deproject Documentation

Release 0.2.0

Tom Aldcroft

Mar 14, 2019

Introduction

`Deproject` is a CIAO Sherpa extension package to facilitate deprojection of two-dimensional circular annular X-ray spectra to recover the three-dimensional source properties. For typical thermal models this would include the radial temperature and density profiles. This basic method has been used extensively for X-ray cluster analysis and is the basis for the XSPEC model projct. The `deproject` module brings this functionality to *Sherpa* as a Python module that is straightforward to use and understand.

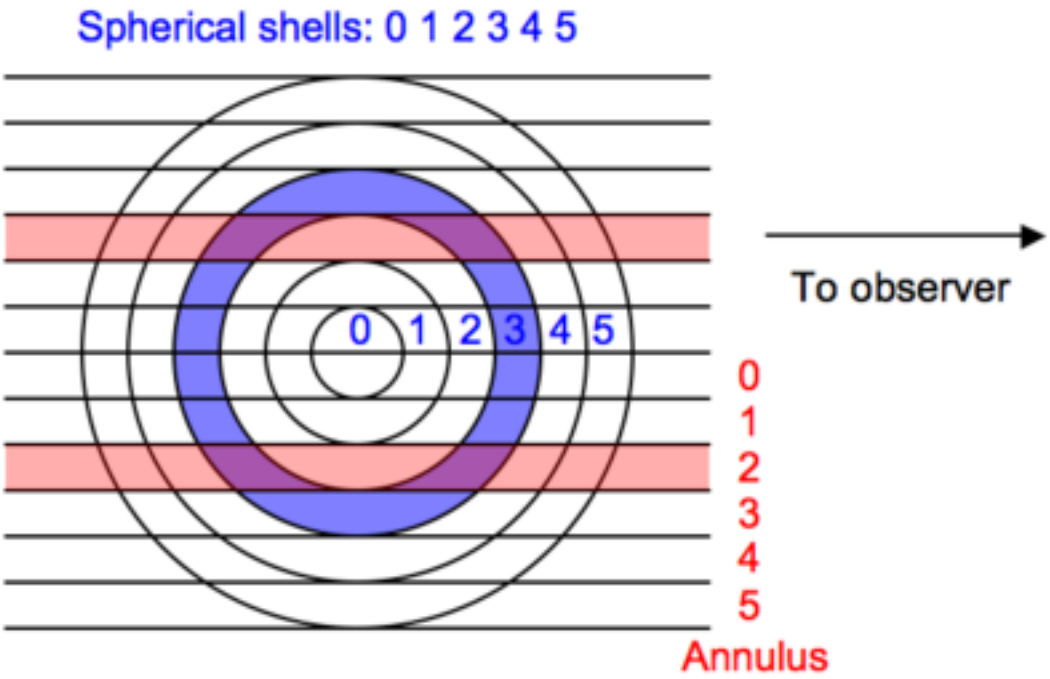
The module can also be used with the standalone Sherpa release, but as it requires support for XSPEC models - for the thermal plasma emission codes - the documentation will focus on using Sherpa with a CIAO environment.

CHAPTER 1

Overview

The `deproject` module uses `specstack` to allow for manipulation of a stack of related input datasets and their models. Most of the functions resemble ordinary *Sherpa* commands (e.g. set_par, set_source, ignore) but operate on a stack of spectra.

The basic physical assumption of `deproject` is that the extended source emissivity is constant and optically thin within spherical shells whose radii correspond to the annuli used to extract the specta. Given this assumption one constructs a model for each annular spectrum that is a linear volume-weighted combination of shell models. The geometry is illustrated in the figure below (which would be rotated about the line to the observer in three-dimensions):



1.1 Model creation

It is assumed that prior to starting `deproject` the user has extracted source and background spectra for each annulus. By convention the annulus numbering starts from the inner radius at 0 and corresponds to the dataset `id` used within *Sherpa*. It is not required that the annuli include the center but they must be contiguous between the inner and outer radii.

Given a spectral model `M[s]` for each shell `s`, the source model for dataset `a` (i.e. annulus `a`) is given by the sum over `s >= a` of `vol_norm[s,a] * M[s]` (normalized volume * shell model). The image above shows shell 3 in blue and annulus 2 in red. The intersection of (purple) has a physical volume defined as `vol_norm[3,2] * v_sphere` where `v_sphere` is the volume of the sphere enclosing the outer shell (as *discussed below*).

The bookkeeping required to create all the source models is handled by the `deproject` module.

1.2 Fitting

Once the composite source models for each dataset are created the fit analysis can begin. Since the parameter space is typically large the usual procedure is to initally fit using the "onion-peeling" method:

- First fit the outside shell model using the outer annulus spectrum
- Freeze the model parameters for the outside shell
- Fit the next inward annulus / shell and freeze those parameters
- Repeat until all datasets have been fit and all shell parameters determined.

From this point the user may choose to do a simultanenous fit of the shell models, possibly freezing some parameters as needed. This process is made manageable with the `specstack` methods that apply normal *Sherpa* commands like freeze or set_par to a stack of spectral datasets.

1.3 Densities

Physical densities (cm^{-3}) for each shell can be calculated with `deproject` assuming the source model is based on a thermal model with the "standard" normalization (from the XSPEC documentation):

$$\mathrm{norm} = \frac{10^{-14}}{4\pi[D_A(1+z)]^2} \int n_e n_H dV$$

where D_A is the angular size distance to the source (in cm), z is the source redshift, and n_e and n_H are the electron and Hydrogen densities (in cm^{-3}).

Inverting this equation and assuming constant values for n_e and n_H gives

$$n_e = \sqrt{\frac{4\pi\,\mathrm{norm}\,[D_A(1+z)]^2\,10^{14}}{\alpha V}}$$

where V is the volume, $n_H = \alpha n_e$, and α is taken to be 1.18 (and this ratio is constant within the source).

Recall that the model components for each volume element (intersection of the annular cylinder `a` with the spherical shell `s`) are multiplied by a volume normalization:

$$V_{\mathrm{norm}}[s,a] = V[s,a]/V_{\mathrm{sphere}}$$

where V_{sphere} is the volumne of the sphere enclosing the outermost annulus.

With this convention the volume (V) used above in calculating the electron density for each shell is always V_{sphere}.

CHAPTER 2

Installation

As of *version 0.2.0*, the `deproject` package can be installed directly from PyPI. This requires CIAO 4.11 or later (as installation with pip in earlier versions of CIAO is not well supported). The module *can* be used with the standalone release of Sherpa, but it is only useful if Sherpa has been built with XSPEC support which is trickier to achieve than we would like.

2.1 Requirements

The package uses Astropy and SciPy, for units support and cosmological-distance calculations. It is assumed that Matplotlib is available for plotting (which is included in CIAO 4.11).

2.2 Using pip

2.2.1 CIAO

Installation within CIAO requires:

- having sourced the CIAO initialisation script (e.g. *ciao.csh* or *ciao.bash*);
- and using a constraints file, to avoid updating NumPy in CIAO.

It should be as simple as:

```
echo "numpy==1.12.1" > constraints.txt
pip install -c constraints.txt 'astropy<3.1' deproject
```

Note: The constraints are for CIAO 4.11, please adjust accordingly if using a newer version of CIAO (the Astropy restriction is because version 3.1 requires NumPy version 1.13 or later but CIAO 4.11 is shipped with NumPy version 1.12).

2.2.2 Standalone

When using the standalone Sherpa intallation, the following should be all that is required:

```
pip install deproject
```

2.3 Manual installation

The source is available on github at https://github.com/sherpa-deproject/deproject, with releases available at https://github.com/sherpa-deproject/deproject/releases.

After downloading the source code (whether from a release or by cloning the repository) and moving into the directory (*deproject-<version>* or *deproject*), installation just requires:

```
python setup.py install
```

Note: This command should only be run after setting up CIAO or whatever Python environment contains your Standalone Sherpa installation.

2.4 Test

The source installation includes a basic test suite, which can be run with either

```
pytest
```

or

```
python setup.py test
```

2.5 Example data

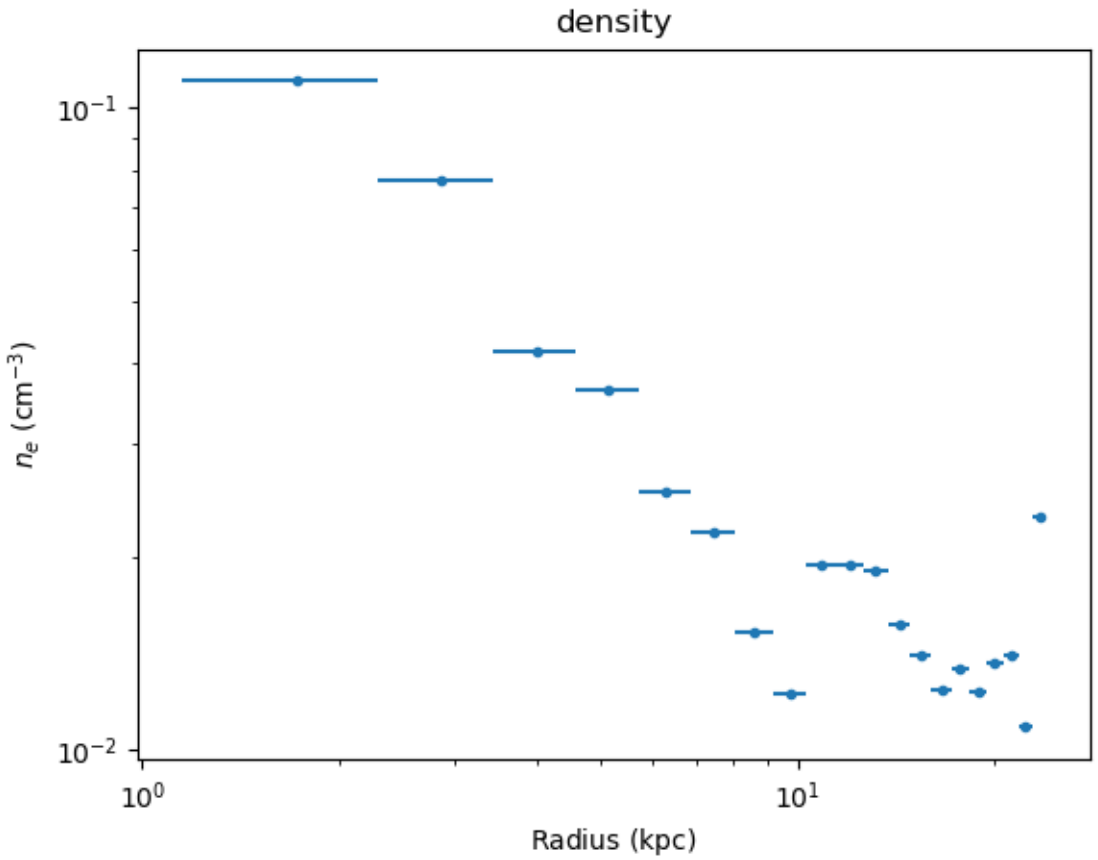
The example data can be download from either http://cxc.cfa.harvard.edu/contrib/deproject/downloads/m87.tar.gz or from GitHub. The source distribution includes scripts - in the examples directory - that can be used to replicate both the *basic M87 example* and the follow-on example *combining annuli*.

As an example (from within an IPython session, such as the Sherpa shell in CIAO):

```
>>> %run fit_m87.py
...
... a lot of screen output will whizz by
...
```

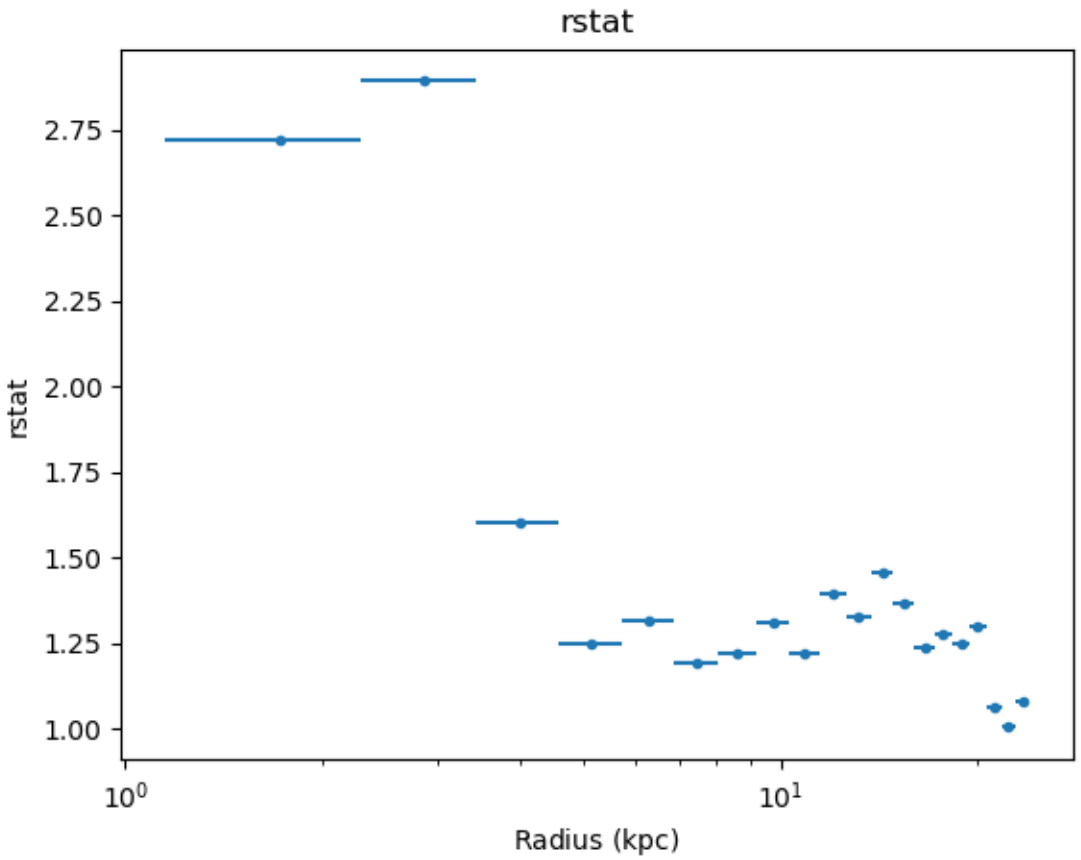
The current density estimates can then be displayed with:

```
>>> dep.density_plot()
```



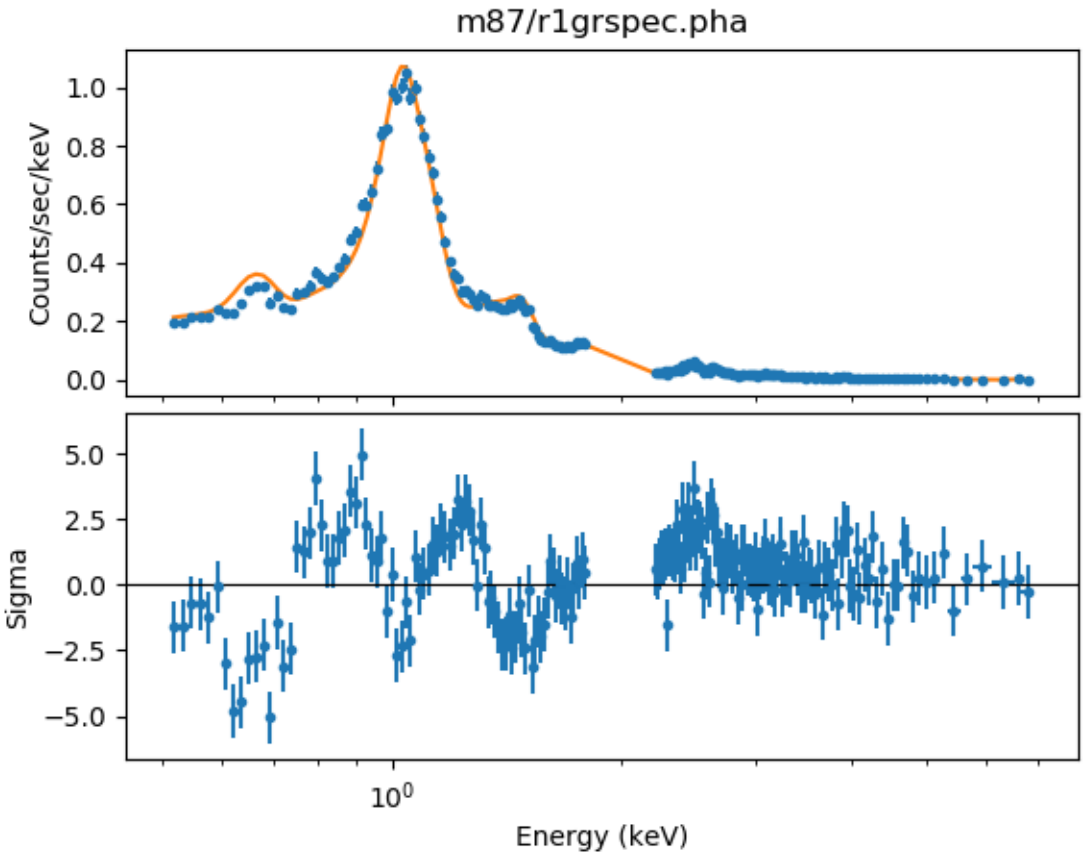
the reduced-statistic for the fit to each shell with:

```
>>> dep.fit_plot('rstat')
```



and the fit results for the first annulus can be displayed using the Sherpa functions:

```
>>> set_xlog()
>>> plot_fit_delchi(0)
```


m87/r1grspec.pha
Counts/sec/keV
1.0
0.8
0.6
0.4
0.2
0.0
Sigma
5.0
2.5
0.0
−2.5
−5.0
10^0
Energy (keV)

CHAPTER 3

Changes

3.1 Version 0.2.0

3.1.1 Overview

The code has been updated to run with Python 3 and can now be installed from PyPI. Documentation has been moved to Read The Docs.

The deproject module now requires Astropy, which can be installed within CIAO 4.11. The three main areas where Astropy functionality is used are:

- the use of Astropy Quantity values (both for arguments to methods and returned values);
- Astropy Data Tables are used to return tabular data;
- and Cosmology calculations now use the Astropy cosmology module rather than the *cosmocalc* module.

The *`deproject_from_xflt()`* helper function has been introduced, which uses the `XFLT0001` to `XFLT0005` keywords in the input files to determine the annulus parameters (radii and covering angle). The covering angle (θ) can now vary per annulus.

Error values can now be generated using the onion-peeling approach, for the confidence and covariance methods, and the values are returned as an Astropy Table. Parameter values can now be tied together (to combine annuli to try and avoid "ringing"). There is improved support for accessing and plotting values.

3.1.2 Details

The code has been re-arranged into the `deproject` package, which means that you really should say `from deproject.deproject import Deproject`, but the `deproject` module re-exports *`deproject.deproject`* so that existing scripts still work, and you do not not have to type in the same word multiple times! The package has been updated so that it is available on PyPI.

The scaling between shells (calculated from the intersection between spheres and cylinders) was limited to 5 decimal places, which could cause problems with certain choices of annuli (such as an annulus making no contribution to interior annuli). This restriction has been removed.

Added support for per-annulus `theta` values (that is, each annulus can have a different opening angle). The `radii`, `theta`, and `angdist` parameters to *Deproject* all now require values that is an Astropy quantity rather than a dimensionless value.

Added the *deproject_from_xflt()* helper function, which creates a *Deproject* instance from PHA files which contain the XSPEC XFLT0001 to XFLT0005 keywords (as used by the projct model), rather than specifying the values from the command line. The routine will error out if the keywords indicate elliptical annuli, and the default is to assume the radii are in arcseconds, but a scaling factor can be given if the radii are in some other units (such as pixels).

Added the *guess()* method to do an initial fit to each annulus, following the approach suggested in the XSPEC documentation for projct, by just fitting the individual (not de-projected) models to each annulus. This can help speed up the deproject fit - *fit()* - as well as help avoid the fit getting stuck in a local minimum.

Added *covar()* and *conf()* methods that estimate errors - using the covariance and confidence methods respectively - using the onion-skin model (i.e. the errors on the outer annuli are evaluated, then this component is frozen and the errors on the next annulus are evaluated).

The *fit()*, *conf()*, and *covar()* methods now all return Astropy Tables containing the results per annulus. These values can also be retrieved with the *get_fit_results()*, *get_conf_results()*, or *get_covar_results()* methods. A number of columns (radii and density) are returned as Astropy quantities.

The `cosmocalc` module has been removed and the Astropy cosmology module is used instead. This is only used if the angular-diameter distance to the source is calculated rather than explicitly given. The default cosmology is now set to Planck15.

Values, as a function of radius, can be plotted with a number of new methods: *fit_plot()*, *conf_plot()*, and *covar_plot()* display the last fit results (with the last two including error estimates), and the *par_plot()* and *density_plot()* methods show the current values. These support a number of options, including switching between angular and physical distances for the radii.

The *get_shells()* method has been added to make it easy to see which annuli are combined together, and the *get_radii()* method to find the radii of the annuli (in a range of units).

Added the `tie_par()` and `untie_par()` methods to make it easy to tie (or untie) parameters in neighbouring annuli. The onion-skin approach - used when fitting or running an error analysis - recognizes annuli that are tied together and fits these simultaneously, rather than individually.

The *set_source()* method can now be called multiple times (previously it would lead to an error).

Added error checking for several routines, such as `thaw()` when given an unknown parameter name.

Updated to support Python 3.5 and to have better support when the `pylab` backend is selected. Support for the ChIPS backend is limited. A basic test suite has been added.

3.2 Version 0.1.0

Initial version.

CHAPTER 4

To Do

- Use the Python logging module to produce output and allow for a verbosity setting. [Easy]
- Move from using obsid to a more-generic label for the multi-data-set case (i.e. when there are multiple data sets in a given annulus)
- Allow the theta value to vary between data set in each annulus (for the multi-data-set case)
- Support elliptical annuli
- Create and use more generalized `ModelStack` and `DataStack` classes to allow for general mixing models. [Hard]
- Add summary methods (similar to Sherpa's show series of commands)
- Add a specialized version of *plot_source_component* which takes a general model expression (i.e. without the *_annulus* suffix) and displays all combos per annulus. A bit tricky to get right. There's also a *plot_model_component*.

CHAPTER 5

M87

Now we step through in detail the `fit_m87.py` script in the `examples` directory to explain each step and illustrate how to use the `deproject` module. This script should serve as the template for doing your own analysis.

This example uses extracted spectra, response products, and analysis results for the Chandra observation of M87 (obsid 2707). These were kindly provided by Paul Nulsen. Results based on this observation can be found in Forman et al 2005 and via the CXC Archive Obsid 2707 Publications list.

The examples assume this is being run directly from the Sherpa shell, which has already imported the Sherpa module. If you are using IPython or a Jupyter notebook then the following command is needed:

```
>>> from sherpa.astro.ui import *
```

5.1 Set up

The first step is to load in the Deproject class:

```
>>> from deproject import Deproject
```

and then set a couple of constants (note that both the angular-diameter distance and thta values *must* be Astropy quantites):

```
>>> from astropy import units as u
>>> redshift = 0.004233                    # M87 redshift
>>> angdist = 16 * u.Mpc                   # M87 distance
>>> theta = 75 * u.deg                     # Covering angle of sectors
```

Next we create an array of the the annular radii in arcsec. The numpy.arange method here returns an array from 30 to 640 in steps of 30. These values were in pixels in the original spectral extraction so we multiply by the pixel size (0.492 arcseconds) to convert to an angle.:

```
>>> radii = numpy.arange(30., 640., 30) * 0.492 * u.arcesc
```

The `radii` parameter must be a list of values that starts with the inner radius of the inner annulus and includes each radius up through the outer radius of the outer annulus. Thus the `radii` list will be one element longer than the number of annuli.

```
>>> print(radii)
[ 14.76  29.52  44.28  59.04  73.8   88.56 103.32 118.08 132.84 147.6
 162.36 177.12 191.88 206.64 221.4  236.16 250.92 265.68 280.44 295.2
 309.96] arcsec
```

Now the key step of creating the `Deproject` object `dep`. This object is the interface to the all the `deproject` methods used for the deprojection analysis.

```
>>> dep = Deproject(radii, theta=theta, angdist=angdist)
```

If you are not familiar with object oriented programming, the `dep` object is just a thingy that stores all the information about the deprojection analysis (e.g. the source redshift, PHA file information and the source model definitions) as object *attributes*. It also has object *methods* (i.e. functions) you can call such as `dep.get_par(parname)` or `dep.load_pha(file)`. The full list of attributes and methods are in the `deproject` module documentation.

In this particular analysis the spectra were extracted from a 75 degree sector of the annuli, hence `theta` is set in the object initialization, where the units are set explicitly using the Astropy support for units. Note that this parameter only needs to be set if any of the annuli are sectors rather than the full circle. Since the redshift is **not** a good distance estimator for M87 we also explicitly set the angular size distance using the `angdist` parameter.

5.2 Load the data

Now load the PHA spectral files for each annulus using the Python `range` function to loop over a sequence ranging from 0 to the last annulus. The *`load_pha()`* call is the first example of a `deproject` method (i.e. function) that mimics a *Sherpa* function with the same name. In this case `dep.load_pha(file, annulus)` loads the PHA file using the *Sherpa* load_pha function but also registers the dataset in the spectral stack:

```
>>> for annulus in range(len(radii) - 1):
...     dep.load_pha('m87/r%dgrspec.pha' % (annulus + 1), annulus)
```

Note: The `annulus` parameter is required in `dep.load_pha()` to allow multiple data sets per annulus, such as repeated Chandra observations or different XMM instruments.

5.3 Create the model

With the data loaded we set the source model for each of the spherical shells with the *`set_source()`* method. This is one of the more complex bits of `deproject`. It automatically generates all the model components for each shell and then assigns volume-weighted linear combinations of those components as the source model for each of the annulus spectral datasets:

```
>>> dep.set_source('xswabs * xsmekal')
```

The model expression can be any valid *Sherpa* model expression with the following caveats:

- Only the generic model type should be specified in the expression. In typical *Sherpa* usage one generates the model component name in the model expression, e.g. `set_source('xswabs.abs1 * xsmekal.`

mek1'). This would create model components named abs1 and mek1. In dep.set_source() the model component names are auto-generated as <model_type>_<shell>.

- Only one of each model type can be used in the model expression. A source model expression like "xsmekal + gauss1d + gauss1d" would result in an error due to the model component auto-naming.

Next any required parameter values are set and their freeze or thaw status are set.

```
>>> dep.set_par('xswabs.nh', 0.0255)
>>> dep.freeze('xswabs.nh')

>>> dep.set_par('xsmekal.abundanc', 0.5)
>>> dep.thaw('xsmekal.abundanc')

>>> dep.set_par('xsmekal.redshift', redshift)
```

As a convenience if any of the model components have a redshift parameter that value will be used as the default redshift for calculating the angular size distance.

5.4 Define the data to fit

Now the energy range used in the fitting is restricted using the stack version of the *Sherpa* ignore command. The notice command is also available.

```
>>> dep.ignore(None, 0.5)
>>> dep.ignore(1.8, 2.2)
>>> dep.ignore(7, None)
```

5.5 Define the optimiser and statistic

At this point the model is completely set up and we are ready to do the initial "onion-peeling" fit. As for normal high-signal fitting with binned spectra we issue the commands to set the optimization method and the fit statistic.

In this case we use the Levenberg-Marquardt optimization method with the χ^2 fit statistic, where the variance is estimated using a similar approach to XSPEC.

```
>>> set_method("levmar")
>>> set_stat("chi2xspecvar")
>>> dep.subtract()
```

Note: The chi2xspecvar statistic is used since the values will be *compared to results from XSPEC*.

5.6 Seeding the fit

We first try to "guess" a good starting point for the fit (since the default fit parameters, in particular the normalization, are often not close to the expected value). In this case the *guess()* method implements a scheme suggested in the projct documentation, which fits each annulus with an un-projected model and then corrects the normalizations for the shell/annulus overlaps:

```
>>> dep.guess()
Projected fit to annulus 19 dataset: 19
Dataset               = 19
...
Change in statistic   = 4.60321e+10
   xsmekal_19.kT   2.748        +/- 0.0551545
   xsmekal_19.Abundanc   0.429754     +/- 0.0350466
   xsmekal_19.norm   0.00147471   +/- 2.45887e-05
Projected fit to annulus 18 dataset: 18
Dataset               = 18
...
Change in statistic   = 4.61945e+10
   xsmekal_18.kT   2.68875      +/- 0.0543974
   xsmekal_18.Abundanc   0.424198     +/- 0.0335045
   xsmekal_18.norm   0.00147443   +/- 2.43454e-05
...
Projected fit to annulus 0 dataset: 0
Dataset               = 0
...
Change in statistic   = 2.15043e+10
   xsmekal_0.kT   1.57041      +/- 0.0121572
   xsmekal_0.Abundanc   1.05816      +/- 0.0373779
   xsmekal_0.norm   0.00152496   +/- 2.93297e-05
```

As shown in the screen output, the `guess` routine fits each annulus in turn, from outer to inner. After each fit, the normalization (the `xsmekal_*.norm` terms) are adjusted by the filling factor of the shell.

Note: The `guess` step is *optional*. Parameter values can also be set, either for an individual annulus with the Sherpa set_par function, such as:

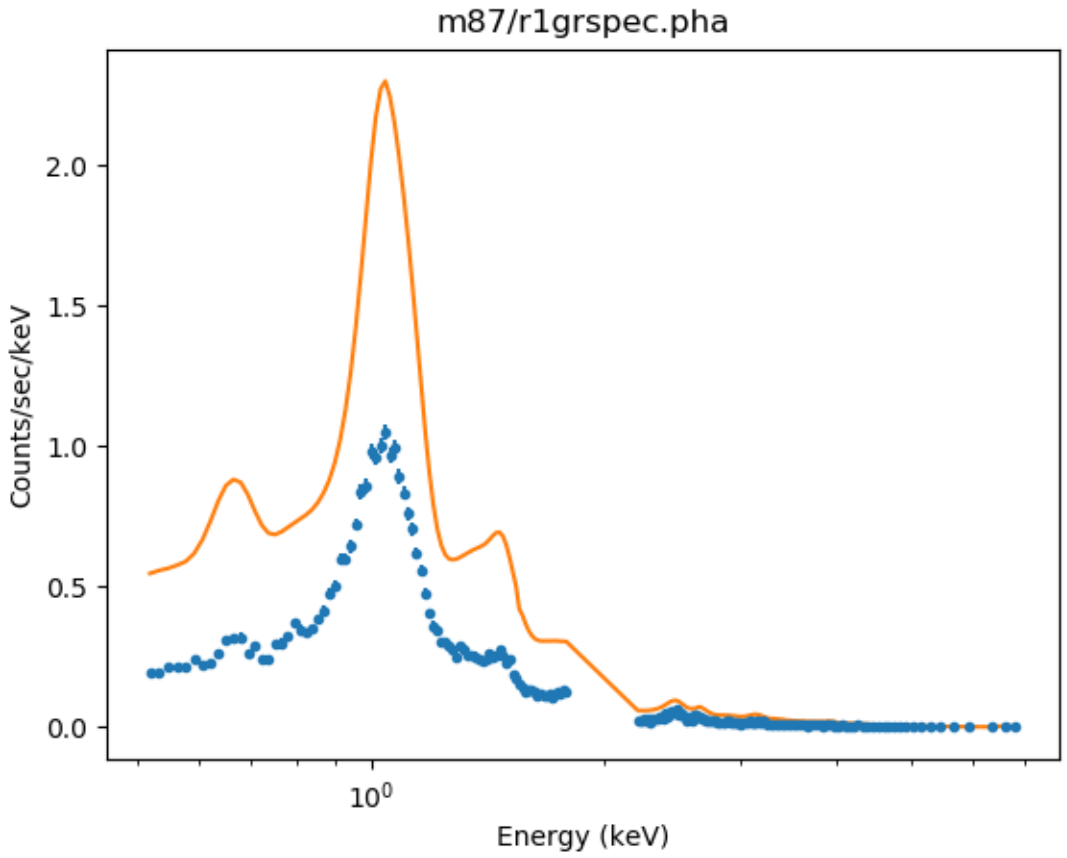
```
set_par('xsmekal_0.kt', 1.5)
```

or to all annuli with the Deproject method *`set_par()`*:

```
dep.set_par('xsmekal.abundanc', 0.3)
```

The data can be inspected using normal Sherpa commands. The following shows the results of the guess for dataset 0, which corresponds to the inner-most annulus (the *`datasets`* attribute can be used to map between annuus and dataset identifier).

```
>>> set_xlog()
>>> plot_fit(0)
```



The overall shape of the model looks okay, but the normalization is not (since it has been adjusted by the volume-filling factor of the intersection of the annulus and shell). The gap in the data around 2 keV is because we *explicitly excluded this range* from the fit.

Note: The *Deproject* class also provides methods for plotting each annulus in a separate window, such as *plot_data()* and *plot_fit()*.

5.7 Fitting the data

The *fit()* method performs the full onion-skin deprojection (the output is similar to the `guess` method, with the addition of parameters being frozen as each annulus is processed):

```
>>> onion = dep.fit()
Fitting annulus 19  dataset: 19
Dataset               = 19
...
Reduced statistic     = 1.07869
Change in statistic   = 1.29468e-08
   xsmekal_19.kT   2.748        +/- 0.0551534
   xsmekal_19.Abundanc   0.429758     +/- 0.0350507
   xsmekal_19.norm   0.249707     +/- 0.00416351
```

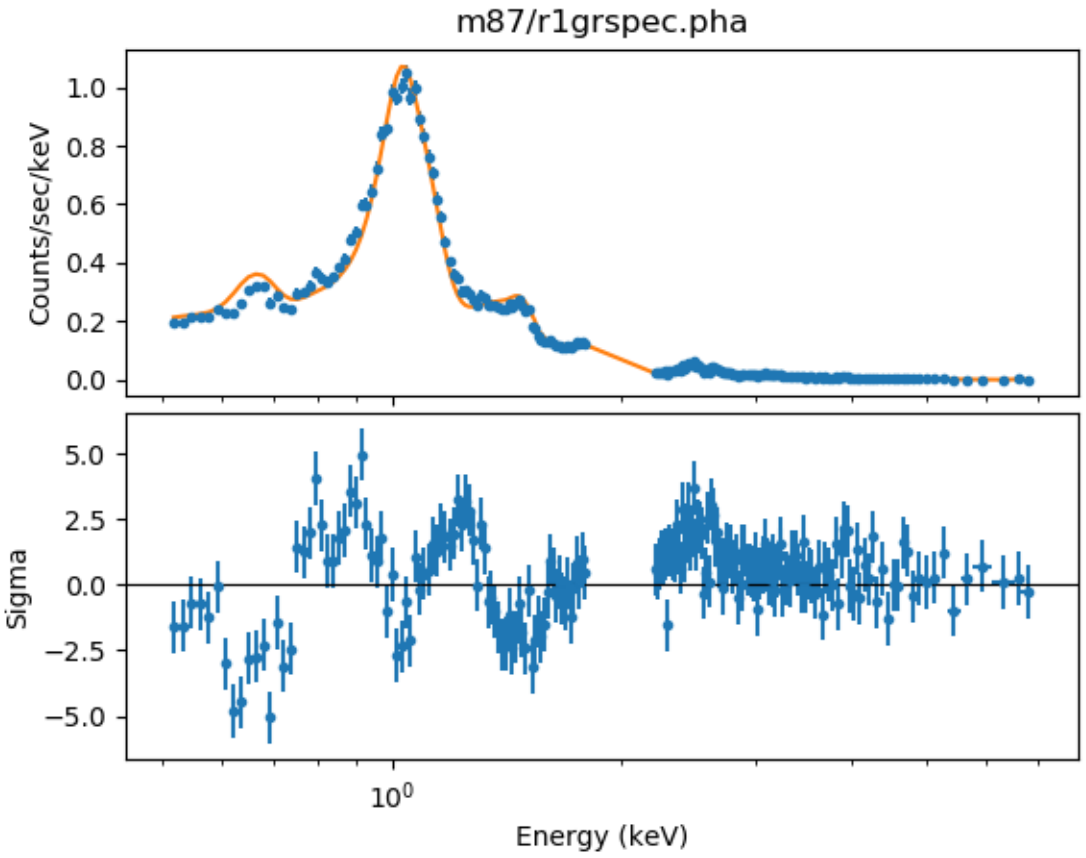
(continues on next page)

(continued from previous page)

```
Freezing xswabs_19
Freezing xsmekal_19
Fitting annulus 18  dataset: 18
Dataset               = 18
...
Reduced statistic     = 1.00366
Change in statistic   = 13444.5
   xsmekal_18.kT   2.41033      +/- 0.274986
   xsmekal_18.Abundanc   0.375824     +/- 0.136926
   xsmekal_18.norm   0.0551815    +/- 0.00441495
Freezing xswabs_18
Freezing xsmekal_18
...
Freezing xswabs_1
Freezing xsmekal_1
Fitting annulus 0  dataset: 0
Dataset               = 0
...
Reduced statistic     = 2.72222
Change in statistic   = 12115.6
   xsmekal_0.kT   1.63329      +/- 0.0278325
   xsmekal_0.Abundanc   1.1217       +/- 0.094871
   xsmekal_0.norm   5.7067       +/- 0.262766
Change in statistic   = 13699.6
   xsmekal_0.kT   1.64884      +/- 0.0242336
   xsmekal_0.Abundanc   1.14629      +/- 0.0903398
   xsmekal_0.norm   5.68824      +/- 0.245884
...
```

The fit to the central annulus (dataset 0) now looks like:

```
>>> plot_fit_delchi(0)
```



After the fit process each shell model has an association normalization that can be used to calculate the densities. This is where the source angular diameter distance is used. If the angular diameter distance is not set explicitly in the original `dep = Deproject(...)` command then it is calculated automatically from the source redshift and an assumed Cosmology, which if not set is taken to be the Planck15 model provided by the astropy.cosmology module.

One can examine the values being used as follows (note that the *angdist* attribute is an Astropy Quantity):

```
>>> print("z={:.5f} angdist={}".format(dep.redshift, dep.angdist))
z=0.00423 angdist=16.0 Mpc
```

New in version 0.2.0 is the behavior of the `fit` method, which now returns an Astropy table which tabulates the fit results as a function of annulus. This *includes* the electron density, which can also be retrieved with *get_density()*. The fit results can also be retrieved with the *get_fit_results()* method.

```
>>> print(onion)
annulus rlo_ang rhi_ang ...     xsmekal.norm          density
         arcsec  arcsec ...                          1 / cm3
------- ------- ------- ... ------------------- --------------------
      0   14.76   29.52 ...  5.6882389946381275   0.1100953546292787
      1   29.52   44.28 ...  2.8089409208987233  0.07736622021374819
      2   44.28   59.04 ...   0.814017947154132  0.04164827967805805
      3   59.04    73.8 ...  0.6184339453006981  0.03630168106524076
      4    73.8   88.56 ...  0.2985327157676323 0.025221797991301052
      5   88.56  103.32 ... 0.22395312678845017 0.021845331641349316
    ...     ...     ... ...                 ...                  ...
     13  206.64   221.4 ... 0.07212121560592619 0.012396857131392835
```

(continues on next page)

(continued from previous page)

```
     14   221.4  236.16 ... 0.08384338967492334  0.01336640115325031
     15  236.16  250.92 ... 0.07104455447410102 0.012303975980575187
     16  250.92  265.68 ... 0.08720295254137615 0.013631563529090736
     17  265.68  280.44 ... 0.09192970392746878 0.013996131292837352
     18  280.44   295.2 ... 0.05518150227052414 0.010843683594144967
     19   295.2  309.96 ... 0.24970680803387219 0.023067220584935984
Length = 20 rows
```

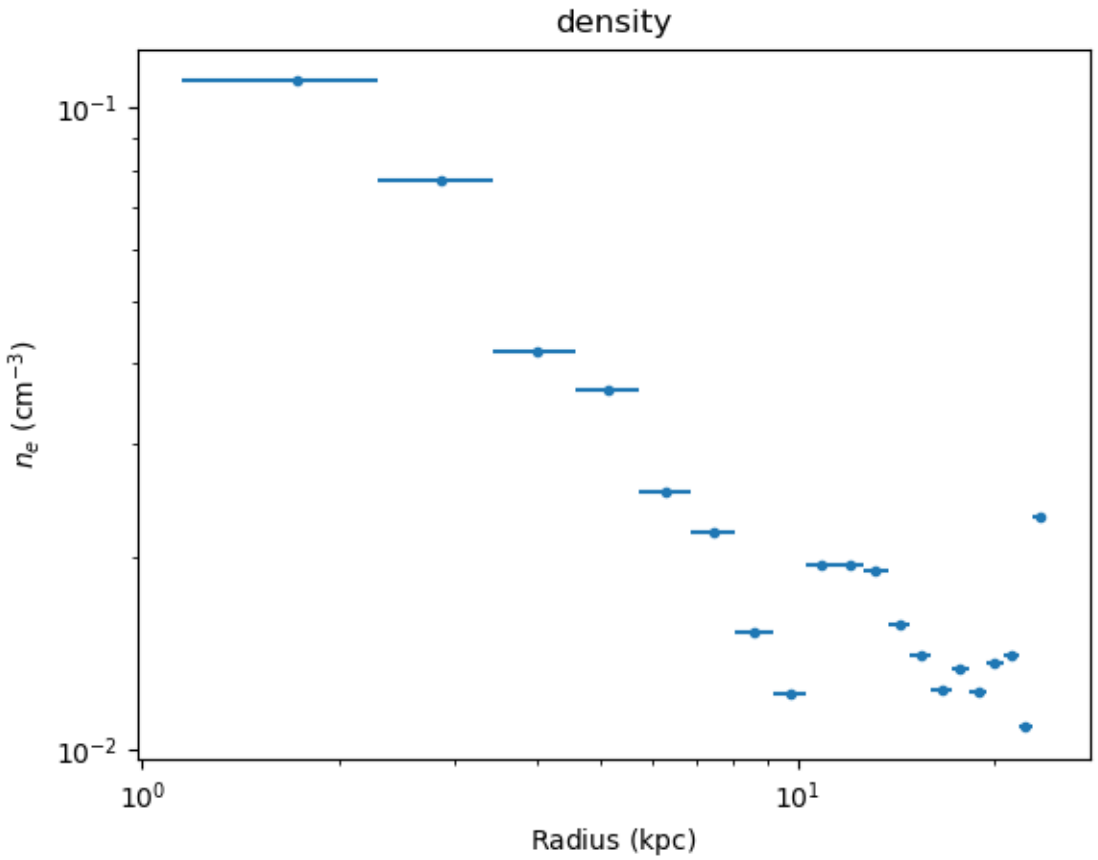
5.8 Inspecting the results

The electron density can be retrieved directly from the fit results, with the *get_density()* method:

```
>>> print(dep.get_density())
[ 0.11009535  0.07736622  0.04164828  0.03630168  0.0252218   0.02184533
  0.01525456  0.01224972  0.01942528  0.01936785  0.01905983  0.01568478
  0.01405426  0.01239686  0.0133664   0.01230398  0.01363156  0.01399613
  0.01084368  0.02306722] 1 / cm3
```

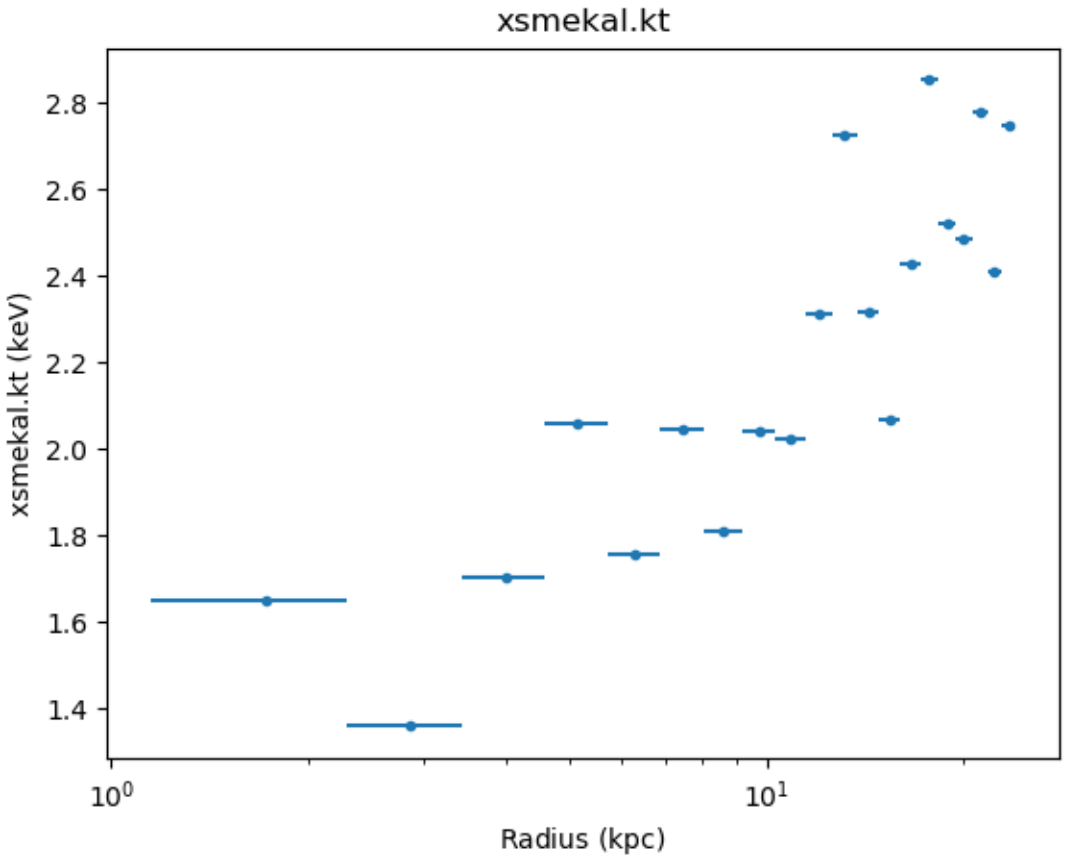
or plotted using *density_plot()*:

```
>>> dep.density_plot()
```



The temperature profile from the deprojection can be plotted using *par_plot()*:

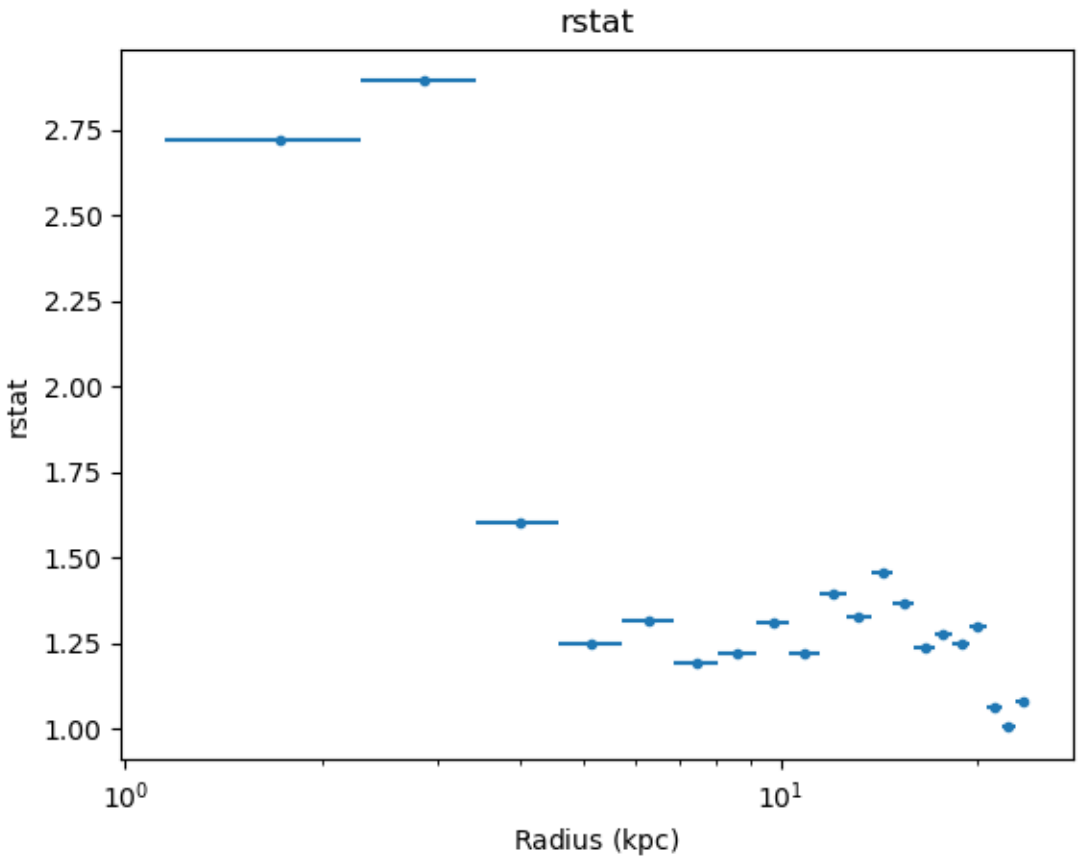
```
>>> dep.par_plot('xsmekal.kt')
```



The unphysical temperature oscillations seen here highlights a known issue with this analysis method (e.g. Russell, Sanders, & Fabian 2008).

It can also be instructive to look at various results from the fit, such as the reduced statistic for each annulus (as will be shown below, there are *fit_plot()*, *conf_plot()*, and *covar_plot()* variants):

```
>>> dep.fit_plot('rstat')
```



5.9 Error analysis

Errors can also be calculated, on both the model parameters and the derived densities, with either the *conf()* or *covar()* methods. These apply the confidence and covariance error-estimation routines from Sherpa using the same onion-skin model used for the fit, and are new in version 0.2.0. In this example we use the covariance version, since it is generally faster, but confidence is the recommended routine as it is more robust (and calculates asymmetric errors):

```
>>> errs = dep.covar()
```

As with the `fit` method, both `conf` and `covar` return the results as an Astropy table. These can also be retrieved with the *get_conf_results()* or *get_covar_results()* methods. The columns depend on the command (i.e. fit or error results):

```
>>> print(errs)
annulus rlo_ang rhi_ang ...        density_lo             density_hi
         arcsec  arcsec ...         1 / cm3                1 / cm3
------- ------- ------- ... ----------------------- ----------------------
      0   14.76   29.52 ...  -0.0023299300992805777  0.0022816336635322065
      1   29.52   44.28 ...  -0.0015878693875514133  0.0015559288091097495
      2   44.28   59.04 ...  -0.0014852395638492444  0.0014340671599212054
      3   59.04    73.8 ...  -0.0011539611188859725    0.00111839214283211
      4    73.8   88.56 ...   -0.001166653616914693   0.001115024462355424
      5   88.56  103.32 ...  -0.0010421595234548324  0.0009946565126391325
```

(continues on next page)

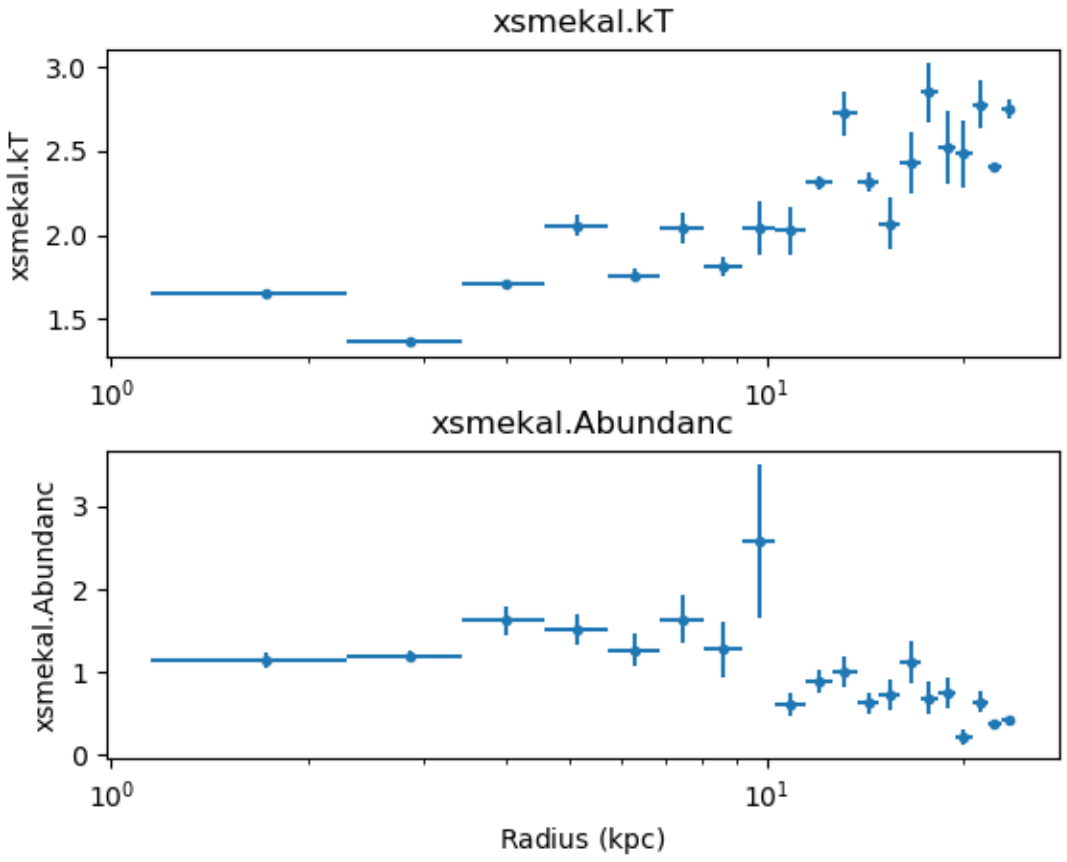
(continued from previous page)

```
    ...      ...      ... ...                      ...                      ...
     13   206.64   221.4 ...  -0.0005679816551559976  0.0005430748019680798
     14    221.4   236.16 ... -0.0004808355652631546 0.0004641288097302735 7
     15   236.16   250.92 ...  -0.0004951659664768401 0.0004759949079704093 4
     16   250.92   265.68 ...  -0.0004031089363366082  0.0003915259159180066
     17   265.68   280.44 ...  -0.0003647519140328147 0.0003554845940160998 6
     18   280.44    295.2 ...                     nan                      nan
     19    295.2   309.96 ... -0.0001964851171975395 8 0.0001948255458952309 6
Length = 20 rows
```

Note: With this set of data, the covariance method failed to calculate errors for the parameters of the last-but one shell, which is indicated by the presence of *NaN* values in the error columns.

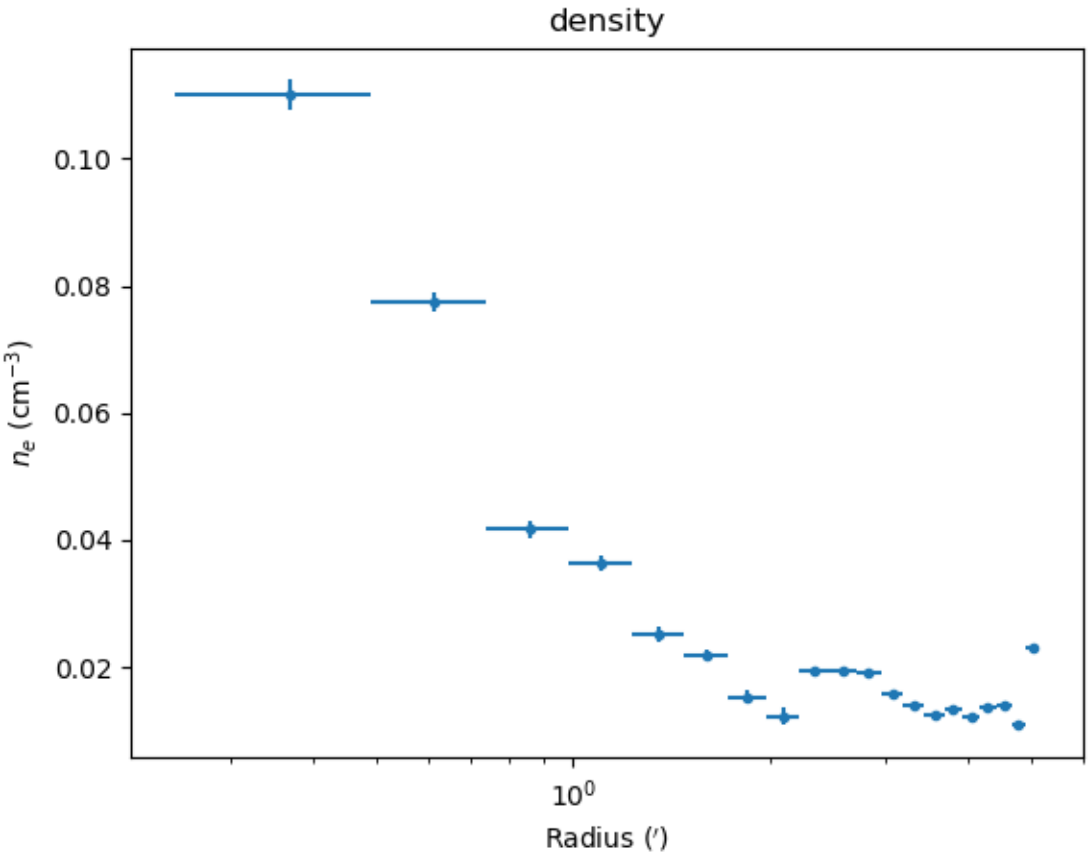
The fit or error results can be plotted as a function of radius with the *fit_plot()*, *conf_plot()*, and *covar_plot()* methods (the latter two include error bars). For example, the following plot mixes these plots with Matplotlib commands to compare the temperature and abundance profiles:

```
>>> plt.subplot(2, 1, 1)
>>> dep.covar_plot('xsmekal.kt', clearwindow=False)
>>> plt.xlabel('')
>>> plt.subplot(2, 1, 2)
>>> dep.covar_plot('xsmekal.abundanc', clearwindow=False)
>>> plt.subplots_adjust(hspace=0.3)
```



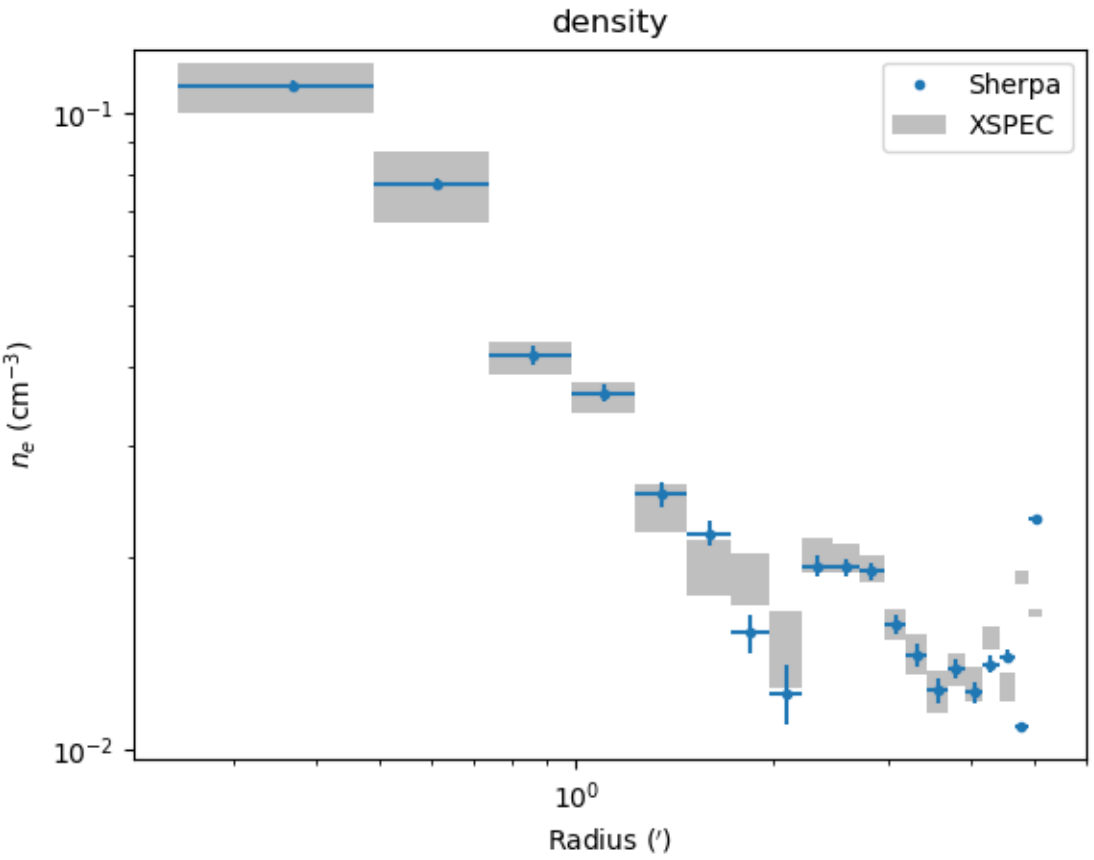
The derived density profile, along with errors, can also be displayed (the X axis is displayed using angular units rather than as a length):

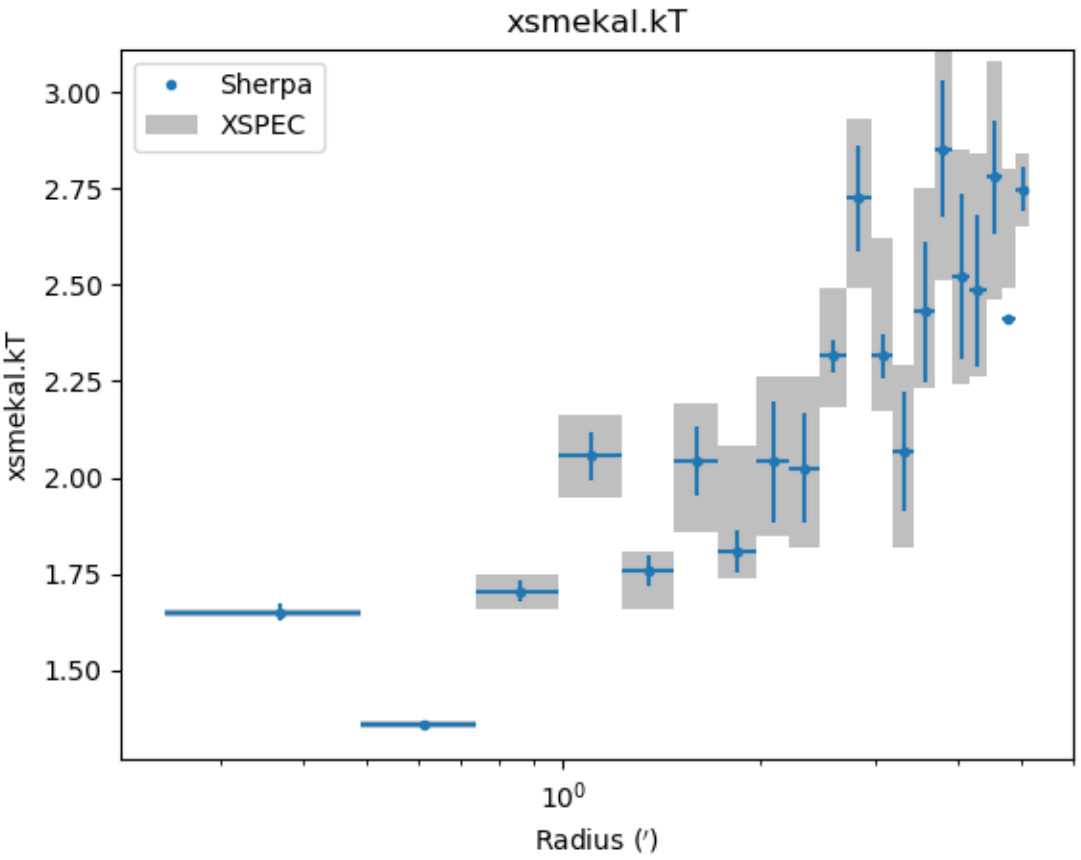
```
>>> dep.covar_plot('density', units='arcmin')
```



5.10 Comparing results

In the images below the `deproject` results (blue) are compared with values (gray boxes) from an independent onion-peeling analysis by P. Nulsen using a custom perl script to generate XSPEC model definition and fit commands. These plots were created with the `plot_m87.py` script in the `examples` directory. The agreement is good (note that the version of XSPEC used for the two cases does not match):


density
Sherpa
XSPEC
10^{-1}
10^{-2}
n_e (cm^{-3})
10^0
Radius (′)

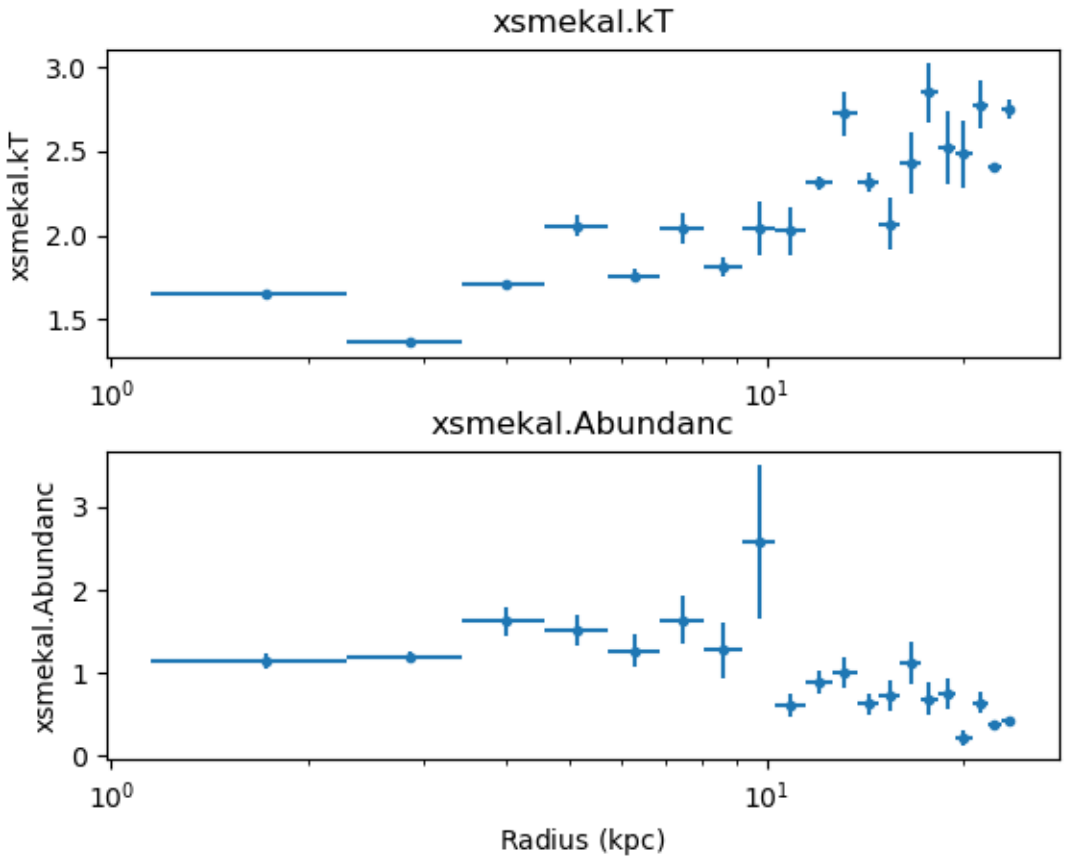

xsmekal.kT
Sherpa
XSPEC
3.00
2.75
2.50
2.25
2.00
1.75
1.50
xsmekal.kT
10^0
Radius (′)

CHAPTER 6

Combining shells

We now look at tie-ing parameters from different annuli (that is, forcing the temperature or abundance of one shell to be the same as a neighbouring shell), which is a technique used to try and reduce the "ringing" that can be seen in deprojected data (e.g. Russell, Sanders, & Fabian 2008 and seen in the *temperature profile of the M87 data*). Sherpa supports complicated links between model parameters with the link and unlink functions, and the Deproject class provides helper methods - `tie_par()` and `untie_par()` - that allows you to easily tie together annuli.

Starting from where the *M87* example left off, we have:



The *get_shells()* method lists the current set of annuli, and show that there are currently no grouped (or tied-together) annuli:

```
>>> dep.get_shells()
[{'annuli': [19], 'dataids': [19]},
 {'annuli': [18], 'dataids': [18]},
 {'annuli': [17], 'dataids': [17]},
 {'annuli': [16], 'dataids': [16]},
 {'annuli': [15], 'dataids': [15]},
 {'annuli': [14], 'dataids': [14]},
 {'annuli': [13], 'dataids': [13]},
 {'annuli': [12], 'dataids': [12]},
 {'annuli': [11], 'dataids': [11]},
 {'annuli': [10], 'dataids': [10]},
 {'annuli': [9], 'dataids': [9]},
 {'annuli': [8], 'dataids': [8]},
 {'annuli': [7], 'dataids': [7]},
 {'annuli': [6], 'dataids': [6]},
 {'annuli': [5], 'dataids': [5]},
 {'annuli': [4], 'dataids': [4]},
 {'annuli': [3], 'dataids': [3]},
 {'annuli': [2], 'dataids': [2]},
 {'annuli': [1], 'dataids': [1]},
 {'annuli': [0], 'dataids': [0]}]
```

For this example we are going to see what happens when we tie together the temperature and abundance of annulus

17 and 18 (the inner-most annulus is numbered 0, the outer-most is 19 in this example).

```
>>> dep.tie_par('xsmekal.kt', 17, 18)
Tying xsmekal_18.kT to xsmekal_17.kT
>>> dep.tie_par('xsmekal.abundanc', 17, 18)
Tying xsmekal_18.Abundanc to xsmekal_17.Abundanc
```

Looking at the first few elements of the list returned by *get_shells* shows us that the two annuli have now been grouped together. This means that when a fit is run then these two annuli will be fit simultaneously.

```
>>> dep.get_shells()[0:5]
[{'annuli': [19], 'dataids': [19]},
 {'annuli': [17, 18], 'dataids': [17, 18]},
 {'annuli': [16], 'dataids': [16]},
 {'annuli': [15], 'dataids': [15]},
 {'annuli': [14], 'dataids': [14]}]
```

Since the model values for the 18th annulus have been changed, the data needs to be re-fit. The screen output is slightly different, as highlighted below, to reflect this new grouping:

```
>>> dep.fit()
Note: annuli have been tied together
Fitting annulus 19  dataset: 19
Dataset               = 19
...
Freezing xswabs_19
Freezing xsmekal_19
Fitting annuli [17, 18]  datasets: [17, 18]
Datasets              = 17, 18
Method                = levmar
Statistic             = chi2xspecvar
Initial fit statistic = 469.803
Final fit statistic   = 447.963 at function evaluation 16
Data points           = 439
Degrees of freedom    = 435
Probability [Q-value] = 0.323566
Reduced statistic     = 1.0298
Change in statistic   = 21.8398
   xsmekal_17.kT   2.64505      +/- 0.0986876
   xsmekal_17.Abundanc   0.520943     +/- 0.068338
   xsmekal_17.norm   0.0960039    +/- 0.00370363
   xsmekal_18.norm   0.0516096    +/- 0.00232468
Freezing xswabs_17
Freezing xsmekal_17
Freezing xswabs_18
Freezing xsmekal_18
Fitting annulus 16  dataset: 16
...
```

The table returned by *fit()* and *get_fit_results()* still contains 20 rows, since each annulus has its own row:

```
>>> tied = dep.get_fit_results()
>>> len(tied)
20
```

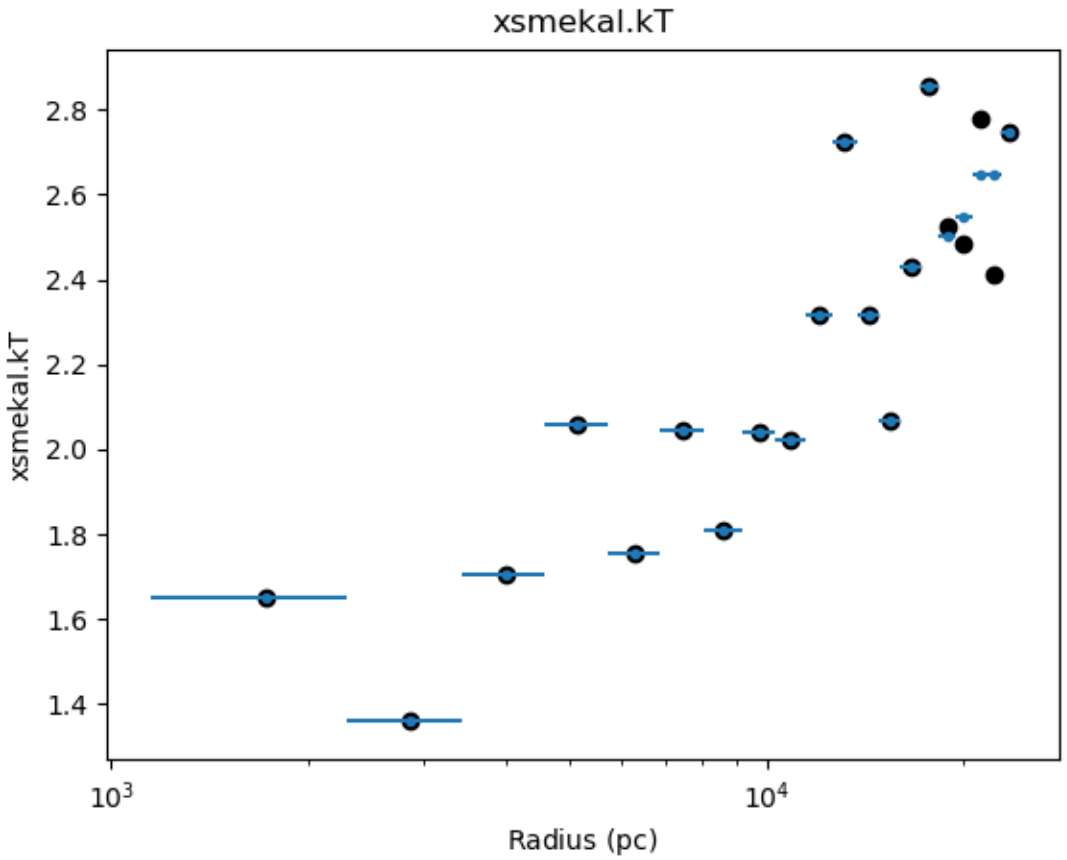
Looking at the outermost annuli, we can see that the temperature and abundance values are the same:

```
>>> print(tied['xsmekal.kT'][16:])
    xsmekal.kT
------------------
2.5470334673573936
 2.645045455995055
 2.645045455995055
2.7480050952727346
>>> print(tied['xsmekal.Abundanc'][16:])
  xsmekal.Abundanc
-------------------
0.24424232427701525
 0.5209430839965413
 0.5209430839965413
 0.4297577517591726
```

Note: The field names in the table are case sensitive - that is, you have to say `xsmekal.kT` and not `xsmekal.kt` - unlike the Sherpa parameter interface.

The results can be compared to the original, such as in the following plot, which shows that the temperature profile hasn't changed significantly in the core, but some of the oscillations at large radii have been reduced (the black circles show the temperature values from the original fit):

```
>>> dep.fit_plot('xsmekal.kt', units='pc')
>>> rmid = (onion['rlo_phys'] + onion['rhi_phys']) / 2
>>> plt.scatter(rmid.to(u.pc), onion['xsmekal.kT', c='k')
```



Note: As the X-axis of the plot created by *fit_plot()* is in pc, the `rmid` value is expliclty converted to these units before plotting.

The `results` parameter of `fit_plot` could have been set to the variable `onion` to display the results of the previous fit, but there is no easy way to change the color of the points.

More shells could be tied together to try and further reduce the oscillations, or errors calculated, with either *conf()* or *covar()*.

The parameter ties can be removed with *untie_par()*:

```
>>> dep.untie_par('xsmekal.kt', 18)
Untying xsmekal_18.kT
>>> dep.untie_par('xsmekal.abundanc', 18)
Untying xsmekal_18.Abundanc
>>> dep.get_shells()[0:4]
[{'annuli': [19], 'dataids': [19]},
 {'annuli': [18], 'dataids': [18]},
 {'annuli': [17], 'dataids': [17]},
 {'annuli': [16], 'dataids': [16]}]
```

CHAPTER 7

Multiple datasets per annulus (3C186)

A second example illustrates the use of `deproject` for a multi-obsid observation of 3C186. It also shows how to set a background model for fitting with the `cstat` statistic. The extracted spectral data for this example are not yet publicly available, but were used in Siemiginowska et al. 2010:

The script starts with some setup:

```
>>> import deproject

>>> radii = ('2.5', '6', '17')
>>> dep = deproject.Deproject(radii=[float(x) for x in radii])

>>> set_method("levmar")
>>> set_stat("cstat")
```

Now we read in the data as before with `dep.load_pha()`. The only difference to the *M87 example* is the additional loop over the obsids. The `dep.load_pha()` function automatically extracts the obsid (the identifier used to discriminate Chandra observations) from the file header. This is used later in the case of setting a background model.

```
>>> obsids = (9407, 9774, 9775, 9408)
>>> for ann in range(len(radii)  -1):
...     for obsid in obsids:
...         dep.load_pha('3c186/%d/ellipse%s-%s.pi' % (obsid, radii[ann],␣
↪radii[ann+1]), annulus=ann)
```

Create and configure the source model expression as usual:

```
>>> dep.set_source('xsphabs*xsapec')
>>> dep.ignore(None, 0.5)
>>> dep.ignore(7, None)
>>> dep.freeze("xsphabs.nh")

>>> dep.set_par('xsapec.redshift', 1.06)
>>> dep.set_par('xsphabs.nh', 0.0564)
```

Set the background model (here we use the IPython `%run` magic command to evaluate the commands in the file `acis-s-bkg.py`):

```
>>> %run acis-s-bkg.py
>>> acis_s_bkg = get_bkg_source()
>>> dep.set_bkg_model(acis_s_bkg)
```

Fit the projection model:

```
>>> dep.fit()
```

CHAPTER 8

The deproject.deproject module

Deproject 2-d circular annular spectra to 3-d object properties.

This module implements the "onion-skin" approach popular in X-ray analysis of galaxy clusters and groups to estimate the three-dimensional temperature, metallicity, and density distributions of an optically-thin plasma from the observed (projected) two-dimensional data, arranged in concentric circular annuli.

Author Tom Aldcroft (taldcroft@cfa.harvard.edu), Douglas Burke (dburke@cfa.harvard.edu)

Classes

Deproject(radii[, theta, angdist, cosmology])	Support deprojecting a set of spectra (2-d concentric circular annuli).

8.1 Deproject

class deproject.deproject.**Deproject**(*radii*, *theta=<Quantity 360. deg>*, *angdist=None*, *cosmology=None*)

Bases: *deproject.specstack.SpecStack*

Support deprojecting a set of spectra (2-d concentric circular annuli).

Parameters

- **radii** (*AstroPy Quantity representing an angle on the sky*) – The edges of each annulus, which must be circular, concentric, in ascending order, and >= 0. If there are n annuli then there are n+1 radii, since the start and end of the sequence must be given. The units are expected to be arcsec, arcminute, or degree.
- **theta** (*AstroPy Quantity (scalar or array) representing an angle*) – The "fill factor" of each annulus, given by the azimuthal coverage of the shell in degrees. The value can be a scalar, so the same value is used for all annuli, or a sequence

with a length matching the number of annuli. Since the annulus assumes circular symmetry there is no need to define the starting point of the measurement, for cases when the value is less than 360 degrees.

- **angdist** (*None or AstroPy.Quantity, optional*) – The angular-diameter distance to the source. If not given then it is calculated using the source redshift along with the *cosmology* attribute.
- **cosmology** (*None or astropy.cosmology object, optional*) – The cosmology used to convert redshift to an angular-diameter distance. This is used when *angdist* is None. If *cosmology* is None then the *astropy.cosmology.Planck15* Cosmology object is used.

Examples

The following highly-simplified example fits a deprojected model to data from three annuli - ann1.pi, ann2.pi, and ann3.pi - and also calculates errors on the parameters using the confidence method:

```
>>> dep = Deproject([0, 10, 40, 100] * u.arcsec)
>>> dep.load_pha('ann1.pi', 0)
>>> dep.load_pha('ann2.pi', 1)
>>> dep.load_pha('ann3.pi', 2)
>>> dep.subtract()
>>> dep.notice(0.5, 7.0)
>>> dep.set_source('xsphabs * xsapec')
>>> dep.set_par('xsapec.redshift', 0.23)
>>> dep.thaw('xsapec.abundanc')
>>> dep.set_par('xsphabs.nh', 0.087)
>>> dep.freeze('xsphabs.nh')
>>> dep.fit()
>>> dep.fit_plot('rstat')
>>> errs = dep.conf()
>>> dep.conf_plot('density')
```

Attributes Summary

angdist	Angular size distance (an AstroPy quantity)
cosmology	Return the cosmology object (only used if angdist not set)
datasets	
n_datasets	How many datasets are registered?
redshift	Source redshift

Methods Summary

conf()	Estimate errors using confidence, using the "onion-peeling" method.
conf_plot(field[, results, units, xlog, ...])	Plot up the confidence errors as a function of radius.
covar()	Estimate errors using covariance, using the "onion-peeling" method.
covar_plot(field[, results, units, xlog, ...])	Plot up the covariance errors as a function of radius.

Continued on next page

Table 3 – continued from previous page

density_plot([units, xlog, ylog, overplot, ...])	Plot up the electron density as a function of radius.
dummyfunc(*args, **kwargs)	
find_norm(shell)	Return the normalization value for the given shell.
find_parval(parname)	Return the value of the first parameter matching the name.
fit()	Fit the data using the "onion-peeling" method.
fit_plot(field[, results, units, xlog, ...])	Plot up the fit results as a function of radius.
freeze(par)	Freeze the given parameter in each shell.
get_conf_results()	What are the conf results, per annulus?
get_covar_results()	What are the covar results, per annulus?
get_density()	Calculate the electron density for each shell.
get_fit_results()	What are the fit results, per annulus?
get_par(par)	Return the parameter value for each shell.
get_radii([units])	What are the radii of the shells?
get_shells()	How are the annuli grouped?
group(*args)	Apply the grouping for each data set.
guess()	Guess the starting point by fitting the projected data.
ignore(*args)	Apply Sherpa ignore command to each dataset.
load_pha(specfile, annulus)	Load a pha file and add to the datasets for stacked analysis.
notice(*args)	Apply Sherpa notice command to each dataset.
par_plot(par[, units, xlog, ylog, overplot, ...])	Plot up the parameter as a function of radius.
plot_arf(*args, **kwargs)	
plot_bkg(*args, **kwargs)	
plot_bkg_chisqr(*args, **kwargs)	
plot_bkg_delchi(*args, **kwargs)	
plot_bkg_fit(*args, **kwargs)	
plot_bkg_fit_delchi(*args, **kwargs)	
plot_bkg_fit_resid(*args, **kwargs)	
plot_bkg_model(*args, **kwargs)	
plot_bkg_ratio(*args, **kwargs)	
plot_bkg_resid(*args, **kwargs)	
plot_bkg_source(*args, **kwargs)	
plot_bkg_unconvolved(*args, **kwargs)	
plot_chisqr(*args, **kwargs)	
plot_data(*args, **kwargs)	
plot_delchi(*args, **kwargs)	
plot_fit(*args, **kwargs)	
plot_fit_delchi(*args, **kwargs)	
plot_fit_resid(*args, **kwargs)	
plot_model(*args, **kwargs)	
plot_order(*args, **kwargs)	
plot_psf(*args, **kwargs)	
plot_ratio(*args, **kwargs)	
plot_resid(*args, **kwargs)	
plot_source(*args, **kwargs)	
print_window(*args, **kwargs)	Create a hardcopy version of each plot window.
set_bkg_model(bkgmodel)	Create a background model for each annulus.
set_par(par, val)	Set the parameter value in each shell.
set_source([srcmodel])	Create a source model for each annulus.

Continued on next page

Table 3 – continued from previous page

subtract(*args)	Subtract the background from each dataset.
thaw(par)	Thaw the given parameter in each shell.
tie_par(par, base, *others)	Tie parameters in one or more shells to the base shell.
ungroup(*args)	Turn off the grouping for each data set.
unsubtract(*args)	Un-subtract the background from each dataset.
untie_par(par, *others)	Remove the parameter tie/link in the shell.

Attributes Documentation

angdist
Angular size distance (an AstroPy quantity)

cosmology
Return the cosmology object (only used if angdist not set)

datasets = None

n_datasets
How many datasets are registered?

This is not the same as the number of annuli.

Returns **ndata** – The number of datasets.

Return type int

redshift
Source redshift

Methods Documentation

conf()
Estimate errors using confidence, using the "onion-peeling" method.

It is assumed that `fit` has been called. The results can also be retrieved with `get_conf_results`.

Returns **errors** – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the sigma and percent values, and parameter results (accessed using <model name>.<par name>, <model name>.<par name>_lo, and <model name>.<par name>_hi syntax, where the match is case sensitive).

Return type astropy.table.Table instance

See also:

covar(), *fit()*, *get_conf_results()*, *conf_plot()*

Examples

Run a fit and then error analysis, then plot up the abundance against temperature values including the error bars. Note that the Matplotlib *errorbar* routine requires "positive" error values whereas the <param>_lo values are negative, hence they are negated in the creation of `dkt` and `dabund`:

```
>>> dep.fit()
>>> errs = dep.conf()
>>> kt, abund = errs['xsapec.kT'], errs['xsapec.Abundanc']
```

(continues on next page)

(continued from previous page)

```
>>> ktlo, kthi = errs['xsapec.kT_lo'], errs['xsapec.kT_hi']
>>> ablo, abhi = errs['xsapec.Abundanc_lo'], errs['xsapec.Abundanc_hi']
>>> dkt = np.vstack((-ktlo, kthi))
>>> dabund = np.vstack((-ablo, abhi))
>>> plt.clf()
>>> plt.errorbar(kt, abund, xerr=dkt, yerr=dabund, fmt='.')
```

Plot up the temperature distibution as a function of radius, including the error bars calculated by the conf routine:

```
>>> dep.fit()
>>> dep.conf()
>>> dep.conf_plot('xsmekal.kt')
```

conf_plot(*field*, *results=None*, *units='kpc'*, *xlog=True*, *ylog=False*, *overplot=False*, *clearwindow=True*)

Plot up the confidence errors as a function of radius.

This method can be used to plot up the last conf results or a previously-stored set. Any error bars are shown at the scale they were calculated (as given by the `sigma` and `percent` columns of the results).

Parameters

- **field** (*str*) – The column to plot from the fit results (the match is case insensitive).
- **results** (*None or astropy.table.Table instance*) – The return value from the `conf` or `get_conf_results` methods.
- **units** (*str or astropy.units.Unit, optional*) – The X-axis units (a length or angle, such as 'Mpc' or 'arcsec', where the case is important).
- **xlog** (*bool, optional*) – Should the x axis be drawn with a log scale (default True)?
- **ylog** (*bool, optional*) – Should the y axis be drawn with a log scale (default False)?
- **overplot** (*bool, optional*) – Clear the plot or add to existing plot?
- **clearwindow** (*bool, optional*) – How does this interact with overplot?

See also:

fit(), *get_conf_results()*, *fit_plot()*, *covar_plot()*, *density_plot()*, *par_plot()*

Notes

Error bars are included on the dependent axis if the results contain columns that match the requested field with suffixes of '_lo' and '_hi'. These error bars are asymmetric, which is different to `covar_plot`.

If a limit is missing (i.e. it is a NaN) then no error bar is drawn. This can make it look like the error is very small.

Examples

Plot the temperature as a function of radius from the last fit, including error bars:

```
>>> dep.conf_plot('xsapec.kt')
```

Plot the density with the radii labelled in arcminutes and the density shown on a log scale:

```
>>> dep.conf_plot('density', units='arcmin', ylog=True)
```

Overplot the current conf results on those from a previous fit, where `conf1` was returned from the `conf` or `get_conf_results` methods:

```
>>> dep.conf_plot('xsapec.abundanc', results=conf1)
>>> dep.conf_plot('xsapec.abundanc', overplot=True)
```

covar()

Estimate errors using covariance, using the "onion-peeling" method.

It is assumed that `fit` has been called. The results can also be retrieved with `get_covar_results`.

Returns **errors** – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the sigma and percent values, and parameter results (accessed using <model name>.<par name>, <model name>.<par name>_lo, and <model name>.<par name>_hi syntax, where the match is case sensitive). The _lo and _hi values are symmetric for covar, that is the _lo value will be the negative of the _hi value.

Return type astropy.table.Table instance

See also:

conf(), *fit()*, *get_covar_results()*, *covar_plot()*

Examples

Run a fit and then error analysis, then plot up the abundance against temperature values including the error bars. Since the covariance routine returns symmetric error bars, the <param>_hi values are used in the plot:

```
>>> dep.fit()
>>> errs = dep.covar()
>>> kt, abund = errs['xsapec.kT'], errs['xsapec.Abundanc']
>>> dkt = errs['xsapec.kT_hi']
>>> dabund = errs['xsapec.Abundanc_hi']
>>> plt.clf()
>>> plt.errorbar(kt, abund, xerr=dkt, yerr=dabund, fmt='.')
```

Plot up the temperature distibution as a function of radius, including the error bars calculated by the covar routine:

```
>>> dep.fit()
>>> dep.covar()
>>> dep.covar_plot('xsmekal.kt')
```

covar_plot(*field*, *results=None*, *units='kpc'*, *xlog=True*, *ylog=False*, *overplot=False*, *clearwindow=True*)

Plot up the covariance errors as a function of radius.

This method can be used to plot up the last covar results or a previously-stored set. Any error bars are shown at the scale they were calculated (as given by the `sigma` and `percent` columns of the results).

Parameters

- **field** (*str*) – The column to plot from the fit results (the match is case insensitive).

- **results** (*None or astropy.table.Table instance*) – The return value from the `covar` or `get_covar_results` methods.
- **units** (*str or astropy.units.Unit, optional*) – The X-axis units (a length or angle, such as 'Mpc' or 'arcsec', where the case is important).
- **xlog** (*bool, optional*) – Should the x axis be drawn with a log scale (default True)?
- **ylog** (*bool, optional*) – Should the y axis be drawn with a log scale (default False)?
- **overplot** (*bool, optional*) – Clear the plot or add to existing plot?
- **clearwindow** (*bool, optional*) – How does this interact with overplot?

See also:

fit(), *get_covar_results()*, *fit_plot()*, *conf_plot()*, *density_plot()*, *par_plot()*

Notes

Error bars are included on the dependent axis if the results contain columns that match the requested field with the suffix '_hi'. The error bars are therefore symmetric, which is different to `conf_plot`.

If a limit is missing (i.e. it is a NaN) then no error bar is drawn. This can make it look like the error is very small.

Examples

Plot the temperature as a function of radius from the last fit, including error bars:

```
>>> dep.covar_plot('xsapec.kt')
```

Plot the density with the radii labelled in arcminutes and the density shown on a log scale:

```
>>> dep.covar_plot('density', units='arcmin', ylog=True)
```

Overplot the current covar results on those from a previous fit, where `covar1` was returned from the `covar` or `get_covar_results` methods:

```
>>> dep.covar_plot('xsapec.abundanc', results=covar1)
>>> dep.covar_plot('xsapec.abundanc', overplot=True)
```

density_plot(*units='kpc'*, *xlog=True*, *ylog=True*, *overplot=False*, *clearwindow=True*)

Plot up the electron density as a function of radius.

The density is displayed with units of cm^-3. This plots up the density calculated using the current normalization parameter values. The `fit_plot`, `conf_plot`, and `covar_plot` routines display the fit and error results for these parameters.

Parameters

- **units** (*str or astropy.units.Unit, optional*) – The X-axis units (a length or angle, such as 'Mpc' or 'arcsec', where the case is important).
- **xlog** (*bool, optional*) – Should the x axis be drawn with a log scale (default True)?
- **ylog** (*bool, optional*) – Should the y axis be drawn with a log scale (default False)?
- **overplot** (*bool, optional*) – Clear the plot or add to existing plot?

- **clearwindow** (*bool, optional*) – How does this interact with overplot?

See also:

conf_plot(), *covar_plot()*, *fit_plot()*, *par_plot()*

Examples

Plot the density as a function of radius.

```
>>> dep.density_plot()
```

Label the radii with units of arcminutes:

```
>>> dep.density_plot(units='arcmin')
```

dummyfunc(**args*, ***kwargs*)

find_norm(*shell*)

Return the normalization value for the given shell.

This is limited to XSPEC-style models, where the parameter is called "norm".

Parameters **shell** (*int*) – The shell number.

Returns **norm** – The normalization of the shell.

Return type float

Raises ValueError – If there is not one *norm* parameter for the shell.

See also:

find_parval(), *set_par()*

find_parval(*parname*)

Return the value of the first parameter matching the name.

Parameters **parname** (*str*) – The parameter name. The case is ignored in the match, and the first match is returned.

Returns **parval** – The parameter value

Return type float

Raises ValueError – There is no match for the parameter.

See also:

find_norm(), *set_par()*

Examples

```
>>> kt = dep.find_parval('kt')
```

fit()

Fit the data using the "onion-peeling" method.

Unlike the normal Sherpa fit, this does not fit all the data simultaneously, but instead fits the outermost annulus first, then freezes its parameters and fits the annulus inside it, repeating this until all annuli have

been fit. At the end of the fit all the parameters that were frozen are freed. The results can also be retrieved with `get_fit_results`.

Returns fits – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the final fit statistic and change in fit statistic (*statval* and *dstatval*), the reduced statistic and q value (as *rstat* and *qval*) if appropriate, and the thawed parameter values (accessed using <model name>.<par name> syntax, where the match is case sensitive).

Return type astropy.table.Table instance

See also:

conf(), *covar()*, *get_fit_results()*, *guess()*, *fit_plot()*

Notes

If there are any tied parameters between annuli then these annuli are combined together (fit simultaneously). The results from the fits to each annulus can be retrieved after `fit` has been called with the `get_fit_results` method.

The results have been separated out per annulus, even if several annuli were combined in a fit due to tied parameters, and there is no information in the returned structure to note this.

Examples

Fit the annuli using the onion-peeling approach, and then plot up the reduced statistic for each dataset:

```
>>> res = dep.fit()
>>> plt.clf()
>>> rmid = 0.5 * (res['rlo_phys'] + res['rhi_phys'])
>>> plt.plot(rmid, res['rstat'])
```

Plot the temperature-abundance values per shell, color-coded by annulus:

```
>>> plt.clf()
>>> plt.plot(res['xsapec.kT'], res['xsapec.Abundanc'],
...          c=res['annulus'])
>>> plt.colorbar()
>>> plt.xlabel('kT')
>>> plt.ylabel('Abundance')
```

Plot up the temperature distibution as a function of radius from the fit:

```
>>> dep.fit()
>>> dep.fit_plot('xsmekal.kt')
```

fit_plot(*field*, *results=None*, *units='kpc'*, *xlog=True*, *ylog=False*, *overplot=False*, *clearwindow=True*)

Plot up the fit results as a function of radius.

This method can be used to plot up the last fit results or a previously-stored set. To include error bars on the dependent values use the *conf_plot* or *covar_plot* methods.

Parameters

- **field** (*str*) – The column to plot from the fit results (the match is case insensitive).

- **results** (*None or astropy.table.Table instance*) – The return value from the `fit` or `get_fit_results` methods.
- **units** (*str or astropy.units.Unit, optional*) – The X-axis units (a length or angle, such as 'Mpc' or 'arcsec', where the case is important).
- **xlog** (*bool, optional*) – Should the x axis be drawn with a log scale (default True)?
- **ylog** (*bool, optional*) – Should the y axis be drawn with a log scale (default False)?
- **overplot** (*bool, optional*) – Clear the plot or add to existing plot?
- **clearwindow** (*bool, optional*) – How does this interact with overplot?

See also:

fit(), *get_fit_results()*, *conf_plot()*, *covar_plot()*, *density_plot()*, *par_plot()*

Examples

Plot the temperature as a function of radius from the last fit:

```
>>> dep.fit_plot('xsapec.kt')
```

Plot the reduced fit statistic from the last fit:

```
>>> dep.fit_plot('rstat')
```

Plot the density with the radii labelled in arcminutes and the density shown on a log scale:

```
>>> dep.fit_plot('density', units='arcmin', ylog=True)
```

Overplot the current fit results on those from a previous fit, where `fit1` was returned from the `fit` or `get_fit_results` methods:

```
>>> dep.fit_plot('xsapec.abundanc', results=fit1)
>>> dep.fit_plot('xsapec.abundanc', overplot=True)
```

freeze(*par*)

Freeze the given parameter in each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

See also:

thaw(), *tie_par()*, *untie_par()*

Examples

```
>>> dep.freeze('clus.abundanc')
```

get_conf_results()

What are the conf results, per annulus?

This returns the fit result for each annulus from the last time that the `conf` method was called. It *does not* check to see if anything has changed since the last `conf` call (e.g. parameters being tied together or untied, or a manual fit to a shell). Note that `get_shells` should be used to find out if the shells were grouped together (although this can be reconstructed from the *datasets* field of each *ErrorEstResults* instance).

Returns **errors** – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the sigma and percent values, and parameter results (accessed using <model name>.<par name>, <model name>.<par name>_lo, and <model name>.<par name>_hi syntax, where the match is case sensitive).

Return type astropy.table.Table instance

See also:

fit(), *get_covar_results()*, *get_fit_results()*, *get_radii()*, *get_shells()*, *conf_plot()*

get_covar_results()

What are the covar results, per annulus?

This returns the fit result for each annulus from the last time that the `covar` method was called. It *does not* check to see if anything has changed since the last `covar` call (e.g. parameters being tied together or untied, or a manual fit to a shell). Note that `get_shells` should be used to find out if the shells were grouped together.

Returns **errors** – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the sigma and percent values, and parameter results (accessed using <model name>.<par name>, <model name>.<par name>_lo, and <model name>.<par name>_hi syntax, where the match is case sensitive).

Return type astropy.table.Table instance

See also:

fit(), *get_conf_results()*, *get_fit_results()*, *get_radii()*, *get_shells()*, *covar_plot()*

get_density()

Calculate the electron density for each shell.

Convert the model normalzations (assumed to match the standard definition for XSPEC thermal-plasma models) for each shell.

Returns **dens** – The densities calculated for each shell, in units of cm^-3.

Return type astropy.units.quantity.Quantity instance

See also:

find_norm()

Notes

The electron density is taken to be:

```
n_e^2 = norm * 4*pi * DA^2 * 1e14 * (1+z)^2 / volume * ne_nh_ratio
```

where:

```
norm        = model normalization from sherpa fit
DA          = angular size distance (cm)
volume      = volume (cm^3)
ne_nh_ratio = 1.18
```

The model components for each volume element (the intersection of the annular cylinder `a` with the spherical shell `s`) are multiplied by a volume normalization:

```
vol_norm[s,a] = volume[s,a] / v_sphere
v_sphere = volume of sphere enclosing outer annulus
```

With this convention the `volume` used in calculating the electron density is simply `v_sphere`.

get_fit_results()

What are the fit results, per annulus?

This returns the fit result for each annulus from the last time that the `fit` method was called. It *does not* check to see if anything has changed since the last `fit` call (e.g. parameters being tied together or untied, or a manual fit to a shell). Note that `get_shells` should be used to find out if the shells were grouped together.

Returns **fits** – This records per-annulus data, such as the inner and outer radius (*rlo_ang*, *rhi_ang*, *rlo_phys*, *rhi_phys*), the final fit statistic and change in fit statistic (*statval* and *dstatval*), the reduced statistic and q value (as *rstat* and *qval*) if appropriate, and the thawed parameter values (accessed using <model name>.<par name> syntax, where the match is case sensitive).

Return type astropy.table.Table instance

See also:

fit(), *get_conf_results()*, *get_covar_results()*, *get_radii()*, *get_shells()*, *fit_plot()*

get_par(*par*)

Return the parameter value for each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

Returns **vals** – The parameter values, in shell order.

Return type ndarray

See also:

find_parval(), *find_norm()*, *set_par()*

Examples

```
>>> kts = dep.get_par('xsapec.kt')
```

get_radii(*units='arcsec'*)

What are the radii of the shells?

Return the inner and outer edge of each annulus, in the given units. Physical units (e.g. 'kpc') can only be used if a redshift or angular-diameter distance has been set. This does not apply the grouping that *get_shells* does.

Parameters **units** (*str or astropy.units.Unit, optional*) – The name of the units to use for the returned radii. They must be an angle - such as 'arcsec' - or a length - such as 'kpc' or 'Mpc' (case is important).

See also:

get_shells()

Returns **rlo, rhi** – The inner and outer radius for each annulus.

Return type astropy.units.Quantity, astropy.units.Quantity

get_shells()

How are the annuli grouped?

An annulus may have multiple data sets associated with it, but it may also be linked to other annuli due to tied parameters. The return value is per group, in the ordering needed for the outside-to-inside onion skin fit, where the keys for the dictionary are 'annuli' and 'dataids'.

Returns **groups** – Each dictionary has the keys 'annuli' and 'dataids', and lists the annuli and data identifiers that are fit together. The ordering matches that of the onion-skin approach, so the outermost group first.

Return type list of dicts

See also:

get_radii(), *tie_par()*

Examples

For a 3-annulus deprojection where there are no parameter ties to combine annului:

```
>>> dep.get_shells()
[{'annuli': [2], 'dataids': [2]},
 {'annuli': [1], 'dataids': [1]},
 {'annuli': [0], 'dataids': [0]}]
```

After tie-ing the abundance parameter for the outer two shells, there are now two groups of annuli:

```
>>> dep.tie_par('xsapec.abundanc', 1, 2)
Tying xsapec_2.Abundanc to xsapec_1.Abundanc
>>> dep.get_shells()
[{'annuli': [1, 2], 'dataids': [1, 2]},
 {'annuli': [0], 'dataids': [0]}]
```

group(**args*)

Apply the grouping for each data set.

See also:

ungroup()

Examples

```
>>> dep.group()
```

guess()

Guess the starting point by fitting the projected data.

Use a fitting scheme - based on the suggestion in the XSPEC projct documention - to estimate the starting position of the fit (the initial fit parameters). This can be useful since it can reduce the time taken to fit the deprojected data and help avoid the deprojection from getting stuck in a local minimum.

See also:

fit()

Notes

Each annulus, from outer to inner, is fit individually, ignoring the contribution from any outer annulus. After the fit, the model normalisation is corrected for the volume-filling factor of the annulus. If there are any tied parameters between annuli then these annuli are combined together (fit simultaneously).

Unlike the Sherpa guess function, this does *not* change the limits of any parameter.

Possible improvements include:

- re-normalize each spectrum before fitting.
- transfer the model parameters of the inner-most shell in a group to the next set of shells to fit.

ignore (**args*)

Apply Sherpa ignore command to each dataset.

The filter is applied to each data set separately.

See also:

notice()

Examples

Restrict the analysis to those bins which fall in the range 0.5 to 7.0 keV, where the limits are not included in the noticed range. The call to *notice* is used to clear any existing filter.

```
>>> dep.notice(None, None)
>>> dep.ignore(None, 0.5)
>>> dep.ignore(7.0, None)
```

load_pha (*specfile*, *annulus*)

Load a pha file and add to the datasets for stacked analysis.

It is required that datasets for all annuli are loaded before the source model is created (to ensure that components are created for each annulus).

Parameters

- **specfile** (*str or sherpa.astro.data.DataPHA object*) – If a string, the name of the file containing the source spectrum, which must be in PHA format (the data is expected to be extracted on the PI column). If a DataPHA object, then this is used (and is assumed to contain any needed background data).
- **annulus** (*int*) – The annulus number for the data.

Returns **dataid** – The Sherpa dataset identifier used for this spectrum.

Return type int

Examples

Load the data for four annuli from the files 'ann1.pi' to 'ann4.pi'.

```
>>> dep.load_pha('ann1.pi', 0)
>>> dep.load_pha('ann2.pi', 1)
>>> dep.load_pha('ann3.pi', 2)
>>> dep.load_pha('ann4.pi', 3)
```

Load in the PHA files into Sherpa DataPHA objects, and then use these objects:

```
>>> s1 = ui.unpack_pha('src1.pi')
>>> s2 = ui.unpack_pha('src2.pi')
>>> s3 = ui.unpack_pha('src3.pi')
>>> dep.load_pha(s1, 0)
>>> dep.load_pha(s2, 1)
>>> dep.load_pha(s3, 2)
```

notice (**args*)

Apply Sherpa notice command to each dataset.

The filter is applied to each data set separately.

See also:

ignore()

Examples

Restrict the analysis to those bins which fall in the range 0.5 to 7.0 keV, where the limits are included in the noticed range. The first call to *notice* is used to clear any existing filter.

```
>>> dep.notice(None, None)
>>> dep.notice(0.5, 7.0)
```

par_plot (*par*, *units='kpc'*, *xlog=True*, *ylog=False*, *overplot=False*, *clearwindow=True*)

Plot up the parameter as a function of radius.

This plots up the current parameter values. The `fit_plot`, `conf_plot`, and `covar_plot` routines display the fit and error results for these parameters.

Parameters

- **par** (*str*) – The parameter name, specified as <model_type>.<par_name>.
- **units** (*str or astropy.units.Unit, optional*) – The X-axis units (a length or angle, such as 'Mpc' or 'arcsec', where the case is important).
- **xlog** (*bool, optional*) – Should the x axis be drawn with a log scale (default True)?
- **ylog** (*bool, optional*) – Should the y axis be drawn with a log scale (default False)?
- **overplot** (*bool, optional*) – Clear the plot or add to existing plot?
- **clearwindow** (*bool, optional*) – How does this interact with overplot?

See also:

conf_plot(), *covar_plot()*, *density_plot()*, *fit_plot()*

Examples

Plot the temperature as a function of radius.

```
>>> dep.par_plot('xsapec.kt')
```

Label the radii with units of arcminutes for the abundanc parameter of the xsapec model:

```
>>> dep.par_plot('xsapec.abundanc', units='arcmin')
```

plot_arf(**args*, ***kwargs*)

plot_bkg(**args*, ***kwargs*)

plot_bkg_chisqr(**args*, ***kwargs*)

plot_bkg_delchi(**args*, ***kwargs*)

plot_bkg_fit(**args*, ***kwargs*)

plot_bkg_fit_delchi(**args*, ***kwargs*)

plot_bkg_fit_resid(**args*, ***kwargs*)

plot_bkg_model(**args*, ***kwargs*)

plot_bkg_ratio(**args*, ***kwargs*)

plot_bkg_resid(**args*, ***kwargs*)

plot_bkg_source(**args*, ***kwargs*)

plot_bkg_unconvolved(**args*, ***kwargs*)

plot_chisqr(**args*, ***kwargs*)

plot_data(**args*, ***kwargs*)

plot_delchi(**args*, ***kwargs*)

plot_fit(**args*, ***kwargs*)

plot_fit_delchi(**args*, ***kwargs*)

plot_fit_resid(**args*, ***kwargs*)

plot_model(**args*, ***kwargs*)

plot_order(**args*, ***kwargs*)

plot_psf(**args*, ***kwargs*)

plot_ratio(**args*, ***kwargs*)

plot_resid(**args*, ***kwargs*)

plot_source(**args*, ***kwargs*)

print_window(**args*, ***kwargs*)

Create a hardcopy version of each plot window.

Parameters

- **args** – The arguments to be sent to the "create a hardcopy" routine (print_window for ChIPS and savefig for Matplotlib). The first argument, if given, is assumed to be the file name and so will have the shell number added to it.
- **kwargs** – Keyword arguments for the call.

Notes

This is not guaranteed to work properly for Matplotlib.

Examples

Create hardcopy versions of the data plots, called "data0", "data1", ...

```
>>> dep.plot_data()
>>> dep.print_window('data')
```

set_bkg_model (*bkgmodel*)
Create a background model for each annulus.

The background model is the same between the annuli, except that a scaling factor is added for each annulus (to allow for normalization uncertainities). The scaling factors are labelled 'bkg_norm_<obsid>', and at least one of these must be frozen (otherwise it is likely to be degenerate with the background normalization, causing difficulties for the optimiser).

Parameters **bkgmodel** (*model instance*) – The background model expression applied to each annulus. Unlike set_source this should be the actual model instance, and not a string.

See also:

set_source(), *set_par()*

Examples

Model the background with a single power-law component:

```
>>> dep.set_bkg_model(xspowerlaw.bpl)
```

set_par (*par*, *val*)
Set the parameter value in each shell.

Parameters

- **par** (*str*) – The parameter name, specified as <model_type>.<par_name>
- **val** (*float*) – The parameter value.

See also:

get_par(), *tie_par()*

Examples

```
>>> dep.set_par('xsapec.abundanc', 0.25)
```

set_source (*srcmodel='xsphabs*xsapec'*)
Create a source model for each annulus.

Unlike the standard *set_source* command, this version just uses the <model name>, not <model name>.<username>, since the <username> is automatically created for users by appending the annulus number to <model name>.

Parameters **srcmodel** (*str, optional*) – The source model expression applied to each annulus.

See also:

set_bkg_model(), *set_par()*

Notes

The data must have been read in for all the data before calling this method (this matches Sherpa, where you can not call set_source unless you have already loaded the data to fit).

Examples

The following two calls have the same result: model instances called 'xsphabs<annulus>' and 'xsapec<annulus>' are created for each annulus, and the source expression for the annulus set to their multiplication:

```
>>> dep.set_source()
>>> dep.set_source('xsphabs * xsapec')
```

Use the XSPEC vapec model rather than the apec model to represent the plasma emission:

```
>>> dep.set_source('xsphabs * xsvapec')
```

subtract (**args*)

Subtract the background from each dataset.

See also:

unsubtract()

Examples

```
>>> dep.substract()
```

thaw (*par*)

Thaw the given parameter in each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

See also:

freeze(), *tie_par()*, *untie_par()*

Examples

```
>>> dep.thaw('clus.abundanc')
```

tie_par (*par*, *base*, **others*)

Tie parameters in one or more shells to the base shell.

This is a limited form of the Sherpa ability to link parameters, since it sets the parameter in the other shells to the same value as the parameter in the base shell. More complex situations will require direct calls to *sherpa.astro.ui.link*.

Parameters

- **par** (*str*) – The parameter specifier, as <model_type>.<par_name>.
- **base** (*int*) – The base shell number.
- ***others** (*scalar*) – The shell, or shells to link to the base shell.

See also:

set_par(), *untie_par()*

Examples

Tie the temperature and abundance parameters in shell 9 to that in shell 8, so that any fits will set the shell 9 values to those used in shell 8 (so reducing the number of free parameters in the fit).

```
>>> dep.tie_par('xsapec.kT', 8, 9)
Tying xsapec_9.kT to xsapec_8.kT
>>> dep.tie_par('xsapec.abundanc', 8, 9)
Tying xsapec_9.Abundanc to xsapec_8.Abundanc
```

Tie three annuli together:

```
>>> dep.tie_par('xsapec.kT', 12, 13, 14)
Tying xsapec_13.kT to xsapec_12.kT
Tying xsapec_14.kT to xsapec_12.kT
```

ungroup(**args*)

Turn off the grouping for each data set.

See also:

group()

Examples

```
>>> dep.ungroup()
```

unsubtract(**args*)

Un-subtract the background from each dataset.

This can be useful when you want to compare the results to the "wstat" stat (a Poisson-based stat which includes the background data as a component and provides a goodness-of-fit estimate).

See also:

subtract()

Examples

```
>>> dep.unsubstract()
```

untie_par(*par*, **others*)

Remove the parameter tie/link in the shell.

This is intended to remove links between shells created by *tie_par*, but will remove any links created by *sherpa.astro.ui.link*.

Parameters

- **par** (*str*) – The parameter specifier, as <model_type>.<par_name>.
- ***others** (*scalar*) – The shell, or shells to un-tie/unlink.

See also:

tie_par()

Notes

It is safe to call on a parameter that is not tied or linked to another parameter.

Examples

Untie the abundance parameter in shell 9; that is, it is now free to vary independently in a fit.

```
>>> dep.untie_par('xsapec.abundanc', 9)
Untying xsapec_9.Abundanc
```

Untie multiple annuli:

```
>>> dep.untie_par('xsmekal.kt', 13, 14)
Untying xsmekal_13.kT
Untying xsmekal_14.kT
```

Functions

deproject_from_xflt(pat, rscale[, rinner, ...])	Set up the projection object from XFLT keywords in the PHA files.

8.2 deproject_from_xflt

deproject.deproject.**deproject_from_xflt**(*pat*, *rscale*, *rinner=0*, *angdist=None*, *cosmology=None*)

Set up the projection object from XFLT keywords in the PHA files.

When using the XSPEC projct model, values are read from XFLT keywords (as used by the XSPEC deprojection code[1]) rather than being specified manually. This function creates a Deproject object and loads in a set of PHA files matching a pattern, using the XFLT keywords to set radii and theta values. The annuli *must* be circular.

Parameters

- **pat** (*str*) – The pattern representing the files to read in. If the stk module, provided by CIAO, is available then CIAO stack syntax[2] can be used. The order of the files does not matter, but it is currently assumed that there is only one file per annulus.
- **rscale** (*AstroPy quantity*) – The scaling factor used to convert the XFLT radii (XFLT001 and XFLT002 keywords) to an angle. If the values are in arcseconds then `rscale` would be set to `1 * u.arcsec`.
- **rinner** (*float, optional*) – The inner radius of the central annulus, in the same system as the XFLT0001 and XFLT002 keyword values (this is a unitless value).
- **angdist** (*None or AstroPy.Quantity, optional*) – The angular-diameter

[1] https://asd.gsfc.nasa.gov/XSPECwiki/projct_model
[2] http://cxc.harvard.edu/ciao/ahelp/stack.html

distance to the source. If not given then it is calculated using the source redshift along with the *cosmology* attribute.

- **cosmology** (*None or astropy.cosmology object, optional*) – The cosmology used to convert redshift to an angular-diameter distance. This is used when *angdist* is None. If *cosmology* is None then the *astropy.cosmology.Planck15* Cosmology object is used.

Returns **dep** – The deproject instance with the files loaded and associated with the correct annuli.

Return type Deproject instance

Notes

This currently is *not* guaranteed to support multiple data sets in the same annulus. There is no check that the annuli are touching and do not overlap.

References

Examples

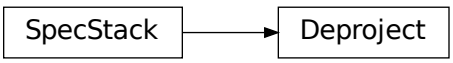
Create a Deproject instance from the files matching the pattern "src*.pi", whose XFLT radii are in ACIS pixels:

```
>>> dep = deproject_from_xflt('src*.pi', 0.492 * u.arcsec)
```

When used in CIAO, the stack syntax can be used to specify the files, so if the file clus.stk contains the file names, one per line, then the following will read them in and create a Deproject instance. In this case the XFLT radii are in arcminutes:

```
>>> dep = deproject_from_xflt('@clus.stk', 1 * u.arcmin)
```

8.3 Class Inheritance Diagram



CHAPTER 9

The deproject.specstack module

Manipulate a stack of spectra in Sherpa.

The methods in the SpecStack class provide a way to automatically apply familiar Sherpa commands such as set_par or freeze or plot_fit to a stack of PHA spectra. This simplifies simultaneous fitting of multiple spectra.

Note that the `specstack` module is currently distributed within with the `deproject` package. *Specstack* is not yet fully documented or tested outside the context of *deproject*.

Author Tom Aldcroft (taldcroft@cfa.harvard.edu), Douglas Burke (dburke@cfa.harvard.edu)

Classes

SpecStack()	Manipulate a stack of spectra in Sherpa.

9.1 SpecStack

class deproject.specstack.**SpecStack**

Bases: `object`

Manipulate a stack of spectra in Sherpa.

This *SpecStack* class provides a number of wrappers around Sherpa routines that handle loading data, setting the source model, setting up the data to fit, such as: the noticed energy range, how to handle the background, extracting parameter values, and plotting data.

Attributes Summary

datasets	Information (dataset identifier, annulus, name) about the loaded data.
n_datasets	How many datasets are registered?

Methods Summary

dummyfunc(*args, **kwargs)	
find_norm(shell)	Return the normalization value for the given shell.
find_parval(parname)	Return the value of the first parameter matching the name.
freeze(par)	Freeze the given parameter in each shell.
get_par(par)	Return the parameter value for each shell.
group(*args)	Apply the grouping for each data set.
ignore(*args)	Apply Sherpa ignore command to each dataset.
load_pha(specfile, annulus)	Load a pha file and add to the datasets for stacked analysis.
notice(*args)	Apply Sherpa notice command to each dataset.
plot_arf(*args, **kwargs)	
plot_bkg(*args, **kwargs)	
plot_bkg_chisqr(*args, **kwargs)	
plot_bkg_delchi(*args, **kwargs)	
plot_bkg_fit(*args, **kwargs)	
plot_bkg_fit_delchi(*args, **kwargs)	
plot_bkg_fit_resid(*args, **kwargs)	
plot_bkg_model(*args, **kwargs)	
plot_bkg_ratio(*args, **kwargs)	
plot_bkg_resid(*args, **kwargs)	
plot_bkg_source(*args, **kwargs)	
plot_bkg_unconvolved(*args, **kwargs)	
plot_chisqr(*args, **kwargs)	
plot_data(*args, **kwargs)	
plot_delchi(*args, **kwargs)	
plot_fit(*args, **kwargs)	
plot_fit_delchi(*args, **kwargs)	
plot_fit_resid(*args, **kwargs)	
plot_model(*args, **kwargs)	
plot_order(*args, **kwargs)	
plot_psf(*args, **kwargs)	
plot_ratio(*args, **kwargs)	
plot_resid(*args, **kwargs)	
plot_source(*args, **kwargs)	
print_window(*args, **kwargs)	Create a hardcopy version of each plot window.
set_par(par, val)	Set the parameter value in each shell.
subtract(*args)	Subtract the background from each dataset.
thaw(par)	Thaw the given parameter in each shell.
tie_par(par, base, *others)	Tie parameters in one or more shells to the base shell.
ungroup(*args)	Turn off the grouping for each data set.
unsubtract(*args)	Un-subtract the background from each dataset.
untie_par(par, *others)	Remove the parameter tie/link in the shell.

Attributes Documentation

datasets = None
Information (dataset identifier, annulus, name) about the loaded data.

n_datasets
How many datasets are registered?

This is not the same as the number of annuli.

Returns **ndata** – The number of datasets.

Return type int

Methods Documentation

dummyfunc(**args*, ***kwargs*)

find_norm(*shell*)
Return the normalization value for the given shell.

This is limited to XSPEC-style models, where the parameter is called "norm".

Parameters **shell** (*int*) – The shell number.

Returns **norm** – The normalization of the shell.

Return type float

Raises ValueError – If there is not one *norm* parameter for the shell.

See also:

find_parval(), *set_par()*

find_parval(*parname*)
Return the value of the first parameter matching the name.

Parameters **parname** (*str*) – The parameter name. The case is ignored in the match, and the first match is returned.

Returns **parval** – The parameter value

Return type float

Raises ValueError – There is no match for the parameter.

See also:

find_norm(), *set_par()*

Examples

```
>>> kt = dep.find_parval('kt')
```

freeze(*par*)
Freeze the given parameter in each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

See also:

thaw(), *tie_par()*, *untie_par()*

Examples

```
>>> dep.freeze('clus.abundanc')
```

get_par(*par*)
Return the parameter value for each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

Returns **vals** – The parameter values, in shell order.

Return type ndarray

See also:

find_parval(), *find_norm()*, *set_par()*

Examples

```
>>> kts = dep.get_par('xsapec.kt')
```

group(**args*)
Apply the grouping for each data set.

See also:

ungroup()

Examples

```
>>> dep.group()
```

ignore(**args*)
Apply Sherpa ignore command to each dataset.

The filter is applied to each data set separately.

See also:

notice()

Examples

Restrict the analysis to those bins which fall in the range 0.5 to 7.0 keV, where the limits are not included in the noticed range. The call to *notice* is used to clear any existing filter.

```
>>> dep.notice(None, None)
>>> dep.ignore(None, 0.5)
>>> dep.ignore(7.0, None)
```

load_pha(*specfile*, *annulus*)
Load a pha file and add to the datasets for stacked analysis.

It is required that datasets for all annuli are loaded before the source model is created (to ensure that components are created for each annulus).

Parameters

- **specfile** (*str or sherpa.astro.data.DataPHA object*) – If a string, the name of the file containing the source spectrum, which must be in PHA format (the data is expected to be extracted on the PI column). If a DataPHA object, then this is used (and is assumed to contain any needed background data).
- **annulus** (*int*) – The annulus number for the data.

Returns **dataid** – The Sherpa dataset identifier used for this spectrum.

Return type int

Examples

Load the data for four annuli from the files 'ann1.pi' to 'ann4.pi'.

```
>>> dep.load_pha('ann1.pi', 0)
>>> dep.load_pha('ann2.pi', 1)
>>> dep.load_pha('ann3.pi', 2)
>>> dep.load_pha('ann4.pi', 3)
```

Load in the PHA files into Sherpa DataPHA objects, and then use these objects:

```
>>> s1 = ui.unpack_pha('src1.pi')
>>> s2 = ui.unpack_pha('src2.pi')
>>> s3 = ui.unpack_pha('src3.pi')
>>> dep.load_pha(s1, 0)
>>> dep.load_pha(s2, 1)
>>> dep.load_pha(s3, 2)
```

notice(**args*)

Apply Sherpa notice command to each dataset.

The filter is applied to each data set separately.

See also:

ignore()

Examples

Restrict the analysis to those bins which fall in the range 0.5 to 7.0 keV, where the limits are included in the noticed range. The first call to *notice* is used to clear any existing filter.

```
>>> dep.notice(None, None)
>>> dep.notice(0.5, 7.0)
```

plot_arf(**args*, ***kwargs*)

plot_bkg(**args*, ***kwargs*)

plot_bkg_chisqr(**args*, ***kwargs*)

plot_bkg_delchi(**args*, ***kwargs*)

plot_bkg_fit(**args*, ***kwargs*)

plot_bkg_fit_delchi(**args*, ***kwargs*)

plot_bkg_fit_resid(**args*, ***kwargs*)

plot_bkg_model(**args*, ***kwargs*)

plot_bkg_ratio(**args*, ***kwargs*)

plot_bkg_resid(**args*, ***kwargs*)

plot_bkg_source(**args*, ***kwargs*)

plot_bkg_unconvolved(**args*, ***kwargs*)

plot_chisqr(**args*, ***kwargs*)

plot_data(**args*, ***kwargs*)

plot_delchi(**args*, ***kwargs*)

plot_fit(**args*, ***kwargs*)

plot_fit_delchi(**args*, ***kwargs*)

plot_fit_resid(**args*, ***kwargs*)

plot_model(**args*, ***kwargs*)

plot_order(**args*, ***kwargs*)

plot_psf(**args*, ***kwargs*)

plot_ratio(**args*, ***kwargs*)

plot_resid(**args*, ***kwargs*)

plot_source(**args*, ***kwargs*)

print_window(**args*, ***kwargs*)

Create a hardcopy version of each plot window.

Parameters

- **args** – The arguments to be sent to the "create a hardcopy" routine (print_window for ChIPS and savefig for Matplotlib). The first argument, if given, is assumed to be the file name and so will have the shell number added to it.
- **kwargs** – Keyword arguments for the call.

Notes

This is not guaranteed to work properly for Matplotlib.

Examples

Create hardcopy versions of the data plots, called "data0", "data1", ...

```
>>> dep.plot_data()
>>> dep.print_window('data')
```

set_par(*par*, *val*)

Set the parameter value in each shell.

Parameters

- **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

- **val** (*float*) – The parameter value.

See also:

get_par(), *tie_par()*

Examples

```
>>> dep.set_par('xsapec.abundanc', 0.25)
```

subtract (**args*)
Subtract the background from each dataset.

See also:

unsubtract()

Examples

```
>>> dep.substract()
```

thaw (*par*)
Thaw the given parameter in each shell.

Parameters **par** (*str*) – The parameter name, specified as <model_type>.<par_name>

See also:

freeze(), *tie_par()*, *untie_par()*

Examples

```
>>> dep.thaw('clus.abundanc')
```

tie_par (*par*, *base*, **others*)
Tie parameters in one or more shells to the base shell.

This is a limited form of the Sherpa ability to link parameters, since it sets the parameter in the other shells to the same value as the parameter in the base shell. More complex situations will require direct calls to *sherpa.astro.ui.link*.

Parameters

- **par** (*str*) – The parameter specifier, as <model_type>.<par_name>.
- **base** (*int*) – The base shell number.
- ***others** (*scalar*) – The shell, or shells to link to the base shell.

See also:

set_par(), *untie_par()*

Examples

Tie the temperature and abundance parameters in shell 9 to that in shell 8, so that any fits will set the shell 9 values to those used in shell 8 (so reducing the number of free parameters in the fit).

```
>>> dep.tie_par('xsapec.kt', 8, 9)
Tying xsapec_9.kT to xsapec_8.kT
>>> dep.tie_par('xsapec.abundanc', 8, 9)
Tying xsapec_9.Abundanc to xsapec_8.Abundanc
```

Tie three annuli together:

```
>>> dep.tie_par('xsapec.kt', 12, 13, 14)
Tying xsapec_13.kT to xsapec_12.kT
Tying xsapec_14.kT to xsapec_12.kT
```

ungroup(**args*)

Turn off the grouping for each data set.

See also:

group()

Examples

```
>>> dep.ungroup()
```

unsubtract(**args*)

Un-subtract the background from each dataset.

This can be useful when you want to compare the results to the "wstat" stat (a Poisson-based stat which includes the background data as a component and provides a goodness-of-fit estimate).

See also:

subtract()

Examples

```
>>> dep.unsubstract()
```

untie_par(*par*, **others*)

Remove the parameter tie/link in the shell.

This is intended to remove links between shells created by *tie_par*, but will remove any links created by *sherpa.astro.ui.link*.

Parameters

- **par** (*str*) – The parameter specifier, as <model_type>.<par_name>.
- ***others** (*scalar*) – The shell, or shells to un-tie/unlink.

See also:

tie_par()

Notes

It is safe to call on a parameter that is not tied or linked to another parameter.

Examples

Untie the abundance parameter in shell 9; that is, it is now free to vary independently in a fit.

```
>>> dep.untie_par('xsapec.abundanc', 9)
Untying xsapec_9.Abundanc
```

Untie multiple annuli:

```
>>> dep.untie_par('xsmekal.kt', 13, 14)
Untying xsmekal_13.kT
Untying xsmekal_14.kT
```

Python Module Index

d

Index

R

S

T

U