properties using a classic relation from literature that has been built into XGA, the (Arnaud et al. (2005)) mass-temperature relation. As well as visualising the relation, we can also easily access the fit parameters that were used to define it, using the 'pars' method; the first column contains the values of all fit parameters, and the second column contains the uncertainty.

[9]:
```
# Demonstrating the basic adminstrative properties
print(MT.arnaud_m500.year)
print(MT.arnaud_m500.author)
print(MT.arnaud_m500.doi, '\n')

# Viewing the fit parameters that were used to define this scaling relation
print(MT.arnaud_m500.pars)
# We can also easily view the x and y units
print(MT.arnaud_m500.x_unit, MT.arnaud_m500.y_unit)
# And the normalisations
print(MT.arnaud_m500.x_norm, MT.arnaud_m500.y_norm)
```

```
2005
Arnaud et al.
10.1051/0004-6361:20052856

[[1.71 0.09]
 [3.84 0.14]]
keV solMass
5.0 keV 100000000000000.0 solMass
```

Once we have a scaling relation, we are also able to make predictions from it, using a built in method that takes an Astropy quantity (with the correct units for the x-axis) as an argument.

The one thing you should remember is that you may need to account for E(z), depending on the relation you have fitted. For instance, the L_x - T_x relations produced by XGA automatically multiply by the $E(z)^{-1}$ of each cluster before fitting occurs. As such any values predicted from that relation have an E(z) dependance that must be accounted for to get a physical value:

[10]:
```
# Defining an astropy quantity in units of keV, to demonstrate the predict␣
↪functionality
test_temp = Quantity(6, 'keV')

# Getting a prediction of the cluster mass from that test temperature
MT.arnaud_m500.predict(test_temp)
```

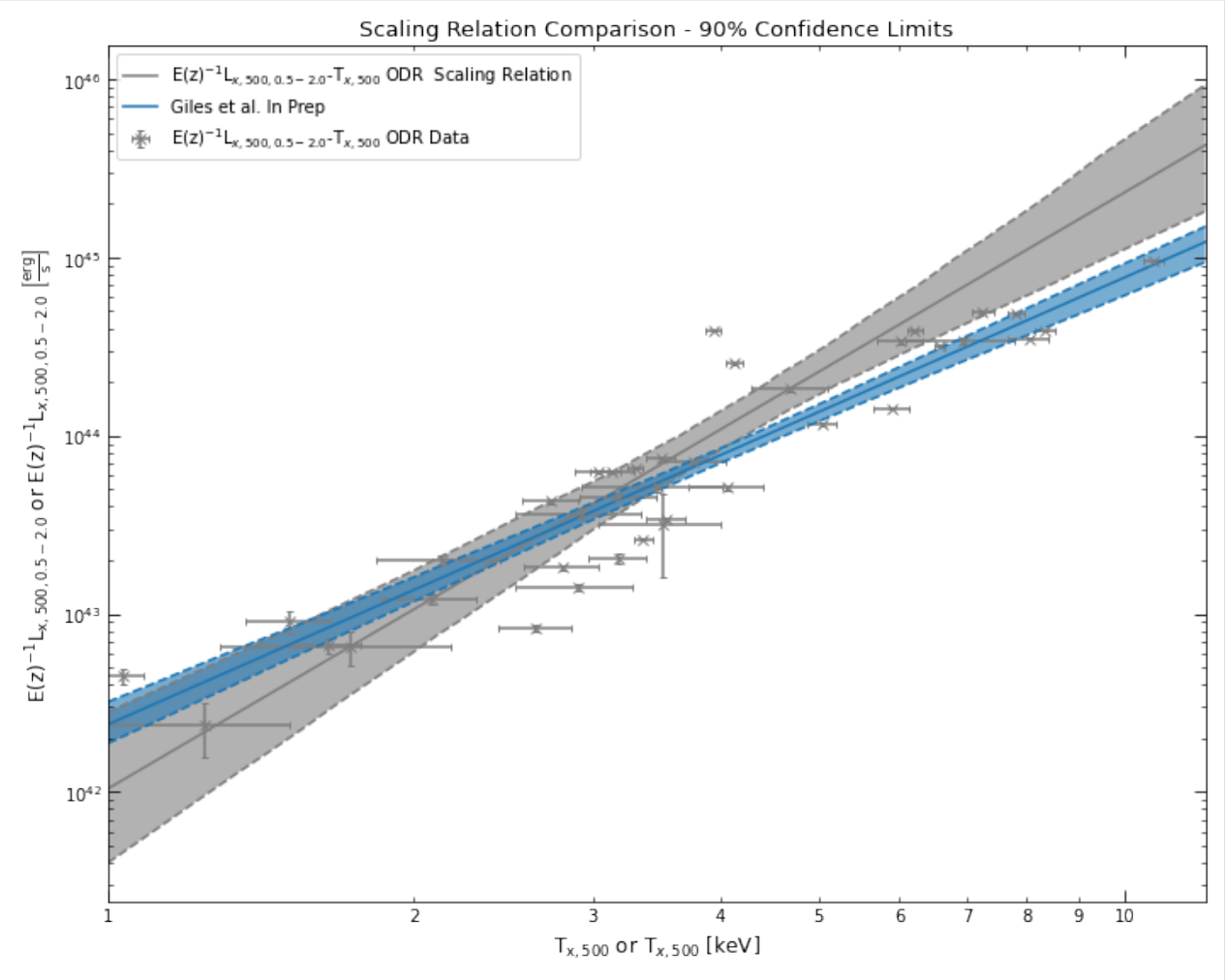
[10]: 5.2448268×10^{14} $M_{\odot}$

3.5.8 Viewing multiple scaling relations

As I mentioned briefly in the 'XGA Products' tutorial, ScalingRelation objects have a useful little trick that allows you to easily view different scaling relations on the same axis. As long as two ScalingRelation instances have the same x and y units, you can simply add them together using the Python addition operator. Here I demonstrate this with the L_x - T_x relation we measured earlier, and an XCS-SDSS L_x - T_x relation imported from XGA:

[11]:
```
# Adding them together is extremely easy
comb_lt_relation = lx_tx_relation + LT.xcs_sdss_r500_52

# Then we just call .view() like we would normally
comb_lt_relation.view()
```

```
/home/dt237/code/PycharmProjects/XGA/xga/products/relation.py:631: UserWarning: Not␣
↪all of these ScalingRelations have the same x-axis names.
  warn('Not all of these ScalingRelations have the same x-axis names.')
/home/dt237/code/PycharmProjects/XGA/xga/products/relation.py:638: UserWarning: Not␣
↪all of these ScalingRelations have the same y-axis names.
  warn('Not all of these ScalingRelations have the same y-axis names.')
```



3.5.9 Fitting scaling relations from external data

I tried to think of an interesting dataset that I could generate a scaling relation from as an example, but came up empty, if you can think of an interesting data to use as an example send me an email!

Instead of something interesting, I'm just going to extract some data from our ClusterSample object and show you how to generate a scaling relation manually, using functions imported from xga.relations.fit. We're going to measure a scaling relation that doesn't seem particularly useful, λ - T_x:

[12]:
```
# Grabbing the temperatures (and temperature errors) from the ClusterSample object
temps = srcs.Tx('r500')
# Printing them on screen
print(temps, '\n')

# And getting the richness values out as well (they only have a gaussian error)
richnesses = srcs.richness
print(richnesses)
```

```
[[ 2.13213     0.20519163  0.39433514]
 [ 2.80067     0.22927479  0.24261076]
 [10.6814      0.20506005  0.29234747]
```

(continues on next page)

(continued from previous page)

```
 [ 4.69087     0.39670878  0.41071715]
 [        nan         nan         nan]
 [ 6.21681     0.10400441  0.1041854 ]
 [ 4.06662     0.33423215  0.36426303]
 [ 8.04773     0.37034005  0.37111371]
 [ 6.01913     0.31320463  0.31522211]
 [ 3.51722     0.35075593  0.61433286]
 [ 8.34206     0.18331185  0.18349742]
 [ 1.03624     0.04787366  0.04563837]
 [ 3.35933     0.07213937  0.0746874 ]
 [ 2.92438     0.33646087  0.48980278]
 [ 4.1308      0.08558627  0.08634711]
 [ 6.94468     0.71801967  0.98840567]
 [ 2.7248      0.14950907  0.19408899]
 [ 5.89392     0.23684146  0.23756279]
 [ 1.24694     0.34040456  0.18172081]
 [ 1.5099      0.14780256  0.14409164]
 [ 7.24241     0.18409504  0.18429442]
 [ 3.49575     0.12190259  0.10648139]
 [ 3.13697     0.15849034  0.15939   ]
 [ 2.89694     0.36272877  0.39730179]
 [ 2.0819      0.17245655  0.27626099]
 [ 3.93636     0.0656564   0.06614467]
 [ 3.53631     0.14068098  0.17172321]
 [ 3.17636     0.20665492  0.20668628]
 [ 2.63633     0.15206357  0.28660748]
 [ 1.7306      0.25395097  0.6337762 ]
 [ 3.29938     0.05445985  0.05463768]
 [ 3.46955     0.43179068  0.66132311]
 [        nan         nan         nan]
 [ 3.77152     0.27875766  0.28276014]
 [ 3.18045     0.26438784  0.28187044]
 [ 3.03856     0.15850275  0.15979955]
 [ 6.57972     0.07462442  0.0746386 ]
 [ 7.81554     0.1637604   0.16379933]
 [ 5.04425     0.16337122  0.16430272]
 [ 1.65025     0.10175277  0.14958277]] keV

[[ 28.66831     3.2433918]
 [ 78.32325     3.12556  ]
 [199.53928     5.2998266]
 [128.19176     5.6970525]
 [ 24.793547    2.2973104]
 [120.09137     4.9689746]
 [ 30.468397    3.0966108]
 [137.75664     5.25023  ]
 [126.30015     5.016507 ]
 [ 50.421906    2.964098 ]
 [101.70913     4.746494 ]
 [ 23.14601     2.1740084]
 [ 24.244368    1.9738064]
 [ 42.87069     3.7267637]
 [ 59.63591     4.6716833]
 [124.6221      9.576292 ]
 [ 36.52462     3.0883944]
 [ 78.58259     4.482442 ]
 [ 23.679934    2.2644103]
```

(continues on next page)

(continued from previous page)

```
 [ 43.11477     3.6073697]
 [189.18115     5.613359 ]
 [ 32.863426    2.2285726]
 [ 38.690826    5.167868 ]
 [ 26.395319    2.646696 ]
 [ 35.91442     3.2383535]
 [ 27.081793    2.5084076]
 [ 58.125664    2.954672 ]
 [ 34.480453    2.3746781]
 [ 20.04925     1.9597924]
 [ 39.270515    3.4333043]
 [ 75.729904    2.898407 ]
 [ 40.68923     3.2910614]
 [ 87.781334    4.778909 ]
 [ 57.317154    3.9977226]
 [ 49.918488    3.82978  ]
 [ 40.338364    4.0101438]
 [ 99.80225     3.735475 ]
 [166.20715     6.2375617]
 [ 51.352455    3.1802998]
 [ 21.559492    3.837598 ]]
```

```
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:425: UserWarning: One of␣
↪the temperature uncertainty values for XCSSDSS-10401 is more than three times␣
↪larger than the other, this means the fit quality is suspect.
  warn("One of the temperature uncertainty values for {s} is more than three times␣
↪larger than "
/home/dt237/code/PycharmProjects/XGA/xga/samples/extended.py:436: UserWarning: There␣
↪are no XSPEC fits associated with XCSSDSS-455
  warn(str(err))
```

The scaling relation generation functions expect the data and uncertainties to be passed in separately, so here we're just going to split the data I just retrieved from my ClusterSample object.

The scaling relation generating methods support data having either two uncertainties (in which case it will understand them as - and + errors), or just the one set of uncertainties. Part of the data preperation step inside of the scaling relation involves calculating an average uncertainty for a value if it is passed in with two uncertainties. This average uncertainty is what is used to fit the data.

[13]:
```
# The first column contains the actual temperature values
x_data = temps[:, 0]
# The second and third columns are the uncertainties
x_errs = temps[:, 1:]

# Again the first column contains the richness values
y_data = richnesses[:, 0]
# However this data only has one uncertainty (rather than - and + uncertainties)
y_errs = richnesses[:, 1]
```

Please make sure you are supplying the x and y data in the correct order, this function needs you to pass the y-data **first**.

Also, when you fit a scaling relation with LIRA, it automatically uses a power-law in logspace as the model to fit with. This is not true of the other fitting methods, and you will need to supply a model for the fit to use; I suggest importing power_law from xga.models.misc and using that.

There are another couple of differences between the LIRA function and other fitting methods, ODR and Curve Fit both allow you to set start parameters (though you don't have to), and the LIRA fitting function allows you to tell it

how many chains should be run, and with how many steps.

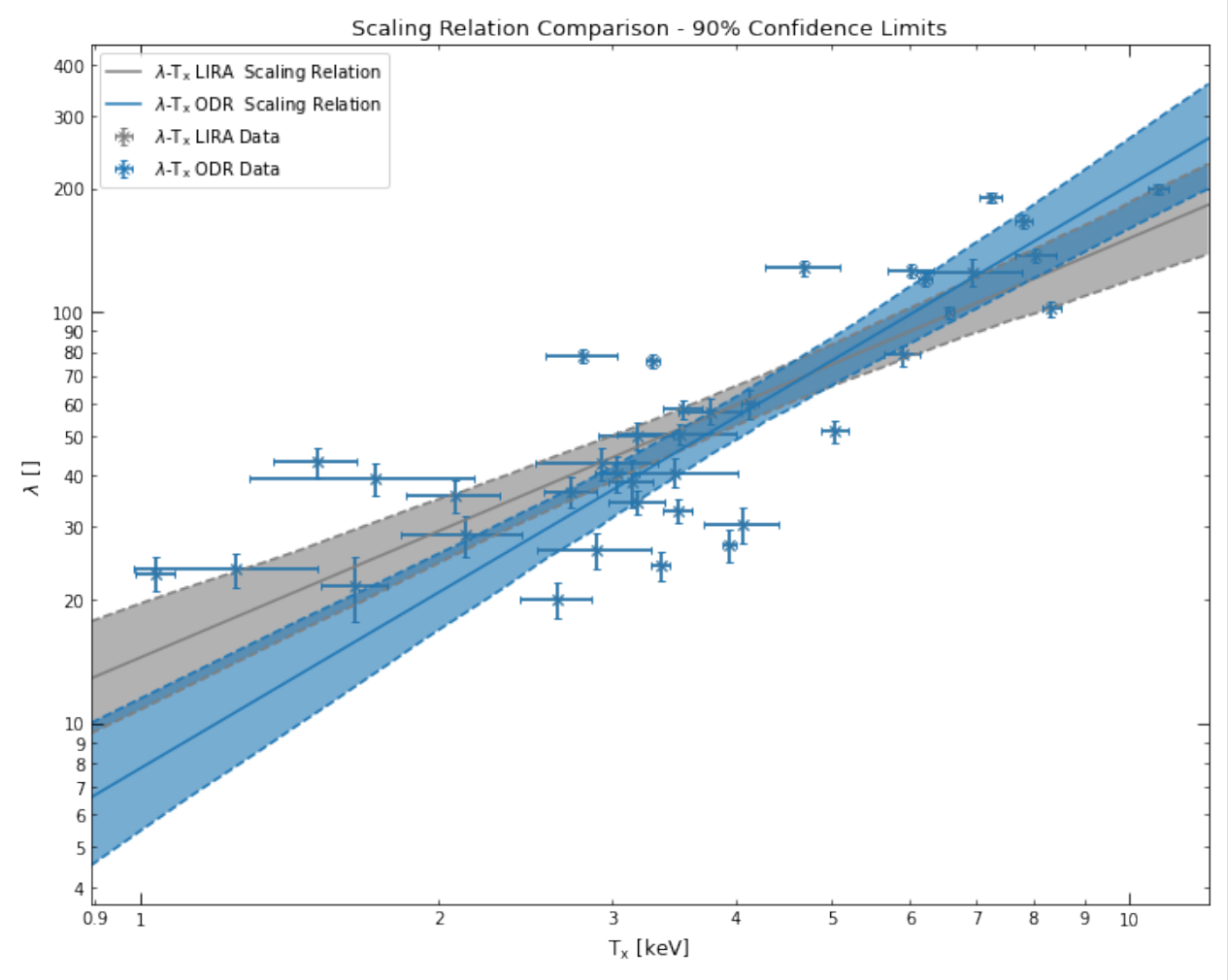
When deciding on what to name your x and y axis, please remember not to put units in the name, XGA will add them itself. Also bear in mind that these names will be passed into matplotlib, so some LaTeX syntax will work:

[14]:
```
# The x values are temperatures, so selecting an arbitrary normalisation of 4keV
x_norm = Quantity(4, 'keV')
# The y values are richness, which has no unit, but still needs to be in an Astropy␣
↪quantity
y_norm = Quantity(60, '')

# This uses a LIRA fit to generate the scaling relation
richness_tx_lira = scaling_relation_lira(y_data, y_errs, x_data, x_errs, y_norm, x_
↪norm,
                                         x_name=r'T$_{\rm{x}}$', y_name=r'$\lambda$')

# This uses ODR to generate a scaling relation, note that I have passed in a model␣
↪function
richness_tx_odr = scaling_relation_odr(power_law, y_data, y_errs, x_data, x_errs, y_
↪norm, x_norm,
                                         x_name=r'T$_{\rm{x}}$', y_name=r'$\lambda$')
```

```
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 2 sources␣
↪have NaN values and have been excluded
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
```

```
  |************************************************| 100%
```

[15]:
```
(richness_tx_lira+richness_tx_odr).view()
```

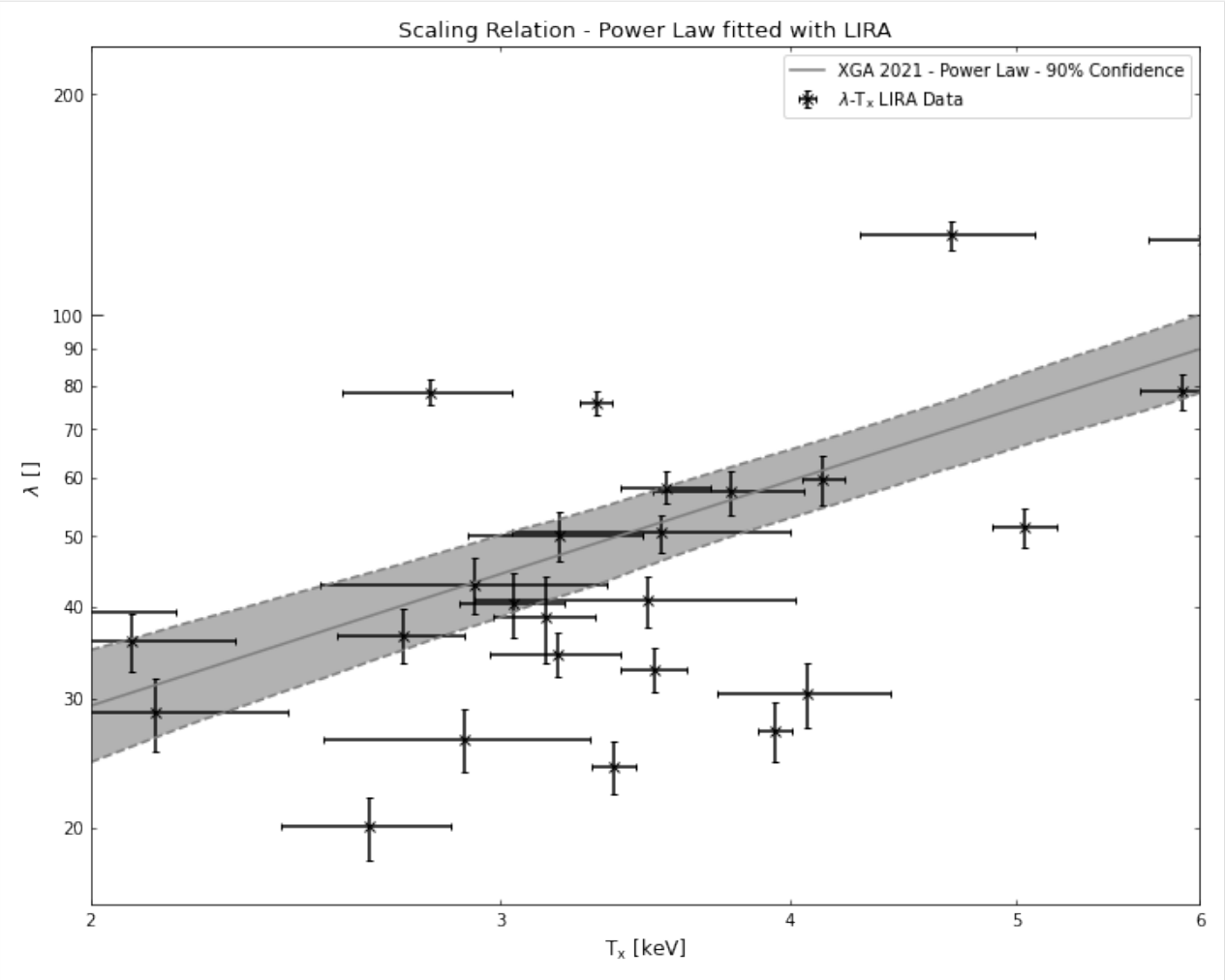


You can also set a range in which you wish a scaling relation to be valid, when you generate it using one of these fitting functions. For instance I generate the same scaling relation we just created, but decide that it is only valid in the temperature range 2-6keV:

[16]:
```
richness_tx_lira_lim = scaling_relation_lira(y_data, y_errs, x_data, x_errs, y_norm,␣
↪x_norm,
                                             x_name=r'T$_{\rm{x}}$', y_name=r'$\lambda
↪$',
                                             x_lims=Quantity([2, 6], 'keV'))
richness_tx_lira_lim.view()
```

```
/home/dt237/code/PycharmProjects/XGA/xga/relations/fit.py:71: UserWarning: 2 sources␣
↪have NaN values and have been excluded
  warn("{} sources have NaN values and have been excluded".format(thrown_away))
```

```
  |**************************************************| 100%
```



3.5.10 Defining scaling relations from literature

Here I will demonstrate how you can find a relation in literature, and then implement it in a form usable by XGA. This particular scaling relation is actually already implemented in XGA, and is the Arnaud mass-temperature relation that I used in the demonstrations earlier.

It really is as simple as filling out as many of the ScalingRelation arguments as you need. You should note that 'power_law' in this case is defined in the XGA models package, and is a function used to generate the fit line that is plotted. ScalingRelation will take its x and y units from the normalisations that are passed in on initialisation (here 5keV for the x axis, and 1e+14M$_\odot$ for the y-axis).

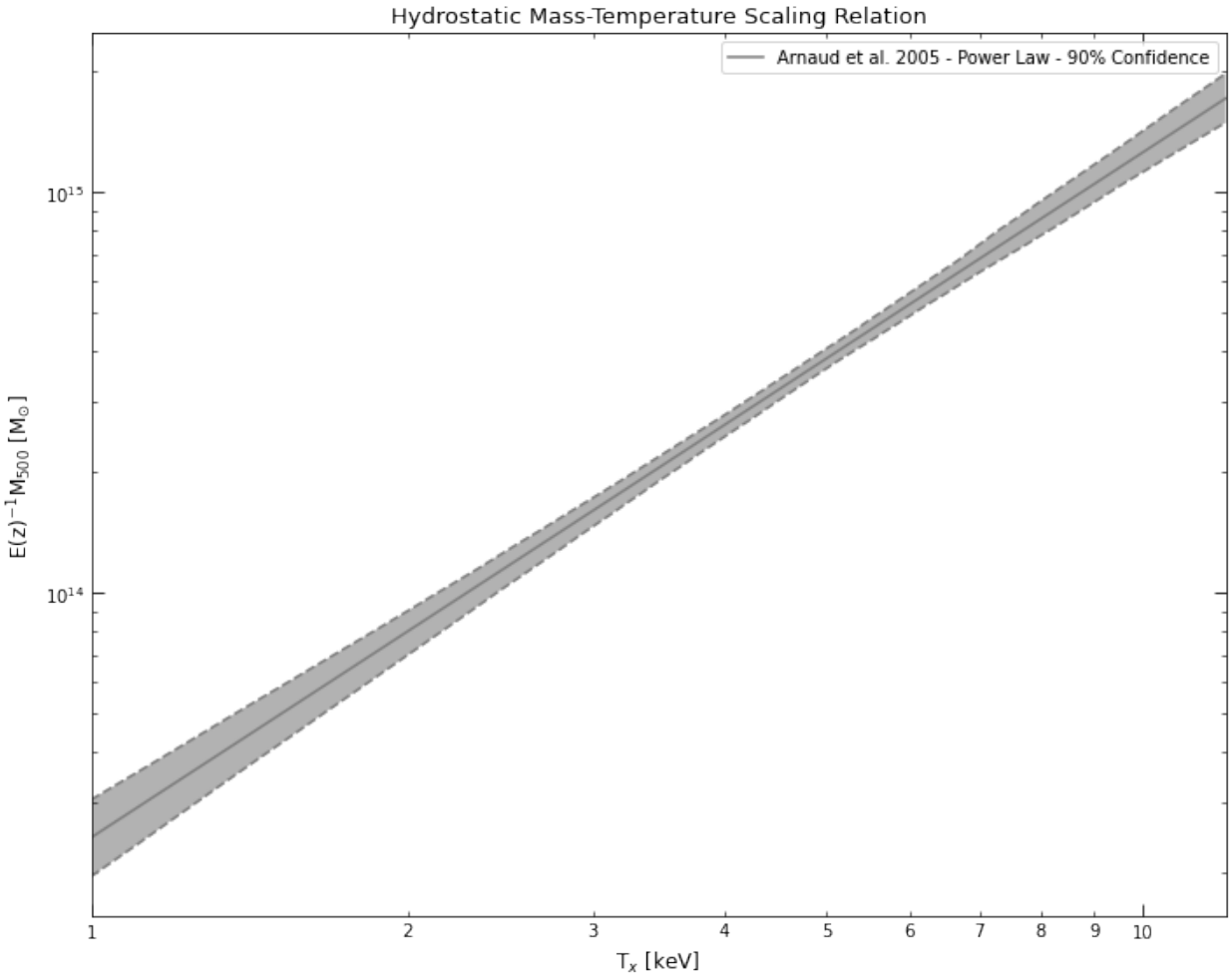
Please don't include units in the values passed for `x_name` and `y_name`, the `view()` method will add its own units, taken from the astropy normalisation quantities:

[17]:
```
mt_relation = ScalingRelation(np.array([1.71, 3.84]), np.array([0.09, 0.14]), power_
↪law, Quantity(5, 'keV'),
                              Quantity(1e+14, 'solMass'), r"T$_{x}$", "E(z)$^{-1}$M$_
↪{500}$",
                              relation_author='Arnaud et al.', relation_year='2005',
                              relation_doi='10.1051/0004-6361:20052856',
```

(continues on next page)

(continued from previous page)

```
                                relation_name='Hydrostatic Mass-Temperature', x_
↪lims=Quantity([1, 12], 'keV'))
mt_relation.view()
```



CHAPTER

FOUR

ADVANCED TUTORIALS

4.1 Basics of XGA profiles - focusing on cluster surface brightness

Here the aim is to go through the basic capabilities of XGA profile products, with a focus on the generation of surface brightness profiles of galaxy clusters. A demonstration of the fitting functionality built into XGA profiles will be given, which will include an exploration of the XGA model classes and their purpose. I'll also show how we can view profiles (both for individually and together), and run through the user-configurable options built into the view method.

Given the nature of this analysis, it isn't really applicable to point-like sources.

[1]:
```
from xga.sources import GalaxyCluster

from astropy.units import Quantity
import numpy as np
```

Firstly, I will define a GalaxyCluster source for Abell 907, a cluster for which I know high quality data is available. **Again, please note that the overdensity radius and redshift that I've used here are approximate and should not be used for a scientific analysis**:

[2]:
```
src = GalaxyCluster(149.59209, -11.05972, 0.16, r500=Quantity(1200, 'kpc'), name='A907
↪')
src.info()
```

```
------------------------------------------------------
Source Name - A907
User Coordinates - (149.59209, -11.05972) degrees
X-ray Peak - (149.59251340970866, -11.063958320861634) degrees
nH - 0.0534 1e+22 / cm2
Redshift - 0.16
XMM ObsIDs - 3
PN Observations - 3
MOS1 Observations - 3
MOS2 Observations - 3
On-Axis - 3
With regions - 3
Total regions - 69
Obs with one match - 3
Obs with >1 matches - 0
Images associated - 18
Exposure maps associated - 18
Combined Ratemaps associated - 1
Spectra associated - 0
R500 - 1200.0 kpc
```

(continues on next page)

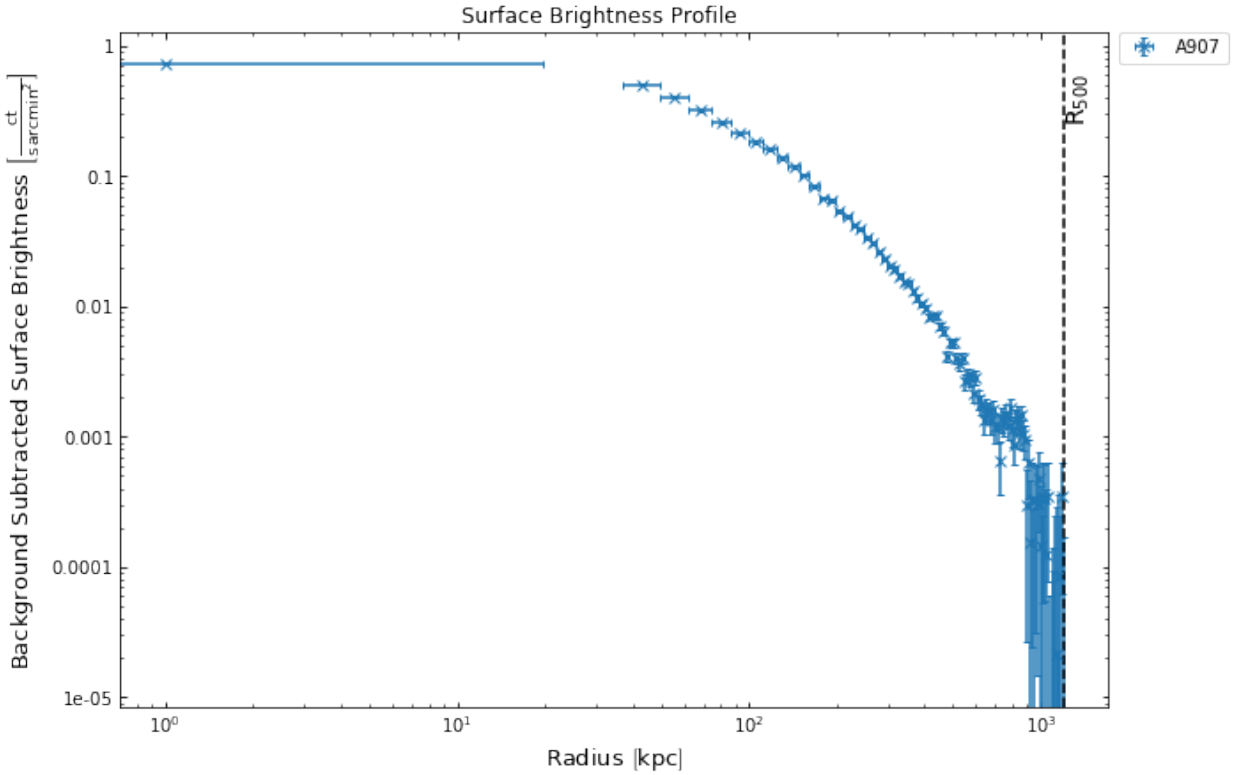
(continued from previous page)

```
R500 SNR - 251.61
----------------------------------------------------
```

4.1.1 Generating and viewing a surface brightness profile

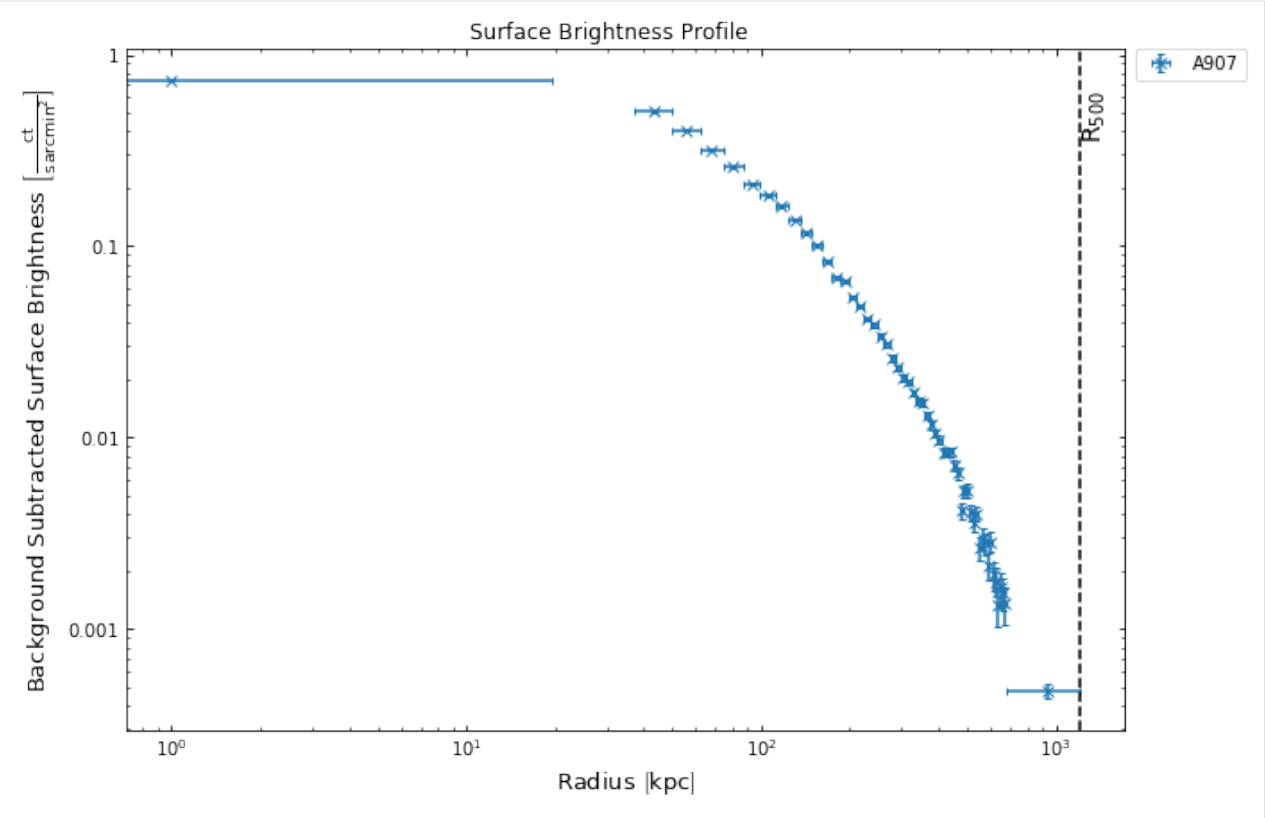
A convenient method for quickly generating and viewing cluster surface brightness profiles has been implemented in the GalaxyCluster source class, with the main input being the desired outer radius of the brightness profile in question. In this demonstration I have made a profile out to the approximate R_{500} that I supplied on declaration, and by default this method has used the combined brightness profile to do so:

[3]:
```
src.view_brightness_profile('r500')
```



The `view_brightness_profile` method doesn't actually create the brightness profile itself however, it is simply a wrapper around the radial_brightness function implemented in the 'imagetools' section of XGA. I will not go into detail on how the generation is undertaken, but the end result is an instance of the XGA SurfaceBrightness1D class. I will mention, however, that you can supply minimum quality requirements when generating these profiles, for instance requiring each bin to have a minimum signal to noise, as shown here:

[4]:
```
src.view_brightness_profile('r500', min_snr=1.1)
```



4.1.2 Retrieving the SurfaceBrightness1D instance from the source

It is likely that you want to do more than just look at the profile, so the next step is to extract it from wherever it has been stored in the source object. Most profile types have a specific get method, but a general `get_profiles` method also exists, which I shall show first. We have to supply the profile type, with a 'combined_' prefix to indicate that the profile was generated using combined data:

[5]:
```
src.get_profiles('combined_brightness')
```

[5]:
```
[<xga.products.profile.SurfaceBrightness1D at 0x7fa2fdbb7520>,
 <xga.products.profile.SurfaceBrightness1D at 0x7fa32c685b50>]
```

We can see that because we only supplied the type of profile we want, both the brightness profiles we generated have been returned to us. If we specifically wanted the profile we generated out to R_{500} without any automatic re-binning however, we can use the `get_1d_brightness_profile` method and tell it that the outer radius of the profile we want is R_{500} (something that, after the re-binning applied to the second profile, is no longer true for both):

[6]:
```
sb_prof = src.get_1d_brightness_profile('r500')
sb_prof
```

[6]:
```
<xga.products.profile.SurfaceBrightness1D at 0x7fa2fdbb7520>
```

4.1.3 Saving XGA profiles to disk

All profiles generated by XGA are automatically saved to the {xga save path}/profiles/{source name} directory, so that if the same source is declared again at some point in the future then they can be loaded back in without having to run the generation again. Running the generation procedure again for brightness profiles wouldn't be much of a problem, considering how fast it is, but other profiles take considerably more time to generate.

A profile will generally automatically re-save itself when a change (such as fitting a model) is made to it, and you can manually save it by calling the `save()` method and passing a path.

The profile is saved by pickling the profile instance, which can result in some relatively large files if there are many points and model fits stored within the profile:

[7]:
```
sb_prof.save("random_brightness_profile.xga")
```

If you have a profile object loaded in then you can access the automatic save path which XGA has assigned it by using the `save_path` property:

[8]:
```
sb_prof.save_path
```

[8]:
```
'/home/dt237/code/PycharmProjects/XGA/docs/source/notebooks/advanced_tutorials/xga_
↪output/profiles/A907/brightness_profile_A907_58634018.xga'
```

4.1.4 Fitting a model to the SurfaceBrightness1D instance

So, we've generated this profile, and now we've decided we want to be able to represent it with a model. You, as a new user, don't know a priori what models have been implemented for whatever profile type you're working with; as such you should call the `allowed_models()` method, which will tell you what models you're allowed to fit:

[9]:
```
# Passing grid here changes the style of the table. The default style looks nicer in␣
↪my opinion, but LaTeX
#  has a hard time rendering it for the PDF version of this documentation
sb_prof.allowed_models('grid')
```

```
+--------------+-----------------------------------------------+------------------------
↪-----------------+
| MODEL NAME   | EXPECTED PARAMETERS                           | DEFAULT START VALUES  ␣
↪                 |
+==============+===============================================+========================
| beta         | beta, r_core, norm                            | 1.0, 100.0 kpc, 1.0 ct␣
↪/ (arcmin2 s)    |
+--------------+-----------------------------------------------+------------------------
↪-----------------+
| double_beta  | beta_one, r_core_one, norm_one, beta_two,     | 1.0, 100.0 kpc, 1.0 ct␣
↪/ (arcmin2 s),   |
|              | r_core_two, norm_two                          | 0.5, 400.0 kpc, 0.5 ct␣
↪/ (arcmin2 s)    |
+--------------+-----------------------------------------------+------------------------
↪-----------------+
```

Now that you can see which models you're allowed to fit to this particular type of profile, we might decide that we're going to fit the simple beta model. There are a couple of slightly different ways we could decide to do this, with the simplest being we just pass the name of the model as a string to the `fit` method of the profile. This would declare an instance of that model class, with the default starting values and priors, and then fit it.

Alternatively, we could decide that we want to customise the model parameters, declare an instance of that particular model manually, and pass it to the fitting function. I'm going to demonstrate this approach, so I can introduce you to the way models work in XGA; firstly, we have to import the model that we chose:

[10]:
```
# I do not condone importing anywhere but the top of a Jupyter notebook, but it
#  helps the demonstration to do it here
from xga.models import BetaProfile1D
```

When we called the `allowed_models` method we were shown the default start values for the model parameters, including the required units, and so we can declare suitable astropy quantities now as we declare a new model instance with non-standard start values (the model will error if start parameters with the wrong units are passed):

[11]:
```
beta_inst = BetaProfile1D(cust_start_pars=[Quantity(0.8), Quantity(50, 'kpc'),␣
↪Quantity(0.6, 'ct/(arcmin^2 s)')])
beta_inst.start_pars
```

[11]:
```
[<Quantity 0.8>, <Quantity 50. kpc>, <Quantity 0.6 ct / (arcmin2 s)>]
```

I am not claiming that these are 'better' start values than the default ones for this particular cluster, I am simply doing this for demonstrative purposes. Model classes have many built in methods, some of which provide visualisations of the model, some provide convenient access to derivatives and integrated values, and some just provide general information about the model:

[12]:
```
beta_inst.info('grid')
```

```
+-----------------+-------------------------------------------------------------------------
↪----------+
| Beta Profile    |                                                                         ␣
↪          |
+=================+=========================================================================================+
| DESCRIBES       | Surface Brightness                                                      ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| UNIT            | ct / (arcmin2 s)                                                        ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| PARAMETERS      | beta, r_core, norm                                                      ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| PARAMETER UNITS | , kpc, ct / (arcmin2 s)                                                 ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| AUTHOR          | placeholder                                                             ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| YEAR            | placeholder                                                             ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| PAPER           | placeholder                                                             ␣
↪          |
+-----------------+-------------------------------------------------------------------------
↪----------+
| INFO            | Essentially a projected isothermal king profile, it can be              ␣
↪          |
|                 | used to describe a simple galaxy cluster radial surface␣
↪brightness profile. |
```

(continues on next page)

(continued from previous page)

```
+-----------------+-------------------------------------------------------------------
↪----------+
```

We can also see and alter the default priors that have been set for the model parameters. Though this is largely only relevant if you wish to fit with the MCMC option, and at the moment is limited to uniform priors. The `par_priors` property returns the priors in the form of a list of dictionaries:

[13]:
```
beta_inst.par_priors
```

[13]:
```
[{'prior': <Quantity [0., 3.]>, 'type': 'uniform'},
 {'prior': <Quantity [   0., 2000.] kpc>, 'type': 'uniform'},
 {'prior': <Quantity [0., 3.] ct / (arcmin2 s)>, 'type': 'uniform'}]
```

The `par_priors` property is also how you set new priors, by passing another list of dictionaries in the same format (an error will be thrown if there aren't the correct number of priors, or if the units are incorrect):

[14]:
```
beta_inst.par_priors = [{'prior': Quantity([-1., 4.]), 'type': 'uniform'},
                        {'prior': Quantity([   1., 1400.], 'kpc'), 'type': 'uniform'},
                        {'prior': Quantity([0., 2.], "ct / (arcmin2 s)"), 'type':
↪'uniform'}]
beta_inst.par_priors
```

[14]:
```
[{'prior': <Quantity [-1.,  4.]>, 'type': 'uniform'},
 {'prior': <Quantity [1.0e+00, 1.4e+03] kpc>, 'type': 'uniform'},
 {'prior': <Quantity [0., 2.] ct / (arcmin2 s)>, 'type': 'uniform'}]
```

Now that we've setup our model instance, we're going to fit it to the profile. There are several fitting methods implemented in the profile base class, including one based on the `emcee` ensemble MCMC sampler, one based around `scipy`'s 'curve_fit' implementation of non-linear least squares, and one based around `scipy`'s implementation of orthogonal distance regression. We shall use the MCMC fitter here, with a simple gaussian likelihood, as I wish to demonstrate some of the extra features that are enabled with an MCMC fit. **I will mention again that you could just pass 'beta' as the first argument here if you just wished to use the default beta model setup**:

[15]:
```
sb_prof.fit(beta_inst, num_steps=30000, num_walkers=20)
```

```
100%|| 30000/30000 [00:33<00:00, 905.55it/s]
```

[15]:
```
<xga.models.sb.BetaProfile1D at 0x7fa2fdce7ca0>
```

The fit method will return a model instance (in this case the same one we passed in), which has had the fit results added to it, though you can also retrieve the fit directly from the profile object (as the model is stored internally). We have to specify the name of the model as well as the fit type, as the same type of model can be fitted and stored with multiple fit methods:

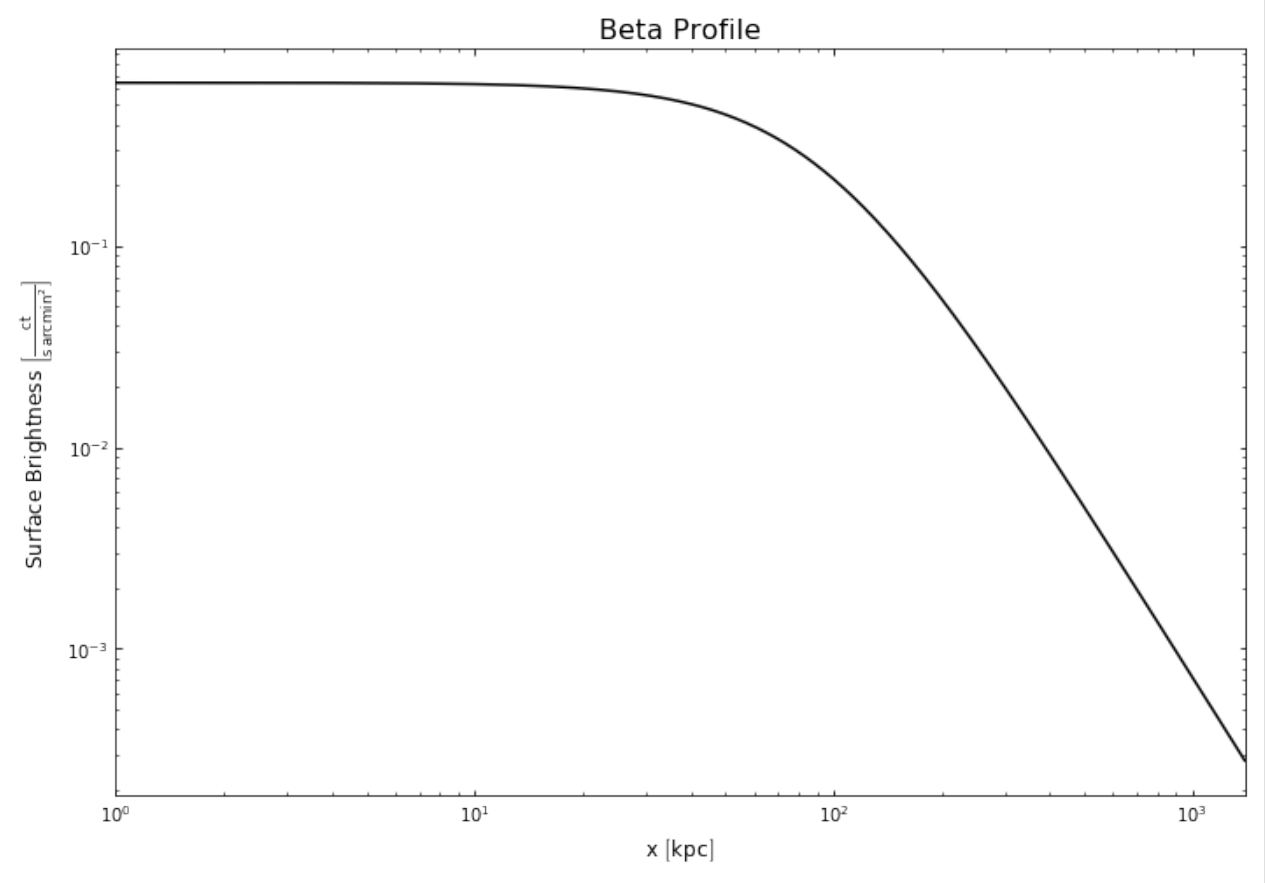
[16]:
```
sb_prof.get_model_fit('beta', 'mcmc')
```

[16]:
```
<xga.models.sb.BetaProfile1D at 0x7fa2fdce7ca0>
```

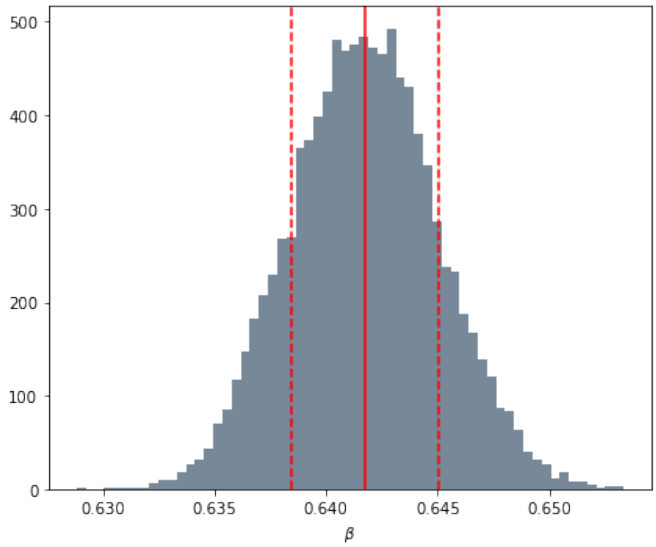
4.1.5 Exploring the fitted model

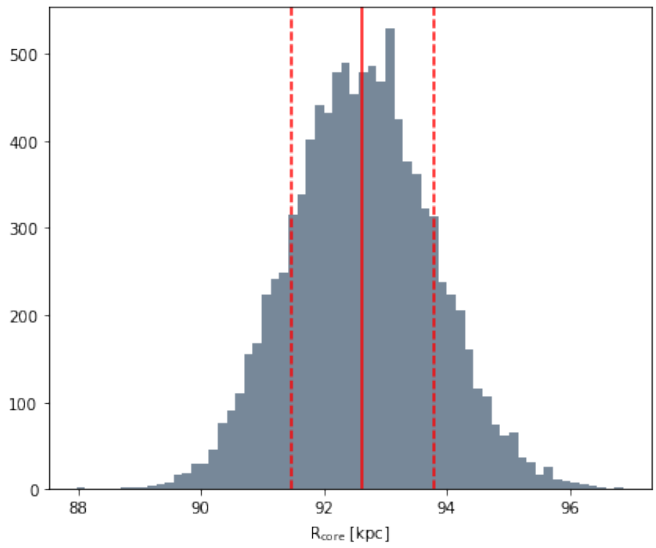
Due to the fact that we setup our own model instance prior to running the fit method (and the way Python memory addressing works), we can actually just look at our initial `beta_inst` model, we don't need to retrieve it from the object or the fit method (notice we never assigned their return to a variable). The model class has three visualisation methods implemented, though they are meant more for convenience than for the production of publishable figures. We can view the model curve (if we pass radii at which we wish to calculate values), and the parameter distributions from the fit:

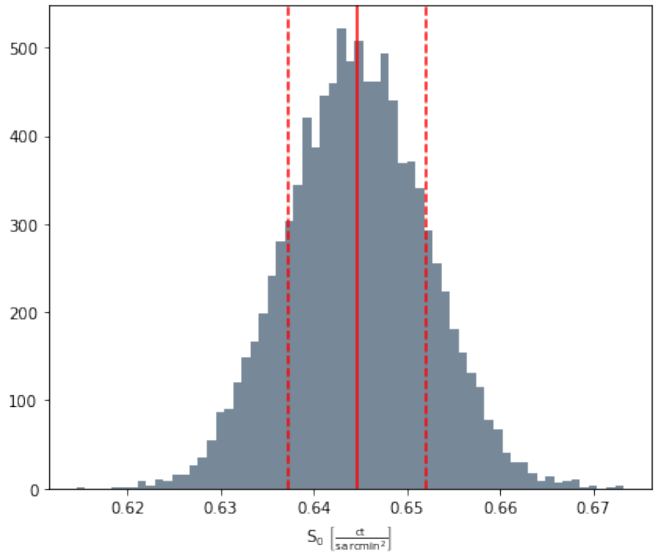
[17]:
```
# We have to pass radii at which to evaluate the model, I just pass a set of radii
#  from 1 (because its a log scale by default) to 1400kpc.
beta_inst.view(Quantity(np.arange(1, 1400, 1), 'kpc'), figsize=(10,7))
```



[18]:
```
beta_inst.par_dist_view()
```


500
400
300
200
100
0
0.630
0.635
0.640
0.645
0.650
β


500
400
300
200
100
0
88
90
92
94
96
R_core [kpc]


500
400
300
200
100
0
0.62
0.63
0.64
0.65
0.66
0.67
S0 [ct / sarcmin²]

4.1.6 Mathematics with the fitted model

There are various ways of interacting with the model mathematically, the simplest of which is using the fitted model to predict a value at a particular radius. For this you just need to call the model instance and pass the radius at which you wish to evaluate the model, no further information is required as the knowledge of the model parameter values is stored within the instance:

[19]:
```
beta_inst(Quantity(600, 'kpc'))
```

[19]: $[0.0030329884] \frac{ct}{s\,arcmin^2}$

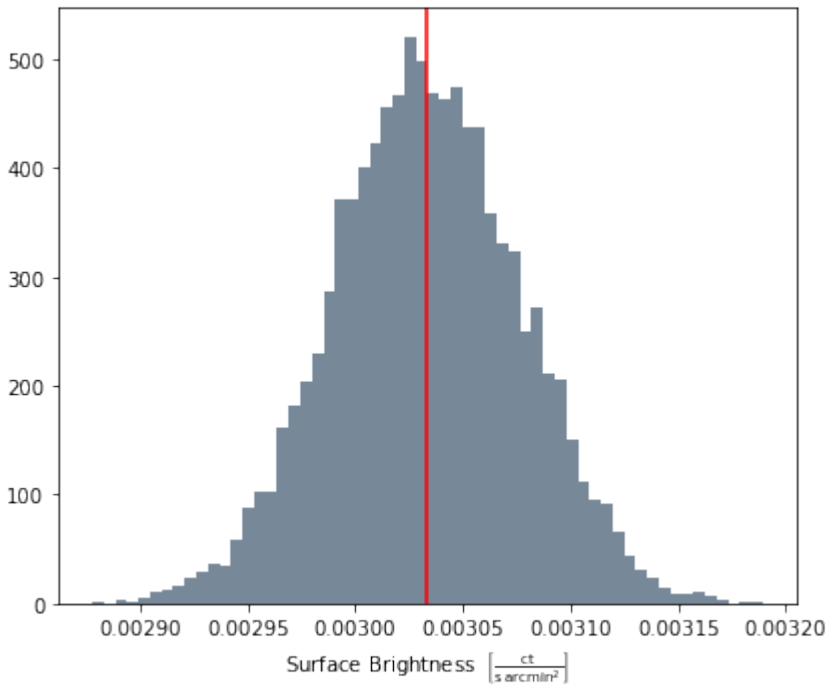
By default this returns a single value, using the best fit values to predict it; however more often than not a single value is not very useful, we need knowledge of the uncertainty associated with it. In this case we can use the parameter posterior distributions to calculate a value distribution at the given radius:

[20]:
```
beta_inst(Quantity(600, 'kpc'), use_par_dist=True)
```

[20]: $[0.0031287637,\ 0.00301519,\ 0.0030929105,\ \ldots,\ 0.0029865089,\ 0.003054497,\ 0.0030366847] \frac{ct}{s\,arcmin^2}$

On a slight side note, you can also have a quick look at the predicted distribution in histogram form:

[21]:
```
beta_inst.predicted_dist_view(Quantity(600, 'kpc'))
```



If you have a model instance and wanted to make a prediction using parameter values other than those from the fitting process, you can use the model method and manually pass parameters:

[22]:
```
beta_inst.model(Quantity(600, 'kpc'), Quantity(2, ''), Quantity(240, 'kpc'),␣
↪Quantity(2, 'ct/(arcmin^2s)'))
```

[22]: $3.7082667 \times 10^{-5} \frac{ct}{s\,arcmin^2}$

There are also convenient ways to calculate derivatives of the model, and again we can either return a single value or a distribution (again by setting the `use_par_dist` argument). In general, if there is an analytical solution to the 1st derivative of a model that has been implemented in XGA, then the analytical solution is what is being used to calculate

the value. If there is no analytical solution then the scipy `derivative` function is being used, and the user must pass an appropriate `dx` value - this function is also what is used to calculate nth order derivatives:

[23]:
```
beta_inst.derivative(Quantity(600, 'kpc'))
```

[23]: $[-1.407308 \times 10^{-5}] \frac{\mathrm{ct}}{\mathrm{kpc\,s\,arcmin^2}}$

[24]:
```
beta_inst.nth_derivative(Quantity(600, 'kpc'), dx=Quantity(0.01, 'kpc'), order=2, use_
↪par_dist=True)
```

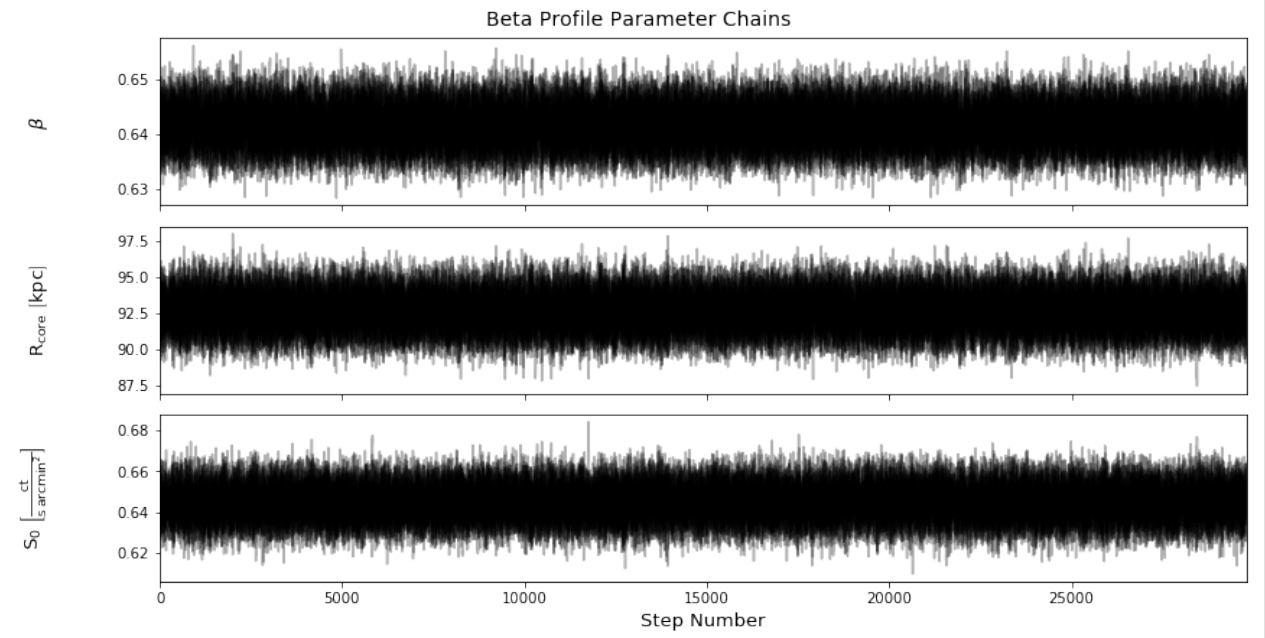
[24]: $[8.7143157 \times 10^{-8},\ 8.7500849 \times 10^{-8},\ 8.8227485 \times 10^{-8},\ \ldots,\ 8.723953 \times 10^{-8},\ 8.8259712 \times 10^{-8},\ 8.7700771 \times 10^{-8}] \frac{\mathrm{ct}}{\mathrm{s\,arcmin^2\,kpc^2}}$

Other mathematical functions include `inverse_abel` for performing inverse abel transforms on the models, and `volume_integral`, for performing volume integrals using the model. You can read more about them in the BaseModel1D API documentation.

4.1.7 Back to the profile

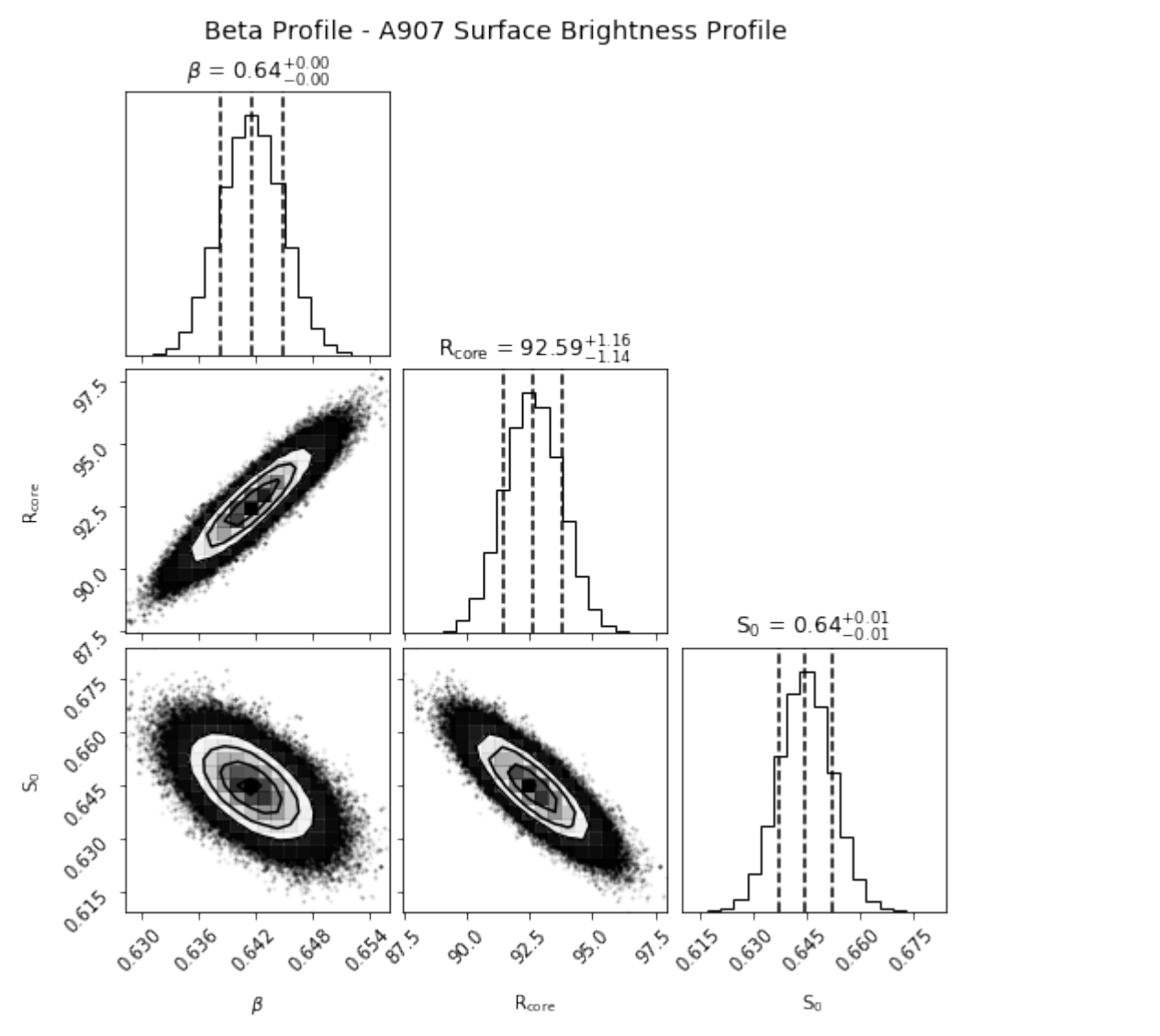
Now that I've shown you a bit of what you can do with the actual model object, I'm going to show you how the surface brightness profile object we started with makes use of the model. Firstly we can take a look at the MCMC chains produced during the fitting of the model using the `view_chains()` method, and again we must tell the method what the name of the model is:

[25]:
```
sb_prof.view_chains('beta')
```



We can also make the contour plots which are so often seen in relation to MCMC, both using the `corner` module and the `getdist` module, depending which you prefer:

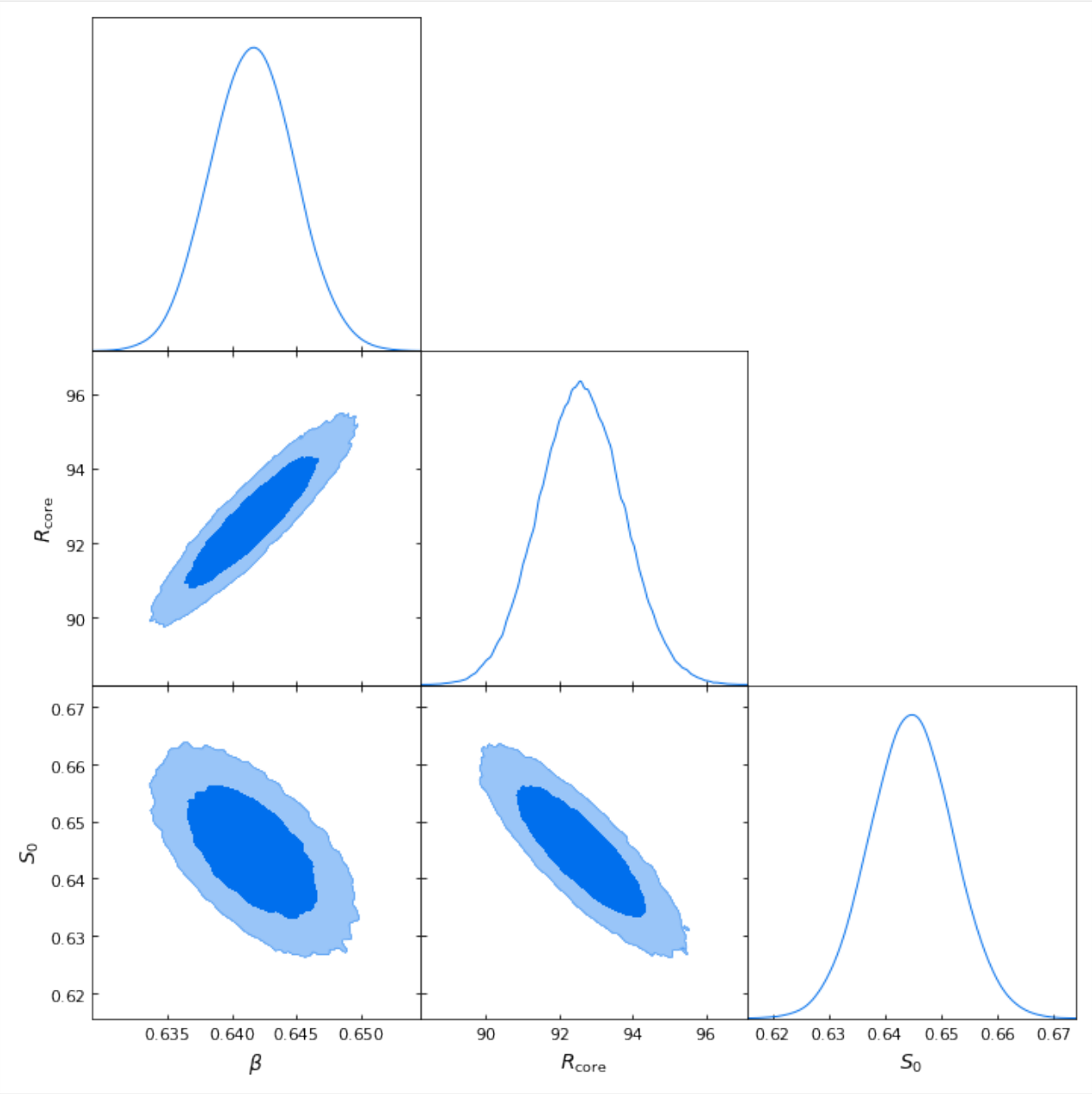
[26]:
```
sb_prof.view_corner('beta')
```



[27]:
```
sb_prof.view_getdist_corner('beta')
```

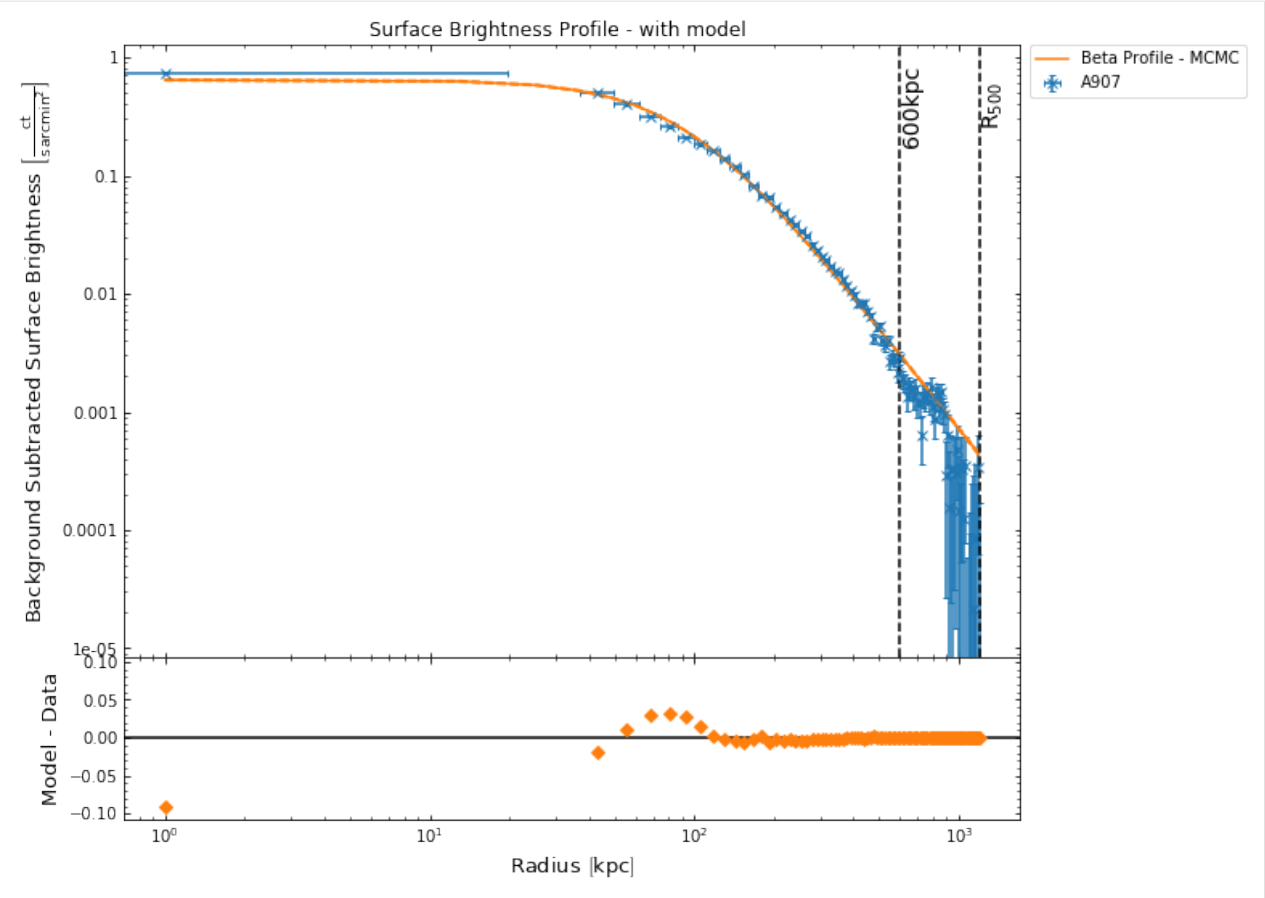
```
Removed no burn in
```

```
WARNING:root:fine_bins_2D not large enough for optimal density
WARNING:root:fine_bins_2D not large enough for optimal density
```



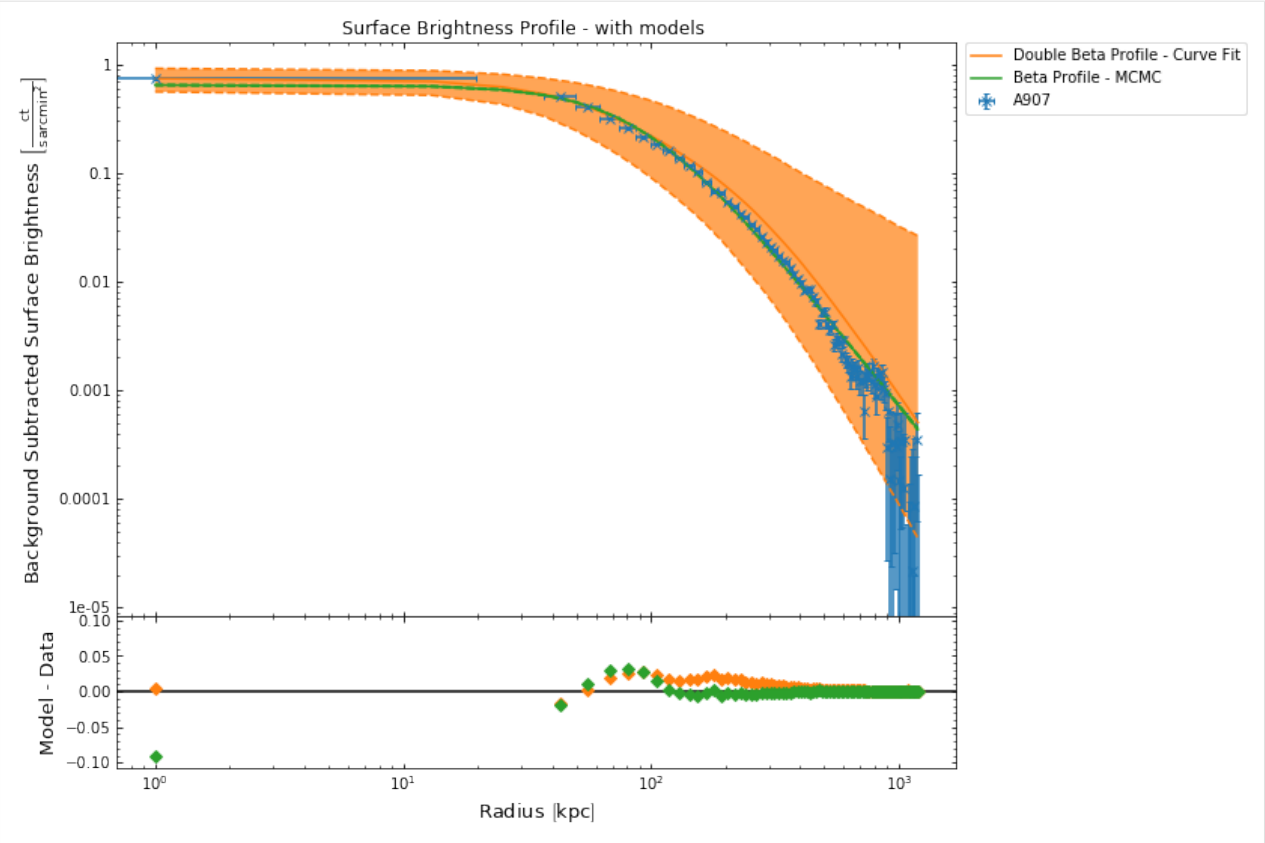
We can now view the profile with both data and model shown, using the profile `view()` method, and add some lines indicating particular radii using the `draw_rads` argument. There are a few other ways that users can configure the plots generated by the profile `view()` method, and they can be explored in the method documentation:

[28]:
```
sb_prof.view(draw_rads={'600kpc': Quantity(600, 'kpc'), 'R$_{500}$': src.r500})
```



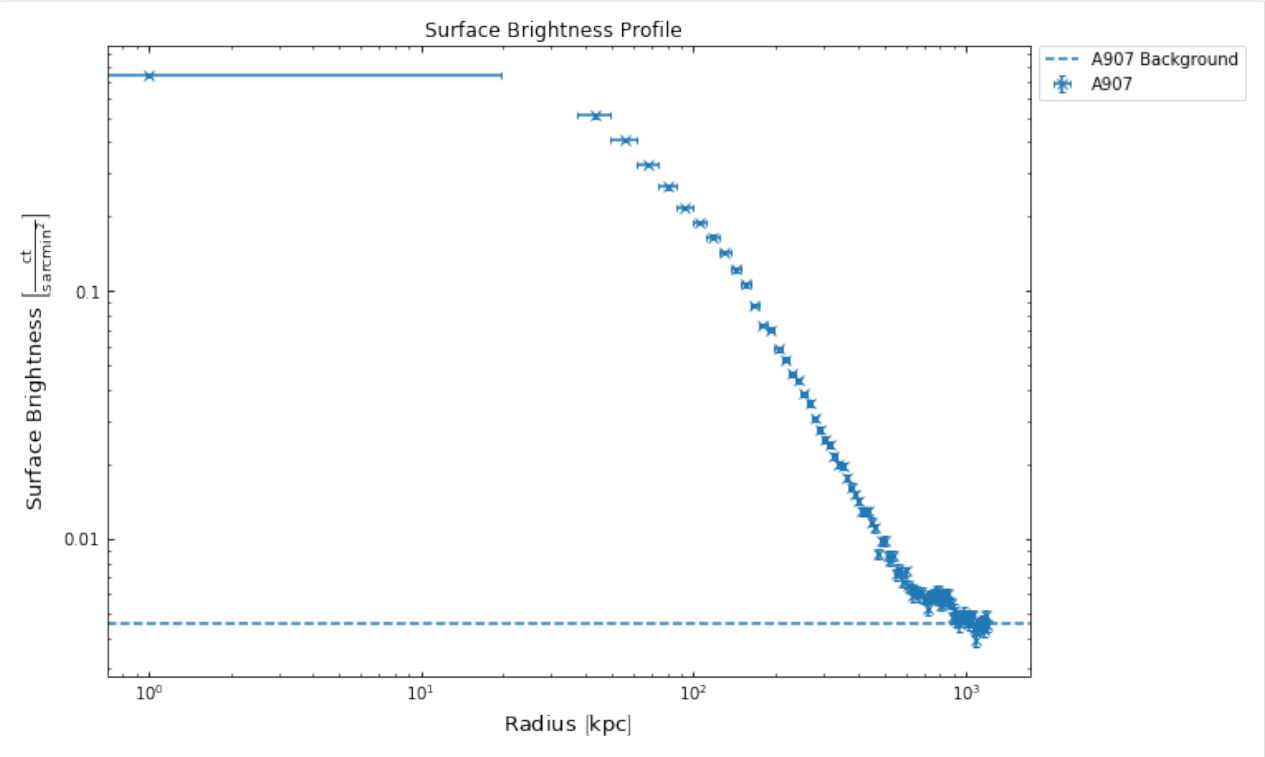
If we were to fit another model to this profile, and then view the profile, it would also be included in the plot generated by this `view()` method. To demonstrate this I will quickly fit a default double beta model to the profile (this time with the curve fit method), and then view the profile once again:

[29]:
```
sb_prof.fit('double_beta', method='curve_fit')
sb_prof.view()
```



We can also turn off the plotting of the models entirely, as well as the background subtraction:

[30]:
```
sb_prof.view(models=False, back_sub=False)
```

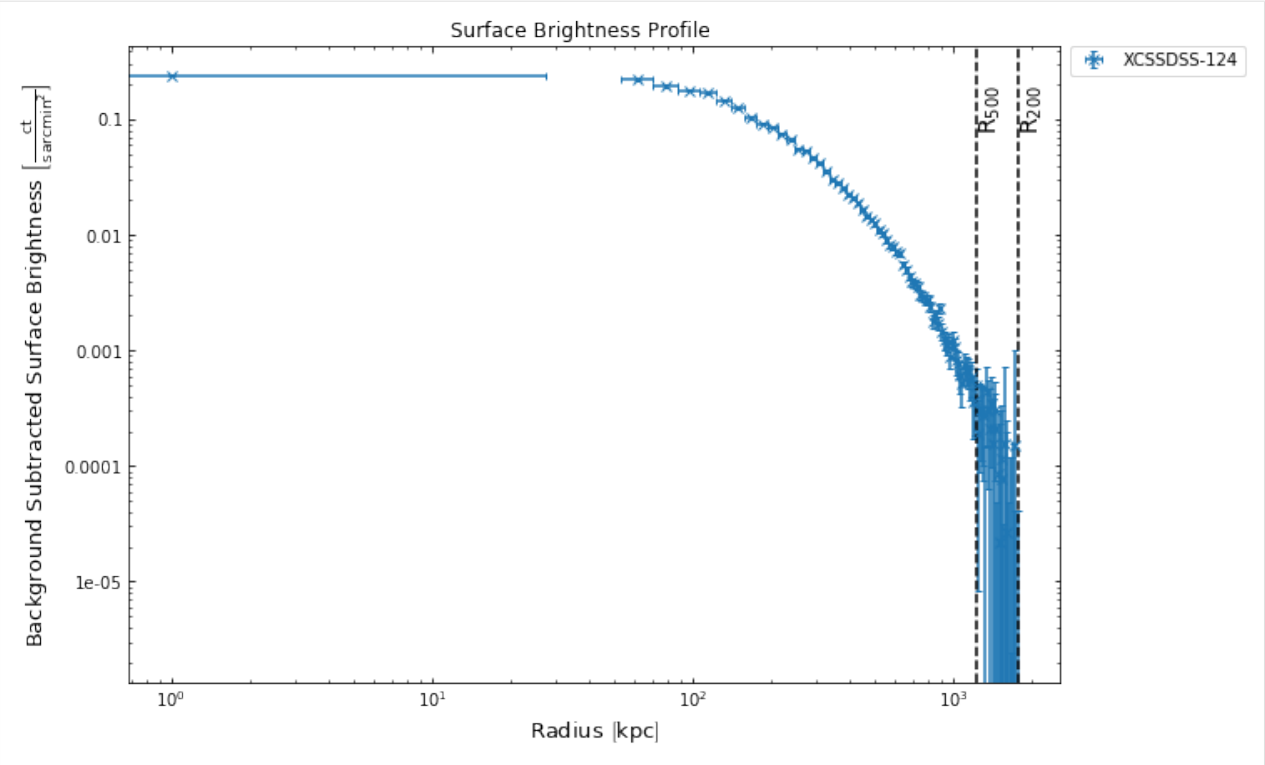


4.1.8 Viewing multiple profiles on one plot

It is often the case that we wish to view multiple profiles on a single plot, either for a comparison or to present results from a sample of objects. This is very easy to do with XGA profiles due to the introduction of the BaseAggregateProfile1D class, an instance of which contains multiple profile objects and allows them to easily be plotted together. The simplest way to create such an object is to add two (or more) profiles together, just using the normal '+' Python operator, though be aware that an error will be thrown if the profiles are not all of the same type.

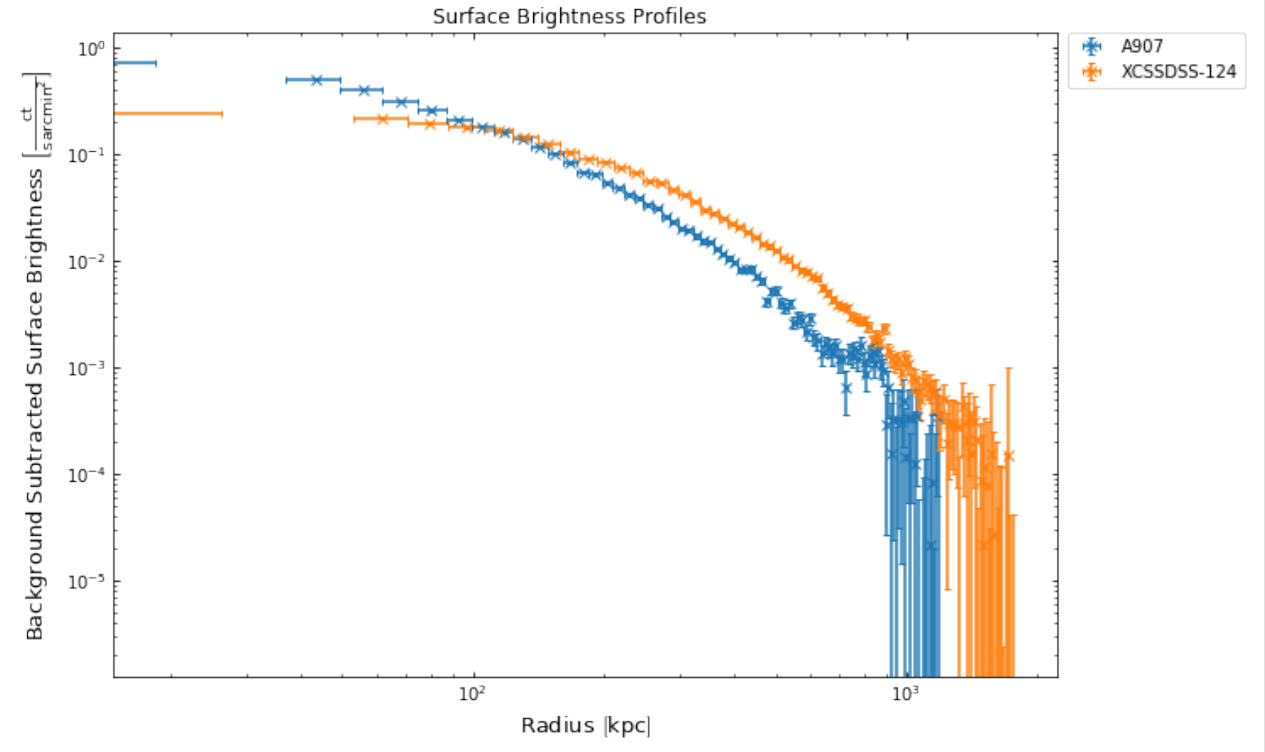
To demonstrate this I will declare another galaxy cluster object, and view the surface brightness profile to make sure one is generated:

[31]:
```
other_src = GalaxyCluster(0.80057775, -6.0918182, 0.251, 'XCSSDSS-124',␣
↪r500=Quantity(1220.11, 'kpc'),
                          r200=Quantity(1777.06, 'kpc'))
other_src.view_brightness_profile('r200')
```



Then I will simply add the profile I just created for XCSSDSS-124 to the profile we have been using throughout this tutorial, then call the `view()` method, just as I did before:

[32]:
```
(sb_prof + other_src.get_1d_brightness_profile('r200')).view()
```



4.2 Probing the baryon content of a galaxy cluster

The aim of this tutorial is to take you through the basic methods with which XGA can probe the hot gas that makes of the intra-cluster medium (ICM). We'll focus on measuring the radial gas density profiles of a sample of galaxy clusters here, then move on to using density profiles to measure gas masses.

Please note that I will not really be fully explaining how the methods to measure gas density from clusters work in this tutorial, that is the subject of a section in the 'Under the Hood' section, where the maths and concepts are detailed.

[1]:
```
from xga.samples import ClusterSample
from xga.sourcetools.density import inv_abel_fitted_model, ann_spectra_apec_norm
from xga.imagetools.psf import rl_psf
from xga.models import KingProfile1D, SimpleVikhlininDensity1D

from astropy.units import Quantity
import numpy as np
import pandas as pd
```

4.2.1 Defining a sample of galaxy clusters

I'm using the same sample of galaxy clusters that I used in the spectroscopy tutorial for the sake of convenience. I've chosen to use a ClusterSample rather than a GalaxyCluster as there are extra, convenient, methods built into ClusterSamples for dealing with samples of gas masses, though I will also show how you interact with gas density profiles/gas masses for an individual cluster:

[2]:
```
# Setting up the column names and numpy array that go into the Pandas dataframe
column_names = ['name', 'ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']
cluster_data = np.array([['XCSSDSS-124', 0.80057775, -6.0918182, 0.251, 1220.11, 1777.
↪06, 109.55, 4.49],
                         ['XCSSDSS-2789', 0.95553986, 2.068019, 0.11, 1039.14, 1519.
↪79, 38.90, 2.83],
                         ['XCSSDSS-290', 2.7226392, 29.161021, 0.338, 935.58, 1359.37,
↪ 105.10, 5.99],
                         ['XCSSDSS-134', 4.9083898, 3.6098177, 0.273, 1157.04, 1684.
↪15, 108.60, 4.79]])

# Possibly I'm overcomplicating this by making it into a dataframe, but it is an␣
↪excellent data structure,
#  and one that is very commonly used in my own analyses.
sample_df = pd.DataFrame(data=cluster_data, columns=column_names)
sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']] = \
    sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']].
↪astype(float)

# Defining the sample of four XCS-SDSS galaxy clusters
demo_smp = ClusterSample(sample_df["ra"].values, sample_df["dec"].values, sample_df["z
↪"].values,
                         sample_df["name"].values, r200=Quantity(sample_df["r200"].
↪values, "kpc"),
                         r500=Quantity(sample_df["r500"].values, 'kpc'),␣
↪richness=sample_df['richness'].values,
                         richness_err=sample_df['richness_err'].values)
```

```
Declaring BaseSource Sample: 100%|| 4/4 [00:01<00:00,  2.74it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:21<00:00,  5.45s/it]
Generating products of type(s) image: 100%|| 4/4 [00:00<00:00,  8.05it/s]
```

(continues on next page)

(continued from previous page)

```
Generating products of type(s) expmap: 100%|| 4/4 [00:00<00:00,  8.64it/s]
Setting up Galaxy Clusters: 100%|| 4/4 [00:04<00:00,  1.21s/it]
```

4.2.2 Surface brightness based gas density measurement

This first method uses the surface brightness profile of a cluster, as well as assumptions about its geometry and plasma emission, to measure the three-dimensional gas density profile. There are two broad steps to this process; we make a surface brightness profile of the cluster, fit a model to it, then use that to deduce the three-dimensional emissivity and then we use a plasma emission model, combined with an absorption model, to convert that emissivity into a density.

Of course the implementation (and details) of this method are more complicated than that makes it sound, but a convenience function has been added to XGA that goes through the whole process, `inv_abel_fitted_model()`, for which the full docstring can be found here.

PSF correction

As we are likely to be using small annular bins for the brightness profile, and the XMM-Newton telescope has a non-negligible point spread function (PSF), we have to account for this effect with a correction, bearing in mind that these effects also vary spatially. The PSFs of the three XMM cameras have been well studied, with Read et al. (2011) producing comprehensive models that take into account the complex two-dimensional structure of the effects. In XGA we can use those models to correct images directly, and prior to the generation of a brightness profile, whereas previously a one-dimensional model was used to correct after the profile had been generated.

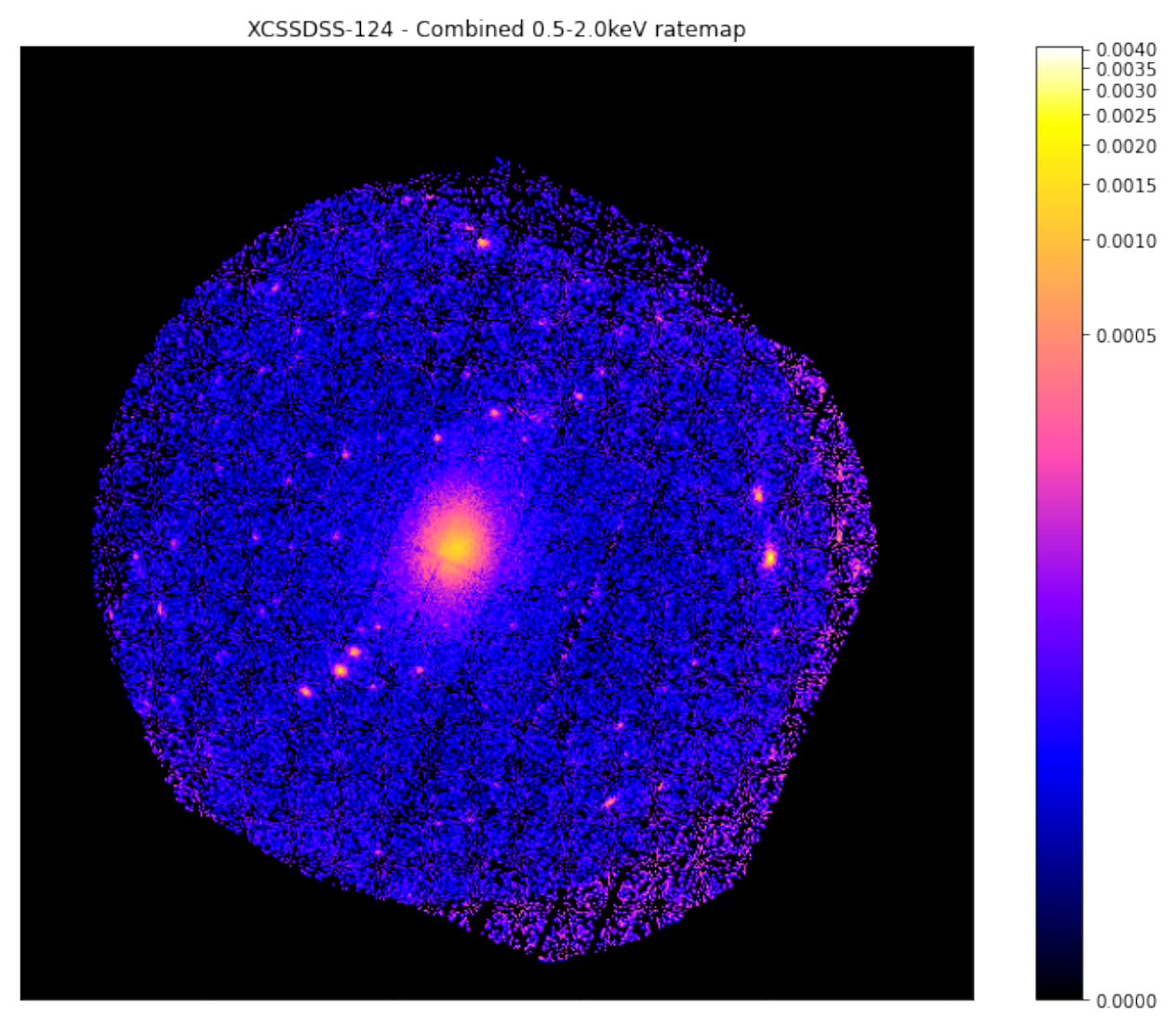
A full explanation of how this process works can be found in PSF correction of XMM images in the 'Under the Hood' section, and the documentation for the correction function is here. Please note that the `inv_abel_fitted_model()` function will automatically generate and use PSF corrected images, but I'm calling the correction function here manually as a demonstration (you can set custom chunking and iteration factors in the function call but I'm leaving it default). You can also automatically run PSF correction when a ClusterSample is declared by setting `psf_corr=True`:

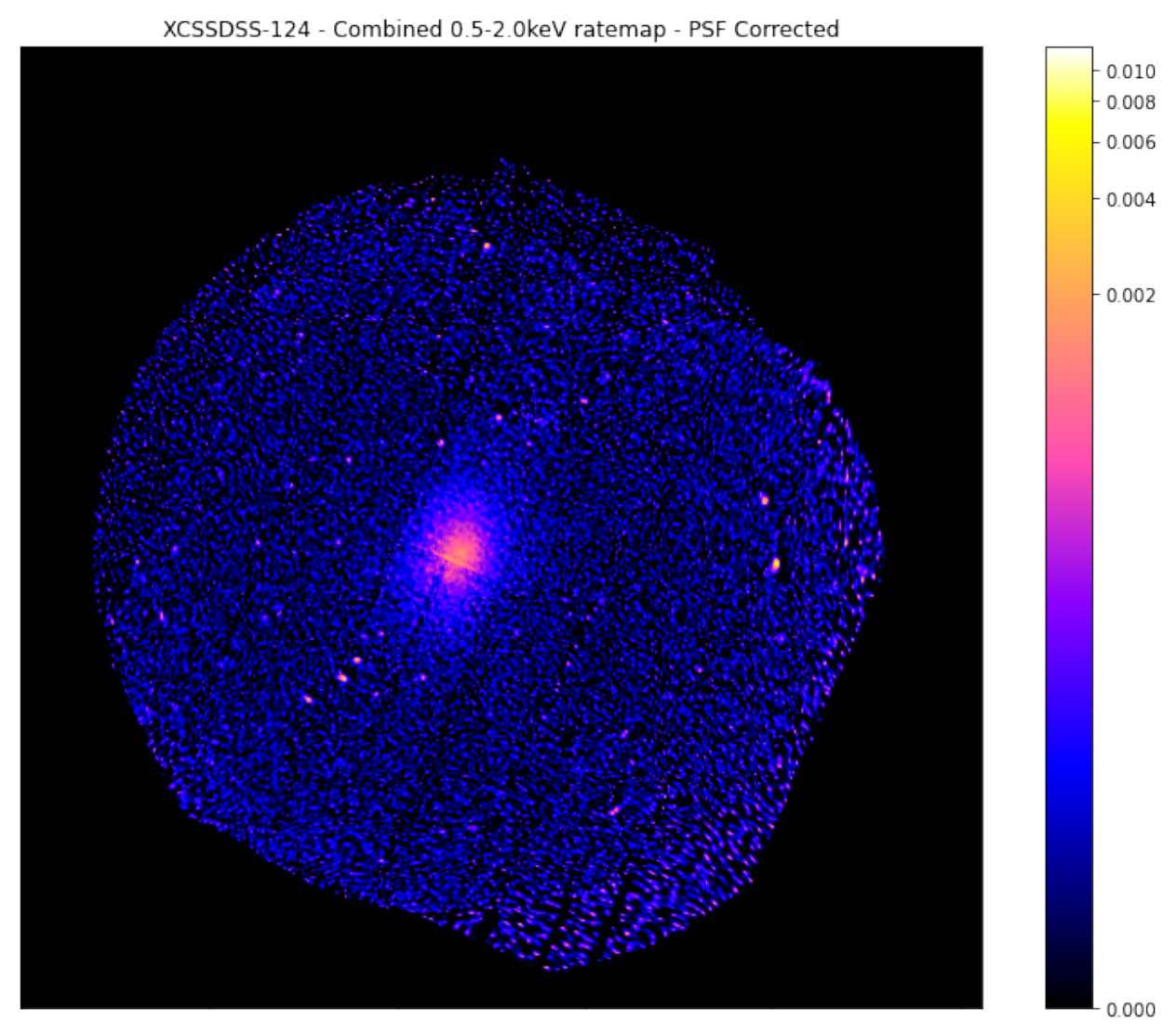
[3]:
```
rl_psf(demo_smp)
```

```
Preparing PSF generation commands: 100%|| 4/4 [00:00<00:00, 15.82it/s]
Generating products of type(s) psf: 100%|| 12/12 [00:04<00:00,  2.77it/s]
PSF Correcting Observations - Currently complete: 100%|| 4/4 [00:35<00:00,  8.82s/it]
Generating products of type(s) image: 100%|| 4/4 [00:00<00:00,  7.76it/s]
```

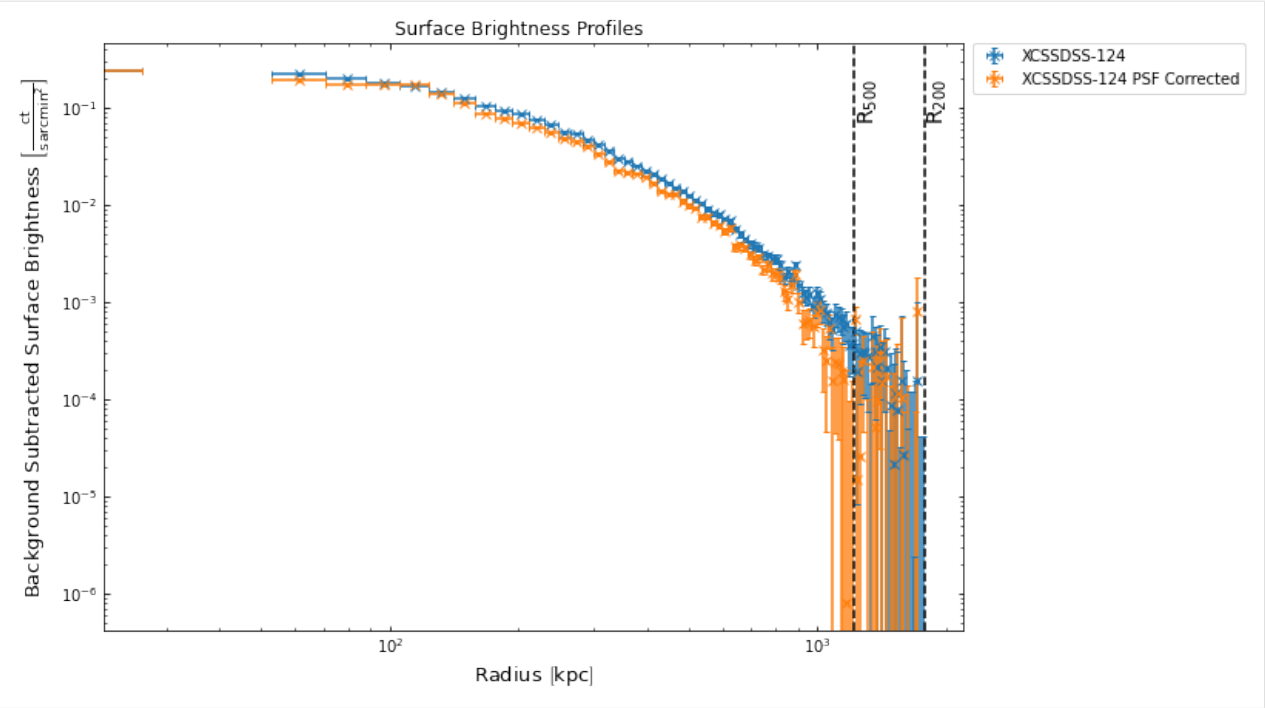
We can see the effect of this correction on the images and brightness profiles, with a demonstration using the first cluster in the sample (XCSSDSS-124):

[4]:
```
rt = demo_smp[0].get_combined_ratemaps(psf_corr=False)
rt.view()

rt = demo_smp[0].get_combined_ratemaps(psf_corr=True)
rt.view()
```


XCSSDSS-124 - Combined 0.5-2.0keV ratemap
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000



[5]:
```
demo_smp[0].view_brightness_profile('r200')
```



Running the density calculation function

There are many arguments that can be passed to this function, and I'm not going to explain them all here, please see the function documentation for explanations. However, I will go through the most important arguments you need to consider when running this function:

- `model` - This is the model that will be fitted to the surface brightness profile, and then transformed to infer the three-dimensional countrate per volume element radial distribution. This can either be a model name, a list of model names (if dealing with a sample), a model instance (see profile tutorial for context), or a list of model instances (if dealing with a sample).
- `outer_radius` - The radius (or radii, if we're dealing with a sample) out to which the profiles should be generated. This has to be carefully balanced, as you don't want to choose too small a radius and have too much cluster emission emission in the background annulus, and you don't want to choose too large a radius and start hitting the edge of the detector.
- `num_dens` - If this is true then a number density profile will be calculated, if False then a mass density profile.
- `obs_id` and `inst` - You can pass an ObsID and instrument (or lists of these, if dealing with a sample) if you wish to specify a single image to measure density with.
- `conv_temp` - The temperature used for the plasma emission model can be specified manually (for samples as well as individual clusters). The default behaviour is to measure the temperature within the `conv_outer_radius` and use that, and if that fit fails then to assume the temperature is 3keV.
- `conv_outer_radius` - The radius (radii for samples) for which spectra are generated (and fitted if `conv_temp` isn't set) to calculate the conversion between count-rate per volume element and density, which involves the use of the ARFs (see the spectroscopic tutorial for an explanation)

Here I generate profiles out to $1.2R_{500}$ of each cluster, and use spectra generated within R_{500} to measure the temperature and calculate the conversion. I also choose to fit a beta profile, and increase the number of steps the MCMC samplers will take from the default (20000) to 40000. This function also returns a list of profiles (if a profile fails

to generate for some reason a None will take the profile's place in the list), as well as storing the profiles within the source objects in the sample:

[6]:
```
sb_d_profs = inv_abel_fitted_model(demo_smp, 'beta', outer_radius=1.2*demo_smp.r500,␣
↪conv_outer_radius='r500',
                                   num_steps=40000)
sb_d_profs
```

```
Generating products of type(s) spectrum: 100%|| 12/12 [38:34<00:00, 192.89s/it]
Running XSPEC Fits: 100%|| 4/4 [00:20<00:00,  5.17s/it]
Running XSPEC Simulations: 100%|| 4/4 [00:02<00:00,  1.50it/s]
Fitting data, inverse Abel transforming, and measuring densities: 100%|| 4/4 [07:42
↪<00:00, 115.59s/it]
```

[6]:
```
[<xga.products.profile.GasDensity3D at 0x7f0890f79970>,
 <xga.products.profile.GasDensity3D at 0x7f0890f79ca0>,
 <xga.products.profile.GasDensity3D at 0x7f0890f79b50>,
 <xga.products.profile.GasDensity3D at 0x7f0890f79e20>]
```

4.2.3 Annular spectra gas density method

The alternative method is based on a similar idea, but instead we generate annular spectra, fit an absorbed plasma model to each annulus, and use the normalisation of that model (and assumptions about the cluster geometry) to measure the density. The main disadvantage of this method is that you require a lot of X-ray counts in a spectrum to get a good fit, so the annuli must be **much** larger than the brightness profile method. However, for clusters with high enough quality data this method can be a check of the results measured from the surface brightness profile method.

This method doesn't really require any PSF correction due to the large annuli required for the annular spectra to have enough counts for a successful fit, and we will set a minimum size to ensure that PSF considerations are unnecessary. Again there are more arguments for this function than I want to go into here, but you can read about all of them in the full function documentation. I will again take you through the most important arguments, however:

- `outer_radius` - The radius (or radii, if we're dealing with a sample) out to which the profiles should be generated. This has to be carefully balanced, as you don't want to choose too small a radius and have too much cluster emission emission in the background annulus, and you don't want to choose too large a radius and start hitting the edge of the detector.
- `num_dens` - If this is true then a number density profile will be calculated, if False then a mass density profile.
- `annulus_method` - The method by which the annuli are designated, the default is a minimimum signal to noise value (`min_snr`) which is targetted, though if it cannot be attained in all annuli the function will allow the number of annuli to drop to four, use them, and give a warning to the user.
- `min_snr` - The minimum signal to noise target for annuli.
- `min_width` - The minimum width of each annulus, default is 20 arcseconds to try and remove the need for accounting for PSF effects.
- `use_worst` - If set to True then the annuli generation algorithm will use the 'worst' observation (defined by global signal to noise of the cluster), to decide how big to make the annuli.

[7]:
```
as_d_profs = ann_spectra_apec_norm(demo_smp, 1.2*demo_smp.r500, min_snr=30)
as_d_profs
```

```
/home/dt237/code/PycharmProjects/XGA/xga/sourcetools/temperature.py:152: UserWarning:␣
↪The requested annuli for XCSSDSS-290 cannot be created, the data quality is too low.
↪ As such a set of four annuli will be returned
  warn("The requested annuli for {s} cannot be created, the data quality is too low.␣
↪As such a set "
```

(continues on next page)

(continued from previous page)

```
Generating products of type(s) spectrum: 100%|| 12/12 [50:47<00:00, 253.96s/it]
Generating products of type(s) annular spectrum set components: 100%|| 129/129 [29:49
↪<00:00, 13.87s/it]
Running XSPEC Fits: 100%|| 43/43 [01:12<00:00,  1.69s/it]
Generating density profiles from annular spectra:  25%|       | 1/4 [00:08<00:26,  8.
↪85s/it]/home/dt237/software/anaconda3/envs/conda4-xcsim/lib/python3.8/site-packages/
↪astropy/units/quantity.py:481: RuntimeWarning: invalid value encountered in sqrt
  result = super().__array_ufunc__(function, method, *arrays, **kwargs)
Generating density profiles from annular spectra: 100%|| 4/4 [00:36<00:00,  9.06s/it]
```

[7]:
```
[<xga.products.profile.GasDensity3D at 0x7f0890f8fac0>,
 <xga.products.profile.GasDensity3D at 0x7f08913a3700>,
 <xga.products.profile.GasDensity3D at 0x7f0890f8b2b0>,
 <xga.products.profile.GasDensity3D at 0x7f0890f8b130>]
```

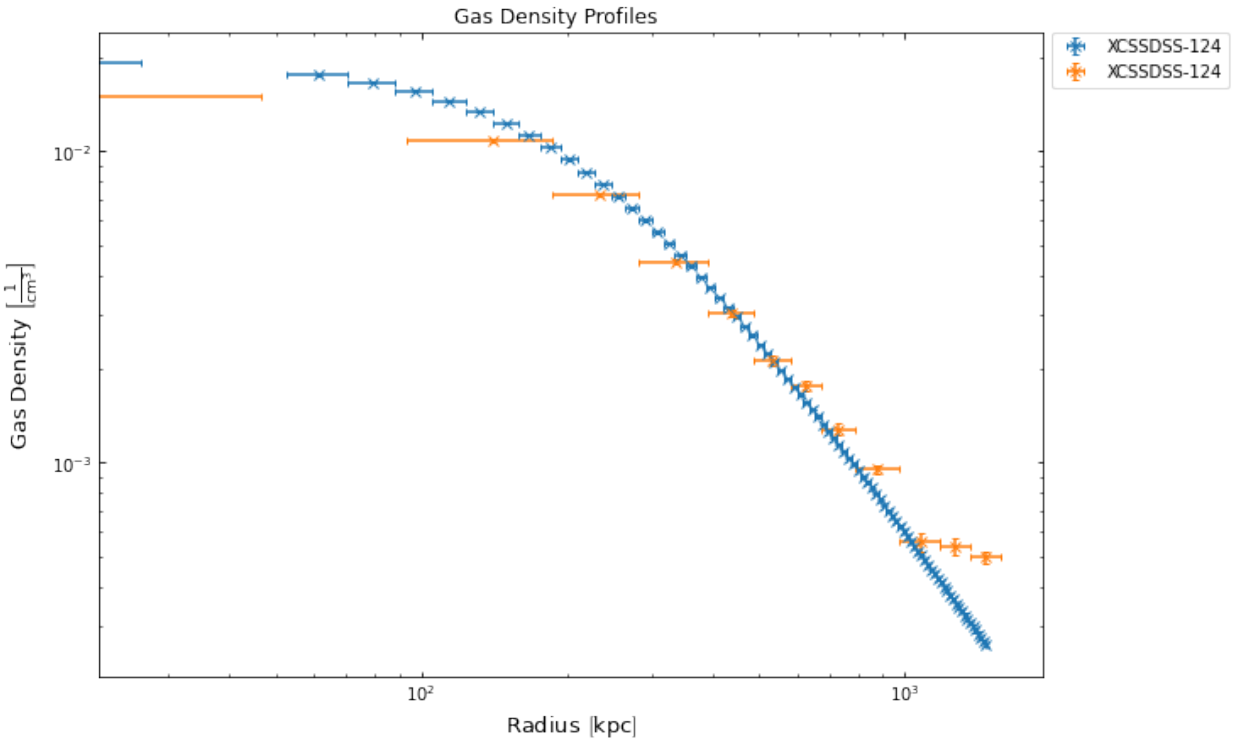
[8]:
```
demo_smp[0].get_combined_profiles('1d_apec_norm').values
```

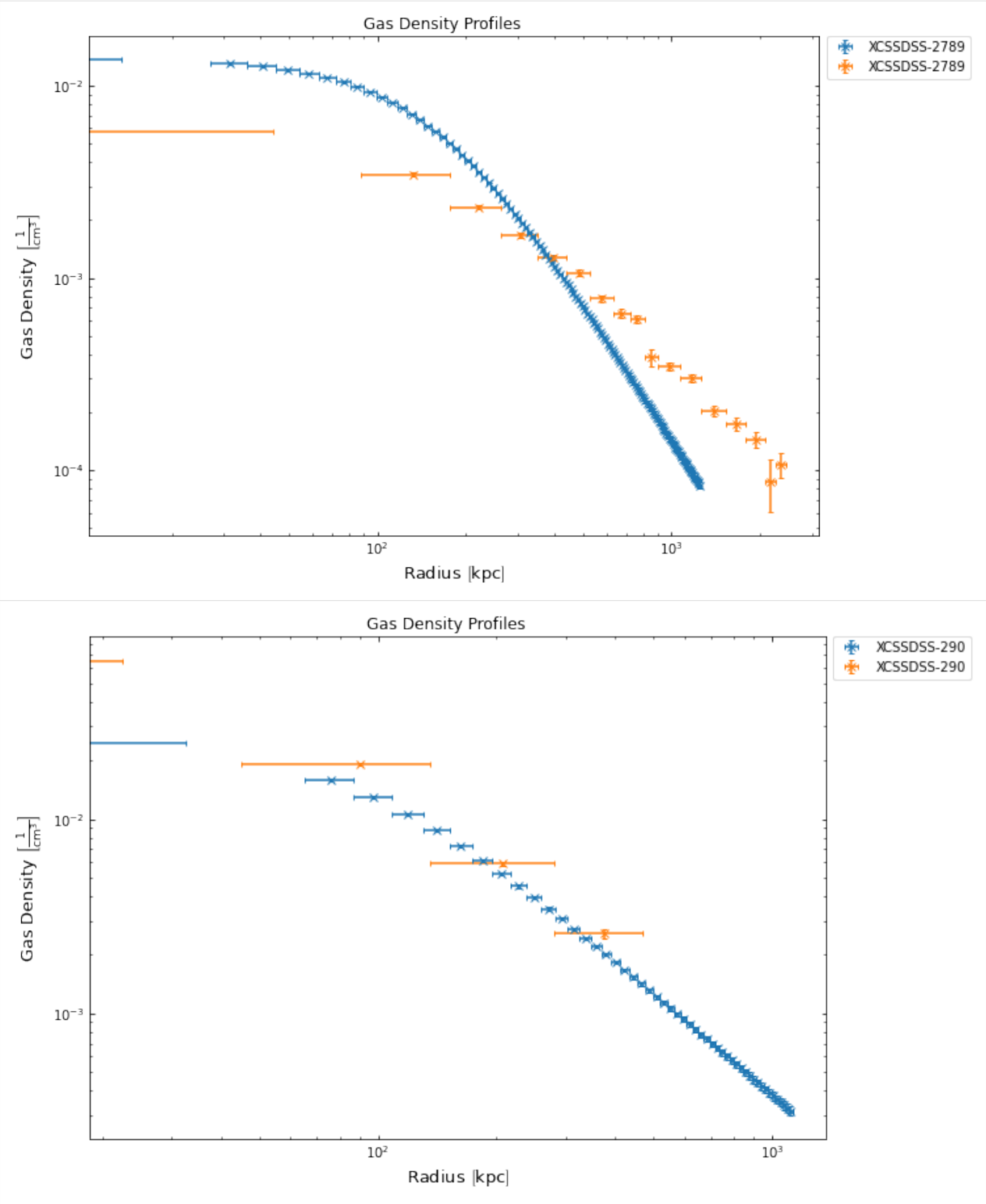
[8]:
```
[0.000784351, 0.00154955, 0.00140047, 0.00120906, 0.00072586, 0.000546847, 0.000442686, 0.000395873, 0.000489743, 0.0003
```

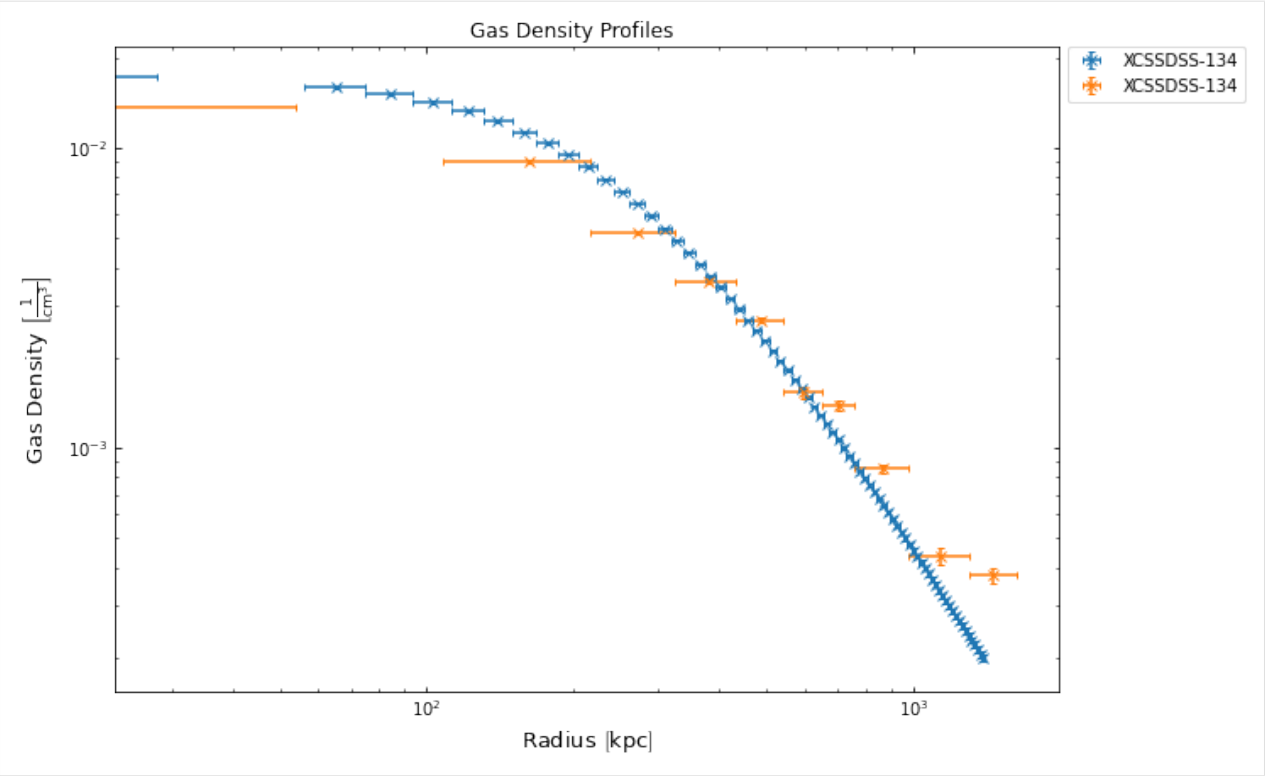
4.2.4 Viewing the gas density profiles

The profile tutorial demonstrated how to use the profile view method to make a visualisation of the radial profile data, as well as the fact that we can add two compatible profiles together to compare them. I will use that ability again here to view (and compare) the density profiles from the different methods. At first I will simply draw the profiles from the lists which were outputted by the measurement functions, but then I will demonstrate how to retrieve a profile directly from a source:

[9]:
```
for src_ind in range(0, len(demo_smp)):
    (sb_d_profs[src_ind]+as_d_profs[src_ind]).view()
```

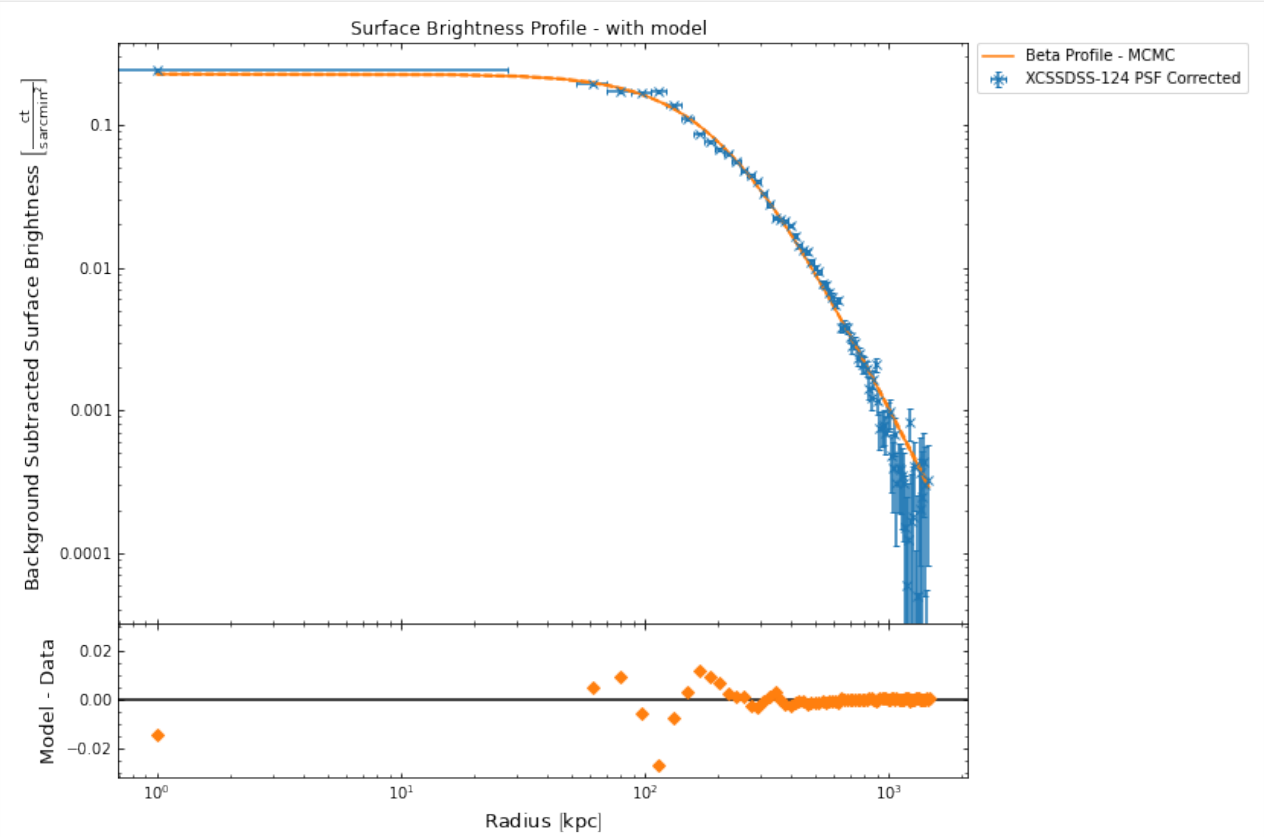


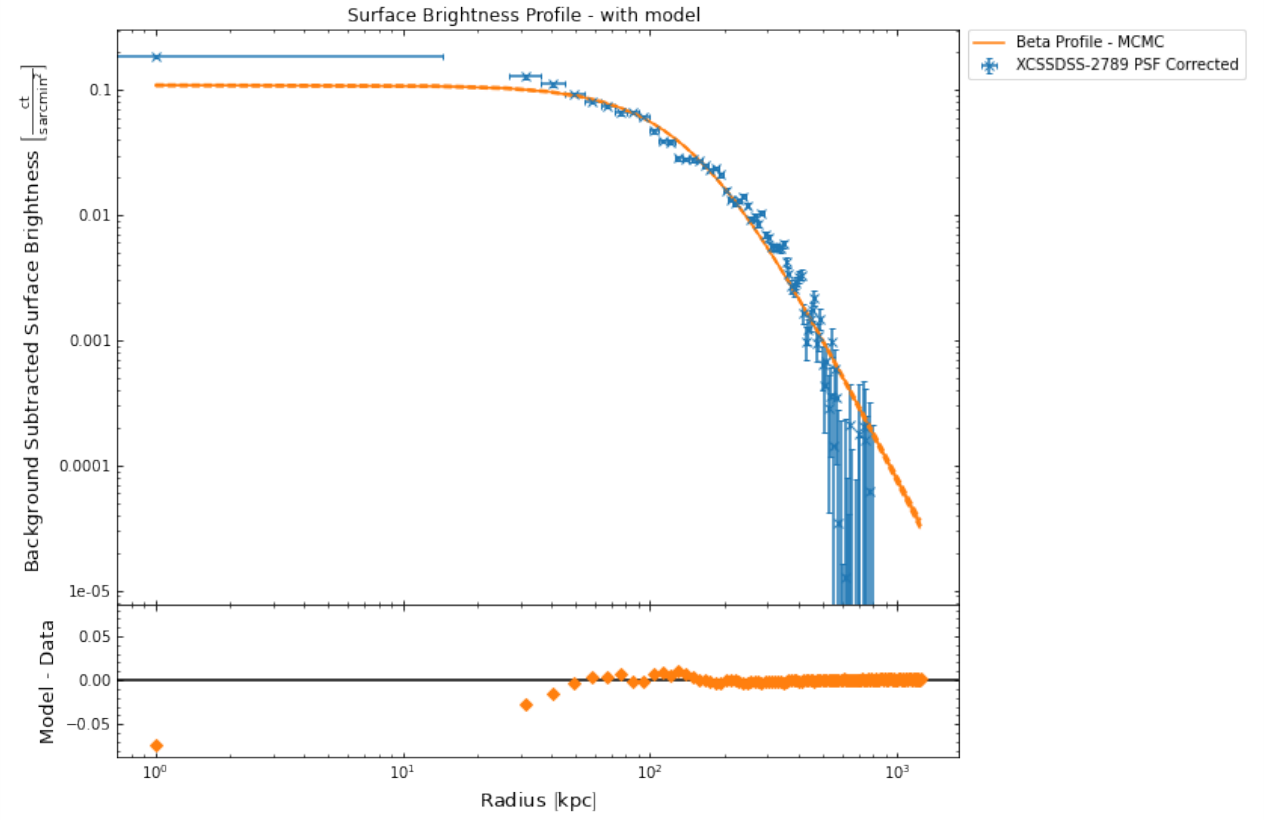

Gas Density Profiles
XCSSDSS-2789
XCSSDSS-2789
Gas Density [$\frac{1}{cm^3}$]
10^{-2}
10^{-3}
10^{-4}
10^{2}
10^{3}
Radius [kpc]
Gas Density Profiles
XCSSDSS-290
XCSSDSS-290
Gas Density [$\frac{1}{cm^3}$]
10^{-2}
10^{-3}
10^{2}
10^{3}
Radius [kpc]

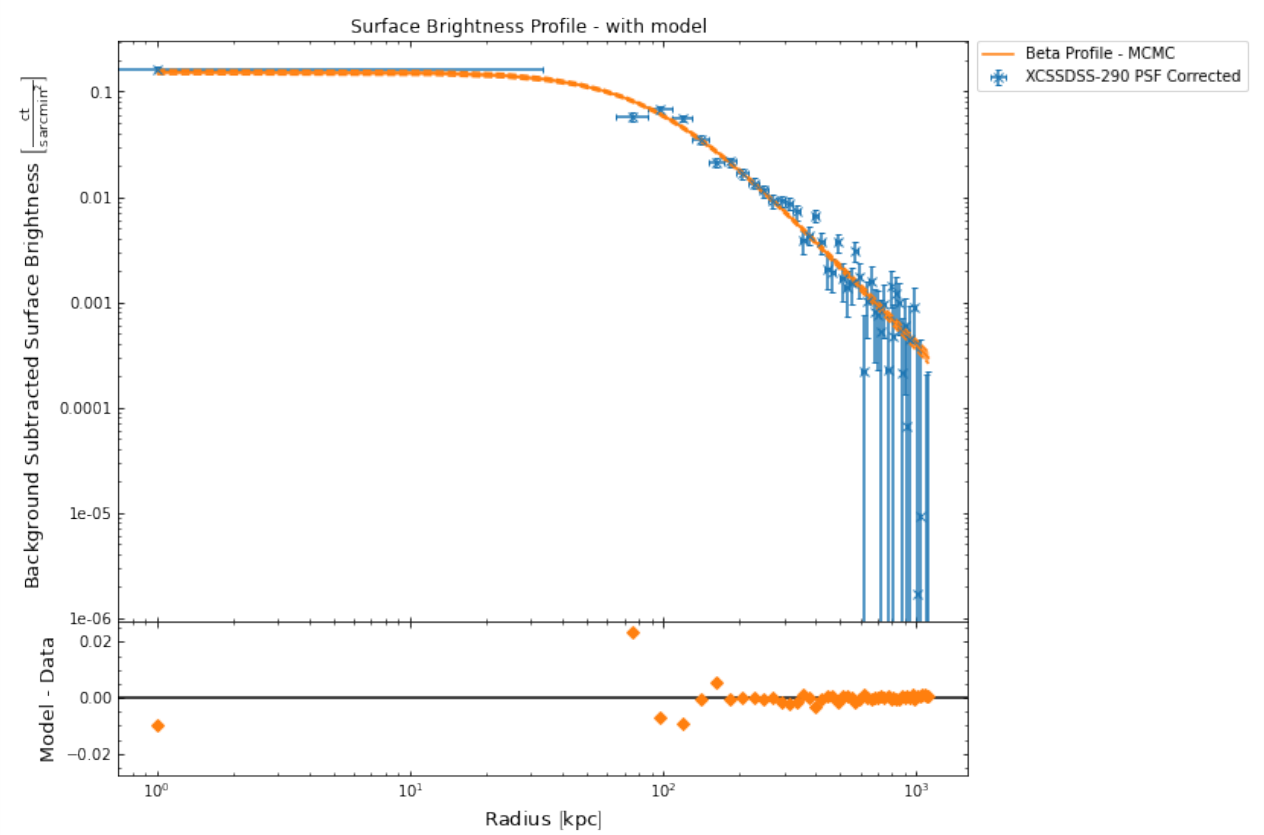


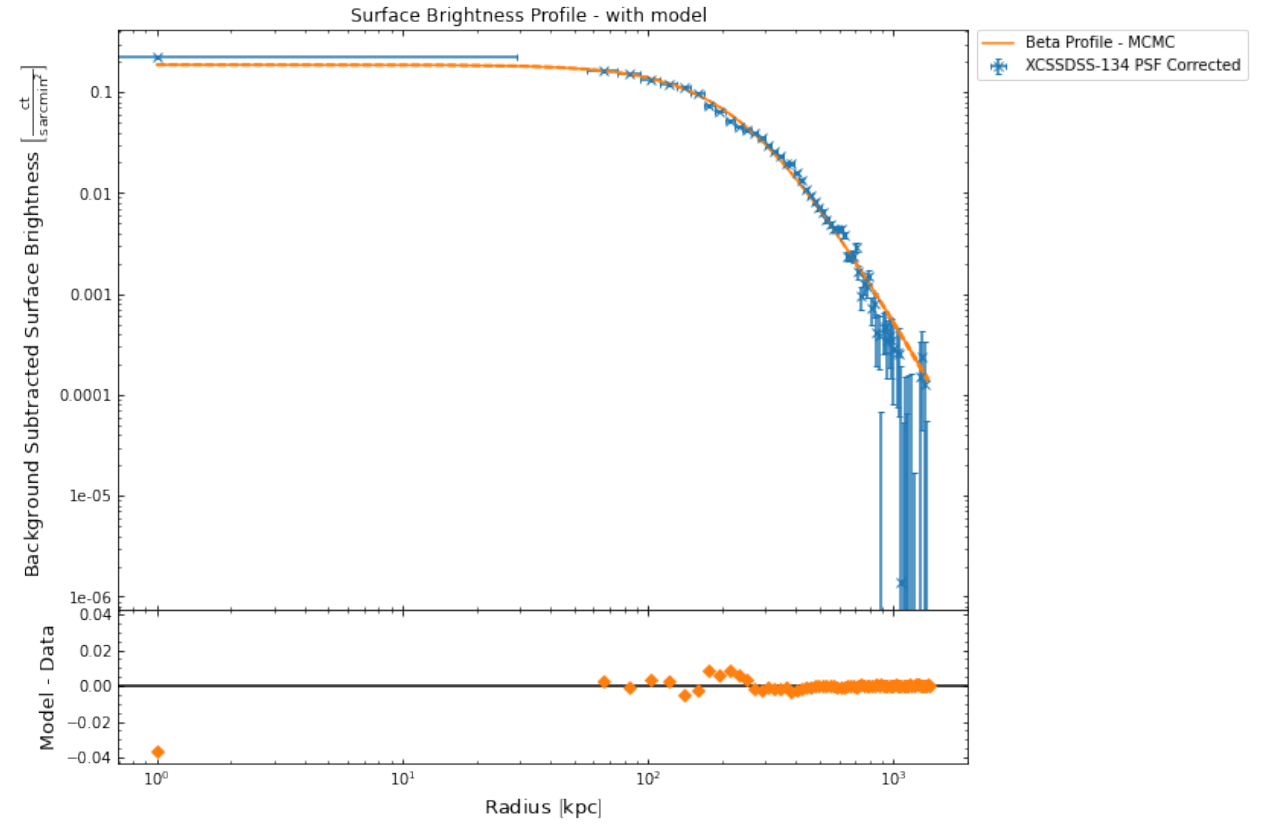
We can also view the surface brightness profiles that went into the creation of the density profiles from the first method, by cycling through them and using the `generation_profile` property:

[10]:
```
for d_prof in sb_d_profs:
    d_prof.generation_profile.view()
```


Surface Brightness Profile - with model
Beta Profile - MCMC
XCSSDSS-124 PSF Corrected
Background Subtracted Surface Brightness [ct / sarcmin²]
0.1
0.01
0.001
0.0001
Model - Data
0.02
0.00
−0.02
10⁰
10¹
10²
10³
Radius [kpc]

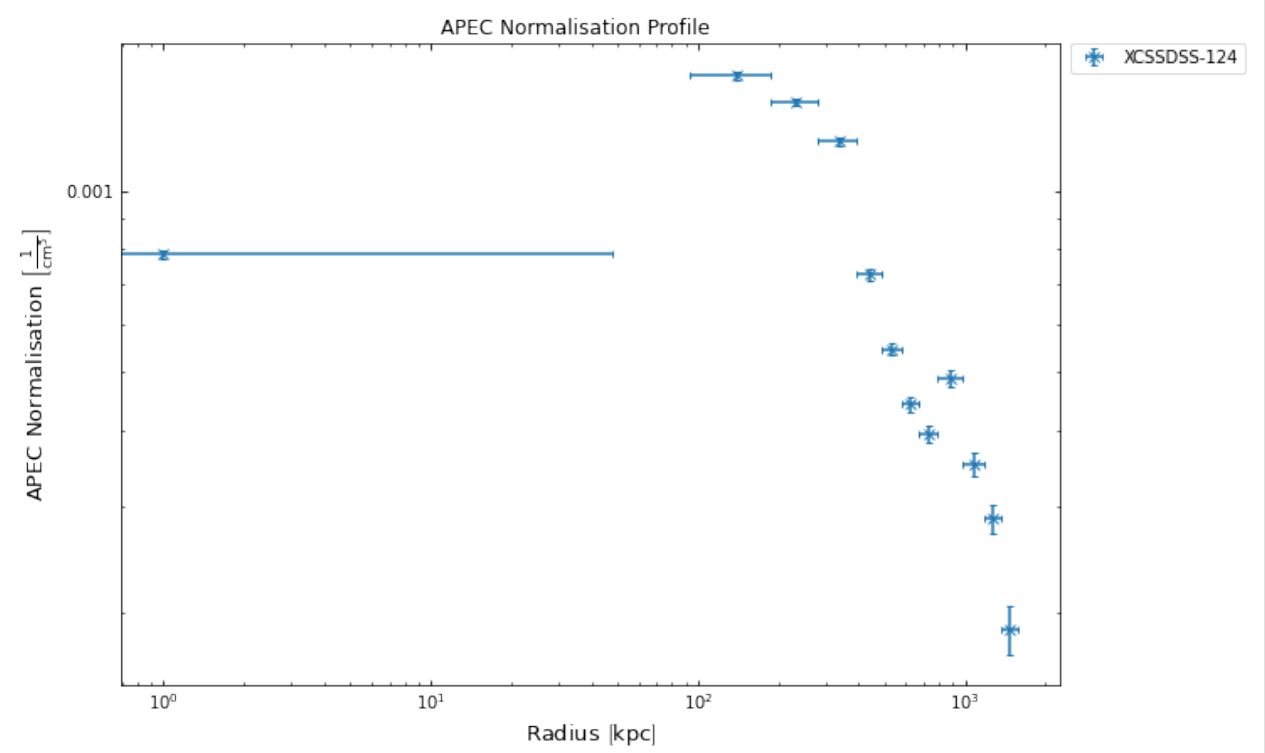

Surface Brightness Profile - with model
Beta Profile - MCMC
XCSSDSS-2789 PSF Corrected
Background Subtracted Surface Brightness [ct / sarcmin²]
0.1
0.01
0.001
0.0001
1e-05
Model - Data
0.05
0.00
−0.05
10⁰
10¹
10²
10³
Radius [kpc]

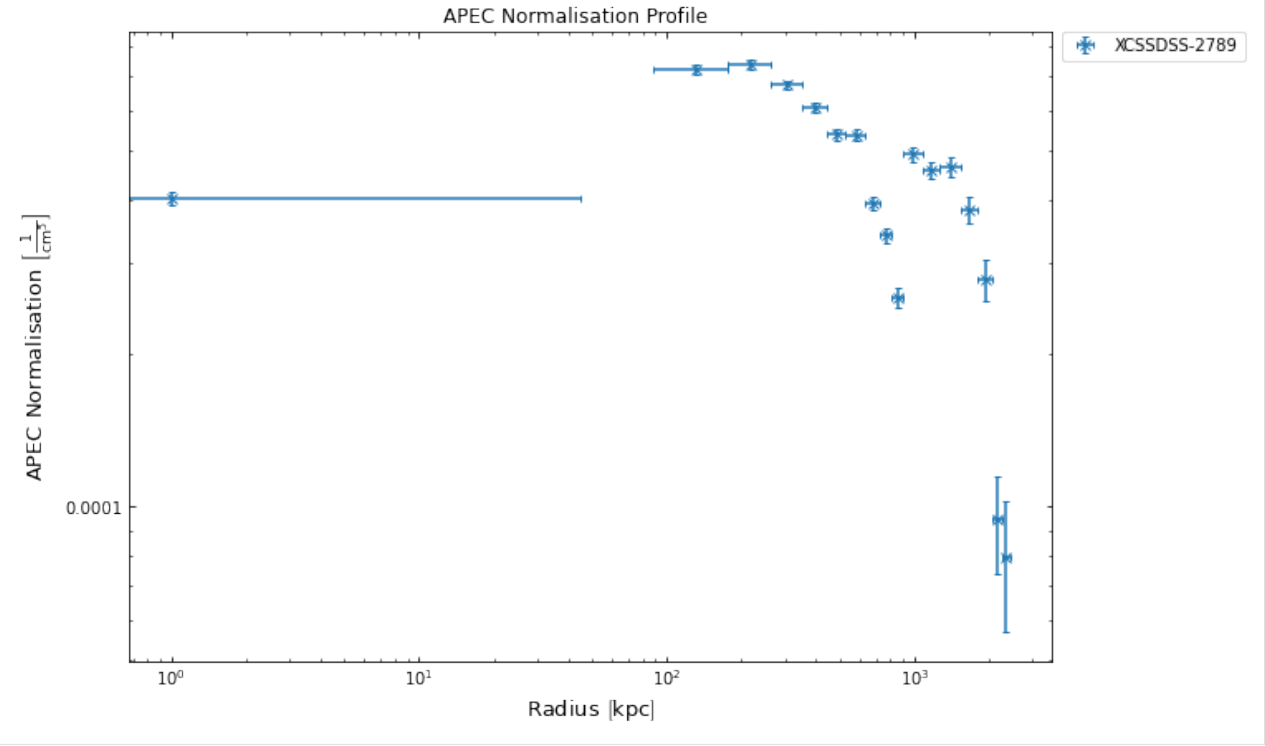

Surface Brightness Profile - with model
Beta Profile - MCMC
XCSSDSS-290 PSF Corrected
Background Subtracted Surface Brightness [ct / sarcmin²]
Model - Data
Radius [kpc]

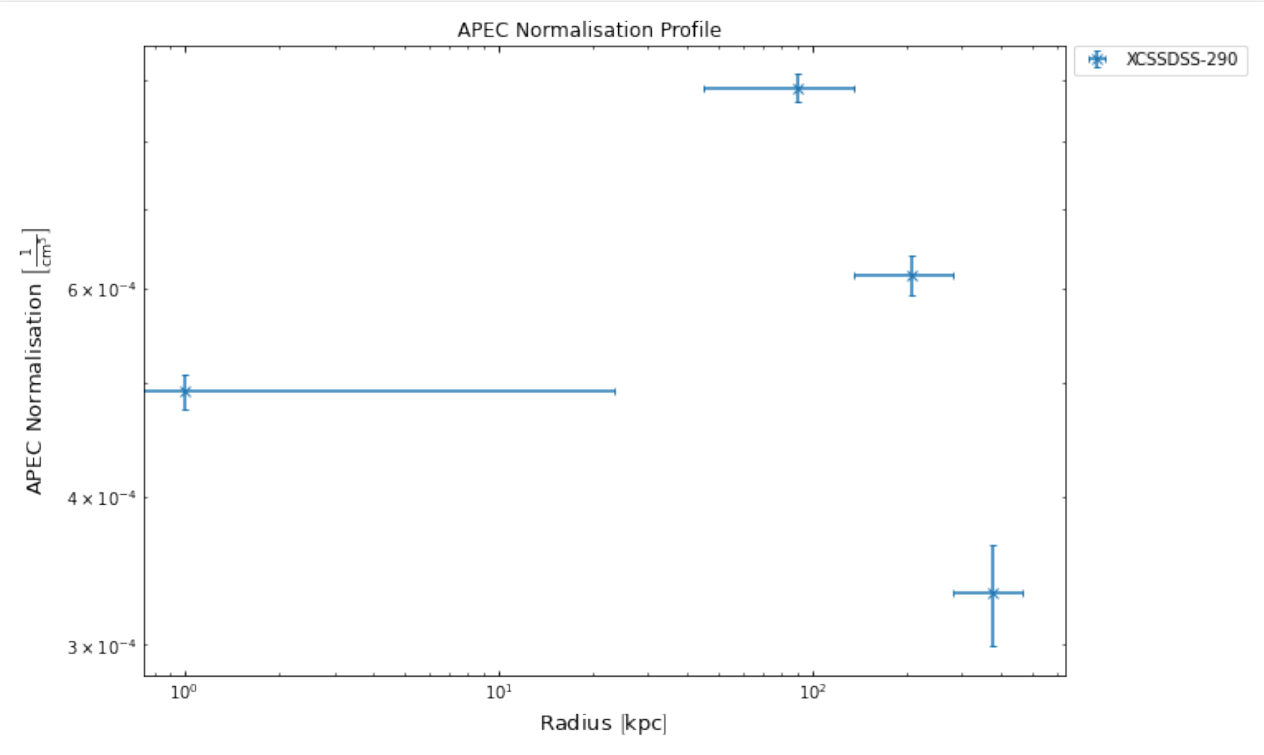

Surface Brightness Profile - with model
Beta Profile - MCMC
XCSSDSS-134 PSF Corrected
Background Subtracted Surface Brightness [ct / sarcmin²]
Model - Data
Radius [kpc]

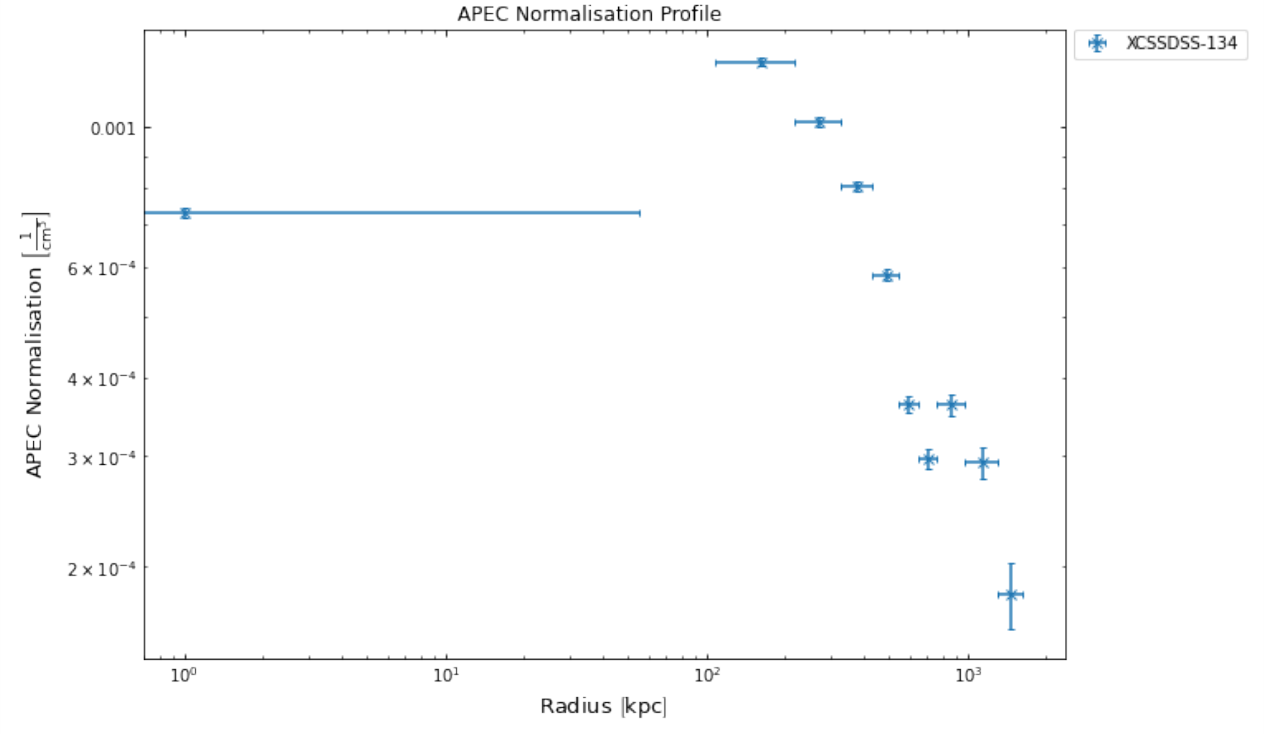
With the same property, `generation_profile`, we can actually have a look at the APEC normalisation profiles that we generated the annular spectra based density profiles from:

[11]:
```
for d_prof in as_d_profs:
    d_prof.generation_profile.view()
```






APEC Normalisation Profile
XCSSDSS-290
APEC Normalisation $[\frac{1}{cm^5}]$
6×10^{-4}
4×10^{-4}
3×10^{-4}
10^0
10^1
10^2
Radius [kpc]


APEC Normalisation Profile
XCSSDSS-134
APEC Normalisation $[\frac{1}{cm^5}]$
0.001
6×10^{-4}
4×10^{-4}
3×10^{-4}
2×10^{-4}
10^0
10^1
10^2
10^3
Radius [kpc]

4.2.5 Measuring a gas mass from density profiles

For a gas mass to be measured, we first must fit a model to the density profile, again using the profile `fit()` method. After this process has run through, we will then be able to request a gas mas at any radius we like, and have both an individual gas mass/uncertainty returned, and a gas mass distribution.

I am choosing to fit a simple King function to the density profile, and though I have imported the `KingProfile1D` class, I've only done this to show you the output of the `info()` method, and I won't actually be setting up a custom instance:

[12]:
```
KingProfile1D().info('grid')
```

```
+-----------------+-------------------------------------------------------------------------------
↪------+
| King Profile    |                                                                               ␣
↪       |
+=================+===============================================================================+
| DESCRIBES       | Gas Density                                                                   ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| UNIT            | solMass / Mpc3                                                                ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| PARAMETERS      | beta, r_core, norm                                                            ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| PARAMETER UNITS | , kpc, solMass / Mpc3                                                         ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| AUTHOR          | placeholder                                                                   ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| YEAR            | placeholder                                                                   ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| PAPER           | placeholder                                                                   ␣
↪       |
+-----------------+-------------------------------------------------------------------------------
↪------+
| INFO            | The un-projected version of the beta profile, suitable for a␣
↪simple fit |
|                 |  to 3D density distributions. Describes a simple isothermal␣
↪sphere.     |
+-----------------+-------------------------------------------------------------------------------
↪------+
```

Now I'm going to demonstrate how you can retrieve a gas density profile from a source object directly, rather than relying on the list output by the density profile generation functions. When you've only generated one density profile for a given source, just calling `get_density_profiles()` will return that for you without any extra information being passed in. If you've generated multiple profiles (be it with different methods or simply for different outermost radii) like we've done in this demonstration, then it will return a list containing all of the available profiles for the source:

[13]:
```
# Just calling it by itself gets you both of the density profiles generated for this particular source
#  Not necessarily what we want!
demo_smp[0].get_density_profiles()
```

[13]:
```
[<xga.products.profile.GasDensity3D at 0x7f0890f79970>,
 <xga.products.profile.GasDensity3D at 0x7f0890f8fac0>]
```

As such, we'll need to pass in some extra information to retrieve the exact profile we want to fit a model to. Seeing as you can generate these profiles in various different ways, the get method can take quite a lot of information to retrieve the exact profiles you want. The outermost radius can be quite a good one to use, but in this case, where we have two different profiles generated to the same outer radius but with two different methods, we can set the `method` argument. If, for instance, we've decided we want to use the profile that was generated by fitting the surface brightness profile then we would pass `method='beta'`, where **beta** is the name of the model that was fitted to the profile. We, for no particular reason, are going to use the profile generated from annular spectra:

[14]:
```
cur_dprof = demo_smp[1].get_density_profiles(method='spec')
cur_dprof
```

[14]:
```
<xga.products.profile.GasDensity3D at 0x7f08913a3700>
```

Now that we've successfully retrieved the density profile we want to use, we're going to fit a model to it. So to start with we're going to see what models are allowed for this type of profile:

[15]:
```
cur_dprof.allowed_models('grid')
```

```
+-----------------------+-------------------------------------------------+--------------
↪-------------------------------------+
| MODEL NAME            | EXPECTED PARAMETERS                             | DEFAULT␣
↪START VALUES                         |
+=======================+=================================================+==============================
| simple_vikhlinin_dens | beta, r_core, alpha, r_s, epsilon, norm         | 1.0, 100.0␣
↪kpc, 1.0, 300.0 kpc, 2.0,            |
|                       |                                                 |␣
↪10000000000000.0 solMass / Mpc3                |
+-----------------------+-------------------------------------------------+--------------
↪-------------------------------------+
| king                  | beta, r_core, norm                              | 1.0, 100.0␣
↪kpc, 10000000000000.0 solMass / Mpc3 |
+-----------------------+-------------------------------------------------+--------------
↪-------------------------------------+
| vikhlinin_dens        | beta_one, r_core_one, alpha, r_s, epsilon,      | 1.0, 100.0␣
↪kpc, 1.0, 300.0 kpc, 2.0,            |
|                       | gamma, norm_one, beta_two, r_core_two,          | 3.0,␣
↪10000000000000.0 solMass / Mpc3,           |
|                       | norm_two                                        | 1.0, 50.0␣
↪kpc, 5000000000000.0 solMass / Mpc3   |
+-----------------------+-------------------------------------------------+--------------
↪-------------------------------------+
```

I'm going to choose to fit a simple Vikhlinin density model, which I've imported at the top of this tutorial so I can call the `info()` method to give you some extra details about it:

[16]:
```
SimpleVikhlininDensity1D().info('grid')
```

```
+--------------------------------+-------------------------------------------------------------
↪-------------------+
| Simplified Vikhlinin Profile   |                                                             ␣
↪                   |
```

(continues on next page)

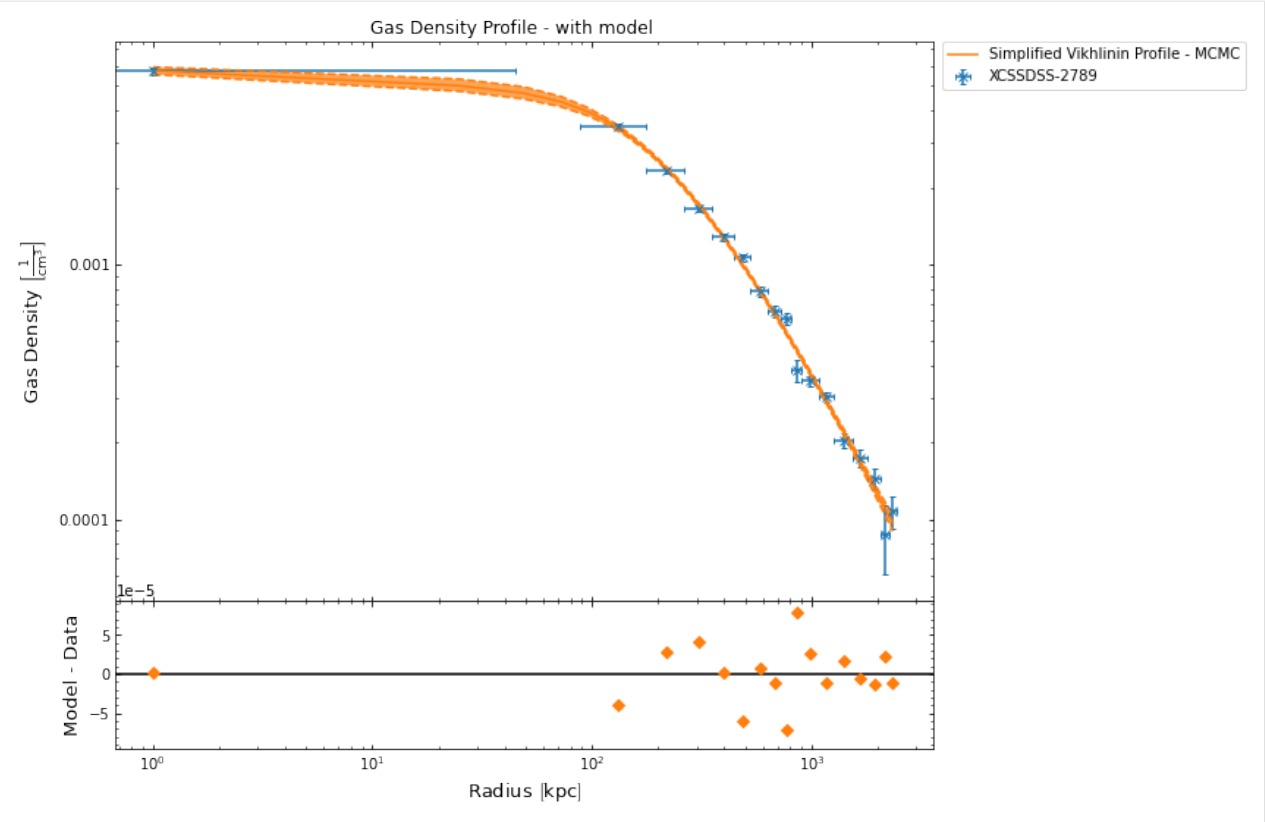
(continued from previous page)

```
+================================+=============================================================
| DESCRIBES                      | Gas Density                                                 ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| UNIT                           | solMass / Mpc3                                              ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| PARAMETERS                     | beta, r_core, alpha, r_s, epsilon, norm                     ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| PARAMETER UNITS                | , kpc, , kpc, , solMass / Mpc3                              ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| AUTHOR                         | Ghirardini et al.                                           ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| YEAR                           | 2019                                                        ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| PAPER                          | https://doi.org/10.1051/0004-6361/201833325                 ␣
↪                    |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
| INFO                           | A simplified form of Vikhlinin's full density␣
↪model, a type of broken |
|                                |  power law that deals well with most galaxy␣
↪cluster density profile.  |
+--------------------------------+-------------------------------------------------------------
↪--------------------+
```

Now we'll run the actual fit using MCMC, then view the model:

[17]:
```
fit_mod = cur_dprof.fit('simple_vikhlinin_dens', num_steps=50000, method='mcmc', show_
↪warn=False)
cur_dprof.view()
```

```
100%|| 50000/50000 [02:09<00:00, 387.30it/s]
```



Now that we have successfully fitted a model to the chosen density profile, we can very easily measure a gas mass by calling the `gas_mass()` method that has been built into the gas density profile class. All we need to tell it is the name of the model which we fitted (this has to be specified as fitting multiple models to a single profile is supported) and the radius within which we would like to know the gas mass. In this case I have chosen to calculate the gas mass within the R_{500} of this cluster.

Two objects are returned by this method, the first is the median gas mass, with - and + uncertainties at the 68.2% confidence level (~1:math:*sigma*), and the second is the entire gas mass distribution calculated by the method. If you didn't wish to use that particular confidence level, you could use the `conf_level` keyword argument to change it:

[18]:
```
gmass, gmass_dist = cur_dprof.gas_mass('simple_vikhlinin_dens', demo_smp[1].r500)
gmass
```

[18]: $[4.3807353 \times 10^{13},\ 6.5393064 \times 10^{11},\ 6.7858617 \times 10^{11}]\ \mathrm{M}_{\odot}$

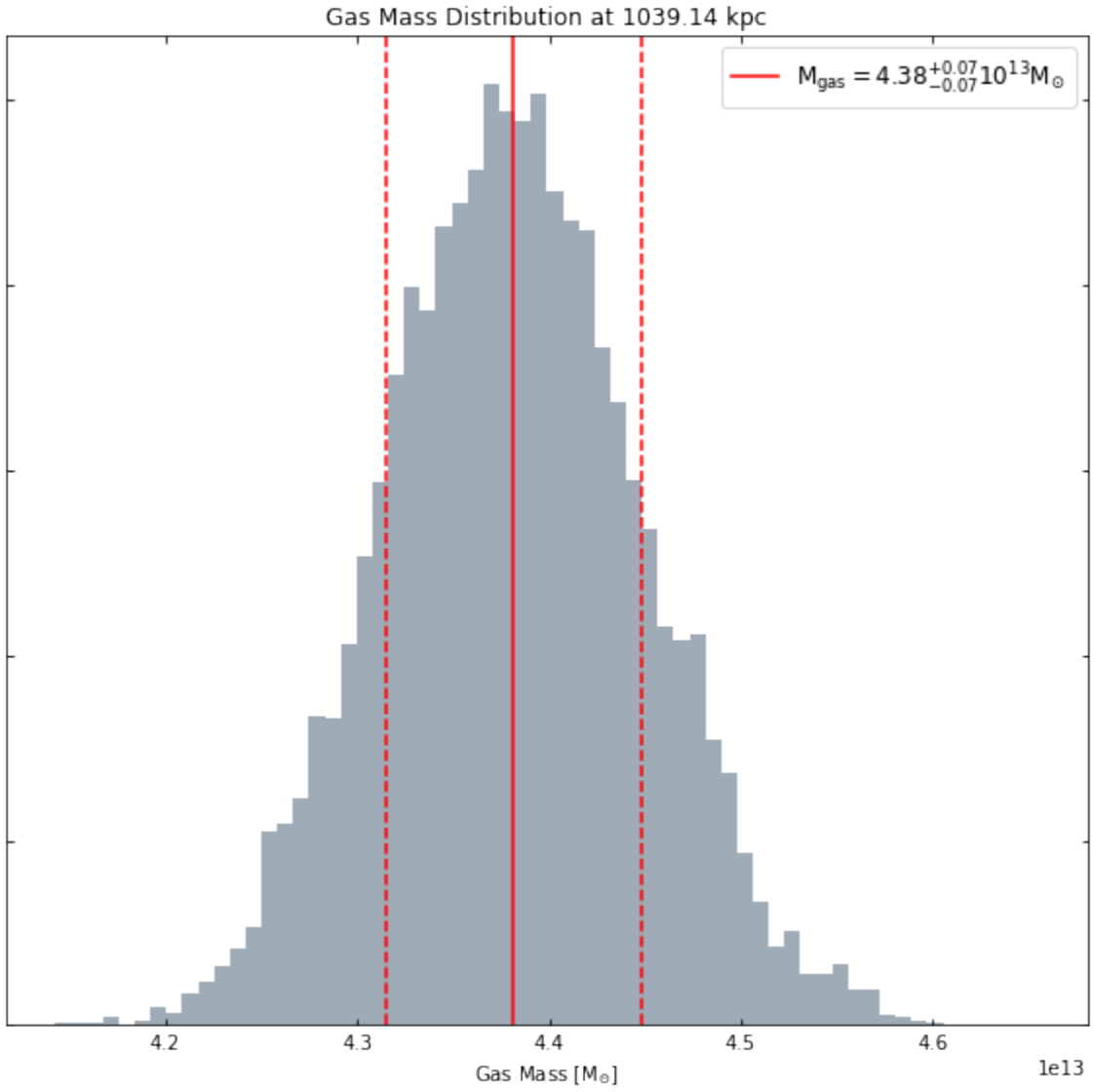
[19]:
```
gmass_dist
```

[19]: $[4.3468629 \times 10^{13},\ 4.4912317 \times 10^{13},\ 4.3363341 \times 10^{13},\ \ldots,\ 4.3413252 \times 10^{13},\ 4.4216002 \times 10^{13},\ 4.3484162 \times 10^{13}]\ \mathrm{M}_{\odot}$

4.2.6 Viewing the gas mass distribution at a given radius

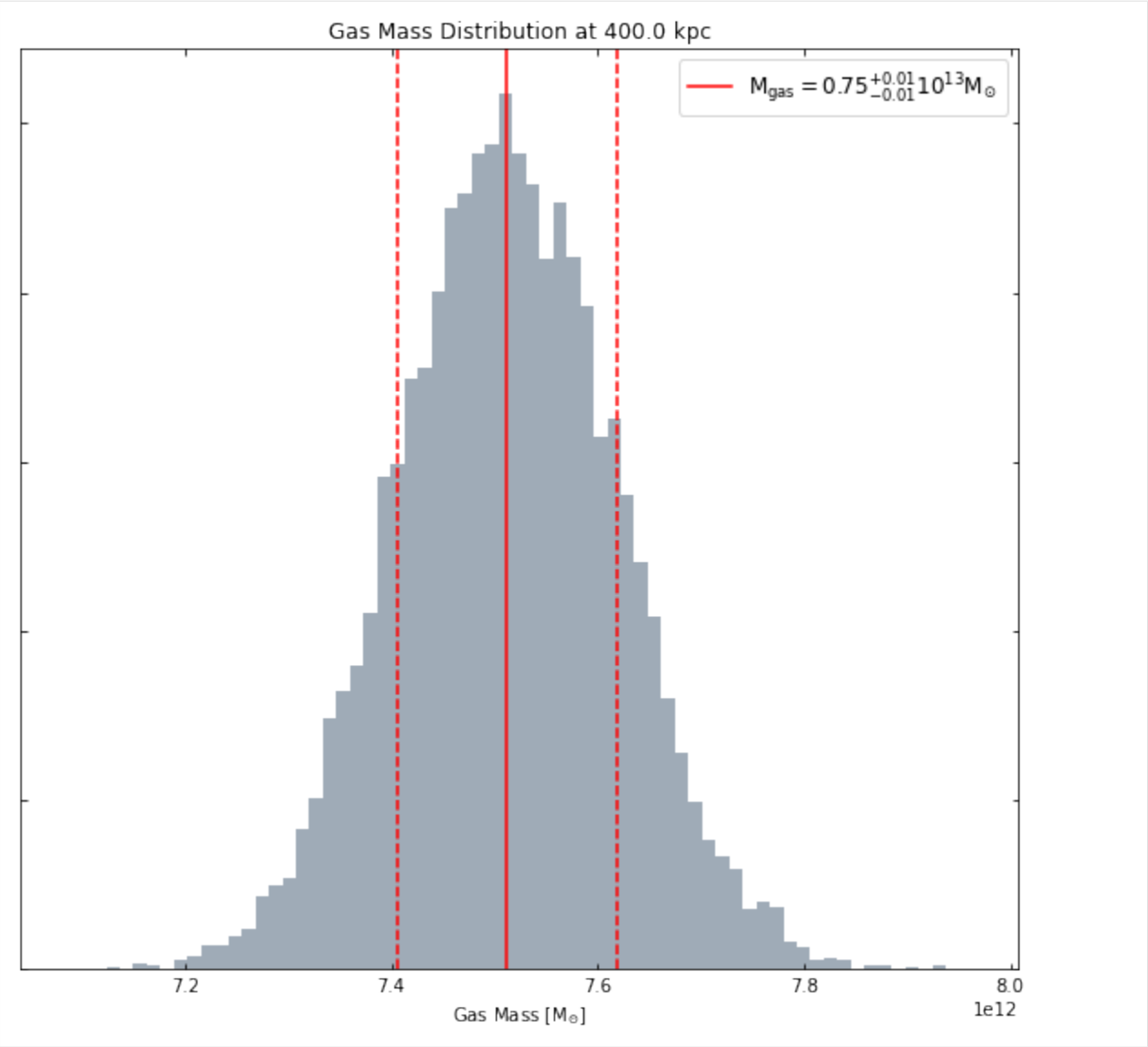
A convenient view method for the gas mass distributions calculated by this object has been implemented, `view_gas_mass_dist()`. It takes the same arguments as `gas_mass()` and simply plots the mass distribution output as a histogram. This seems a good time to point out that any gas mass values and distribution that has been calculated by a density profile (at a specific radius) is stored for later use, so if a gas mass at the same radius (from the same model) is requested again later then the relatively expensive calculated doesn't have to be repeated.

[20]:
```
cur_dprof.view_gas_mass_dist('simple_vikhlinin_dens', demo_smp[1].r500)
```



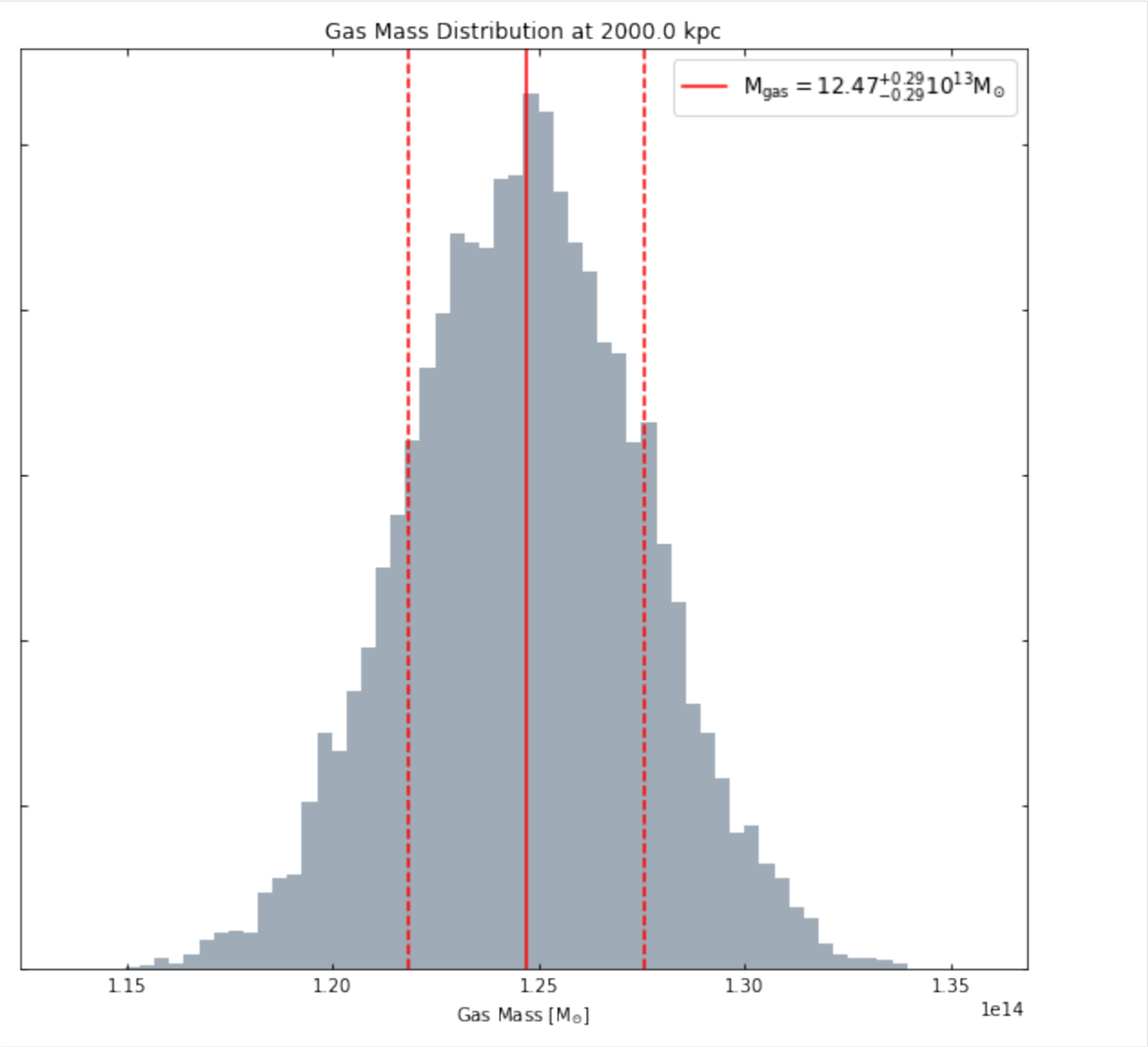
You can request a gas mass within any radius you like:

[21]:
```
cur_dprof.view_gas_mass_dist('simple_vikhlinin_dens', Quantity(400, 'kpc'))
```



Even if that radius is outside the scope of the data which was used to measure the density profile in the first place (though of course you should always be cautious when extrapolating from models):

[22]:
```
cur_dprof.view_gas_mass_dist('simple_vikhlinin_dens', Quantity(2000, 'kpc'))
```



4.2.7 Generating gas mass profiles

As I just demonstrated, we can measure gas masses at whatever radius we like, which of course means that it is possible to create a profile of how the cumulative gas mass changes with radius. This has been implemented as another method of the `GasDensity3D` class, and all you need to supply is the name of the model from which the profile should be generated, and the radii where you wish to measure the values.

Here I choose to use the same radii at which the gas densities were measured (this is the default behaviour when you don't pass custom `radii` and `deg_radii` values), but you could use whatever radii you like:

[23]:
```
gmass_prof = cur_dprof.gas_mass_profile('simple_vikhlinin_dens')
```

```
/home/dt237/code/PycharmProjects/XGA/xga/products/profile.py:463: UserWarning: The␣
↪outer radius you supplied is greater than or equal to the outer radius covered by␣
↪the data, so you are effectively extrapolating using the model.
  warn("The outer radius you supplied is greater than or equal to the outer radius␣
↪covered by the data, so"
```

You have to pass both the radii and the equivalent radii in degrees, as the profile is unable to convert proper radii for itself, as it has no knowledge of the chosen cosmology and the redshift of the source. This was by design, as all XGA products were meant to be able to be defined completely separate from a source object. If you did want to choose custom radii however, you could set them up like this:

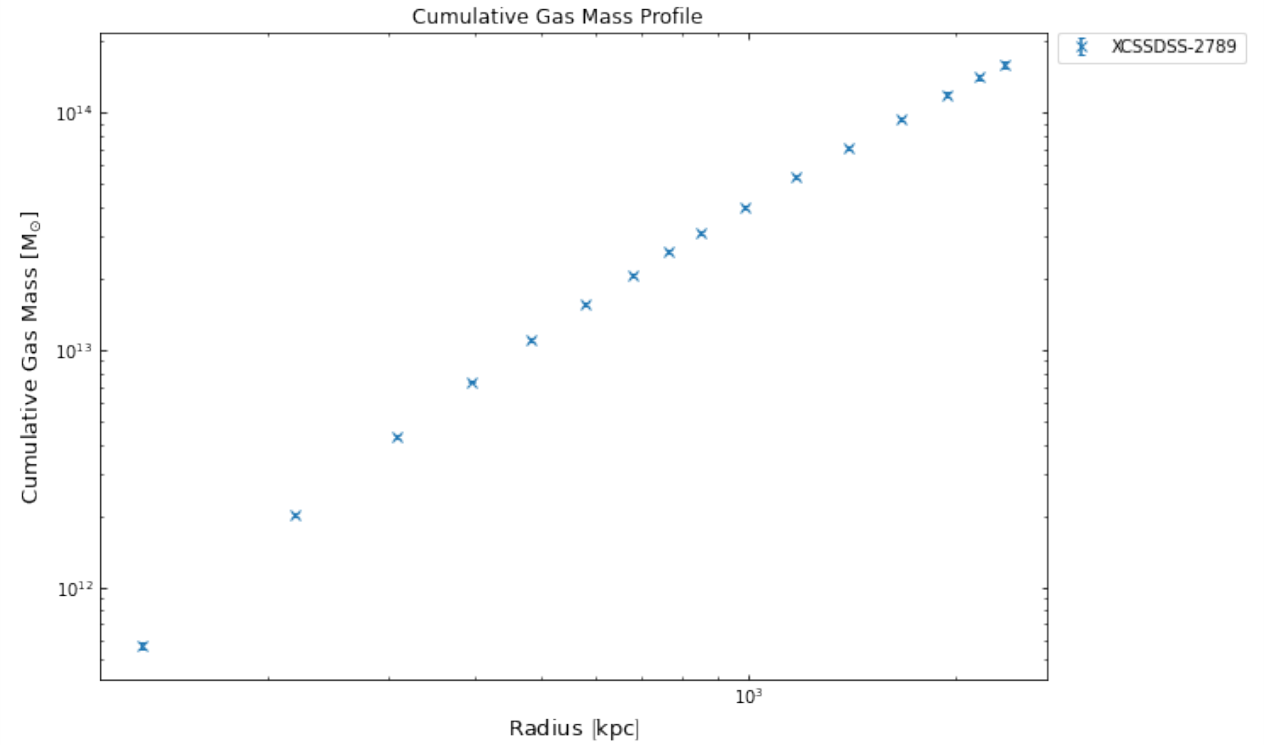
[24]:
```
rads = Quantity([100, 200, 300, 400, 500, 600, 700], 'kpc')
deg_rads = demo_smp[1].convert_radius(rads, 'deg')
deg_rads
```

[24]: $[0.013412552,\ 0.026825104,\ 0.040237656,\ 0.053650209,\ 0.067062761,\ 0.080475313,\ 0.093887865]\ ^\circ$

Here the `convert_radius()` method of the source we generated the density profile from is used to convert the kpc radii to degrees, as it has knowledge of the cluster redshift and the cosmology that we chose to analyse it.

Now we can take a quick look at the gas mass profile we just made:

[25]:
```
gmass_prof.view()
```



4.2.8 Retrieving a sample of gas masses

Finally, I will demonstrate how to retrieve gas masses from a whole sample of density profiles. The `ClusterSample` class has a method called `gas_mass()` built in which will iterate through all the clusters in the sample, attempt to retrieve the specified gas density profile, and then calculate the specified gas mass.

You have to tell the method which density model you want to measure the gas mass for, and in this case I've chosen the King model to demonstrate that if you haven't already fitted a model then it will attempt to do so for you (using the default settings for the `fit()` method of the profile.

[26]:
```
demo_smp.gas_mass('r500', 'king', method='beta')
```

```
100%|| 30000/30000 [01:14<00:00, 403.91it/s]
100%|| 30000/30000 [01:22<00:00, 362.25it/s]
100%|| 30000/30000 [01:11<00:00, 417.61it/s]
100%|| 30000/30000 [01:13<00:00, 408.02it/s]
```

[26]: $[[1.1816284 \times 10^{14},\ 8.825163 \times 10^{10},\ 8.5862588 \times 10^{10}],\ [2.9581939 \times 10^{13},\ 3.9242556 \times 10^{10},\ 4.0931691 \times 10^{10}],\ [4.8668179 \times 10^{13},\ 1.9847375 \times 10^{11},\ 2.0144428 \times 10^{11}],\ [9.6621655 \times 10^{13},\ 1.0204632 \times 10^{11},\ 1.0242727 \times 10^{11}]]\ \mathrm{M}_{\odot}$

4.3 Annular Spectra of Extended Sources

This section aims to run through the basics of how to generate, and interact with, annular spectra of extended sources in XGA. This will include running XSPEC fits on the annuli, as you would for a galaxy cluster were you would like to measure a projected temperature profile, and an apec normalisation profile (which was used in gas density profiles tutorial.

I'll also demonstrate how to use the visualisation abilities which are built in the AnnularSpectra class, how to access results from XSPEC fits for each annulus individually, and how to retrieve annular spectra from where they have been stored in a source object.

[1]:
```
from xga.samples import ClusterSample
from xga.sas import spectrum_set
from xga.xspec import single_temp_apec_profile

from astropy.units import Quantity
import numpy as np
import pandas as pd
```

Yet again we will be using these four clusters from the SDSSRM-XCS sample, the same clusters that were used for the gas density profiles tutorial and the spectroscopy tutorial. There was no particular rationale behind selecting these particular clusters for this demonstration, other than that the observations of them of are a high enough quality that there shouldn't be any problem generating and fitting the spectra.

[2]:
```
# Setting up the column names and numpy array that go into the Pandas dataframe
column_names = ['name', 'ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']
cluster_data = np.array([['XCSSDSS-124', 0.80057775, -6.0918182, 0.251, 1220.11, 1777.
↪06, 109.55, 4.49],
                         ['XCSSDSS-2789', 0.95553986, 2.068019, 0.11, 1039.14, 1519.
↪79, 38.90, 2.83],
                         ['XCSSDSS-290', 2.7226392, 29.161021, 0.338, 935.58, 1359.37,
↪ 105.10, 5.99],
                         ['XCSSDSS-134', 4.9083898, 3.6098177, 0.273, 1157.04, 1684.
↪15, 108.60, 4.79]])

# Possibly I'm overcomplicating this by making it into a dataframe, but it is an␣
↪excellent data structure,
#  and one that is very commonly used in my own analyses.
sample_df = pd.DataFrame(data=cluster_data, columns=column_names)
sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']] = \
    sample_df[['ra', 'dec', 'z', 'r500', 'r200', 'richness', 'richness_err']].
↪astype(float)

# Defining the sample of four XCS-SDSS galaxy clusters
demo_smp = ClusterSample(sample_df["ra"].values, sample_df["dec"].values, sample_df["z
↪"].values,
```

(continues on next page)

(continued from previous page)

```
                          sample_df["name"].values, r200=Quantity(sample_df["r200"].
↪values, "kpc"),
                          r500=Quantity(sample_df["r500"].values, 'kpc'),␣
↪richness=sample_df['richness'].values,
                          richness_err=sample_df['richness_err'].values)
```

```
Declaring BaseSource Sample: 100%|| 4/4 [00:01<00:00,  2.08it/s]
Generating products of type(s) ccf: 100%|| 4/4 [00:30<00:00,  7.53s/it]
Generating products of type(s) image: 100%|| 4/4 [00:02<00:00,  1.82it/s]
Generating products of type(s) expmap: 100%|| 4/4 [00:01<00:00,  2.40it/s]
Setting up Galaxy Clusters: 100%|| 4/4 [00:04<00:00,  1.25s/it]
```

4.3.1 Manually generating sets of annular spectra

There are several functions in XGA designed for generating particular types of profile from a set of annular spectra, and in most cases they will have algorithms designed to automatically decide where they're going to place annuli. However, it is entirely possible that users will want to decide for themselves where exactly the annuli should be placed, and as such I will initially demonstrate how to manually generate annular spectra for our sample.

The `spectrum_set()` function is where XGA generated sets of annular spectra by setting up the annular regions (including the removal of interloper sources), running all the SAS commands, and then combining the resulting files into a single `AnnularSpectra` instance. The full documentation for the function can be found here, but it is quite simple to use, with the most important argument being `radii`, where you specify what annuli should be generated.

When deciding where the annuli should be placed, please bear in mind that the `radii` argument expects information on the **boundaries** of the annuli, so should start at zero if you wish the innermost annulus to be a circle. Here we set up four sets of annuli, one for each cluster:

[3]:
```
ann_rads = [np.linspace(0, 1, 5)*demo_smp[0].r500, np.linspace(0, 1.2, 6)*demo_smp[1].
↪r500,
            Quantity([0, 100, 200, 300, 1200], 'kpc'), np.linspace(0, 1, 6)*demo_
↪smp[3].r500]
ann_rads
```

[3]:
```
[<Quantity [   0.    ,  305.0275,  610.055 ,  915.0825, 1220.11  ] kpc>,
 <Quantity [   0.    ,  249.3936,  498.7872,  748.1808,  997.5744,
            1246.968 ] kpc>,
 <Quantity [   0.,  100.,  200.,  300., 1200.] kpc>,
 <Quantity [   0.    ,  231.408,  462.816,  694.224,  925.632, 1157.04 ] kpc>]
```

These radii were chosen essentially randomly, and have no physical meaning or importance, we merely wish to demonstrate the different ways you might want to define sets of annular boundary radii. Now that we've set those up, all we need to do is pass them into the `spectrum_set()` function, and they will be generated. This function will always generate a spectrum between the first and last entries for each set of radii as well, as it is often convenient to normalise values measured from these spectra by a global value.

The background spectrum defined for the set of annular spectra is generated between `back_inn_rad_factor`*outermost-radius and `back_out_rad_factor`*outermost radius - those factors were set when you initially defined the source or sample:

[4]:
```
spectrum_set(demo_smp, radii=ann_rads)
```

```
Generating products of type(s) spectrum: 100%|| 12/12 [55:25<00:00, 277.13s/it]
Generating products of type(s) annular spectrum set components: 100%|| 54/54 [28:05
↪<00:00, 31.22s/it]
```

```
[4]: <xga.samples.extended.ClusterSample at 0x7f0851661250>
```

4.3.2 XSPEC fit to annular spectra

Now we want to fit plasma emission models to the set of spectra (there will be multiple spectra per annulus if there are multiple observations of the source), just as we fit a model to the global spectra we produced in the spectroscopy tutorial. Again we'll be making use of the XGA XSPEC interface, and the fitting process works in exactly the same way for the constant*tbabs*apec model that describes absorbed plasma emission from a cluster. In this case however, we use the `single_temp_apec_profile()` function (look here for documentation).

To run a default fit with this model to our set of clusters, we just need to pass the same list of annular boundaries into the fitting function (currently the only model that can be used to fit these profiles):

```
[5]: single_temp_apec_profile(demo_smp, radii=ann_rads)
```

```
Running XSPEC Fits: 100%|| 18/18 [00:33<00:00,  1.87s/it]
```

```
[5]: <xga.samples.extended.ClusterSample at 0x7f0851661250>
```

If we hadn't already generated these spectra, this function would actually have run the `spectrum_set()` function for us, then fit them as we requested, but I wished to demonstrate the use of the annular spectra generation function.

4.3.3 Fetching annular spectra from a source

As is the case for most XGA products, there is a specific method to retrieve annular spectra from a source object, in this case the `get_annular_spectra` method (you can find the full documentation here). Since we know a priori the radii we used to generate the annular spectra, all we need to do here is to pass those radii and the correct set of annular spectra will be fetched:

```
[6]: cur_ann_spec = demo_smp[0].get_annular_spectra(radii=ann_rads[0])
     cur_ann_spec
```

```
[6]: <xga.products.spec.AnnularSpectra at 0x7f0851650dc0>
```

Please note that each set of annular spectra is issued a unique 'set id' when it is generated, and if you already knew that identifying number you could pass it to the `get_annular_spectra()` method through the `set_id` argument and retrieve the correct set of annular spectra.

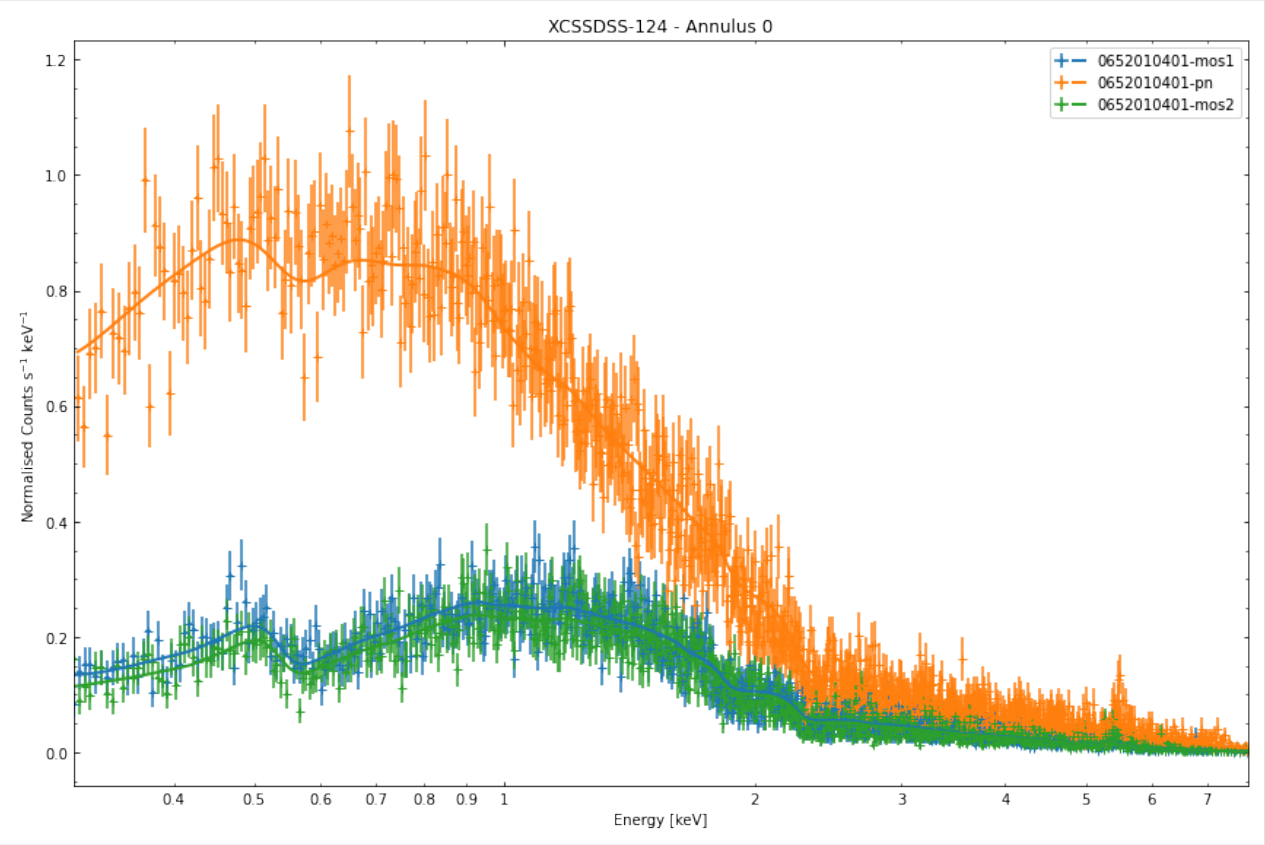
4.3.4 Visualising annular spectra

Now that we've generated, fitted, and retrieved these annular spectra, we might want to visualise the data somehow. Several methods to do just that have been implemented in the `AnnularSpectra` class.

An individual annulus with `view_annulus()`

The first is akin to the `view()` method which is present in the `Spectrum` class, although in this case all spectra for a single annulus are shown. You need to pass the annulus ID (for instance the innermost annulus would be 0, the next 1, etc.), and the model that was fitted (in this case constant*tbabs*apec):

```
[7]: # Just choosing the first annulus for not particular reason
     cur_ann_spec.view_annulus(0, 'constant*tbabs*apec')
```



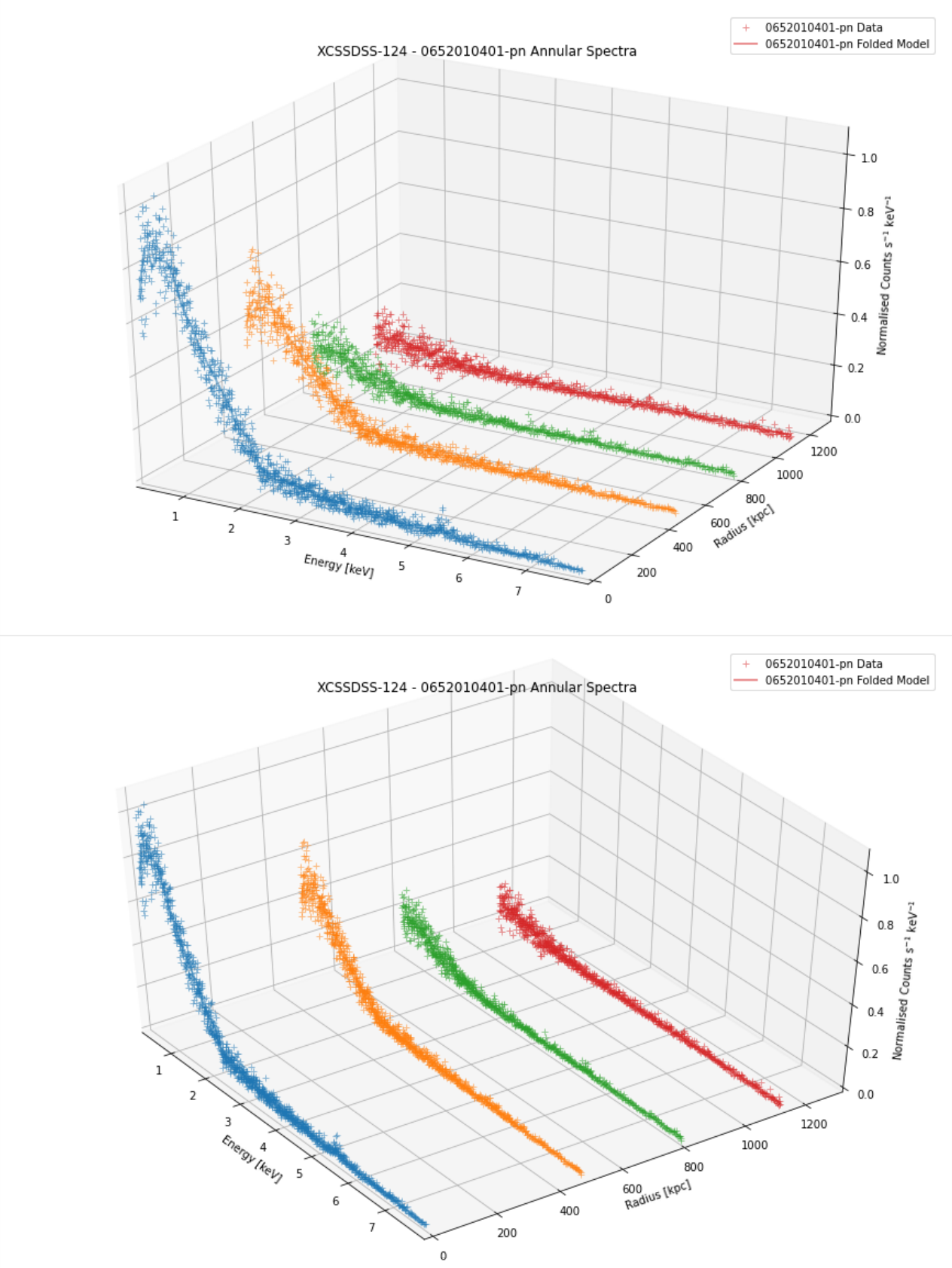
A set of annuli for a particular ObsID-Instrument with `view_annuli()`

The next method lets you see a whole set of annular spectra, for a single ObsID-Instrument combination. The `view_annuli()` method requires a little more information, as you have to pass an ObsID and instrument as well as the model that was fitted. This visualisation is in 3D, so you can also set the `elevation_angle` and `azimuthal_angle` to change perspective. I also use the `instruments` property that each source object has to remind myself which ObsIDs and instruments are associated with this source:

[8]:
```
demo_smp[0].instruments
```

[8]:
```
{'0652010401': ['pn', 'mos1', 'mos2']}
```

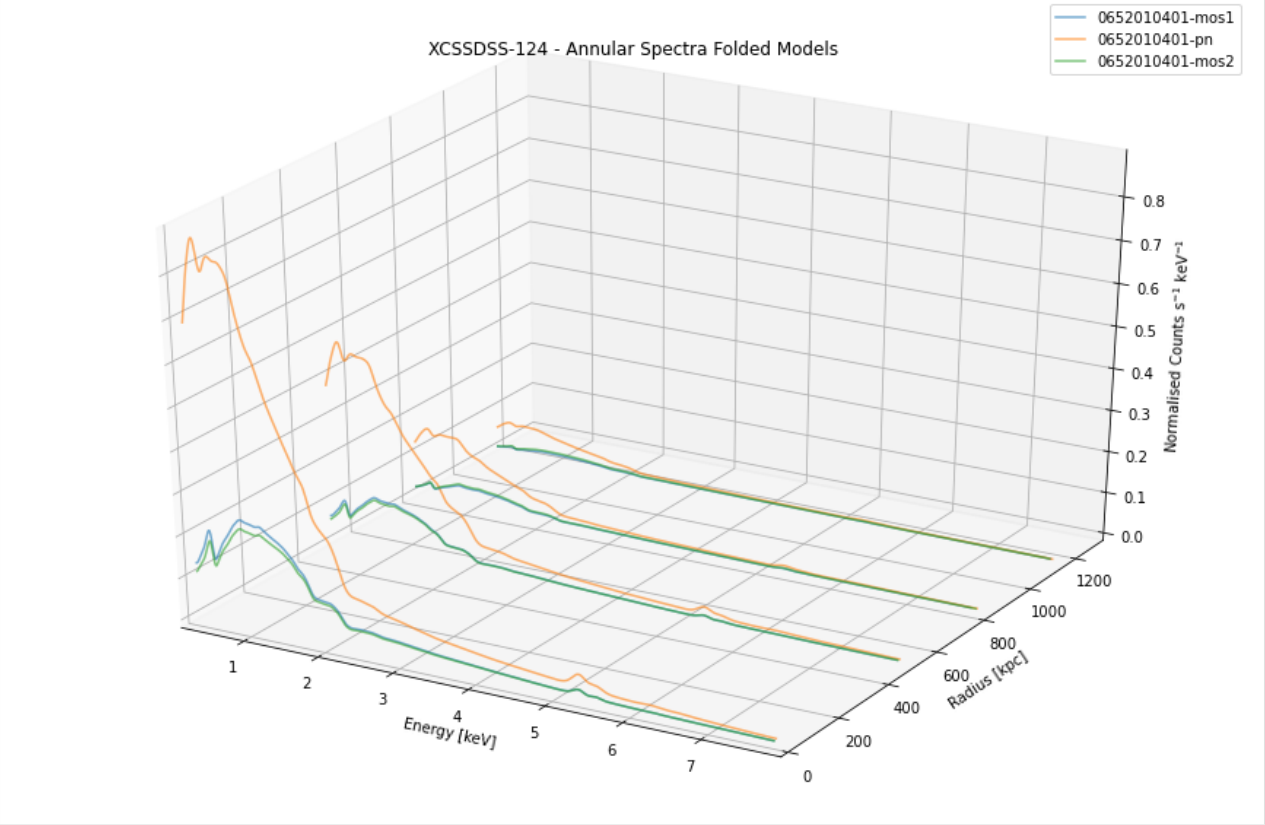
[9]:
```
cur_ann_spec.view_annuli('0652010401', 'pn', 'constant*tbabs*apec')

# Changing the perspective, just a demonstration I'm not saying this is more useful
cur_ann_spec.view_annuli('0652010401', 'pn', 'constant*tbabs*apec', elevation_
↪angle=45, azimuthal_angle=-35)
```


0652010401-pn Data
0652010401-pn Folded Model
XCSSDSS-124 - 0652010401-pn Annular Spectra
Energy [keV]
Radius [kpc]
Normalised Counts s⁻¹ keV⁻¹
0652010401-pn Data
0652010401-pn Folded Model
XCSSDSS-124 - 0652010401-pn Annular Spectra
Energy [keV]
Radius [kpc]
Normalised Counts s⁻¹ keV⁻¹

All fitted models for all annuli with `view()`

Another three dimensional view method, but rather than a set of annular spectra for a specific ObsID-instrument combination, this will show the fitted models for **all** ObsID-instrument combinations associated with this `AnnularSpectra` instance. No data points are plotted here because it makes the figure far too confusing. The same angle arguments can be passed to change the perspective of the plot:

[10]:
```
cur_ann_spec.view('constant*tbabs*apec')
```



4.3.5 Retrieving a spectrum from an `AnnularSpectra` instance

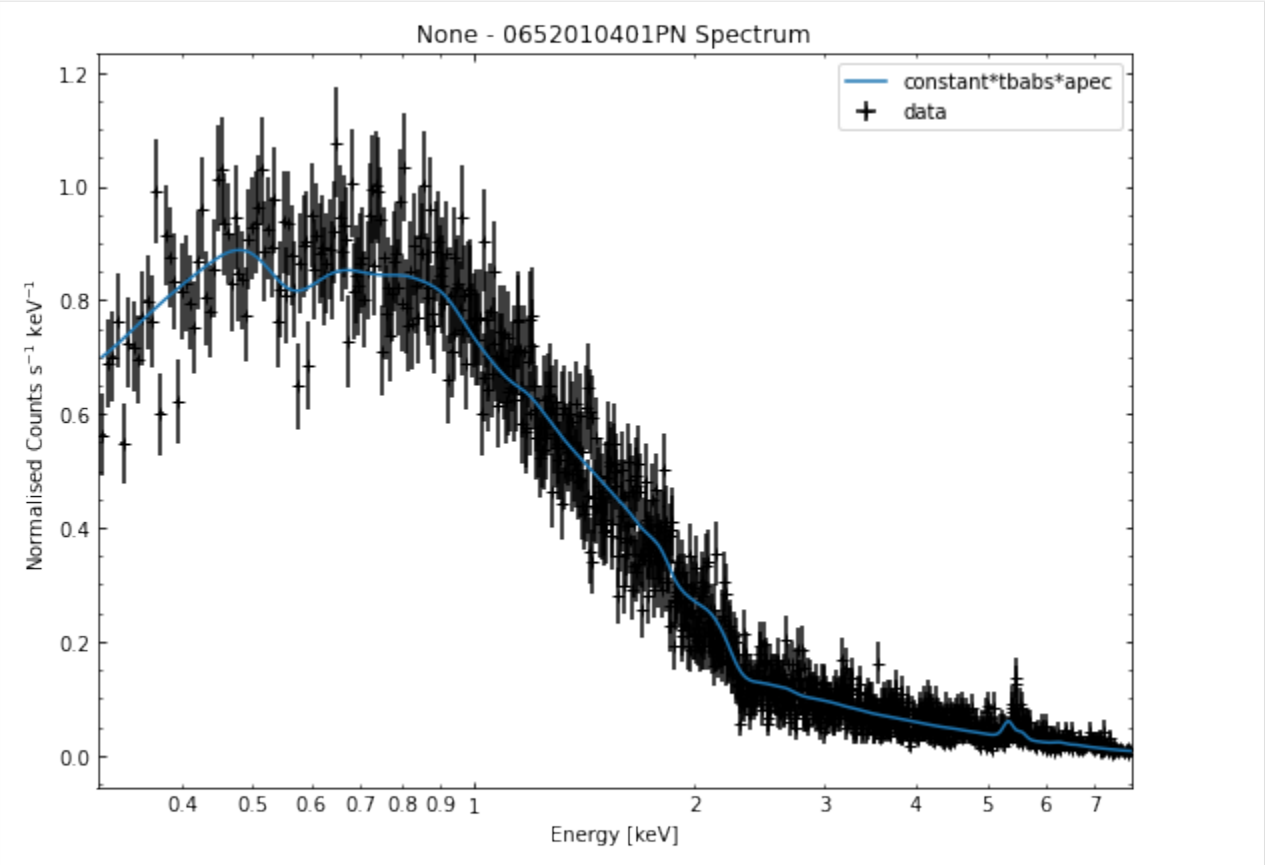
An `AnnularSpectra` instance is what is termed an 'aggregate product' in XGA; that is to say that its actually a container for multiple individual product instances (in this case `Spectrum` instances). That means that those individual instances (with all their built in features and information) are stored within the aggregate product, and can be retrieved with a convenient get method (`get_spectra()` in this case).

So if we decided we wanted to retrieve the 0652010401-pn spectrum for the innermost annulus, we would run this command:

[11]:
```
ind_spec = cur_ann_spec.get_spectra(0, '0652010401', 'pn')
```

Then we could decide to view that spectrum, if we wanted:

[12]:
```
ind_spec.view()
```



We aren't limited to retrieving one spectrum at a time however, if we wanted to we could grab a list of all the `Spectrum` objects that make up the first annulus:

[13]:
```
cur_ann_spec.get_spectra(0)
```

[13]:
```
[<xga.products.spec.Spectrum at 0x7f08568e8ca0>,
 <xga.products.spec.Spectrum at 0x7f08568e8e20>,
 <xga.products.spec.Spectrum at 0x7f08568e8880>]
```

If, for some reason, you wanted to do the same thing but for **all** spectra in the `AnnularSpectra` object, you could just use the `all_spectra` property:

[14]:
```
cur_ann_spec.all_spectra
```

[14]:
```
[<xga.products.spec.Spectrum at 0x7f08568e8ca0>,
 <xga.products.spec.Spectrum at 0x7f08568e8e20>,
 <xga.products.spec.Spectrum at 0x7f08568e8880>,
 <xga.products.spec.Spectrum at 0x7f08568e8850>,
 <xga.products.spec.Spectrum at 0x7f08568e8760>,
 <xga.products.spec.Spectrum at 0x7f08568c7580>,
 <xga.products.spec.Spectrum at 0x7f08568e8be0>,
 <xga.products.spec.Spectrum at 0x7f08568e8340>,
 <xga.products.spec.Spectrum at 0x7f085690d0a0>,
 <xga.products.spec.Spectrum at 0x7f08568c5040>,
 <xga.products.spec.Spectrum at 0x7f085690dd60>,
 <xga.products.spec.Spectrum at 0x7f08568e8220>]
```

4.3.6 Retrieving fit results and luminosities

Retrieving fit results is as simple as calling another get method, `get_results`. You must specify which annulus you want to get the results for, as well as the model that was fitted, and optionally you can specify the name of the parameter that you particularly want to grab:

[15]:
```
cur_ann_spec.get_results(0, 'constant*tbabs*apec')
```

[15]:
```
{'kT': array([7.41271   , 0.12935868, 0.1301684 ]),
 'norm': array([4.2421700e-03, 3.3772432e-05, 3.4043752e-05]),
 'factor': array([[0.958273  , 0.00932076, 0.00942246],
        [1.0042    , 0.01143088, 0.01155637]])}
```

[16]:
```
cur_ann_spec.get_results(3, 'constant*tbabs*apec', 'norm')
```

[16]:
```
array([3.66600000e-04, 2.16666992e-05, 2.19063438e-05])
```

The behaviour of the method with which you fetch luminosities (`get_luminosities`) is much the same, but instead of optionally specifying a parameter name you can pass `lo_en` and `hi_en` values to get back a luminosity measured within specific energy limits:

[17]:
```
cur_ann_spec.get_luminosities(0, 'constant*tbabs*apec')
```

[17]:
```
{'bound_0.5-2.0': <Quantity [2.95390183e+44, 1.75799566e+42, 1.42047898e+42] erg / s>,
 'bound_0.01-100.0': <Quantity [1.16941378e+45, 1.26719656e+43, 1.23961762e+43] erg /␣
↪s>}
```

[18]:
```
cur_ann_spec.get_luminosities(0, 'constant*tbabs*apec', lo_en=Quantity(0.5, 'keV'),␣
↪hi_en=Quantity(2.0, 'keV'))
```

[18]: $[2.9539018 \times 10^{44},\ 1.7579957 \times 10^{42},\ 1.420479 \times 10^{42}]\ \frac{\mathrm{erg}}{\mathrm{s}}$

4.3.7 Other useful information about `AnnularSpectra`

Here I will quickly run through other useful pieces of information which are stored in an AnnularSpectra object. The first piece of information that can be handy is the number of annuli in a given spectrum, for which you can just use the standard Python `len()` function (or the `num_annuli` property):

[19]:
```
print(len(cur_ann_spec))
print(cur_ann_spec.num_annuli)
```

```
4
4
```

There is also a property that will return the set id that, as I mentioned earlier, is a unique identifier that can be used to retrieve the `AnnularSpectra` object:

[20]:
```
cur_ann_spec.set_ident
```

[20]:
```
1295248
```

The boundary radii which were used to generated the set of annular spectra initially are given by the `radii` property:

[21]:
```
cur_ann_spec.radii
```

[21]: $[0,\ 0.020943087,\ 0.041886174,\ 0.062829261,\ 0.083772348]\ ^{\circ}$

And finally, the `obs_ids` property gives a list of all the ObsIDs that went into the `AnnularSpectra` (it is essentially the same as the `obs_ids` property of `BaseSource`), and `AnnularSpectra` has an `instruments` property that performs the same function as the `instruments` property of `BaseSource`:

[22]:
```
cur_ann_spec.obs_ids
```

[22]:
```
['0652010401']
```

[23]:
```
cur_ann_spec.instruments
```

[23]:
```
{'0652010401': ['mos1', 'pn', 'mos2']}
```

4.4 Three-dimensional temperature and mass profiles of galaxy clusters

We have seen in the gas density profiles tutorial that XGA has the capacity to measure the gas density profile of a cluster, and in the annular spectra tutorial we have seen that XGA can make (and fit models to) sets of annular spectra. That means that we can measure a projected temperature profile for a galaxy cluster

This tutorial will demonstrate how to go from a set of annular spectra to a projected temperature profile, from there to a three-dimensional temperature profile, and then how to combine that with a gas density profile to measure the hydrostatic mass of a galaxy cluster. We will detail the 'manual' way of generating a hydrostatic mass profile, and then demonstrate the convenience function that has been implemented to do the same job.

[1]:
```
from xga.sources import GalaxyCluster
from xga.sourcetools.temperature import min_snr_proj_temp_prof, onion_deproj_temp_prof
from xga.sourcetools.density import ann_spectra_apec_norm
from xga.products.profile import HydrostaticMass

from astropy.units import Quantity, Unit
import numpy as np
```

I'm setting up a single galaxy cluster to (eventually) measure a hydrostatic mass for, though please bear in mind that the R_{500}, R_{200}, and redshift are approximate and shouldn't be used for real scientific analysis:

[2]:
```
demo_src = GalaxyCluster(149.59209, -11.05972, 0.16, r500=Quantity(1200, 'kpc'),␣
↪r200=Quantity(1700, 'kpc'),
                         name="A907")
```

4.4.1 Convenience function for creating a projected temperature profile

The annular spectra tutorial has already shown you how to generate and fit a set of annular spectra, so we won't go through that whole process again here. Instead I will demonstrate a convenient way to go through the steps necessary to make a projected temperature profile by calling a single function `min_snr_proj_temp_prof`, for which you can find documentation here.

This function will decide for itself how large the annuli should be, based on both a minimum signal to noise and a minimum annulus size. We also have to decide what the outer radius is going to be, and bear in mind that algorithm used to make the annuli alter the outer radius slightly so that it is an integer multiple of the minimum width, though it will never make the outer radius smaller, only larger. In this tutorial we are only using a single cluster, but this function is also compatible with a cluster sample. We're going to set `min_snr=35` (this is something you may have to experiment with), leave the minimum width set to 20 arcseconds, and generate the profile out to $1.3R_{500}$:

[3]:
```
min_snr_proj_temp_prof(demo_src, min_snr=35, outer_radii=demo_src.r500*1.3)
```

```
Generating products of type(s) spectrum: 100%|| 9/9 [41:09<00:00, 274.42s/it]
Generating products of type(s) annular spectrum set components: 100%|| 117/117 [26:58
↪<00:00, 13.83s/it]
Running XSPEC Fits: 100%|| 13/13 [04:17<00:00, 19.82s/it]
```

[3]:
```
[<Quantity [  0.        ,  21.74999258,  43.49998517,  65.24997775,
             86.99997034, 108.74996292, 130.49995551, 152.24994809,
            173.99994068, 239.24991843, 304.49989619, 391.49986653,
            478.49983687, 548.09981314] arcsec>]
```

4.4.2 Retrieving a projected temperature profile from a source

There is a specific get method designed to retrieve *projected* temperature profiles that have been stored in a source object, `get_proj_temp_profiles`. As with most of the other get methods implemented in XGA, if you have only made one of these profiles then you can call the method with no arguments and it will be returned. Otherwise it asks you to pass the annular radii that were used to generate the profile, or the `set_id` that was assigned to the `AnnularSpectra` from which the profile was made:

[4]:
```
proj_temp = demo_src.get_proj_temp_profiles()
proj_temp
```

[4]:
```
<xga.products.profile.ProjectedGasTemperature1D at 0x7f73ba1d2eb0>
```

4.4.3 Generating profiles from a fitted `AnnularSpectra`

While we have already generated a projected temperature profile using the handy `min_snr_proj_temp_prof` function, we also wanted to demonstrate how you could manually create such a profile from a fitted `AnnularSpectra`. For this we first have to retrieve the object using the get method demonstrated in the annular spectra tutorial:

[5]:
```
ann_spec = demo_src.get_annular_spectra()
```

Once we've read it out into a variable, we can call the `generate_profile()` method to tell it to assemble a profile out of a specific parameter of the fit. When it comes to temperature profiles this is rarely going to be necessary, as the temperature profile is automatically generated after the fitting process is complete, but we demonstrate this for completeness' sake:

[6]:
```
manual_proj_temp = ann_spec.generate_profile('constant*tbabs*apec', 'kT', 'keV')
manual_proj_temp
```

[6]:
```
<xga.products.profile.ProjectedGasTemperature1D at 0x7f73ba33ac70>
```

4.4.4 Creating a deprojected temperature profile

'Deprojection' is a process that takes us from a temperature profile that is the result of the emission of the 3D shells of a cluster being 'projected' along the line of site onto the plane that we observe with the telescope. Temperatures measured in an these annulus are a combination of the temperatures of all the 3D shells of cluster that intersect the annulus when that annulus is extended back through the cluster.

There are different methods of dealing with this, but in XGA the 'onion peeling' deprojection method has been implemented in the `onion_deproj_temp_prof()` function. The function is an implementation of a fairly old

technique, though it has been used recently in this paper. For a more in depth discussion of this technique and its uses I would currently recommend this paper.

We need to call `onion_deproj_temp_prof()` to generate our desired deprojected profile from the projected profile we have already measured, although it is important to note that if we hadn't already made that profile then this function would have done that for us (including the generation and fitting of annular spectra). As such for a 'real-world' use case you should just call `onion_deproj_temp_prof()` and let it do everything. So now we call this function with the same arguments we used earlier, and it will automatically fetch the profile we generated, then deproject it:

[7]:
```
deproj_temp = onion_deproj_temp_prof(demo_src, min_snr=35, outer_radii=demo_src.
↪r500*1.2)[0]
```

```
Generating products of type(s) spectrum: 100%|| 9/9 [32:30<00:00, 216.69s/it]
Generating products of type(s) annular spectrum set components: 100%|| 108/108 [25:21
↪<00:00, 14.09s/it]
Running XSPEC Fits: 100%|| 12/12 [03:04<00:00, 15.41s/it]
```

I do not give an explanation of how this deprojection method works here, but you can find a detailed walk-through in the 'Under the Hood' section.

4.4.5 Retrieving a deprojected temperature profile from a source

The `get_3d_temp_profiles()` method of can be used to easily retrieve a deprojected temperature profile, behaving the same way as the `get_proj_temp_profiles()`:

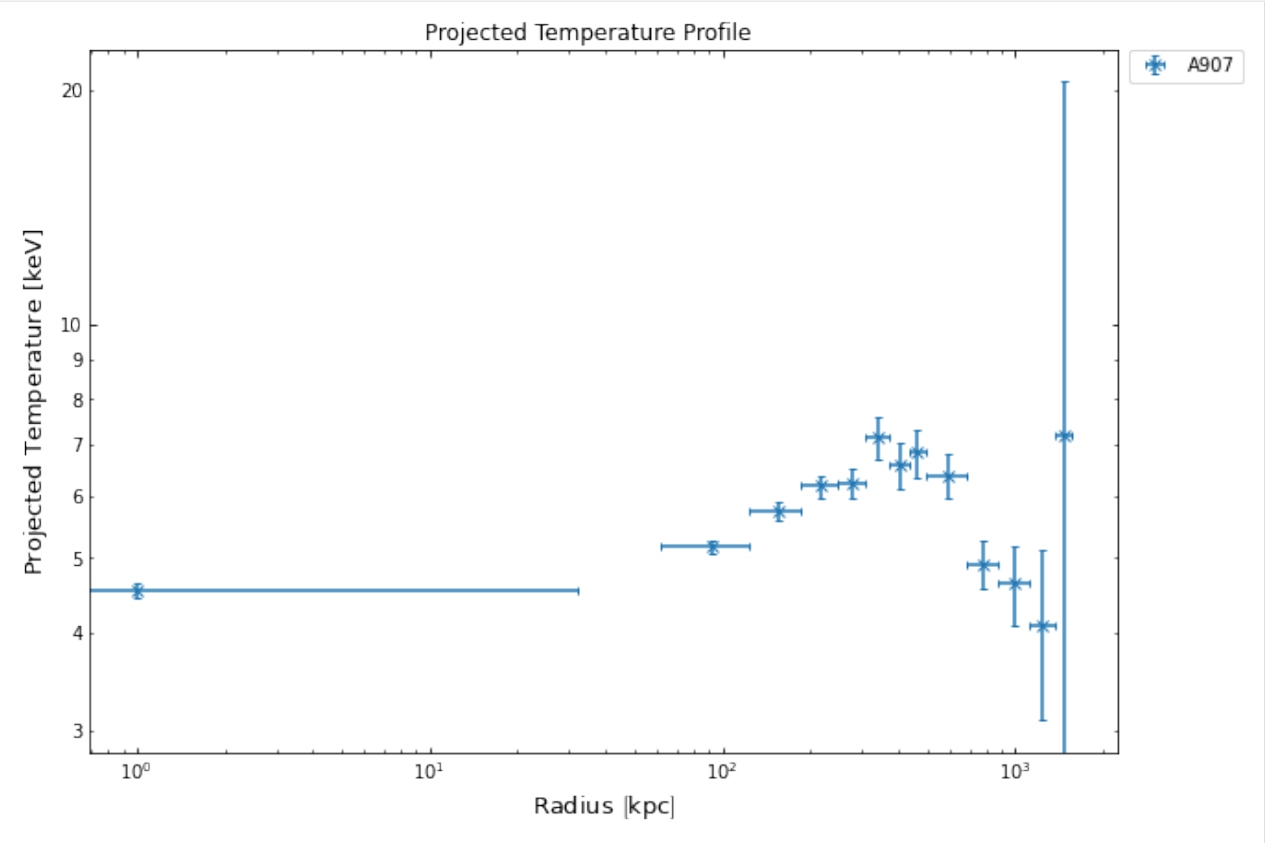
[8]:
```
demo_src.get_3d_temp_profiles()
```

[8]:
```
<xga.products.profile.GasTemperature3D at 0x7f73ba08deb0>
```

4.4.6 Viewing the profiles

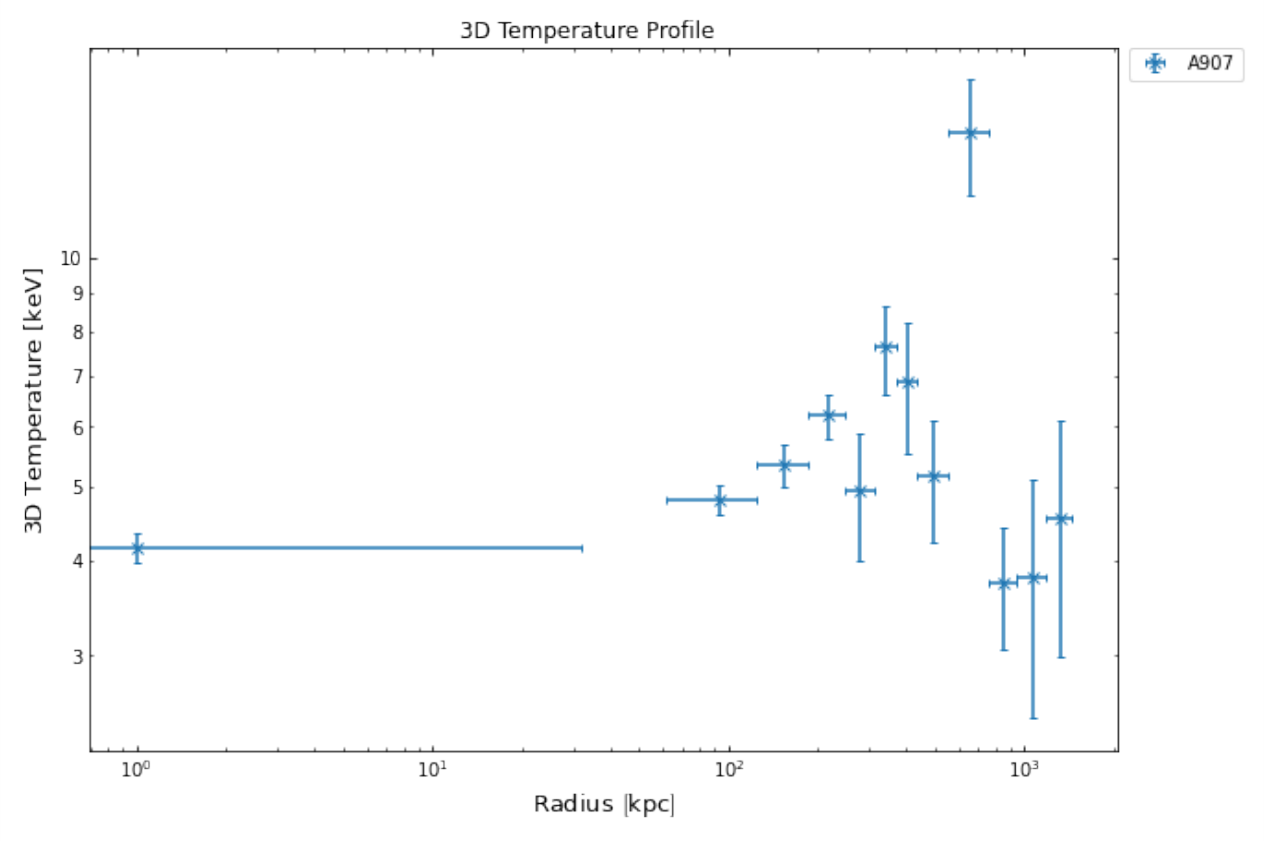
We're quickly going to look at the profiles we've just generated, using the profile view method that was introduced in the profile tutorial. First the original, projected, profile:

[9]:
```
proj_temp.view()
```



Then the newly deprojected temperature profile, which will help us measure the mass of cluster:

[10]:
```
deproj_temp.view()
```



4.4.7 Creating a hydrostatic mass profile

Here we will show you how to create a hydrostatic mass profile from a density and temperature profile, though we will not be explaining how hydrostatic masses work specifically as that is the subject of a another detailed walk-through in the 'Under the Hood' section.

We will define a mass profile manually here, but will also touch on a convenience function that can perform this whole process in a single line.

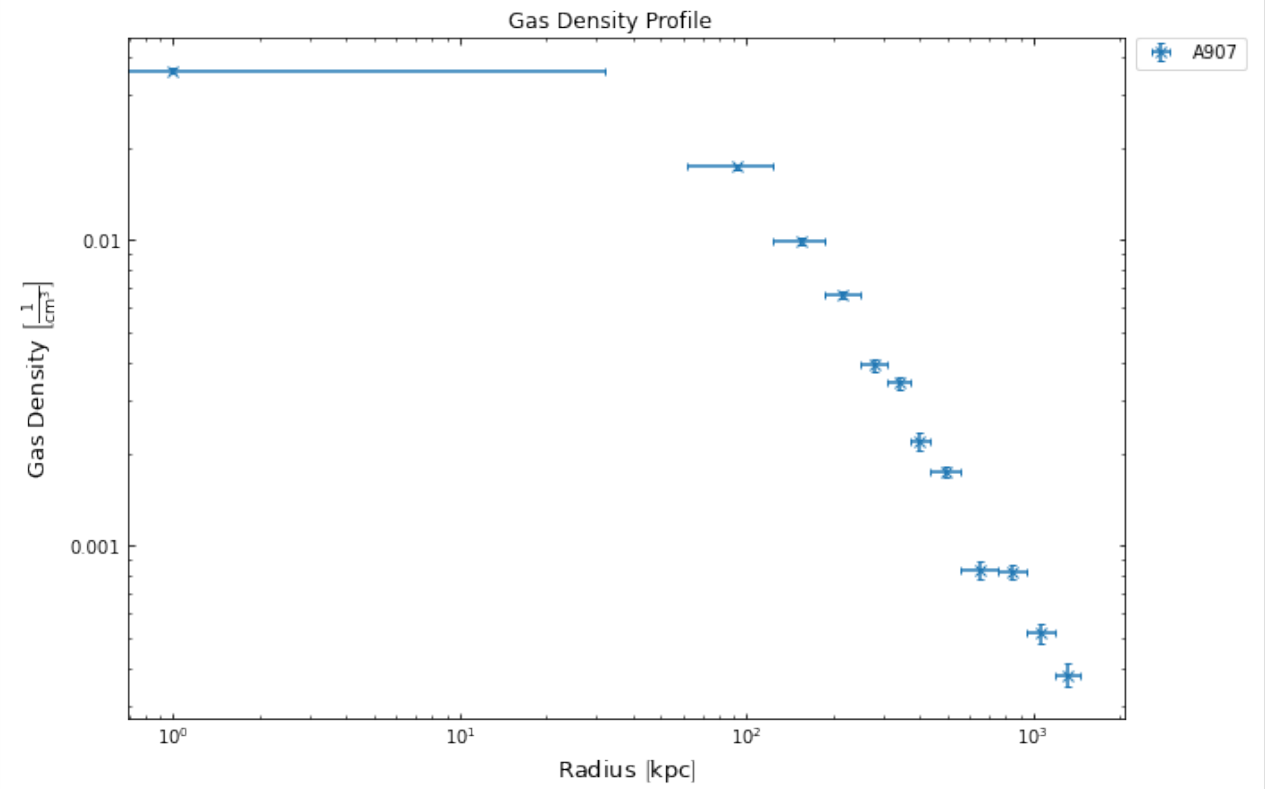
Measuring hydrostatic masses of galaxy clusters in XGA is entirely based around the `HydrostaticMass` profile class, there is no way to measure a mass without one. This profile class is a special case as you do not pass the values on declaration, but pass two other profiles (deprojected temperature and three-dimensional gas density) and allow the profile class itself to do the calculation.

Speaking of which, we don't yet have the gas density profile which we need to declare the `HydrostaticMass` instance. As we've already gone to the trouble of running the generation and fitting of an `AnnularSpectra`, we're just going to run `ann_spectra_apec_norm()` (which was introduced in the gas density profiles tutorial), with the same `min_snr` and `outer_radii` values. This will use the normalisation profile from the XSPEC model fits to measure a density profile:

[11]:
```
dens_prof = ann_spectra_apec_norm(demo_src, min_snr=35, outer_radii=demo_src.r500*1.
↪2)[0]
```

```
/home/dt237/code/PycharmProjects/XGA/xga/xspec/run.py:186: UserWarning: All XSPEC␣
↪operations had already been run.
  warnings.warn("All XSPEC operations had already been run.")
Generating density profiles from annular spectra: 100%|| 1/1 [00:06<00:00,  6.29s/it]
```

[12]:
```
dens_prof.view()
```



The `HydrostaticMass` class requires both the input profiles to have model fits, and if they haven't been run prior to mass profile being declared then it will do it for you. As such you also have to provide the model names (or declared model instances) when you declare the mass profile. As we have not fitted a deprojected temperature profile before in these tutorials, we can quickly run the `allowed_models()` method to check what models are available:

[13]:
```
deproj_temp.allowed_models('grid')
```

```
+----------------------+-----------------------------------------+------------------
↪------------------------------+
| MODEL NAME           | EXPECTED PARAMETERS                     | DEFAULT START␣
↪VALUES                          |
+======================+=========================================+==================================
| vikhlinin_temp       | r_cool, a_cool, t_min, t_zero, r_tran,  | 100.0 kpc, 1.0,␣
↪1.0 keV, 5.0 keV, 400.0 kpc,    |
|                      | a_power, b_power, c_power               | 1.0, 1.0, 1.0    ␣
↪                                |
+----------------------+-----------------------------------------+------------------
↪------------------------------+
| simple_vikhlinin_temp | r_cool, a_cool, t_min, t_zero, r_tran, | 100.0 kpc, 1.0,␣
↪1.0 keV, 5.0 keV, 400.0 kpc,    |
|                      | c_power                                 | 1.0              ␣
↪                                |
+----------------------+-----------------------------------------+------------------
↪------------------------------+
```

We're going to set `temperature_model='simple_vikhlinin_temp'` and `density_model='simple_vikhlinin_dens'`, change the `num_steps` value to 40000, and decide on which radii we should use. As the mass profile is generated from mathematics performed on continuous models,

you could decide to set the radii to any values (within reason) and this would work. We, however, are going to use the radii at which the temperature values were measured with the AnnularSpectra:

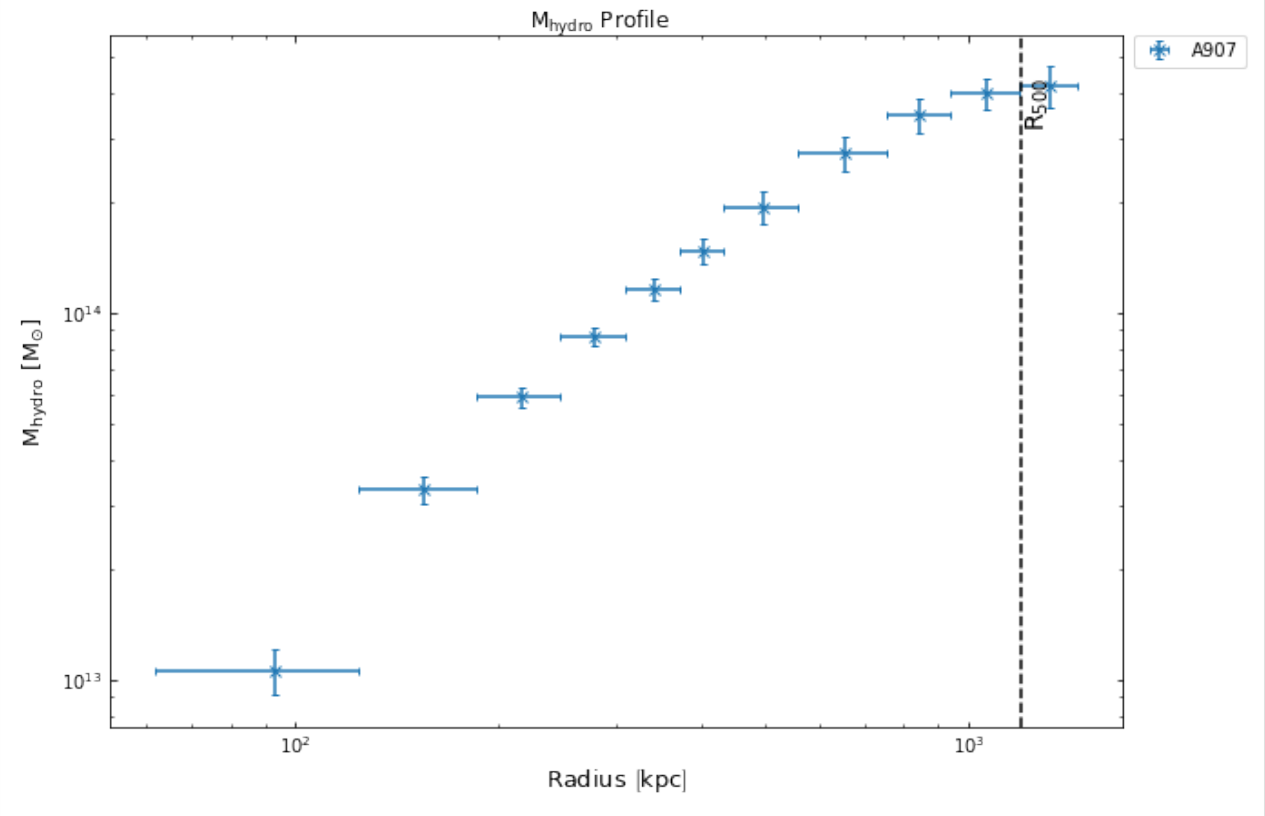
[14]:
```
hym_prof = HydrostaticMass(deproj_temp, 'simple_vikhlinin_temp', dens_prof, 'simple_
↪vikhlinin_dens',
                           deproj_temp.radii[1:], deproj_temp.radii_err[1:], deproj_
↪temp.deg_radii[1:],
                           num_steps=40000)
```

```
100%|| 40000/40000 [01:32<00:00, 432.27it/s]
100%|| 40000/40000 [01:31<00:00, 435.21it/s]
```

```
The chain is shorter than 50 times the integrated autocorrelation time for 6␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 800;
tau: [1886.81022727 1984.87460631 1483.3064957  1619.67813139 2676.38323939
 2377.75272609]
```

Now we can again use the `view()` method as we could with any other profile, we shall also highlight where the R_{500} that was passed in at the declaration of the `GalaxyCluster` instance is:

[15]:
```
hym_prof.view(draw_rads={'R$_{500}$': demo_src.r500})
```



4.4.8 Conveniently creating mass profiles with `inv_abel_dens_onion_temp()`

This function wraps various other XGA abilities and allows you to generate mass profiles for a cluster (or sample of clusters) with a single line of code. This particular convenience function uses the `onion_deproj_temp_prof()` function for the creation of the temperature profiles, and the `inv_abel_fitted_model()` function to calculate gas density from surface brightness profiles.

We aren't making use of this in this tutorial as here we want to detail the whole process.

4.4.9 Measuring a mass from a `HydrostaticMass` instance

Of course we are likely to want to measure the mass of the cluster contained within a specific radius (R:math:*_{500}* for instance), and of course there is a method built in to enable that, `mass()`. It works almost identically to the `gas_mass()` method built into the `GasDensity3D` class, which we showed you how to use in the gas density profiles tutorial. The only difference is that you do not need to tell the method which models were used to fit the temperature and density profiles, as the `HydrostaticMass` instance is already aware that.

The method returns a single mass value (with uncertainties calculated at `conf_level`), and a mass distribution:

[16]:
```
mass_val, mass_dist = hym_prof.mass(demo_src.r500)
mass_val
```

[16]: $[4.1269095 \times 10^{14},\ 4.5371415 \times 10^{13},\ 4.5234292 \times 10^{13}]\ \mathrm{M}_{\odot}$

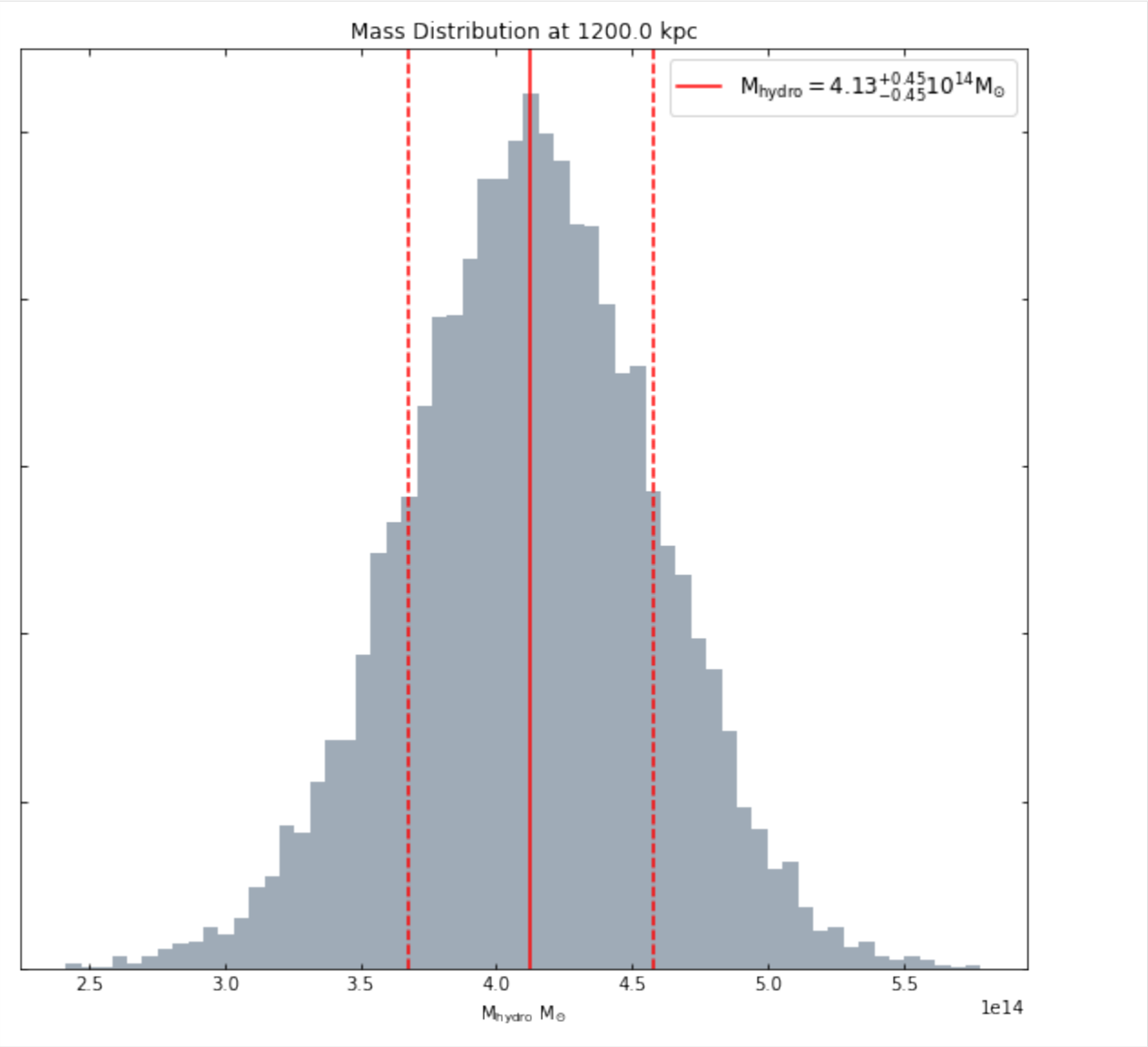
[17]:
```
mass_dist
```

[17]: $[3.8027968 \times 10^{14},\ 3.5735732 \times 10^{14},\ 2.7504132 \times 10^{14},\ \ldots,\ 4.6360991 \times 10^{14},\ 4.483208 \times 10^{14},\ 4.1140427 \times 10^{14}]\ \mathrm{M}_{\odot}$

4.4.10 Viewing the mass distribution at a given radius

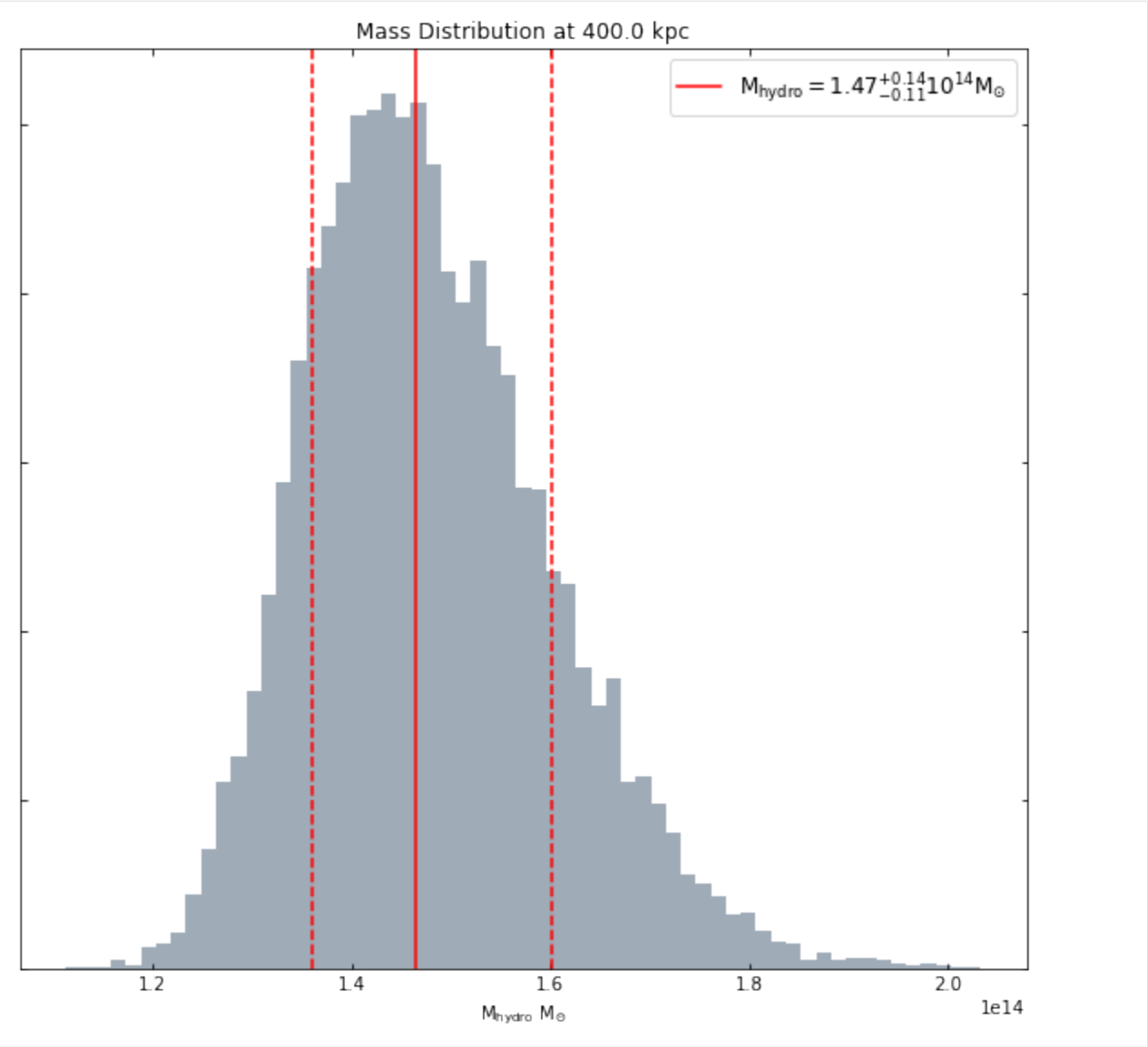
Again in a very similar manner to the gas density profiles, you can view a mass distribution calculated at whatever radius you would like. It takes the secondary output of `mass()` (mass_dist in the cell above) and plots it in a histogram:

[18]:
```
hym_prof.view_mass_dist(demo_src.r500)
```



You can request a mass within any radius you like:

[19]:
```
hym_prof.view_mass_dist(Quantity(400, 'kpc'))
```



4.4.11 Measuring the baryon fraction within a radius

As we now have information both about the total mass of the baryons within the cluster (from the gas density profile that was fed into the `HydrostaticMass` declaration), and the total mass of the whole halo (from the calculation of the mass), we can easily calculate the baryon fraction of the cluster within a given radius.

Simply call `baryon_fraction()`, setting the radius within which the baryon fraction should be measured, and the method will measure both the hydrostatic and gas masses within that radius, then divide the latter by the former. This method is called in the same way as `mass()` was, and returns both a single baryon fraction measurement (with uncertainty), and a baryon fraction distribution:

[20]:
```
bfrac_val, bfrac_dist = hym_prof.baryon_fraction(demo_src.r500)
bfrac_val
```

[20]:
```
[0.22813241, 0.023563652, 0.028947003]
```

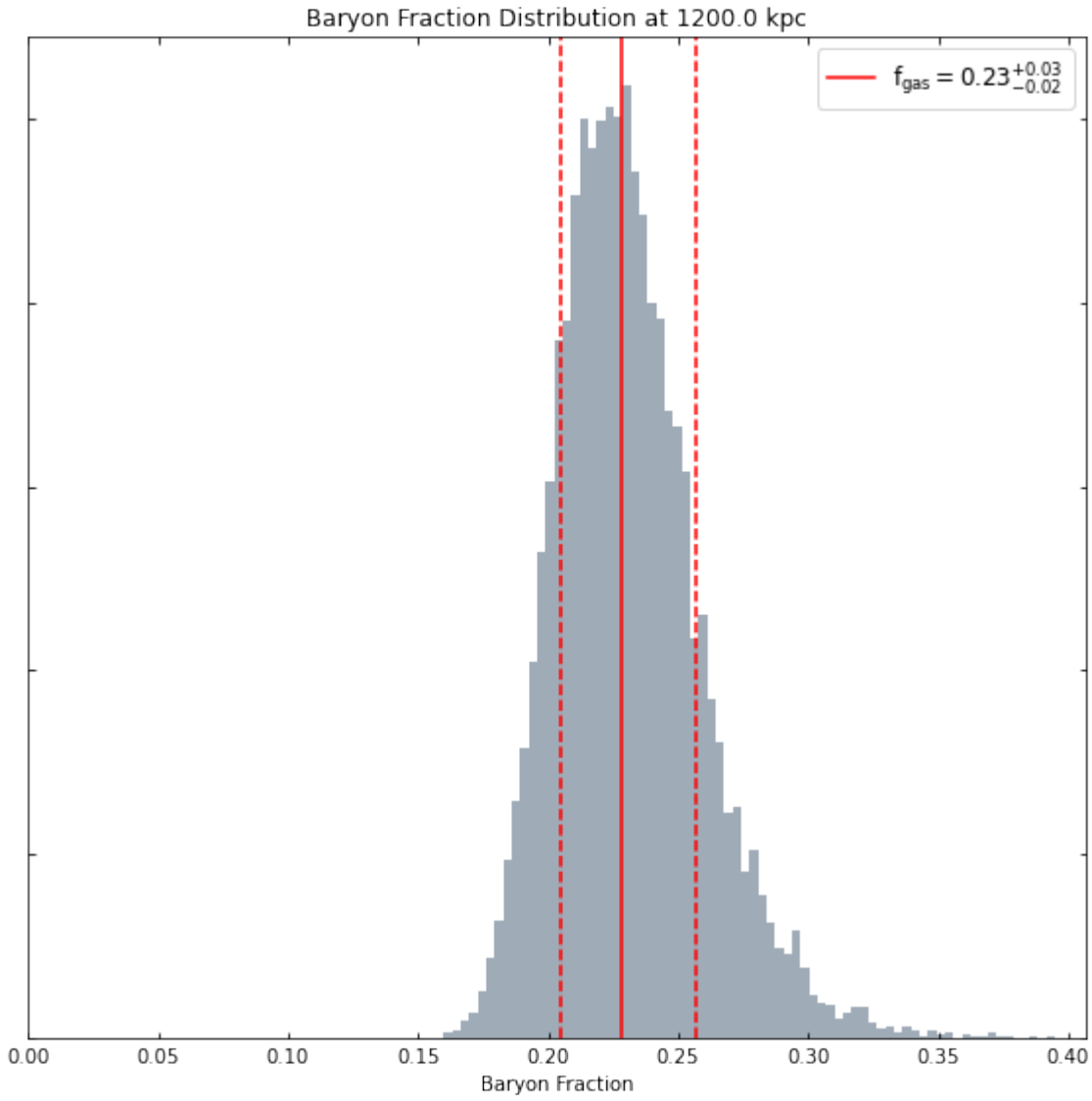
[21]:
```
bfrac_dist
```

[21]:
```
[0.24699649, 0.26298398, 0.33176081, ..., 0.19946204, 0.2088122, 0.2305421]
```

4.4.12 Viewing the baryon fraction distribution within a radius

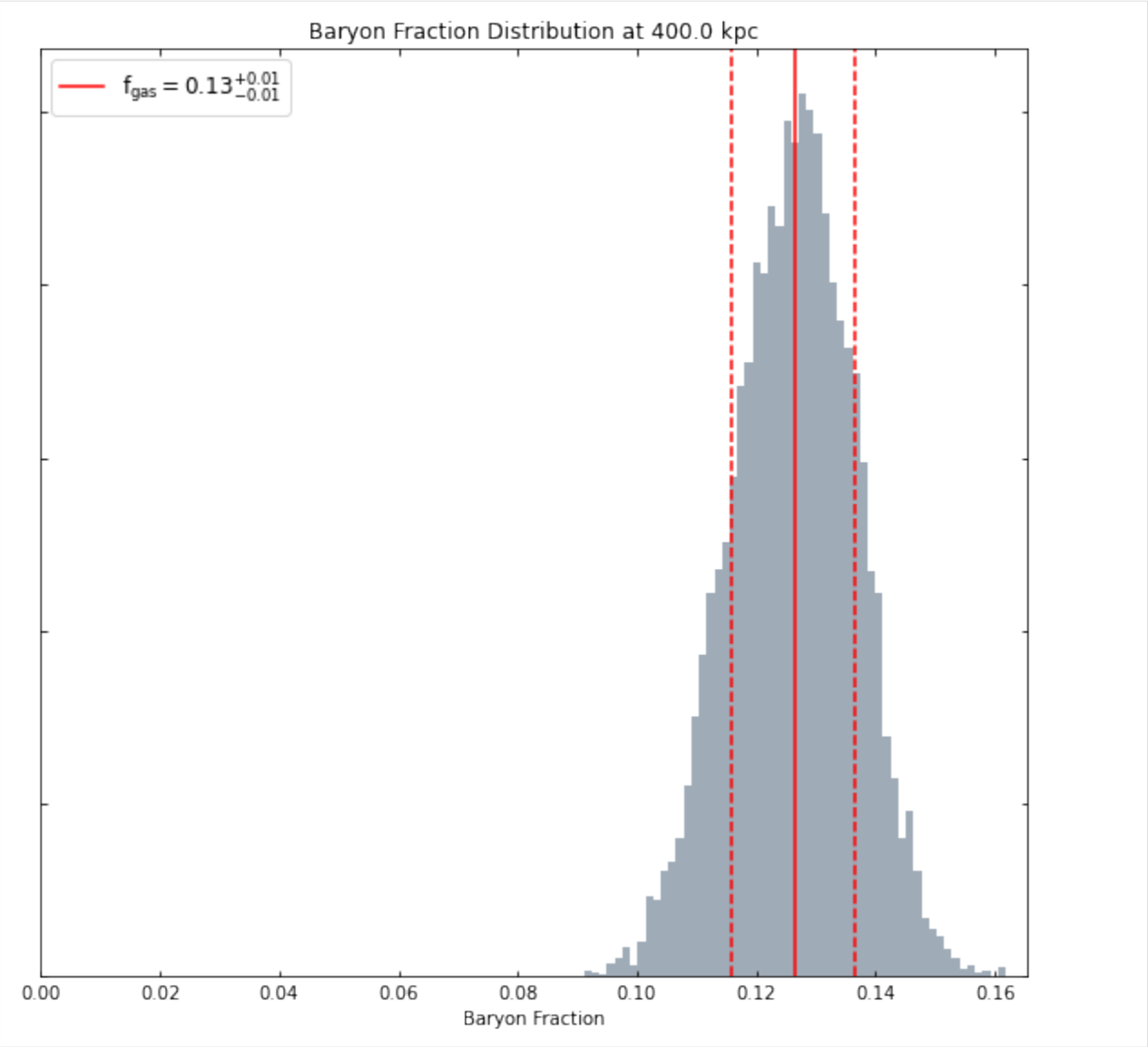
Just as we can view distributions of mass at a specific radius, we can view a baryon fraction distribution:

[22]:
```
hym_prof.view_baryon_fraction_dist(demo_src.r500)
```



At whatever radius we like:

[23]:
```
hym_prof.view_baryon_fraction_dist(Quantity(400, 'kpc'))
```



4.4.13 Generating baryon fraction profiles

As I've just demonstrated, we can easily measure the baryon fraction of the cluster at different radii, and as such it is possible for a profile showing that behaviour to be constructed. A method, `baryon_fraction_profile()` has been implemented in the `HydrostaticMass` class. By default it measures baryon fraction values at the same points that the hydrostatic mass profile measures masses, but again (as the whole process is based upon continuous density and temperature models), you can use whatever radii you like:

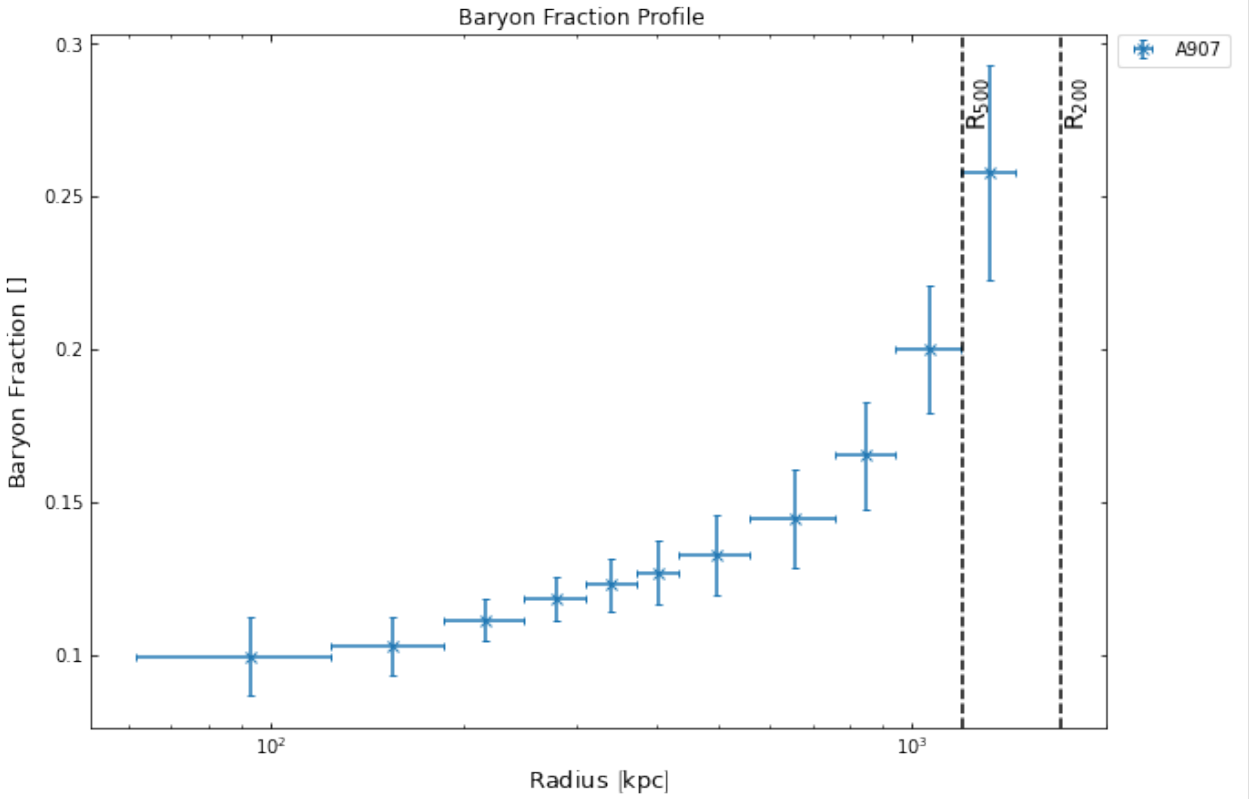
[24]:
```
bfrac_prof = hym_prof.baryon_fraction_profile()
bfrac_prof
```

```
/home/dt237/code/PycharmProjects/XGA/xga/products/profile.py:463: UserWarning: The␣
↪outer radius you supplied is greater than or equal to the outer radius covered by␣
↪the data, so you are effectively extrapolating using the model.
  warn("The outer radius you supplied is greater than or equal to the outer radius␣
↪covered by the data, so"
```

[24]:
```
<xga.products.profile.BaryonFraction at 0x7f73bf4e6c40>
```

Running the method shows us the change in baryon fraction as we move away from the centre of the cluster towards the outskirts, we also use one of the user configurable options of the `view()` method to set the y-axis scale to linear:

[25]:
```
bfrac_prof.view(yscale='linear', draw_rads={'R$_{500}$': demo_src.r500, 'R$_{200}$':␣
↪demo_src.r200})
```



4.4.14 Retrieving a `HydrostaticMass` profile from a `GalaxyCluster`

Yet again we have implemented a handy get method to do this job, `get_hydrostatic_mass_profiles()`. This get method behaves in a slightly different way to others, as it wants to know which temperature and density profiles were used to generate the mass profile, along with the chosen models. It does allow you to pass no information at all however, in which case it will return every hydrostatic mass profile associated with the cluster.

As we manually made this mass profile, it hasn't actually been stored in the source at all, so if we ran the get method now we wouldn't see any results:

[26]:
```
demo_src.get_hydrostatic_mass_profiles()
```

```
---------------------------------------------------------------------------
NoProductAvailableError                   Traceback (most recent call last)
<ipython-input-26-0ce3d76cc5c2> in <module>
----> 1 demo_src.get_hydrostatic_mass_profiles()

~/code/PycharmProjects/XGA/xga/sources/extended.py in get_hydrostatic_mass_
↪profiles(self, temp_prof, temp_model_name, dens_prof, dens_model_name, radii)
    689         """
```

(continues on next page)

(continued from previous page)

```
    690         # Get all the hydrostatic mass profiles associated with this source
--> 691         matched_prods = self.get_profiles('combined_hydrostatic_mass')
    692
    693         # Convert the radii to degrees for comparison with deg radii later

~/code/PycharmProjects/XGA/xga/sources/base.py in get_profiles(self, profile_type,␣
↪obs_id, inst, central_coord, radii, lo_en, hi_en)
   3003             matched_prods = matched_prods[0]
   3004         elif len(matched_prods) == 0:
-> 3005             raise NoProductAvailableError("Cannot find any {p} profiles␣
↪matching your input.".format(p=profile_type))
   3006
   3007         return matched_prods

NoProductAvailableError: Cannot find any combined_hydrostatic_mass profiles matching␣
↪your input.
```

This would not be the case if we had used the mass profile convenience function mentioned earlier, as that automatically stores each profile within the source it was generated for. Here we shall manually add the profile to the source object (which you can do for any product or profile you have defined manually):

[27]:
```
demo_src.update_products(hym_prof)
```

Now if we run the same get method again, we'll retrieve that profile:

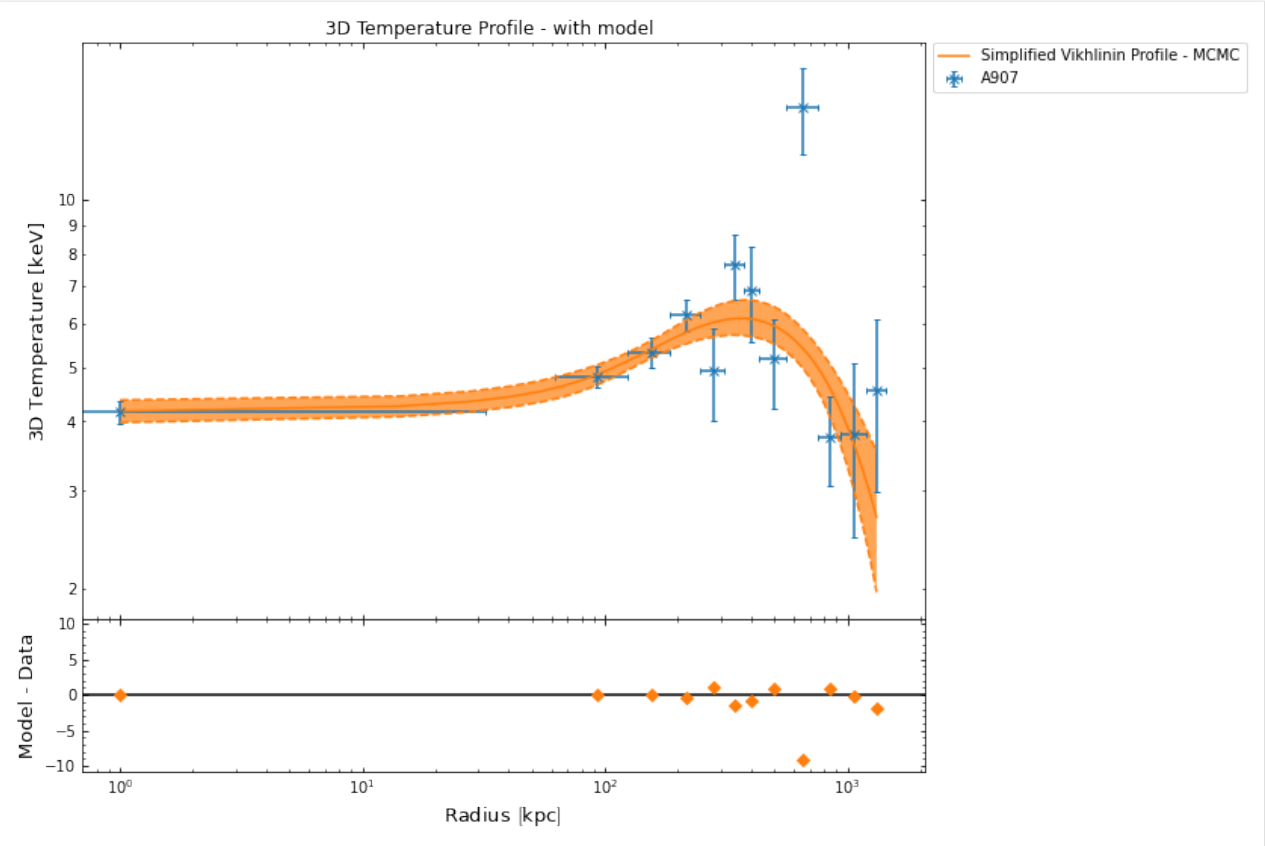
[28]:
```
demo_src.get_hydrostatic_mass_profiles()
```

[28]:
```
<xga.products.profile.HydrostaticMass at 0x7f73abff66a0>
```

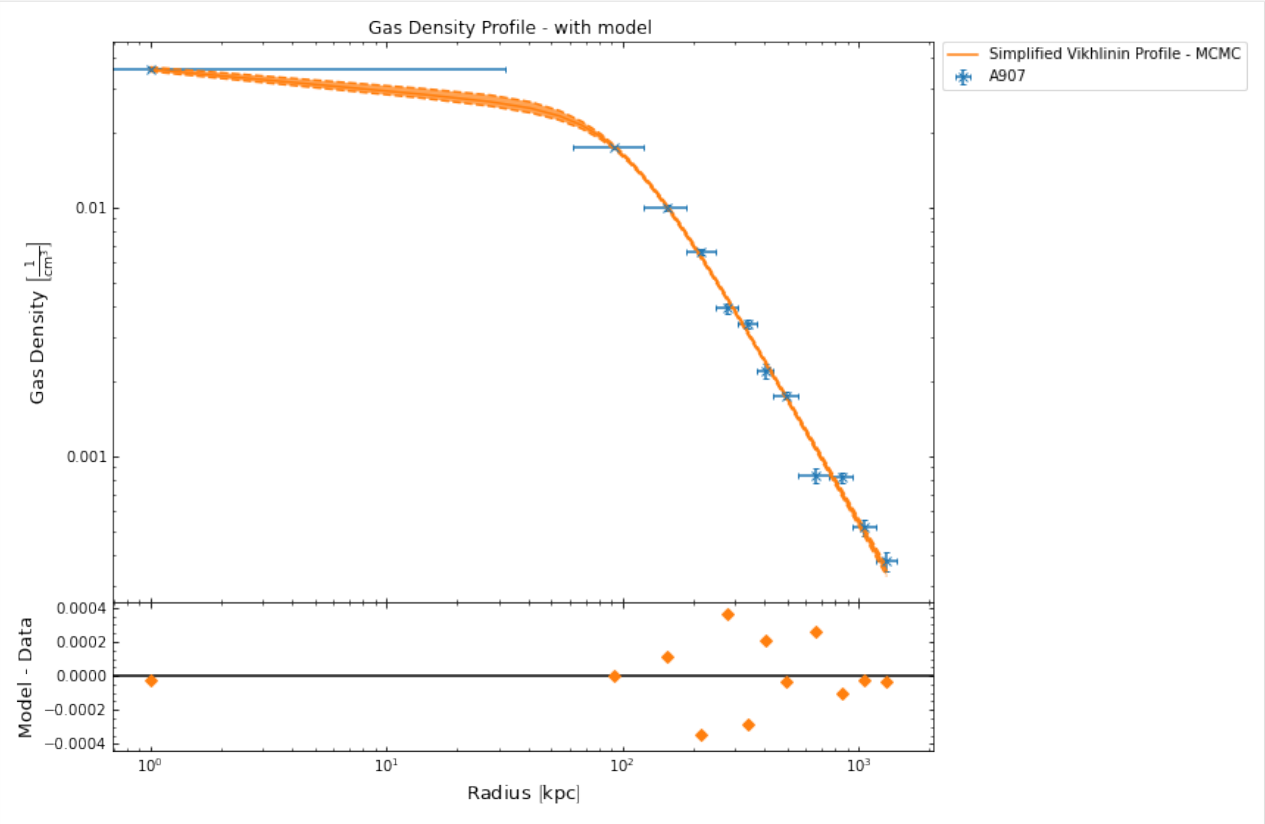
4.4.15 Other useful features of `HydrostaticMass` instances

On the whole, the `HydrostaticMass` profile behaves much like any other. Though there are a couple of useful abilities built in which are specific to this class. Just like gas density profiles have a property that allows them easily find the profile from which they were generated, the hydrostatic mass profile class has `tempererature_profile` and `density_profile` properties to easily retrieve those profiles:

[29]:
```
hym_prof.temperature_profile.view()
```

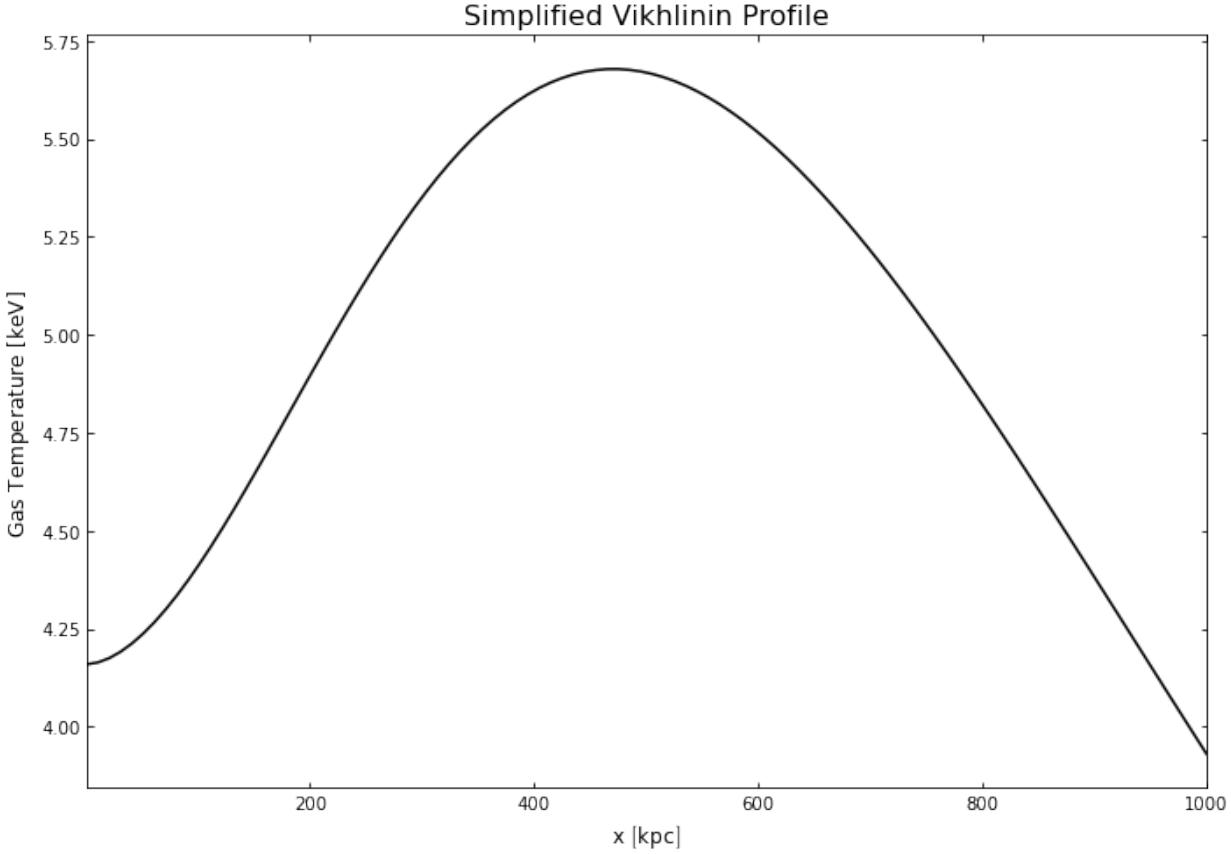


[30]:
```
hym_prof.density_profile.view()
```

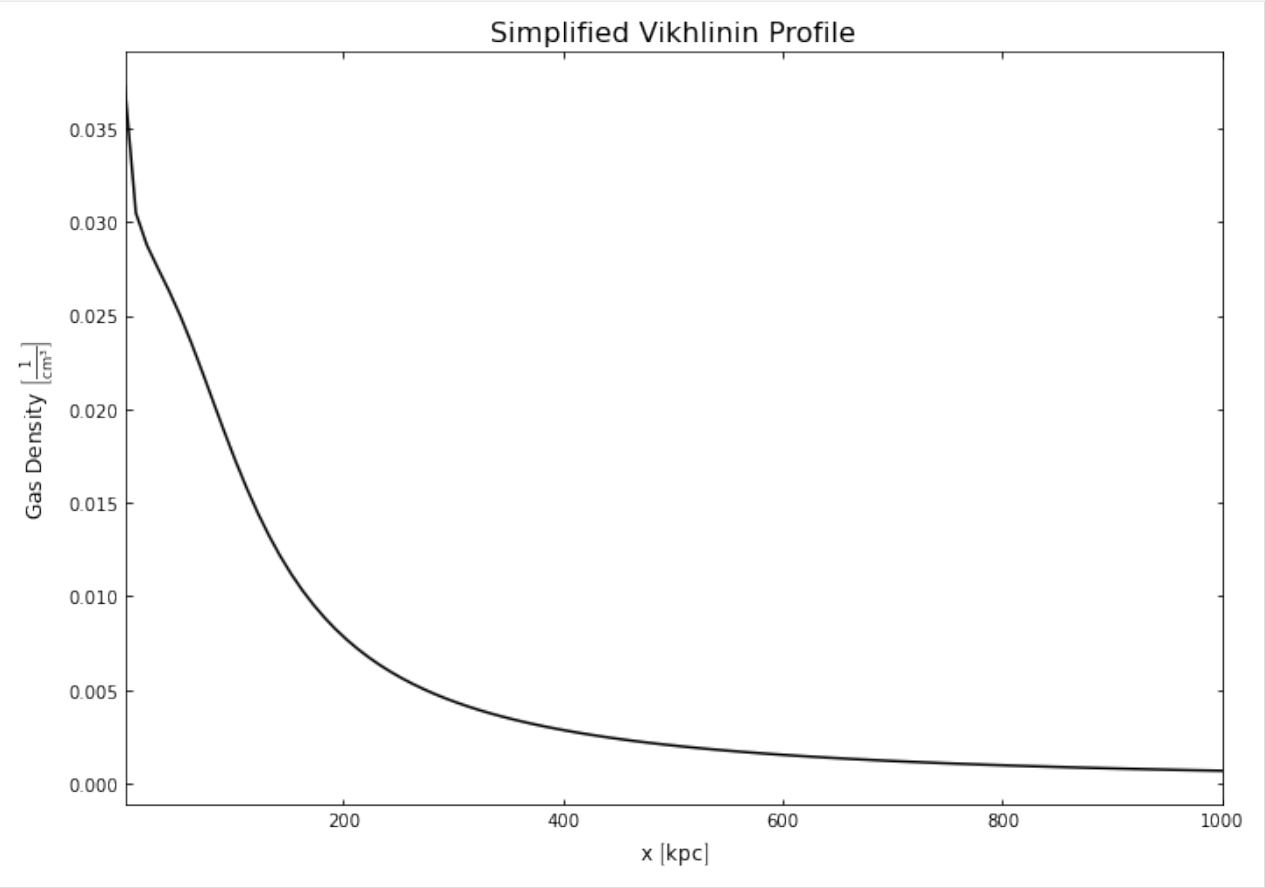


There are also equivalent properties for the fitted models, `temperature_model` and `density_model`, though they could also be accessed through the temperature and density profiles that were just retrieved:

[31]:
```
hym_prof.temperature_model.view(Quantity(np.linspace(1, 1000, 100), 'kpc'), xscale=
↪'linear',
                                yscale='linear', figsize=(10, 7))
```



[32]:
```
hym_prof.density_model.view(Quantity(np.linspace(1, 1000, 100), 'kpc'), xscale='linear
↪',
                            yscale='linear', figsize=(10, 7))
```


Simplified Vikhlinin Profile
Gas Density [1/cm³]
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
200
400
600
800
1000
x [kpc]

CHAPTER

FIVE

UNDER THE HOOD - HOW DO THE METHODS WORK?

5.1 An explanation of XGA's hierarchical clustering peak finder

This section aims to explain how the XGA hierarchical clustering peak finder works, rather than how to actually use it. By the end of this you will hopefully have an understanding of the various steps it goes through to measure a peak for an extended source.

[1]:
```
from xga.sources import GalaxyCluster

from astropy.units import Quantity, pix
from matplotlib import pyplot as plt
import numpy as np
from astropy.visualization import LogStretch, MinMaxInterval, ImageNormalize,␣
↪LinearStretch
```

5.1.1 Point source disclaimer

This method hasn't been properly tested on point sources, but you should not expect it to work well/at all. The basic premise of this peak finder is that you're looking for the peak of the largest collection of bright pixels, which if you're investigating a cluster is most likely to be the intra-cluster medium emission.

For point sources however a poorly removed extended source, or just a cluster of vaguely bright pixels could throw off this method. As such the `PointSource` class peak finding is done with the simple method of just looking for the brightest pixel, using the `simple_peak` method.

5.1.2 Why find X-ray peaks?

For galaxy cluster analysis it can be difficult to define the 'centre' of a given cluster, but it is crucial that we measure some kind of value for the centre, especially when we wish to produce radial profiles of cluster properties. Different methods are used by different analysis teams, which vary depending on the wavelength of the observation; in optical observations, for instance, the brightest cluster galaxy (BCG) is often used as a proxy for the centre.

X-ray astronomers have a distinct advantage here, as the photons we observe trace the actual intra-cluster medium, and so we are in effect tracing the potential well of the halo. As such, the X-ray peak of the cluster is considered a good proxy for the centre (see Yan et al. (2020) for a comparison of various measures of the centre of cluster using hydrodynamical simulations), and has been used to 'correct' miscentering biases encountered when stacking weak lensing profiles in the DES Y1 cluster cosmology analysis (Zhang et al. (2019)).

Once we know the location of the centre of the cluster, we can create analysis regions around it.

5.1.3 Motivation for a new peak finding method

To correct for the spatial variation of exposure times across the XMM field of view (caused primarily by vignetting), peak finding methods should be performed on a 'ratemap', created by dividing an image by an exposure map. Unfortunately, the calculation of exposure maps seems to become difficult near the edges of CCDs, and can the exposure values can be artificially reduced - this in turn can create artificially bright pixels in the ratemap.

If one of those boosted pixel values happens to be in the region we're searching for a cluster centre, then it is possible that it will be selected over the true centre. Not only that, but if point sources are not properly removed it is possible that the brightest pixel in a given region could be in the remnants of the point source emission.

It is possible to apply a smoothing kernel to try and alleviate some of these problems, but I wanted to try another approach that didn't decrease the spatial resolution, and doesn't need to be checked by eye after the fact.

5.1.4 Declaring a `GalaxyCluster` instance

The procedure explained in this section is automatically performed (if the user doesn't change the `use_peak` keyword argument to False on initilisation of the source) on the combined `RateMap` data when an `ExtendedSource` or `GalaxyCluster` object is declared. By default, 0.5-2.0keV ratemaps are used, but the user can override this when initialising the source object. This method is also used to find peaks for the individual ratemaps as well, but the combined peak is the main result used by XGA.

Unlike the ratemaps used in this demonstration, the peak finding algorithm built into the `ExtendedSource` and `GalaxyCluster` classes places masks its ratemaps with a search aperture, as well as removing point sources. We have written the demonstration without the use of source masks to demonstrate an absolute worst case scenario.

We choose to use an Abell cluster that has appeared in several of the XGA tutorials, largely because there are two nearby bright point sources that will help illustrate how the peak finder behaves. **Please note that the overdensity radius and redshift used to declare the object are approximate, and shouldn't be used for a real scientific analysis**.

[2]:
```
# Abell 907, my favourite Galaxy Cluster!
source = GalaxyCluster(149.5904478, -11.0628750, redshift=0.16, name="A907",␣
↪r500=Quantity(1200, 'kpc'))
```

5.1.5 Accessing the automatically measured peak of a `GalaxyCluster`

Before we get stuck in to the details of the peak finder, you can use the `peak` property to access the coordinates that have been measured as the X-ray peak. If, however, `use_peak` was set to False, these coordinates will actually just be the coordinates passed in on the initialisation of the source:

[3]:
```
print(source.peak)
```

```
[149.59251341 -11.06395832] deg
```

If you need a reminder of XGA's photometric products (`RateMap` for instance), I would recommend the photometry with XGA tutorial.

5.1.6 The simplest possible peak finder

Firstly we will demonstrate the simplest possible peak finding method, simply selecting the brighest pixel in the ratemap, though first we shall have to retrieve a ratemap from the source storage storage structure, then directly access its data array. We've chosen the PN ratemap from an individual observation, simply to make the visualisation a little simpler:

[4]:
```
pn_rt = source.get_ratemaps('0201903501', 'pn', lo_en=Quantity(0.5, 'keV'), hi_
↪en=Quantity(2.0, 'keV'))
```

We implement an efficient method using numpy to find the coordinates of the maximum value point in an array, then we store the output in an astropy quantity (as well as then converting those pixel coordinates to degrees):

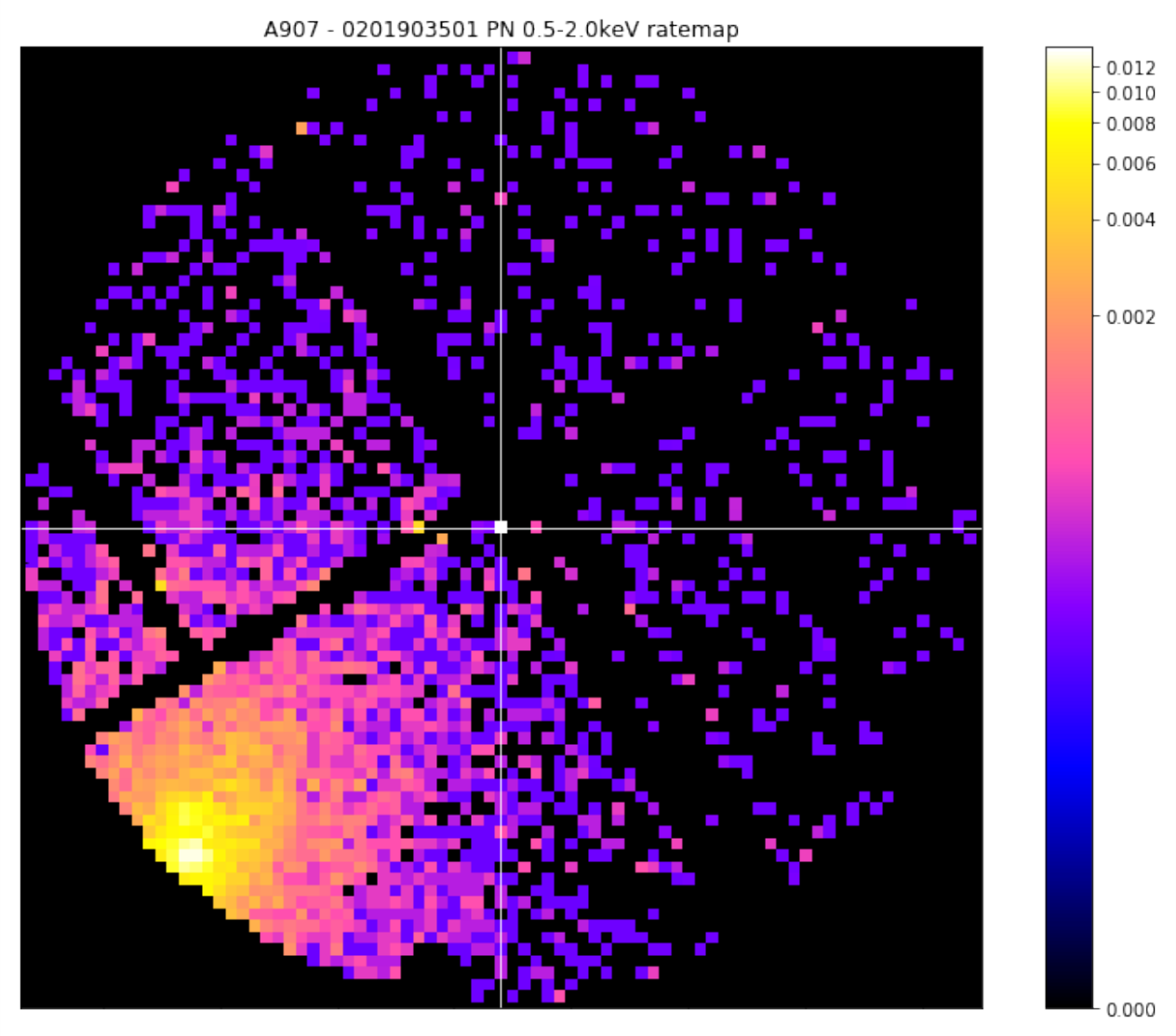
[5]:
```
peak = np.unravel_index(np.argmax(pn_rt.data), pn_rt.shape)

# peak[1] goes into the Quantity first because numpy uses row, column ordering,␣
↪whereas people expect
#  pixel coordinates in column, row ordering.
pn_simple_pix_peak = Quantity([peak[1], peak[0]], pix)
pn_simple_deg_peak = pn_rt.coord_conv(pn_simple_pix_peak, 'deg')
```

To make our point about the problems that these edge pixels can cause, we use the `view()` method to display the coordinate which has been selected by this simplest peak finding method. A mask (which is the size of the usual search aperture when a source is doing this peak finding iteratively) is applied, and we use the `zoom_in` option so that you can see the bright pixel.

We also have to set the `mask_edges` argument to False, otherwise an edge mask would have automatically been applied, and you would not be able to see the brightest pixel.

We change the central coordinate of the mask so that it is centered on the simple peak we have just measured:

[6]:
```
vis_mask = source.get_mask('search', obs_id=pn_rt.obs_id, central_coord=pn_simple_deg_
↪peak)[0]
pn_rt.view(pn_simple_pix_peak, vis_mask, mask_edges=False, zoom_in=True)
```



You can quite clearly see that the pixel that has been selected is a) on a chip edge, and b) much brighter than anything else in the ratemap (look at the colourbar on the side of the visualisation). Ideally this would have selected the cluster that we can see nearby, but evidently we must deal with these spuriously bright pixels that can appear on chip edges.

5.1.7 A solution to the edge problem

We're going to exclude all pixels that sit on the edge of a chip from being peak candidates. As such we need a way to choose which pixels are on an edge and which aren't. We repurposed a method of mapping XMM detectors from another recent piece of work, and used an edge finding algorithm on an exposure map where every value that wasn't 0 was set to 1.

The way this works is that the modified exposure map array is differentiated in the X and Y directions, then the two resultant arrays are added together. As such an edgemap pixel $e_{i,j}$ is

$$e_{i,j} = (p_{i,j} - p_{i+1,j}) + (p_{i,j} - p_{i,j+1})$$

where $p_{i,j}$ is the pixel value of the modified exposure map at coordinates i, j.

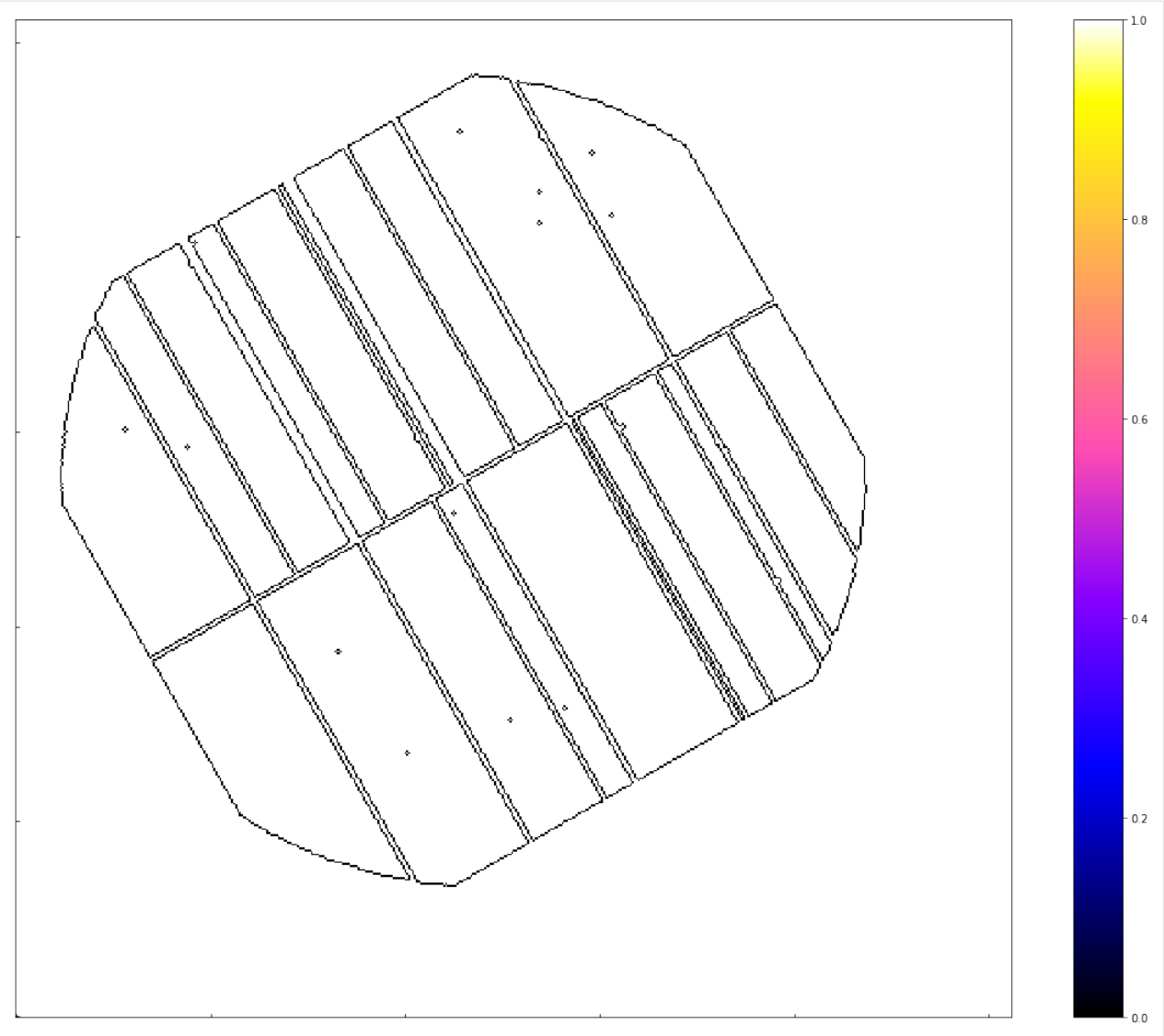
This makes it easy to know when you're going from 'not on a chip' to 'on a chip' and vice versa; $e_{i,j}$ is -1 when moving from 'not on a chip' to 'on a chip', and 1 when moving from 'on a chip' to 'not on a chip'. The modified exposure map is differentiated in both X and Y directions to make sure we capture all the edges, but some will be detected in both directions, so when the arrays are added together there could be edge map pixels with a value of 2, which generally means they are a corner of a chip.

As we wish to know where the edges are, any values of -1 are shifted over by 1, then set to 1, in the direction of the differentiation they belong to. The array is then inverted, so everywhere **but** the edges is 1.

This technique is implemented in the `RateMap` class, and is run automatically on initialisation. You can manually access the edge mask through the `edge_mask` property, and can see the difference that it makes by changing the `mask_edges` argument of the view method.

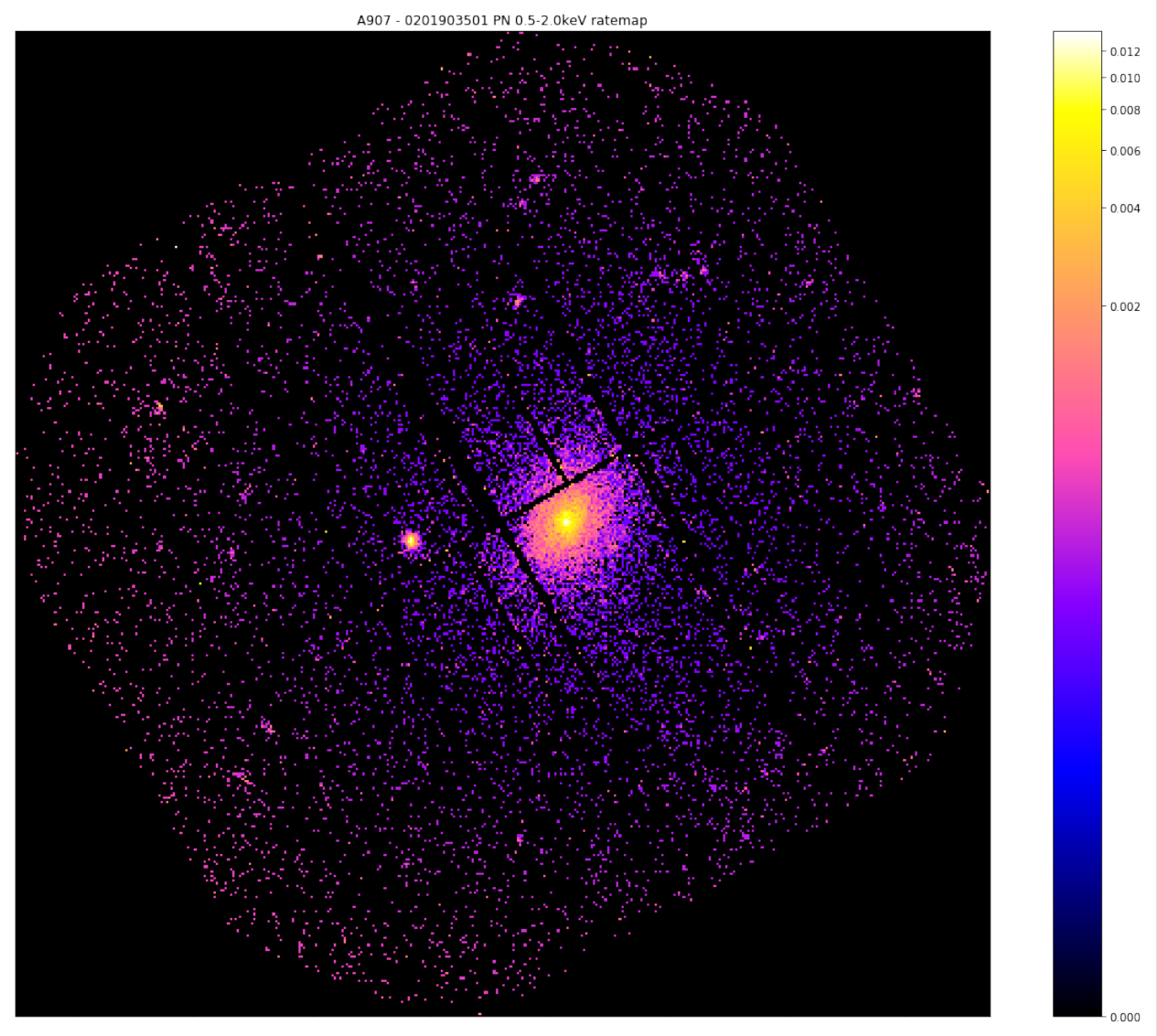
Here we show you the ratemap before and after edge masking, as well as the edgemap itself, to demonstrate that we are obviously actually mapping the EPIC pn camera:

[7]:
```
fig = plt.figure(figsize=(16, 13))

ax = plt.gca()

ax.tick_params(axis='both', direction='in', which='both', top=False, right=False)
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])

norm = ImageNormalize(data=pn_rt.edge_mask, interval=MinMaxInterval(),␣
↪stretch=LinearStretch())
ax.imshow(pn_rt.edge_mask, norm=norm, origin="lower", cmap="gnuplot2")

plt.colorbar(ax.images[0])
plt.tight_layout()
plt.show()
```



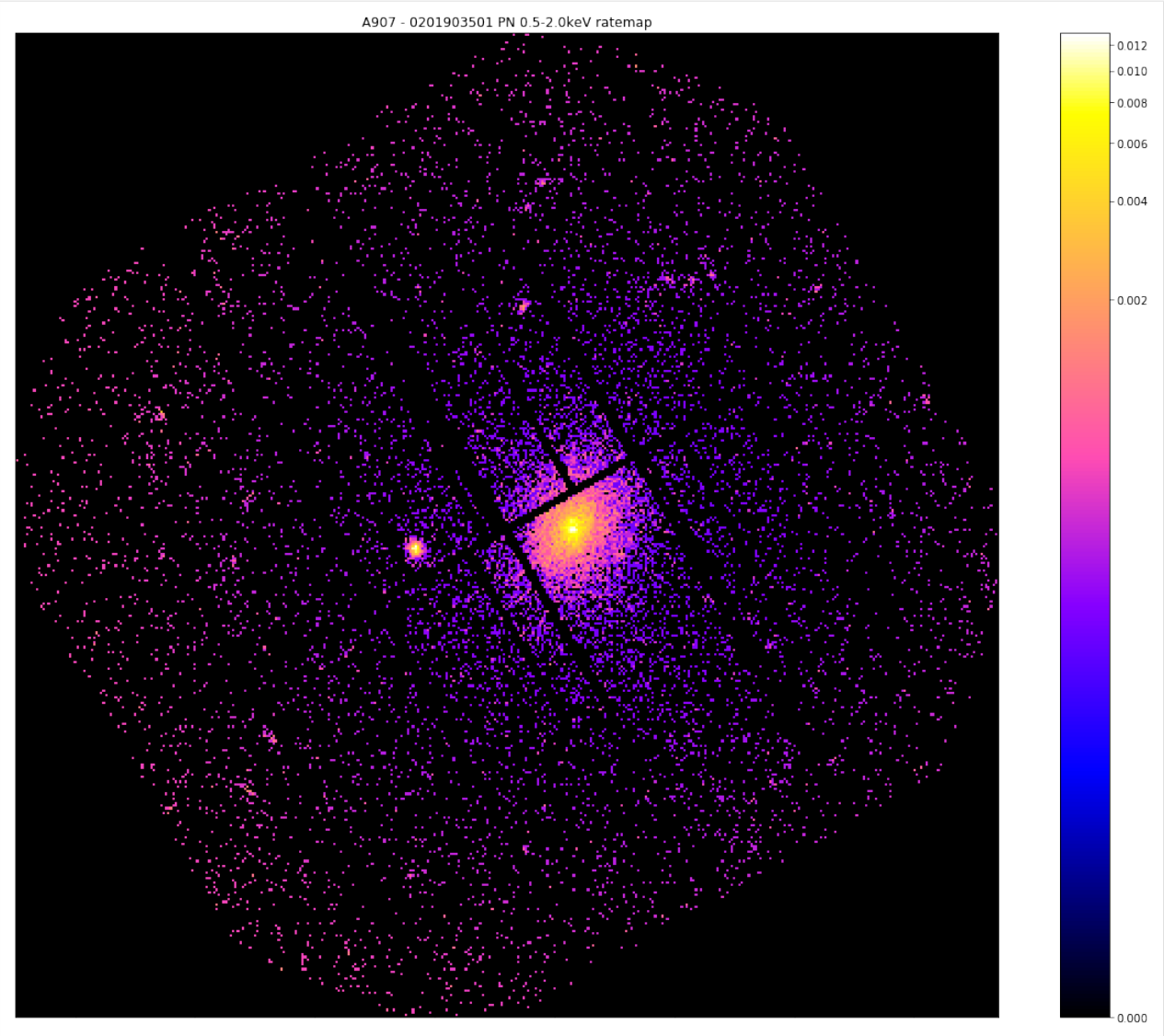
The before view:

[8]:
```
pn_rt.view(zoom_in=True, mask_edges=False, figsize=(16, 13))
```



The after view:

[9]:
```
pn_rt.view(zoom_in=True, mask_edges=True, figsize=(16, 13))
```



Please note that the simple peak finder technique (of looking for the brightest pixel) is implemented as the `simple_peak` method of the `RateMap` class, but we apply the edge mask which we just discussed **before** searching for the brightest pixel.

5.1.8 New Peak Finding Method

To (hopefully) account for any point source remnants in the RateMap, we introduce a new method that uses a hierarchical clustering algorithm to choose the pixels most likely to belong to the cluster we're searching for.

You must bear in mind that by this point in the peak finding function of the `GalaxyCluster` class (for instance) the data is masked to remove point sources, and a search aperture mask is placed around the original coordinates supplied by the user. We choose not to mask the data as it makes it easier to show you what the new method does, and also demonstrates how well it performs even under the worst conditions.

Demonstrating the downside of the simple peak method

We shall just another ratemap that helps us make our point here:

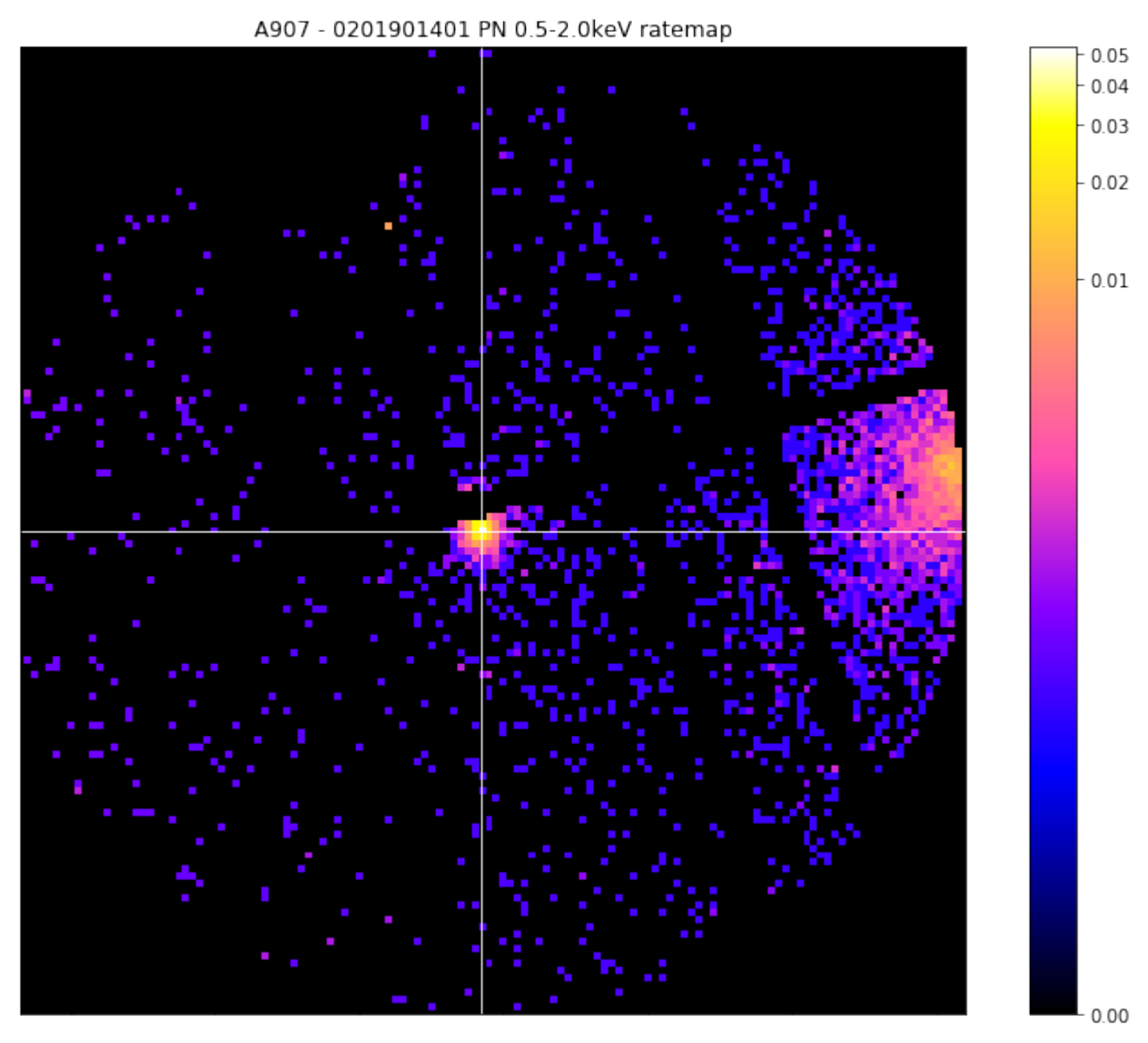
[10]:
```
pn_rt = source.get_ratemaps('0201901401', 'pn', Quantity(0.5, 'keV'), Quantity(2.0,
↪'keV'))
```

So now that we have developed this edge mask we can apply it and once again search for the brightest pixel, though this time we shall use the built in `simple_peak()` method. When this is called for an actual analysis it expects a mask to be passed to the `mask` argument, to remove sources that shouldn't be considered for the peak position. As we have already stated, we are deliberately not masking this image, so we need to create a flat mask of ones to pass to the peak finder:

[11]:
```
# The peak finding methods of a RateMap require a numpy mask, but for the␣
↪demonstration I don't want to
#  mask the data, so I construct a numpy array of ones to act as a flat mask
flat_mask = np.ones(pn_rt.shape)
simple_peak, near_edge = pn_rt.simple_peak(flat_mask, 'deg')
```

We call the view option, having created a custom mask centered on the simple peak we just found and set the `zoom_in` argument to True:

[12]:
```
# Then we can view the combined data with the crosshair at the 'simple peak'
vis_mask = source.get_custom_mask(Quantity(0.08, 'deg'), remove_interlopers=False,␣
↪obs_id=pn_rt.obs_id,
                                  central_coord=simple_peak)
pn_rt.view(simple_peak, vis_mask, zoom_in=True)
```



We find that we're still not getting a good answer, and have selected a bright point source instead of the cluster that we can clearly see on the right hand side of the image.

Now, hopefully in any real use case, the source finder you've used would have found, and produced a region for, a powerful point source like the one we have selected - but the same thing can happen with a particularly bright point source remnant.

What is hierarchical clustering and how does help?

A clever type of unsupervised machine learning algorithm that can be used to cluster data points in some arbitrary feature space until a completion criteria has been met. In this case we are clustering on X and Y spatial coordinates, so intuitively it is quite easy to understand, but the algorithm can cluster any kind of information.

It starts off by assuming that each point is a separate cluster, then repeatedly identifies the clusters that are closest together and combines them. This XGA implementation uses the scipy `fclusterdata` function, with a distance criterion of five pixels by default, though the distance can be changed by setting the `clustering_peak()` [method's]((../../xga.products.html#xga.products.phot.RateMap.clustering_peak) `max_dist` argument.

We take the spatial coordinates of the pixels with the top 5% (by default) of values in the ratemap, then run this hierarchical clustering algorithm on them. Once it's complete we select the largest cluster of points and assume that this is the source that we're looking for. This excludes any small patches of emission that might be left over from point sources, and so you can just select the pixel with the maximum value **in the chosen point cluster**.

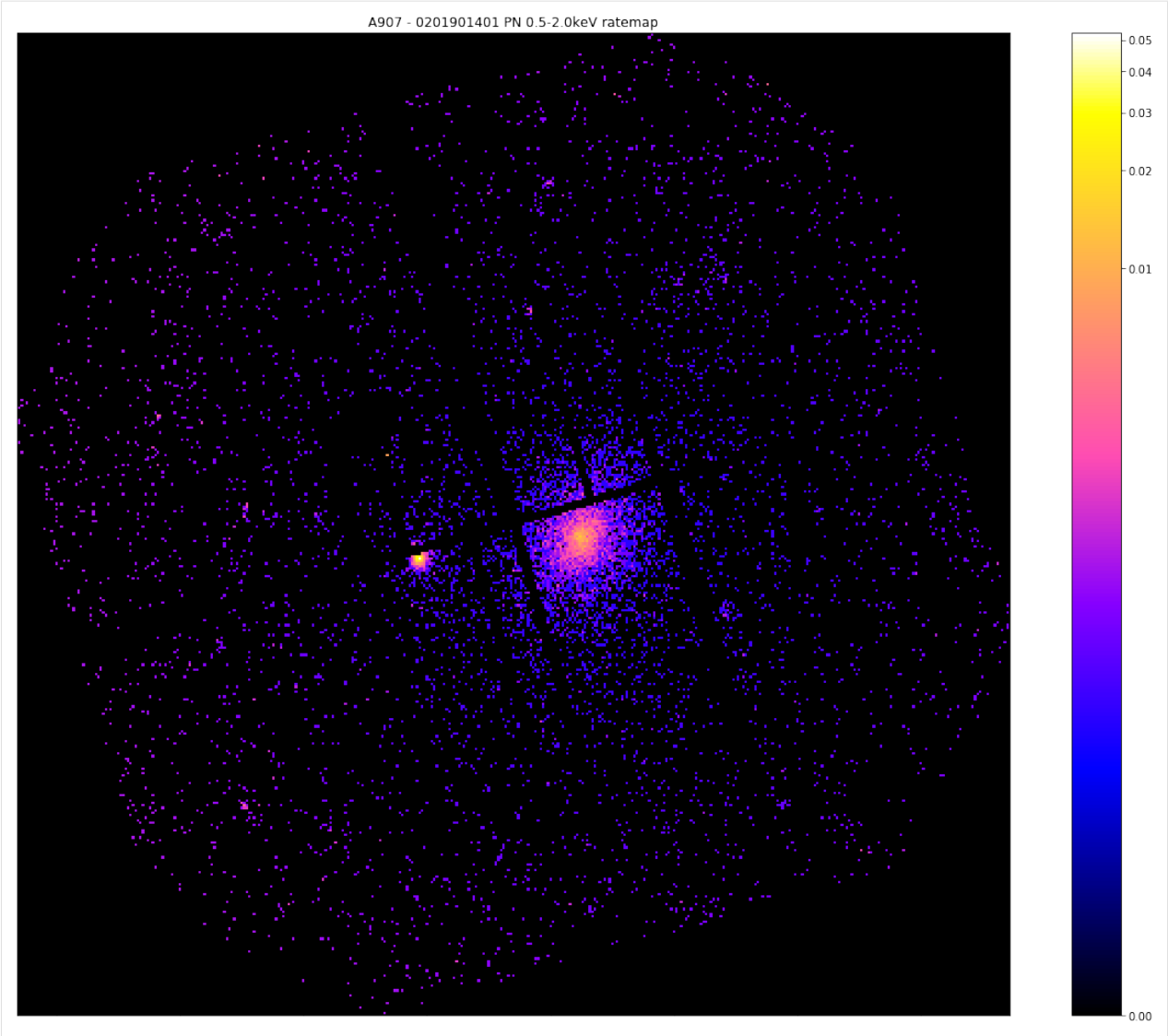
Here we show how the method is called manually, and we have enabled the `clean_point_clusters` function so that any point clusters with fewer than four pixels associated with them are removed from the returned, non-source, point cluster list. This is purely for aesthetic reasons when making the visualisation:

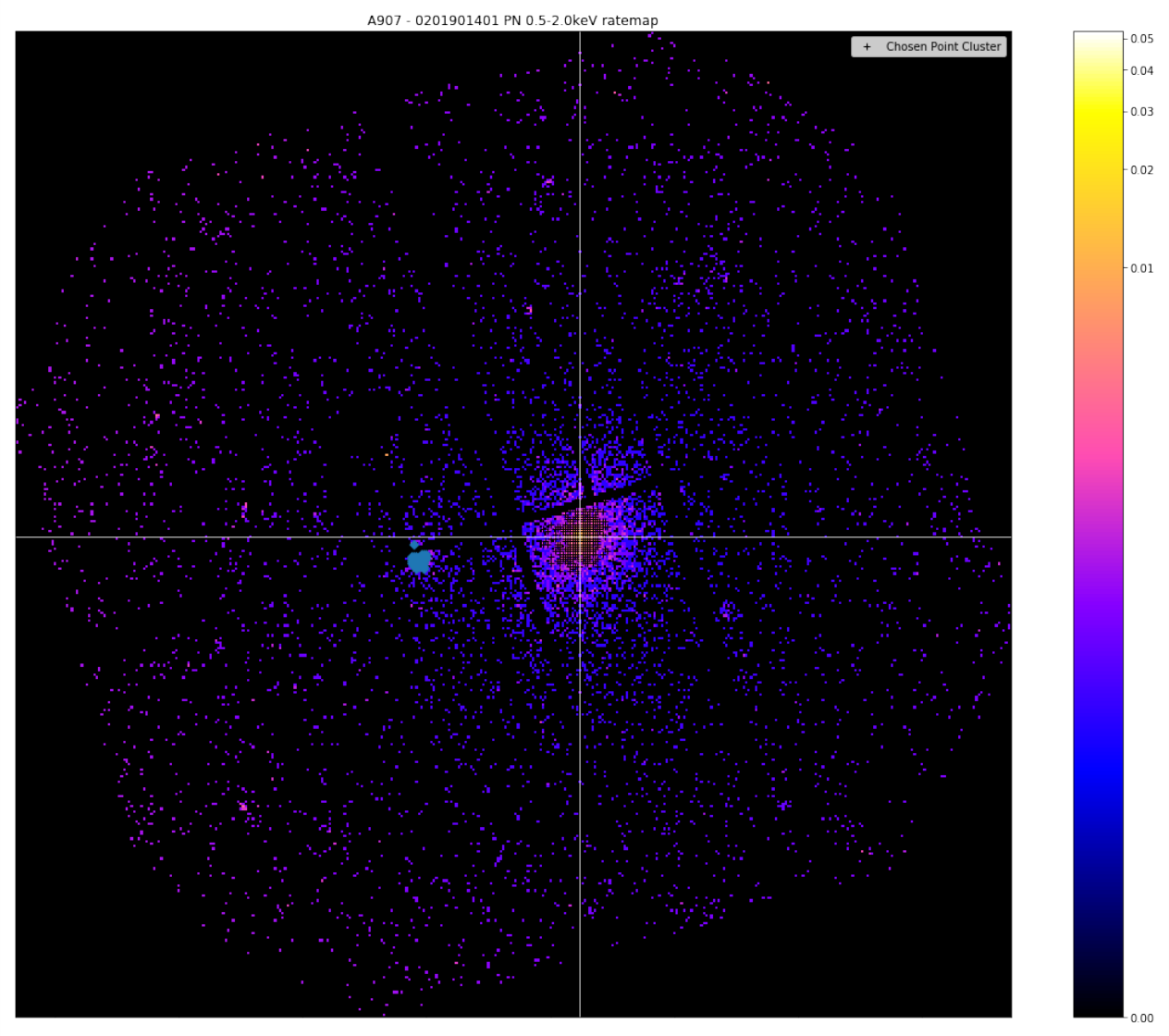
[13]:
```
clever_pix_peak, near_edge, src, others = pn_rt.clustering_peak(flat_mask, 'deg',␣
↪clean_point_clusters=True)
```

Now we make the visualisation using the `view()` method of the `RateMap` class, particulary with the `chosen_points` and `other_points` arguments, which were specifically designed to plot point clusters from this peak finding algorithm. At first we show the ratemap which the peak finder worked upon, then we show it with the point clusters (and the chosen peak coordinate) plotted on top.

We can see that the clustering peak finder has noticed the bright point source to the left, but has decided that because the galaxy cluster point cluster is larger, it will take the peak from there:

[14]:
```
vis_mask = source.get_custom_mask(Quantity(1000, 'kpc'), obs_id=pn_rt.obs_id, central_
↪coord=clever_pix_peak,
                                  remove_interlopers=False)

pn_rt.view(zoom_in=True, figsize=(16, 13))
pn_rt.view(clever_pix_peak, chosen_points=src, other_points=others, zoom_in=True,␣
↪figsize=(16, 13))
```


A907 - 0201901401 PN 0.5-2.0keV ratemap
0.05
0.04
0.03
0.02
0.01
0.00



5.1.9 How is it implemented in `GalaxyCluster`

The implementation of this process in the ExtendedSource and GalaxyCluster classes is actually iterative. Firstly, an aperture with a radius of 500kpc (or 5 arcminutes) is placed at the user supplied source coordinates, with interloper sources being masked out. The radius is only set to 5 arcminutes if the user has defined an extended source with no redshift information (which is allowed), if the source is a GalaxyCluster object, redshift information is required so it will always use 500kpc.

Then the algorithm follows these steps (for upto 20 iterations before it throws an error):

1. Runs the hierarchical peak finder.
2. Re-centres the search aperture at the new coordinates.
3. Checks to see if the new peak is within 15kpc (or 0.15 arcminutes) of the last central coordinate.
4. If it is, consider the peak converged and exit, if it isn't (or if the algorithm is on the first iteration), go back to 1.

Initially run on the combined ratemap, then on the individual ratemaps of the different observations/instruments. If the

combined ratemap peak won't converge, a hard error will be thrown (as this is what XGA analyses uses as the centre), the individual peaks are allowed to not converge.

The user can also choose to use their initial coordinates as the cluster centre, if they don't want to find and use the peak.

[ ]:

[ ]:

[ ]:

[ ]:

CHAPTER

SIX

A NOTE ON XGA'S PARALLELISM

In every SAS function built into XGA (and indeed many functions in other parts of the module), you will find a **num_cores** keyword argument which tells the function how many cores on your local machine it is allowed to use. The default value is 90% of the available cores on your system, though you are of course free to set your own value when you call the functions.

To see the number of cores which have automatically allocated to XGA, you can import the NUM_CORES constant from the base xga module. You can also manually set this value globally, before running anything. Either set the *num_cores* option in the [XGA_SETUP] section of the configuration file, or simply set the NUM_CORES constant imported from xga.

CHAPTER

SEVEN

XGA PACKAGE

7.1 sources

7.1.1 sources.base module

class xga.sources.base.**BaseSource**(*ra*, *dec*, *redshift=None*, *name=None*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_products=True*, *load_fits=False*)

Bases: object

The overlord of all XGA classes, the superclass for all source classes. This contains a huge amount of functionality upon which the rest of XGA is built, includes selecting observations, reading in data products, and storing newly created data products.

property ra_dec

A getter for the original ra and dec entered by the user.

Returns The ra-dec coordinates entered by the user when the source was first defined

Return type Quantity

property default_coord

A getter for the default analysis coordinate of this source. :return: An Astropy quantity containing the default analysis coordinate. :rtype: Quantity

update_products(*prod_obj*)

Setter method for the products attribute of source objects. Cannot delete existing products, but will overwrite existing products. Raises errors if the ObsID is not associated with this source or the instrument is not associated with the ObsID. Lists of products can also be passed and will be added to the source storage structure, these lists may also contain None values, as typically XGA will return None if a profile fails to generate (for instance), in which case that entry will simply be ignored.

Parameters **prod_obj** (*BaseProduct/BaseAggregateProduct/BaseProfile1D/List[BaseProduct]/List[BaseProfile1D]*) – The new product object(s) to be added to the source object.

get_products(*p_type*, *obs_id=None*, *inst=None*, *extra_key=None*, *just_obj=True*)

This is the getter for the products data structure of Source objects. Passing a 'product type' such as 'events' or 'images' will return every matching entry in the products data structure.

Parameters

- **p_type** (*str*) – Product type identifier. e.g. image or expmap.

- **obs_id** (*str*) – Optionally, a specific obs_id to search can be supplied.
- **inst** (*str*) – Optionally, a specific instrument to search can be supplied.
- **extra_key** (*str*) – Optionally, an extra key (like an energy bound) can be supplied.
- **just_obj** (*bool*) – A boolean flag that controls whether this method returns just the product objects, or the other information that goes with it like ObsID and instrument.

Returns List of matching products.

Return type List[*BaseProduct*]

update_queue(*cmd_arr*, *p_type_arr*, *p_path_arr*, *extra_info*, *stack=False*)

Small function to update the numpy array that makes up the queue of products to be generated.

Parameters

- **cmd_arr** (*np.ndarray*) – Array containing SAS commands.
- **p_type_arr** (*np.ndarray*) – Array of product type identifiers for the products generated by the cmd array. e.g. image or expmap.
- **p_path_arr** (*np.ndarray*) – Array of final product paths if cmd is successful
- **extra_info** (*np.ndarray*) – Array of extra information dictionaries
- **stack** – Should these commands be executed after a preceding line of commands, or at the same time.

Returns

get_queue()

Calling this indicates that the queue is about to be processed, so this function combines SAS commands along columns (command stacks), and returns N SAS commands to be run concurrently, where N is the number of columns.

Returns List of strings, where the strings are bash commands to run SAS procedures, another list of strings, where the strings are expected output types for the commands, a list of lists of strings, where the strings are expected output paths for products of the SAS commands.

Return type Tuple[List[str], List[str], List[List[str]]]

get_att_file(*obs_id*)

Fetches the path to the attitude file for an XMM observation.

Parameters **obs_id** – The ObsID to fetch the attitude file for.

Returns The path to the attitude file.

Return type str

property obs_ids

Property getter for ObsIDs associated with this source that are confirmed to have events files.

Returns A list of the associated XMM ObsIDs.

Return type List[str]

property detected

A property getter to return if a match of the correct type has been found.

Returns The detected boolean attribute.

Return type bool

source_back_regions(*reg_type*, *obs_id=None*, *central_coord=None*)
A method to retrieve source region and background region objects for a given source type with a given central coordinate.

Parameters

- **reg_type** (*str*) – The type of region which we wish to get from the source.
- **obs_id** (*str*) – The ObsID that the region is associated with (if appropriate).
- **central_coord** (*Quantity*) – The central coordinate of the region.

Returns The method returns both the source region and the associated background region.

Return type

within_region(*region*)
This method finds interloper sources that lie within the user supplied region.

Parameters **region** (*SkyRegion*) – The region in which we wish to search for interloper sources (for instance a source region or background region).

Returns A list of regions that lie within the user supplied region.

Return type List[SkyRegion]

get_source_mask(*reg_type*, *obs_id=None*, *central_coord=None*)
Method to retrieve source and background masks for the given region type.

Parameters

- **reg_type** (*str*) – The type of region for which to retrieve the mask.
- **obs_id** (*str*) – The ObsID that the mask is associated with (if appropriate).
- **central_coord** (*Quantity*) – The central coordinate of the region.

Returns The source and background masks for the requested ObsID (or the combined image if no ObsID).

Return type Tuple[np.ndarray, np.ndarray]

get_interloper_mask(*obs_id=None*)
Returns a mask for a given ObsID (or combined data if no ObsID given) that will remove any sources that have not been identified as the source of interest.

Parameters **obs_id** (*str*) – The ObsID that the mask is associated with (if appropriate).

Returns A numpy array of 0s and 1s which acts as a mask to remove interloper sources.

Return type ndarray

get_mask(*reg_type*, *obs_id=None*, *central_coord=None*)
Method to retrieve source and background masks for the given region type, WITH INTERLOPERS REMOVED.

Parameters

- **reg_type** (*str*) – The type of region for which to retrieve the interloper corrected mask.
- **obs_id** (*str*) – The ObsID that the mask is associated with (if appropriate).
- **central_coord** (*Quantity*) – The central coordinate of the region.

Returns The source and background masks for the requested ObsID (or the combined image if no ObsID).

Return type Tuple[np.ndarray, np.ndarray]

get_custom_mask(*outer_rad*, *inner_rad=<Quantity 0. arcsec>*, *obs_id=None*, *central_coord=None*, *remove_interlopers=True*)

A simple, but powerful method, to generate mask a mask within a custom radius for a given ObsID.

Parameters

- **outer_rad** (*Quantity*) – The outer radius of the mask.
- **inner_rad** (*Quantity*) – The inner radius of the mask, the default is zero arcseconds.
- **obs_id** (*str*) – The ObsID for which to generate the mask, default is None which will return a mask generated from a combined image.
- **central_coord** (*Quantity*) – The central coordinates of the mask, the default is None which will use the default coordinates of the source.
- **remove_interlopers** (*bool*) – Whether an interloper mask should be combined with the custom mask to remove interloper point sources.

Returns A numpy array containing the desired mask.

Return type np.ndarray

get_snr(*outer_radius*, *central_coord=None*, *lo_en=None*, *hi_en=None*, *obs_id=None*, *inst=None*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *allow_negative=False*, *exp_corr=True*)

This takes a region type and central coordinate and calculates the signal to noise ratio. The background region is constructed using the back_inn_rad_factor and back_out_rad_factor values, the defaults of which are 1.05*radius and 1.5*radius respectively.

Parameters

- **outer_radius** (*Quantity/str*) – The radius that SNR should be calculated within, this can either be a named radius such as r500, or an astropy Quantity.
- **central_coord** (*Quantity*) – The central coordinate of the region.
- **lo_en** (*Quantity*) – The lower energy bound of the ratemap to use to calculate the SNR. Default is None, in which case the lower energy bound for peak finding will be used (default is 0.5keV).
- **hi_en** (*Quantity*) – The upper energy bound of the ratemap to use to calculate the SNR. Default is None, in which case the upper energy bound for peak finding will be used (default is 2.0keV).
- **obs_id** (*str*) – An ObsID of a specific ratemap to use for the SNR calculation. Default is None, which means the combined ratemap will be used. Please note that inst must also be set to use this option.
- **inst** (*str*) – The instrument of a specific ratemap to use for the SNR calculation. Default is None, which means the combined ratemap will be used.
- **psf_corr** (*bool*) – Sets whether you wish to use a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want to use is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want to use is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of iterations.

- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).
- **exp_corr** (*bool*) – Should signal to noises be measured with exposure time correction, default is True. I recommend that this be true for combined observations, as exposure time could change quite dramatically across the combined product.

Returns The signal to noise ratio.

Return type float

regions_within_radii(*inner_radius*, *outer_radius*, *deg_central_coord*, *interloper_regions=None*)

This function finds and returns any interloper regions that have any part of their boundary within the specified radii, centered on the specified central coordinate.

Parameters

- **inner_radius** (*Quantity*) – The inner radius of the area to search for interlopers in.
- **outer_radius** (*Quantity*) – The outer radius of the area to search for interlopers in.
- **deg_central_coord** (*Quantity*) – The central coordinate (IN DEGREES) of the area to search for interlopers in.
- **interloper_regions** (*np.ndarray*) – An optional parameter that allows the user to pass a specific list of regions to check. Default is None, in which case the interloper_regions internal list will be used.

Returns A numpy array of the interloper regions within the specified area.

Return type np.ndarray

get_annular_sas_region(*inner_radius*, *outer_radius*, *obs_id*, *inst*, *output_unit=Unit("xmm_sky")*, *rot_angle=<Quantity 0. deg>*, *interloper_regions=None*, *central_coord=None*)

A method to generate a SAS region string for an arbitrary circular or elliptical annular region, with interloper sources removed.

Parameters

- **inner_radius** (*Quantity*) – The inner radius/radii of the region you wish to generate in SAS, if the quantity has multiple elements then an elliptical region will be generated, with the first element being the inner radius on the semi-major axis, and the second on the semi-minor axis.
- **outer_radius** (*Quantity*) – The inner outer_radius/radii of the region you wish to generate in SAS, if the quantity has multiple elements then an elliptical region will be generated, with the first element being the outer radius on the semi-major axis, and the second on the semi-minor axis.
- **obs_id** (*str*) – The ObsID of the observation you wish to generate the SAS region for.
- **inst** (*str*) – The instrument of the observation you to generate the SAS region for.
- **output_unit** (*UnitBase/str*) – The output unit for this SAS region, either xmm_sky or xmm_det.
- **interloper_regions** (*np.ndarray*) – The interloper regions to remove from the source region, default is None, in which case the function will run self.regions_within_radii.

- **rot_angle** (*Quantity*) – The rotation angle of the source region, default is zero degrees.
- **central_coord** (*Quantity*) – The coordinate on which to centre the source region, default is None in which case the function will use the default_coord of the source object.

Returns A string for use in a SAS routine that describes the source region, and the regions to cut out of it.

Return type str

property nH
Property getter for neutral hydrogen column attribute.

Returns Neutral hydrogen column surface density.

Return type Quantity

property redshift
Property getter for the redshift of this source object.

Returns Redshift value

Return type float

property on_axis_obs_ids
This method returns an array of ObsIDs that this source is approximately on axis in.

Returns ObsIDs for which the source is approximately on axis.

Return type list

property cosmo
This method returns whatever cosmology object is associated with this source object.

Returns An astropy cosmology object specified for this source on initialization.

Return type Cosmology

property name
The name of the source, either given at initialisation or generated from the user-supplied coordinates.

Returns The name of the source.

Return type str

add_fit_data(*model*, *tab_line*, *lums*, *spec_storage_key*)
A method that stores fit results and global information about a the set of spectra in a source object. Any variable parameters in the fit are stored in an internal dictionary structure, as are any luminosities calculated. Other parameters of interest are store in other internal attributes. This probably shouldn't ever be used by the user, just other parts of XGA, hence why I've asked for a spec_storage_key to be passed in rather than all the spectrum configuration options individually.

Parameters

- **model** (*str*) – The XSPEC definition of the model used to perform the fit. e.g. constant*tbabs*apec
- **tab_line** – The table line with the fit data.
- **lums** (*dict*) – The various luminosities measured during the fit.
- **spec_storage_key** (*str*) – The storage key of any spectrum that was used in this particular fit. The ObsID and instrument used don't matter, as the storage key will be the same and is based off of the settings when the spectra were generated.

get_results(*outer_radius*, *model*, *inner_radius=<Quantity 0. arcsec>*, *par=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Important method that will retrieve fit results from the source object. Either for a specific parameter of a given region-model combination, or for all of them. If a specific parameter is requested, all matching values from the fit will be returned in an N row, 3 column numpy array (column 0 is the value, column 1 is err-, and column 2 is err+). If no parameter is specified, the return will be a dictionary of such numpy arrays, with the keys corresponding to parameter names.

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **par** (*str*) – The name of the parameter you want a result for.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.

Returns The requested result value, and uncertainties.

get_luminosities(*outer_radius*, *model*, *inner_radius=<Quantity 0. arcsec>*, *lo_en=None*, *hi_en=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Get method for luminosities calculated from model fits to spectra associated with this source. Either for given energy limits (that must have been specified when the fit was first performed), or for all luminosities associated with that model. Luminosities are returned as a 3 column numpy array; the 0th column is the value, the 1st column is the err-, and the 2nd is err+.

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **model** (*str*) – The name of the fitted model that you're requesting the luminosities from (e.g. constant*tbabs*apec).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.

- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.

Returns The requested luminosity value, and uncertainties.

convert_radius(*radius*, *out_unit='deg'*)

A simple method to convert radii between different distance units, it automatically checks whether the requested conversion is possible, given available information. For instance it would fail if you requested a conversion from arcseconds to a proper distance if no redshift information were available.

Parameters

- **radius** (*Quantity*) – The radius to convert to a new unit.
- **out_unit** (*Unit/str*) – The unit to convert the input radius to.

Returns The converted radius

Return type Quantity

get_radius(*rad_name*, *out_unit='deg'*)

Allows a radius associated with this source to be retrieved in specified distance units. Note that physical distance units such as kiloparsecs may only be used if the source has redshift information.

Parameters

- **rad_name** (*str*) – The name of the desired radius, r200 for instance.
- **out_unit** (*Unit/str*) – An astropy unit, either a Unit instance or a string representation. Default is degrees.

Returns The desired radius in the desired units.

Return type Quantity

property num_pn_obs

Getter method that gives the number of PN observations.

Returns Integer number of PN observations associated with this source

Return type int

property num_mos1_obs

Getter method that gives the number of MOS1 observations.

Returns Integer number of MOS1 observations associated with this source

Return type int

property num_mos2_obs

Getter method that gives the number of MOS2 observations.

Returns Integer number of MOS2 observations associated with this source

Return type int

property instruments
A property of a source that details which instruments have valid data for which observations.

Returns A dictionary of ObsIDs and their associated valid instruments.

Return type Dict

property disassociated
Property that describes whether this source has had ObsIDs disassociated from it.

Returns A boolean flag, True means that ObsIDs/instruments have been removed, False means they haven't.

Return type bool

property disassociated_obs
Property that details exactly what data has been disassociated from this source, if any.

Returns Dictionary describing which instruments of which ObsIDs have been disassociated from this source.

Return type dict

disassociate_obs(*to_remove*)
Method that uses the supplied dictionary to safely remove data from the source. This data will no longer be used in any analyses, and would typically be removed because it is of poor quality, or doesn't contribute enough to justify its presence.

Parameters **to_remove** (*dict/str*) – A dictionary of observations to remove, either in the style of the source.instruments dictionary (with the top level keys being ObsIDs, and the lower levels being instrument names), or a string containing an ObsID.

property luminosity_distance
Tells the user the luminosity distance to the source if a redshift was supplied, if not returns None.

Returns The luminosity distance to the source, calculated using the cosmology associated with this source.

Return type Quantity

property angular_diameter_distance
Tells the user the angular diameter distance to the source if a redshift was supplied, if not returns None.

Returns The angular diameter distance to the source, calculated using the cosmology associated with this source.

Return type Quantity

property background_radius_factors
The factors by which to multiply outer radius by to get inner and outer radii for background regions.

Returns An array of the two factors.

Return type ndarray

obs_check(*reg_type*, *threshold_fraction=0.5*)
This method uses exposure maps and region masks to determine which ObsID/instrument combinations are not contributing to the analysis. It calculates the area intersection of the mask and exposure maps, and if (for a given ObsID-Instrument) the ratio of that area to the full area of the region calculated is less than the threshold fraction, that ObsID-instrument will be included in the returned rejection dictionary.

Parameters

- **reg_type** (*str*) – The region type for which to calculate the area intersection.
- **threshold_fraction** (*float*) – Intersection area/ full region area ratios below this value will mean an ObsID-Instrument is rejected.

Returns A dictionary of ObsID keys on the top level, then instruments a level down, that should be rejected according to the criteria supplied to this method.

Return type Dict

get_spectra(*outer_radius*, *obs_id=None*, *inst=None*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

A useful method that wraps the get_products function to allow you to easily retrieve XGA Spectrum objects. Simply pass the desired ObsID/instrument, and the same settings you used to generate the spectrum in evselect_spectrum, and the spectra(um) will be provided to you. If no match is found then a NoProductAvailableError will be raised.

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectrum (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied. The default is None, which means all spectra matching the other criteria will be returned.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied. The default is None, which means all spectra matching the other criteria will be returned.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectrum (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **group_spec** (*bool*) – Was the spectrum you wish to retrieve grouped?
- **min_counts** (*float*) – If the spectrum you wish to retrieve was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum you wish to retrieve was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum you wish to retrieve was over sampled, what was the level of over sampling used?

Returns An XGA Spectrum object (if there is an exact match), or a list of XGA Spectrum objects (if there were multiple matching products).

Return type Union[*Spectrum*, List[*Spectrum*]]

get_annular_spectra(*radii=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *set_id=None*)

Another useful method that wraps the get_products function, though this one gets you AnnularSpectra. Pass the radii used to generate the annuli, and the same settings you used to generate the spectrum in spectrum_set, and the AnnularSpectra will be returned (if it exists). If no match is found then a NoProductAvailableError will be raised. This method has an additional way of looking for a matching spectrum, if the set ID is known then that can be passed by the user and used to find an exact match.

Parameters

- **radii** (*Quantity*) – The annulus boundary radii that were used to generate the annular spectra set that you wish to retrieve. By default this is None, which means the method will return annular spectra with any radii.
- **group_spec** (*bool*) – Was the spectrum set you wish to retrieve grouped?
- **min_counts** (*float*) – If the spectrum set you wish to retrieve was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum set you wish to retrieve was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum set you wish to retrieve was over sampled, what was the level of over sampling used?
- **set_id** (*int*) – The unique identifier of the annular spectrum set. Passing a value for this parameter will override any other information that you have given this method.

Returns An XGA AnnularSpectra object if there is an exact match.

Return type *AnnularSpectra*

get_images(*obs_id=None*, *inst=None*, *lo_en=None*, *hi_en=None*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*)

A method to retrieve XGA Image objects. This supports the retrieval of both PSF corrected and non-PSF corrected images, as well as setting the energy limits of the specific image you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied. The default is None, which means all images matching the other criteria will be returned.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied. The default is None, which means all images matching the other criteria will be returned.
- **lo_en** (*Quantity*) – The lower energy limit of the image you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the image you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **psf_corr** (*bool*) – Sets whether you wish to retrieve a PSF corrected image or not.
- **psf_model** (*str*) – If the image you want is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the image you want is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the image you want is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the image you want is PSF corrected, this is the number of iterations.

Returns An XGA Image object (if there is an exact match), or a list of XGA Image objects (if there were multiple matching products).

Return type Union[*Image*, List[*Image*]]

get_expmaps(*obs_id=None*, *inst=None*, *lo_en=None*, *hi_en=None*)

A method to retrieve XGA ExpMap objects. This supports setting the energy limits of the specific exposure maps you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied. The default is None, which means all exposure maps matching the other criteria will be returned.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied. The default is None, which means all exposure maps matching the other criteria will be returned.
- **lo_en** (*Quantity*) – The lower energy limit of the exposure maps you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the exposure maps you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).

Returns An XGA ExpMap object (if there is an exact match), or a list of XGA ExpMap objects (if there were multiple matching products).

Return type Union[*ExpMap*, List[*ExpMap*]]

get_ratemaps(*obs_id=None*, *inst=None*, *lo_en=None*, *hi_en=None*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*)

A method to retrieve XGA RateMap objects. This supports the retrieval of both PSF corrected and non-PSF corrected ratemaps, as well as setting the energy limits of the specific ratemap you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied. The default is None, which means all ratemaps matching the other criteria will be returned.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied. The default is None, which means all ratemaps matching the other criteria will be returned.
- **lo_en** (*Quantity*) – The lower energy limit of the ratemaps you wish to retrieve, the default is None (which will retrieve all ratemaps regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the ratemaps you wish to retrieve, the default is None (which will retrieve all ratemaps regardless of energy limit).
- **psf_corr** (*bool*) – Sets whether you wish to retrieve a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the ratemap you want is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the ratemap you want is PSF corrected, this is the number of iterations.

Returns An XGA RateMap object (if there is an exact match), or a list of XGA RateMap objects (if there were multiple matching products).

Return type Union[*RateMap*, List[*RateMap*]]

get_combined_images(*lo_en=None*, *hi_en=None*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*)

A method to retrieve combined XGA Image objects, as in those images that have been created by merging all available data for this source. This supports the retrieval of both PSF corrected and non-PSF corrected images, as well as setting the energy limits of the specific image you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **lo_en** (*Quantity*) – The lower energy limit of the image you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the image you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **psf_corr** (*bool*) – Sets whether you wish to retrieve a PSF corrected image or not.
- **psf_model** (*str*) – If the image you want is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the image you want is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the image you want is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the image you want is PSF corrected, this is the number of iterations.

Returns An XGA Image object (if there is an exact match), or a list of XGA Image objects (if there were multiple matching products).

Return type Union[*Image*, List[*Image*]]

get_combined_expmaps(*lo_en=None*, *hi_en=None*)

A method to retrieve combined XGA ExpMap objects, as in those exposure maps that have been created by merging all available data for this source. This supports setting the energy limits of the specific exposure maps you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **lo_en** (*Quantity*) – The lower energy limit of the exposure maps you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the exposure maps you wish to retrieve, the default is None (which will retrieve all images regardless of energy limit).

Returns An XGA ExpMap object (if there is an exact match), or a list of XGA Image objects (if there were multiple matching products).

Return type Union[*ExpMap*, List[*ExpMap*]]

get_combined_ratemaps(*lo_en=None*, *hi_en=None*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*)

A method to retrieve combined XGA RateMap objects, as in those ratemap that have been created by merging all available data for this source. This supports the retrieval of both PSF corrected and non-PSF corrected ratemaps, as well as setting the energy limits of the specific ratemap you would like. A NoProductAvailableError error will be raised if no matches are found.

Parameters

- **lo_en** (*Quantity*) – The lower energy limit of the ratemaps you wish to retrieve, the default is None (which will retrieve all ratemaps regardless of energy limit).
- **hi_en** (*Quantity*) – The upper energy limit of the ratemaps you wish to retrieve, the default is None (which will retrieve all ratemaps regardless of energy limit).
- **psf_corr** (*bool*) – Sets whether you wish to retrieve a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the ratemap you want is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want is PSF corrected, this is the algorithm used.

- **psf_iter** (*int*) – If the ratemap you want is PSF corrected, this is the number of iterations.

Returns An XGA RateMap object (if there is an exact match), or a list of XGA RateMap objects (if there were multiple matching products).

Return type Union[*RateMap*, List[*RateMap*]]

get_profiles(*profile_type*, *obs_id=None*, *inst=None*, *central_coord=None*, *radii=None*, *lo_en=None*, *hi_en=None*)

This is the generic get method for XGA profile objects stored in this source. You still must remember the profile type value to use it, but once entered it will return a list of all matching profiles (or a single object if only one match is found).

Parameters

- **profile_type** (*str*) – The string profile type of the profile(s) you wish to retrieve.
- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied. The default is None, which means all profiles matching the other criteria will be returned.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied. The default is None, which means all profiles matching the other criteria will be returned.
- **central_coord** (*Quantity*) – The central coordinate of the profile you wish to retrieve, the default is None which means the method will use the default coordinate of this source.
- **radii** (*Quantity*) – The central radii of the profile points, it is not likely that this option will be used often as you likely won't know the radial values a priori.
- **lo_en** (*Quantity*) – The lower energy bound of the profile you wish to retrieve (if applicable), default is None, and if this argument is passed hi_en must be too.
- **hi_en** (*Quantity*) – The higher energy bound of the profile you wish to retrieve (if applicable), default is None, and if this argument is passed lo_en must be too.

Returns An XGA profile object (if there is an exact match), or a list of XGA profile objects (if there were multiple matching products).

Return type Union[*BaseProfile1D*, List[*BaseProfile1D*]]

get_combined_profiles(*profile_type*, *central_coord=None*, *radii=None*, *lo_en=None*, *hi_en=None*)

The generic get method for XGA profiles made using all available data which are stored in this source. You still must remember the profile type value to use it, but once entered it will return a list of all matching profiles (or a single object if only one match is found).

Parameters

- **profile_type** (*str*) – The string profile type of the profile(s) you wish to retrieve.
- **central_coord** (*Quantity*) – The central coordinate of the profile you wish to retrieve, the default is None which means the method will use the default coordinate of this source.
- **radii** (*Quantity*) – The central radii of the profile points, it is not likely that this option will be used often as you likely won't know the radial values a priori.
- **lo_en** (*Quantity*) – The lower energy bound of the profile you wish to retrieve (if applicable), default is None, and if this argument is passed hi_en must be too.
- **hi_en** (*Quantity*) – The higher energy bound of the profile you wish to retrieve (if applicable), default is None, and if this argument is passed lo_en must be too.

Returns An XGA profile object (if there is an exact match), or a list of XGA profile objects (if there were multiple matching products).

Return type Union[*BaseProfile1D*, List[*BaseProfile1D*]]

property fitted_models
This property cycles through all the available fit results, and finds the unique names of XSPEC models that have been fitted to this source.

Returns A list of model names.

Return type List[str]

snr_ranking(*outer_radius*, *lo_en=None*, *hi_en=None*, *allow_negative=False*)
This method generates a list of ObsID-Instrument pairs, ordered by the signal to noise measured for the given region, with element zero being the lowest SNR, and element N being the highest.

Parameters

- **outer_radius** (*Quantity/str*) – The radius that SNR should be calculated within, this can either be a named radius such as r500, or an astropy Quantity.
- **lo_en** (*Quantity*) – The lower energy bound of the ratemap to use to calculate the SNR. Default is None, in which case the lower energy bound for peak finding will be used (default is 0.5keV).
- **hi_en** (*Quantity*) – The upper energy bound of the ratemap to use to calculate the SNR. Default is None, in which case the upper energy bound for peak finding will be used (default is 2.0keV).
- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).

Returns Two arrays, the first an N by 2 array, with the ObsID, Instrument combinations in order of ascending signal to noise, then an array containing the order SNR ratios.

Return type Tuple[np.ndarray, np.ndarray]

info()
Very simple function that just prints a summary of important information related to the source object..

class xga.sources.base.**NullSource**(*obs=None*)
Bases: object

A useful, but very limited, source class. By default this source class will include all ObsIDs present in the XGA census, and as such can be used for bulk generation of SAS products. It can also be made to only include certain ObsIDs.

get_att_file(*obs_id*)
Fetches the path to the attitude file for an XMM observation.

Parameters **obs_id** – The ObsID to fetch the attitude file for.

Returns The path to the attitude file.

Return type str

property obs_ids
Property getter for ObsIDs associated with this source that are confirmed to have events files.

Returns A list of the associated XMM ObsIDs.

Return type List[str]

property instruments
A property of a source that details which instruments have valid data for which observations.

Returns A dictionary of ObsIDs and their associated valid instruments.

Return type Dict

update_queue(*cmd_arr*, *p_type_arr*, *p_path_arr*, *extra_info*, *stack=False*)
Small function to update the numpy array that makes up the queue of products to be generated.

Parameters

- **cmd_arr** (*np.ndarray*) – Array containing SAS commands.
- **p_type_arr** (*np.ndarray*) – Array of product type identifiers for the products generated by the cmd array. e.g. image or expmap.
- **p_path_arr** (*np.ndarray*) – Array of final product paths if cmd is successful
- **extra_info** (*np.ndarray*) – Array of extra information dictionaries
- **stack** – Should these commands be executed after a preceding line of commands, or at the same time.

Returns

get_queue()
Calling this indicates that the queue is about to be processed, so this function combines SAS commands along columns (command stacks), and returns N SAS commands to be run concurrently, where N is the number of columns.

Returns List of strings, where the strings are bash commands to run SAS procedures, another list of strings, where the strings are expected output types for the commands, a list of lists of strings, where the strings are expected output paths for products of the SAS commands.

Return type Tuple[List[str], List[str], List[List[str]]]

update_products(*prod_obj*)
This method will ONLY store images and exposure maps. Ideally I wouldn't store them as product objects at all, but unfortunately exposure maps require an image to be generated. Unlike all other source classes, ratemaps will not be generated when matching images and exposure maps are added.

Parameters **prod_obj** (BaseProduct) – The new product object to be added to the source object.

get_products(*p_type*, *obs_id=None*, *inst=None*, *extra_key=None*, *just_obj=True*)
This is the getter for the products data structure of Source objects. Passing a 'product type' such as 'events' or 'images' will return every matching entry in the products data structure.

Parameters

- **p_type** (*str*) – Product type identifier. e.g. image or expmap.
- **obs_id** (*str*) – Optionally, a specific obs_id to search can be supplied.
- **inst** (*str*) – Optionally, a specific instrument to search can be supplied.
- **extra_key** (*str*) – Optionally, an extra key (like an energy bound) can be supplied.
- **just_obj** (*bool*) – A boolean flag that controls whether this method returns just the product objects, or the other information that goes with it like ObsID and instrument.

Returns List of matching products.

Return type List[*BaseProduct*]

property name
The name of the source, either given at initialisation or generated from the user-supplied coordinates.

Returns The name of the source.

Return type str

property num_pn_obs
Getter method that gives the number of PN observations.

Returns Integer number of PN observations associated with this source

Return type int

property num_mos1_obs
Getter method that gives the number of MOS1 observations.

Returns Integer number of MOS1 observations associated with this source

Return type int

property num_mos2_obs
Getter method that gives the number of MOS2 observations.

Returns Integer number of MOS2 observations associated with this source

Return type int

info()
Just prints a couple of pieces of information about the NullSource

7.1.2 sources.general module

class xga.sources.general.**ExtendedSource**(*ra*, *dec*, *redshift=None*, *name=None*, *custom_region_radius=None*, *use_peak=True*, *peak_lo_en=<Quantity 0.5 keV>*, *peak_hi_en=<Quantity 2. keV>*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_products=True*, *load_fits=False*)

Bases: *xga.sources.base.BaseSource*

The general extended source XGA class, for extended X-ray sources that do not have a specific source for their astrophysical class. This class is subclassed by GalaxyCluster, which then adds more specific analyses, for instance.

find_peak(*rt*, *method='hierarchical'*, *num_iter=20*, *peak_unit=Unit('deg')*)
A method that will find the X-ray centroid for the RateMap that has been passed in. It takes the user supplied coordinates from source initialisation as a starting point, finds the peak within a 500kpc radius, re-centres the region, and iterates until the centroid converges to within 15kpc, or until 20 20 iterations has been reached.

Parameters

- **rt** (RateMap) – The ratemap which we want to find the peak (local to our user supplied coordinates) of.

- **method** (*str*) – Which peak finding method to use. Currently either hierarchical or simple can be chosen.
- **num_iter** (*int*) – How many iterations should be allowed before the peak is declared as not converged.
- **peak_unit** (*UnitBase*) – The unit the peak coordinate is returned in.

Returns The peak coordinate, a boolean flag as to whether the returned coordinates are near a chip gap/edge, and a boolean flag as to whether the peak converged. It also returns the coordinates of the points within the chosen point cluster, and a list of all point clusters that were not chosen.

Return type Tuple[Quantity, bool, bool, ndarray, List]

get_peaks(*obs_id=None*, *inst=None*)

A get method to return the peak of the X-ray emission of this GalaxyCluster.

Parameters

- **obs_id** (*str*) – The ObsID to return the X-ray peak coordinates for.
- **inst** (*str*) – The instrument to return the X-ray peak coordinates for.

Returns The X-ray peak coordinates for the input parameters.

Return type Quantity

get_1d_brightness_profile(*outer_rad*, *obs_id='combined'*, *inst='combined'*, *central_coord=None*, *radii=None*, *lo_en=None*, *hi_en=None*, *pix_step=1*, *min_snr=0.0*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*)

A specific get method for 1D brightness profiles. Should provide a relatively simple way of retrieving specific brightness profiles from XGA's storage system. Please note that there is not a separate get method for brightness profiles made from combined data, instead this method will search for combined profiles if either obs_id or inst is set to 'combined'. - Retrieving combined profiles is the default behaviour.

Parameters

- **outer_rad** (*Quantity/str*) – The outermost radius of the profile, either as a Quantity or a name (e.g. r500).
- **obs_id** (*str*) – The ObsID used to generate the profile in question, default is None. If this is set to combined then this method will search for profiles based on combined data.
- **inst** (*str*) – The instrument used to generate the profile in question, default is None. If this is set to combined then this method will search for profiles based on combined data.
- **central_coord** (*Quantity*) – The central coordinate from which the profile was generated. Default is None, which means we shall use the default coordinate of this source.
- **radii** (*Quantity*) – Specific radii to check for in the profiles.
- **lo_en** (*Quantity*) – The lower energy bound of the RateMap used to generate the profile.
- **hi_en** (*Quantity*) – The upper energy bound of the RateMap used to generate the profile.
- **pix_step** (*int*) – The width of each annulus in pixels used to generate the profile.
- **min_snr** (*float*) – The minimum signal to noise imposed upon the profile.
- **psf_corr** (*bool*) – Is the brightness profile corrected for PSF effects?
- **psf_model** (*str*) – If PSF corrected, the PSF model used.

- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.

Returns

property peak
A property getter for the combined X-ray peak coordinates. Most analysis will be centered on these coordinates.

Returns The X-ray peak coordinates for the combined ratemap.

Return type Quantity

property custom_radius
A getter for the custom region that can be defined on initialisation.

Returns The radius (in kpc) of the user defined custom region.

Return type Quantity

property point_clusters
This allows you to retrieve the point cluster positions from the hierarchical clustering peak finding method run on the combined ratemap. This includes both the chosen cluster and all others that were found.

Returns A numpy array of the positions of points of the chosen cluster (not galaxy cluster, a cluster of points). A list of numpy arrays with the same information for all the other clusters that were found

Return type Tuple[ndarray, List[ndarray]]

class xga.sources.general.**PointSource**(*ra*, *dec*, *redshift=None*, *name=None*, *point_radius=<Quantity 30. arcsec>*, *use_peak=False*, *peak_lo_en=<Quantity 0.5 keV>*, *peak_hi_en=<Quantity 2. keV>*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_products=True*, *load_fits=False*, *regen_merged=True*)

Bases: *xga.sources.base.BaseSource*

The general point source XGA class, for point X-ray sources that do not have a specific source for their astrophysical class.

property point_radius
Property getter to access the point_radius declared on initialisation of the source, the radius of the region that is used for point source analysis.

Returns The radius of the point source analysis region.

Return type Quantity

find_peak(*rt*, *peak_unit=Unit('deg')*)
Uses a simple 'brightest pixel' method to measure a peak coordinate for the point source.

Parameters

- **rt** (RateMap) – The RateMap to measure the peak from.
- **peak_unit** (*UnitBase*) – The desired output unit of the peak.

Returns The peak, and a boolean flag as to whether the peak is near an edge.

Return type Tuple[Quantity, bool]

property peak
A property getter for the combined X-ray peak coordinates.

Returns The X-ray peak coordinates for the combined ratemap.

Return type Quantity

7.1.3 sources.extended module

class xga.sources.extended.**GalaxyCluster**(*ra*, *dec*, *redshift*, *name=None*, *r200=None*, *r500=None*, *r2500=None*, *richness=None*, *richness_err=None*, *wl_mass=None*, *wl_mass_err=None*, *custom_region_radius=None*, *use_peak=True*, *peak_lo_en=<Quantity 0.5 keV>*, *peak_hi_en=<Quantity 2. keV>*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_products=True*, *load_fits=False*, *clean_obs=True*, *clean_obs_reg='r200'*, *clean_obs_threshold=0.3*, *regen_merged=True*)

Bases: *xga.sources.general.ExtendedSource*

This class is for the declaration and analysis of GalaxyCluster sources, and is a subclass of ExtendedSource.

property r200
Getter for the radius at which the average density is 200 times the critical density.

Returns The R200 in kpc.

Return type Quantity

property r500
Getter for the radius at which the average density is 500 times the critical density.

Returns The R500 in kpc.

Return type Quantity

property r2500
Getter for the radius at which the average density is 2500 times the critical density.

Returns The R2500 in kpc.

Return type Quantity

property weak_lensing_mass
Gets the weak lensing mass passed in at initialisation of the source.

Returns Two quantities, the weak lensing mass, and the weak lensing mass error in Msun. If the values were not passed in at initialisation, the returned values will be None.

Return type Quantity

property richness

Gets the richness passed in at initialisation of the source.

Returns Two floats, the richness, and the richness error. If the values were not passed in at initialisation, the returned values will be None.

Return type Quantity

get_results(*outer_radius*, *model='constant*tbabs*apec'*, *inner_radius=<Quantity 0. arcsec>*, *par=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Important method that will retrieve fit results from the source object. Either for a specific parameter of a given region-model combination, or for all of them. If a specific parameter is requested, all matching values from the fit will be returned in an N row, 3 column numpy array (column 0 is the value, column 1 is err-, and column 2 is err+). If no parameter is specified, the return will be a dictionary of such numpy arrays, with the keys corresponding to parameter names.

This overrides the BaseSource method, but the only difference is that this has a default model, which is what single_temp_apec fits (constant*tbabs*apec).

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **par** (*str*) – The name of the parameter you want a result for.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.

Returns The requested result value, and uncertainties.

get_luminosities(*outer_radius*, *model='constant*tbabs*apec'*, *inner_radius=<Quantity 0. arcsec>*, *lo_en=None*, *hi_en=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Get method for luminosities calculated from model fits to spectra associated with this source. Either for given energy limits (that must have been specified when the fit was first performed), or for all luminosities associated with that model. Luminosities are returned as a 3 column numpy array; the 0th column is the value, the 1st column is the err-, and the 2nd is err+.

This overrides the BaseSource method, but the only difference is that this has a default model, which is what single_temp_apec fits (constant*tbabs*apec).

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **model** (*str*) – The name of the fitted model that you're requesting the luminosities from (e.g. constant*tbabs*apec).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.

Returns The requested luminosity value, and uncertainties.

get_temperature(*outer_radius*, *model='constant*tbabs*apec'*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Convenience method that calls get_results to retrieve temperature measurements. All matching values from the fit will be returned in an N row, 3 column numpy array (column 0 is the value, column 1 is err-, and column 2 is err+).

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.

Returns The temperature value, and uncertainties.

get_proj_temp_profiles(*radii=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *set_id=None*)

A get method for projected temperature profiles generated by XGA's XSPEC interface. This works identically to the get_annular_spectra method, because projected temperature profiles are generated from annular spectra, and as such can be described by the same parameters.

Parameters

- **radii** (*Quantity*) – The annulus boundary radii that were used to generate the annular spectra set from which the projected temperature profile was measured.
- **group_spec** (*bool*) – Was the spectrum set used to generate the profile grouped
- **min_counts** (*float*) – If the spectrum set used to generate the profile was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum set used to generate the profile was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum set used to generate the profile was over sampled, what was the level of over sampling used?
- **set_id** (*int*) – The unique identifier of the annular spectrum set used to generate the profile. Passing a value for this parameter will override any other information that you have given this method.

Returns An XGA ProjectedGasTemperature1D object if there is an exact match, a list of such objects if there are multiple matches.

Return type Union[*ProjectedGasTemperature1D*, List[*ProjectedGasTemperature1D*]]

get_3d_temp_profiles(*radii=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *set_id=None*)

A get method for 3D temperature profiles generated by XGA's de-projection routines.

Parameters

- **radii** (*Quantity*) – The annulus boundary radii that were used to generate the annular spectra set from which the projected temperature profile was measured.
- **group_spec** (*bool*) – Was the spectrum set used to generate the profile grouped
- **min_counts** (*float*) – If the spectrum set used to generate the profile was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum set used to generate the profile was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum set used to generate the profile was over sampled, what was the level of over sampling used?
- **set_id** (*int*) – The unique identifier of the annular spectrum set used to generate the profile. Passing a value for this parameter will override any other information that you have given this method.

Returns An XGA ProjectedGasTemperature1D object if there is an exact match, a list of such objects if there are multiple matches.

Return type Union[*ProjectedGasTemperature1D*, List[*ProjectedGasTemperature1D*]]

get_apec_norm_profiles(*radii=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *set_id=None*)

A get method for APEC normalisation profiles generated by XGA's XSPEC interface.

Parameters

- **radii** (*Quantity*) – The annulus boundary radii that were used to generate the annular spectra set from which the normalisation profile was measured.
- **group_spec** (*bool*) – Was the spectrum set used to generate the profile grouped
- **min_counts** (*float*) – If the spectrum set used to generate the profile was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum set used to generate the profile was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum set used to generate the profile was over sampled, what was the level of over sampling used?
- **set_id** (*int*) – The unique identifier of the annular spectrum set used to generate the profile. Passing a value for this parameter will override any other information that you have given this method.

Returns An XGA APECNormalisation1D object if there is an exact match, a list of such objects if there are multiple matches.

Return type Union[*ProjectedGasTemperature1D*, List[*ProjectedGasTemperature1D*]]

get_density_profiles(*outer_rad=None*, *method=None*, *obs_id=None*, *inst=None*, *central_coord=None*, *radii=None*, *pix_step=1*, *min_snr=0.0*, *psf_corr=True*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *set_id=None*)

This is a get method for density profiles generated by XGA, both using surface brightness profiles and spectra. Having to account for two different methods is why this get method has so many arguments that can be passed. If multiple matches for the passed variables are found, then a list of density profiles will be returned, otherwise only a single profile will be returned.

Parameters

- **outer_rad** (*Quantity/str*) – The outer radius of the density profile, either as a name ('r500' for instance) or an astropy Quantity.
- **method** (*str*) – The method used to generate the density profile. For a profile created by fitting a model to a surface brightness profile this should be the name of the model, for a profile from annular spectra this should be 'spec', and for a profile generated directly from the data of a surface brightness profile this should be 'onion'.
- **obs_id** (*str*) – The ObsID used to generate the profile in question, default is None (which will search for profiles generated from combined data).
- **inst** (*str*) – The instrument used to generate the profile in question, default is None (which will search for profiles generated from combined data).
- **central_coord** (*Quantity*) – The central coordinate of the density profile. Default is None, which means we shall use the default coordinate of this source.
- **radii** (*Quantity*) – If known, the radii that were used to measure the density profile.
- **pix_step** (*int*) – The width of each annulus in pixels used to generate the profile, for profiles based on surface brightness.
- **min_snr** (*float*) – The minimum signal to noise imposed upon the profile, for profiles based on surface brightness.
- **psf_corr** (*bool*) – Is the brightness profile corrected for PSF effects, for profiles based on surface brightness.

- **psf_model** (*str*) – If PSF corrected, the PSF model used, for profiles based on surface brightness.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side, for profiles based on surface brightness.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used, for profiles based on surface brightness.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations, for profiles based on surface brightness.
- **group_spec** (*bool*) – Was the spectrum set used to generate the profile grouped.
- **min_counts** (*float*) – If the spectrum set used to generate the profile was grouped on minimum counts, what was the minimum number of counts?
- **min_sn** (*float*) – If the spectrum set used to generate the profile was grouped on minimum signal to noise, what was the minimum signal to noise.
- **over_sample** (*float*) – If the spectrum set used to generate the profile was over sampled, what was the level of over sampling used?
- **set_id** (*int*) – The unique identifier of the annular spectrum set used to generate the profile. Passing a value for this parameter will override any other information that you have given this method.

Returns

Return type Union[*GasDensity3D*, List[*GasDensity3D*]]

get_hydrostatic_mass_profiles(*temp_prof=None*, *temp_model_name=None*, *dens_prof=None*, *dens_model_name=None*, *radii=None*)

A get method for hydrostatic mass profiles associated with this galaxy cluster. This works in a slightly different way to the temperature and density profile get methods, as you can pass the gas temperature and density profiles used to generate a hydrostatic mass profile to find it. If none of the optional arguments are passed then all hydrostatic mass profiles associated with this source will be returned, if only some are passed then mass profiles which match the limited information will be found.

Parameters

- **temp_prof** (GasTemperature3D) – The temperature profile used to generate the required hydrostatic mass profile, default is None.
- **temp_model_name** (*str*) – The name of the model used to fit the temperature profile used to generate the required hydrostatic mass profile, default is None.
- **dens_prof** (GasDensity3D) – The density profile used to generate the required hydrostatic mass profile, default is None.
- **dens_model_name** (*str*) – The name of the model used to fit the density profile used to generate the required hydrostatic mass profile, default is None.
- **radii** (*Quantity*) – The radii at which the hydrostatic mass profile was measured, default is None.

Returns Either a single hydrostatic mass profile, when there is a unique match, or a list of hydrostatic mass profiles if there is not.

Return type Union[*HydrostaticMass*, List[*HydrostaticMass*]]

view_brightness_profile(*reg_type*, *central_coord=None*, *pix_step=1*, *min_snr=0.0*, *figsize=(10, 7)*, *xscale='log'*, *yscale='log'*, *back_sub=True*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*)

A method that generates and displays brightness profile objects for this galaxy cluster. Interloper sources are excluded, and any fits performed to pre-existing brightness profiles which are being viewed will also be displayed. The profile will be generated using a RateMap between the energy bounds specified by lo_en and hi_en.

Parameters

- **reg_type** (*str*) – The region in which to view the radial brightness profile.
- **central_coord** (*Quantity*) – The central coordinate of the brightness profile.
- **pix_step** (*int*) – The width (in pixels) of each annular bin, default is 1.
- **min_snr** (*float/int*) – The minimum signal to noise allowed for each radial bin. This is 0 by default, which disables any automatic re-binning.
- **figsize** (*tuple*) – The desired size of the figure, the default is (10, 7)
- **xscale** (*str*) – The scaling to be applied to the x axis, default is log.
- **yscale** (*str*) – The scaling to be applied to the y axis, default is log.
- **back_sub** (*bool*) – Should the plotted data be background subtracted, default is True.
- **lo_en** (*Quantity*) – The lower energy bound of the RateMap to generate the profile from.
- **hi_en** (*Quantity*) – The upper energy bound of the RateMap to generate the profile from.

combined_lum_conv_factor(*outer_radius*, *lo_en*, *hi_en*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*)

Combines conversion factors calculated for this source with individual instrument-observation spectra, into one overall conversion factor.

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius of the spectra that should be used to calculate conversion factors (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius of the spectra that should be used to calculate conversion factors (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **lo_en** (*Quantity*) – The lower energy limit of the conversion factors.
- **hi_en** (*Quantity*) – The upper energy limit of the conversion factors.
- **group_spec** (*bool*) – Whether the spectra that were used for fakeit were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were used for fakeit were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were used for fakeit were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were used for fakeit.

Returns A combined conversion factor that can be applied to a combined ratemap to calculate luminosity.

Return type Quantity

norm_conv_factor(*outer_radius*, *lo_en*, *hi_en*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *obs_id=None*, *inst=None*)

Combines count-rate to normalisation conversion factors associated with this source.

Parameters

- **outer_radius** (*str/Quantity*) – The name or value of the outer radius of the spectra that should be used to calculate conversion factors (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius of the spectra that should be used to calculate conversion factors (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **lo_en** (*Quantity*) – The lower energy limit of the conversion factors.
- **hi_en** (*Quantity*) – The upper energy limit of the conversion factors.
- **group_spec** (*bool*) – Whether the spectra that were used for fakeit were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were used for fakeit were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were used for fakeit were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were used for fakeit.
- **obs_id** (*str*) – The ObsID to fetch a conversion factor for, default is None which means the combined conversion factor will be returned.
- **inst** (*str*) – The instrument to fetch a conversion factor for, default is None which means the combined conversion factor will be returned.

Returns A combined conversion factor that can be applied to a combined ratemap to calculate luminosity.

Return type Quantity

7.1.4 sources.point module

7.2 samples

7.2.1 samples.base module

class xga.samples.base.**BaseSample**(*ra*, *dec*, *redshift=None*, *name=None*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_products=True*, *load_fits=False*, *no_prog_bar=False*)

Bases: object

The superclass for all sample classes. These store whole samples of sources, to make bulk analysis of interesting X-ray sources easy.

property names
Property getter for the list of source names in this sample.

Returns List of source names.

Return type list

property ra_decs
Property getter for the list of RA-DEC positions of the sources in this sample.

Returns List of source RA-DEC positions as supplied at sample initialisation.

Return type Quantity

property redshifts
Property getter for the list of redshifts of the sources in this sample (if available). If no redshifts were supplied, None will be returned.

Returns List of redshifts.

Return type ndarray

property nHs
Property getter for the list of nH values of the sources in this sample.

Returns List of nH values.

Return type Quantity

property cosmo
Property getter for the cosmology defined at initialisation of the sample. This cosmology is what is used for all analyses performed on the sample.

Returns The chosen cosmology.

property obs_ids
Property meant to inform the user about the number (and identities) of ObsIDs associated with the sources in a given sample.

Returns A dictionary (where the top level keys are the source names) of the ObsIDs associated with the

individual sources in this sample. :rtype: dict

property instruments
Property meant to inform the user about the number (and identities) of instruments associated with ObsIDs associated with the sources in a given sample.

Returns A dictionary (where the top level keys are the source names) of the instruments associated with

ObsIDs associated with the individual sources in this sample. :rtype: dict

property failed_names
Yields the names of those sources that could not be declared for some reason.

Returns A list of source names that could not be declared.

Return type List[str]

property failed_reasons
Returns a dictionary containing sources that failed to be declared successfully, and a simple reason why they couldn't be.

Returns A dictionary of source names as keys, and reasons as values.

Return type Dict[str, str]

Lx(*outer_radius*, *model*, *inner_radius=<Quantity 0. arcsec>*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *quality_checks=True*)

A get method for luminosities measured for the constituent sources of this sample. An error will be thrown if luminosities haven't been measured for the given region and model, no default model has been set, unlike the Tx method of ClusterSample. An extra condition that aims to only return 'good' data has been included, so that any Lx measurement with an uncertainty greater than value will be set to NaN, and a warning will be issued.

Parameters

- **model** (*str*) – The name of the fitted model that you're requesting the luminosities from (e.g. constant*tbabs*apec).
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). You may also pass a quantity containing radius values, with one value for each source in this sample.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **quality_checks** (*bool*) – Whether the quality checks to make sure a returned value is good enough to use should be performed.

Returns An Nx3 array Quantity where N is the number of sources. First column is the luminosity, second column is the -err, and 3rd column is the +err. If a fit failed then that entry will be NaN

Return type Quantity

check_spectra()

This method checks through the spectra associated with each source in the sample, printing a summary of which aren't usable and the reasons.

info()

Simple function to show basic information about the sample.

7.2.2 samples.general module

class xga.samples.general.**PointSample**(*ra*, *dec*, *redshift=None*, *name=None*, *point_radius=<Quantity 30. arcsec>*, *use_peak=False*, *peak_lo_en=<Quantity 0.5 keV>*, *peak_hi_en=<Quantity 2. keV>*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_fits=False*, *no_prog_bar=False*, *psf_corr=False*)

Bases: *xga.samples.base.BaseSample*

property point_radii

Property getter for the radii of the regions used for analysis of the point sources in this sample.

Returns A non-scalar Quantity of the point source radii used for analysis of the point sources in this sample.

Return type Quantity

property point_radii_unit

Property getter for the unit which the point radii values are stored in.

Returns The unit that the point radii are stored in.

Return type Unit

7.2.3 samples.extended module

class xga.samples.extended.**ClusterSample**(*ra*, *dec*, *redshift*, *name*, *r200=None*, *r500=None*, *r2500=None*, *richness=None*, *richness_err=None*, *wl_mass=None*, *wl_mass_err=None*, *custom_region_radius=None*, *use_peak=True*, *peak_lo_en=<Quantity 0.5 keV>*, *peak_hi_en=<Quantity 2. keV>*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *cosmology=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *load_fits=False*, *clean_obs=True*, *clean_obs_reg='r200'*, *clean_obs_threshold=0.3*, *no_prog_bar=False*, *psf_corr=False*)

Bases: *xga.samples.base.BaseSample*

A sample class to be used for declaring and analysing populations of galaxy clusters, with many cluster-science specific functions, such as the ability to create common scaling relations.

property r200_snr

Fetches and returns the R200 signal to noises from the constituent sources.

Returns The signal to noise ration calculated at the R200.

Return type np.ndarray

property r500_snr
Fetches and returns the R500 signal to noises from the constituent sources.

Returns The signal to noise ration calculated at the R500.

Return type np.ndarray

property r2500_snr
Fetches and returns the R2500 signal to noises from the constituent sources.

Returns The signal to noise ration calculated at the R2500.

Return type np.ndarray

property richness
Provides the richnesses of the clusters in this sample, if they were passed in on definition.

Returns A unitless Quantity object of the richnesses and their error(s).

Return type Quantity

property wl_mass
Provides the weak lensing masses of the clusters in this sample, if they were passed in on definition.

Returns A Quantity object of the WL masses and their error(s), in whatever units they were when

they were passed in originally. :rtype: Quantity

property r200
Returns all the R200 values passed in on declaration, but in units of kpc.

Returns A quantity of R200 values.

Return type Quantity

property r500
Returns all the R500 values passed in on declaration, but in units of kpc.

Returns A quantity of R500 values.

Return type Quantity

property r2500
Returns all the R2500 values passed in on declaration, but in units of kpc.

Returns A quantity of R2500 values.

Return type Quantity

Lx(*outer_radius*, *model='constant*tbabs*apec'*, *inner_radius=<Quantity 0. arcsec>*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *quality_checks=True*)
A get method for luminosities measured for the constituent sources of this sample. An error will be thrown if luminosities haven't been measured for the given region and model, no default model has been set, unlike the Tx method of ClusterSample. An extra condition that aims to only return 'good' data has been included, so that any Lx measurement with an uncertainty greater than value will be set to NaN, and a warning will be issued.

This overrides the BaseSample method, but the only difference is that this has a default model, which is what single_temp_apec fits (constant*tbabs*apec).

Parameters

- **model** (*str*) – The name of the fitted model that you're requesting the luminosities from (e.g. constant*tbabs*apec).
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). You may also pass a quantity containing radius values, with one value for each source in this sample.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **quality_checks** (*bool*) – Whether the quality checks to make sure a returned value is good enough to use should be performed.

Returns An Nx3 array Quantity where N is the number of sources. First column is the luminosity, second column is the -err, and 3rd column is the +err. If a fit failed then that entry will be NaN

Return type Quantity

Tx(*outer_radius='r500'*, *model='constant*tbabs*apec'*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *quality_checks=True*)

A get method for temperatures measured for the constituent clusters of this sample. An error will be thrown if temperatures haven't been measured for the given region (the default is R_500) and model (default is the constant*tbabs*apec model which single_temp_apec fits to cluster spectra). Any clusters for which temperature fits failed will return NaN temperatures, and with temperature greater than 25keV is considered failed, any temperature with a negative error value is considered failed, any temperature where the Tx-low err is less than zero isn't returned, and any temperature where one of the errors is more than three times larger than the other is considered failed (if quality checks are on).

Parameters

- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. You may also pass a quantity containing radius values, with one value for each source in this sample. The default for this method is r500.

- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **quality_checks** (*bool*) – Whether the quality checks to make sure a returned value is good enough to use should be performed.

Returns An Nx3 array Quantity where N is the number of clusters. First column is the temperature, second column is the -err, and 3rd column is the +err. If a fit failed then that entry will be NaN.

Return type Quantity

gas_mass(*rad_name*, *dens_model*, *method*, *prof_outer_rad=None*, *pix_step=1*, *min_snr=0.0*, *psf_corr=True*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *set_ids=None*, *quality_checks=True*)

A convenient get method for gas masses measured for the constituent clusters of this sample, though the arguments that can be passed to retrieve gas density profiles are limited to tone-down the complexity. Largely this method assumes you have measured density profiles from combined surface brightness profiles, though if you have generated density profiles from annular spectra and know their set ids you may pass them. Any more nuanced access to density profiles will have to be done yourself.

If the specified density model hasn't been fitted to the density profile, this method will run a fit using the default settings of the fit method of XGA profiles.

A gas mass will be set to NaN if either of the uncertainties are larger than the gas mass value, if the gas mass value is less than 1e+9 solar masses, if the gas mass value is greater than 1e+16 solar masses, if quality checks are on.

Parameters

- **rad_name** (*str*) – The name of the radius (e.g. r500) to calculate the gas mass within.
- **dens_model** (*str*) – The model fit to the density profiles to be used to calculate gas mass. If a fit doesn't already exist then one will be performed with default settings.
- **method** (*str*) – The method used to generate the density profile. For a profile created by fitting a model to a surface brightness profile this should be the name of the model, for a profile from annular spectra this should be 'spec', and for a profile generated directly from the data of a surface brightness profile this should be 'onion'.
- **prof_outer_rad** (*Quantity/str*) – The outer radii of the density profiles, either a single radius name or a Quantity containing an outer radius for each cluster. For instance if you defined a ClusterSample called srcs you could pass srcs.r500 here.
- **pix_step** (*int*) – The width of each annulus in pixels used to generate the profile, for profiles based on surface brightness.

- **min_snr** (*float*) – The minimum signal to noise imposed upon the profile, for profiles based on surface brightness.
- **psf_corr** (*bool*) – Is the brightness profile corrected for PSF effects, for profiles based on surface brightness.
- **psf_model** (*str*) – If PSF corrected, the PSF model used, for profiles based on surface brightness.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side, for profiles based on surface brightness.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used, for profiles based on surface brightness.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations, for profiles based on surface brightness.
- **set_ids** (*List[int]*) – A list of AnnularSpectra set IDs used to generate the density profiles, if you wish to use spectrum based density profiles here.
- **quality_checks** (*bool*) – Whether the quality checks to make sure a returned value is good enough to use should be performed.

Returns An Nx3 array Quantity where N is the number of clusters. First column is the gas mass, second column is the -err, and 3rd column is the +err. If a fit failed then that entry will be NaN.

Return type Quantity

hydrostatic_mass(*rad_name*, *temp_model_name=None*, *dens_model_name=None*, *quality_checks=True*)

A simple method for fetching hydrostatic masses of this sample of clusters. This function is limited, and if you have generated multiple hydrostatic mass profiles you may have to use the get_hydrostatic_mass_profiles function of each source directly, or use the returned profiles from the function that generated them.

If only one hydrostatic mass profile has been generated for each source, then you do not need to specify model names, but if the same temperature and density profiles have been used to make a hydrostatic mass profile but with different models then you may use them.

A mass will be set to NaN if either of the uncertainties are larger than the mass value, if the mass value is less than 1e+12 solar masses, if the mass value is greater than 1e+16 solar masses (if quality checks are on), or if no hydrostatic mass profile is available.

Parameters

- **rad_name** (*str*) – The name of the radius (e.g. r500) to calculate the hydrostatic mass within.
- **temp_model_name** (*str*) – The name of the model used to fit the temperature profile used to generate the required hydrostatic mass profile, default is None.
- **dens_model_name** (*str*) – The name of the model used to fit the density profile used to generate the required hydrostatic mass profile, default is None.
- **quality_checks** (*bool*) – Whether the quality checks to make sure a returned value is good enough to use should be performed.

Returns An Nx3 array Quantity where N is the number of clusters. First column is the hydrostatic mass, second column is the -err, and 3rd column is the +err. If a fit failed then that entry will be NaN.

Return type Quantity

gm_richness(*rad_name*, *dens_model*, *prof_outer_rad*, *dens_method*, *x_norm=<Quantity 60.>*, *y_norm=<Quantity 1.e+12 solMass>*, *fit_method='odr'*, *start_pars=None*, *pix_step=1*, *min_snr=0.0*, *psf_corr=True*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *set_ids=None*, *inv_efunc=False*)

This generates a Gas Mass vs Richness scaling relation for this sample of Galaxy Clusters.

Parameters

- **rad_name** (*str*) – The name of the radius (e.g. r500) to measure gas mass within.
- **dens_model** (*str*) – The model fit to the density profiles to be used to calculate gas mass. If a fit doesn't already exist then one will be performed with default settings.
- **prof_outer_rad** (*Quantity/str*) – The outer radii of the density profiles, either a single radius name or a Quantity containing an outer radius for each cluster. For instance if you defined a ClusterSample called srcs you could pas srcs.r500 here.
- **dens_method** (*str*) – The method used to generate the density profile. For a profile created by fitting a model to a surface brightness profile this should be the name of the model, for a profile from annular spectra this should be 'spec', and for a profile generated directly from the data of a surface brightness profile this should be 'onion'.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **pix_step** (*int*) – The width of each annulus in pixels used to generate the profile, for profiles based on surface brightness.
- **min_snr** (*float*) – The minimum signal to noise imposed upon the profile, for profiles based on surface brightness.
- **psf_corr** (*bool*) – Is the brightness profile corrected for PSF effects, for profiles based on surface brightness.
- **psf_model** (*str*) – If PSF corrected, the PSF model used, for profiles based on surface brightness.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side, for profiles based on surface brightness.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used, for profiles based on surface brightness.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations, for profiles based on surface brightness.
- **set_ids** (*List[int]*) – A list of AnnularSpectra set IDs used to generate the density profiles, if you wish to use spectrum based density profiles here.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

gm_Tx(*rad_name*, *dens_model*, *prof_outer_rad*, *dens_method*, *x_norm=<Quantity 4. keV>*, *y_norm=<Quantity 1.e+12 solMass>*, *fit_method='odr'*, *start_pars=None*, *model='constant*tbabs*apec'*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *pix_step=1*, *min_snr=0.0*, *psf_corr=True*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *set_ids=None*, *inv_efunc=False*)

This generates a Gas Mass vs Tx scaling relation for this sample of Galaxy Clusters.

Parameters

- **rad_name** (*str*) – The name of the radius (e.g. r500) to get values for.
- **dens_model** (*str*) – The model fit to the density profiles to be used to calculate gas mass. If a fit doesn't already exist then one will be performed with default settings.
- **prof_outer_rad** (*Quantity/str*) – The outer radii of the density profiles, either a single radius name or a Quantity containing an outer radius for each cluster. For instance if you defined a ClusterSample called srcs you could pas srcs.r500 here.
- **dens_method** (*str*) – The method used to generate the density profile. For a profile created by fitting a model to a surface brightness profile this should be the name of the model, for a profile from annular spectra this should be 'spec', and for a profile generated directly from the data of a surface brightness profile this should be 'onion'.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **model** (*str*) – The name of the model that the temperatures were measured with.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the Tx values were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the Tx values were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the Tx values were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **pix_step** (*int*) – The width of each annulus in pixels used to generate the profile, for profiles based on surface brightness.
- **min_snr** (*float*) – The minimum signal to noise imposed upon the profile, for profiles based on surface brightness.
- **psf_corr** (*bool*) – Is the brightness profile corrected for PSF effects, for profiles based on surface brightness.
- **psf_model** (*str*) – If PSF corrected, the PSF model used, for profiles based on surface brightness.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side, for profiles based on surface brightness.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used, for profiles based on surface brightness.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations, for profiles based on surface brightness.

- **set_ids** (*List[int]*) – A list of AnnularSpectra set IDs used to generate the density profiles, if you wish to use spectrum based density profiles here.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

Lx_richness(*outer_radius='r500'*, *x_norm=<Quantity 60.>*, *y_norm=<Quantity 1.e+44 erg / s>*, *fit_method='odr'*, *start_pars=None*, *model='constant*tbabs*apec'*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *inv_efunc=True*)

This generates a Lx vs richness scaling relation for this sample of Galaxy Clusters. If you have run fits to find core excised luminosity, and wish to use it in this scaling relation, then please don't forget to supply an inner_radius to the method call.

Parameters

- **outer_radius** (*str*) – The name of the radius (e.g. r500) to get values for.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **model** (*str*) – The name of the model that the luminosities were measured with.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the desired result (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

Lx_Tx(*outer_radius='r500'*, *x_norm=<Quantity 4. keV>*, *y_norm=<Quantity 1.e+44 erg / s>*, *fit_method='odr'*, *start_pars=None*, *model='constant*tbabs*apec'*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *tx_inner_radius=<Quantity 0. arcsec>*, *lx_inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *inv_efunc=True*)

This generates a Lx vs Tx scaling relation for this sample of Galaxy Clusters. If you have run fits to find core excised luminosity, and wish to use it in this scaling relation, then you can specify the inner radius of those spectra using lx_inner_radius, as well as ensuring that you use the temperature fit you want by setting tx_inner_radius.

Parameters

- **outer_radius** (*str*) – The name of the radius (e.g. r500) to get values for.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **model** (*str*) – The name of the model that the luminosities and temperatures were measured with.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **tx_inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the temperature (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **lx_inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the Lx. The same rules as tx_inner_radius apply, and this option is particularly useful if you have measured core-excised luminosity an wish to use it in a scaling relation.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

mass_Tx(*outer_radius='r500'*, *x_norm=<Quantity 4. keV>*, *y_norm=<Quantity 5.e+14 solMass>*, *fit_method='odr'*, *start_pars=None*, *model='constant*tbabs*apec'*, *tx_inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *temp_model_name=None*, *dens_model_name=None*, *inv_efunc=False*)

A convenience function to generate a hydrostatic mass-temperature relation for this sample of galaxy

clusters.

Parameters

- **outer_radius** (*str*) – The outer radius of the region used to measure temperature and the radius out to which you wish to measure mass.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **model** (*str*) – The name of the model that the luminosities and temperatures were measured with.
- **tx_inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the temperature (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. You may also pass a quantity containing radius values, with one value for each source in this sample.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **temp_model_name** (*str*) – The name of the model used to fit the temperature profile used to generate the required hydrostatic mass profile, default is None.
- **dens_model_name** (*str*) – The name of the model used to fit the density profile used to generate the required hydrostatic mass profile, default is None.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

mass_richness(*outer_radius='r500'*, *x_norm=<Quantity 60.>*, *y_norm=<Quantity 5.e+14 solMass>*, *fit_method='odr'*, *start_pars=None*, *temp_model_name=None*, *dens_model_name=None*, *inv_efunc=False*)

A convenience function to generate a hydrostatic mass-richness relation for this sample of galaxy clusters.

Parameters

- **outer_radius** (*str*) – The name of the radius (e.g. r500) to get values for.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.

- **temp_model_name** (*str*) – The name of the model used to fit the temperature profile used to generate the required hydrostatic mass profile, default is None.
- **dens_model_name** (*str*) – The name of the model used to fit the density profile used to generate the required hydrostatic mass profile, default is None.
- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

mass_Lx(*outer_radius='r500'*, *x_norm=<Quantity 1.e+44 erg / s>*, *y_norm=<Quantity 5.e+14 solMass>*, *fit_method='odr'*, *start_pars=None*, *model='constant*tbabs*apec'*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *lx_inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *temp_model_name=None*, *dens_model_name=None*, *inv_efunc=False*)

This generates a mass vs Lx scaling relation for this sample of Galaxy Clusters. If you have run fits to find core excised luminosity, and wish to use it in this scaling relation, then you can specify the inner radius of those spectra using lx_inner_radius.

Parameters

- **outer_radius** (*str*) – The name of the radius (e.g. r500) to get values for.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **fit_method** (*str*) – The name of the fit method to use to generate the scaling relation.
- **start_pars** (*list*) – The start parameters for the fit run.
- **model** (*str*) – The name of the model that the luminosities and temperatures were measured with.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.
- **lx_inner_radius** (*str/Quantity*) – The name or value of the inner radius that was used for the generation of the spectra which were fitted to produce the Lx. The same rules as tx_inner_radius apply, and this option is particularly useful if you have measured core-excised luminosity an wish to use it in a scaling relation.
- **group_spec** (*bool*) – Whether the spectra that were fitted for the desired result were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were fitted for the desired result were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were fitted for the desired result were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were fitted.
- **temp_model_name** (*str*) – The name of the model used to fit the temperature profile used to generate the required hydrostatic mass profile, default is None.
- **dens_model_name** (*str*) – The name of the model used to fit the density profile used to generate the required hydrostatic mass profile, default is None.

- **inv_efunc** (*bool*) – Should the inverse E(z) function be applied to the y-axis, if False then the non-inverse will be applied.

Returns The XGA ScalingRelation object generated for this sample.

Return type *ScalingRelation*

7.2.4 samples.point module

7.3 sas

7.3.1 sas.misc module

xga.sas.misc.**cifbuild**(*sources*, *num_cores=1*, *disable_progress=False*)

A wrapper for the XMM cifbuild command, which will be run before many of the more complex SAS commands, to check that a CIF compatible with the local version of SAS is available. The observation date is taken from an event list for a given ObsID, and the analysis date is set to the date which this function is run.

Parameters

- **sources** (*BaseSource/NullSource/BaseSample*) – A single source object, or a sample of sources.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

7.3.2 sas.phot module

xga.sas.phot.**evselect_image**(*sources*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *add_expr=''*, *num_cores=1*, *disable_progress=False*)

A convenient Python wrapper for a configuration of the SAS evselect command that makes images. Images will be generated for every observation associated with every source passed to this function. If images in the requested energy band are already associated with the source, they will not be generated again.

Parameters

- **sources** (*BaseSource/NullSource/BaseSample*) – A single source object, or a sample of sources.
- **lo_en** (*Quantity*) – The lower energy limit for the image, in astropy energy units.
- **hi_en** (*Quantity*) – The upper energy limit for the image, in astropy energy units.
- **add_expr** (*str*) – A string to be added to the SAS expression keyword
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

xga.sas.phot.**eexpmap**(*sources*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *num_cores=1*, *disable_progress=False*)

A convenient Python wrapper for the SAS eexpmap command. Expmaps will be generated for every observation

associated with every source passed to this function. If expmaps in the requested energy band are already associated with the source, they will not be generated again.

Parameters

- **sources** (*BaseSource/NullSource/BaseSample*) – A single source object, or sample of sources.
- **lo_en** (*Quantity*) – The lower energy limit for the expmap, in astropy energy units.
- **hi_en** (*Quantity*) – The upper energy limit for the expmap, in astropy energy units.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

xga.sas.phot.**emosaic**(*sources*, *to_mosaic*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_cores=1*, *disable_progress=False*)

A convenient Python wrapper for the SAS emosaic command. Every image associated with the source, that is in the energy band specified by the user, will be added together.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.
- **to_mosaic** (*str*) – The data type to produce a mosaic for, can be either image or expmap.
- **lo_en** (*Quantity*) – The lower energy limit for the combined image, in astropy energy units.
- **hi_en** (*Quantity*) – The upper energy limit for the combined image, in astropy energy units.
- **psf_corr** (*bool*) – If True, PSF corrected images will be mosaiced.
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

xga.sas.phot.**psfgen**(*sources*, *bins=4*, *psf_model='ELLBETA'*, *num_cores=1*, *disable_progress=False*)

A wrapper for the psfgen SAS task. Used to generate XGA PSF objects, which in turn can be used to correct XGA images/ratemaps for optical effects. By default we use the ELLBETA model reported in Read et al. 2011 (doi:10.1051/0004-6361/201117525), and generate a grid of binsxbins PSFs that can be used to correct for the PSF over an entire image. The energy dependence of the PSF is assumed to be minimal, and the resultant PSF object will be paired up with an image that matches it's ObsID and instrument.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.

- **bins** (*int*) – The image coordinate space will be divided into a grid of size binsxbins, PSFs will be generated at the central coordinates of the grid chunks.
- **psf_model** (*str*) – Which model to use when generating the PSF, default is ELLBETA, the best available.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

7.3.3 sas.run module

xga.sas.run.**execute_cmd**(*cmd*, *p_type*, *p_path*, *extra_info*, *src*)

This function is called for the local compute option, and runs the passed command in a Popen shell. It then creates an appropriate product object, and passes it back to the callback function of the Pool it was called from.

Parameters

- **cmd** (*str*) – SAS command to be executed on the command line.
- **p_type** (*str*) – The product type that will be produced by this command.
- **p_path** (*str*) – The final output path of the product.
- **extra_info** (*dict*) – Any extra information required to define the product object.
- **src** (*str*) – A string representation of the source object that this product is associated with.

Returns The product object, and the string representation of the associated source object.

Return type Tuple[*BaseProduct*, str]

xga.sas.run.**sas_call**(*sas_func*)

This is used as a decorator for functions that produce SAS command strings. Depending on the system that XGA is running on (and whether the user requests parallel execution), the method of executing the SAS command will change. This supports both simple multi-threading and submission with the Sun Grid Engine. :return:

7.3.4 sas.spec module

xga.sas.spec.**region_setup**(*sources*, *outer_radius*, *inner_radius*, *disable_progress*, *obs_id*)

The preparation and value checking stage for SAS spectrum generation.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectrum (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')).
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius to use for the generation of the spectrum (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

- **obs_id** (*str*) – Only used if the 'region' radius name is passed, the ObsID to retrieve the region for.

Returns The source objects, a list of inner radius quantities, and a list of outer radius quantities.

Return type Tuple[Union[*BaseSource*, *BaseSample*], List[Quantity], List[Quantity]]

xga.sas.spec.**evselect_spectrum**(*sources*, *outer_radius*, *inner_radius=<Quantity 0. arcsec>*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *num_cores=1*, *disable_progress=False*)

A wrapper for all of the SAS processes necessary to generate an XMM spectrum that can be analysed in XSPEC. Every observation associated with this source, and every instrument associated with that observation, will have a spectrum generated using the specified outer and inner radii as a boundary. The default inner radius is zero, so by default this function will produce circular spectra out to the outer_radius. It is possible to generate both grouped and ungrouped spectra using this function, with the degree of grouping set by the min_counts, min_sn, and oversample parameters.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectrum (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **inner_radius** (*str/Quantity*) – The name or value of the inner radius to use for the generation of the spectrum (for instance 'r500' would be acceptable for a GalaxyCluster, or Quantity(300, 'kpc')). By default this is zero arcseconds, resulting in a circular spectrum. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

xga.sas.spec.**spectrum_set**(*sources*, *radii*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *num_cores=1*, *force_regen=False*, *disable_progress=False*)

This function can be used to produce 'sets' of XGA Spectrum objects, generated in concentric circular annuli.

Such spectrum sets can be used to measure projected spectroscopic quantities, or even be de-projected to attempt to measure spectroscopic quantities in a three dimensional space.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.
- **radii** (*List[Quantity]/Quantity*) – A list of non-scalar quantities containing the boundary radii of the annuli for the sources. A single quantity containing at least three radii may be passed if one source is being analysed, but for multiple sources there should be a quantity (with at least three radii), PER source.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **force_regen** (*bool*) – This will force all the constituent spectra of the set to be regenerated, use this if your call to this function was interrupted and an incomplete AnnularSpectrum is being read in.
- **disable_progress** (*bool*) – Setting this to true will turn off the SAS generation progress bar.

7.4 xspec

7.4.1 xspec.fit package

Submodules

xspec.fit.general module

xga.xspec.fit.general.**single_temp_apec**(*sources, outer_radius, inner_radius=<Quantity 0. arcsec>, start_temp=<Quantity 3. keV>, start_met=0.3, lum_en=<Quantity [[5.e-01, 2.e+00], [1.e-02, 1.e+02]] keV>, freeze_nh=True, freeze_met=True, lo_en=<Quantity 0.3 keV>, hi_en=<Quantity 7.9 keV>, par_fit_stat=1.0, lum_conf=68.0, abund_table='angr', fit_method='leven', group_spec=True, min_counts=5, min_sn=None, over_sample=None, one_rmf=True, num_cores=1, spectrum_checking=True, timeout=<Quantity 1. h>*)

This is a convenience function for fitting an absorbed single temperature apec model(constant*tbabs*apec) to an object. It would be possible to do the exact same fit using the custom_model function, but as it will be a very common fit a dedicated function is in order. If there are no existing spectra with the passed settings, then they will be generated automatically.

If the spectrum checking step of the XSPEC fit is enabled (using the boolean flag spectrum_checking), then each individual spectrum available for a given source will be fitted, and if the measured temperature is less than or equal to 0.01keV, or greater than 20keV, or the temperature uncertainty is greater than 15keV, then that spectrum will be rejected and not included in the final fit. Spectrum checking also involves rejecting any spectra with fewer than 10 noticed channels.

Parameters

- **sources** (*List[BaseSource]*) – A single source object, or a sample of sources.
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are fitting for multiple sources then you can also pass a Quantity with one entry per source.
- **inner_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. By default this is zero arcseconds, resulting in a circular spectrum. If you are fitting for multiple sources then you can also pass a Quantity with one entry per source.
- **start_temp** (*Quantity*) – The initial temperature for the fit.
- **start_met** – The initial metallicity for the fit (in ZSun).
- **lum_en** (*Quantity*) – Energy bands in which to measure luminosity.
- **freeze_nh** (*bool*) – Whether the hydrogen column density should be frozen.
- **freeze_met** (*bool*) – Whether the metallicity parameter in the fit should be frozen.
- **lo_en** (*Quantity*) – The lower energy limit for the data to be fitted.
- **hi_en** (*Quantity*) – The upper energy limit for the data to be fitted.
- **par_fit_stat** (*float*) – The delta fit statistic for the XSPEC 'error' command, default is 1.0 which should be equivelant to 1 errors if I've understood (https://heasarc.gsfc.nasa.gov/xanadu/xspec/manual/XSerror.html) correctly.
- **lum_conf** (*float*) – The confidence level for XSPEC luminosity measurements.

- **abund_table** (*str*) – The abundance table to use for the fit.
- **fit_method** (*str*) – The XSPEC fit method to use.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **spectrum_checking** (*bool*) – Should the spectrum checking step of the XSPEC fit (where each spectrum is fit individually and tested to see whether it will contribute to the simultaneous fit) be activated?
- **timeout** (*Quantity*) – The amount of time each individual fit is allowed to run for, the default is one hour. Please note that this is not a timeout for the entire fitting process, but a timeout to individual source fits.

xga.xspec.fit.general.**power_law**(*sources, outer_radius, inner_radius=<Quantity 0. arcsec>, redshifted=False, lum_en=<Quantity [[5.e-01, 2.e+00], [1.e-02, 1.e+02]] keV>, start_pho_index=1.0, lo_en=<Quantity 0.3 keV>, hi_en=<Quantity 7.9 keV>, freeze_nh=True, par_fit_stat=1.0, lum_conf=68.0, abund_table='angr', fit_method='leven', group_spec=True, min_counts=5, min_sn=None, over_sample=None, one_rmf=True, num_cores=1, timeout=<Quantity 1. h>*)

This is a convenience function for fitting a tbabs absorbed powerlaw (or zpowerlw if redshifted is selected) to source spectra, with a multiplicative constant included to deal with different spectrum normalisations (constant*tbabs*powerlaw, or constant*tbabs*zpowerlw).

Parameters

- **sources** (*List[BaseSource]*) – A single source object, or a sample of sources.
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are fitting for multiple sources then you can also pass a Quantity with one entry per source.
- **inner_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. By default this is zero arcseconds, resulting in a circular spectrum. If you are fitting for multiple sources then you can also pass a Quantity with one entry per source.

- **redshifted** (*bool*) – Whether the powerlaw that includes redshift (zpowerlw) should be used.
- **lum_en** (*Quantity*) – Energy bands in which to measure luminosity.
- **start_pho_index** (*float*) – The starting value for the photon index of the powerlaw.
- **lo_en** (*Quantity*) – The lower energy limit for the data to be fitted.
- **hi_en** (*Quantity*) – The upper energy limit for the data to be fitted.
- **freeze_nh** (*bool*) – Whether the hydrogen column density should be frozen. :param start_pho_index:
- **par_fit_stat** (*float*) – The delta fit statistic for the XSPEC 'error' command, default is 1.0 which should be equivelant to 1sigma errors if I've understood (https://heasarc.gsfc.nasa.gov/xanadu/xspec /manual/XSerror.html) correctly.
- **lum_conf** (*float*) – The confidence level for XSPEC luminosity measurements.
- **abund_table** (*str*) – The abundance table to use for the fit.
- **fit_method** (*str*) – The XSPEC fit method to use.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **timeout** (*Quantity*) – The amount of time each individual fit is allowed to run for, the default is one hour. Please note that this is not a timeout for the entire fitting process, but a timeout to individual source fits.

xspec.fit.profile module

xga.xspec.fit.profile.**single_temp_apec_profile**(*sources, radii, start_temp=<Quantity 3. keV>, start_met=0.3, lum_en=<Quantity [[5.e-01, 2.e+00], [1.e-02, 1.e+02]] keV>, freeze_nh=True, freeze_met=True, lo_en=<Quantity 0.3 keV>, hi_en=<Quantity 7.9 keV>, par_fit_stat=1.0, lum_conf=68.0, abund_table='angr', fit_method='leven', group_spec=True, min_counts=5, min_sn=None, over_sample=None, one_rmf=True, num_cores=1, spectrum_checking=True, timeout=<Quantity 1. h>*)

A function that allows for the fitting of sets of annular spectra (generated from objects such as galaxy clusters) with an absorbed plasma emission model (tbabs*apec). This function fits the annuli completely independently of one another.

If the spectrum checking step of the XSPEC fit is enabled (using the boolean flag spectrum_checking), then each individual spectrum available for a given source will be fitted, and if the measured temperature is less than or equal to 0.01keV, or greater than 20keV, or the temperature uncertainty is greater than 15keV, then that spectrum will be rejected and not included in the final fit. Spectrum checking also involves rejecting any spectra with fewer than 10 noticed channels.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or a sample of sources.
- **radii** (*List[Quantity]/Quantity*) – A list of non-scalar quantities containing the boundary radii of the annuli for the sources. A single quantity containing at least three radii may be passed if one source is being analysed, but for multiple sources there should be a quantity (with at least three radii), PER source.
- **start_temp** (*Quantity*) – The initial temperature for the fit.
- **start_met** – The initial metallicity for the fit (in ZSun).
- **lum_en** (*Quantity*) – Energy bands in which to measure luminosity.
- **freeze_nh** (*bool*) – Whether the hydrogen column density should be frozen.
- **freeze_met** (*bool*) – Whether the metallicity parameter in the fit should be frozen.
- **lo_en** (*Quantity*) – The lower energy limit for the data to be fitted.
- **hi_en** (*Quantity*) – The upper energy limit for the data to be fitted.
- **par_fit_stat** (*float*) – The delta fit statistic for the XSPEC 'error' command, default is 1.0 which should be equivelant to 1 errors if I've understood (https://heasarc.gsfc.nasa.gov/xanadu/xspec/manual/XSerror.html) correctly.
- **lum_conf** (*float*) – The confidence level for XSPEC luminosity measurements.
- **abund_table** (*str*) – The abundance table to use for the fit.
- **fit_method** (*str*) – The XSPEC fit method to use.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.

- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.
- **spectrum_checking** (*bool*) – Should the spectrum checking step of the XSPEC fit (where each spectrum is fit individually and tested to see whether it will contribute to the simultaneous fit) be activated?
- **timeout** (*Quantity*) – The amount of time each individual fit is allowed to run for, the default is one hour. Please note that this is not a timeout for the entire fitting process, but a timeout to individual source fits.

7.4.2 xspec.fakeit module

xga.xspec.fakeit.**cluster_cr_conv**(*sources*, *outer_radius*, *inner_radius=<Quantity 0. arcsec>*, *sim_temp=<Quantity 3. keV>*, *sim_met=0.3*, *conv_en=<Quantity [[0.5, 2.]] keV>*, *abund_table='angr'*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *num_cores=1*)

This function uses the xspec fakeit tool to calculate conversion factors between count rate and luminosity for ARFs and RMFs associated with spectra in the given sources. Once complete the conversion factors are stored within the relevant XGA spectrum object. If the requested spectra do not already exist then they will automatically be generated for you. Please be aware that this function does not support calculating conversion factors from AnnularSpectra.

Parameters

- **sources** (GalaxyCluster) – The GalaxyCluster objects to calculate conversion factors for.
- **outer_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating factors for multiple sources then you can also pass a Quantity with one entry per source.
- **inner_radius** (*str/Quantity*) – The name or value of the outer radius of the region that the desired spectrum covers (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. By default this is zero arcseconds, resulting in a circular spectrum. If you are generating factors for multiple sources then you can also pass a Quantity with one entry per source.
- **sim_temp** (*Quantity*) – The temperature(s) to use for the apec model.
- **sim_met** (*float/List*) – The metallicity(s) (in solar met) to use for the apec model.

- **conv_en** (*Quantity*) – The energy limit pairs to calculate conversion factors for.
- **abund_table** (*str*) – The name of the XSPEC abundance table to use.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.

7.4.3 xspec.run module

xga.xspec.run.**execute_cmd**(*x_script*, *out_file*, *src*, *run_type*, *timeout*)

This function is called for the local compute option. It will run the supplied XSPEC script, then check parse the output for errors and check that the expected output file has been created.

Parameters

- **x_script** (*str*) – The path to an XSPEC script to be run.
- **out_file** (*str*) – The expected path for the output file of that XSPEC script.
- **src** (*str*) – A string representation of the source object that this fit is associated with.
- **run_type** (*str*) – A flag that tells this function what type of run this is; e.g. fit or conv_factors.
- **timeout** (*float*) – The length of time (in seconds) which the XSPEC script is allowed to run for before being killed.

Returns FITS object of the results, string repr of the source associated with this fit, boolean variable describing if this fit can be used, list of any errors found, list of any warnings found.

Return type Tuple[Union[FITS, str], str, bool, list, list]

xga.xspec.run.**xspec_call**(*xspec_func*)

This is used as a decorator for functions that produce XSPEC scripts. Depending on the system that XGA is running on (and whether the user requests parallel execution), the method of executing the XSPEC commands will change. This supports multi-threading. :return:

7.5 products

7.5.1 products.base module

class xga.products.base.**BaseProduct**(*path*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*)

Bases: object

The super class for all SAS generated products in XGA. Stores relevant file path information, parses the std_err output of the generation process, and stores the instrument and ObsID that the product was generated for.

property usable

Returns whether this product instance should be considered usable for an analysis.

Returns A boolean flag describing whether this product should be used.

Return type bool

property path

Property getter for the attribute containing the path to the product.

Returns The product path.

Return type str

parse_stderr()

This method parses the stderr associated with the generation of a product into errors confirmed to have come from SAS, and other unidentifiable errors. The SAS errors are returned with the actual error name, the error message, and the SAS routine that caused the error.

Returns A list of dictionaries containing parsed, confirmed SAS errors, another containing SAS warnings, and another list of unidentifiable errors that occured in the stderr.

Return type Tuple[List[Dict], List[Dict], List]

property sas_errors

Property getter for the confirmed SAS errors associated with a product.

Returns The list of confirmed SAS errors.

Return type List[Dict]

property sas_warnings

Property getter for the confirmed SAS warnings associated with a product.

Returns The list of confirmed SAS warnings.

Return type List[Dict]

raise_errors()

Method to raise the errors parsed from std_err string.

property obs_id

Property getter for the ObsID of this image. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

Returns The XMM ObsID of this image.

Return type str

property instrument

Property getter for the instrument used to take this image. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

Returns The XMM instrument used to take this image.

Return type str

property type
Property getter for the string identifier for the type of product this object is, mostly useful for internal methods of source objects.

Returns The string identifier for this type of object.

Return type str

property errors
Property getter for non-SAS errors detected during the generation of a product.

Returns A list of errors that aren't related to SAS.

Return type List[str]

property energy_bounds
Getter method for the energy_bounds property, which returns the rest frame energy band that this product was generated in.

Returns Tuple containing the lower and upper energy limits as Astropy quantities.

Return type Tuple[Quantity, Quantity]

property src_name
Method to return the name of the object a product is associated with. The product becomes aware of this once it is added to a source object.

Returns The name of the source object this product is associated with.

Return type str

property not_usable_reasons
Whenever the usable flag of a product is set to False (indicating you shouldn't use the product), a string indicating the reason is added to a list, which this property returns.

Returns A list of reasons why this product is unusable.

Return type List

property sas_command
A property that returns the original SAS command used to generate this object.

Returns String containing the command.

Return type str

class xga.products.base.**BaseAggregateProduct**(*file_paths*, *prod_type*, *obs_id*, *instrument*)
Bases: object

A base class for any XGA products that are an aggregate of an XGA SAS product.

property src_name
Method to return the name of the object a product is associated with. The product becomes aware of this once it is added to a source object. This is overridden in the AnnularSpectra class.

Returns The name of the source object this product is associated with.

Return type str

property obs_id
Property getter for the ObsID of this image. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

Returns The XMM ObsID of this image.

Return type str

property instrument
Property getter for the instrument used to take this image. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

Returns The XMM instrument used to take this image.

Return type str

property type
Property getter for the string identifier for the type of product this object is, mostly useful for internal methods of source objects.

Returns The string identifier for this type of object.

Return type str

property usable
Property getter for the boolean variable that tells you whether all component products have been found to be usable.

Returns Boolean variable, are all component products usable?

Return type bool

property energy_bounds
Getter method for the energy_bounds property, which returns the rest frame energy band that this product was generated in, if relevant.

Returns Tuple containing the lower and upper energy limits as Astropy quantities.

Return type Tuple[Quantity, Quantity]

property sas_errors
Equivelant to the BaseProduct sas_errors property, but reports any SAS errors stored in the component products.

Returns A list of SAS errors related to component products.

Return type List

property errors
Equivelant to the BaseProduct errors property, but reports any non-SAS errors stored in the component products.

Returns A list of non-SAS errors related to component products.

Return type List

property unprocessed_stderr
Equivelant to the BaseProduct sas_errors unprocessed_stderr, but returns a list of all the unprocessed standard error outputs.

Returns List of stderr outputs.

Return type List

class xga.products.base.**BaseProfile1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*, *x_norm=<Quantity 1.>*, *y_norm=<Quantity 1.>*)

Bases: object

The superclass for all 1D radial profile products, with built in fitting, viewing, and result retrieval functionality. Classes derived from BaseProfile1D can be added together to create Aggregate Profiles.

emcee_fit(*model*, *num_steps*, *num_walkers*, *progress_bar*, *show_warn*, *num_samples*)

A fitting function to fit an XGA model instance to the data in this profile using the emcee affine-invariant MCMC sampler, this should be called through .fit() for full functionality. An initial run of curve_fit is used to find start parameters for the sampler, though if that fails a maximum likelihood estimate is run, and if that fails the method will revert to using the start parameters set in the model instance.

Parameters

- **model** (BaseModel1D) – The model to be fit to the data.
- **num_steps** (*int*) – The number of steps each chain should take.
- **num_walkers** (*int*) – The number of walkers to be run for the ensemble sampler.
- **progress_bar** (*bool*) – Whether a progress bar should be displayed.
- **show_warn** (*bool*) – Should warnings be printed out, otherwise they are just stored in the model instance (this also happens if show_warn is True).
- **num_samples** (*int*) – The number of random samples to take from the posterior distributions of the model parameters.

Returns The model instance, and a boolean flag as to whether this was a successful fit or not.

Return type Tuple[*BaseModel1D*, bool]

nlls_fit(*model*, *num_samples*, *show_warn*)

A function to fit an XGA model instance to the data in this profile using the non-linear least squares curve_fit routine from scipy, this should be called through .fit() for full functionality

Parameters

- **model** (BaseModel1D) – An instance of the model to be fit to this profile.
- **num_samples** (*int*) – The number of random samples to be drawn and stored in the model parameter distribution property.
- **show_warn** (*bool*) – Should warnings be printed out, otherwise they are just stored in the model instance (this also happens if show_warn is True).

Returns The model (with best fit parameters stored within it), and a boolean flag as to whether the fit was successful or not.

Return type Tuple[*BaseModel1D*, bool]

fit(*model*, *method='mcmc'*, *num_samples=10000*, *num_steps=30000*, *num_walkers=20*, *progress_bar=True*, *show_warn=True*)

Method to fit a model to this profile's data, then store the resulting model parameter results. Each profile can store one instance of a type of model per fit method. So for instance you could fit both a 'beta' and 'double_beta' model to a surface brightness profile with curve_fit, and then you could fit 'double_beta' again with MCMC.

If any of the parameters of the passed model have a uniform prior associated, and the chosen method is curve_fit, then those priors will be used to place bounds on those parameters.

Parameters

- **model** (*str/BaseModel1D*) – Either an instance of an XGA model to be fit to this profile, or the name of a profile (e.g. 'beta', or 'simple_vikhlinin_dens').
- **method** (*str*) – The fit method to use, either 'curve_fit', 'mcmc', or 'odr'.

- **num_samples** (*int*) – The number of random samples to draw to create the parameter distributions that are saved in the model.
- **num_steps** (*int*) – Only applicable if using MCMC fitting, the number of steps each walker should take.
- **num_walkers** (*int*) – Only applicable if using MCMC fitting, the number of walkers to initialise for the ensemble sampler.
- **progress_bar** (*bool*) – Only applicable if using MCMC fitting, should a progress bar be shown.
- **show_warn** (*bool*) – Should warnings be printed out, otherwise they are just stored in the model instance (this also happens if show_warn is True).

Returns The fitted model object. The fitted model is also stored within the profile object.

Return type *BaseModel1D*

allowed_models(*table_format='fancy_grid'*)

This is a convenience function to tell the user what models can be used to fit a profile of the current type, what parameters are expected, and what the defaults are.

Parameters **table_format** (*str*) – The desired format of the allowed models table. This is passed to the tabulate module (allowed formats can be found here - https://pypi.org/project/tabulate/), and alters the way the printed table looks.

get_model_fit(*model*, *method*)

A get method for fitted model objects associated with this profile. Models for which the fit failed will also be returned, but a warning will be shown to inform the user that the fit failed.

Parameters

- **model** (*str*) – The name of the model to retrieve.
- **method** (*str*) – The method which was used to fit the model.

Returns An instance of an XGA model object that was fitted to this profile and updated with the parameter values.

Return type *BaseModel1D*

add_model_fit(*model*, *method*)

There are rare circumstances where XGA processes might wish to add a model to a profile from the outside, which is what this method allows you to do.

Parameters

- **model** (BaseModel1D) – The XGA model object to add to the profile.
- **method** (*str*) – The method used to fit the model.

get_sampler(*model*)

A get method meant to retrieve the MCMC ensemble sampler used to fit a particular model (supplied by the user). Checks are applied to the supplied model, to make sure that it is valid for the type of profile, that a good fit has actually been performed, and that the fit was performed with Emcee and not another method.

Parameters **model** (*str*) – The name of the model for which to retrieve the sampler.

Returns The Emcee sampler used to fit the user supplied model.

Return type em.EnsembleSampler

get_chains(*model*, *discard=True*, *flatten=True*, *thin=1*)
Get method for the sampler chains of an MCMC fit to the user supplied model. get_sampler is called to retrieve the sampler object, as well as perform validity checks on the model name.

Parameters

- **model** (*str*) – The name of the model for which to retrieve the chains.
- **discard** (*bool/int*) – Whether steps should be discarded for burn-in. If True then the cut off decided using the auto-correlation time will be used. If an integer is passed then this will be used as the number of steps to discard, and if False then no steps will be discarded.
- **flatten** (*bool*) – Should the chains of the multiple walkers be flattened into one chain per parameter.
- **thin** (*int*) – The thinning that should be applied to the chains. The default is 1, which means no thinning is applied.

Returns The requested chains.

Return type np.ndarray

view_chains(*model*, *discard=True*, *thin=1*, *figsize=None*)
Simple view method to quickly look at the MCMC chains for a given model fit.

Parameters

- **model** (*str*) – The name of the model for which to view the MCMC chains.
- **discard** (*bool/int*) – Whether steps should be discarded for burn-in. If True then the cut off decided using the auto-correlation time will be used. If an integer is passed then this will be used as the number of steps to discard, and if False then no steps will be discarded.
- **thin** (*int*) – The thinning that should be applied to the chains. The default is 1, which means no thinning is applied.
- **figsize** (*Tuple*) – Desired size of the figure, if None will be set automatically.

view_corner(*model*, *figsize=8, 8*)
A convenient view method to examine the corner plot of the parameter posterior distributions.

Parameters

- **model** (*str*) – The name of the model for which to view the corner plot.
- **figsize** (*Tuple*) – The desired figure size.

view_getdist_corner(*model*, *settings={}*, *figsize=10, 10*)
A view method to see a corner plot generated with the getdist module, using flattened chains with burn-in removed (whatever the getdist message might say).

Parameters

- **model** (*str*) – The name of the model for which to view the corner plot.
- **settings** (*dict*) – The settings dictionary for a getdist MCSample.
- **figsize** (*tuple*) – A tuple to set the size of the figure.

generate_data_realisations(*num_real*)
A method to generate random realisations of the data points in this profile, using their y-axis values and uncertainties. This can be useful for error propagation for instance, and does not require a model fit to work. This method assumes that the y-errors are 1-sigma, which isn't necessarily the case.

Parameters **num_real** (*int*) – The number of random realisations to generate.

Returns An N x R astropy quantity, where N is the number of realisations and R is the number of radii at which there are data points in this profile.

Return type Quantity

view(*figsize=10, 7*, *xscale='log'*, *yscale='log'*, *xlim=None*, *ylim=None*, *models=True*, *back_sub=True*, *just_models=False*, *custom_title=None*, *draw_rads={}*, *x_norm=False*, *y_norm=False*, *x_label=None*, *y_label=None*)

A method that allows us to view the current profile, as well as any models that have been fitted to it, and their residuals. The models are plotted by generating random model realisations from the parameter distributions, then plotting the median values, with 1sigma confidence limits.

Parameters

- **figsize** (*Tuple*) – The desired size of the figure, the default is (10, 7)
- **xscale** (*str*) – The scaling to be applied to the x axis, default is log.
- **yscale** (*str*) – The scaling to be applied to the y axis, default is log.
- **xlim** (*Tuple*) – The limits to be applied to the x axis, upper and lower, default is to let matplotlib decide by itself.
- **ylim** (*Tuple*) – The limits to be applied to the y axis, upper and lower, default is to let matplotlib decide by itself.
- **models** (*str*) – Should the fitted models to this profile be plotted, default is True
- **back_sub** (*bool*) – Should the plotted data be background subtracted, default is True.
- **just_models** (*bool*) – Should ONLY the fitted models be plotted? Default is False
- **custom_title** (*str*) – A plot title to replace the automatically generated title, default is None.
- **draw_rads** (*dict*) – A dictionary of extra radii (as astropy Quantities) to draw onto the plot, where the dictionary key they are stored under is what they will be labelled. e.g. ({'r500': Quantity(), 'r200': Quantity()}
- **x_norm** (*bool*) – Controls whether the x-axis of the profile is normalised by another value, the default is False, in which case no normalisation is applied. If it is set to True then it will attempt to use the internal normalisation value (which can be set with the x_norm property), and if a quantity is passed it will attempt to normalise using that.
- **y_norm** (*bool*) – Controls whether the y-axis of the profile is normalised by another value, the default is False, in which case no normalisation is applied. If it is set to True then it will attempt to use the internal normalisation value (which can be set with the y_norm property), and if a quantity is passed it will attempt to normalise using that.
- **x_label** (*str*) – Custom label for the x-axis (excluding units, which will be added automatically).
- **y_label** (*str*) – Custom label for the y-axis (excluding units, which will be added automatically).

save(*save_path=None*)

This method pickles and saves the profile object. This will be called automatically when the profile is initialised, and when changes are made to the profile (such as when a model is fitted). The save file is a pickled version of this object.

Parameters **save_path** (*str*) – The path where this profile should be saved. By default this is None, which means this method will use the save_path attribute of the profile.

property save_path
Property getter that assembles the default XGA save path of this profile. The file name contains limited information; the type of profile, the source name, and a random integer.

Returns The default XGA save path for this profile.

Return type str

property good_model_fits
A list of the names of models that have been successfully fitted to the profile.

Returns A list of model names.

Return type Dict

property radii
Getter for the radii passed in at init. These radii correspond to radii where the values were measured.

Returns Astropy quantity array of radii.

Return type Quantity

property radii_err
Getter for the uncertainties on the profile radii.

Returns Astropy quantity array of radii uncertainties, or a None value if no radii_err where passed.

Return type Quantity

property fit_radii
This property gives the user a sanitised set of radii that is safe to use for fitting to XGA models, by which I mean if the first element is zero (true for many of XGA's profiles), then it will be replaced by a value slightly above zero that won't cause divide by zeros in the fit process.

If the radius units are convertible to kpc then the zero value will be set to the equivelant of 1kpc, if they have pixel units then it will be set to one pixel, and if they are equivelant to degrees then it will be set to 1e5 degrees. The value for degrees is loosely based on the value of 1kpc at a redshift of 1.

Returns A Quantity with a set of radii that are 'safe' for fitting

Return type Quantity

property radii_unit
Getter for the unit of the radii passed by the user at init.

Returns An astropy unit object.

Return type Unit

property annulus_bounds
Getter for the original boundary radii of the annuli this profile may have been generated from. Only available if radii errors were passed on init.

Returns An astropy quantity containing the boundary radii of the annuli, or None if not available.

Return type Quantity

property values
Getter for the values passed by user at init.

Returns Astropy quantity array of values.

Return type Quantity

property values_err
Getter for uncertainties on the profile values.

Returns Astropy quantity array of values uncertainties, or a None value if no values_err where passed.

Return type Quantity

property values_unit
Getter for the unit of the values passed by the user at init.

Returns An astropy unit object.

Return type Unit

property background
Getter for the background associated with the profile values. If no background is set this will be zero.

Returns Astropy scalar quantity.

Return type Quantity

property centre
Property that returns the central coordinate that the profile was generated from.

Returns An astropy quantity of the central coordinate

Return type Quantity

property type
Getter for a string representing the type of profile stored in this object.

Returns String description of profile.

Return type str

property src_name
Getter for the name attribute of this profile, what source object it was derived from.

Returns

Return type object

property obs_id
Property getter for the ObsID this profile was made from. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

Returns XMM ObsID string.

Return type str

property instrument
Property getter for the instrument this profile was made from. Admittedly this information is implicit in the location this object is stored in a source object, but I think it worth storing directly as a property as well.

directly as a property as well. :return: XMM instrument name string. :rtype: str

property energy_bounds
Getter method for the energy_bounds property, which returns the rest frame energy band that this profile was generated from

Returns Tuple containing the lower and upper energy limits as Astropy quantities.

Return type Union[Tuple[Quantity, Quantity], Tuple[None, None]]

property set_ident
If this profile was generated from an annular spectrum, this will contain the set_id of that annular spectrum.

Returns The integer set ID of the annular spectrum that generated this, or None if it wasn't generated from an AnnularSpectra object.

Return type int

property y_axis_label
Property to return the name used for labelling the y-axis in any plot generated by a profile object.

Returns The y_axis label.

Return type str

property associated_set_storage_key
This property provides the storage key of the associated AnnularSpectra object, if the profile was generated from an AnnularSpectra. If it was not then a None value is returned.

Returns The storage key of the associated AnnularSpectra, or None if not applicable.

Return type str

property deg_radii
The radii in degrees if available.

Returns An astropy quantity containing the radii in degrees, or None.

Return type Quantity

property storage_key
This property returns the storage key which this object assembles to place the profile in an XGA source's storage structure. If the profile was generated from an AnnularSpectra then the key is based on the properties of the AnnularSpectra, otherwise it is based upon the properties of the specific profile.

Returns String storage key.

Return type str

property usable
Whether the profile object can be considered usable or not, reasons for this decision will vary for different profile types.

Returns A boolean variable.

Return type bool

property x_norm
The normalisation value for x-axis data passed on the definition of the this profile object.

Returns An astropy quantity containing the normalisation value.

Return type Quantity

property y_norm
The normalisation value for y-axis data passed on the definition of the this profile object.

Returns An astropy quantity containing the normalisation value.

Return type Quantity

property fit_options
Returns the supported fit options for XGA profiles.

Returns List of supported fit options.

Return type List[str]

property nice_fit_names
Returns nicer looking names for the supported fit options of XGA profiles.

Returns List of nice fit options.

Return type List[str]

property outer_radius
Property that returns the outer radius used for the generation of this profile.

Returns The outer radius used in the generation of the profile.

Return type Quantity

class xga.products.base.**BaseAggregateProfile1D**(*profiles*)
Bases: object

Quite a simple class that is generated when multiple 1D radial profile objects are added together. The purpose of instances of this class is simply to make it easy to view 1D radial profiles on the same axes.

property radii_unit
Getter for the unit of the radii passed by the user at init.

Returns An astropy unit object.

Return type Unit

property values_unit
Getter for the unit of the values passed by the user at init.

Returns An astropy unit object.

Return type Unit

property type
Getter for a string representing the type of profile stored in this object.

Returns String description of profile.

Return type str

property profiles
This property is for the constituent profiles that makes up this aggregate profile.

Returns A list of the profiles that make up this object.

Return type List[*BaseProfile1D*]

property energy_bounds
Getter method for the energy_bounds property, which returns the rest frame energy band that the component profiles of this object were generated from.

Returns Tuple containing the lower and upper energy limits as Astropy quantities.

Return type Union[Tuple[Quantity, Quantity], Tuple[None, None]]

property x_norms
The collated x normalisation values for the constituent profiles of this aggregate profile.

Returns A list of astropy quantities which represent the x-normalisations of the different profiles.

Return type List[Quantity]

property y_norms
The collated y normalisation values for the constituent profiles of this aggregate profile.

Returns A list of astropy quantities which represent the y-normalisations of the different profiles.

Return type List[Quantity]

view (*figsize=10, 7*, *xscale='log'*, *yscale='log'*, *xlim=None*, *ylim=None*, *model=None*, *back_sub=True*, *legend=True*, *just_model=False*, *custom_title=None*, *draw_rads={}*, *normalise_x=False*, *normalise_y=False*, *x_label=None*, *y_label=None*)

A method that allows us to see all the profiles that make up this aggregate profile, plotted on the same figure.

Parameters

- **figsize** (*Tuple*) – The desired size of the figure, the default is (10, 7)
- **xscale** (*str*) – The scaling to be applied to the x axis, default is log.
- **yscale** (*str*) – The scaling to be applied to the y axis, default is log.
- **xlim** (*Tuple*) – The limits to be applied to the x axis, upper and lower, default is to let matplotlib decide by itself.
- **ylim** (*Tuple*) – The limits to be applied to the y axis, upper and lower, default is to let matplotlib decide by itself.
- **model** (*str*) – The name of the model fit to display, default is None. If the model hasn't been fitted, or it failed, then it won't be displayed.
- **back_sub** (*bool*) – Should the plotted data be background subtracted, default is True.
- **legend** (*bool*) – Should a legend with source names be added to the figure, default is True.
- **just_model** (*bool*) – Should only the models, not the data, be plotted. Default is False.
- **custom_title** (*str*) – A plot title to replace the automatically generated title, default is None.
- **draw_rads** (*dict*) – A dictionary of extra radii (as astropy Quantities) to draw onto the plot, where the dictionary key they are stored under is what they will be labelled. e.g. ({'r500': Quantity(), 'r200': Quantity()}. If normalise_x option is also used, and the x-norm values are not the same for each profile, then draw_rads will be disabled.
- **normalise_x** (*bool*) – Should the x-axis values be normalised with the x_norm value passed on the definition of the constituent profile objects.
- **normalise_y** (*bool*) – Should the y-axis values be normalised with the y_norm value passed on the definition of the constituent profile objects.
- **x_label** (*str*) – Custom label for the x-axis (excluding units, which will be added automatically).
- **y_label** (*str*) – Custom label for the y-axis (excluding units, which will be added automatically).

7.5.2 products.misc module

class xga.products.misc.**EventList**(*path*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*)
Bases: *xga.products.base.BaseProduct*

7.5.3 products.phot module

class xga.products.phot.**Image**(*path*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*, *lo_en*, *hi_en*, *reg_file_path=''*)
Bases: *xga.products.base.BaseProduct*

This class stores image data from X-ray observations. It also allows easy, direct, access to that data, and implements many helpful methods with extra functionality (including coordinate transforms, peak finders, and a powerful view method).

property regions
Property getter for regions associated with this image.

Returns Returns a list of regions, if they have been associated with this object.

Return type List[PixelRegion]

property shape
Property getter for the resolution of the image. Standard XGA settings will make this 512x512.

Returns The shape of the numpy array describing the image.

Return type Tuple[int, int]

property data
Property getter for the actual image data, in the form of a numpy array. Doesn't include any of the other stuff you get in a fits image, thats found in the hdulist property.

Returns A numpy array of shape self.shape containing the image data.

Return type np.ndarray

property radec_wcs
Property getter for the WCS that converts back and forth between pixel values and RA-DEC coordinates. This one is the only WCS guaranteed to not-None.

Returns The WCS object for RA and DEC.

Return type wcs.WCS

property skyxy_wcs
Property getter for the WCS that converts back and forth between pixel values and XMM XY Sky coordinates.

Returns The WCS object for XMM X and Y sky coordinates.

Return type wcs.WCS

property detxy_wcs
Property getter for the WCS that converts back and forth between pixel values and XMM DETXY detector coordinates.

Returns The WCS object for XMM DETX and DETY detector coordinates.

Return type wcs.WCS

property header
Property getter allowing access to the astropy fits header object created when the image was read in.

Returns The header of the primary data table of the image that was read in.

Return type FITSHDR

coord_conv(*coords*, *output_unit*)

This will use the loaded WCSes, and astropy coordinates (including custom ones defined for this module), to perform common coordinate conversions for this product object.

Parameters

- **coords** (*Quantity*) – The input coordinates quantity to convert, in units of either deg, pix, xmm_sky, or xmm_det (xmm_sky and xmm_det are defined for this module).
- **output_unit** (*Unit/str*) – The astropy unit to convert to, can be either deg, pix, xmm_sky, or xmm_det (xmm_sky and xmm_det are defined for this module).

Returns The converted coordinates.

Return type Quantity

property psf_corrected

Tells the user (and XGA), whether an Image based object has been PSF corrected or not.

Returns Boolean flag, True means this object has been PSF corrected, False means it hasn't

Return type bool

property psf_algorithm

If this object has been PSF corrected, this property gives the name of the algorithm used.

Returns The name of the algorithm used to correct for PSF effects, or None if the object hasn't been PSF corrected.

Return type Union[str, None]

property psf_bins

If this object has been PSF corrected, this property gives the number of bins that the X and Y axes were divided into to generate the PSFGrid.

Returns The number of bins in X and Y for which PSFs were generated, or None if the object hasn't been PSF corrected.

Return type Union[int, None]

property psf_iterations

If this object has been PSF corrected, this property gives the number of iterations that the algorithm went through to create this image.

Returns The number of iterations the PSF correction algorithm went through, or None if the object hasn't been PSF corrected.

Return type Union[int, None]

property psf_model

If this object has been PSF corrected, this property gives the name of the PSF model used.

Returns The name of the PSF model used to correct for PSF effects, or None if the object hasn't been PSF corrected.

Return type Union[str, None]

get_view(*ax*, *cross_hair=None*, *mask=None*, *chosen_points=None*, *other_points=None*, *zoom_in=False*, *manual_zoom_xlims=None*, *manual_zoom_ylims=None*, *radial_bins_pix=array([], dtype=float64)*, *back_bin_pix=None*, *stretch=<astropy.visualization.stretch.LogStretch object>*, *mask_edges=True*, *view_regions=False*, *ch_thickness=0.8*)

The method that creates and populates the view axes, separate from actual view so outside methods can add a view to other matplotlib axes.

Parameters

- **ax** (*Axes*) – The matplotlib axes on which to show the image.
- **cross_hair** (*Quantity*) – An optional parameter that can be used to plot a cross hair at the coordinates. Up to two cross-hairs can be plotted, as any more can be visually confusing. If passing two, each row of a quantity is considered to be a separate coordinate pair.
- **mask** (*np.ndarray*) – Allows the user to pass a numpy mask and view the masked data if they so choose.
- **chosen_points** (*np.ndarray*) – A numpy array of a chosen point cluster from a hierarchical peak finder.
- **other_points** (*list*) – A list of numpy arrays of point clusters that weren't chosen by the hierarchical peak finder.
- **zoom_in** (*bool*) – Sets whether the figure limits should be set automatically so that borders with no data are reduced.
- **manual_zoom_xlims** (*tuple*) – If set, this will override the automatic zoom in and manually set a part of the x-axis to limit the image to, default is None. Pass a tuple with two elements, first being the lower limit, second the upper limit. Variable zoom_in must still be true for these limits to be applied.
- **manual_zoom_ylims** (*tuple*) – If set, this will override the automatic zoom in and manually set a part of the y-axis to limit the image to, default is None. Pass a tuple with two elements, first being the lower limit, second the upper limit. Variable zoom_in must still be true for these limits to be applied.
- **radial_bins_pix** (*np.ndarray*) – Radii (in units of pixels) of annuli to plot on top of the image, will only be triggered if a cross_hair coordinate is also specified and contains only one coordinate.
- **back_bin_pix** (*np.ndarray*) – The inner and outer radii (in pixel units) of the annulus used to measure the background value for a given profile, will only be triggered if a cross_hair coordinate is also specified and contains only one coordinate.
- **stretch** (*BaseStretch*) – The astropy scaling to use for the image data, default is log.
- **mask_edges** (*bool*) – If viewing a RateMap, this variable will control whether the chip edges are masked to remove artificially bright pixels, default is True.
- **view_regions** (*bool*) – If regions have been associated with this object (either on init or using the 'regions' property setter, should they be displayed. Default is False.
- **ch_thickness** (*float*) – The desired linewidth of the crosshair(s), can be useful to increase this in certain circumstances. Default is 0.8.

Returns A populated figure displaying the view of the data.

Return type Axes

view(*cross_hair=None*, *mask=None*, *chosen_points=None*, *other_points=None*, *figsize=(10*, *8)*, *zoom_in=False*, *manual_zoom_xlims=None*, *manual_zoom_ylims=None*, *radial_bins_pix=array([]*, *dtype=float64)*, *back_bin_pix=None*, *stretch=<astropy.visualization.stretch.LogStretch object>*, *mask_edges=True*, *view_regions=False*, *ch_thickness=0.8*)

Powerful method to view this Image/RateMap/Expmap, with different options that can be used for eye-balling and producing figures for publication.

Parameters

- **cross_hair** (*Quantity*) – An optional parameter that can be used to plot a cross hair at the coordinates. Up to two cross-hairs can be plotted, as any more can be visually confusing. If passing two, each row of a quantity is considered to be a separate coordinate pair.
- **mask** (*np.ndarray*) – Allows the user to pass a numpy mask and view the masked data if they so choose.
- **chosen_points** (*np.ndarray*) – A numpy array of a chosen point cluster from a hierarchical peak finder.
- **other_points** (*list*) – A list of numpy arrays of point clusters that weren't chosen by the hierarchical peak finder.
- **figsize** (*Tuple*) – Allows the user to pass a custom size for the figure produced by this method.
- **zoom_in** (*bool*) – Sets whether the figure limits should be set automatically so that borders with no data are reduced.
- **manual_zoom_xlims** (*tuple*) – If set, this will override the automatic zoom in and manually set a part of the x-axis to limit the image to, default is None. Pass a tuple with two elements, first being the lower limit, second the upper limit. Variable zoom_in must still be true for these limits to be applied.
- **manual_zoom_ylims** (*tuple*) – If set, this will override the automatic zoom in and manually set a part of the y-axis to limit the image to, default is None. Pass a tuple with two elements, first being the lower limit, second the upper limit. Variable zoom_in must still be true for these limits to be applied.
- **radial_bins_pix** (*np.ndarray*) – Radii (in units of pixels) of annuli to plot on top of the image, will only be triggered if a cross_hair coordinate is also specified and contains only one coordinate.
- **back_bin_pix** (*np.ndarray*) – The inner and outer radii (in pixel units) of the annulus used to measure the background value for a given profile, will only be triggered if a cross_hair coordinate is also specified and contains only one coordinate.
- **stretch** (*BaseStretch*) – The astropy scaling to use for the image data, default is log.
- **mask_edges** (*bool*) – If viewing a RateMap, this variable will control whether the chip edges are masked to remove artificially bright pixels, default is True.
- **view_regions** (*bool*) – If regions have been associated with this object (either on init or using the 'regions' property setter, should they be displayed. Default is False.
- **ch_thickness** (*float*) – The desired linewidth of the crosshair(s), can be useful to increase this in certain circumstances. Default is 0.8.

class xga.products.phot.**ExpMap**(*path*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*, *lo_en*, *hi_en*)

Bases: *xga.products.phot.Image*

A very simple subclass of the Image product class - designed to allow for easy interaction with exposure maps.

get_exp(*at_coord*)
A simple method that converts the given coordinates to pixels, then finds the exposure time at those coordinates.

Parameters **at_coord** (*Quantity*) – A pair of coordinates to find the exposure time for.

Returns The exposure time at the supplied coordinates.

Return type Quantity

class xga.products.phot.**RateMap**(*xga_image*, *xga_expmap*, *reg_file_path=''*)
Bases: *xga.products.phot.Image*

A very powerful class which allows interactions with 'RateMaps', though these are not directly generated by SAS, they are images divided by matching exposure maps, to provide a count rate image.

property shape
Property getter for the resolution of the ratemap. Standard XGA settings will make this 512x512.

Returns The shape of the numpy array describing the ratemap.

Return type Tuple[int, int]

property data
Property getter for ratemap data, overrides the method in the base Image class. This is because the ratemap class has a _construct_on_demand method that creates the ratemap data, which needs to be called instead of _read_on_demand.

Returns A numpy array of shape self.shape containing the ratemap data.

Return type np.ndarray

get_rate(*at_coord*)
A simple method that converts the given coordinates to pixels, then finds the rate (in photons per second) and returns it.

Parameters **at_coord** (*Quantity*) – A pair of coordinates to find the photon rate for.

Returns The photon rate at the supplied coordinates.

Return type Quantity

simple_peak(*mask*, *out_unit=Unit('deg')*)
Simplest possible way to find the position of the peak of X-ray emission in a ratemap. This method takes a mask in the form of a numpy array, which allows the user to mask out parts of the ratemap that shouldn't be searched (outside of a certain region, or within point sources for instance).

Parameters

- **mask** (*np.ndarray*) – A numpy array used to weight the data. It should be 0 for pixels that aren't to be searched, and 1 for those that are.
- **out_unit** (*UnitBase*) – The desired output unit of the peak coordinates, the default is degrees.

Returns An astropy quantity containing the coordinate of the X-ray peak of this ratemap (given the user's mask), in units of out_unit, as specified by the user. Also returned is a boolean flag that tells the caller if the peak is near a chip edge.

Return type Tuple[Quantity, bool]

clustering_peak(*mask*, *out_unit=Unit('deg')*, *top_frac=0.05*, *max_dist=5*, *clean_point_clusters=False*)
An experimental peak finding function that cuts out the top 5% (by default) of array elements (by value), and runs a hierarchical clustering algorithm on their positions. The motivation for this is that the cluster

peak will likely be contained in that top 5%, and the only other pixels that might be involved are remnants of poorly removed point sources. So when clusters have been formed, we can take the one with the most entries, and find the maximal pixel of that cluster. Should be consistent with simple_peak under ideal circumstances.

Parameters

- **mask** (*np.ndarray*) – A numpy array used to weight the data. It should be 0 for pixels that aren't to be searched, and 1 for those that are.
- **out_unit** (*UnitBase*) – The desired output unit of the peak coordinates, the default is degrees.
- **top_frac** (*float*) – The fraction of the elements (ordered in descending value) that should be used to generate clusters, and thus be considered for the cluster centre.
- **max_dist** (*float*) – The maximum distance criterion for the hierarchical clustering algorithm, in pixels.
- **clean_point_clusters** (*bool*) – If this is set to true then the point clusters which are not believed to host the peak pixel will be cleaned, meaning that if they have less than 4 pixels associated with them then they will be removed.

Returns An astropy quantity containing the coordinate of the X-ray peak of this ratemap (given the user's mask), in units of out_unit, as specified by the user. Finally, the coordinates of the points in the chosen cluster are returned, as is a list of all the coordinates of all the other clusters.

Return type Tuple[Quantity, bool]

convolved_peak(*mask*, *redshift*, *cosmology*, *out_unit=Unit('deg')*)

A very experimental peak finding algorithm, credit for the idea and a lot of the code in this function go to Lucas Porth. A radial profile (for instance a project king profile for clusters) is convolved with the ratemap, using a suitable radius for the object type (so for a cluster r might be ~1000kpc). As such objects that are similar to this profile will be boosted preferentially over objects that aren't, making it less likely that we accidentally select the peak brightness pixel from a point source remnant or something similar. The convolved image is then masked to only look at the area of interest, and the peak brightness pixel is found.

Parameters

- **mask** (*np.ndarray*) – A numpy array used to weight the data. It should be 0 for pixels that aren't to be searched, and 1 for those that are.
- **redshift** (*float*) – The redshift of the source that we wish to find the X-ray centroid of.
- **cosmology** – An astropy cosmology object.
- **out_unit** (*UnitBase*) – The desired output unit of the peak coordinates, the default is degrees.

Returns An astropy quantity containing the coordinate of the X-ray peak of this ratemap (given the user's mask), in units of out_unit, as specified by the user.

Return type Tuple[Quantity, bool]

near_edge(*coord*)

Uses the edge mask generated for RateMap objects to determine if the passed coordinates are near an edge/chip gap. If the coordinates are within +- 2 pixels of an edge the result will be true.

Parameters **coord** (*Quantity*) – The coordinates to check.

Returns A boolean flag as to whether the coordinates are near an edge.

Return type bool

signal_to_noise(*source_mask*, *back_mask*, *exp_corr=True*, *allow_negative=False*)
A signal to noise calculation method which takes information on source and background regions, then uses that to calculate a signal to noise for the source. This was primarily motivated by the desire to produce valid SNR values for combined data, where uneven exposure times across the combined field of view could cause issues with the usual approach of just summing the counts in the region images and scaling by area. This method can also measure signal to noises without exposure time correction.

Parameters

- **source_mask** (*np.ndarray*) – The mask which defines the source region, ideally with interlopers removed.
- **back_mask** (*np.ndarray*) – The mask which defines the background region, ideally with interlopers removed.
- **exp_corr** (*bool*) – Should signal to noises be measured with exposure time correction, default is True. I recommend that this be true for combined observations, as exposure time could change quite dramatically across the combined product.
- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).

Returns A signal to noise value for the source region.

Return type float

property edge_mask
Returns the edge mask calculated for this RateMap in the form of a numpy array

Returns A boolean numpy array in the same shape as the RateMap.

Return type ndarray

property sensor_mask
Returns the detector map calculated for this RateMap. Values of 1 mean on chip, values of 0 mean off chip.

Returns A boolean numpy array in the same shape as the RateMap.

Return type ndarray

property expmap_path
Similar to the path property, but for the exposure map that went into this ratemap.

Returns The exposure map path.

Return type str

property image
This property allows the user to access the input Image object for this ratemap.

Returns The input XGA Image object used to create this ratemap.

Return type *Image*

property expmap
This property allows the user to access the input ExpMap object for this ratemap.

Returns The input XGA ExpMap object used to create this ratemap.

Return type *ExpMap*

class xga.products.phot.**PSF**(*path*, *psf_model*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*)

Bases: *xga.products.phot.Image*

get_val(*at_coord*)

A simple method that converts the given coordinates to pixels, then finds the exposure time at those coordinates.

Parameters **at_coord** (*Quantity*) – A pair of coordinates to find the exposure time for.

Returns The exposure time at the supplied coordinates.

Return type Quantity

resample(*im_prod*, *half_side_length*)

This method resamples a psfgen created PSF image to the same scale as the passed Image object. This is very important because psfgen makes these PSF images with a standard pixel size of 1 arcsec x 1 arcsec, and it can't be changed when calling the routine. Thankfully, due to the wonders of WCS, it is possible to construct a new array with the same pixel size as a given image. Very important for when we want to deconvolve with an image and correct for the PSF.

Parameters

- **im_prod** (Image) –
- **half_side_length** (*Quantity*) –

Returns The resampled PSF.

Return type np.ndarray

property ra_dec

A property that fetches the RA-DEC that the PSF was generated at.

Returns An astropy quantity of the ra and dec that the PSF was generated at.

Return type Quantity

property model

This is the model that was used to generate this PSF.

Returns XMM SAS psfgen model name.

Return type str

class xga.products.phot.**PSFGrid**(*file_paths*, *bins*, *psf_model*, *x_bounds*, *y_bounds*, *obs_id*, *instrument*, *stdout_str*, *stderr_str*, *gen_cmd*)

Bases: *xga.products.base.BaseAggregateProduct*

property num_bins

Getter for the number of bins in X and Y that this PSFGrid has PSF objects for.

Returns The number of bins per side used to generate this PSFGrid

Return type int

property model

This is the model that was used to generate the component PSFs in this PSFGrid.

Returns XMM SAS psfgen model name.

Return type str

property x_bounds

The x lower (column 0) and x upper (column 1) bounds of the PSFGrid bins. :return: N x 2 numpy array, where N is the total number of PSFGrid bins. :rtype: np.ndarray

property y_bounds
The y lower (column 0) and y upper (column 1) bounds of the PSFGrid bins.

Returns N x 2 numpy array, where N is the total number of PSFGrid bins.

Return type np.ndarray

property grid_locs
A 3D quantity containing the central position of each PSF in the grid.

Returns A 3D Quantity

Return type

unload_data()
A convenience method that will iterate through the component PSFs of this object and remove their data from memory using the data property deleter. This ensures that, if the data needs to be accessed again, the call to .data will read in the PSFs and all will be well, hopefully.

7.5.4 products.profile module

class xga.products.profile.**SurfaceBrightness1D**(*rt*, *radii*, *values*, *centre*, *pix_step*, *min_snr*, *outer_rad*, *radii_err=None*, *values_err=None*, *background=None*, *pixel_bins=None*, *back_pixel_bin=None*, *ann_areas=None*, *deg_radii=None*, *min_snr_succeeded=True*)
Bases: *xga.products.base.BaseProfile1D*

This class provides an interface to radially symmetric X-ray surface brightness profiles of extended objects.

property pix_step
Property that returns the integer pixel step size used to generate the annuli that make up this profile.

Returns The pixel step used to generate the surface brightness profile.

Return type int

property min_snr
Property that returns minimum signal to noise value that was imposed upon this profile during generation.

Returns The minimum signal to noise value used to generate this profile.

Return type float

property psf_corrected
Tells the user (and XGA), whether the RateMap this brightness profile was generated from has been PSF corrected or not.

Returns Boolean flag, True means this object has been PSF corrected, False means it hasn't

Return type bool

property psf_algorithm
If the RateMap this brightness profile was generated from has been PSF corrected, this property gives the name of the algorithm used.

Returns The name of the algorithm used to correct for PSF effects, or None if there was no PSF correction.

Return type Union[str, None]

property psf_bins
If the RateMap this brightness profile was generated from has been PSF corrected, this property gives the number of bins that the X and Y axes were divided into to generate the PSFGrid.

Returns The number of bins in X and Y for which PSFs were generated, or None if the object

hasn't been PSF corrected. :rtype: Union[int, None]

property psf_iterations
If the RateMap this brightness profile was generated from has been PSF corrected, this property gives the number of iterations that the algorithm went through.

Returns The number of iterations the PSF correction algorithm went through, or None if there has been

no PSF correction. :rtype: Union[int, None]

property psf_model
If the RateMap this brightness profile was generated from has been PSF corrected, this property gives the name of the PSF model used.

Returns The name of the PSF model used to correct for PSF effects, or None if there has been no

PSF correction. :rtype: Union[str, None]

property min_snr_succeeded
If True then the minimum signal to noise re-binning that can be applied to surface brightness profiles by some functions was successful, if False then it failed and the profile with no re-binning is stored here.

Returns A boolean flag describing whether re-binning was successful or not.

Return type bool

property pixel_bins
The annuli radii used to generate this profile, assuming they were passed on initialisation, otherwise None.

Returns Numpy array containing the pixel bins used to measure this radial brightness profile.

Return type np.ndarray

property back_pixel_bin
The annulus used to measure the background for this profile, assuming they were passed on initialisation, otherwise None.

Returns Numpy array containing the pixel bin used to measure the background.

Return type np.ndarray

property areas
Returns the areas of the annuli used to make this profile as an astropy Quantity.

Returns Astropy non-scalar quantity containing the areas.

Return type Quantity

check_match(*rt*, *centre*, *pix_step*, *min_snr*, *outer_rad*)
A method for external use to check whether this profile matches the requested configuration of surface brightness profile, put here just because I imagine it'll be used in quite a few places.

Parameters

- **rt** (RateMap) – The RateMap to compare to this profile.
- **centre** (*Quantity*) – The central coordinate to compare to this profile.

- **pix_step** (*int*) – The width of each annulus in pixels to compare to this profile.
- **min_snr** (*float*) – The minimum signal to noise to compare to this profile.
- **outer_rad** (*Quantity*) – The outer radius to compare to this profile.

Returns Whether this profile matches the passed parameters or not.

Return type bool

class xga.products.profile.**GasMass1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *dens_method*, *associated_prof*, *radii_err=None*, *values_err=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

This class provides an interface to a cumulative gas mass profile of a Galaxy Cluster.

property density_method

Gives the user the method used to generate the density profile used to make this gas mass profile.

Returns The string describing the method

Return type str

property generation_profile

Provides the profile from which the density profile used to make this gas mass profile was measured. Either a surface brightness profile if measured using SB methods, or an APEC normalisation profile if inferred from annular spectra.

Returns The profile from which the density profile that made this profile was measured.

Return type Union[*SurfaceBrightness1D*, *APECNormalisation1D*]

class xga.products.profile.**GasDensity3D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *dens_method*, *associated_prof*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

This class provides an interface to a gas density profile of a galaxy cluster.

gas_mass(*model*, *outer_rad*, *conf_level=68.2*, *fit_method='mcmc'*)

A method to calculate and return the gas mass (with uncertainties). This method uses the model to generate a gas mass distribution (using the fit parameter distributions from the fit performed using the model), then measures the median mass, along with lower and upper uncertainties.

Parameters

- **model** (*str*) – The name of the model from which to derive the gas mass.
- **outer_rad** (*Quantity*) – The radius to measure the gas mass out to.
- **conf_level** (*float*) – The confidence level to use to calculate the mass errors
- **fit_method** (*str*) – The method that was used to fit the model, default is 'mcmc'.

Returns A Quantity containing three values (mass, -err, +err), and another Quantity containing the entire mass distribution from the whole realisation.

Return type Tuple[Quantity, Quantity]

property density_method

Gives the user the method used to generate this density profile.

Returns The string describing the method

Return type str

property generation_profile
Provides the profile from which this density profile was measured. Either a surface brightness profile if measured using SB methods, or an APEC normalisation profile if inferred from annular spectra.

Returns The profile from which the densities were measured.

Return type Union[*SurfaceBrightness1D*, *APECNormalisation1D*]

view_gas_mass_dist(*model*, *outer_rad*, *conf_level=68.2*, *figsize=8, 8*, *bins='auto'*, *colour='lightslategrey'*, *fit_method='mcmc'*)
A method which will generate a histogram of the gas mass distribution that resulted from the gas mass calculation at the supplied radius. If the mass for the passed radius has already been measured it, and the mass distribution, will be retrieved from the storage of this product rather than re-calculated.

Parameters

- **model** (*str*) – The name of the model from which to derive the gas mass.
- **outer_rad** (*Quantity*) – The radius within which to calculate the gas mass.
- **conf_level** (*float*) – The confidence level for the mass uncertainties, this doesn't affect the distribution, only the vertical lines indicating the measured value of gas mass.
- **colour** (*str*) – The desired colour of the histogram.
- **figsize** (*tuple*) – The desired size of the histogram figure.
- **bins** (*int/str*) – The argument to be passed to plt.hist, either a number of bins or a binning algorithm name.
- **fit_method** (*str*) – The method that was used to fit the model, default is 'mcmc'.

gas_mass_profile(*model*, *radii=None*, *deg_radii=None*, *fit_method='mcmc'*)
A method to calculate and return a gas mass profile.

Parameters

- **model** (*str*) – The name of the model from which to derive the gas mass.
- **radii** (*Quantity*) – The radii at which to measure gas masses. The default is None, in which case the radii at which this density profile has data points will be used.
- **deg_radii** (*Quantity*) – The equivelant radii to *radii* but in degrees, required for defining a profile. The default is None, but if custom radii are passed then this variable must be passed too.
- **fit_method** (*str*) – The method that was used to fit the model, default is 'mcmc'.

Returns A cumulative gas mass distribution.

Return type *GasMass1D*

class xga.products.profile.**ProjectedGasTemperature1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)
Bases: *xga.products.base.BaseProfile1D*

A profile product meant to hold a radial profile of projected X-ray temperature, as measured from a set of annular spectra by XSPEC. These are typically only defined by XGA methods.

class xga.products.profile.**APECNormalisation1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

A profile product meant to hold a radial profile of XSPEC normalisation, as measured from a set of annular spectra by XSPEC. These are typically only defined by XGA methods. This is a useful profile because it allows to not only infer 3D profiles of temperature and metallicity, but can also allow us to infer the 3D density profile.

gas_density_profile(*redshift*, *cosmo*, *abund_table='angr'*, *num_real=10000*, *sigma=1*, *num_dens=True*)

A method to calculate the gas density profile from the APEC normalisation profile, which in turn was measured from XSPEC fits of an AnnularSpectra. This method supports the generation of both number density and mass density profiles through the use of the num_dens keyword.

Parameters

- **redshift** (*float*) – The redshift of the source that this profile was generated from.
- **cosmo** – The chosen cosmology.
- **abund_table** (*str*) – The abundance table to used for the conversion from n_e x n_H to n_e^2 during density calculation. Default is the famous Anders & Grevesse table.
- **num_real** (*int*) – The number of data realisations which should be generated to infer density errors.
- **sigma** (*int*) – What sigma of error should the density profile be created with, the default is 1.
- **num_dens** (*bool*) – If True then a number density profile will be generated, otherwise a mass density profile

will be generated. :return: The gas density profile which has been calculated from the APEC normalisation profile. :rtype: GasDensity3D

emission_measure_profile(*redshift*, *cosmo*, *abund_table='angr'*, *num_real=100*, *sigma=2*)

A method to calculate the emission measure profile from the APEC normalisation profile, which in turn was measured from XSPEC fits of an AnnularSpectra.

Parameters

- **redshift** (*float*) – The redshift of the source that this profile was generated from.
- **cosmo** – The chosen cosmology.
- **abund_table** (*str*) – The abundance table to used for the conversion from n_e x n_H to n_e^2 during density calculation. Default is the famous Anders & Grevesse table.
- **num_real** (*int*) – The number of data realisations which should be generated to infer emission measure errors.
- **sigma** (*int*) – What sigma of error should the density profile be created with, the default is 2.

Returns

Return type

class xga.products.profile.**EmissionMeasure1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

A profile product meant to hold a radial profile of X-ray emission measure.

class xga.products.profile.**ProjectedGasMetallicity1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

A profile product meant to hold a radial profile of projected X-ray metallicities/abundances, as measured from a set of annular spectra by XSPEC. These are typically only defined by XGA methods.

class xga.products.profile.**GasTemperature3D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

A profile product meant to hold a 3D radial profile of X-ray temperature, as measured by some form of deprojection applied to a projected temperature profile

class xga.products.profile.**BaryonFraction**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)

Bases: *xga.products.base.BaseProfile1D*

A profile product which will hold a profile showing how the baryon fraction of a galaxy cluster changes with radius. These profiles are typically generated from a HydrostaticMass profile product instance.

class xga.products.profile.**HydrostaticMass**(*temperature_profile*, *temperature_model*, *density_profile*, *density_model*, *radii*, *radii_err*, *deg_radii*, *fit_method='mcmc'*, *num_walkers=20*, *num_steps=20000*, *num_samples=10000*, *show_warn=True*, *progress=True*)

Bases: *xga.products.base.BaseProfile1D*

A profile product which uses input GasTemperature3D and GasDensity3D profiles to generate a hydrostatic mass profile, which in turn can be used to measure the hydrostatic mass at a particular radius. In contrast to other profile objects, this one calculates the y values itself, as such any radii may be passed.

mass(*radius*, *conf_level=68.2*)

A method which will measure a hydrostatic mass and hydrostatic mass uncertainty within the given radius/radii. No corrections are applied to the values calculated by this method, it is just the vanilla hydrostatic mass.

If the models for temperature and density have analytical solutions to their derivative wrt to radius then those will be used to calculate the gradients at radius, but if not then a numerical method will be used for which dx will be set to radius/1e+6.

Parameters

- **radius** (*Quantity*) – An astropy quantity containing the radius/radii that you wish to calculate the mass within.
- **conf_level** (*float*) – The confidence level for the mass uncertainties, the default is 68.2% (~1).

Returns An astropy quantity containing the mass/masses, lower and upper uncertainties, and another containing the mass realisation distribution.

Return type Union[Quantity, Quantity]

view_mass_dist(*radius*, *conf_level=68.2*, *figsize=8, 8*, *bins='auto'*, *colour='lightslategrey'*)

A method which will generate a histogram of the mass distribution that resulted from the mass calculation at the supplied radius. If the mass for the passed radius has already been measured it, and the mass distribution, will be retrieved from the storage of this product rather than re-calculated.

Parameters

- **radius** (*Quantity*) – An astropy quantity containing the radius/radii that you wish to calculate the mass within.
- **conf_level** (*float*) – The confidence level for the mass uncertainties, the default is 68.2% (~1).
- **bins** (*int/str*) – The argument to be passed to plt.hist, either a number of bins or a binning algorithm name.
- **colour** (*str*) – The desired colour of the histogram.
- **figsize** (*tuple*) – The desired size of the histogram figure.

baryon_fraction(*radius*, *conf_level=68.2*)

A method to use the hydrostatic mass information of this profile, and the gas density information of the input gas density profile, to calculate a baryon fraction within the given radius.

Parameters

- **radius** (*Quantity*) – An astropy quantity containing the radius/radii that you wish to calculate the baryon fraction within.
- **conf_level** (*float*) – The confidence level for the mass uncertainties, the default is 68.2% (~1).

Returns An astropy quantity containing the baryon fraction, -ve error, and +ve error, and another quantity containing the baryon fraction distribution.

Return type Tuple[Quantity, Quantity]

view_baryon_fraction_dist(*radius*, *conf_level=68.2*, *figsize=8, 8*, *bins='auto'*, *colour='lightslategrey'*)

A method which will generate a histogram of the baryon fraction distribution that resulted from the mass calculation at the supplied radius. If the baryon fraction for the passed radius has already been measured it, and the baryon fraction distribution, will be retrieved from the storage of this product rather than re-calculated.

Parameters

- **radius** (*Quantity*) – An astropy quantity containing the radius/radii that you wish to calculate the baryon fraction within.
- **conf_level** (*float*) – The confidence level for the mass uncertainties, the default is 68.2% (~1).
- **bins** (*int/str*) – The argument to be passed to plt.hist, either a number of bins or a binning algorithm name.

- **figsize** (*tuple*) – The desired size of the histogram figure.
- **colour** (*str*) – The desired colour of the histogram.

baryon_fraction_profile()
A method which uses the baryon_fraction method to construct a baryon fraction profile at the radii of this HydrostaticMass profile. The uncertainties on the baryon fraction are calculated at the 1 level.

Returns An XGA BaryonFraction object.

Return type *BaryonFraction*

property temperature_profile
A method to provide access to the 3D temperature profile used to generate this hydrostatic mass profile.

Returns The input temperature profile.

Return type *GasTemperature3D*

property density_profile
A method to provide access to the 3D density profile used to generate this hydrostatic mass profile.

Returns The input density profile.

Return type *GasDensity3D*

property temperature_model
A method to provide access to the model that was fit to the temperature profile.

Returns The fit temperature model.

Return type *BaseModel1D*

property density_model
A method to provide access to the model that was fit to the density profile.

Returns The fit density profile.

Return type *BaseModel1D*

rad_check(*rad*)
Very simple method that prints a warning if the radius is outside the range of data covered by the density or temperature profiles.

Parameters **rad** (*Quantity*) – The radius to check.

class xga.products.profile.**Generic1D**(*radii*, *values*, *centre*, *source_name*, *obs_id*, *inst*, *y_axis_label*, *prof_type*, *radii_err=None*, *values_err=None*, *associated_set_id=None*, *set_storage_key=None*, *deg_radii=None*)
Bases: *xga.products.base.BaseProfile1D*

A 1D profile product meant to hold profiles which have been dynamically generated by XSPEC profile fitting of models that I didn't build into XGA. It can also be used to make arbitrary profiles using external data.

7.5.5 products.relation module

class xga.products.relation.**ScalingRelation**(*fit_pars*, *fit_par_errs*, *model_func*, *x_norm*, *y_norm*, *x_name*, *y_name*, *fit_method='unknown'*, *x_data=None*, *y_data=None*, *x_err=None*, *y_err=None*, *x_lims=None*, *odr_output=None*, *chains=None*, *relation_name=None*, *relation_author='XGA'*, *relation_year='2021'*, *relation_doi=''*, *scatter_par=None*, *scatter_chain=None*)

Bases: object

This class is designed to store all information pertaining to a scaling relation fit, either performed by XGA or from literature. It also aims to make creating publication quality plots simple and easy.

property pars

The parameters that describe this scaling relation, along with their uncertainties. They are in the order in which they are expected to be passed into the model function.

Returns A numpy array of the fit parameters and their uncertainties, first column are parameters, second column are uncertainties.

Return type np.ndarray

property model_func

Provides the model function used to fit this relation.

Returns The Python function of this relation's model.

property x_name

A string containing the name of the x-axis of this relation.

Returns A Python string containing the name.

Return type str

property y_name

A string containing the name of the x-axis of this relation.

Returns A Python string containing the name.

Return type str

property x_norm

The astropy quantity containing the x-axis normalisation used during fitting.

Returns An astropy quantity object.

Return type Quantity

property y_norm

The astropy quantity containing the y-axis normalisation used during fitting.

Returns An astropy quantity object.

Return type Quantity

property x_unit

The astropy unit object relevant to the x-axis of this relation.

Returns An Astropy Unit object.

Return type Unit

property y_unit
The astropy unit object relevant to the y-axis of this relation.

Returns An Astropy Unit object.

Return type Unit

property x_data
An astropy Quantity of the x-data used to fit this relation, or an empty quantity if that data is not available. The first column is the data, the second is the uncertainties.

Returns An Astropy Quantity object, containing the data and uncertainties.

Return type Quantity

property y_data
An astropy Quantity of the y-data used to fit this relation, or an empty quantity if that data is not available. The first column is the data, the second is the uncertainties.

Returns An Astropy Quantity object, containing the data and uncertainties.

Return type Quantity

property x_lims
If the user passed an x range in which the relation is valid on initialisation, then this will return those limits in the same units as the x-axis.

Returns A quantity containing upper and lower x limits, or None.

Return type Quantity

property fit_method
A descriptor for the fit method used to generate this scaling relation.

Returns A string containing the name of the fit method.

Return type str

property name
A property getter for the name of the relation, this may not be unique in cases where no name was passed on declaration.

Returns String containing the name of the relation.

Return type str

property author
A property getter for the author of the relation, if not from literature it will be XGA.

Returns String containing the name of the author.

Return type str

property year
A property getter for the year that the relation was created/published, if not from literature it will be the current year.

Returns String containing the year of publication/creation.

Return type str

property doi
A property getter for the doi of the original paper of the relation, if not from literature it will an empty string.

Returns String containing the doi.

Return type str

property scatter_par
A getter for the scatter information.

Returns The scatter parameter and its uncertainty. If no scatter information was passed on definition then this will return None.

Return type np.ndarray

property scatter_chain
A getter for the scatter information chain.

Returns The scatter chain. If no scatter information was passed on definition then this will return None.

Return type np.ndarray

property chains
Property getter for the parameter chains.

Returns The MCMC chains of the fit for this scaling relation, if they were passed. Otherwise None.

Return type np.ndarray

property par_names
Getter for the parameter names.

Returns The names of the model parameters.

Return type List

view_chains(*figsize=None*)
Simple view method to quickly look at the MCMC chains for a scaling relation fit.

Parameters **figsize** (*tuple*) – Desired size of the figure, if None will be set automatically.

view_corner(*figsize=10, 10*, *cust_par_names=None*, *colour='tab:gray'*, *save_path=None*)
A convenient view method to examine the corner plot of the parameter posterior distributions.

Parameters

- **figsize** (*tuple*) – The size of the figure.
- **cust_par_names** (*List[str]*) – A list of custom parameter names. If the names include LaTeX code do not include $$ math environment symbols - you may also need to pass a string literal (e.g. r"sigma"). Do not include an entry for a scatter parameter.
- **colour** (*List[str]*) – Colour for the contours, the default is tab:gray.
- **save_path** (*str*) – The path where the figure produced by this method should be saved. Default is None, in which case the figure will not be saved.

predict(*x_values*)
This method allows for the prediction of y values from this scaling relation, you just need to pass in an appropriate set of x values.

Parameters **x_values** (*Quantity*) – The x values to predict y values for.

Returns The predicted y values

Return type Quantity

view(*x_lims=None*, *log_scale=True*, *plot_title=None*, *figsize=10, 8*, *data_colour='black'*, *model_colour='grey'*, *grid_on=False*, *conf_level=90*, *custom_x_label=None*, *custom_y_label=None*, *fontsize=15*, *legend_fontsize=13*, *x_ticks=None*, *x_minor_ticks=None*, *y_ticks=None*, *y_minor_ticks=None*, *save_path=None*)

A method that produces a high quality plot of this scaling relation (including the data it is based upon, if available).

Parameters

- **x_lims** (*Quantity*) – If not set, this method will attempt to take appropriate limits from the x-data this relation is based upon, if that data is not available an error will be thrown.
- **log_scale** (*bool*) – If true then the x and y axes of the plot will be log-scaled.
- **plot_title** (*str*) – A custom title to be used for the plot, otherwise one will be generated automatically.
- **figsize** (*tuple*) – A custom figure size for the plot, default is (8, 8).
- **data_colour** (*str*) – The colour to use for the data points in the plot, default is black.
- **model_colour** (*str*) – The colour to use for the model in the plot, default is grey.
- **grid_on** (*bool*) – If True then a grid will be included on the plot. Default is True.
- **conf_level** (*int*) – The confidence level to use when plotting the model.
- **custom_x_label** (*str*) – Passing a string to this variable will override the x axis label of this plot, including the unit string.
- **custom_y_label** (*str*) – Passing a string to this variable will override the y axis label of this plot, including the unit string.
- **fontsize** (*float*) – The fontsize for axis labels.
- **legend_fontsize** (*float*) – The fontsize for text in the legend.
- **x_ticks** (*list*) – Customise which major x-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **x_minor_ticks** (*list*) – Customise which minor x-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **y_ticks** (*list*) – Customise which major y-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **y_minor_ticks** (*list*) – Customise which minor y-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **save_path** (*str*) – The path where the figure produced by this method should be saved. Default is None, in which case the figure will not be saved.

class xga.products.relation.**AggregateScalingRelation**(*relations*)

Bases: object

This class is akin to the BaseAggregateProfile class, in that it is the result of a sum of ScalingRelation objects. References to the component objects will be stored within the structure of this class, and it primarily exists to allow plots with multiple relations to be generated.

property relations

This returns the list of ScalingRelation instances that make up this aggregate scaling relation.

Returns A list of ScalingRelation instances.

Return type List[*ScalingRelation*]

property x_unit
The astropy unit object relevant to the x-axis of this relation.

Returns An Astropy Unit object.

Return type Unit

property y_unit
The astropy unit object relevant to the y-axis of this relation.

Returns An Astropy Unit object.

Return type Unit

view_corner(*figsize=10, 10*, *cust_par_names=None*, *contour_colours=None*, *save_path=None*)
A corner plot viewing method that will combine chains from all the relations that make up this aggregate scaling relation and display them using getdist.

Parameters

- **figsize** (*tuple*) – The size of the figure.
- **cust_par_names** (*List[str]*) – A list of custom parameter names. If the names include LaTeX code do not include $$ math environment symbols - you may also need to pass a string literal (e.g. r"sigma"). Do not include an entry for a scatter parameter.
- **contour_colours** (*List[str]*) – Custom colours for the contours, there should be one colour per scaling relation.
- **save_path** (*str*) – The path where the figure produced by this method should be saved. Default is None, in which case the figure will not be saved.

view(*x_lims=None*, *log_scale=True*, *plot_title=None*, *figsize=10, 8*, *colour_list=None*, *grid_on=False*, *conf_level=90*, *show_data=True*, *fontsize=15*, *legend_fontsize=13*, *x_ticks=None*, *x_minor_ticks=None*, *y_ticks=None*, *y_minor_ticks=None*, *save_path=None*)
A method that produces a high quality plot of the component scaling relations in this AggregateScalingRelation.

Parameters

- **x_lims** (*Quantity*) – If not set, this method will attempt to take appropriate limits from the x-data this relation is based upon, if that data is not available an error will be thrown.
- **log_scale** (*bool*) – If true then the x and y axes of the plot will be log-scaled.
- **plot_title** (*str*) – A custom title to be used for the plot, otherwise one will be generated automatically.
- **figsize** (*tuple*) – A custom figure size for the plot, default is (8, 8).
- **colour_list** (*list*) – A list of matplotlib colours to use as a custom colour cycle.
- **grid_on** (*bool*) – If True then a grid will be included on the plot. Default is True.
- **conf_level** (*int*) – The confidence level to use when plotting the model.
- **show_data** (*bool*) – Controls whether data points are shown on the view, as it can quickly become confusing with multiple relations on one axis.
- **fontsize** (*float*) – The fontsize for axis labels.
- **legend_fontsize** (*float*) – The fontsize for text in the legend.
- **x_ticks** (*list*) – Customise which major x-axis ticks and labels are on the figure, default is None in which case they are determined automatically.

- **x_minor_ticks** (*list*) – Customise which minor x-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **y_ticks** (*list*) – Customise which major y-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **y_minor_ticks** (*list*) – Customise which minor y-axis ticks and labels are on the figure, default is None in which case they are determined automatically.
- **save_path** (*str*) – The path where the figure produced by this method should be saved. Default is None, in which case the figure will not be saved.

7.5.6 products.spec module

class xga.products.spec.**Spectrum**(*path*, *rmf_path*, *arf_path*, *b_path*, *central_coord*, *inn_rad*, *out_rad*, *obs_id*, *instrument*, *grouped*, *min_counts*, *min_sn*, *over_sample*, *stdout_str*, *stderr_str*, *gen_cmd*, *region=False*, *b_rmf_path=''*, *b_arf_path=''*)

Bases: *xga.products.base.BaseProduct*

This class is the XGA product responsible for storing an individual spectrum. Various qualities that can be measured from it (X-ray luminosity for example) can be associated with an instance of this object, as well as conversion factors that can be calculated from XSPEC. If a model has been fitted then the data and model can be viewed.

property path

This method returns the path to the spectrum file of this object.

Returns The path to the spectrum file associated with this object.

Return type str

property rmf

This method returns the path to the RMF file of the main spectrum of this object.

Returns The path to the RMF file associated with the main spectrum of this object.

Return type str

property arf

This method returns the path to the ARF file of the main spectrum of this object.

Returns The path to the ARF file associated with the main spectrum of this object.

Return type str

property background

This method returns the path to the background spectrum.

Returns Path of the background spectrum.

Return type str

property background_rmf

This method returns the path to the background spectrum's RMF file.

Returns The path the the background spectrum's RMF.

Return type str

property background_arf

This method returns the path to the background spectrum's ARF file.

Returns The path the the background spectrum's ARF.

Return type str

property storage_key
This property returns the storage key which this object assembles to place the Spectrum in an XGA source's storage structure. The key is based on the properties of the spectrum, and some of the configuration options, and is basically human readable.

Returns String storage key.

Return type str

property central_coord
This property provides the central coordinates (RA-Dec) of the region that this spectrum was generated from.

Returns Astropy quantity object containing the central coordinate in degrees.

Return type Quantity

property shape
Returns the shape of the outer edge of the region this spectrum was generated from.

Returns The shape (either circular or elliptical).

Return type str

property inner_rad
Gives the inner radius (if circular) or radii (if elliptical - semi-major, semi-minor) of the region in which this spectrum was generated.

Returns The inner radius(ii) of the region.

Return type Quantity

property outer_rad
Gives the outer radius (if circular) or radii (if elliptical - semi-major, semi-minor) of the region in which this spectrum was generated.

Returns The outer radius(ii) of the region.

Return type Quantity

property grouped
A property stating whether SAS was told to group this spectrum during generation or not.

Returns Boolean variable describing whether the spectrum is grouped or not

Return type bool

property grouped_on
A property stating what metric this spectrum was grouped on.

Returns String representation of the metric this spectrum was grouped on (None if not grouped).

Return type str

property min_counts
A property stating the minimum number of counts allowed in a grouped channel.

Returns The integer minimum number of counts per grouped channel (if this spectrum was grouped on minimum numbers of counts).

Return type int

property min_sn
A property stating the minimum signal to noise allowed in a grouped channel.

Returns The minimum signal to noise per grouped channel (if this spectrum was grouped on minimum signal to noise).

Return type Union[float, int]

property over_sample
A property string stating the amount of oversampling applied by evselect during the spectrum generation process.

Returns Oversampling applied during generation

Return type float

property region
This property states whether this spectrum was generated directly from a region file region or not. If true then this isn't from any arbitrary radii or an overdensity radius, but instead directly from a source finder.

Returns A boolean flag describing if this is a region spectrum or not.

Return type bool

property annulus_ident
This property returns the integer identifier of which annulus in a set this Spectrum is, if it is part of a set.

Returns Integer annulus identifier, None if not part of a set.

Return type object

property set_ident
This property returns the random id of the spectrum set this is a part of.

Returns Set identifier, None if not part of a set.

Return type int

property exposure
Property that returns the spectrum exposure time used by XSPEC.

Returns Spectrum exposure time.

Return type Quantity

add_fit_data(*model*, *tab_line*, *plot_data*)
Method that adds information specific to a spectrum from an XSPEC fit to this object. This includes individual spectrum exposure and count rate, as well as calculated luminosities, and plotting information for data and model.

Parameters

- **model** (*str*) – String representation of the XSPEC model fitted to the data.
- **tab_line** – The line of the SPEC_INFO table produced by xga_extract.tcl that is relevant to this spectrum object.
- **plot_data** (*hdu.table.TableHDU*) – The PLOT{N} table in the file produced by xga_extract.tcl that is relevant to this spectrum object.

get_luminosities(*model*, *lo_en=None*, *hi_en=None*)
Returns the luminosities measured for this spectrum from a given model.

Parameters

- **model** – Name of model to fetch luminosities for.
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.

Returns Luminosity measurement, either for all energy bands, or the one requested with the energy limit parameters. Luminosity measurements are presented as three column numpy arrays, with column 0 being the value, column 1 being err-, and column 2 being err+.

get_rate(*model*)

Fetches the count rate for a particular model fitted to this spectrum.

Parameters **model** – The model to fetch count rate for.

Returns Count rate in counts per second.

Return type Quantity

add_conv_factors(*lo_ens*, *hi_ens*, *rates*, *lums*, *model*)

Method used to store countrate to luminosity conversion factors derived from fakeit spectra, as well as the actual countrate and luminosity measured in case the user wants to create a combined factor for multiple observations.

Parameters

- **lo_ens** (*np.ndarray*) – A numpy array of string representations of the lower energy bounds for the cntrate and luminosity measurements.
- **hi_ens** (*np.ndarray*) – A numpy array of string representations of the upper energy bounds for the cntrate and luminosity measurements.
- **rates** (*np.ndarray*) – A numpy array of the rates measured for this arf/rmf combination for the energy ranges specified in lo_ens and hi_end.
- **lums** (*np.ndarray*) – A numpy array of the luminosities measured for this arf/rmf combination for the energy ranges specified in lo_ens and hi_end.
- **model** (*str*) – The name of the model used to calculate this factor.

get_conv_factor(*lo_en*, *hi_en*, *model*)

Retrieves a conversion factor between count rate and luminosity for a given energy range, if one has been calculated.

Parameters

- **lo_en** (*Quantity*) – The lower energy bound for the desired conversion factor.
- **hi_en** (*Quantity*) – The upper energy bound for the desired conversion factor.
- **model** (*str*) – The model used to generate the desired conversion factor.

Returns The conversion factor, luminosity, and rate for the supplied model-energy combination.

Return type Tuple[Quantity, Quantity, Quantity]

get_plot_data(*model*)

Simply grabs the plot data dictionary for a given model, if the spectrum has had a fit performed on it.

Parameters **model** (*str*) –

Returns All information required to plot the data and model.

Return type dict

get_arf_data()

Reads in and returns the ARF effective areas for this spectrum.

Returns The mid point of the energy bins and their corresponding effective areas.

Return type Tuple[Quantity, Quantity]

view_arf (*figsize=(8, 6)*, *xscale='linear'*, *yscale='linear'*, *lo_en=<Quantity 0. keV>*, *hi_en=<Quantity 16. keV>*)
Plots the response curve for this spectrum.

Parameters

- **figsize** (*tuple*) – The desired size of the output figure.
- **xscale** (*str*) – The xscale to use for the plot.
- **yscale** (*str*) – The yscale to use for the plot.
- **lo_en** (*Quantity*) – The lower energy limit for the x-axis.
- **hi_en** (*Quantity*) – The upper energy limit for the y-axis.

view (*lo_en=<Quantity 0. keV>*, *hi_en=<Quantity 30. keV>*, *figsize=(8, 6)*)
Very simple method to plot the data/models associated with this Spectrum object, between certain energy limits.

Parameters

- **lo_en** (*Quantity*) – The lower energy limit from which to plot the spectrum.
- **hi_en** (*Quantity*) – The upper energy limit to plot the spectrum to.
- **figsize** (*Tuple*) – The desired size of the output figure.

class xga.products.spec.**AnnularSpectra**(*spectra*)
Bases: *xga.products.base.BaseAggregateProduct*

A class designed to hold a set of XGA spectra generated in concentric, circular annuli.

property src_name
Method to return the name of the object a product is associated with. The product becomes aware of this once it is added to a source object.

Returns The name of the source object this product is associated with.

Return type str

property central_coord
This property provides the central coordinates (RA-Dec) that this set of spectra was generated around.

Returns Astropy quantity object containing the central coordinate in degrees.

Return type Quantity

property num_annuli
A property getter for the number of annular spectra.

Returns The number of annular spectra associated with this product.

Return type int

background(*obs_id*, *inst*)
This method returns the path to the background spectrum for a particular ObsID and instrument. It is the background associated with the outermost annulus of this object.

Parameters

- **obs_id** (*str*) – The ObsID to get the background spectrum for.
- **inst** (*str*) – The instrument to get the background spectrum for.

Returns Path of the background spectrum.

Return type str

background_rmf(*obs_id*, *inst*)
This method returns the path to the background spectrum's RMF for a particular ObsID and instrument. It is the RMF of the background associated with the outermost annulus of this object.

Parameters

- **obs_id** (*str*) – The ObsID to get the background spectrum's RMF for.
- **inst** (*str*) – The instrument to get the background spectrum' RMF for.

Returns Path of the background spectrum RMF.

Return type str

background_arf(*obs_id*, *inst*)
This method returns the path to the background spectrum's ARF for a particular ObsID and instrument. It is the ARF of the background associated with the outermost annulus of this object.

Parameters

- **obs_id** (*str*) – The ObsID to get the background spectrum's ARF for.
- **inst** (*str*) – The instrument to get the background spectrum' ARF for.

Returns Path of the background spectrum ARF.

Return type str

property obs_ids
A property of this spectrum set that details which ObsIDs have contributed spectra to this object.

Returns A list of ObsIDs.

containing instruments associated with those ObsIDs. :rtype: dict

property instruments
A property of this spectrum set that details which ObsIDs and instruments have contributed spectra to this object.

Returns A dictionary of lists, with the top level keys being ObsIDs, and the lists

containing instruments associated with those ObsIDs. :rtype: dict

get_spectra(*annulus_ident*, *obs_id=None*, *inst=None*)
This is the getter for the spectra stored in the AnnularSpectra data storage structure. They can be retrieved based on annulus identifier, ObsID, and instrument.

Parameters

- **annulus_ident** (*int*) – The annulus identifier to retrieve spectra for.
- **obs_id** (*str*) – Optionally, a specific obs_id to search for can be supplied.
- **inst** (*str*) – Optionally, a specific instrument to search for can be supplied.

Returns List of matching spectra, or just a Spectrum object if one match is found.

Return type Union[List[*Spectrum*], *Spectrum*]

property all_spectra
Simple extra wrapper for get_spectra that allows the user to retrieve every single spectrum associated with this AnnularSpectra instance, for all annulus IDs.

Returns A list of every single spectrum associated with this object.

Return type List[*Spectrum*]

property radii
A property to return all the boundary radii of the constituent annuli.

Returns Astropy quantity of the radii.

Return type Quantity

property annulus_centres
Returns the centres of all the annuli, in the original units the radii were passed in with.

Returns An astropy quantity containing radii.

Return type Quantity

property proper_radii
A property to return the boundary radii of the constituent annuli in kpc. This has to be set using the setter first, otherwise the value is None.

Returns Astropy quantity of the proper radii.

Return type Quantity

property proper_annulus_centres
Returns the centres of all the annuli, in the units of proper radii which the user has to have set through the property setter.

Returns An astropy quantity containing radii, or None if no proper radii exist

Return type Quantity

property set_ident
This property returns the ID of this set of spectra.

Returns The integer ID of this set.

Return type int

property storage_key
This property returns the storage key which this object assembles to place the AnnularSpectra in an XGA source's storage structure. The key is based on the properties of the AnnularSpectra, and some of the configuration options, and is basically human readable.

Returns String storage key.

Return type str

property grouped
A property stating whether SAS was told to group the spectra in this set during generation or not.

Returns Boolean variable describing whether the spectra are grouped or not

Return type bool

property grouped_on
A property stating what metric the spectra in this set were grouped on.

Returns String representation of the metric the spectra were grouped on (None if not grouped).

Return type str

property min_counts
A property stating the minimum number of counts allowed in a grouped channel for the spectra in this set.

Returns The integer minimum number of counts per grouped channel (if these spectra were grouped on minimum numbers of counts).

Return type int

property min_sn
A property stating the minimum signal to noise allowed in a grouped channel for the spectra in this set.

Returns The minimum signal to noise per grouped channel (if these spectra were grouped on minimum signal to noise).

Return type Union[float, int]

property over_sample
A property string stating the amount of oversampling applied by evselect during the generation of the spectra in this set. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.

Returns Oversampling applied during generation.

Return type float

add_fit_data(*model*, *tab_line*, *lums*, *obs_order*)
An equivelant to the add_fit_data method built into all source objects. The final fit results and luminosities are housed in a storage structure within the AnnularSpectra, which makes sense because this is an aggregate product of all the relevant spectra, storing them just as source objects store spectra that don't exist in a spectrum set.

Parameters

- **model** (*str*) – The XSPEC definition of the model used to perform the fit. e.g. constant*tbabs*apec
- **tab_line** – A dictionary of table lines with fit data, the keys of the dictionary being the relevant annulus ID for the fit.
- **lums** (*dict*) – A dictionary of the luminosities measured during the fits, the keys of the outermost dictionary being annulus IDs, and the luminosity dictionaries being energy based.
- **obs_order** (*dict*) – A dictionary (with keys being annuli idents) of lists of lists describing the order the data is being passed, so that specific results can be related back to specific observations later (if applicable). The lists should be structured like [[obsid1, inst1], [obsid1, inst2], [obsid1, inst3]] for instance.

get_results(*annulus_ident*, *model*, *par=None*)
Important method that will retrieve fit results from the AnnularSpectra object. Either for a specific parameter of the supplied model combination, or for all of them. If a specific parameter is requested, all matching values from the fit will be returned in an N row, 3 column numpy array (column 0 is the value, column 1 is err-, and column 2 is err+). If no parameter is specified, the return will be a dictionary of such numpy arrays, with the keys corresponding to parameter names.

Parameters

- **annulus_ident** (*int*) – The annulus for which you wish to retrieve the fit results.
- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **par** (*str*) – The name of the parameter you want a result for.

Returns The requested result value, and uncertainties.

get_luminosities(*annulus_ident*, *model*, *lo_en=None*, *hi_en=None*)
This will retrieve luminosities of specific annuli from fits performed on this AnnularSpectra object. A model name must be supplied, and if a luminosity from a specific energy range is desired then lower and upper energy bounds may be passed.

Parameters

- **annulus_ident** (*int*) – The annulus for which you wish to retrieve the luminosities.
- **model** (*str*) – The name of the fitted model that you're requesting the results from (e.g. constant*tbabs*apec).
- **lo_en** (*Quantity*) – The lower energy limit for the desired luminosity measurement.
- **hi_en** (*Quantity*) – The upper energy limit for the desired luminosity measurement.

Returns The requested luminosity value, and uncertainties. If a specific energy range has been supplied then a quantity containing the value (col 1), -err (col 2), and +err (col 3), will be returned. If no energy range is supplied then a dictionary of all available luminosity quantities will be returned.

Return type Union[Quantity, Dict[str, Quantity]]

generate_profile(*model*, *par*, *par_unit*, *upper_limit=None*)

This generates a radial profile of the requested fit parameter using the stored results from an XSPEC model fit run on this AnnularSpectra. The profile is added to AnnularSpectra internal storage, and also returned to the user.

Parameters

- **model** (*str*) – The name of the fitted model you wish to generate a profile from.
- **par** (*str*) – The name of the free model parameter that you wish to generate a profile for.
- **par_unit** (*Unit/str*) – The unit of the free model parameter as an astropy unit object, or a string representation (e.g. keV).
- **upper_limit** (*Quantity*) – Allows an allowed upper limit for the y values in the profile to be passed.

Returns The requested profile object.

Return type Union[*BaseProfile1D*, *ProjectedGasTemperature1D*, *ProjectedGasMetallicity1D*]

view_annulus(*ann_ident*, *model*, *figsize=12, 8*)

An equivelant to the Spectrum view method, but allows all spectra from the same annulus to be displayed on the same axis.

Parameters

- **ann_ident** (*int*) – The integer identifier of the annulus you wish to see spectra for.
- **model** (*str*) – The fitted model to display on the data.
- **figsize** (*tuple*) – The size of the plot.

view_annuli(*obs_id*, *inst*, *model*, *figsize=12, 8*, *elevation_angle=30*, *azimuthal_angle=- 60*)

This view method is one of several in the AnnularSpectra class, and will display data and associated model fits for a single ObsID-Instrument combination for all annuli in this AnnularSpectra, in a 3D plot. The output of this can be quite visually confusing, so you may wish to use view_annulus to see the spectrum of a particular annulus for a particular ObsID-Instrument in a more traditional way, or just view to see all model fits at all annuli.

Parameters

- **obs_id** (*str*) – The ObsID of the spectra to display.
- **inst** (*str*) – The instrument of the spectra to display.
- **model** (*str*) – The model fit to display

- **figsize** (*tuple*) – The size of the figure.
- **elevation_angle** (*int*) – The elevation angle in the z plane, in degrees.
- **azimuthal_angle** (*int*) – The azimuth angle in the x,y plane, in degrees.

view(*model*, *figsize=12, 8*, *elevation_angle=30*, *azimuthal_angle=- 60*)
This view method is one of several in the AnnularSpectra class, and will display model fits to all spectra for each annuli in a 3D plot. No data is displayed in this viewing method, primarily because its so visually confusing. If you wish to see model fits displayed over actual data in this style, please use view_annuli.

Parameters

- **model** (*str*) – The model fit to display
- **figsize** (*tuple*) – The size of the figure.
- **elevation_angle** (*int*) – The elevation angle in the z plane, in degrees.
- **azimuthal_angle** (*int*) – The azimuth angle in the x,y plane, in degrees.

7.6 imagetools

7.6.1 imagetools.misc module

xga.imagetools.misc.**pix_deg_scale**(*coord*, *input_wcs*, *small_offset=<Quantity 1. arcmin>*)
Very heavily inspired by the regions module version of this function, just tweaked to work better for my use case. Perturbs the given coordinates with the small_offset value, converts the changed ra-dec coordinates to pixel, then calculates the difference between the new and original coordinates in pixel. Then small_offset is converted to degrees and divided by the pixel distance to calculate a pixel to degree factor.

Parameters

- **coord** (*Quantity*) – The starting coordinates.
- **input_wcs** (*WCS*) – The world coordinate system used to calculate the pixel to degree scale
- **small_offset** (*Quantity*) – The amount you wish to perturb the original coordinates

Returns Factor that can be used to convert pixel distances to degree distances, returned as an astropy quantity with units of deg/pix.

Return type Quantity

xga.imagetools.misc.**sky_deg_scale**(*im_prod*, *coord*, *small_offset=<Quantity 1. arcmin>*)
This is equivelant to pix_deg_scale, but instead calculates the conversion factor between XMM's XY sky coordinate system and degrees.

Parameters

- **im_prod** (*Image/Ratemap/ExpMap*) – The image product to calculate the conversion factor for.
- **coord** (*Quantity*) – The starting coordinates.
- **small_offset** (*Quantity*) – The amount you wish to perturb the original coordinates

Returns A scaling factor to convert sky distances to degree distances, returned as an astropy quantity with units of deg/xmm_sky.

Return type Quantity

xga.imagetools.misc.**pix_rad_to_physical**(*im_prod*, *pix_rad*, *out_unit*, *coord*, *z=None*, *cosmo=None*)

Pure convenience function to convert a list of pixel radii to whatever unit we might want at the end. Used quite a lot in the imagetools.profile functions, which is why it was split off into its own function. Redshift and cosmology must be supplied if proper distance units (like kpc) are chosen for out_unit.

Parameters

- **im_prod** (*Image/RateMap/ExpMap*) – The image/ratemap product for which the conversion is taking place.
- **pix_rad** (*Quantity*) – The array of pixel radii to convert to out_unit.
- **out_unit** (*UnitBase*) – The desired output unit for the radii.
- **coord** (*Quantity*) – The position of the object being analysed.
- **z** (*float/int*) – The redshift of the object (only required for proper distance units like kpc).
- **cosmo** – The chosen cosmology for the analysis (only required for proper distance units like kpc).

Returns An astropy Quantity with the radii in units of out_unit.

Return type Quantity

xga.imagetools.misc.**physical_rad_to_pix**(*im_prod*, *physical_rad*, *coord*, *z=None*, *cosmo=None*)

Another convenience function, this time to convert physical radii to pixels. It can deal with both angular and proper radii, so long as redshift and cosmology information is provided for the conversion from proper radii to pixels.

Parameters

- **im_prod** (*Image/RateMap/ExpMap*) –
- **physical_rad** (*Quantity*) – The physical radius to be converted to pixels.
- **coord** (*Quantity*) – The position of the object being analysed.
- **z** (*float/int*) – The redshift of the object (only required for input proper distance units like kpc).
- **cosmo** – The chosen cosmology for the analysis (only required for input proper distance units like kpc).

Returns The converted radii, in an astropy Quantity with pix units.

Return type Quantity

xga.imagetools.misc.**data_limits**(*im_prod*)

A function that finds the pixel coordinates that bound where data is present in Image or RateMap object.

Parameters **im_prod** (*Image/RateMap/ndarray*) – An Image, RateMap, or numpy array that you wish to find boundary coordinates for.

Returns Two lists, the first with the x lower and upper bounding coordinates, and the second with the y lower and upper bounding coordinates.

Return type Tuple[List[int, int], List[int, int]]

xga.imagetools.misc.**edge_finder**(*data*, *keep_corners=True*, *border=False*)

A simple edge finding algorithm designed to locate 'edges' in binary data, or in special cases produce a detector map of an instrument using an exposure map. The algorithm takes the difference of one column from the next,

over the entire array, then does the same with rows. Different difference values indicate where edges are in the array, and when added together all edges should be located.

Depending on how the 'border' option is set, the returned array will either represent the exact edge, or a boundary that is 1 pixel outside the actual edge.

Parameters

- **data** (*RateMap/ExpMap/ndarray*) – The 2D array or exposure map to run edge detection on. If an array is passed it must only consist of 0s and 1s.
- **keep_corners** (*bool*) – Should corner information be kept in the output array. If True then 2s in the output will indicate vertices.
- **border** (*bool*) – If True, then the returned array will represent a border running around the boundary of the true edge, rather than the outer boundary of the edge itself.

Returns An array of 0s and 1s. 1s indicate a detected edge.

Return type np.ndarray

7.6.2 imagetools.profile module

xga.imagetools.profile.**annular_mask**(*centre*, *inn_rad*, *out_rad*, *shape*, *start_ang=<Quantity 0. deg>*, *stop_ang=<Quantity 360. deg>*)

A handy little function to generate annular (or circular) masks in the form of numpy arrays. It produces the src_mask for a given shape of image, centered at supplied coordinates, and with inner and outer radii supplied by the user also. Angular limits can also be supplied to give the src_mask an annular dependence. This function should be properly vectorised, and accepts inner and outer radii in the form of arrays. The result will be an len_y, len_x, N dimensional array, where N is equal to the length of inn_rad.

Parameters

- **centre** (*Quantity*) – Astropy pix quantity of the form Quantity([x, y], pix).
- **inn_rad** (*np.ndarray*) – Pixel radius for the inner part of the annular src_mask.
- **out_rad** (*np.ndarray*) – Pixel radius for the outer part of the annular src_mask.
- **start_ang** (*Quantity*) – Lower angular limit for the src_mask.
- **stop_ang** (*Quantity*) – Upper angular limit for the src_mask.
- **shape** (*tuple*) – The output from the shape property of the numpy array you are generating masks for.

Returns The generated src_mask array.

Return type np.ndarray

xga.imagetools.profile.**ann_radii**(*im_prod*, *centre*, *rad*, *z=None*, *pix_step=1*, *rad_units=Unit("arcsec")*, *cosmo=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *min_central_pix_rad=3*, *start_pix_rad=0*)

Will probably only ever be called by an internal brightness calculation, but two different methods need it so it gets its own method.

Parameters

- **im_prod** (*Image*) – An Image or RateMap product object that you wish to calculate annuli for.

- **centre** (*Quantity*) – The coordinates of the centre of the set of annuli.
- **rad** (*Quantity*) – The outer radius of the set of annuli.
- **z** (*float*) – The redshift of the source of interest, required if the output radius units are a proper radius.
- **pix_step** (*int*) – The width (in pixels) of each annular bin, default is 1.
- **rad_units** (*UnitBase*) – The output units for the centres of the annulli returned by this function. The inner and outer radii will always be in pixels.
- **cosmo** – An instance of an astropy cosmology, the default is Planck15.
- **start_pix_rad** (*int*) – The pixel radius at which the innermost annulus starts, default is zero.
- **min_central_pix_rad** (*int*) – The minimum radius of the innermost circular annulus (will only be used if start_pix_rad is 0, otherwise the innermost annulus is not a circle), default is three.

Returns Returns the inner and outer radii of the annuli (in pixels), and the centres of the annuli in cen_rad_units.

Return type Tuple[np.ndarray, np.ndarray, Quantity]

xga.imagetools.profile.**radial_brightness**(*rt*, *centre*, *outer_rad*, *back_inn_rad_factor=1.05*, *back_out_rad_factor=1.5*, *interloper_mask=None*, *z=None*, *pix_step=1*, *rad_units=Unit("arcsec")*, *cosmo=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*, *min_snr=0.0*, *min_central_pix_rad=3*, *start_pix_rad=0*)

A simple method to calculate the average brightness in circular annuli upto the radius of the chosen region. The annuli are one pixel in width, and as this uses the masks that were generated earlier, interloper sources should be removed.

Parameters

- **rt** (RateMap) – A RateMap object to construct a brightness profile from.
- **centre** (*Quantity*) – The coordinates for the centre of the brightness profile.
- **outer_rad** (*Quantity*) – The outer radius of the brightness profile.
- **back_inn_rad_factor** (*float*) – This factor is multiplied by the outer pixel radius, which gives the inner radius for the background mask.
- **back_out_rad_factor** (*float*) – This factor is multiplied by the outer pixel radius, which gives the outer radius for the background mask.
- **interloper_mask** (*np.ndarray*) – A numpy array that masks out any interloper sources.
- **z** (*float*) – The redshift of the source of interest.
- **pix_step** (*int*) – The width (in pixels) of each annular bin, default is 1.
- **rad_units** (*BaseUnit*) – The desired output units for the central radii of the annuli.
- **cosmo** – An astropy cosmology object for source coordinate conversions.

- **min_snr** (*float*) – The minimum signal to noise allowed for each bin in the profile. If any point is below this threshold the profile will be rebinned. Default is 0.0
- **start_pix_rad** (*int*) – The pixel radius at which the innermost annulus starts, default is zero.
- **min_central_pix_rad** (*int*) – The minimum radius of the innermost circular annulus (will only be used if start_pix_rad is 0, otherwise the innermost annulus is not a circle), default is three.

Returns The brightness is returned in a flat numpy array, then the radii at the centre of the bins are returned in units of kpc, the width of the bins, and finally the average brightness in the background region is returned.

Return type Tuple[*SurfaceBrightness1D*, bool]

xga.imagetools.profile.**pizza_brightness**(*im_prod*, *src_mask*, *back_mask*, *centre*, *rad*, *num_slices=4*, *z=None*, *pix_step=1*, *cen_rad_units=Unit("arcsec")*, *cosmo=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*)

7.6.3 imagetools.psf module

xga.imagetools.psf.**rl_psf**(*sources*, *iterations=15*, *psf_model='ELLBETA'*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *bins=4*, *num_cores=1*)

An implementation of the Richardson-Lucy (doi:10.1364/JOSA.62.000055) PSF deconvolution algorithm that also takes into account the spatial variance of the XMM Newton PSF. The sources passed into this function will have all images matching the passed energy range deconvolved, the image objects will have the result stored in them alongside the original data, and a combined image will be generated. I view this method as quite crude, but it does seem to work, and I may implement a more intelligent way of doing PSF deconvolutions later. I initially tried convolving the PSFs generated at different spatial points with the chunks of data relevant to them in isolation, but the edge effects were very obvious. So I settled on convolving the whole original image with each PSF, and after it was finished taking the relevant chunks and patchworking them into a new array.

Parameters

- **sources** (*BaseSource/BaseSample*) – A single source object, or list of source objects.
- **iterations** (*int*) – The number of deconvolution iterations performed by the Richardson-Lucy algorithm.
- **psf_model** (*str*) – Which model of PSF should be used for this deconvolution. The default is ELLBETA, the best available.
- **lo_en** (*Quantity*) – The lower energy bound of the images to be deconvolved.
- **hi_en** (*Quantity*) – The upper energy bound of the images to be deconvolved.
- **bins** (*int*) – Number of bins that the X and Y axes will be divided into when generating a PSFGrid.
- **num_cores** (*int*) – The number of cores to use (if running locally), the default is set to 90% of available cores in your system.

7.7 sourcetools

7.7.1 sourcetools.density module

xga.sourcetools.density.**inv_abel_fitted_model**(*sources*, *model*, *fit_method='mcmc'*, *outer_radius='r500'*, *num_dens=True*, *use_peak=True*, *pix_step=1*, *min_snr=0.0*, *abund_table='angr'*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *psf_corr=True*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_walkers=20*, *num_steps=20000*, *num_samples=10000*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *obs_id=None*, *inst=None*, *conv_temp=None*, *conv_outer_radius='r500'*, *inv_abel_method=None*, *num_cores=1*, *show_warn=True*)

A photometric galaxy cluster gas density calculation method where a surface brightness profile is fit with a model and an inverse abel transform is used to infer the 3D count-rate/volume profile. Then a conversion factor calculated from simulated spectra is used to infer the number density profile.

Depending on the chosen surface brightness model, the inverse abel transform may be performed using an analytical solution, or numerical methods.

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – A GalaxyCluster or ClusterSample object to measure density profiles for.
- **model** (*str/List[str]/BaseModel1D/List[BaseModel1D]*) – The model(s) to be fit to the cluster surface profile(s). You may pass the string name of a model (for single or multiple clusters), a single instance of an XGA model class (for single or multiple clusters), a list of string names (one entry for each cluster being analysed), or a list of XGA model instances (one entry for each cluster being analysed).
- **fit_method** (*str*) – The method for the profile object to use to fit the model, default is mcmc.
- **outer_radius** (*str/Quantity*) –
- **num_dens** (*bool*) – If True then a number density profile will be generated, otherwise a mass density profile will be generated.
- **use_peak** (*bool*) – If true the measured peak will be used as the central coordinate of the profile.
- **pix_step** (*int*) – The width (in pixels) of each annular bin for the profiles, default is 1.
- **min_snr** (*int/float*) – The minimum allowed signal to noise for the surface brightness profiles. Default is 0, which disables automatic re-binning.
- **abund_table** (*str*) – Which abundance table should be used for the XSPEC fit, FakeIt run, and for the electron/hydrogen number density ratio.
- **lo_en** (*Quantity*) – The lower energy limit of the combined ratemap used to calculate density.

- **hi_en** (*Quantity*) – The upper energy limit of the combined ratemap used to calculate density.
- **psf_corr** (*bool*) – Default True, whether PSF corrected ratemaps will be used to make the surface brightness profile, and thus the density (if False density results could be incorrect).
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_walkers** (*int*) – If using mcmc fitting, the number of walkers to use. Default is 20.
- **num_steps** (*int*) – If using mcmc fitting, the number of steps each walker should take. Default is 20000.
- **num_samples** (*int*) – The number of samples drawn from the posterior distributions of model parameters after the fitting process is complete.
- **group_spec** (*bool*) – Whether the spectra that were used for fakeit were grouped.
- **min_counts** (*float*) – The minimum counts per channel, if the spectra that were used for fakeit were grouped by minimum counts.
- **min_sn** (*float*) – The minimum signal to noise per channel, if the spectra that were used for fakeit were grouped by minimum signal to noise.
- **over_sample** (*float*) – The level of oversampling applied on the spectra that were used for fakeit.
- **obs_id** (*str/list*) – A specific ObsID(s) to measure the density from. This should be a string if a single source is being analysed, and a list of ObsIDs the same length as the number of sources otherwise. The default is None, in which case the combined data will be used to measure the density profile.
- **inst** (*str/list*) – A specific instrument(s) to measure the density from. This can either be passed as a single string (e.g. 'pn') if only one source is being analysed, or the same instrument should be used for every source in a sample, or a list of strings if different instruments are required for each source. The default is None, in which case the combined data will be used to measure the density profile.
- **conv_temp** (*Quantity*) – If set this will override XGA measured temperatures within the conv_outer_radius, and the fakeit run to calculate the normalisation conversion factor will use these temperatures. The quantity

  should have an entry for each cluster being analysed. Default is None.
- **conv_outer_radius** (*str/Quantity*) – The outer radius within which to generate spectra and measure temperatures for the conversion factor calculation, default is 'r500'. An astropy quantity may also be passed, with either a single value or an entry for each cluster being analysed.
- **inv_abel_method** (*str*) – The method which should be used for the inverse abel transform of model which is fitted to the surface brightness profile. This overrides the default method for the model, which is either 'analytical' for models with an analytical solution to the inverse abel transform, or 'direct' for models which don't have an analytical solution. Default is None.

- **num_cores** (*int*) – The number of cores that the evselect call and XSPEC functions are allowed to use.
- **show_warn** (*bool*) – Should fit warnings be shown on screen.

Returns A list of the 3D gas density profiles measured by this function, though if the measurement was not successful an entry of None will be added to the list.

Return type List[*GasDensity3D*]

xga.sourcetools.density.**ann_spectra_apec_norm**(*sources*, *outer_radii*, *num_dens=True*, *annulus_method='min_snr'*, *min_snr=20*, *min_width=<Quantity 20. arcsec>*, *use_combined=True*, *use_worst=False*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *allow_negative=False*, *exp_corr=True*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *abund_table='angr'*, *num_data_real=10000*, *sigma=1*, *num_cores=1*)

A method of measuring density profiles using XSPEC fits of a set of Annular Spectra. First checks whether the required annular spectra already exist and have been fit using XSPEC, if not then they are generated and fitted, and APEC normalisation profiles will be produced (with projected temperature profiles also being made as a useful extra). Then the apec normalisation profile will be used, with knowledge of the source's redshift and chosen analysis cosmology, to produce a density profile from the APEC normalisation.

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – An individual or sample of sources to calculate 3D gas density profiles for.
- **outer_radii** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectrum (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **num_dens** (*bool*) – If True then a number density profile will be generated, otherwise a mass density profile will be generated.
- **annulus_method** (*str*) – The method by which the annuli are designated, this can be 'min_snr' (which will use the min_snr_proj_temp_prof function), or 'growth' (which will use the grow_ann_proj_temp_prof function).
- **min_snr** (*float*) – The minimum signal to noise which is allowable in a given annulus.
- **min_width** (*Quantity*) – The minimum allowable width of an annulus. The default is set to 20 arcseconds to try and avoid PSF effects.
- **use_combined** (*bool*) – If True then the combined RateMap will be used for signal to noise annulus calculations, this is overridden by use_worst.
- **use_worst** (*bool*) – If True then the worst observation of the cluster (ranked by global signal to noise) will be used for signal to noise annulus calculations.
- **lo_en** (*Quantity*) – The lower energy bound of the ratemap to use for the signal to noise calculations.

- **hi_en** (*Quantity*) – The upper energy bound of the ratemap to use for the signal to noise calculations.
- **psf_corr** (*bool*) – Sets whether you wish to use a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want to use is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want to use is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of iterations.
- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).
- **exp_corr** (*bool*) – Should signal to noises be measured with exposure time correction, default is True. I recommend that this be true for combined observations, as exposure time could change quite dramatically across the combined product.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **abund_table** (*str*) – The abundance table to use both for the conversion from n_exn_p to n_e^2 during density calculation, and the XSPEC fit.
- **num_data_real** (*int*) – The number of random realisations to generate when propagating profile uncertainties.
- **sigma** (*int*) – What sigma uncertainties should newly created profiles have, the default is 2.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.

Returns A list of the 3D gas density profiles measured by this function, though if the measurement was not successful an entry of None will be added to the list.

Return type List[*GasDensity3D*]

7.7.2 sourcetools.temperature module

xga.sourcetools.temperature.**min_snr_proj_temp_prof**(*sources*, *outer_radii*, *min_snr=20*, *min_width=<Quantity 20. arcsec>*, *use_combined=True*, *use_worst=False*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *allow_negative=False*, *exp_corr=True*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *abund_table='angr'*, *num_cores=1*)

This is a convenience function that allows you to quickly and easily start measuring projected temperature profiles of galaxy clusters, deciding on the annular bins using signal to noise measurements from photometric products. This function calls single_temp_apec_profile, but doesn't expose all of the more in depth variables, so if you want more control then use single_temp_apec_profile directly. The projected temperature profiles which are generated are added to their source's storage structure.

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – An individual or sample of sources to measure projected temperature profiles for.
- **outer_radii** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectra (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **min_snr** (*float*) – The minimum signal to noise which is allowable in a given annulus.
- **min_width** (*Quantity*) – The minimum allowable width of an annulus. The default is set to 20 arcseconds to try and avoid PSF effects.
- **use_combined** (*bool*) – If True then the combined RateMap will be used for signal to noise annulus calculations, this is overridden by use_worst.
- **use_worst** (*bool*) – If True then the worst observation of the cluster (ranked by global signal to noise) will be used for signal to noise annulus calculations.
- **lo_en** (*Quantity*) – The lower energy bound of the ratemap to use for the signal to noise calculations.
- **hi_en** (*Quantity*) – The upper energy bound of the ratemap to use for the signal to noise calculations.
- **psf_corr** (*bool*) – Sets whether you wish to use a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want to use is PSF corrected, this is the PSF model used.

- **psf_bins** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want to use is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of iterations.
- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).
- **exp_corr** (*bool*) – Should signal to noises be measured with exposure time correction, default is True. I recommend that this be true for combined observations, as exposure time could change quite dramatically across the combined product.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **abund_table** (*str*) – The abundance table to use during the XSPEC fits.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.

Returns A list of non-scalar astropy quantities containing the annular radii used to generate the projected temperature profiles created by this function. Each Quantity element of the list corresponds to a source.

Return type List[Quantity]

xga.sourcetools.temperature.**grow_ann_proj_temp_prof**(*sources*, *outer_radii*, *growth_factor=1.3*, *start_radius=<Quantity 20. arcsec>*, *num_ann=None*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *num_cores=1*)

This is a convenience function that allows you to quickly and easily start measuring projected temperature profiles of galaxy clusters where the outer radius of each annulus is some factor larger than that of the last annulus:

$$R_{i+1} = R_i F$$

If a growth factor is passed then the start radius and outer radius of a particular source will be used to solve for the number of annuli which should be generated. However if a number of annuli is passed (through num_ann), then this function will again use the start and outer radii and solve for the growth factor instead, over-riding any growth factor that may have been passed in.

This function calls single_temp_apec_profile, but doesn't expose all of the more in depth variables, so if you want more control then use single_temp_apec_profile directly. The projected temperature profiles which are generated are added to their source's storage structure.

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – An individual or sample of sources to measure projected temperature profiles for.
- **outer_radii** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectra (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **growth_factor** (*float*) – The factor by which the outer radius of the Nth annulus should be larger than the outer radius of the N-1th annulus. This will be over-ridden by a re-calculated value if a value is passed to num_ann.
- **start_radius** (*Quantity*) – The radius of the innermost circular annulus, the default is 20 arcseconds, which was chosen to try and avoid PSF effects.
- **num_ann** (*int*) – The number of annuli which should be used, default is None, in which case the value will be calculated using the growth factor, outer radius, and start radius. If this parameter is passed then growth_factor will be overridden by a recalculated value.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.

xga.sourcetools.temperature.**onion_deproj_temp_prof**(*sources*, *outer_radii*, *annulus_method='min_snr'*, *min_snr=30*, *min_width=<Quantity 20. arcsec>*, *use_combined=True*, *use_worst=False*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *allow_negative=False*, *exp_corr=True*, *group_spec=True*, *min_counts=5*, *min_sn=None*, *over_sample=None*, *one_rmf=True*, *abund_table='angr'*, *num_data_real=300*, *sigma=1*, *num_cores=1*)

This function will generate de-projected, three-dimensional, gas temperature profiles of galaxy clusters using the 'onion peeling' deprojection method. It will also generate any projected temperature profiles that may be necessary, using the method specified in the function call (the default is the minimum signal to noise annuli method). As a side effect of that process APEC normalisation profiles will also be created, as well as Emission Measure profiles. The function is an implementation of a fairly old technique, though it has been used recently in https://doi.org/10.1051/0004-6361/201731748. For a more in depth discussion of this technique and its uses I would currently recommend https://doi.org/10.1051/0004-6361:20020905.

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – An individual or sample of sources to calculate 3D temperature profiles for.
- **outer_radii** (*str/Quantity*) – The name or value of the outer radius to use for the generation of the spectra (for instance 'r200' would be acceptable for a GalaxyCluster, or Quantity(1000, 'kpc')). If 'region' is chosen (to use the regions in region files), then any inner radius will be ignored. If you are generating for multiple sources then you can also pass a Quantity with one entry per source.
- **annulus_method** (*str*) – The method by which the annuli are designated, this can be 'min_snr' (which will use the min_snr_proj_temp_prof function), or 'growth' (which will use the grow_ann_proj_temp_prof function).
- **min_snr** (*float*) – The minimum signal to noise which is allowable in a given annulus.
- **min_width** (*Quantity*) – The minimum allowable width of an annulus. The default is set to 20 arcseconds to try and avoid PSF effects.
- **use_combined** (*bool*) – If True then the combined RateMap will be used for signal to noise annulus calculations, this is overridden by use_worst.
- **use_worst** (*bool*) – If True then the worst observation of the cluster (ranked by global signal to noise) will be used for signal to noise annulus calculations.
- **lo_en** (*Quantity*) – The lower energy bound of the ratemap to use for the signal to noise calculations.

- **hi_en** (*Quantity*) – The upper energy bound of the ratemap to use for the signal to noise calculations.
- **psf_corr** (*bool*) – Sets whether you wish to use a PSF corrected ratemap or not.
- **psf_model** (*str*) – If the ratemap you want to use is PSF corrected, this is the PSF model used.
- **psf_bins** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of PSFs per side in the PSF grid.
- **psf_algo** (*str*) – If the ratemap you want to use is PSF corrected, this is the algorithm used.
- **psf_iter** (*int*) – If the ratemap you want to use is PSF corrected, this is the number of iterations.
- **allow_negative** (*bool*) – Should pixels in the background subtracted count map be allowed to go below zero, which results in a lower signal to noise (and can result in a negative signal to noise).
- **exp_corr** (*bool*) – Should signal to noises be measured with exposure time correction, default is True. I recommend that this be true for combined observations, as exposure time could change quite dramatically across the combined product.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **min_counts** (*float*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*float*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **one_rmf** (*bool*) – This flag tells the method whether it should only generate one RMF for a particular ObsID-instrument combination - this is much faster in some circumstances, however the RMF does depend slightly on position on the detector.
- **abund_table** (*str*) – The abundance table to use both for the conversion from n_exn_p to n_e^2 during density calculation, and the XSPEC fit.
- **num_data_real** (*int*) – The number of random realisations to generate when propagating profile uncertainties.
- **sigma** (*int*) – What sigma uncertainties should newly created profiles have, the default is 1.
- **num_cores** (*int*) – The number of cores to use (if running locally), default is set to 90% of available.

Returns A list of the 3D temperature profiles measured by this function, though if the measurement was not successful an entry of None will be added to the list.

Return type List[*GasTemperature3D*]

7.7.3 sourcetools.mass module

xga.sourcetools.mass.**inv_abel_dens_onion_temp**(*sources*, *outer_radius*, *sb_model*, *dens_model*, *temp_model*, *global_radius*, *fit_method='mcmc'*, *num_walkers=20*, *num_steps=20000*, *sb_pix_step=1*, *sb_min_snr=0.0*, *inv_abel_method=None*, *temp_min_snr=20*, *abund_table='angr'*, *group_spec=True*, *spec_min_counts=5*, *spec_min_sn=None*, *over_sample=None*, *num_cores=1*, *show_warn=True*)

A convenience function that should allow the user to easily measure hydrostatic masses of a sample of galaxy clusters, elegantly dealing with any sources that have inadequate data or aren't fit properly. For the sake of convenience, I have taken away a lot of choices that can be passed into the density and temperature measurement routines, and if you would like more control then please manually define a hydrostatic mass profile object.

This function uses the inv_abel_fitted_model density measurement function, and the onion_deproj_temp_prof temperature measurement function (with the minimum signal to noise criteria for deciding on the annular spectra sizes).

Parameters

- **sources** (*GalaxyCluster/ClusterSample*) – The galaxy cluster, or sample of galaxy clusters, that you wish to measure hydrostatic masses for.
- **outer_radius** (*str/Quantity*) – The radius out to which you wish to measure gas density and temperature profiles. This can either be a string radius name (like 'r500') or an astropy quantity. That quantity should have as many entries as there are sources.
- **sb_model** (*str/List[str]/BaseModel1D/List[BaseModel1D]*) – The model(s) to be fit to the cluster surface profile(s). You may pass the string name of a model (for single or multiple clusters), a single instance of an XGA model class (for single or multiple clusters), a list of string names (one entry for each cluster being analysed), or a list of XGA model instances (one entry for each cluster being analysed).
- **dens_model** (*str/List[str]/BaseModel1D/List[BaseModel1D]*) – The model(s) to be fit to the cluster density profile(s). You may pass the string name of a model (for single or multiple clusters), a single instance of an XGA model class (for single or multiple clusters), a list of string names (one entry for each cluster being analysed), or a list of XGA model instances (one entry for each cluster being analysed).
- **temp_model** (*str/List[str]/BaseModel1D/List[BaseModel1D]*) – The model(s) to be fit to the cluster temperature profile(s). You may pass the string name of a model (for single or multiple clusters), a single instance of an XGA model class (for single or multiple clusters), a list of string names (one entry for each cluster being analysed), or a list of XGA model instances (one entry for each cluster being analysed).
- **global_radius** (*str/Quantity*) – This is a radius for a 'global' temperature measurement, which is both used as an initial check of data quality, and feeds into the conversion factor required for density measurements. This may also be passed as either a named radius or a quantity.
- **fit_method** (*str*) – The method to use for fitting profiles within this function, default is 'mcmc'.
- **num_walkers** (*int*) – If fit_method is 'mcmc' this is the number of walkers to initialise for the ensemble sampler.

- **num_steps** (*int*) – If fit_method is 'mcmc' this is the number steps for each walker to take.
- **sb_pix_step** (*int*) – The width (in pixels) of each annular bin for the surface brightness profiles, default is 1.
- **sb_min_snr** (*int/float*) – The minimum allowed signal to noise for the surface brightness profiles. Default is 0, which disables automatic re-binning.
- **inv_abel_method** (*str*) – The method which should be used for the inverse abel transform of the model which is fitted to the surface brightness profile. This overrides the default method for the model, which is either 'analytical' for models with an analytical solution to the inverse abel transform, or 'direct' for models which don't have an analytical solution. Default is None.
- **temp_min_snr** (*int/float*) – The minimum signal to noise for a temperature measurement annulus, default is 30.
- **abund_table** (*str*) – The abundance table to use for fitting, and the conversion factor required during density calculations.
- **group_spec** (*bool*) – A boolean flag that sets whether generated spectra are grouped or not.
- **spec_min_counts** (*int*) – If generating a grouped spectrum, this is the minimum number of counts per channel. To disable minimum counts set this parameter to None.
- **spec_min_sn** (*float*) – If generating a grouped spectrum, this is the minimum signal to noise in each channel. To disable minimum signal to noise set this parameter to None.
- **over_sample** (*bool*) – The minimum energy resolution for each group, set to None to disable. e.g. if over_sample=3 then the minimum width of a group is 1/3 of the resolution FWHM at that energy.
- **num_cores** (*int*) – The number of cores on your local machine which this function is allowed, default is 90% of the cores in your system.
- **show_warn** (*bool*) – Should profile fit warnings be shown, or only stored in the profile models.

Returns A list of the hydrostatic mass profiles measured by this function, though if the measurement was not successful an entry of None will be added to the list.

Return type List[*HydrostaticMass*]

7.7.4 sourcetools.match module

xga.sourcetools.match.**simple_xmm_match**(*src_ra*, *src_dec*, *distance=<Quantity 30. arcmin>*)

Returns ObsIDs within a given distance from the input ra and dec values.

Parameters

- **src_ra** (*float*) – RA coordinate of the source, in degrees.
- **src_dec** (*float*) – DEC coordinate of the source, in degrees.
- **distance** (*Quantity*) – The distance to search for XMM observations within, default should be able to match a source on the edge of an observation to the centre of the observation.

Returns The ObsID, RA_PNT, and DEC_PNT of matching XMM observations.

Return type DataFrame

7.7.5 sourcetools.misc module

xga.sourcetools.misc.**nh_lookup**(*coord_pair*)
Uses HEASOFT to lookup hydrogen column density for given coordinates.

Parameters **coord_pair** (*Quantity*) – An astropy quantity with RA and DEC of interest.

Returns Average and weighted average nH values (in units of cm^{-2})

Return type ndarray

xga.sourcetools.misc.**rad_to_ang**(*rad*, *z*, *cosmo=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*)
Converts radius in length units to radius on sky in degrees.

Parameters

- **rad** (*Quantity*) – Radius for conversion.
- **cosmo** (*Cosmology*) – An instance of an astropy cosmology, the default is Planck15.
- **z** (*float*) – The _redshift of the source.

Returns The radius in degrees.

Return type Quantity

xga.sourcetools.misc.**ang_to_rad**(*ang*, *z*, *cosmo=FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05, m_nu=[0. 0. 0.06] eV, Ob0=0.0486)*)
The counterpart to rad_to_ang, this converts from an angle to a radius in kpc.

Parameters

- **ang** (*Quantity*) – Angle to be converted to radius.
- **cosmo** (*Cosmology*) – An instance of an astropy cosmology, the default is Planck15.
- **z** (*float*) – The _redshift of the source.

Returns The radius in kpc.

Return type Quantity

xga.sourcetools.misc.**name_to_coord**(*name*)
I'd like it to be known that I hate papers and resources that either only give the name of an object or its sexagesimal coordinates - however it happens upsettingly often so here we are. This function will take a standard format name (e.g. XMMXCS J041853.9+555333.7) and return RA and DEC in degrees.

Parameters **name** –

xga.sourcetools.misc.**coord_to_name**(*coord_pair*, *survey=None*)
This was originally just written in the init of BaseSource, but I figured I should split it out into its own function really. This will take a coordinate pair, and optional survey name, and spit out an object name in the standard format.

Returns Source name based on coordinates.

Return type str

xga.sourcetools.misc.**model_check**(*sources*, *model*)
Very simple function that checks if a passed set of models is appropriately structured for the number of sources that have been passed. I can't imagine why a user would need this directly, its only here as these checks have to be performed in multiple places in sourcetools.

Parameters

- **sources** (*List[BaseSource]/BaseSample/BaseSource*) – The source(s).
- **model** (*str/List[str]/BaseModel1D/List[BaseModel1D]*) – The model(s).

Returns A list of model instances, or names of models.

Return type Union[List[*BaseModel1D*], List[str]]

7.7.6 sourcetools.stack module

xga.sourcetools.stack.**radial_data_stack**(*sources*, *scale_radius='r200'*, *use_peak=True*, *pix_step=1*, *radii=array([0.01, 0.06210526, 0.11421053, 0.16631579, 0.21842105, 0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737, 0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368, 0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.])*, *min_snr=0.0*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *custom_temps=None*, *sim_met=0.3*, *abund_table='angr'*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_cores=1*)

Creates and scales radial brightness profiles for a set of galaxy clusters so that they can be combined and compared, like for like. This particular function does not fit models, and outputs a mean brightness profile, as well as the scaled stack data and covariance matrices. This is based on the method in https://doi.org/10.1093/mnras/stv1366, though modified to work with profiles rather than 2D images.

Parameters

- **sources** (*ClusterSample*) – The source objects that will contribute to the stacked brightness profile.
- **scale_radius** (*str*) – The overdensity radius to scale the cluster radii by, all GalaxyCluster objects must have an entry for this radius.
- **use_peak** (*bool*) – Controls whether the peak position is used as the centre of the brightness profile for each GalaxyCluster object.
- **pix_step** (*int*) – The width (in pixels) of each annular bin for the individual profiles, default is 1.
- **radii** (*ndarray*) – The radii (in units of scale_radius) at which to measure and stack surface brightness.
- **min_snr** (*int/float*) – The minimum allowed signal to noise for individual cluster profiles. Default is 0, which disables automatic rebinning.
- **lo_en** (*Quantity*) – The lower energy limit of the data that goes into the stacked profiles.
- **hi_en** (*Quantity*) – The upper energy limit of the data that goes into the stacked profiles.
- **custom_temps** (*Quantity*) – Temperatures at which to calculate conversion factors for each cluster in sources, they will overwrite any temperatures measured by XGA. A single temperature can be passed to be used for all clusters in sources. If None, appropriate temperatures will be retrieved from the source objects.

- **sim_met** (*float/List*) – The metallicity(s) to use when calculating the conversion factor. Pass a single float to use the same value for all sources, or pass a list to use a different value for each.
- **abund_table** (*str*) – The abundance table to use for the temperature fit and conversion factor calculation.
- **psf_corr** (*bool*) – If True, PSF corrected ratemaps will be used to make the brightness profile stack.
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_cores** (*int*) – The number of cores to use when calculating the brightness profiles, the default is 90% of available cores.

Returns This function returns the average profile, the scaled brightness profiles with the cluster changing along the y direction and the bin changing along the x direction, an array of the radii at which the brightness was measured (in units of scale_radius), and finally the covariance matrix and normalised covariance matrix. I also return a list of source names that WERE included in the stack.

Return type Tuple[ndarray, ndarray, ndarray, ndarray, ndarray, list]

xga.sourcetools.stack.**view_radial_data_stack**(*sources*, *scale_radius='r200'*, *use_peak=True*, *pix_step=1*, *radii=array([0.01, 0.06210526, 0.11421053, 0.16631579, 0.21842105, 0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737, 0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368, 0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.])*, *min_snr=0.0*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *custom_temps=None*, *sim_met=0.3*, *abund_table='angr'*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_cores=1*, *show_images=False*, *figsize=(14, 14)*)

A convenience function that calls radial_data_stack and makes plots of the average profile, individual profiles, covariance, and normalised covariance matrix.

Parameters

- **sources** (ClusterSample) – The source objects that will contribute to the stacked brightness profile.
- **scale_radius** (*str*) – The overdensity radius to scale the cluster radii by, all GalaxyCluster objects must have an entry for this radius.
- **use_peak** (*bool*) – Controls whether the peak position is used as the centre of the brightness profile for each GalaxyCluster object.
- **pix_step** (*int*) – The width (in pixels) of each annular bin for the individual profiles, default is 1.

- **radii** (*ndarray*) – The radii (in units of scale_radius) at which to measure and stack surface brightness.
- **min_snr** (*int/float*) – The minimum allowed signal to noise for individual cluster profiles. Default is 0, which disables automatic rebinning.
- **lo_en** (*Quantity*) – The lower energy limit of the data that goes into the stacked profiles.
- **hi_en** (*Quantity*) – The upper energy limit of the data that goes into the stacked profiles.
- **custom_temps** (*Quantity*) – Temperatures at which to calculate conversion factors for each cluster in sources, they will overwrite any temperatures measured by XGA. A single temperature can be passed to be used for all clusters in sources. If None, appropriate temperatures will be retrieved from the source objects.
- **sim_met** (*float/List*) – The metallicity(s) to use when calculating the conversion factor. Pass a single float to use the same value for all sources, or pass a list to use a different value for each.
- **abund_table** (*str*) – The abundance table to use for the temperature fit and conversion factor calculation.
- **psf_corr** (*bool*) – If True, PSF corrected ratemaps will be used to make the brightness profile stack.
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_cores** (*int*) – The number of cores to use when calculating the brightness profiles, the default is 90% of available cores.
- **show_images** (*bool*) – If true then for each source in the stack an image and profile will be displayed side by side, with annuli overlaid on the image.
- **figsize** (*tuple*) – The desired figure size for the plot.

xga.sourcetools.stack.**radial_model_stack**(*sources*, *model*, *scale_radius='r200'*, *fit_method='mcmc'*, *use_peak=True*, *model_priors=None*, *model_start_pars=None*, *pix_step=1*, *radii=array([0.01, 0.06210526, 0.11421053, 0.16631579, 0.21842105, 0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737, 0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368, 0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.])*, *min_snr=0.0*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *custom_temps=None*, *sim_met=0.3*, *abund_table='angr'*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_cores=1*, *model_realisations=500*, *conf_level=90*, *num_walkers=20*, *num_steps=20000*)

Creates, fits, and scales radial brightness profiles for a set of galaxy clusters so that they can be combined and compared, like for like. This function fits models of a user's choice, and then uses the models to retrieve brightness values at user-defined radii as a fraction of the scale radius. From that point it functions much as radial_data_stack does.

Parameters

- **sources** (ClusterSample) – The source objects that will contribute to the stacked brightness profile.
- **model** (*str*) – The model to fit to the brightness profiles.
- **scale_radius** (*str*) – The overdensity radius to scale the cluster radii by, all GalaxyCluster objects must have an entry for this radius.
- **fit_method** (*str*) – The method to use when fitting the model to the profile.
- **use_peak** (*bool*) – Controls whether the peak position is used as the centre of the brightness profile for each GalaxyCluster object.
- **model_priors** (*list*) – A list of priors to use when fitting the model with MCMC, default is None in which case the default priors for the selected model are used.
- **model_start_pars** (*list*) – A list of start parameters to use when fitting with methods like curve_fit, default is None in which case the default start parameters for the selected model are used.
- **pix_step** (*int*) – The width (in pixels) of each annular bin for the individual profiles, default is 1.
- **radii** (*ndarray*) – The radii (in units of scale_radius) at which to measure and stack surface brightness.
- **min_snr** (*int/float*) – The minimum allowed signal to noise for individual cluster profiles. Default is 0, which disables automatic rebinning.
- **lo_en** (*Quantity*) – The lower energy limit of the data that goes into the stacked profiles.
- **hi_en** (*Quantity*) – The upper energy limit of the data that goes into the stacked profiles.
- **custom_temps** (*Quantity*) – Temperatures at which to calculate conversion factors for each cluster in sources, they will overwrite any temperatures measured by XGA. A single temperature can be passed to be used for all clusters in sources. If None, appropriate temperatures will be retrieved from the source objects.
- **sim_met** (*float/List*) – The metallicity(s) to use when calculating the conversion factor. Pass a single float to use the same value for all sources, or pass a list to use a different value for each.
- **abund_table** (*str*) – The abundance table to use for the temperature fit and conversion factor calculation.
- **psf_corr** (*bool*) – If True, PSF corrected ratemaps will be used to make the brightness profile stack.
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_cores** (*int*) – The number of cores to use when calculating the brightness profiles, the default is 90% of available cores.
- **model_realisations** (*int*) – The number of random realisations of a model to generate.

- **conf_level** (*int*) – The confidence level at which to measure uncertainties of parameters and profiles.
- **num_walkers** (*int*) – The number of walkers in the MCMC ensemble sampler.
- **num_steps** (*int*) – The number of steps in the chain that each walker should take.

Returns This function returns the average profile, the scaled brightness profiles with the cluster changing along the y direction and the bin changing along the x direction, an array of the radii at which the brightness was measured (in units of scale_radius), and finally the covariance matrix and normalised covariance matrix. I also return a list of source names that WERE included in the stack.

Return type Tuple[ndarray, ndarray, ndarray, ndarray, ndarray, list]

xga.sourcetools.stack.**view_radial_model_stack**(*sources*, *model*, *scale_radius='r200'*, *fit_method='mcmc'*, *use_peak=True*, *model_priors=None*, *model_start_pars=None*, *pix_step=1*, *radii=array([0.01, 0.06210526, 0.11421053, 0.16631579, 0.21842105, 0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737, 0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368, 0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.])*, *min_snr=0.0*, *lo_en=<Quantity 0.5 keV>*, *hi_en=<Quantity 2. keV>*, *custom_temps=None*, *sim_met=0.3*, *abund_table='angr'*, *psf_corr=False*, *psf_model='ELLBETA'*, *psf_bins=4*, *psf_algo='rl'*, *psf_iter=15*, *num_cores=1*, *model_realisations=500*, *conf_level=90*, *ml_mcmc_start=True*, *ml_rand_dev=0.0001*, *num_walkers=30*, *num_steps=20000*, *show_images=False*, *figsize=(14, 14)*)

A convenience function that calls radial_model_stack and makes plots of the average profile, individual profiles, covariance, and normalised covariance matrix.

Parameters

- **sources** (ClusterSample) – The source objects that will contribute to the stacked brightness profile.
- **model** (*str*) – The model to fit to the brightness profiles.
- **scale_radius** (*str*) – The overdensity radius to scale the cluster radii by, all GalaxyCluster objects must have an entry for this radius.
- **fit_method** (*str*) – The method to use when fitting the model to the profile.
- **use_peak** (*bool*) – Controls whether the peak position is used as the centre of the brightness profile for each GalaxyCluster object.
- **model_priors** (*list*) – A list of priors to use when fitting the model with MCMC, default is None in which case the default priors for the selected model are used.
- **model_start_pars** (*list*) – A list of start parameters to use when fitting with methods like curve_fit, default is None in which case the default start parameters for the selected model are used.

- **pix_step** (*int*) – The width (in pixels) of each annular bin for the individual profiles, default is 1.
- **radii** (*ndarray*) – The radii (in units of scale_radius) at which to measure and stack surface brightness.
- **min_snr** (*int/float*) – The minimum allowed signal to noise for individual cluster profiles. Default is 0, which disables automatic rebinning.
- **lo_en** (*Quantity*) – The lower energy limit of the data that goes into the stacked profiles.
- **hi_en** (*Quantity*) – The upper energy limit of the data that goes into the stacked profiles.
- **custom_temps** (*Quantity*) – Temperatures at which to calculate conversion factors for each cluster in sources, they will overwrite any temperatures measured by XGA. A single temperature can be passed to be used for all clusters in sources. If None, appropriate temperatures will be retrieved from the source objects.
- **sim_met** (*float/List*) – The metallicity(s) to use when calculating the conversion factor. Pass a single float to use the same value for all sources, or pass a list to use a different value for each.
- **abund_table** (*str*) – The abundance table to use for the temperature fit and conversion factor calculation.
- **psf_corr** (*bool*) – If True, PSF corrected ratemaps will be used to make the brightness profile stack.
- **psf_model** (*str*) – If PSF corrected, the PSF model used.
- **psf_bins** (*int*) – If PSF corrected, the number of bins per side.
- **psf_algo** (*str*) – If PSF corrected, the algorithm used.
- **psf_iter** (*int*) – If PSF corrected, the number of algorithm iterations.
- **num_cores** (*int*) – The number of cores to use when calculating the brightness profiles, the default is 90% of available cores.
- **model_realisations** (*int*) – The number of random realisations of a model to generate.
- **conf_level** (*int*) – The confidence level at which to measure uncertainties of parameters and profiles.
- **ml_mcmc_start** (*bool*) – If True then maximum likelihood estimation will be used to generate start parameters for MCMC fitting, otherwise they will be randomly drawn from parameter priors
- **ml_rand_dev** (*float*) – The scale of the random deviation around start parameters used for starting the different walkers in the MCMC ensemble sampler.
- **num_walkers** (*int*) – The number of walkers in the MCMC ensemble sampler.
- **num_steps** (*int*) – The number of steps in the chain that each walker should take.
- **show_images** (*bool*) – If true then for each source in the stack an image and profile will be displayed side by side, with annuli overlaid on the image.
- **figsize** (*tuple*) – The desired figure size for the plot.

7.7.7 sourcetools.deproj module

xga.sourcetools.deproj.**shell_ann_vol_intersect**(*shell_radii*, *ann_radii*)
This function calculates the volume intersection matrix of a set of circular annuli and a set of spherical shells. It is assumed that the annuli and shells have the same x and y origin. The intersection is derived using simple geometric considerations, have a look in the appendix of DOI 10.1086/300836.

Parameters

- **shell_radii** (*ndarray/Quantity*) – The radii of the spherical shells.
- **ann_radii** (*ndarray/Quantity*) – The radii of the circular annuli (DOES NOT need to be the same length as shell_radii).

Returns A 2D array containing the volumes of intersections between the circular annuli defined by i_ann and o_ann, and the spherical shells defined by i_sph and o_sph. Annular radii are along the 'x' axis and shell radii are along the 'y' axis.

Return type Union[np.ndarray, Quantity]

xga.sourcetools.deproj.**shell_volume**(*inn_radius*, *out_radius*)
Silly little function that calculates the volume of a spherical shell with inner radius inn_radius and outer radius out_radius.

Parameters

- **inn_radius** (*Quantity/np.ndarray*) – The inner radius of the spherical shell.
- **out_radius** (*Quantity/np.ndarray*) – The outer radius of the spherical shell.

Returns The volume of the specified shell

Return type Union[Quantity, np.ndarray]

7.8 relations

7.8.1 relations.clusters package

Submodules

relations.clusters.LT module

relations.clusters.L module

relations.clusters.MT module

relations.clusters.M module

7.8.2 relations.fit module

xga.relations.fit.**scaling_relation_curve_fit**(*model_func*, *y_values*, *y_errs*, *x_values*, *x_errs=None*, *y_norm=None*, *x_norm=None*, *x_lims=None*, *start_pars=None*, *y_name='Y'*, *x_name='X'*)
A function to fit a scaling relation with the scipy non-linear least squares implementation (curve fit), generate an XGA ScalingRelation product, and return it.

Parameters

- **model_func** (*FunctionType*) – The function object of the model you wish to fit. PLEASE NOTE, the function must be defined in the style used in xga.models.misc; i.e. powerlaw(x: np.ndarray, k: float, a: float), where the first argument is for x values, and the following arguments are all fit parameters.
- **y_values** (*Quantity*) – The y data values to fit to.
- **y_errs** (*Quantity*) – The y error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of y_values), or an Nx2 Quantity with lower and upper errors.
- **x_values** (*Quantity*) – The x data values to fit to.
- **x_errs** (*Quantity*) – The x error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of x_values), or an Nx2 Quantity with lower and upper errors.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **x_lims** (*Quantity*) – The range of x values in which this relation is valid, default is None. If this information is supplied, please pass it as a Quantity array, with the first element being the lower bound and the second element being the upper bound.
- **start_pars** (*list*) – The start parameters for the curve_fit run, default is None, which means curve_fit will use all ones.
- **y_name** (*str*) – The name to be used for the y-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.
- **x_name** (*str*) – The name to be used for the x-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.

Returns An XGA ScalingRelation object with all the information about the data and fit, a view method, and a predict method.

Return type *ScalingRelation*

xga.relations.fit.**scaling_relation_odr**(*model_func*, *y_values*, *y_errs*, *x_values*, *x_errs=None*, *y_norm=None*, *x_norm=None*, *x_lims=None*, *start_pars=None*, *y_name='Y'*, *x_name='X'*)

A function to fit a scaling relation with the scipy orthogonal distance regression implementation, generate an XGA ScalingRelation product, and return it.

Parameters

- **model_func** (*FunctionType*) – The function object of the model you wish to fit. PLEASE NOTE, the function must be defined in the style used in xga.models.misc; i.e. powerlaw(x: np.ndarray, k: float, a: float), where the first argument is for x values, and the following arguments are all fit parameters. The scipy ODR implementation requires functions of a different style, and I try to automatically convert the input function to that style, but to help that please avoid using one letter parameter names in any custom function you might want to use.
- **y_values** (*Quantity*) – The y data values to fit to.
- **y_errs** (*Quantity*) – The y error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of y_values), or an Nx2 Quantity with lower and upper errors.

- **x_values** (*Quantity*) – The x data values to fit to.
- **x_errs** (*Quantity*) – The x error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of x_values), or an Nx2 Quantity with lower and upper errors.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **x_lims** (*Quantity*) – The range of x values in which this relation is valid, default is None. If this information is supplied, please pass it as a Quantity array, with the first element being the lower bound and the second element being the upper bound.
- **start_pars** (*list*) – The start parameters for the ODR run, default is all ones.
- **y_name** (*str*) – The name to be used for the y-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.
- **x_name** (*str*) – The name to be used for the x-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.

Returns An XGA ScalingRelation object with all the information about the data and fit, a view method, and a predict method.

Return type *ScalingRelation*

xga.relations.fit.**scaling_relation_lira**(*y_values*, *y_errs*, *x_values*, *x_errs=None*, *y_norm=None*, *x_norm=None*, *x_lims=None*, *y_name='Y'*, *x_name='X'*, *num_steps=100000*, *num_chains=4*, *num_burn_in=10000*)

A function to fit a power law scaling relation with the excellent R fitting package LIRA (https://doi.org/10.1093/mnras/stv2374), this function requires a valid R installation, along with LIRA (and its dependencies such as JAGS), as well as the Python module rpy2.

Parameters

- **y_values** (*Quantity*) – The y data values to fit to.
- **y_errs** (*Quantity*) – The y error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of y_values), or an Nx2 Quantity with lower and upper errors.
- **x_values** (*Quantity*) – The x data values to fit to.
- **x_errs** (*Quantity*) – The x error values of the data. These should be supplied as either a 1D Quantity with length N (where N is the length of x_values), or an Nx2 Quantity with lower and upper errors.
- **y_norm** (*Quantity*) – Quantity to normalise the y data by.
- **x_norm** (*Quantity*) – Quantity to normalise the x data by.
- **x_lims** (*Quantity*) – The range of x values in which this relation is valid, default is None. If this information is supplied, please pass it as a Quantity array, with the first element being the lower bound and the second element being the upper bound.
- **y_name** (*str*) – The name to be used for the y-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.
- **x_name** (*str*) – The name to be used for the x-axis of the scaling relation (DON'T include the unit, that will be inferred from the astropy Quantity.
- **num_steps** (*int*) – The number of steps to take in each chain.

- **num_chains** (*int*) – The number of chains to run.
- **num_burn_in** (*int*) – The number of steps to discard as a burn in period. This is also used as the adapt parameter of the LIRA fit.

Returns An XGA ScalingRelation object with all the information about the data and fit, a view method, and a predict method.

Return type *ScalingRelation*

xga.relations.fit.**scaling_relation_emcee**()

7.9 models

7.9.1 Module contents

xga.models.**convert_to_odr_compatible**(*model_func*, *new_par_name=''*, *new_data_name='x_values'*)

This is a bit of a weird one; its meant to convert model functions from the standard XGA setup (i.e. pass x values, then parameters as individual variables), into the form expected by Scipy's ODR. I'd recommend running a check to compare results from the original and converted functions where-ever this function is called - I don't completely trust it.

Parameters

- **model_func** (*FunctionType*) – The original model function to be converted.
- **new_par_name** (*str*) – The name we want to use for the new list/array of fit parameters.
- **new_data_name** (*str*) – The new name we want to use for the x_data.

Returns A successfully converted model function (hopefully) which can be used with ODR.

Return type FunctionType

7.9.2 models.base module

class xga.models.base.**BaseModel1D**(*x_unit*, *y_unit*, *start_pars*, *par_priors*, *model_name*, *model_pub_name*, *par_pub_names*, *describes*, *info*, *x_lims=None*)

Bases: object

The superclass of XGA's 1D models, with base functionality implemented, including the numerical methods for calculating derivatives and abel transforms which can be overwritten by subclasses if analytical solutions are available. The BaseModel class shouldn't be instantiated by itself, as it won't do anything.

get_realisations(*x*)

This method uses the parameter distributions added to this model by a fitting process to generate random realisations of this model at a given x-position (or positions).

Parameters **x** (*Quantity*) – The x-position(s) at which realisations of the model should be generated from the associated parameter distributions.

Returns The model realisations, in a Quantity with shape (len(x), num_samples) if x has multiple radii in it (num_samples is the number of samples in the parameter distributions), and (num_samples,) if only a single x value is passed.

Return type Quantity

abstract static model(*x*, *pars*)
This is where the model function is actually defined, this MUST be overridden by every subclass model, hence why I've used the abstract method decorator.

Parameters

- **x** (*Quantity*) – The x-position at which the model should be evaluated.
- **pars** (*List[Quantity]*) – The parameters of model to be evaluated.

Returns The y-value of the model at x.

derivative(*x*, *dx*, *use_par_dist=False*)
Calculates a numerical derivative of the model at the specified x value, using the specified dx value. This method will be overridden in models that have an analytical solution to their first derivative, in which case the dx value will become irrelevant.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – The dx value to use during the calculation.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

nth_derivative(*x*, *dx*, *order*, *use_par_dist=False*)
A method to calculate the nth order derivative of the model using a numerical method.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – The dx value to use during the calculation.
- **order** (*int*) – The order of the desired derivative.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The value(s) of the nth order derivative of the model at x, either calculated from the current best fit parameters, or a distribution.

Return type Quantity

inverse_abel(*x*, *use_par_dist=False*, *method='direct'*)
This method uses numerical methods to generate the inverse abel transform of the model. It may be overridden by models that have analytical solutions to the inverse abel transform. All numerical inverse abel transform methods are from the pyabel module, and please be aware that in my (limited) experience the numerical solutions tend to diverge from analytical solutions at large radii.

Parameters

- **x** (*Quantity*) – The x location(s) at which to calculate the value of the inverse abel transform.

- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a inverse abel transform distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.
- **method** (*str*) – The method that should be used to calculate the values of this inverse abel transform. You may pass 'direct', 'basex', 'hansenlaw', 'onion_bordas', 'onion_peeling', 'two_point', or 'three_point'.

Returns The inverse abel transform result.

Return type Quantity

volume_integral(*outer_radius*, *use_par_dist=False*)

Calculates a numerical value for the volume integral of the function over a sphere of radius outer_radius. The scipy quad function is used. This method can either return a single value calculated using the current model parameters, or a distribution of values using the parameter distributions (assuming that this model has had a fit run on it).

This method will be overridden if there is an analytical solution to a particular model's volume integration over a sphere.

Parameters

- **outer_radius** (*Quantity*) – The radius to integrate out to.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a volume integral distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value

Returns The result of the integration, either a single value or a distribution.

Return type Quantity

allowed_prior_types(*table_format='fancy_grid'*)

Simple method to display the allowed prior types and their expected formats. :param str table_format: The desired format of the allowed models table. This is passed to the

tabulate module (allowed formats can be found here - https://pypi.org/project/tabulate/), and alters the way the printed table looks.

static compare_units(*check_pars*, *good_pars*)

Simple method that will be used in the inits of subclasses to make sure that any custom start values passed in by the user match the expected units of the default start parameters for that model.

Parameters

- **check_pars** (*List[Quantity]*) – The first list of parameters, these are being checked.
- **good_pars** (*List[Quantity]*) – The second list of parameters, these are taken as having 'correct' units.

Returns Only if the check pars pass the tests. We return the check pars list but with all elements converted to EXACTLY the same units as good_pars, not just equivelant.

Return type List[Quantity]

info(*table_format='fancy_grid'*)

A method that gives some information about this particular model. :param str table_format: The desired format of the allowed models table. This is passed to the

tabulate module (allowed formats can be found here - https://pypi.org/project/tabulate/), and alters the way the printed table looks.

predicted_dist_view(*radius*, *bins='auto'*, *colour='lightslategrey'*, *figsize=6, 5*)

A simple view method, to visualise the predicted value distribution at a particular radius. Only usable if this model has had parameter distributions assigned to it.

Parameters

- **radius** (*Quantity*) – The radius at which you wish to evaluate this model and view the predicted distribution.
- **int] bins** (*Union[str,*) – Equivelant to the plt.hist bins argument, set either the number of bins or the algorithm to decide on the number of bins.
- **colour** (*str*) – Set the colour of the histogram.
- **figsize** (*tuple*) – The desired dimensions of the figure.

par_dist_view(*bins='auto'*, *colour='lightslategrey'*)

Very simple method that allows you to view the parameter distributions that have been added to this model. The model parameter and uncertainties are indicated with red lines, highlighting the value and enclosing the 1sigma confidence region.

Parameters

- **int] bins** (*Union[str,*) – Equivelant to the plt.hist bins argument, set either the number of bins or the algorithm to decide on the number of bins.
- **colour** (*str*) – Set the colour of the histogram.

view(*radii=None*, *xscale='log'*, *yscale='log'*, *figsize=8, 8*, *colour='black'*)

Very simple view method to visualise XGA models with the current parameters.

Parameters

- **radii** (*Quantity*) – Radii at which to calculate points to plot, doesn't need to be set if the model has x limits defined.
- **xscale** (*str*) – The scale to apply to the x-axis, default is log.
- **yscale** (*str*) – The scale to apply to the y-axis, default is log.
- **figsize** (*tuple*) – The size of figure to be set up.
- **colour** (*str*) – The colour that the line in the plot should be.

property model_pars

Property that returns the current parameters of the model, by default they are the same as the parameter start values.

Returns A list of astropy quantities representing the values of the parameters of this model.

Return type List[Quantity]

property model_par_errs

Property that returns the uncertainties on the current parameters of the model, by default these will be zero as the default model_pars are the same as the start_pars.

Returns A list of astropy quantities representing the uncertainties on the parameters of this model.

Return type List[Quantity]

property start_pars
Property that returns the current start parameters of the model, by which I mean the values that certain types of fitting function will use to start their fit.

Returns A list of astropy quantities representing the values of the start parameters of this model.

Return type List[Quantity]

property unitless_start_pars
Returns sanitised start parameters which are floats rather than astropy quantities, sometimes necessary for fitting methods.

Returns Array of floats representing model start parameters.

Return type np.ndarray

property par_priors
Property that returns the current priors on parameters of the model, these will be used by any fitting function that sets priors on parameters. Each entry in this list will be a dictionary with two keys 'prior' and 'type'. The value for prior will be an astropy quantity, and the value for type will be a prior type (so uniform, gaussian, etc.)

Returns A list of astropy quantities representing the values of the start parameters of this model.

Return type List[Quantity]

property x_unit
Property to access the expected x-unit of this model.

Returns Astropy unit of the x values of the model.

Return type Unit

property y_unit
Property to access the expected y-unit of this model.

Returns Astropy unit of the y values of the model.

Return type Unit

property x_lims
Property to access the x limits within which the model is considered valid, the default is None if no x limits were set for the model on instantiation.

Returns A non-scalar astropy quantity with two entries, the first is a lower limit, and the second an upper limit. The default is None if no x limits were set.

Return type Quantity

property par_units
A list of units for the parameters of this model.

Returns A list of astropy units.

Return type List[Unit]

property name
Property getter for the simple name of the model, which the user would enter when requesting a particular model to be fit to a profile, for instance.

Returns String representation of the simple name of the model.

Return type str

property publication_name
Property getter for the publication name of the model, which is what would be added in a plot meant for publication, for instance.

Returns String representation of the publication (i.e. pretty) name of the model.

Return type str

property par_publication_names
Property getter for the publication names of the model parameters. These would be used in a plot for instance, and so can make use of Matplotlib's ability to render LaTeX math code.

Returns List of string representation of the publication (i.e. pretty) names of the model parameters.

Return type List[str]

property describes
A one or two word description of the type of data this model describes.

Returns A string description.

Return type str

property num_pars
Property getter for the number of parameters associated with this model.

Returns Number of parameters.

Return type int

property par_dists
A property that returns the currently stored distributions for the model parameters, by default these will be empty quantities as no fitting will have occurred. Once a fit has been performed involving the model however, the distributions can be set externally.

Returns A list of astropy quantities containing parameter distributions for all model parameters.

Return type List[Quantity]

property fit_warning
Returns any warnings generated by a fitting function that acted upon this model.

Returns A string containing warnings.

Return type str

property acceptance_fraction
Property getter for the acceptance fraction of an MCMC fit (if one has been associated with this model).

Returns The acceptance fraction.

Return type int

property emcee_sampler
Property getter for the emcee sampler used to fit this model, if applicable. By default this will be None, as the it has to be set externally, as the model won't necessarily be fit by emcee

Returns The emcee sampler used to fit this model.

Return type em.EnsembleSampler

property cut_off

Property getter for the number of steps that an MCMC fitting method decided should be removed for burn-in.
By default this will be None, as the it has to be set externally, as the model won't necessarily be fit by emcee

Returns The number of steps to be removed for burn-in.

Return type int

property fit_method
Property getter for the method used to fit the model instance, this will be None if no fit has been run using this model.

Returns The fit method.

Return type str

property par_names
The names of the parameters as they appear in the signature of the model python function.

Returns A list of parameter names.

Return type List[str]

property success
If an fit has been run using this model then this property will tell you whether the fit method considered it to be 'successful' or not. If no fit has been run using this model then the value is None.

Returns Was the fit successful?

Return type bool

7.9.3 models.density module

class xga.models.density.**KingProfile1D**(*x_unit='kpc'*, *y_unit=Unit('solMass / Mpc3')*, *cust_start_pars=None*)
Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of the King profile, describing an isothermal sphere. This describes a radial density profile and assumes spherical symmetry.

static model(*x*, *beta*, *r_core*, *norm*)
The model function for the king profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **beta** (*Quantity*) – The beta slope parameter of the model.
- **r_core** (*Quantity*) – The core radius.
- **norm** (*Quantity*) – The normalisation of the model.

Returns The y values corresponding to the input x values.

Return type Quantity

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)
Calculates the gradient of the king profile at a given point, overriding the numerical method implemented in the BaseModel1D class, as this simple model has an easily derivable first derivative.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.

- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

class xga.models.density.**SimpleVikhlininDensity1D**(*x_unit='kpc'*, *y_unit=Unit('solMass / Mpc3')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of a simplified version of Vikhlinin's full density model. Used relatively recently in https://doi.org/10.1051/0004-6361/201833325 by Ghirardini et al., a simplified form of Vikhlinin's full density model, which can be found in https://doi.org/10.1086/500288.

static model(*x*, *beta*, *r_core*, *alpha*, *r_s*, *epsilon*, *norm*)

The model function for the simplified Vikhlinin density profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **beta** (*Quantity*) – The beta parameter of the model.
- **r_core** (*Quantity*) – The core radius of the model.
- **alpha** (*Quantity*) – The alpha parameter of the model.
- **r_s** (*Quantity*) – The radius near where a change of slope by epsilon occurs.
- **epsilon** (*Quantity*) – The epsilon parameter of the model.
- **norm** (*Quantity*) – The overall normalisation of the model.

Returns The y values corresponding to the input x values.

Return type Quantity

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)

Calculates the gradient of the simple Vikhlinin density profile at a given point, overriding the numerical method implemented in the BaseModel1D class.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

class xga.models.density.**VikhlininDensity1D**(*x_unit='kpc'*, *y_unit=Unit('solMass / Mpc3')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of Vikhlinin's full density model for galaxy cluster intra-cluster medium, which can be found in https://doi.org/10.1086/500288. It is a radial profile, so an assumption of spherical symmetry is baked in.

static model(*x*, *beta_one*, *r_core_one*, *alpha*, *r_s*, *epsilon*, *gamma*, *norm_one*, *beta_two*, *r_core_two*, *norm_two*)

The model function for the full Vikhlinin density profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **beta_one** (*Quantity*) – The beta parameter of the model.
- **r_core_one** (*Quantity*) – The core radius of the model.
- **alpha** (*Quantity*) – The alpha parameter of the model.
- **r_s** (*Quantity*) – The radius near where a change of slope by epsilon occurs.
- **epsilon** (*Quantity*) – The epsilon parameter of the model.
- **gamma** (*Quantity*) – Width of slope change transition region.
- **norm_one** (*Quantity*) – The normalisation of the model first part of the model.
- **beta_two** (*Quantity*) – The beta parameter slope of the small core part of the model.

:param Quantity r_core_two:The core radius of the small core part of the model. :param Quantity norm_two: The normalisation of the additive, small core part of the model.

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)

Calculates the gradient of the full Vikhlinin density profile at a given point, overriding the numerical method implemented in the BaseModel1D class.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

7.9.4 models.fitting module

xga.models.fitting.**log_likelihood**(*theta*, *r*, *y*, *y_err*, *m_func*)

Uses a simple Gaussian likelihood function, returns the logged value.

Parameters

- **theta** (*np.ndarray*) – The knowledge we have (think theta in Bayesian parlance) - gets fed into the model we've chosen.
- **r** (*np.ndarray*) – The radii at which we have measured profile values.
- **y** (*np.ndarray*) – The values we have measured for the profile.

- **y_err** (*np.ndarray*) – The uncertainties on the measured profile values.
- **m_func** – The model function that is being fit to.

Returns The log-likelihood value.

Return type np.ndarray

xga.models.fitting.**log_uniform_prior**(*theta*, *pr*)
This function acts as a uniform prior. Using the limits for the parameters in the chosen model (either user defined or default), the function checks whether the passed theta values sit within those limits. If they do then of course probability is 1, so we return the natural log (as this is a log prior), otherwise the probability is 0, so return -infinity.

Parameters

- **theta** (*np.ndarray*) – The knowledge we have (think theta in Bayesian parlance) - gets fed into the model we've chosen.
- **pr** (*List*) – A list of upper and lower limits for the parameters in theta, the limits of the uniform, uninformative priors.

Returns The log prior value.

Return type float

xga.models.fitting.**log_prob**(*theta*, *r*, *y*, *y_err*, *m_func*, *pr*)
The combination of the log prior and log likelihood.

Parameters

- **theta** (*np.ndarray*) – The knowledge we have (think theta in Bayesian parlance) - gets fed into the model we've chosen.
- **r** (*np.ndarray*) – The radii at which we have measured profile values.
- **y** (*np.ndarray*) – The values we have measured for the profile.
- **y_err** (*np.ndarray*) – The uncertainties on the measured profile values.
- **m_func** – The model function that is being fit to.
- **pr** (*List*) – A list of upper and lower limits for the parameters in theta, the limits of the uniform, uninformative priors.

Returns The log probability value.

Return type np.ndarray

7.9.5 models.misc module

xga.models.misc.**straight_line**(*x_values*, *gradient*, *intercept*)
As simple a model as you can get, a straight line. Possible uses include fitting very simple scaling relations.

Parameters

- **x_values** (*np.ndarray/float*) – The x_values to retrieve corresponding y values for.
- **gradient** (*float*) – The gradient of the straight line.
- **intercept** (*float*) – The intercept of the straight line.

Returns The y values corresponding to the input x values.

Return type Union[np.ndarray, float]

xga.models.misc.**power_law**(*x_values*, *slope*, *norm*)

A simple power law model, with slope and normalisation parameters. This is the standard model for fitting cluster scaling relations in XGA.

Parameters

- **x_values** (*np.ndarray/float*) – The x_values to retrieve corresponding y values for.
- **slope** (*float*) – The slope parameter of the power law.
- **norm** (*float*) – The normalisation parameter of the power law.

Returns The y values corresponding to the input x values.

Return type Union[np.ndarray, float]

7.9.6 models.sb module

class xga.models.sb.**BetaProfile1D**(*x_unit='kpc'*, *y_unit=Unit('ct / (arcmin2 s)')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of the beta profile, essentially a projected isothermal king profile, it can be used to describe a simple galaxy cluster radial surface brightness profile.

static model(*x*, *beta*, *r_core*, *norm*)

The model function for the beta profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **beta** (*Quantity*) – The beta slope parameter of the model.
- **r_core** (*Quantity*) – The core radius.
- **norm** (*Quantity*) – The normalisation of the model.

Returns The y values corresponding to the input x values.

Return type Quantity

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)

Calculates the gradient of the beta profile at a given point, overriding the numerical method implemented in the BaseModel1D class, as this simple model has an easily derivable first derivative.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

inverse_abel(*x*, *use_par_dist=False*, *method='analytical'*)

This overrides the inverse abel method of the model superclass, as there is an analytical solution to the inverse abel transform of the single beta model. The form of the inverse abel transform is that of the king profile, but with an extra transformation applied to the normalising parameter. This method can either return a single value calculated using the current model parameters, or a distribution of values using the parameter distributions (assuming that this model has had a fit run on it).

Parameters

- **x** (*Quantity*) – The x location(s) at which to calculate the value of the inverse abel transform.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a inverse abel transform distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.
- **method** (*str*) – The method that should be used to calculate the values of this inverse abel transform. Default for this overriding method is 'analytical', in which case the analytical solution is used. You may pass 'direct', 'basex', 'hansenlaw', 'onion_bordas', 'onion_peeling', 'two_point', or 'three_point' to calculate the transform numerically.

Returns The inverse abel transform result.

Return type Quantity

class xga.models.sb.**DoubleBetaProfile1D**(*x_unit='kpc'*, *y_unit=Unit('ct / (arcmin2 s)')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of the double beta profile, a summation of two single beta models. Often thought to deal better with peaky cluster cores that you might get from a cool-core cluster, this model can be used to describe a galaxy cluster radial surface brightness profile.

static model(*x*, *beta_one*, *r_core_one*, *norm_one*, *beta_two*, *r_core_two*, *norm_two*)

The model function for the double beta profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **norm_one** (*Quantity*) – The normalisation of the first beta profile.
- **beta_one** (*Quantity*) – The beta slope parameter of the first component beta profile.
- **r_core_one** (*Quantity*) – The core radius of the first component beta profile.
- **norm_two** (*Quantity*) – The normalisation of the second beta profile.
- **beta_two** (*Quantity*) – The beta slope parameter of the second component beta profile.
- **r_core_two** (*Quantity*) – The core radius of the second component beta profile.

Returns The y values corresponding to the input x values.

Return type Quantity

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)

Calculates the gradient of the double beta profile at a given point, overriding the numerical method implemented in the BaseModel1D class, as this simple model has an easily derivable first derivative.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.

- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

inverse_abel (*x*, *use_par_dist=False*, *method='analytical'*)

This overrides the inverse abel method of the model superclass, as there is an analytical solution to the inverse abel transform of the double beta model. The form of the inverse abel transform is that of two summed king profiles, but with extra transformations applied to the normalising parameters. This method can either return a single value calculated using the current model parameters, or a distribution of values using the parameter distributions (assuming that this model has had a fit run on it).

Parameters

- **x** (*Quantity*) – The x location(s) at which to calculate the value of the inverse abel transform.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a inverse abel transform distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.
- **method** (*str*) – The method that should be used to calculate the values of this inverse abel transform. Default for this overriding method is 'analytical', in which case the analytical solution is used. You may pass 'direct', 'basex', 'hansenlaw', 'onion_bordas', 'onion_peeling', 'two_point', or 'three_point' to calculate the transform numerically.

Returns The inverse abel transform result.

Return type Quantity

7.9.7 models.temperature module

class xga.models.temperature.**SimpleVikhlininTemperature1D**(*x_unit='kpc'*, *y_unit=Unit('keV')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of the simplified version of Vikhlinin's temperature model. This is for the description of 3D temperature profiles of galaxy clusters.

static model(*x*, *r_cool*, *a_cool*, *t_min*, *t_zero*, *r_tran*, *c_power*)

The model function for the simplified Vikhlinin temperature profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **r_cool** (*Quantity*) – Parameter describing the radius of the cooler core region.
- **a_cool** (*Quantity*) – Power law parameter for the cooler core region.
- **t_min** (*Quantity*) – A minimum temperature parameter for the model.
- **t_zero** (*Quantity*) – A normalising temperature parameter for the model.

- **r_tran** (*Quantity*) – The radius of the transition region of this broken power law model.
- **c_power** (*Quantity*) – The power law index for the part of the model which describes the outer region of the cluster.

Returns The y values corresponding to the input x values.

Return type Quantity

derivative(*x*, *dx=<Quantity 0.>*, *use_par_dist=False*)

Calculates the gradient of the simple Vikhlinin temperature profile at a given point, overriding the numerical method implemented in the BaseModel1D class.

Parameters

- **x** (*Quantity*) – The point(s) at which the slope of the model should be measured.
- **dx** (*Quantity*) – This makes no difference here, as this is an analytical derivative. It has been left in so that the inputs for this method don't vary between models.
- **use_par_dist** (*bool*) – Should the parameter distributions be used to calculate a derivative distribution; this can only be used if a fit has been performed using the model instance. Default is False, in which case the current parameters will be used to calculate a single value.

Returns The calculated slope of the model at the supplied x position(s).

Return type Quantity

class xga.models.temperature.**VikhlininTemperature1D**(*x_unit='kpc'*, *y_unit=Unit('keV')*, *cust_start_pars=None*)

Bases: *xga.models.base.BaseModel1D*

An XGA model implementation of the full version of Vikhlinin's temperature model. This is for the description of 3D temperature profiles of galaxy clusters.

static model(*x*, *r_cool*, *a_cool*, *t_min*, *t_zero*, *r_tran*, *a_power*, *b_power*, *c_power*)

The model function for the full Vikhlinin temperature profile.

Parameters

- **x** (*Quantity*) – The radii to calculate y values for.
- **r_cool** (*float*) – Parameter describing the radius of the cooling region (I THINK - NOT CERTAIN YET).
- **a_cool** (*float*) – Power law parameter for the cooling region (I THINK - NOT CERTAIN YET).
- **t_min** (*float*) – A minimum temperature parameter for the model (I THINK - NOT CERTAIN YET).
- **t_zero** (*float*) – A normalising temperature parameter for the model (I THINK - NOT CERTAIN YET).
- **r_tran** (*float*) – The radius of the transition region of this broken power law model.
- **a_power** (*float*) – The first power law index.
- **b_power** (*float*) – The second power law index.
- **c_power** (*float*) – the third power law index.

Returns The y values corresponding to the input x values.

Return type Quantity

CHAPTER

EIGHT

PLANNED XGA FEATURES

- **Upper Limit X-ray Luminosities** - These luminosities are measured from photometric products when there are not sufficient X-ray counts to generate a spectrum. This will involve integrating a tool that I have already written for XCS into the structure of XGA, though rather than measuring upper limit luminosities from a single image, I intend to measure them from the combined data.
- **Source Finder** - It is likely that I will extend my Hierarchical Clustering Peak Finder into a full source finder, as well as completing the implementation of a convolutional peak finder/source finder, which was inspired by my friend and colleague Lucas Porth.
- **More XSPEC Models** - Including a two temperature APEC model, and support for custom user defined models.
- **Implement emcee scaling relation fitter** - What it says on the tin really. Implement an emcee fitter for the scaling relations so that there is an MCMC option that doesn't require the installation of a bunch of optional dependencies.
- **Ability to save ScalingRelation objects** - The ability to save ScalingRelation objects to disk in some way, so that code to generate them doesn't need to be run multiple times.
- **Support for other X-ray telescopes** - Support for generation and analysis of data products from other telescopes (I guess than XGA will come to mean 'X-ray: Generate and Analyse', rather than 'XMM: Generate and Analyse').
- **Creating a Docker image for users to download** - Creating a Docker environment with SAS and HEASoft already installed, for ease of use.
- **Method for finding poorly removed point sources** - An attempt to mitigate occasional mistakes by source finders that produce regions that don't necessarily remove the entire point source.
- **Overdensity radius measurement** - Using HydrostaticMass profiles to measure the overdensity radius of a cluster.

CHAPTER

NINE

XGA PUBLICATIONS

This page contains a list of publications (that I'm aware of) that use XGA in some way. If you have published work using XGA please cite the XGA software paper, which is the first entry in this table, and contact me so I can add your work to this page.

Author	Year	DOI	NASA/ADS	Notes
D. J. Turner	In Prep			XGA Software Paper
D. J. Turner	In Prep			eFEDS-XCS Cluster Comparison

CHAPTER

TEN

GETTING HELP AND SUPPORT

If you encounter a bug, or would like to make a feature request, please use the GitHub issues page, it really helps to keep track of everything.

However, if you have further questions, or just want to make doubly sure I notice the issue, please feel free to send me an email at david.turner@sussex.ac.uk, I will be happy to help you.

If you have a specific feature request, or wish to contribute to XGA, and opening an issue isn't sufficient to communicate your idea properly then please send me an email and we can arrange a virtual meeting.

PYTHON MODULE INDEX

X

INDEX

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y