PyXspec Documentation

Release 2.1.3

Feb 15, 2024

CONTENTS

PyXspec is an object oriented Python interface to the XSPEC spectral-fitting program. It provides an alternative to Tcl, the sole scripting language for standard Xspec usage. With PyXspec loaded, a user can run Xspec with Python language scripts or interactively at a Python shell prompt.

Note: The source code distribution of XSPEC is required for using PyXspec

---

CHAPTER

ONE

BUILD AND INSTALL

1.1 Build and Install PyXspec

1.1.1 Requirements

Since we do not distribute Python with the HEASOFT packages, you'll need to have it already installed on your system (which it is with most Linux and Mac OSX distributions). PyXspec is compatible with Python **3.x**. The **python** executable must be in your path, and with the library and header files located in the standard directories relative to the executable (see the **Troubleshooting** section for more info).

1.1.2 Building/Installing

PyXspec is fully integrated into the general HEASOFT build procedure, as described at http://heasarc.gsfc.nasa.gov/lheasoft/install.html . So it will be **built and installed automatically** with the rest of XSPEC/HEASOFT, requiring no additional effort from the user.

By default, the HEASOFT configure/build process will first search for **python3** versions on your system, and only if it finds none will it then search for **python**. You can override this behavior by setting your PYTHON environment variable to the full path to your desired python executable *before* running the HEASOFT configure. You can also switch between python distributions *after* your original build. Please see *Changing Python Versions After Original Build* for details.

Once HEASOFT is finished building and installing, you should find PyXspec's code files and *lib_pyXspec.so* library in the directory *$HEADAS/lib/python/xspec*.

When you run the HEASOFT initialization script (*$HEADAS/headas-init.csh* or *.sh*), it will add *$HEADAS/lib/python* to your PYTHONPATH environment. This allows Python find the PyXspec module so that you may load it into your session from anywhere, using the *import xspec* statement.

1.1.3 Changing Python Versions After Original Build

If you wish to change the Python version from which PyXspec was built during the original HEASOFT configure/build, you can do so by manually editing the file: *heasoft-<ver>/Xspec/BUILD_DIR/hmakerc*. This file contains definitions for the 2 variables: PYTHON_INC and PYTHON_LIB, which set the compiler paths for the Python header file inclusion and dynamic library linkage, and these must be changed to point to your new Python location.

For example, to point to an Anaconda Python3 distribution on your system, your new settings might look something like:

```
PYTHON_INC="-I/path/to/your/anaconda3/include/python3.8m"
PYTHON_LIB="-L/path/to/your/anaconda3/lib -lpython3.8m"
```

Note: You can determine the Python path information with the *python-config* (or *python3-config*) utility. Call it with the *--cflags* option for the include path information, and with *--ldflags* for the library path information.

After making these edits, simply rebuild PyXspec by doing:

```
cd /path/to/heasoft-<ver>/Xspec/src/XSUser/Python/xspec
hmake clean
hmake
hmake install

cd ../mxspec
hmake clean
hmake
hmake install
```

1.1.4 Running on Mac OS

These issues apply only to Mac OS users. Linux users may skip this section.

As seen on the HEASOFT builds page for Mac OS https://heasarc.gsfc.nasa.gov/lheasoft/macos.html, one can build HEASOFT using XCode compilers (clang,clang++) for C/C++, with a 3rd party compiler (MacPorts, Homebrew) for Fortran. Or one can use the 3rd party package for all three C/C++/Fortran. We recommend building HEASOFT (and therefore PyXspec) using a Python distribution that matches your C/C++ compilers, ie. XCode in the first case and a Python from MacPorts or Homebrew in the second case.

It is also possible to succesfully build and run PyXspec from an Anaconda python distribution.

1.1.5 Troubleshooting

If the HEASOFT configuration stage fails when it's processing PyXspec, it will just issue a warning and continue. Its failure should not affect the rest of the XSPEC and HEASOFT build. Standard XSPEC will still be fully functional, but its Python interface won't be available.

The most likely cause of a PyXspec build failure is that the HEASOFT configuration script can't find a **python** executable and/or its accompanying library and header files. You should first check that the command "*which python*" can find an executable on your path (*python* is generally a symbolic link to a version-specific executable).

If you don't explicitly set the PYTHON environment variable, the configuration script first looks for **python3** versions followed by **python**. Once it's found an executable, it looks for the files *Python.h* and *libpython[m.n].so* (or .dylib) with the help of the *python-config* (or *python3-config*) utility. The configuration fails if it is unable to find either file.

If you are manually trying to reset the PYTHON_INC and PYTHON_LIB variables (in *heasoft-<ver>/Xspec/BUILD_DIR/hmakerc*) to point to these files in a different Python distribution (for instance if switching between Python 2 and 3, or to a 3rd-party Python distribution installed on your system), we recommend using the *python[3]-config* utility to help you determine the correct path information.

Other Errors

If you are running on **Mac OS X** and get a Python "ImportError" message containing such statements as **no suitable image found** and **mach-o, but wrong architecture**, it's likely you are trying to run a 32-bit mode Python while your default HEASoft build is now 64-bit.

If you are running on a **Mac** and have built your HEASOFT installation with **all 3 compilers (gcc, g++, gfortran)** coming from **Homebrew** or **MacPorts**, AND you get a runtime error that begins with something like:

```
python(86419) malloc: *** error for object 0x574b160: pointer being
                          freed was not allocated
```

then it likely means there's a conflict between your default Python distribution and the compiler libraries used to build PyXspec. Please see the **Changing Python Versions After Original Build** section above for switching away from the default Xcode Python distribution.

CHAPTER

TWO

TUTORIALS

2.1 Quick Tutorial

This assumes the user already has a basic familiarity with both XSPEC and Python. Everything in PyXspec is accessible by importing the package **xspec** into your Python script.

PyXspec can be utilized in a Python script or from the command line of the plain interactive Python interpreter. PyXspec does not implement its own command handler, so it is NOT intended to be run as the Python equivalent of a traditional interactive XSPEC session (which is really an *enhanced* interactive Tcl interpreter). In other words you launch an interactive PyXspec session with:

```
LINUX> python
>>> import xspec
>>>
```

rather than:

```
LINUX> xspec
XSPEC12>
```

Note that in all the tutorial examples the **xspec** package name qualifier is left off. You must either include the **xspec** qualifier:

```
s = xspec.Spectrum("file1.pha")
```

or use a variation of the Python `import` or `from...import` commands:

```
from xspec import *
s = Spectrum("file1.pha")
```

2.1.1 Jumping In: The *Really* Quick Tutorial

A simple Xspec load-fit-plot Python script may look something like this:

```
#!/usr/bin/python
from xspec import *

Spectrum("file1.pha")
Model("phabs*pow")
Fit.perform()
```

(continues on next page)

(continued from previous page)

```
Plot.device = "/xs"
Plot("data")
```

Keeping this template in mind, we'll proceed to fill in the details...

2.1.2 Terminology

This description uses the standard Python object-oriented terminology, distinguishing between **classes** and *objects*. **Class** is used when referring to the *type* or definition of an *object*. An *object* refers to a specific instance of a **class** and is normally assigned to a variable. For example a user may load 3 data files by creating 3 spectral data objects *s1*, *s2*, and *s3*, which are all instances of the class *Spectrum*.

The functions and stored data members that make up the definition of a **class** are referred to as **methods** and **attributes** respectively.

The term **Standard XSPEC** refers to the traditional ways of using XSPEC, either with a Tcl script or an interactive XSPEC session.

2.1.3 Getting Help

Documentation for PyXspec classes can be found in the *Classes* section of this manual. You can also access the same information from the Python shell command line using Python's built-in `help(<class-name>)` function.

2.1.4 The 6 Global Objects

An XSPEC session fundamentally consists of loading data, fitting that data to a model, and plotting the results. To manage these operations, PyXspec offers the user 6 global objects: *AllChains*, *AllData*, *AllModels*, *Fit*, *Plot*, and *Xset*. Note that these are NOT the names of classes. They are instantiated *objects* of the **class** types shown below.

Note: Operations involving these 6 global ojbects should ALWAYS be performed through the object names and NEVER their class names. Their class names should never appear in your code.

Object Name	Class	Role
AllChains	*ChainManager*	Monte Carlo Markov *Chain* container
AllData	*DataManager*	Container for all loaded data sets (objects of class *Spectrum*)
AllModels	*ModelManager*	Container for all *Model* objects
Fit	*FitManager*	Manager class for setting properties and running a fit
Plot	*PlotManager*	Manager class for performing XSPEC plots
Xset	*XspecSettings*	Storage class for XSPEC settings

PyXspec instantiates these objects immediately upon the importing of the **xspec** package. You cannot create any other objects of these class types, as they each allow only 1 instance of their type. (They are **singletons** in the language of design patterns.)

2.1.5 Loading And Removing Data

Spectral data files can be loaded in several ways. You can create an object of the *Spectrum* class by passing it the data file name:

```
s1 = Spectrum("file1.pha")
```

which also adds the new object *s1* to the *AllData* container. Or you can simply add the new file directly to the container without retrieving a *Spectrum* object:

```
AllData += "file1.pha"
```

Later you can always obtain a *Spectrum* object reference to any of the loaded spectra by passing *AllData* an integer:

```
s2 = AllData(2)  # s2 is a reference to the 2nd loaded spectrum
```

For more complicated data loading, you have access to the same functionality in Standard XSPEC's **data** command. Simply pass a string to the *AllData* object's __call__ method:

```
AllData("file1 file2 2:3 file3")
```

Note that only the last example allows you to assign multiple data groups, the 3rd spectrum being assigned to data group 2. Also note that in the last example any previously loaded data sets are removed, thus reproducing the behavior of Standard XSPEC's **data** command.

Other ways of removing Spectrum objects (ie. data sets) from the container:

```
AllData -= 3    # Removes the 3rd Spectrum object (the spectrum with
                    # index number 3) from the container.
AllData -= s1   # Removes the Spectrum object s1.
AllData -= "*"  # Removes all Spectrum objects.
AllData.clear() # Removes all Spectrum objects.
```

You can check the current state of the *AllData* container at any time by doing:

```
AllData.show()
```

Similarly, to view information about a single *Spectrum* object:

```
s2.show()
```

2.1.6 Defining Models

The basic way of defining an XSPEC model is to create an object of the PyXspec class *Model*. Simply pass in a string containing a combination of 1 or more XSPEC model **components**. Since this uses the same syntax as Standard XSPEC's **model** command, component abbreviations are allowed:

```
m1 = Model("phabs*po + ga")
```

and to see a complete listing of available XSPEC model components, do:

```
Model.showList()
```

When you define a model like this, PyXspec also automatically adds the new object to the global *AllModels* container. If the model is applied to multiple data groups, object copies are added to the container for each data group.

Similar to the case of spectral data, you can also load models directly into the global container:

```
# Another way to define a new model and create an object for each data group.
AllModels += "phabs*po + ga"
# Retrieve the model object assigned to data group 2.
m2 = AllModels(2)
# Various ways to remove all model objects from the container.
AllModels.clear()
AllModels -= "*"
```

To display models and their parameters:

```
# This displays all parameters in all model objects:
AllModels.show()
# While this displays just parameters 1,2,3 and 5:
AllModels.show("1-3, 5")
# This displays a single model object:
m2.show()
```

For defining multiple (or named) models and assigning multiple sources, please see the *Extended Tutorial* section.

2.1.7 Component and Parameter Objects

Model objects contain *Component* objects and Component objects contain *Parameter* objects. There are several ways to access and set components and parameters individually (and if you want to change many parameter values at once, it may be faster to use the Model or *AllModels* **setPars** methods described in the next section). Examples of individual Component and Parameter object access:

```
# Component objects are accessible-by-name as Model object attributes:
comp1 = m1.phabs
comp2 = m1.powerlaw
# Parameter objects are accessible-by-name as Component object attributes:
par4 = m1.gaussian.LineE
# ...and we can modify their values:
par4.values = 3.895
m1.phabs.nH = 5.0
comp2.PhoIndex = 1.5

# Can also get a Parameter object directly from a Model, without going
# through a Component.  Just pass the Model the Parameter index number:
par5 = m1(5)

# Examples of numerical operations allowed with Parameter objects:
par4 += 0.75
par4 *= 2.0
y1 = m1.phabs.nH*100.0
y2 = par4 + par5
```

Note: For models with duplicate copies of components, see the *Extended Tutorial* for accessing Component objects by name.

Note that in the above examples, only the parameter's *value* is being accessed or modified. To change all or part of its FULL list of settings including auxiliary values: *value, fit delta, min, bot, top, max*, you can set its **values** attribute to a tuple or list of size 1-6:

```
par4.values = 4.3, .01, 1e-3
par4.values = [4.3, .01, 1e-3, 1e-2, 100, 200]
```

Or for greater flexibility you can set it to a string using Standard XSPEC's **newpar** command syntax:

```
# This allows you to set new values non-consecutively.
par4.values = "1.0, -.01,,,,150"
```

A quick way to freeze or thaw a parameter is to toggle its **frozen** attribute:

```
par4.frozen = False
par5.frozen = True
```

To link a parameter to one or more others, set its **link** attribute to a link expression string as you would have with the **newpar** command. Or if you simply want to link to a single parameter with no numerical operations, you can link it directly to that Parameter object. To remove the link, set **link** to an empty string or call the parameter's **untie** method:

```
par5.link = "2.3 * 4" # Link par 5 to par 4 with a multiplicative constant.
par5.link = m.phabs.nH # Link par 5 to the specified par object.
par5.link = "" # Removes the link.
par5.untie()   # Also removes the link.
```

Also ALL linked parameters in a model object can be untied with a single call to the *Model* class **untie** method.

To display a parameter's full set of values (including auxiliary values), just print its **values** attribute:

```
>>> print(par4.values)
[6.5, 0.05, 0.0, 0.0, 1000000.0, 1000000.0]
```

2.1.8 Setting Multiple Parameters At A Time

You can set multiple parameter values with a single call using the *Model* or *AllModels* (see *ModelManager*) **setPars** methods. This may be considerably faster than setting parameters one at a time through the individual *Parameter* objects as shown in the previous section. With **setPars**, the model will be recalculated just ONCE after all the changes have been made. But when setting through individual Parameter objects, the model will be recalculated after EACH parameter change:

```
# For Model object m1, supply 1 or more new parameter values in consecutive␣
↪order:
m1.setPars(2.5, 1.4, 1.0e3)   # This changes pars 1, 2, and 3.

# Can also change paramater auxiliary values by passing a string using the same
# syntax as with Standard XSPEC's `newpar` command:
```

(continues on next page)

(continued from previous page)

```
m1.setPars(.95, "1.8,,-5,-4,10,10")

# Now set parameters NON-CONSECUTIVELY by passing a Python dictionary object.
# This example changes pars 1, 2, 4, and 6:
m1.setPars(.95, 1.2, {4:9.8, 6:2.0})
```

Parameters can also be initialized by passing values to the *Model* object constructor. You do this by setting the Model constructor's *setPars* keyword argument to a tuple, list, or dictionary (or just a single value or string if only setting the first parameter):

```
# Supply values for parameters 1 and 3, use defaults for the rest.
m = Model("phabs*ga", setPars={1:1.5, 3:.2})
# Supply values for 1 and 2, use defaults for the rest.
m = Model("phabs*ga", setPars=(1.5, 0.7))
# Supply value only for 1.
m = Model("phabs*ga", setPars=1.5)
```

Finally, if you wish to set multiple parameters that belong to *different* model objects, you must use the *AllModels* container's **setPars** method (see *ModelManager*). This follows the same syntax rules as the single Model **setPars**, except that you also supply the Model objects as arguments:

```
# Change pars 1 and 3 in m1, and pars 1 and 2 in m2:
AllModels.setPars(m1, {1:6.4, 3:1.78}, m2, 3.5, 0.99)
```

2.1.9 Fitting

Once data and models are loaded, fitting is performed by calling the **perform** method of the *Fit* global object:

```
Fit.perform()
```

Some of the more frequently modified fit settings are the type of statistic to minimize and the maximum number of fit iterations to perform. These settings are attributes of *Fit*:

```
Fit.nIterations = 100
Fit.statMethod = "cstat"
Fit.statMethod = "chi"
```

Please see *FitManager* and the *Extended Tutorial* for *Fit*'s complete functionality.

To display the fit results at any time:

```
Fit.show()
```

2.1.10 Plotting

In Standard XSPEC, plot settings are adjusted using the **setplot** command while the plot is displayed through the **plot** command. In PyXspec, all plot settings and functionality is handled through the global *Plot* object (see *PlotManager* for full description). A device must be set before any plots can be displayed, and this done through the **device** attribute:

```
Plot.device = "/xs"
```

The device can also be set to print to an output file in several formats. The list of possible devices is given by the **cpd** command in the Standard XSPEC manual.

A typical setting to adjust is the X-axis units. You can choose to plot channel numbers, or select from various energy and wavelength units. The strings can also be abbreviated. Examples:

```
Plot.xAxis = "channel"
Plot.xAxis = "MeV"
Plot.xAxis = "Hz"
Plot.xAxis = "angstrom"
```

The displays of individual additive components or background spectra is toggled by setting their attributes to a bool:

```
Plot.add = True
Plot.background = False
```

Similarly log/linear settings for data plots (when using energy or wavelength units):

```
Plot.xLog = True
Plot.yLog = False
```

The current plot settings are displayed with:

```
Plot.show()
```

To actually display a plot, send 1 or more string arguments to the *Plot* __call__ method:

```
# Single panel plots
Plot("data")
Plot("model")
Plot("ufspec")
# Multi panel plots
Plot("data chisq")
Plot("data","model","resid")
# Call Plot with no arguments to repeat the previously entered Plot command
Plot()
```

After displaying a plot, you can get an array of the plotted values by calling one of *Plot*'s retrieval methods. All of these functions take an optional plot group number argument for the case of multiple plot groups, and all return the plot values in a Python list:

```
Plot("data")
xVals = Plot.x()
yVals = Plot.y()
yVals2 = Plot.y(2)  # Gets values for data in the second plot group
```

(continues on next page)

(continued from previous page)

```
modVals = Plot.model()
# To get a background array, Plot.background must be set prior to plot
Plot.background = True
Plot("data")
bkg = Plot.backgroundVals()
# Retrieve error arrays
xErrs = Plot.xErr()
yErrs = Plot.yErr()
```

2.2 Extended Tutorial

This assumes the user is familiar with the basics of PyXspec as explained in the *Quick Tutorial*.

2.2.1 Data

Background, Response, and Arf

When a *Spectrum* object is created from a spectral data file, PyXspec also reads the file's BACKFILE, RESPFILE, and ANCRFILE keywords and will load the corresponding background, response, and arf files. The spectrum's *Background* and *Response* objects are then available as attributes of the **Spectrum** class, while the arf file name becomes an attribute of the **Response** class:

```
s1 = Spectrum("file1")
b1 = s1.background
r1 = s1.response
arfFileName = r1.arf
```

Note: You never create **Background** and **Response** objects directly. They are accessible only through the **Spectrum** class attributes.

These attributes may also be used to add, change, or remove auxiliary files to an existing *Spectrum* object:

```
# Add or replace files:
s1.background = "newBackground.pha"
s1.response.arf = "newArf.pha"
# Removal examples:
s1.response = None
s1.background = ""
```

Background and *Spectrum* store their original file names in their **fileName** attribute. This means that while you SET the Spectrum.background object by assigning it a file name (as shown above), to GET the file name you must access its **fileName** attribute:

```
bkgFileName = s1.background  # Wrong!!! This returns the entire Background
                             # object, not a string.
bkgFileName = s1.background.fileName  # Correct
```

Response stores its RMF and optional ARF file names in its **rmf** and **arf** attributes respectively:

```
rmfFileName = r1.rmf
arfFileName = r1.arf
```

Background objects have some of the same attributes as *Spectrum* objects, such as **areaScale**, **exposure**, **energies**, and **values**. The Spectrum object's **values** array (actually a tuple) does NOT include contributions from the background. Those are stored separately in the associated Background object.

The *Spectrum* class also provides a **multiresponse** array attribute for assigning multiple detectors (or sources) to a spectrum. The standard 0-based Python array indices corresponding to the 1-based XSPEC source numbers:

```
# Set a response for source 2
s1.multiresponse[1] = "resp2.rsp"
# Get the response object for source 2
r2 = s1.multiresponse[1]
# Remove the response from source 2
s1.multiresponse[1] = None
# This is the same as doing s1.response = "resp1.rsp"
s1.multiresponse[0] = "resp1.rsp"
```

The rule is: when doing single-source analysis (typical of most XSPEC sessions) use the **response** attribute, otherwise use the **multiresponse** array.

Ignore/Notice

To ignore channels for a SINGLE spectrum, call the *Spectrum* object's **ignore** method passing a string following the same syntax as for Standard XSPEC's **ignore** command:

```
s1.ignore("20-30 50-**")
s1.ignore("**-5")
```

Similarly, to notice channels in a single spectrum:

```
s1.notice("10-30,80-**")
s1.notice("all")
```

As with Standard XSPEC, if the **x-axis** plot units are set to energies or wavelengths, **ignore** and **notice** will accept floating-point input assumed to be in those same units:

```
Plot.xAxis = "nm"
# Ignore channel bins corresponding to 15.0 to 20.0 nm wavelengths:
s1.ignore("15.-20.")
```

The currently noticed channel ranges are displayed for each spectrum in the `AllData.show()` output. You can also get a list of the individual noticed channel numbers from *Spectrum*'s **noticed** attribute:

```
>>> s1.noticed
[3,4,5,7,8,10]
```

To apply **ignore** and **notice** commands to ALL loaded spectra, call the methods from the global *AllData* object. To apply to a subset of loaded spectra, add a range specifier to the left of the colon:

```
# These apply to all loaded spectra
AllData.ignore("100-120, 150-200")
AllData.notice("all")
AllData.ignore("bad")
# These apply to a subset of loaded spectra
AllData.ignore("1-3: 60-65")
AllData.notice("2-**:50-60")
```

2.2.2 Models

Model With Multiple Data Groups

When a model is defined and spectra are assigned to multiple data groups, PyXspec will generate a *Model* object copy for each data group (assuming the spectra also have responses attached). So if:

```
m1 = Model("phabs*ga")
AllData("file1 2:2 file2")
```

then there are 2 *Model* objects for the model definition `phabs*gaussian`. The variable *m1* is set to the object belonging to data group 1, and to get the object for data group 2 do:

```
m2 = AllModels(2)
```

m1 and *m2* will each have the same set of *Component* and *Parameter* objects.

Parameters can be accessed directly by index from the *Model* objects, and these indices are numbered from 1 to **nParameters** for ALL data group copies. So for the *phabs*ga* example above:

```
p = m1(2) # Returns the 2nd ('LineE') parameter from the model for data group 1.
p = m2(2) # Returns the 2nd ('LineE') parameter from the model for data group 2.
p = m2(6) # Wrong!!
```

Defining Multiple Models

Beginning with XSPEC12, it became possible to assign multiple sources to spectra, and each source may have its own model function definition. To keep track of multiple model definitions, XSPEC requires that you assign them names. In PyXspec, the model name and source number are supplied as additional arguments to the *Model* __init__ function:

```
# Define a model named "alpha" assigned to source 1
m_1_1 = Model("phabs*po","alpha")
# Define a model named "beta" assigned to source 2
m_2_1 = Model("const*bbody","beta",2)
# (In both of these cases, the returned object belongs to data group 1)
```

Note: As with Standard XSPEC, to define a model for source numbers > 1 you first must load a detector response for the source. See "Background, Response, and Arf" in the previous section.

Note that in all previous examples in this tutorial, we have been using unnamed models which were assigned to source 1. Named models and source numbers may also be defined directly into the *AllModels* container by passing in a tuple:

```
# Define a model named "defn1" assigned to source 1
AllModels += ("phabs*po", "defn1")
# Define a model named "defn2" assigned to source 2
AllModels += ("const*bbody", "defn2", 2)
# This replaces "defn1" with an unnamed model for source 1
AllModels += "phabs*gaussian"
```

and from which *Model* objects can be retrieved:

```
# Get the "defn2" Model object for data group 1
m_2_1 = AllModels(1,"defn2")
# ...and for data group 2
m_2_2 = AllModels(2,"defn2")
```

To view all current source number and model assignments, see the *AllModels.sources* attribute, which displays a dictionary of the [source number]:[model name] pairs.

To remove model definitions:

```
# Remove all data group copies of "defn2"
AllModels -= "defn2"
# Remove all data group copies of the unnamed model (defined above
#    as "phabs*gaussian")
AllModels -= ""
# Remove all copies of ALL model definitions
AllModels.clear()
```

Component And Parameter Access Part 2

When PyXspec constructs a *Model* object, it immediately adds to it an attribute of type *Component* for every component in the model expression. The attribute has the same (full) name as the component in the original expression, allowing you to access it like:

```
m = Model("phabs*pow")
c2 = m.powerlaw
```

However when a model contains multiple copies of the same component, this type of access becomes ambiguous. So to distinguish among copies, for any component making its 2nd or more appearance (from left to right), PyXspec will append *_n* to the attribute name where *n* refers to the component's position in the expression (again from left to right). Or to put it more simply:

```
m = Model("phabs*po + po")
# This gets the leftmost powerlaw component
pow1 = m.powerlaw
# This gets the rightmost, which is the 3rd component in the expression.
pow2 = m.powerlaw_3
```

The *Model* object also stores an attribute which is a just a list of the names of its constituent *Component* attributes:

```
>>> m.componentNames
['phabs', 'powerlaw', 'powerlaw_3']
```

This may be useful for example if writing a loop to access each of a model's components. Similarly *Component* objects have a **parameterNames** attribute, listing the names of their constituent *Parameter* attributes:

```
>>> m.powerlaw.parameterNames
['PhoIndex', 'norm']
```

Gain Parameters (Response Models)

Response Models differ from the regular kind in that they act on a Response rather than directly calculate a flux. At present there is only one kind of Response Model in Xspec, and this is **gain**. **gain** is a built-in attribute of all *Response* objects, and is of the class type *RModel*. It has 2 parameters for adjusting the energies of a Response: **slope** and **offset**. Gain parameters are initially off by default, but may be turned on simply by setting either one. For example:

```
s = Spectrum("file1")
# The spectrum's response has a gain attribute that is not in use,
# which is the equivalent of having a slope fixed at 1.0 and offset = 0.0.
r = s.response
# Setting either the slope or offset turns the gain on for this response.
# Both slope and offset will now be fit parameters.
r.gain.slope = 1.05
# The previous setting leaves the offset at 0.0.  Now we'll change it.
r.gain.offset = .05
# You can set slope and offset at the same time using Response's setPars method.
r.setPars(.99, .03)
```

slope and **offset** are *Parameter* objects and therefore have the same interface as regular model parameters:

```
# Modify the parameter's auxilliary values
r.gain.offset = ".08,,.01,.01,.5,.5"
# Set a parameter link
r.gain.offset.link = ".005*1"
```

To remove the response fit parameters and return the *Response* back to its original state, call the *gain.off()* method:

```
# This deletes the slope and offset parameters.
# Any references to them become invalid.
r.gain.off()
```

Flux Calculations

To perform a Standard XSPEC **flux** or **lumin** calculation, call the *AllModels* methods **calcFlux** or **calcLumin** respectively:

```
AllModels.calcFlux(".3 1.0")
AllModels.calcFlux(".1 10.0 err")
AllModels.calcLumin(".1 10. .05 err")
```

As in Standard XSPEC the results will be stored with the currently loaded spectra:

```
>>> s1 = AllData(1)
>>> s1.flux
(5.7141821510911499e-14, 0.0, 0.0, 4.0744161672429196e-05, 0.0, 0.0)
>>> s1.lumin
(30.972086553634504, 0.0, 0.0, 0.056670019567301853, 0.0, 0.0)
```

unless there are no spectra, in which case the results are stored with the model object:

```
>>> AllModels(1).flux
(5.6336924399373855e-10, 0.0, 0.0, 0.05929616315253175, 0.0, 0.0)
```

Local Models in C/C++/Fortran

In Standard XSPEC, local model libraries are built with the **initpackage** comamnd and then loaded with **lmod**. The *AllModels* container supplies both of these functions for doing the same thing in PyXspec:

```
AllModels.initpackage("myLocalMods","lmodel.dat")
AllModels.lmod("myLocalMods")
```

By default this looks in the directory set by the LOCAL_MODEL_DIRECTORY variable in your *~/.xspec/Xspec.init* start-up file. You can override this by giving these functions an absolute or relative path as a *dirPath* keyword argument (see *ModelManager* for details).

Local Models in Python

You can also write model functions in Python and insert them into the XSPEC library with the *AllModels* **addPyMod** method. You simply define a function with 3 arguments for energies, parameters, and flux. For example a powerlaw model function might look like:

```
def lpow(engs, params, flux):
   for i in range(len(engs)-1):
      pconst = 1.0 - params[0]
      val = math.pow(engs[i+1],pconst)/pconst - math.pow(engs[i],pconst)/pconst
      flux[i] = val
```

XSPEC will pass **tuples** containing the energy and parameter values to your function. For the flux array, it will pass a **list** pre-sized to *nE-1*, where *nE* is the size of the energies array. Your model function should fill in this list with the proper flux values. (For additional optional arguments to your model function, please see the documentation for the **addPyMod** method in *ModelManager*.)

The second thing you must define is a **tuple** containing the parameters' information strings, one string for each parameter in your model. This is equivalent to the parameter strings you would define in a *model.dat* file when adding local models in standard XSPEC, and it requires the same format. (See Appendix C of the XSPEC manual for more details.) So with the powerlaw function above which takes just 1 parameter, you might define a tuple as:

```
powInfo = ("phoIndex   \"\"  1.1  -3.  -2.  9.  10.  0.01",)
```

Note the need for the trailing comma when there's just one parameter string. This is to let Python know that *powInfo* is a tuple type and not a string.

Once you've defined your function and parameter information, simply call:

```
AllModels.addPyMod(lpow, powInfo, 'add')
```

The 3rd argument tells XSPEC the type of your model function ('add', 'mul', or 'con'). After this call your function will be added to the list of available model components, which you can see by doing *Model.showList()*. Your model will show up with the same name as your original Python function ('lpow'), and is ready for use in future model definitions.

2.2.3 Fitting

Error

The **error** command is implemented through *Fit* (see *FitManager*), and the results are stored with the chosen *Parameter* object(s). The **error** attribute stores a tuple containing the low and high range values for the parameter, and the 9-letter status string to report problems incurred during the error calculation.

Example: Estimate the 90% confidence range for the 4th parameter

```
>>> Fit.error("2.706 4")
>>> par4 = AllModels(1)(4)
>>> par4.error
(0.11350354517707145, 0.14372981075906774, 'FFFFFFFFF')
```

Query

During a *Fit.perform()* operation, the default is to query the user whenever the fit has run the maximum number of iterations, as set by the *Fit.nIterations* attribute. You can change this behavior with the **query** attribute:

```
# When nIterations is reached, continue the fit without stopping to query.
Fit.query = "yes"
# Stop fit at nIterations and do not query.
Fit.query = "no"
# Query the user when nIterations is reached.
Fit.query = "on"
```

Steppar

The Standard XSPEC **steppar** command is also implemented through the global *Fit* object. You supply it with a string following the same **steppar** command syntax rules. For example:

```
# Step parameters 1 and 2 through the given range values
# over a 10x10 2-D grid.
Fit.steppar("1 20. 30. 10 2 .05 .08 10")
```

2.2.4 Fakeit

PyXspec provides access to standard XSPEC's **fakeit** command, which is for creating spectra with simulated data. It is called through the *AllData.fakeit* method:

```
AllData.fakeit(nSpectra=1, settings=None, applyStats=True, filePrefix="")
```

Note: If *AllData.fakeit* is run when spectra are currently loaded, it will follow the same rules as the standard XSPEC **fakeit** function: It will REMOVE ALL pre-existing spectra and replace each one with a simulated spectrum (even if *nSpectra* is less than the number originally loaded).

As those familiar with standard **fakeit** know, the user is normally prompted for quite a bit of additional information needed to generate the fakeit files. However the goal here is to have NO additional prompting, and that requires that all information must be entered as arguments to the *AllData* fakeit method call. This is done by passing objects of the *FakeitSettings* class to *AllData.fakeit*, as we'll show further below.

Note: Unless stated otherwise, assume all spectra are OGIP **type-1** (1 spectrum per file).

For the simplest of cases, you don't need to create any *FakeitSettings* objects. Just pass in the number of fake spectra you'd like to create:

```
# Create 3 fake spectra using only default settings.
AllData.fakeit(3)
```

The *fakeit* function will then create a default *FakeitSettings* object for each of the 3 spectra. By default, a FakeitSettings object will have empty strings for all of its attributes, and these are handled differently depending on whether the fake spectrum is replacing a currently loaded spectrum, or creating one from scratch.

From Existing Spectra

When replacing an existing spectrum, *FakeitSettings* attributes with empty strings will simply take their value from the original spectrum. Also note that the **response** and **arf** settings for the original spectrum CANNOT be modified for the fakeit spectrum. If a name is filled in for either of these attributes, it will be ignored. If you wish to modify these, you can make the change to the original spectrum prior to calling fakeit. [The one exception is when the original spectrum has no response, in which case the response attribute MUST be filled in.] If the **fileName** attribute is empty, XSPEC will generate a default output name derived from the original file name.

From Scratch

When creating from scratch, an empty string implies "none" for the arf and background, 1.0 for exposure and correction, and XSPEC's default dummy response for the response attribute. If the fileName attribute is empty, XSPEC will generate a default output file name based on the response name, and it will include an auto-incremented index to prevent multiple output files from overwriting each other.

FakeitSettings Objects

To create a fake spectrum with anything other than default settings, you must supply a *FakeitSettings* object for that spectrum. The FakeitSettings attributes are: response, arf, background, exposure, correction, backExposure, and fileName. All are string types, though exposure, backExposure, and correction can also be entered as floats. Attributes can be set upon object construction, or anytime afterwards:

```
fs1 = FakeitSettings("response1.rsp", exposure = 1500.0)
fs1.background = "back1.pha"
```

A new FakeitSettings object can also be made by copying an existing one:

```
fs2 = FakeitSettings(fs1)
```

And now pass the objects to the fakeit method, either in a list, dictionary, or as a single object:

```
# Apply settings to fakeit spectra 1 and 2:
AllData.fakeit(2,[fs1,fs2])
# Apply setting to fakeit spectrum 1, use defaults for spectrum 2:
AllData.fakeit(2, fs1)
# Apply settings to fakeit spectra 2 and 4, use defaults for 1 and 3:
settingsDict = {2:fs1, 4:fs2}
AllData.fakeit(4, settingsDict)
# Create 4 fakeit spectra from the same settings object:
settingsList = 4*[fs1]
AllData.fakeit(4, settingsList)
```

The remaining 2 arguments to the *AllData.fakeit* function are for choosing whether to apply statistical fluctuations (default = True), and whether to add an optional prefix string to the names of all output files.

OGIP Type-2 Files

With **OGIP type-2** files, multiple spectra may be placed in a single file. The important thing to recognize when generating type-2 fakeit files is that the **exposure, correction, backExposure,** and **fileName** attributes apply to the output *files* and not the individual spectra. Therefore these settings will be ignored for all but the first spectrum in a file. For example:

```
# Start with 4 spectra loaded, in 2 type-2 files:
AllData("myDataFile1.pha{1-2} myDataFile2.pha{7-8}")
# Create settings for the 4 fake spectra that will be generated from these:
fs1 = FakeitSettings(background="back1.pha", exposure=250.)
# The exposure setting in fs2 will be ignored!!!
fs2 = FakeitSettings(background="back2.pha", exposure=100.)
fs3 = FakeitSettings(fileName="myFakeitFile_2.pha")
fs4 = FakeitSettings(fs3)
# The following change will be ignored!!!
fs4.fileName = "myFakeitFile_3.pha"
# Now generate the fakeit files:
AllData.fakeit(4, [fs1,fs2,fs3,fs4])
```

The above will generate 4 fakeit spectra, placed in 2 type-2 files. The exposure setting for spectrum 2 and the fileName setting for spectrum 4 will be ignored. Those values are only set by spectra 1 and 3.

For more fakeit details and examples, please check *FakeitSettings* and the **fakeit** method description in *DataManager*.

2.2.5 Monte Carlo Markov Chains (MCMC)

All MCMC operations are handled either by objects of class *Chain*, or the global *AllChains* container object (see the *ChainManager* class description). To create a new chain based on the current fit parameters, simply create a **Chain** object by passing it an output file name:

```
c1 = Chain("chain1.fits")
```

The above call creates the file *chain1.fits*, performs an MCMC run using the default *burn, fileType, length, proposal, rand, and temperature* values, and automatically places the new object in the *AllChains* container. These default settings are stored as attributes of *AllChains*:

```
# Ensure that new chains will burn the first 100 iterations, will
# have length 1000, and will use the proposal "gaussian fit"
AllChains.defBurn = 100
AllChains.defLength = 1000
AllChains.defProposal = "gaussian fit"
c2 = Chain("chain2.fits")
```

You can also override the *AllChains* default settings by passing additional arguments to *Chain* upon construction:

```
# Length will be 2000 for this chain, use defaults for all other settings.
c3 = Chain("chain3.fits", runLength = 2000)
```

The new chain objects will then store their own settings as attributes:

```
>>> c2.burn
100
>>> c2.runLength
1000
>>> c3.runLength
2000
```

All of a chain object's attributes will be displayed when calling its *show()* method.

To append a new run to an existing chain object, call the object's *run()* method. The appending run will use the object's current attribute settings, and not the *AllChains* default settings:

```
# This will append a run of length 3000 to the c3 chain object, and with a
# Metropolis-Hastings temperature of 50.0:
c3.runLength = 3000
c3.temperature = 50.0
c3.run()
```

Display the new chain length:

```
>>> c3.totalLength
5000
```

To **overwrite** rather than append to an existing chain object, call run with its append argument set to *False*:

```
# This erases the results of any previous runs for object c3.
c3.run(False)
```

```
>>> c3.totalLength
3000
```

New chains are loaded into *AllChains* by default, but you can unload or reload them using the *AllChains* arithmetic operators:

```
# Chain c2 may be unloaded by passing its chain index number
AllChains -= 2
# OR by passing the object itself
AllChains -= c2
# 2 ways to remove ALL chains
AllChains -= '*'
AllChains.clear()

# Reload an existing chain object
AllChains += c2
# Load a chain from an existing chain file
AllChains += "earlierChain.fits"
# Create a new chain, to be stored in file "chain4.fits"
AllChains += "chain4.fits"
```

As with Standard XSPEC, unloading a chain will leave the chain's file intact. It merely removes the chain from XSPEC's calculations. To display information about the currently loaded chains, call *AllChains.show()*.

You may also get a chain object from the container at any time by passing it an index number:

```
# Retrieve a chain object for the 4th chain in the container
c4 = AllChains(4)
```

2.2.6 Plotting

All of the plotting options available in Standard XSPEC's **setplot** command are now implemented as attributes of the *Plot* object. Some of these are mentioned in the *Quick Tutorial*, and please see the *PlotManager* class reference for the complete guide.

One setting of particular interest is the **commands** attribute. This is a tuple of user-entered **PLT** command strings which are added to XSPEC's auto-generated commands when performing a plot, and is modified through *Plot*'s **addCommand** and **delCommand** methods. For example, to enter a PLT command to place an additional label at the specified coordinates on the plot:

```
Plot.addCommand("label 1 pos 10 .05 \"Another Label\"")
```

To view the currently loaded commands:

```
print(Plot.commands)
```

and to remove the 3rd command from the tuple:

```
Plot.delCommand(3)
```

2.2.7 XSPEC Settings

Most of the internal switches set through Standard XSPEC's **xset** command are now set through attributes of the global *Xset* object. (See *XspecSettings* for the full class description.)

Examples:

```
Xset.abund = "angr"
Xset.cosmo = "50 .5 0."
Xset.xsect = "bcmc"
```

Xset also provides the methods **addModelString** and **delModelString** to set the <*string name*>,<*string value*> pairs which are used by certain models. The <*string name*> argument is case-insensitive:

```
Xset.addModelString("APECROOT","1.3.1")
Xset.addModelString("APECTHERMAL","yes")
Xset.delModelString("neivers")
```

The entire collection of <*name*>,<*value*> pairs may be set or retrieved with the *Xset.modelStrings* attribute:

```
# Replace all previous entries with a new dictionary
Xset.modelStrings = {"neivers":"1.1", "apecroot":"1.3.1"}
# Clear out all entries:
Xset.modelStrings = {}
```

Xset.show() will display all of the current settings including the current <*string name*>,<*string value*> pairs.

2.2.8 Logging And XSPEC Output

The *Xset* object provides attributes and methods for controlling output chatter level and for creating log files:

```
# Get/Set the console chatter level
ch = Xset.chatter
Xset.chatter = 10
# Get/Set the log chatter level
lch = Xset.logChatter
Xset.logChatter = 20

# Create and open a log file for XSPEC output
# This returns a Python file object
logFile = Xset.openLog("newLogFile.txt")
# Get the Python file object for the currently opened log
logFile = Xset.log
# Close XSPEC's currently opened log file.
Xset.closeLog()
```

2.2.9 Exceptions And Error Handling

PyXspec utilizes the standard Python **try/except/raise** mechanism for handling and reporting errors. Currently only exception objects of the class **Exception** are ever raised. In the future other (more specific) error classes may be used, but they should always be derived from **Exception**. So you can catch all PyXspec exceptions with code such as:

```
try:
    # Only 4 spectra are currently loaded
     s = xspec.AllData(5)
except Exception as msg:
     print(str(msg))
```

which will print the error message

Error: Spectrum index number is out of range: 5

PyXspec raises errors in a variety of situations, such as for invalid input argument syntax, or for input which is invalid within the context of the call (as in the example above). It can also raise exceptions if you try to rebind a class attribute when such modification is not permitted.

2.2.10 Adding Attributes To PyXspec Objects

A particularly novel feature of Python (in comparison with say C++) is that it allows you to create new attributes "on the fly". The attributes don't have to have been part of the original class definition:

```
class C:
    pass

x = C()
x.pi = 3.1416
```

The downside of course is that spelling or case sensitive errors become much harder to detect. For example, with PyXspec's *Plot* object:

```
Plot.yLog = True  # Correct
Plot.ylog = True  # Wrong!
```

In the second case, standard Python will simply add a new attribute named *ylog* to *Plot*, and this will have no effect on the actual plot since PyXspec is only looking for an attribute named *yLog*.

So operating under the assumption that this downside outweighs the benefits, we've decided to **disable** the ability to add new attributes to PyXspec class objects. A misspelling or case error will instead raise an **Exception** object. And since some users may genuinely wish to add their own attributes to PyXspec classes, this default behavior may be overridden by toggling the *Xset.allowNewAttributes* flag:

```
s = Spectrum("dataFile.pha")
s.myNewIndex = 10  # Error: Will raise an exception
Xset.allowNewAttributes = True

s.myNewIndex = 10 # OK
#  .
#  .
#  .  In this section you can now add new attributes to any PyXspec object,
#  .  but attribute spelling errors will go undetected.
```

(continues on next page)

(continued from previous page)

```
#  .
#  .

Xset.allowNewAttributes = False
```

2.2.11 Using With Other Packages

One of the primary benefits of PyXspec is that it makes it much easier to use XSPEC data and results in 3rd party packages. For example you can bypass XSPEC's built-in plotting functions in favor of a Python plotting library such as Matplotlib:

```
#!/usr/bin/python

from xspec import *
import matplotlib.pyplot as plt

# PyXspec operations:
s = Spectrum("file1.pha")
m = Model("phabs*po")
Fit.perform()
Plot.device = '/null'
Plot('data')
# Get coordinates from plot:
chans = Plot.x()
rates = Plot.y()
folded = Plot.model()

# Plot using Matplotlib:
plt.plot(chans, rates, 'ro', chans, folded)
plt.xlabel('channels')
plt.ylabel('counts/cm^2/sec/chan')
plt.savefig('myplot')
```

The above code produces a Matplotlib plot of the spectral data and folded model vs. channels (similar to what you get with Standard XSPEC's "plot data" command). It makes use of the *Plot* object's **x** and **y** methods to return the coordinates of the spectrum plot, and the **model** method to get the folded model values. These are then forwarded to Matplotlib's plotting command.

CHAPTER

THREE

PYXSPEC CLASS REFERENCE

3.1 Classes

3.1.1 Background

class xspec.**Background**(*backTuple*, *parent*)

Background spectral data class.

Attributes (all are get-only)

- areaScale
- backScale
- exposure
- fileName
- isPoisson
- values
- variance

property areaScale

The Background area scaling factor (GET only).

This is either a single float (if file stores it as a keyword), or a tuple of floats (if file stores column).

property backScale

The Background back scaling factor (GET only).

This is either a single float (if file stores it as a keyword), or a tuple of floats (if file stores column).

property exposure

The exposure time keyword value [float] (GET only).

property fileName

The spectrum's file name [string] (GET only).

property isPoisson

Boolean flag, True if spectrum has Poisson errors (GET only).

property values

Tuple of floats containing the background rates array in counts/sec (GET only).

property variance

Tuple of floats containing the variance of each channel (GET only).

3.1.2 Chain

class xspec.**Chain**(*fileName*, *fileType=None*, *burn=None*, *runLength=None*, *proposal=None*, *rand=None*, *rescale=None*, *temperature=None*, *algorithm=None*, *walkers=None*)

Monte Carlo Markov Chain class.

Methods

- __init__
- run
- show

Attributes ((*) = get-only)

- algorithm
- burn
- fileName*
- fileType*
- proposal
- rand
- rescale
- runLength
- temperature
- totalLength*
- walkers

The following attribute settings will apply to the NEXT run for this chain: *algorithm*, *runLength*, *proposal*, *rescale*, *temperature*, *burn*, *rand*, *walkers*.

The *algorithm*, *burn*, *rand*, and *walkers* settings are irrelevant if run is performing an appending operation.

__init__(*fileName*, *fileType=None*, *burn=None*, *runLength=None*, *proposal=None*, *rand=None*, *rescale=None*, *temperature=None*, *algorithm=None*, *walkers=None*)

Construct a chain object, perform a run, and load into AllChains
container.

The only required argument is *fileName*. All other arguments will take their default values from the current settings in the *AllChains* container.

run(*append=True*)

Perform a new chain run, either appending to or overwriting an
existing chain.

Args:

***append*: If this is set to True the new run will be appended.**
If False, the new run will overwrite. Note that the *algorithm*, *burn*, *rand*, and *walkers* settings do not apply when appending.

show()

Display current settings of Chain object's attributes.

property algorithm

The current chain algorithm. Valid settings are 'gw' (Goodman-Weare) or 'mh' (Metropolis-Hastings).

property burn

The number of steps that will be thrown away prior to storing the chain [int].

property fileName

Chain output file name.

property fileType

Output format of the chain file [string]. Will be either 'fits' (the default), or 'ascii'.

property proposal

The proposal distribution and source of covariance information to be used for the next run [string].

Examples: 'gaussian fit', 'cauchy fit','gaussian chain', etc.

See the 'chain' command in the standard XSPEC manual for more information.

property rand

Determines whether chain start point will be randomized (True) or taken from the current parameters (False).

property rescale

Determines whether a rescale fraction will be applied to the covariance matrix. [float] or None

property runLength

The length of chain to be added during the next run [int].

property temperature

The temperature parameter used in the Metropolis-Hastings algorithm for the proposal acceptance or rejection [float].

property totalLength

The cumulative length of the chain [int]. This will increase every time a run is performed.

property walkers

The number of walkers to be used for 'gw' chains [int].

3.1.3 ChainManager

class xspec.**ChainManager**

Monte Carlo Markov Chain container.

PyXspec automatically creates a single object of this class, named *AllChains*.

Methods

- __call__ (the '()' operator)
- __iadd__ (the '+=' operator)
- __isub__ (the '-=' operator)
- best
- clear
- dic
- margin
- marginResults
- show
- stat

Attributes

These are the values which will be used when creating new Chain objects, unless they are explicitly overridden as arguments to the Chain class constructor. For more detail, see the descriptions for the corresponding attributes in the Chain class doc.

- defAlgorithm
- defBurn
- defFileType
- defLength
- defProposal
- defRand
- defRescale
- defTemperature
- defWalkers

`__call__`(*index*)

Get a Chain object from the *AllChains* container.

Args:

***index*: The index of a currently loaded chain file. The list**
of currently loaded chains can be seen with the AllChains.show() method. The valid range is: 1 <= *index* <= nLoadedChains.

Note that the returned Chain object's modifiable attributes will be initialized with the current AllChains *def<attribute>* settings:

```
# Example: Load 2 chains from pre-existing files:
AllChains += "chain1.fits"
AllChains += "chain2.fits"
# and use the __call__ method to retrieve a Chain object
#   for the 2nd chain:
c2 = AllChains(2)
```

`__iadd__`(*chain*)

Load a pre-existing chain into the *AllChains* container.

Args:

***chain*: may be a currently existing chain object which had been**
unloaded earlier:

```
>>> AllChains += myChain1
```

the filename of an existing chain file:

```
>>> AllChains += "chainFile.fits"
```

or the filename of a new chain:

```
>>> AllChains += "newChainFile.fits"
```

The last example will also perform a chain run using the default settings.

__init__()

__isub__(*chain*)

Unload one or more chain objects from container.

Args:

chain: may either be a chain object:

```
>>> AllChains -= myChain1
```

a filename:

```
>>> AllChains -= "chainFile.fits"
```

the chain's current index [int] in the *AllChains* container:

```
>>> AllChains -= 2
```

or a '*' to unload ALL chains (equivalent to AllChains.clear()):

```
>>> AllChains -= '*'
```

best()

Return the best-fit parameters and statistic in the loaded chains.

The values are returned in a tuple with the best-fit statistic appearing at the end.

clear()

Unload all chains from container

dic()

Calculate the deviance information criterion from all chains

margin(*argString*)

Calculate a multi-dimensional probability distribution.

Args:

argString: [string]

This uses identical syntax to the standard interactive XSPEC margin command. "<step spec> [<step spec> ...]" where:

<step spec> ::= [<log|nolog>] [<modName>:]<param index>
<low value> <high value> <# steps>

See the XSPEC manual for a more detailed description of specs.

Examples:

```
# Calculate a 2-D probability distribution of parameter 1
# from 10.0-12.0 in 20 linear bins, and parameter 3 from
# 1.0-10.0 in 5 logarithmic bins.
AllChains.margin("1 10.0 12.0 20 log 3 1.0 10.0 5")
# Now calculate for parameter 2 in 10 log bins and parameter 4
# in 10 linear bins.
AllChains.margin("2 10.0 100.0 10 nolog 4 20. 30. 10")
```

marginResults(*arg*)

Retrieve values from the most recent margin calculation.

Args:

arg:

argument should either be 'probability' or a parameter specifier. A parameter specifier should be a string of the form:

'[<modName>:]<parNum>'

or simply an integer <parNum>.

Returns the requested values as a list of floats.

show()

Display information for current attributes and loaded chains.

stat(*parIdx*)

Display statistical information on a particular chain parameter.

Args:

***parIdx*: The parameter index number, including optional model**
name: [<modName>:]<idx>. May be entered as a string or int (if no model name).

property defAlgorithm

Default chain algorithm (orig = 'gw').

property defBurn

Default burn length for new Chain objects (orig = 0).

property defFileType

Default output file format (orig = 'fits').

property defLength

Default chain length (orig = 100).

property defProposal

Default chain proposal (orig = 'gaussian fit').

property defRand

Default randomization setting (orig = False).

property defRescale

Default covariance rescale fraction (orig = None).

property defTemperature

Default chain temperature (orig = 1.0).

property defWalkers

Default walkers parameter for 'gw' chains (orig = 10).

3.1.4 Component

class xspec.**Component**(*compName*, *parNames*)

Model component class.

Attributes

- **name**
  The full name of the Component (get only).
- **<parameters>**
  Component contains an attribute of type **Parameter** for every parameter in the component. The attribute name is the same as the parameter name in xspec.
- **parameterNames**
  List of Component's parameter names (get only).

3.1.5 DataManager

class xspec.**DataManager**

Spectral data container.

PyXspec automatically creates a single object of this class, named *AllData*.

Methods

- __call__ (the '()' operator)
- __iadd__ (the '+=' operator)
- __isub__ (the '-=' operator)
- clear
- diagrsp
- dummyrsp
- fakeit
- ignore
- notice
- removeDummyrsp
- show

Attributes (get-only)

- nGroups
- nSpectra

__call__(*expr*)

DataManager get or set spectra.

Args:

***expr*:**

Get:
An integer referring to the spectrum index number. Returns the spectrum, or raises an Exception if the integer is out of range.

Set:
A string following the same syntax rules as Xspec's traditional "data" command handler.

__iadd__(*spectra*)

Add 1 spectrum to the data container.

Args:
spectra: The data filename string.

__init__()

__isub__(*spectra*)

Remove 1 or all spectra from the data container.

Args:

***spectra*: Either a single spectrum index number [int], a single**
Spectrum object, or the string '*' to remove all.

clear()

Remove all spectra from the data container.

diagrsp()

Diagonalize the current response matrix for ideal response.

All currently loaded responses will be replaced with diagonal response matrices. The energy range and channel binning information are retained from the original response, as is the effective area. The channel values are mapped directly into the corresponding energy ranges to simulate a detector with perfect spectral resolution.

To remove diagonal responses and restore the originals, call the *AllData.removeDummyrsp()* method.

dummyrsp(*lowE=None*, *highE=None*, *nBins=None*, *scaleType=None*, *chanOffset=None*, *chanWidth=None*)

Create a dummy response and apply it to all spectra.

Args: (all are optional)
lowE: Input response energy lower bound, in keV. [float]

highE: Input response energy higher bound, in keV. [float]

***nBins*: Number of bins into which the energy range is**
divided [int].

scaleType: 'log' or 'lin' [string]

***chanOffset*: Starting value of dummy channel**
energies. [float]

chanWidth: Energy width of the channel bins. [float]

If this is set to 0, the dummy response can only be used for evaluating model arrays, and not for fitting to spectra.

Examples:

```
# All values are optional, use keywords to enter values
# non-consecutively.  Unspecified values revert to the
# current defaults.
AllData.dummyrsp(.3, 30., 100, chanWidth=.5)
AllData.dummyrsp(highE = 50.)
AllData.dummyrsp(.1,10.,100,"lin",.0, 1.0)
```

Initial defaults: *lowE* = .1, *highE* = 50., *nBins* = 50, *scaleType* = 'log', *chanOffset* = .0, *chanWidth* = .0.

The defaults for *lowE*, *highE*, *nBins*, *scaleType*, and *chanOffset* will be modified for each explicit new entry. *chanWidth* always defaults to 0.

To remove dummy responses and restore actual responses (if any), call the *removeDummyrsp()* method.

To apply a dummy response to just a single spectrum, use the *Spectrum.dummyrsp* method.

fakeit(*nSpectra=1*, *settings=None*, *applyStats=True*, *filePrefix=''*, *noWrite=False*)

Produce spectra with simulated data using XSPEC's fakeit command.

Note that if this method is run when spectra are currently loaded, it will follow the same rule as the standard XSPEC fakeit function: It will REMOVE ALL pre-existing spectra and replace each one with a simulated spectrum (even if *nSpectra* is less than the number originally loaded).

Args: (all are optional)

nSpectra: The number of fake spectra to produce. [int]

If there are nOrig pre-existing spectra loaded at the time this function is called and nSpectra < nOrig, nSpectra will be RESET to nOrig (see note above).

If nSpectra == nOrig, then each of the fake spectra will use the settings from the respective original spectra for their defaults (see the FakeitSettings class description).

If nSpectra > nOrig, then settings for the fake spectra numbered above nOrig will not be based on pre-existing spectra (if any).

settings: A collection of 0 to nSpectra FakeitSettings objects.

This may be entered as a list, a dictionary, a single FakeitSettings object, or `None`. If *settings* is a dictionary, the key,value pairs should be the spectrum index number (1 is lowest) and the FakeitSettings object.

This function will match up FakeitSettings objects 1-to-1 with the nSpectra fake spectra to be created.

If user provides FEWER than nSpectra FakeitSettings objects, fakeit will generate the necessary additional objects with their default settings.

If MORE than nSpectra FakeitSettings objects are provided, the extra objects will be ignored.

***applyStats*: If set to True, statistical fluctuations will be**
included in the generation of fake spectra. [bool]

***filePrefix*: Optional string to attach as a prefix to default fakeit**
output file names.

Note that this only applies when using the default file names. If a file name is explicitly entered in the FakeitSettings.fileName attribute, it will not make use of this.

***noWrite*: If set to True, no fakeit output files will be generated.**
Default is False. [bool]

Examples:

```
# Assume no data is loaded, but a model is defined:

AllData.fakeit()
# Creates 1 fake spectrum using the default FakeitSettings object,
# which has all input strings empty.  So it will use XSPEC's internal
# dummy response and its output file name will be dummy_rsp_1.fak.
```

(continues on next page)

(continued from previous page)

```
# Now assume AllData contains 2 spectra PRIOR to running EACH of the
# following commands, then:

AllData.fakeit()
# Creates 2 fake spectra with all settings (response, arf,
# background, exposure, corrscale, backExposure, filenames) based
# on the original spectra.  The original 2 spectra are removed from
# AllData.

AllData.fakeit(3)
# Creates the first 2 spectra as above.  The 3rd fake spectrum is
# based on the default FakeitSettings object and its output filename
# will be dummy_rsp_3.fak

fs = FakeitSettings(background="back1.pha", exposure=2000.0)
sl = 3*[fs]
AllData.fakeit(3, sl)
# Same as above, but all 3 fake spectra will have a background file
# based on back1.pha, and exposure time = 2000.0 sec.

AllData.fakeit(3, sl, False, "my_fake_")
# Same as above, but no statistical fluctuations will be applied to
# fake spectra, and all output files will have the "my_fake_"
# prefix attached.

fs1 = FakeitSettings("resp1.rmf","arf1.pha",exposure=1500.)
fs2 = FakeitSettings(fs1)
fs2.response = "resp2.rmf"
sd = {3:fs1, 5:fs2}
AllData.fakeit(5, sd)
# Creates 5 fake spectra.  The first 2 use the settings from the
# originally loaded data.  Spectra 3 and 5 use the settings from
# the fs1 and fs2 FakeitSettings objects, which differ only in their
# response names.  Spectrum 4 uses the default FakeitSettings object.
```

ignore(*ignoreRange*)

Apply an ingore channels range to multiple loaded spectra.

Args:

***ignoreRange*: String specifying the spectra ranges and/or**
channel ranges to ignore, or "bad".

This follows the same syntax as used in the standard Xspec "ignore" command, except that the spectrum range always defaults to ALL spectra.

If the channel ranges are floats rather than ints, they will be treated as energies or wavelengths (depending on the *Plot* settings).

notice(*noticeRange*)

Apply a notice channels range to multiple loaded spectra.

Args:

***noticeRange*: String specifying the spectra ranges and/or**
channel ranges to notice.

This follows the same syntax as used in the standard Xspec "notice" command, except that the spectrum range always defaults to ALL spectra.

If the numbers are floats rather than ints, they will be treated as energies or wavelengths (depending on the Plot settings). If the string is 'all', it will notice all channels in all spectra.

`removeDummyrsp()`

Remove all dummy responses, restore original responses (if any).

`show()`

Display information for all loaded spectra.

`property nGroups`

The number of data groups [int].

`property nSpectra`

The number of loaded spectra [int].

3.1.6 FakeitSettings

`class xspec.FakeitSettings(`*response=''*, *arf=''*, *background=''*, *exposure=''*, *correction=''*, *backExposure=''*, *fileName=''*`)`

Fakeit command settings class.

Note: To run *fakeit*, you call the *AllData.fakeit()* function (see the **DataManager** class description). FakeitSettings objects hold the information which gets passed to *AllData.fakeit()*.

The *AllData.fakeit* function will apply 1 FakeitSettings object to every fake spectrum that is to be created. If the user does not explicitly supply their own FakeitSettings objects, *AllData.fakeit* will create its own as necessary, with default settings.

Methods

- __init__

Attributes

- **response**
  Name of detector response file to use for creating the fake spectrum.

  When a fake spectrum is based on a pre-existing spectrum which already has a response, this should be left empty. If a name is given it will be IGNORED. However if the pre-existing spectrum has no response, then this MUST be filled.

  If the fake spectrum is not based on an existing spectrum, this may be filled or left empty. If it is empty, XSPEC will just use its built-in dummy response.

- **arf**
  Name of optional arf to use with the response. This is ignored if no response is given.

- **background**
  Name of optional background file to use when creating the fake spectrum.

  If based on an original spectrum, leave this empty to use the original spectrum's background settings.

- **exposure**
  The fake spectrum exposure time.
- **correction**
  Optional correction norm factor.
- **backExposure**
  Optional background exposure time modifier.

  For *exposure* and *correction*, if left empty fakeit will use the values from the original spectrum, or 1.0 if not based on an original spectrum. Each of these may be entered as a string or float.
- **fileName**
  Optional fake spectrum output file name.

  If left empty, fakeit will create a default file name based on the original spectrum, or the response name if no original spectrum. In the latter case, the default names will also have an incremented suffix to prevent file overwriting.

When writing to a multiple-spectrum output file (OGIP type-2), *exposure*, *correction*, *backExposure*, and *fileName* are applied to the entire file rather than a single spectrum. Therefore entries for these attributes will be IGNORED for all but the first fake spectrum in a type-2 output file.

__init__(*response=''*, *arf=''*, *background=''*, *exposure=''*, *correction=''*, *backExposure=''*, *fileName=''*)

Create a FakeitSettings object.

All arguments are optional, and all may be entered as strings. The *exposure* and *correction* arguments may also be entered as floats.

This can also create a new copy of a pre-existing FakeitSettings object, in which case the pre-existing object should be the only argument entered.

Examples:

```
fs1 = FakeitSettings("resp1.pha", exposure=1500.0)
# Reuse fs1's settings, but with a new fileName attribute:
fs2 = FakeitSettings(fs1)
fs2.fileName = "fakeit2.pha"
# Now generate 2 fake spectra
AllData.fakeit(2, [fs1, fs2])
```

3.1.7 FitManager

class xspec.**FitManager**

Xspec fitting class.

PyXspec automatically creates a single object of this class, named *Fit*.

Methods

- error
- ftest
- goodness
- improve
- perform
- renorm
- show
- steppar
- stepparResults

Attributes ((*) = get-only)

- bayes
- covariance*
- criticalDelta
- delta
- dof*
- method
- nIterations
- nullhyp*
- nVarPars*
- previousGoodness*
- previousGoodnessSims*
- query
- useChainRule
- statMethod
- statTest
- statistic*
- testStatistic*
- weight

__init__()

error(*argString*, *respPar=False*)

Determine confidence intervals of a fit.

Args:

***argString*:**
A string with identical syntax to the standard interactive XSPEC error command.

"[[stopat <ntrial> <toler>] [maximum <redchi>] [<delta fit statistic>] [<model param range>...]]"

where <model param range> ::=[<modelName>:]<first param> -
<last param>

See the XSPEC manual for a more detailed description.

***respPar*: Optional flag. Set this to True if the parameters**
are response parameters. [bool]

The results of the error command are stored in the *error* attributes of the individual Parameter objects.

Examples:

```
# Estimate the 90% confidence ranges for parameters 1-3
Fit.error("1-3")
# Repeat but with delta fit statistic = 9.0, equivalent to the
# 3 sigma range.
Fit.error("9.0")
# Estimate for parameter 3 after setting the number of trials to 20.
# Note that the tolerance field has to be included (or skipped over).
Fit.error("stop 20,,3")
# Perform an error calculation on response parameter 1
Fit.error("1",True)
```

ftest(*chisq2*, *dof2*, *chisq1*, *dof1*)

Calculate the F-statistic and its probability given new and old values of chisq and number of degrees of freedom (DOF).

Args:

chisq2: [float]

dof2: [int]

chisq1: [float]

dof1: [int]

chisq2 and *dof2* should come from a new fit, in which an extra model component was added to (or a frozen parameter thawed from) the model which gave *chisq1* and *dof1*. If the F-test probability is low then it is reasonable to add the extra model component.

> **Warning:** It is not correct to use the F-test statistic to test for the presence of a line (see Protassov et al 2002, ApJ 571, 545).

Returns: The F-test probability [float].

`goodness`(*nRealizations=100*, *sim=False*, *fit='fit'*)

Perform a Monte Carlo calculation of the goodness-of-fit.

Args:

nRealizations: Number of spectra to simulate [int].

sim: flag [bool]

If False (default), all simulations are drawn from the best fit model parameter values. If True, parameters will be drawn from a Gaussian centered on the best fit.

fit: switch indicating fitting will be performed [string]

Only "fit" and "nofit" are accepted. Default is "fit" (fitting will be performed).

`improve`()

Try to find a new minimum.

When *Fit.method* is set to one of the MINUIT algorithms, this will run the MINUIT 'improve' command. This does nothing when Fit.method is set to Levenberg-Marquardt.

`perform`()

Perform a fit.

`renorm`(*setting=None*)

Renormalize the model to minimize statistic with current parameters.

Args:

setting: [string]

If `None`, this will perform an explicit immediate renormalization. Other options determine when renormalization will be performed automatically. They are the following strings:

- 'auto': Renormalize after a model command or parameter change, and at the beginning of a fit.
- 'prefit': Renormalize only at the beginning of a fit.
- 'none': Perform no automatic renormalizations.

`show`()

Show fit information.

`steppar`(*argString*)

Perform a steppar run.

Generate the statistic "surface" for 1 or more parameters.

Args:

argString: [string]

This uses identical syntax to the standard interactive XSPEC steppar command. "<step spec> [<step spec> ...]" where:

<step spec> ::= [<log|nolog>] [<current|best>]
[<modName>:]<param index> <low value> <high value> <# steps>

See the XSPEC manual for a more detailed description of specs.

Examples:

```
# Step parameter 3 from 1.5 to 2.5 in 10 linear steps
Fit.steppar("3 1.5 2.5 10")
# Repeat the above but with logarithmic steps
Fit.steppar("log")
# Step parameter 2 linearly from -.2 to .2 in steps of .02
Fit.steppar("nolog 2 -.2 .2 20")
```

stepparResults(*arg*)

Retrieve values from the most recent steppar run.

Args:
arg:

argument should either be 'statistic', 'delstat', or a parameter specifier. A parameter specifier should be a string of the form:

'[<modName>:]<parNum>'

or simply an integer <parNum>.

Returns the requested values as a list of floats.

property bayes

Turn Bayesian inference on or off [string].

Valid settings are 'on', 'off' (default), or 'cons'. 'cons' turns Bayesian inference on AND gives ALL parameters a constant prior. Priors can be set for parameters individually through the Parameter object's *prior* attribute.

property covariance

The covariance matrix from the most recent fit [tuple of floats] (GET only).

As with standard XSPEC's "tclout covar", this only returns the diagonal and below-diagonal matrix elements.

property criticalDelta

Critical delta for fit statistic convergence [float].

The absolute change in the fit statistic between iterations, less than which the fit is deemed to have converged.

property delta

Set fit delta values to be proportional to the parameter value [float].

Get:
Returns the current proportional setting, or 0.0 if currently using the fixed fit delta values.

Set:
Enter the constant factor which will multiply the parameter value to produce a fit delta. A constant factor of 0.0 or negative will turn off the use of proportional fit deltas.

property dof

The degrees of freedom for the fit [int] (GET only).

property method

The fitting algorithm to use [string].

Choices are: 'leven', 'migrad', 'simplex'. The default is 'leven'.

When setting the method, additional arguments for <nFitIterations> and <fit critical delta> may also be entered. Valid formats for entering multiple arguments are:

```
# Single string
Fit.method = "migrad  100 .05"
# List of strings
Fit.method = ["migrad","100",".05"]
# List of strings and numbers
Fit.method = ["migrad", 100, .05]
```

property nIterations

The maximum number of fit iterations prior to query [int].

property nVarPars

The number of variable parameters for the fit [int] (GET only).

property nullhyp

The null hypothesis probability for the chi-sq fit (GET only).

property previousGoodness

The goodness value from the immediately previous call [float] (GET only).

property previousGoodnessSims

The array of simulation values from the immediately previous goodness calculation [list] (GET only).

property query

The fit query setting [string].

- 'yes': Fit will continue through query.
- 'no' : Fit will end at query.
- 'on' : User will be prompted for "y/n" response.

property statMethod

The type of fit statistic in use [string].

Valid names: 'chi' | 'cstat' | 'lstat' | 'pgstat' | 'pstat' | 'whittle'. To set for individual spectra, add a spectrum number (or range) to the string: ie.

```
>>> Fit.statMethod = "cstat 2"
```

property statTest

The type of test statistic in use [string].

Valid names: 'ad' | 'chi' | 'cvm' | 'ks' | 'pchi' | 'runs'. To set for individual spectra, add a spectrum number (or range) to the string: ie.

```
>>> Fit.statTest = "ad 2"
```

property statistic

Fit statistic value from the most recent fit [float] (GET only).

This returns the total fit statistic. An individual spectrum's contribution to the total is stored in the Spectrum object's *statistic* attribute.

property testStatistic

Test statistic value from the most recent fit [float] (GET only).

property useChainRule

The fit useChainRule setting [string].

- 'yes' or 'on': Fast second derivative calculation using chain rule.
- 'no' or 'off': Slow second derivative calculation.

property weight

Change the weighting function used in the calculation of chi-sq [string].

Available functions: 'standard', 'gehrels', 'churazov', 'model'

3.1.8 Model

class xspec.**Model**(*exprString*, *modName=''*, *sourceNum=1*, *setPars=None*)

Xspec model class.

Methods

- __init__
- __call__ (the '()' operator)
- energies
- folded
- setPars
- show
- showList
- untie
- values

Attributes (all are get-only)

- **expression**
  The model expression string, using full component names.
- **name**
  The model name, optional in Xspec. This is an empty string for un-named models.
- **<components>**
  Model includes an attribute of type **Component** for every Xspec component in the model. The attribute name is the same as the full name of the Xspec component (ie. m=Model("po") produces an m.powerlaw attribute).
- **componentNames**
  List of component name strings.

- **flux**
  A tuple containing the results of the most recent flux calculation for this model.

  The tuple values are: (*value*, *errLow*, *errHigh* (in ergs/cm^2), *value*, *errLow*, *errHigh* (in photons)). This will be filled in after an *AllModels.calcFlux()* call ONLY when no spectra are loaded. Otherwise results are stored in the Spectrum objects.

- **lumin**
  Same as *flux* but for luminosity calculations.

  The tuple values are: (*value*, *errLow*, *errHigh* (in 10^44 ergs/s), *value*, *errLow*, *errHigh* (in photons)).

- **nParameters**
  Number of parameters in Model object [int].

- **startParIndex**
  Global index of the first parameter in this Model object [int].

`__call__`(*parIdx*)

Get a Parameter object from the Model.

Args:
parIdx: The parameter index number.

Regardless of the data group to which the Model object belongs, its parameters are numbered from 1 to *nParameters*.

Returns the specified Parameter object.

`__init__`(*exprString*, *modName=''*, *sourceNum=1*, *setPars=None*)

Model constructor.

New model is automatically added to the *AllModels* container, with one Model object constructed (internally) for each data group to which the model applies. This function returns the Model object corresponding to the lowest numbered data group.

Args:
exprString: The model expression string.

Component names may be abbreviated.

modName: Optional name assigned to model.

Any whitespace in string will be removed. This is required if souce number is > 1.

sourceNum: Optional integer for model's source number.

setPars: Optional initial values for the model's parameters.

These may be sent in a tuple, list, or dictionary (or as a single float or string if only setting the first parameter). Examples:

```
# Create a model with all default parameter settings:
m1 = Model("gauss")

# Create phabs*powerlaw and initialize pars 1 and 3 to
#   something other than their default values.
m2 = Model("phabs*po", setPars={1:5.5, 3:".18,,.01,.02"})

# Create another model named 'b', and reset par 2 to 5.0:
m3 = Model("phabs*bbody", "b", setPars={2:5.0})
```

If any mistakes are made with the optional *setPars* parameter arguments, the model will be created using all default values.

You can always reset the parameters later with the *Model.setPars()* method, or directly through the Parameter object's *values* attribute.

__setattr__(*attrName*, *value*)

Implement setattr(self, name, value).

energies(*spectrumIndex*)

Get the Model object's energies array for a given spectrum.

Args:

spectrumIndex: The spectrum index number.

If this is 0, it will return the energies array used by the default dummy response.

Returns a list of energy array elements, the size will be 1 larger than the corresponding flux array.

This will return the energies array as specified by the *AllModels.setEnergies* function if that has been used to override the response energies array.

folded(*spectrumIndex*)

Get the Model object's folded flux array for a given spectrum.

Args:

spectrumIndex: The spectrum index number.

This number should be 0 if model is not presently applied to any spectra (ie. in the "off" state).

Returns a list of folded flux array elements.

setPars(**parVals*)

Change the value of multiple parameters in a single function call.

This is a quick way to change multiple parameter values at a time since only a SINGLE model recalculation will be performed at the end. In contrast, when parameter values are changed through the individual parameter objects, the model is recalculated after EACH parameter change. (See also *AllModels.setPars()*, for changing multiple parameters belonging to multiple model objects.)

Args:

parVals: An arbitrary number of parameter values.

These may be listed singly (as floats or strings), or collected into tuple, list or dictionary containers. Dictionaries must be used if parameters are not in consecutive order, in which case the parameter index number is the dictionary key.

Examples: Assume we have a model object *m1* with 5 parameters.

Simplest case: change only the parameter values (and not the auxiliary values, 'sigma', 'min', 'bot', etc.), and change them in consecutive order:

```
# Pass in 1 or more floats
m1.setPars(5.5, 7.83, 4.1e2)  # changes pars 1-3
m1.setPars(2.0, 1.3e-5, -.05, 6.34, 9.2)  # changes all 5 pars
```

Still changing only the parameter values, but skipping over some:

```
m1.setPars(.02, 4.4, {5:3.2e5})  # changes pars 1-2, 5
m1.setPars({2:3.0, 4:-1.2})  # changes pars 2, 4
m1.setPars({2:1.8}, 9.3, 5.32)  # changes pars 2, 3, 4
```

Now also change the auxiliary values for some of the parameters. Pass in a STRING containing "<val>,<sigma>,<min>,<bottom>,<top>, <max>" This uses the same syntax as Standard XSPEC's "newpar" command. Aux values can be skipped by using multiple commas:

```
# This sets a new <val>, <sigma>, and <max> for parameter 1, and
# a new <val> of 5.3 for parameter 2.
m1.setPars(".3,.01,,,,100", 5.3)

# This sets all new auxiliary values for parameter 3.
m1.setPars({3:".8 -.01 1e-4 1e-3 1e5 1e6"})
```

show()

Display information for a single Model object.

static showList()

Show the list of all available XSPEC model components.

untie()

Remove links for all parameters in Model object

values(*spectrumIndex*)

Get the Model object's values array for a given spectrum.

Args:

spectrumIndex: The spectrum index number.

This number should be 0 if model is not presently applied to any spectra (ie. in the "off" state).

Returns the values array as a list.

3.1.9 ModelManager

class xspec.**ModelManager**

Models container.

PyXspec automatically creates a single object of this class, named *AllModels*.

Methods

- __call__ (the '()' operator)
- __iadd__ (the '+=' operator)
- __isub__ (the '-=' operator)
- addPyMod
- calcFlux
- calcLumin
- clear
- eqwidth
- initpackage
- lmod
- mdefine
- setActive
- setEnergies
- setInactive
- setPars
- show
- simpars
- systematicSingleModel
- setSystematicSingleModel
- tclLoad

Attributes

- sources (get-only)
- systematic

__call__(*groupNum*, *modName=''*)

Get Model objects from the AllModels container.

Args:

groupNum: The data group number to which the Model object corresponds.

modName: Optional string containing the Model's name (if any).

Returns the Model object.

__iadd__(*modelInfo*)

Define a new model and add it to the *AllModels* container.

This operation is equivalent to the Model class constructor, except that it does not return a Model object.

Args:

***modelInfo*:**

A string containing the model expression (component names may be abbreviated). The model will be unnamed and assigned to source number = 1.

OR

If supplying a model name and a source number, this should be a tuple with:

modelInfo[0] = model expression string

modelInfo[1] = model name string

modelInfo[2] = source number

__init__()

__isub__(*modName*)

Remove all copies of the given model from the AllModels container.

Args:

***modName*:**

The name of the model to be removed, or an empty string if the model has no name. If set to "*", this will behave like the *clear()* function and remove all models.

addPyMod(*func*, *parInfo*, *compType*, *calcsErrors=False*, *spectrumDependent=False*)

Add a user-defined Python model function to XSPEC's models library.

This provides a way to add to XSPEC local models written in Python. It performs the same role as the combination of *initpackage/lmod* commands do for C/C++/Fortran local models. The first 3 arguments (*func*, *parInfo*, and *compType*) are mandatory.

Args:

func: The user-defined model function (Python type = 'function').

Function must define at least 3 arguments for energies, parameters, and flux.

A optional fourth argument may be added if your model calculates flux errors, and a fifth if your model requires that XSPEC pass it the spectrum number.

parInfo: A tuple of strings.

One string for each parameter your model requires. The format of these strings is identicalto what is placed in a 'model.dat' file (see Appendix C of the XSPEC manual).

compType: A string telling XSPEC the type of your model.

Currently allowed types: 'add', 'mul', 'con'

calcsErrors: OPTIONAL bool flag.

If your model function also calculates model errors, set this to True.

spectrumDependent: OPTIONAL bool flag.

Set this to True only if your model function has an explicit dependence on the spectrum.

Example usage: A local additive model written in Python, named 'myModel',
which takes parameters named 'par1' and 'par2'.

```
def myModel(engs, pars, flux):
   #    [... your model code, fill in
   #      flux array based on input
   #      engs and pars arrays ...]

myModelParInfo=("par1  "" 2.0  -10.0  -9.0  9.0  10.0  0.01",
                "par2  keV  1e-3 1e-5  1e-5  100. 200. .01" )

AllModels.addPyMod(myModel, myModelParInfo, 'add')
```

calcFlux(*cmdStr*)

Calculate the model flux for a given energy range.

Args:
cmdStr: string

Should contain the energy limit values and optional error specifiers. This follows the same syntax rules as the standard XSPEC 'flux' command.

The flux will be calculated for all loaded spectra, and the results will be stored in the Spectrum objects' *flux* attribute. If no spectra are loaded, the flux will be stored in the Model objects' *flux* attribute.

calcLumin(*cmdStr*)

Calculate the model luminosity for a given energy range and redshift.

Args:
cmdStr: string

Should contain the energy limit values and optional error specifiers. This follows the same syntax rules as the standard XSPEC 'lumin' command.

The lumin will be calculated for all loaded spectra, and the results will be stored in the Spectrum objects' *lumin* attribute. If no spectra are loaded, the flux will be stored in the Model objects' *lumin* attribute.

clear()

Remove all models.

eqwidth(*component*, *rangeFrac=None*, *err=False*, *number=None*, *level=None*)

Calculate the equivalent width of a model component.

Please see the Standard XSPEC Manual for a discussion on how the eqwidth of a component is calculated.

Args:

***component*:**
An integer specifying the model component number for which to calculate the eqwidth (left-most component is 1). If the component belongs to a NAMED model, then this must be a **string** of the form "<modelName>:<compNumber>".

rangeFrac: Determines the energy range for the continuum calculation.

Range will be from E(1-<rangeFrac>) to E(1+<rangeFrac>) where E is the location of the peak of the photon spectrum. The initial default rangeFrac is 0.05. Setting this will change the future default value.

err: Bool flag.

If set to True, errors will be estimated on the equivalent width calculation. This will also require the setting of the "number" and "level" arguments.

number: Only set this if *err* = True.

This determines the number of sets of randomized parameter values to draw to make the error estimation. [int]

level: Only set this if *err* = True.

The error algorithm will order the equivalent widths of the *number* sets of parameter values, and the central *level* percent will determine the error range. [float]

The results of the most recent eqwidth calculation are stored as attributes of the currently loaded Spectrum objects.

identify(*energy=None*, *delta=None*, *redshift=None*, *lineList=None*, *tplasma=None*, *emiss=None*)

Identify spectral lines.

List possible lines in the specified energy range.

This function also returns the list of lines as a string.

Args (all are optional):

energy: The center of the energy range (energy +/- delta) to search.

This should be entered in keV unless working with plot x-axis settings in wavelength mode, in which case enter as wavelengths in Angstroms. [float]

delta: Specifies the spread of energies (energy +/- delta). [float]

redshift: [float]

lineList: The list of lines to be searched [string].

Options are 'bearden', 'mekal', and 'apec'.

***tplasma*: For 'apec' list only, temperature of the plasma in keV.**
[float]

***emiss*: For 'apec' list only, minimum emissivity of lines to be**
shown. [float]

Examples:

```
# All values are optional, use keywords to enter values
# non-consecutively.  Unspecified values revert to the
# current defaults.

# Search the 'mekal' list between .4 and .6 keV
AllModels.identify(.5, .1, lineList='mekal')

# Search the 'apec' list with additional temperature and
# minimum emissivity values, and save the results in string 's'.
s = AllModels.identify(1.0, .1, .0, 'apec', 5.0, 1.0e-18)
```

Initial defaults: energy = 1.0, delta = .01, redshift = 0.0, lineList = 'apec', tplasma = 1.0, emiss = 1.0e-19. These defaults will be modified by each new entry.

initpackage(*packageName*, *modDescrFile*, *dirPath=None*, *udmget=False*)

Initialize a package of local models.

Use this method to compile your local model source code and build a library, which can then be loaded into XSPEC with the *lmod* method.

Args:

packageName: The name of the model package [string].

The name should be all lower-case and contain NO numerals or spaces. The local models library file will be based upon this name, and this is also the name you will use when loading the library with the *lmod* method.

modDescrFile: Name of your local model description file [string].

This file is typically named 'lmodel.dat', but you're free to name it something else.

dirPath: Optional directory path to your local models [string].

This may be an absolute or relative path. If you don't enter this argument, XSPEC will look in the directory given by the LOCAL_MODEL_DIRECTORY in your Xspec.init start-up file.

udmget: Optional bool flag.

Set this True only when your models need to call XSPEC's udmget function. Udmget is a function for allocating dynamic memory in Fortran routines, and is no longer used within XSPEC itself. If this flag is set to 'True', initpackage will copy the necessary files and build the udmget function within your local models directory.

lmod(*packageName*, *dirPath=None*)

Load a local models library.

Args:

packageName: The name of the model package to be loaded.

This is the same name that is the first argument in the *initpackage* command.

dirPath: An optional string argument.

This should specify the (absolute or relative) path to the local model directory. If this argument is not entered, XSPEC will look in the directory given by the LOCAL_MODEL_DIRECTORY in the Xspec.init start-up file.

mdefine(*commandStr=None*)

Define a simple model using an arithmetic expression.

Args:

commandStr: The full mdefine expression string.

If this is blank or None, PyXspec will simply display the list of currently defined mdefine components.

Otherwise this should be a string of the form: <name> [<expression>[: [<type>] [<emin> <emax>]]] where:

<name> = the name of the model component

<expression> = a string of arithmetic expressions
defining the model.

<type> = type of the component (ie. add, mul, con)

<emin> <emax> = optional min and max energy values
for the model.

If the string is just "<name>", it will display information of the previously defined component with name = <name> (if any).

Doing "<name> :" will delete a previously defined component with name = <name>.

See the 'mdefine' command in the standard XSPEC manual for more information.

Examples:

```
AllModels.mdefine('dplaw e**p1 + f*E**p2')
AllModels.mdefine('test1 exp(-a*e) : mul')
AllModels.mdefine('test2 a*e**2 ')
# Show all currently loaded mdefine components
AllModels.mdefine()
# Delete component named 'test2'
AllModels.mdefine('test2 :')
```

setActive(*modl_nm*)

Sets the model specified with modl_nm to active.

Args:
modl_nm: The name of the model to be activated.

setEnergies(*arg1*, *arg2=None*)

Specify new energy binning for model fluxes.

Supply an energy binning array to be used in model evalutations in place of the associated response energies, or add an extension to the response energies.

Args:
arg1: A string containing either:

"<range specifier> [<additional range specifiers>...]"

"<name of input ascii file>"

"extend" [This option also uses *arg2*]

"reset"

where the first <range specifier> ::= <lowE> <highE> <nBins> log|lin

<additional range specifier> ::= <highE> <nBins> log|lin

This uses the same syntax as standard XSPEC's 'energies' command. Values can be delimited by spaces or commas.

arg2: String only needed when *arg1* is "extend".

This requires an extension specifier string of the form: "low|high <energy> <nBins> log|lin"

All energies are in keV. Multiple ranges may be specified to allow for varied binning in different segments of the array, but note that no gaps are allowed in the overall array. Therefore only the first range specifier accepts a <lowE> parameter. Additional ranges will automatically begin at the <highE> value of the previous range.

With the "extend" option, the specifier string supplied to *arg2* will extend the existing response energy array by an additional <nBins> to the new <energy>, in either the high or low direction.

Once an energy array is specified, it will apply to all models and will be used in place of any response energy array (from actual or dummy responses) for calculating and binning the model flux. It will also

apply to any models that are created after it is specified. To turn off this behavior and return all models back to using their response energies, set *arg1* to "reset".

arg1 can also be the name of an ascii text file containing a custom energy array. To see the proper file format, and for more details in general about the 'energies' command, please see the standard XSPEC manual.

Examples:

```
# Create an array of 1000 logarithmic-spaced bins, from .1 to 50. keV
AllModels.setEnergies(".1 50. 1000 log")
# Change it to 500 bins
AllModels.setEnergies(",,500")
# Now restore original response energies, but with an extension of the
#   high end to 75.0 keV with 100 additional linear bins.
AllModels.setEnergies("extend","high,75.,100 lin")
# Return to using original response energies with no extensions.
AllModels.setEnergies("reset")
```

setInactive(*modl_nm*)

Sets the model specified by modl_nm to inactive.

Args:

modl_nm: The name of the model to be set to inactivate.

setPars(**args*)

Change the value of multiple parameters from multiple
model objects with a single function call.

This is a quick way to change multiple parameter values at a time since only a SINGLE recalculation will be performed at the end. In contrast, when parameter values are changed through the individual parameter objects, the model is recalculated after EACH parameter change. (If all the parameters belong to a single model object, you can also use the *Model.setPars()* function.)

args: An arbitrary number model objects and parameter values.

The first argument must be model object, followed by one or more of its new parameter values. Additional groups of model objects and parameter values may follow.

The parameter values follow the same syntax rules as with the single *Model.setPars()* function. They can be listed singly (as floats or strings), or collected into tuple, list, or dictionary containers. Dictionaries must be used when parameters are not in consecutive order, in which case the parameter index number is the dictionary key.

Parameter indices are local to each model object. That is, they are always numbered from 1 to N where N is the number of parameters in the model object.

Examples:

```
# Assume we've already assigned a 3 parameter model to 2 data groups:
m1 = AllModels(1)
m2 = AllModels(2)

# Various ways of changing parameters in consecutive order.

# This changes pars 1-2 in m1 and 1-3 in m2:
AllModels.setPars(m1, .4, "1.3 -.01", m2, "5.3 ,,3.0e-4", 2.2, 1.9)
#   ...and these 2 examples do the exact same thing as above:
valList = [.4, "1.3 -.01"]
```

(continues on next page)

(continued from previous page)

```
valTuple = ("5.3 ,,3.0e-4", 2.2, 1.9)
AllModels.setPars(m1, valList, m2, valTuple)
AllModels.setPars(m1, valList, m2,"5.3 ,,3.0e-4", [2.2, 1.9])

# Parameters in non-consecutive order, must use Python
#   dictionaries:

# Change parameter 2 in m1, parameter 1 and 3 in m2:
AllModels.setPars(m1, {2:8.3}, m2, {1:0.99, 3:"7.15 -.01"})
# ...same thing as above:
AllModels.setPars(m1, {2:8.3}, m2, 0.99, {3:"7.15 -.01"})

# Note that identical syntax is used for model objects belonging
# to different sources.  All of the above examples are still valid
# had we obtained m1 and m2 like this:

m1 = Model("phabs*pow", "firstMod", 1)
m2 = Model("gauss", "secondMod", 2)
```

setSystematicSingleModel(*modelName*, *value*)

Set a fractional systematic error for a specific model.

The *AllModels.systematic* attribute is for setting the default systematic error which applies to all models. Use this method to set a value that only applies to a specific model, overriding the default value. If the model is un-named, enter 'unnamed' for the *modelName* argument.

show(*parIDs=None*)

Show all or a subset of Xspec model parameters.

Args:

***parIDs*:**
An optional string specifying a range of parameters as with Xspec's "show parameter" function. If no string is supplied, this will show all parameters in all models.

simpars()

Create a list of simulated parameter values.

Values are drawn from a multivariate normal distribution based on the covariance matrix from the last fit, or from Monte Carlo Markov chains if they are loaded. This method is identical to doing 'tclout simpars' in standard XSPEC.

Returns a tuple of the simulated parameter values.

systematicSingleModel(*modelName*)

Get the systematic error setting for a specific model.

This will be the same as the default value stored in the *AllModels.systematic* attribute unless this model was given its own value with the *setSystematicSingleModel* method. To get the value of an un-named model, enter 'unnamed'.

tclLoad(*fullLibPath*)

Load a local model library by calling Tcl's 'load' command.

This by-passes *lmod* (with its pkgIndex.tcl requirements) and allows the user to load a local model by directly calling Tcl's lower level 'load' command. May also be useful for error diagnostics when *lmod* has failed.

Args:
fullLibPath: The full local model library path and filename.

property sources

A dictionary containing the currently active <source number>:<model name> assignments.

If the model has no name, <model name> will be an empty string. (GET only)

property systematic

The fractional model systematic error.

This will be added in quadrature to the error on the data when evaluating chi-squared. This sets the default value that will be applied to all models. To override this setting for a specific model, use the *setSystematicSingleModel* method.

3.1.10 Parameter

class xspec.**Parameter**(*parName*, *parStrategy*)

Model or response parameter class.

Methods

- untie

Attributes ((*) = get-only)

- error*
- frozen
- index*
- link
- name*
- prior
- sigma*
- unit*
- values

untie()

Remove parameter link (if any)

property error

A tuple containing the results of the most recent fit *error* command performed on the parameter (GET only). The tuple values are: (<error low bound>, <error high bound>, <error status code string>)

property frozen

Bool, if True then parameter is frozen.

property index

Position of the parameter within the Model object.

(The first parameter has *index* = 1) Note that this is the same value that would be used to obtain a Parameter object from its Model, ie: par = mod(<index>) (GET only).

property link

Parameter link expression string (empty if not linked).

property name

Name of Parameter (GET only).

property prior

A tuple containing the settings for the prior used when Bayesian inference is turned on.

Get: Returns a tuple containing:
(<priorType>, <optional hyperparameters>)

Set with:
string: <priorType>

or tuple: (<priorType>, <optional hyperparameters>)

Valid priorTypes are 'cons', 'exp', 'jeffreys', 'gauss'. Hyperparameters should be entered as floats.

property sigma

The Parameter fit sigma (-1.0 when not applicable) (GET only).

property unit

An optional string for the parameter's units (GET only).

property values

List of value floats [val,delta,min,bot,top,max].

This may be set with:
string: x.values = "3.2,,,,1e2, 1e3"

single float: x.values = 4.1 (sets 'val' only)

tuple: x.values = 8.2,.02, -10.

list: x.values = [8.2,.02, -10.]

Note that Tuple and List input do not allow the use of consecutive commas for argument spacing.

3.1.11 PlotManager

class xspec.**PlotManager**(*deviceStr*)

Xspec plotting class.

PyXspec automatically creates a single object of this class, named *Plot*.

Methods

- __call__ (the '()' operator)
- addCommand
- addComp
- backgroundVals
- contourLevels
- delCommand
- iplot
- labels
- model
- nAddComps
- noID
- setGroup
- setID
- setRebin
- show
- x
- xErr
- y
- yErr
- z

Attributes

- add
- area
- background
- commands
- device
- perHz

- redshift
- splashPage
- xAxis
- xLog
- yLog

__call__(**panes*)

Display the plot.

Input 1 or more plot command strings.

Examples:

```
# Single Plots:
Plot("data")
Plot("model")
Plot("ufspec")

# Multiple Plots (or single plots taking additional arguments):
Plot("data","model","resid")
Plot("data model resid")
Plot("data,model,resid")
Plot("data","model m1")  # Plots data and a model named "m1".

# To repeat a plot using the previously entered arguments, simply do:
Plot()
```

__init__(*deviceStr*)

addCommand(*cmd*)

Add a plot command [string] to the end of the plot commands list.

addComp(*addCompNum=1*, *plotGroup=1*, *plotWindow=1*)

Return a list of Y-coordinates for a particular add component of a model

The individual add component arrays of a model are generated in 'data' and 'model' plots when Plot.add = True. Add components are numbered from 1 to N where N is the total number of add components from all models in the plot group. (A data plot may have multiple models associated with a spectrum.) See also the *Plot.nAddComps()* method for obtaining the value N.

This call is valid only if the add component belongs to a model which has multiple add components.

backgroundVals(*plotGroup=1*, *plotWindow=1*)

Return a list of background data values for a plot group and plot window

Background value arrays only exist for data plots when the *Plot.background* flag is set to True.

contourLevels()

Return a list of the values of the drawn levels in a contour plot.

This method should only be called when the most recent Plot() call produces a 2D contour plot. Otherwise this will raise an error.

delCommand(*num*)

Remove a plot command by (1-based) number [int].

This is intended for removal of single commands. To remove ALL commands, set the *Plot.commands* attribute to an empty tuple, ie:

```
>>> Plot.commands = ()
```

iplot(**panes*)

Display the plot and leave it in interactive plotting mode.

This function takes the same arguments and syntax as when displaying plots in the regular mode (through Plot's *__call__* method). Examples:

```
Plot.iplot("data")   # 1-panel data plot
Plot.iplot("data model")    # 2-panel data and model
Plot.iplot()                # Repeats the previous plot.
```

labels(*plotWindow=1*)

Get the X, Y, and Title labels for the specified plot window.

The plot window argument is optional and defaults to 1.

Returns a tuple of strings (X_label, Y_label, Title). If any labels are missing the corresponding string will be empty. Special characters (ie. sub/superscripts, Greek, etc.) are produced in TeX format for compatibility with matplotlib.

model(*plotGroup=1*, *plotWindow=1*)

Return a list of Y-coordinate model values for a plot group and plot window

nAddComps(*plotGroup=1*, *plotWindow=1*)

Return the number of add component plots for a given plot group.

For plots showing the contributions from the individual additive components of a model, this returns the number of add components displayed for a particular plot group in a plot window. This may be helpful for determining the valid range of the *addCompNum* input argument to *Plot.addComps()*.

This returns 0 if individual add components are not plotted, either because add components are not relevant to the type of plot or because the model(s) do not consist of multiple add components.

noID()

Turn off the plotting of line IDs.

setGroup(*groupStr*)

Define a range of spectra to be in the same plot group.

Input argument is a string specifying one or more ranges, delimited by commas and/or spaces. Examples:

```
# Spectra 1-3 in plot group 1, 4-6 in group 2.
Plot.setGroup("1-3  4-6")

# Spectra 1, 2, and 4 are each now in their own group.
Plot.setGroup("1,2 4")

# All spectra are in a single plot group.
Plot.setGroup("1-**")

# If input argument is Python's 'None' variable, all
# plot grouping will be removed.
Plot.setGroup(None)
```

setID(*temperature=None*, *emissivity=None*, *redshift=None*)

Switch on plotting of line IDs.

All input arguments are floats and are optional. If they are omitted they will retain their previous values.

- *temperature*: Selects the temperature of the APEC line list.
- *emissitivity*: Only lines with emissivities above this setting will be displayed.
- *redshift*: Line display will be redshifted by this amount.

To turn off plotting of line IDs, use the *noID()* function.

setRebin(*minSig=None*, *maxBins=None*, *groupNum=None*, *errType=None*)

Define characteristics used in rebinning the data (for plotting purposes ONLY).

All input arguments are optional. If they are omitted they will retain their previous values.

- *minSig*: Bins will be combined until this minimum significance is reached (in units of sigma). [float]
- *maxBins*: The maximum number of bins to combine in attempt to reach *minSig*. [int]
- *groupNum*: The plot group number to which this setting applies. If number is negative, it will apply to ALL plot groups. [int]
- *errType*: Specifies how to calculate the error bars on the new bins. Valid entries are "quad", "sqrt", "poiss-1","poiss-2", "poiss-3". [string] See the "setplot" description in the XSPEC manual for more information.

show()

Display current plot settings

x(*plotGroup=1*, *plotWindow=1*)

Return a list of X-coordinate data values for a plot group and plot window

xErr(*plotGroup=1*, *plotWindow=1*)

Return a list of X-coordinate errors for a plot group and plot window

y(*plotGroup=1*, *plotWindow=1*)

Return a list of Y-coordinate data values for a plot group and plot window

yErr(*plotGroup=1*, *plotWindow=1*)

Return a list of Y-coordinate errors for a plot group and plot window

z()

Return a 2D list of the grid values for a 2D contour plot

The returned format is particularly useful for insertion into matplotlib.pyplot's contour function. If the current plot is not a 2D contour plot, this returns an error.

property add

Turn on/off the display of individual additive components [bool].

property area

Toggle displaying the data divided by the response effective area for each channel [bool].

property background

Toggle displaying the background spectrum (if any) when plotting data [bool].

property commands

Custom plot commands to be appended to Xspec-generated commands.

Get: Returns a tuple of the currently entered command strings.

Set: Replaces all commands with the new tuple of strings.

To remove ALL plot commands, set to an empty tuple, ie:

```
>>> Plot.commands = ()
```

For inserting and deleting individual commands, use *addCommand* and *delCommand* functions.

property device

The plotting device name [string].

property perHz

Toggle displaying Y-axis units per Hz when using wavelength units for X-axis [bool].

property redshift

Apply a redshift to the X-axis energy or wavelength values [float].

This will multiply X-axis energies by a factor of (1+z) to allow for viewing in the source frame. Y-axis values will be equally affected in plots which are normalized by energy or wavelength. Note that this is not connected in any way to redshift parameters in the model (or the setplot id redshift parameter) and should only be used for illustrative purposes.

property splashPage

When set to False, the usual XSPEC version and build data information will not be printed to the screen when the first plot window is initially opened [bool].

property xAxis

X-Axis Units [string].

Valid options are: "channel", (energies) "keV", "MeV", "GeV", "Hz", (wavelengths) "angstrom", "cm", "micron", nm"

These are case-insensitive and may be abbreviated.

This setting also affects the ignore/notice range interpretation.

property xLog

Set the x-axis to logarithmic or linear for energy or wavelength plots [bool].

xLog has no effect on plots in channel space. *xLog* and *yLog* will not work for model-related plots (eg. *model*, *ufspec*, and their variants) as their axes are always set to log scale.

property yLog

See xLog.

3.1.12 Response

class xspec.**Response**(*parent*, *respTuple*)

Detector response class.

Methods

- setPars
- show

Attributes (get-only unless stated otherwise)

- **arf**
  Get/Set the arf filename string. Enter `None` or empty string to remove an existing arf.
- **chanEnergies**
  Tuple of floats, the detector channel energies in keV. These are the energies normally stored in the EBOUNDS extension.
- **energies**
  Tuple of floats, the photon energies in keV. These are the energies normally stored in the MATRIX extension.
- **gain**
  A response model object (class RModel) for applying a shift in response file gain.

  (Also see *Response.setPars()* for setting multiple gain parameters at a time.)

  When gain is turned ON, it creates two variable fit Parameter object members:

  ```
  >>> gain.slope   # (default = 1.0)
  >>> gain.offset  # (default = 0.0)
  ```

  To turn gain ON simply assign a value to EITHER parameter, ie.:

  ```
  >>> gain.slope = 1.05
  ```

  This automatically also creates a *gain.offset* parameter with default value 0.0, which you may want to re-adjust. Examples:

  ```
  >>> gain.offset = .02
  >>> gain.offset.values = ".015,.001,,,,0.1"
  ```

  slope and *offset* are of the same type as regular model parameters, and therefore have the same functions, attributes, and syntax rules for setting values. (See the Parameter class help for more details.)

  To turn gain OFF, call its *off()* method:

  ```
  >>> gain.off()
  ```

  gain.off() restores the response to its original state, and renders the *slope* and *offset* parameters inaccessible.
- **rmf**
  The response file name string.
- **sourceNumber**
  The 1-based source number for which the response is assigned. This is normally always 1 unless multiple sources are loaded for multiple-model evaluation.

setPars(**seqPars*)

Set multiple response parameters with a single function call.

Similar to the *Model.setPars()* function, this allows multiple response parameters to be changed with just a SINGLE recalculation performed at the end.

Args:

***seqPars*:**
An arbitrary number of CONSECUTIVE parameter values to be matched 1-to-1 with the response model's parameters.

Currently just 1 response model is available (*gain*), which has 2 response parameters (*slope* and *offset*).

Examples:

```
s = Spectrum("file1")
resp = s.response

# 'gain' is off by default and response parameters don't yet exist.
#   The following call automatically turns 'gain' on and creates
#   both 'slope' and 'offset' parameters even though it is only
#   assigning to 'slope'.  'offset' will retain its default value
#   of 0.0.
resp.setPars(1.05)  # Equivalent to doing: resp.gain.slope = 1.05

# This is equivalent to: resp.gain.slope = .995
#                        resp.gain.offset = .08
#   except that the recalculation is only performed at the end
#   rather than after each parameter is changed:
resp.setPars(.995, .08)

# Can also assign auxiliary values by passing 1 or 2 string
#   arguments.
resp.setPars("1.1,,.02,.02,1.8,1.8","-.05,,-2,-2")

# Remove gain and restore response to original state:
resp.gain.off()
```

show()

Display response information including (optional) response parameters.

3.1.13 RModel

class xspec.**RModel**(*resp*, *parNames*, *rmodName*)

Response Model class.

Response models are functions which act upon the detector RMF. XSPEC currently has just one response model: *gain*, which is a built-in attribute of the Response class. RModel objects are not intended for stand-alone creation: its __init__ function should be considered private.

Attributes

- **<parameters>**
  When RModel is ON, it contains an attribute of type Parameter for every parameter in the model. An RModel is turned ON by a 'set' operation on ANY of its parameters. For example with the *gain* RModel:

```
>>> resp.gain.offset = .03
```

automatically creates *offset* AND *slope* parameters if they don't already exist (*slope* would be initialized to its default value of 1.0). The shift is then applied immediately to the Response object *resp*.

When RModel is OFF (see the *RModel.off()* method), the parameters are not accessible.

- **isOn**
  Boolean flag showing the On/Off status of the RModel (get only).
- **parameterNames**
  List of the response model's parameter names (get only).

off()

Remove response parameters and turn the model OFF.

The Response is restored to its original state.

property isOn

On/Off indicator for RModel object.

3.1.14 Spectrum

class xspec.**Spectrum**(*dataFile*, *backFile='USE_DEFAULT'*, *respFile='USE_DEFAULT'*, *arfFile='USE_DEFAULT'*)

Spectral data class.

Methods

- __init__
- dummyrsp
- fileinfo
- ignore
- ignoredString
- notice
- noticedString
- show

Attributes ((*) = get-only)

- areaScale*
- background
- backScale*
- cornorm
- correction
- dataGroup*
- energies*
- eqwidth*
- exposure*
- fileName*
- flux*
- ignored*
- index*
- isPoisson*
- lumin*
- multiresponse
- noticed*
- rate*
- response
- responsesUsed*
- statistic*
- values*
- variance*
- xflt*

__init__(*dataFile*, *backFile='USE_DEFAULT'*, *respFile='USE_DEFAULT'*, *arfFile='USE_DEFAULT'*)

Construct a Spectrum object.

Read in a spectrum and any associated background, response and arf files. The only required argument is the dataFile name. By default, it will also read in all associated files as given by the keywords stored in the data file.

You may override the default associated files with additional optional arguments. These may also be changed at a later time with Spectrum's *background* and *response* attributes (and Response's *arf* attribute).

Note that PyXspec will always first try to load the associated files as given by the dataFile's internal keywords. Then it replaces them with the specified optional argument file names.

The Spectrum object is automatically added to the *AllData* container.

Args:
dataFile: Spectral data filename [string].

Optional Args:
backFile: Background filename [string].

respFile: Response filename [string].

arfFile: Arf filename [string].

These may be used to override the filenames given by the keywords stored in the spectral data file. If they are blank or the Python `None` variable, the corresponding default file will be removed with no replacement,

dummyrsp(*lowE=None*, *highE=None*, *nBins=None*, *scaleType=None*, *chanOffset=None*, *chanWidth=None*, *sourceNum=1*)

Create a dummy response for this spectrum only.

Args (all are optional):
lowE: Input response energy lower bound, in keV. [float]

highE: Input response energy higher bound, in keV. [float]

nBins: Number of bins into which the energy range is divided. [int]

scaleType: 'log' or 'lin'. [string]

chanOffset: Starting value of dummy channel energies. [float]

chanWidth: Energy width of the channel bins. [float]

If *chanWidth* is set to 0, the dummy response can only be used for evaluating model arrays, and not for fitting to spectra.

sourceNum: Optional source number for the dummy response. [int]

Examples:

```
# All values are optional, use keywords to enter values
# non-consecutively.  Unspecified values revert to the
# current defaults.
s = Spectrum("dataFile.pha")
s.dummyrsp(.3, 30., 100, chanWidth=.5)
s.dummyrsp(highE = 50., sourceNum = 2)
s.dummyrsp(.1,10.,100,"lin",.0, 1.0, 1)
```

Initial defaults: lowE = .1, highE = 50., nBins = 50, scaleType = "log" chanOffset = .0, chanWidth = .0, sourceNum = 1

The defaults for *lowE*, *highE*, *nBins*, *scaleType*, and *chanOffset* will be modified for each explicit new entry. *chanWidth* always defaults to 0 and *sourceNum* always defaults to 1.

To remove the spectrum's dummy response(s) and restore actual responses (if any), call *AllData.removeDummyrsp()*.

fileinfo(*keyword*)

Return the value of a particular keyword in the SPECTRUM extension.

Args:
keyword: Name of the keyword

The value is returned as a string.

ignore(*ignoreRange*)

Ignore a range of the spectrum by channels or energy/wavelengths.

Args:
ignoreRange: String specifying the channel range to ignore.

This follows the same syntax as used in the standard Xspec "ignore" command. If the numbers are floats rather than ints, they will be treated as energies or wavelengths (depending on the *Plot* settings).

Note that "bad" will not work from here, as it can only be applied to ALL of the loaded spectra.

To apply range(s) to multiple spectra, use the *AllData* ignore function.

ignoredString()

Return a string of ignored channel ranges.

This produces a string in compact (hyphenated) form, which can be used as input to a subsequent 'ignore' command. Example:

If ignored channels are [1,3,4,5,7], this function will output "1 3-5 7".

notice(*noticeRange*)

Notice a range of the spectrum by channels or energy/wavelengths.

Args:
noticeRange: String specifying the channel range to notice.

This follows the same syntax as used in the standard Xspec "notice" command. If the numbers are floats rather than ints, they will be treated as energies or wavelengths (depending on the *Plot* settings). If the string is "all", it will notice all channels in spectrum.

To apply range(s) to multiple spectra, use the *AllData* notice function.

noticedString()

Return a string of noticed channel ranges.

This produces a string in compact (hyphenated) form, which can be used as input to a subsequent 'notice' command. Example:

If noticed channels are [1,3,4,5,7], this function will output "1 3-5 7".

show()

Display information for this Spectrum object

property areaScale

The Spectrum area scaling factor.

This is either a single float (if file stores it as a keyword), or a tuple of floats (if file stores column).

property backScale

The Spectrum background scaling factor.

This is either a single float (if file stores it as a keyword), or a tuple of floats (if file stores column).

property background

Get/Set the spectrum's background.

Get:

Returns the Background object associated with the Spectrum. If Spectrum has no background object, this will raise an Exception.

Set:

Supply a background filename [string]. This will become the new background to the Spectrum object, and any previously existing background will be removed. If string is empty, all whitespace, or the Python `None` variable, the background (if any) will be removed.

property cornorm

Get/Set the normalization of a spectrum's correction file. [float]

property correction

Get/Set the correction file.

Get:

Returns the Spectrum's current correction information as an object of class Background. This raises an Exception if Spectrum has no correction.

Set:

Enter the filename string for the new correction. This will remove any previously existing correction. Returns the new correction info as an object of class Background. If string is "none", empty, or all whitespace, the current correction will be removed and this will return `None`.

property dataGroup

The data group to which the spectrum belongs [int].

property energies

Tuple of pairs of floats (also implemented as tuples) giving the E_Min and E_Max of each noticed channel.

property eqwidth

Tuple of 3 floats containing the results of the most recent eqwidth calculation for this spectrum (performed with the *AllModels.eqwidth* method).

The results are stored as:

[0] - eqwidth calculation

[1] - eqwidth error lower bound

[2] - eqwidth error upper bound

The error bounds will be 0.0 if no error calculation was performed, and all will be 0.0 if eqwidth wasn't performed for this spectrum.

property exposure

The exposure time keyword value [float].

property fileName

The spectrum's file name [string].

property flux

A tuple containing the results of the most recent flux calculation for this spectrum.

The tuple values are: (value, errLow, errHigh (in ergs/cm^2), value, errLow, errHigh (in photons)) for each model applied to the spectrum.

property ignored

A list of the currently ignored (1-based) channel numbers.

property index

The spectrum's current index number within the AllData container [int].

property isPoisson

Boolean flag, true if spectrum has Poisson errors.

property lumin

Similar to flux, the results of the most recent luminosity calculation.

The tuple values are: (value, errLow, errHigh (in 10^44 ergs/sec), value, errLow, errHigh (in photons)) for each model applied to the spectrum.

property multiresponse

Get/Set detector response ARRAY elements when using multiple sources.

This is for use only when assigning multiple responses to a spectrum, for multi-source/multi-model analysis. For standard single-source analysis, use the *response* attribute instead.

You must provide an array index for all *multiresponse* get/set operations. Note that array indices ARE 0-BASED, so multiresponse[0] corresponds to source 1. Examples:

```
# Get the response assigned to source 1.
# This particular call is the same as doing
# "r1 = s.response"
r1 = spec.multiresponse[0]

# Get the response for the second source.
# Can only do this with multiresponse.
r2 = spec.multiresponse[1]

# Define a third source by adding a new response:
spec.multiresponse[2] = "myResp3.pha"

# Now remove the response for the second source:
spec.multiresponse[1] = None
```

property noticed

A list of the currently noticed (1-based) channel numbers.

property rate

A tuple containing the total Spectrum rates in counts/sec.

The tuple consists of:

[0] - current net rate (w/ background subtracted),

[1] - net rate variance,

[2] - total rate (without background),

[3] - predicted model rate

property response

Get/Set the detector response.

Use this for standard SINGLE-SOURCE analysis. To add other responses for multi-source and multi-model analysis, use the *multiresponse* attribute.

Get:
Returns a Response object, or raises an Exception if none exists

Set:
Supply a response filename string. To remove a response, supply an empty string or `None`.

`property responsesUsed`

Return a list of detector slot numbers with currently assigned responses.

The values returned are 0-based ([0] = source 1, [1] = source 2, etc.) If no responses are assigned to spectrum, the list will be empty.

`property statistic`

Spectrum's contribution to the total fit statistic [float].

`property values`

Tuple of floats containing the spectrum rates for noticed channels in counts/sec.

`property variance`

Tuple of floats containing the variance of each noticed channel.

`property xflt`

XFLT key-value pairs returned as a tuple of tuples

3.1.15 XspecSettings

`class xspec.XspecSettings`

Storage class for Xspec settings.

PyXspec automatically creates a single object of this class, named *Xset*.

Methods

- addModelString
- closeLog
- delModelString
- openLog
- restore
- save

Attributes ((*) = get-only)

- abund
- allowNewAttributes
- allowPrompting
- chatter
- logChatter
- cosmo
- log*
- modelStrings
- parallel
- seed
- version*
- xsect

`addModelString`(*key*, *value*)

Add a key,value pair of strings to XSPEC's internal database.

This database provides a way to pass string values to certain model functions which are hardcoded to search for "key". (See the XSPEC manual description for the "xset" command for a table showing model/key usage.)

If the key,value pair already exists, it will be replaced with the new entries.

`closeLog()`

Close XSPEC's current log file.

`delModelString(key)`

Remove a key,value pair from XSPEC's internal string database.

`openLog(fileName)`

Open a file and set it to be XSPEC's log file.

Args:
fileName: The name of the log file.

If Xspec already has an open log file, it will close it. Returns a Python file object for the new log file.

Once opened, the log file object is also stored as the *Xset.log* attribute.

Note: To ensure proper cleanup and file flushing, it is recommended that you call *Xset.closeLog()* before exiting PyXspec.

`restore(fileName)`

Restore the data/model configuration and settings.

This will restore the data, models, and settings from a previous PyXspec session, as saved in file generated by the *Xset.save()* function.

Args:
fileName: The output file from a previous *Xset.save* command.

`save(fileName, info='a')`

Save the data and model configuration and XSPEC settings

Args:
fileName: The name of the output file.

If the file name has no extension, '.xcm' will be appended.

info: A flag specifying which information to save:

- 'a' = save all (the default)
- 'f' = only save the data files information
- 'm' = only save the model information

`property abund`

Get/Set the abundance table used in the plasma emission and photoelectric absorption models.

Set:
Enter one of the following:

The name of one of Xspec's built-in abundance tables:
'angr', 'aspl', 'feld', 'aneb', 'grsa', 'wilm', 'lodd'

'file <filename>' where <filename> contains 30 abundance
values, one per line.

A list or tuple containing 30 abundance values.

A single string containing 30 space-delimited values.

Get:
Returns the name of the input table followed by its 30 values [string].

property allowNewAttributes

Get/Set the flag which allows the setting of new instance attributes for ALL PyXspec classes [bool].

This is False by default, and is intended to catch the user's attention if they misspell an attribute name when attempting to set it. Under normal Python behavior, a misspelling would simply create a new attribute and issue no warnings or errors.

You must make sure this flag is set to True if you genuinely wish to add new attributes.

property allowPrompting

Get/Set flag determining whether user prompting occurs.

Get/Set whether user will be prompted in situations where XSPEC may require additional information to complete a task. Default is 'True'. [bool]

property chatter

Get/Set the console chatter level [int].

property cosmo

Get/Set the cosmology values.

Get: Returns a tuple of floats containing (H0, q0, l0), where

- H0 is the Hubble constant in km/(s-Mpc),
- q0 is the deceleration parameter, and
- l0 is the cosmological constant.

Set: Enter a single string containing one or more of H0, q0, l0.

Examples:

```
Xset.cosmo = "100"  # sets H0 to 100.0
Xset.cosmo = ",0"   # sets q0 to 0.0
Xset.cosmo = ",,0.7"  # sets l0 to 0.7
Xset.cosmo = "50 .5 0." # sets H0=50.0, q0=0.5, l0=0.0
```

property log

Get only: Returns the currently opened log file object, or `None` if no log file is open (also see the *openLog* and *closeLog* methods).

property logChatter

Get/Set the log chatter level [int].

property modelStrings

XSPEC's internal database of <string_name>, <string_value> pairs for settings which may be accessed by model functions.

Get:
Returns a tuple of tuples, the inner tuples being composed of <string_name>,<string_value> string pairs.

Set:
Replaces ENTIRE database with user-supplied new database. Input may be a dictionary of <string_name>:<string_value> entries, or a tuple of (<string_name>,<string_value>) tuples.

For inserting and deleting INDIVIDUAL string name and value pairs, use the *addModelString* and *delModelString* methods.

property parallel

An attribute for controlling the number of parallel processes in use during various XSPEC contexts.

Examples:

```
# Use up to 4 parallel processes during
#   Levenberg-Marquardt fitting.
Xset.parallel.leven = 4

# Use up to 4 parallel processes during
#   Fit.error() command runs.
Xset.parallel.error = 4

# Other available contexts are:
# Xset.parallel.steppar, walkers, and goodness.

# Display current settings of all parallel contexts.
Xset.parallel.show()

# Reset all contexts to single process usage.
Xset.parallel.reset()
```

property seed

Re-seed and re-initialize XSPEC's random-number generator with the supplied integer value (SET only).

property version

The version strings for PyXspec and standard XSPEC.

GET only, this returns a tuple containing:

(<PyXspec version string>,<standard XSPEC version string>)

property xsect

Get/set the photoelectric absorption cross-sections in use [string].

Available options: 'bcmc', 'obcm', 'vern'

3.2 Module-Level Functions

xspec.**callModelFunction**(*modelName*, *energies*, *params*, *flux*, *fluxError=None*, *spectrumNum=None*, *initString=None*)

Directly call an XSPEC model function.

This provides an interface for directly accessing the functions in the XSPEC models library. Note that this utility does NOT create PyXspec Model objects for fitting to spectra. Nor does it interact with any other PyXspec objects and settings. (For that you would use the **Model** class and/or the *AllModels* container.) This merely takes input arrays of energy and parameter values and calls on the requested model function to calculate a flux array.

Args:

modelName: Name of model component to perform the calculation. [string]

The full name must be given here (as it appears in Model.showList()). Abbreviations are not handled.

energies: Input array of energy bin values. [list or tuple]

params: Input array of parameter values. [list or tuple]

The size of this array must match the number of parameters required by the model function, otherwise an error is thrown.

flux: Input/output array of calculated flux. [list]

This list should normally be size 0 upon input. It will be resized to nEnergies-1 and filled with the calculated flux values upon output.

However if calling a model function that requires a pre-filled flux array (such as a convolution component), it should be filled to size nEngeries-1 upon input.

***fluxError*: Optional input/output array for model functions which also calculate errors. [list]**

This should normally be set to the default = None. But if used, it follows the same size requirements as the flux array.

spectrumNum: Optional spectrum number value. [int]

This is generally not required in this context and should be left to the default = None. It exists as part of the model function interface for internal XSPEC usage (particularly mix model components).

***initString*: Optional input string for model functions which may take additional initialization information. [string]**

Generally not required in this context. Defaults to None.

Examples:

```
# Create an energy array of 5 bins, each bin centered on 1-5 keV,
# and pass to XSPEC's powerlaw model function to calculate the flux.
# The powerlaw model requires 1 parameter, the photon index.
engs = (.5, 1.5, 2.5, 3.5, 4.5, 5.5)   # can be a tuple or list
pars = [2.0]    # tuple or list
flux = []       # must be a list
callModelFunction('powerlaw', engs, pars, flux)
# flux array will now be size 5 with the calculated values
```

xspec.**callTableModel**(*tableType*, *tableFile*, *energies*, *params*, *flux*, *fluxError=None*, *spectrumNum=None*, *initString=None*)

Directly call an XSPEC table model.

This provides an interface for directly accessing XSPEC table models. Note that this utility does NOT create PyXspec Model objects for fitting to spectra. Nor does it interact with any other PyXspec objects and settings. (For that you would use the **Model** class with an expression containing table model syntax, ie. atable{<filename>}) This merely takes input arrays of energy and parameter values and interpolates a flux array from the requested table model file.

Args:

***tableType*: Allowed table types are 'a', 'm', or 'e' for additive, multiplicative, and exponential table models respectively.**

tableFile: Name of the table model file. [string]

energies: Input array of energy bin values. [list or tuple]

params: Input array of table parameter values. [list or tuple]

flux: Input/output array of calculated flux. [list]

This list should normally be size 0 upon input. It will be resized to nEnergies-1 and filled with the calculated flux values upon output.

***fluxError*: Optional input/output array for model functions which also calculate errors.** [list]

This should normally be set to the default = None. But if used, it follows the same size requirements as the flux array.

spectrumNum: Optional spectrum number value. [int]

This is generally not required in this context and should be left to the default = None. It exists as part of the model function interface for internal XSPEC usage (particularly mix model components).

***initString*: Optional input string for model functions which may take additional initialization information.** [string]

Generally not required in this context. Defaults to None.

Examples:

```
# Create an energy array of 5 bins, each bin centered on 1-5 keV,
# and pass to a powerlaw table model file named testpo.mod to
# calculate the flux. The table model requires 1 parameter, the
# photon index.
engs = (.5, 1.5, 2.5, 3.5, 4.5, 5.5)   # can be a tuple or list
pars = [2.0]    # tuple or list
flux = []       # must be a list
callTableModel('a','testpo.mod', engs, pars, flux)
# flux array will now be size 5 with the calculated values
```

CHAPTER

FOUR

NOTES

4.1 Release Notes

4.1.1 Version 2.1.3 Feb 2024 [XSPEC 12.14.0]

New Features

- New methods *AllModels.setActive()* and *AllModels.setInactive()* to provide access to standard Xspec's "model active" and "model inactive" functionality.

Fixes

previously available as an XSPEC patch

- In *AllModels.addPyMod()*, the use of deprecated Python function was preventing execution with Anaconda Python v3.11.

4.1.2 Version 2.1.2 Jul 2023 [XSPEC 12.13.1]

New Features

- New method *Plot.nAddComps()* for returning the number of add component plots for a given plot group. This can be helpful for determining the valid range of the *addCompNum* input argument to *Plot.addComps()*.
- Two new methods to take advantage of standard Xspec's new ability to set systematic model errors for specific models: *AllModels.systematicSingleModel* and *AllModels.setSystematicSingleModel*.

4.1.3 Version 2.1.1 Nov 2022 [XSPEC 12.13.0]

New Features

- The *Fit.goodness()* function takes new optional argument, *[no]fit*, to provide same functionality available in standard Xspec's *goodness* command.
- New attributes for the *Fit* class: *previousGoodness* and *previousGoodnessSims*
- New function *AllModels.mdefine()* to provide access to standard Xspec's *mdefine* capability.
- The *Xset.abund* attribute can now take an input list of 30 values in addition to the already available selection of built-in tables.

4.1.4 Version 2.1.0 Feb 2022 [XSPEC 12.12.1]

New Features

- Parameter links may now be assigned directly to other parameter objects rather just to a string listing index numbers. For example it is now possible to do something like *m2.gaussian.LineE.link = m1.gaussian.LineE*.
- A new function *xspec.callTableModel* gives local models written in Python access to Xspec's table model functionality.
- A single spectrum's contribution to the overall fit statistic can now be accessed through the *Spectrum.statistic* attribute.
- PyXspec has been reorganized internally such that the portion that is compatible with the Xspec models-library-only distribution is now separated into it's own module, named *mxspec*. The original *xspec* module includes this, so the change should only be noticeable to those using a models-library-only distribution.

4.1.5 Version 2.0.5 Apr 2021 [XSPEC 12.12.0]

New Features

- To help with passing information to a 3rd-party plotting package (such as matplotlib), several plot retrieval methods have been added. These are: *Plot.labels()*, *Plot.contourLevels()*, and *Plot.z()*
- The *Spectrum* object constructor (*__init__* function) now takes optional arguments for specifying background, response, and arf files. This is useful when you don't want to use the default file names (if any) that are stored in the Spectrum file.
- Code added for /svg graphics to work when PyXspec is run in Jupyter notebooks.
- Beginning with Xspec 12.12.0, spectrum and background value arrays are no longer divided by the area scale. In PyXspec, that carries over to the *Background.values* and *Spectrum.values* attributes.

Fixes

- In certain cases where errors occur while the ctrl-c signal handler is active, the Python session will crash when the user attempts to exit, or a script finishes.
- PyXspec terminates when *AllData.fakeit()* can't find a response file and allowPrompting is turned off.

4.1.6 Version 2.0.4 Aug 2020 [XSPEC 12.11.1]

Fixes

- The *Spectrum.dummyrsp()* function had been mixing up the arguments for spectrum number and source number.

4.1.7 Version 2.0.3 Mar 2020 [XSPEC 12.11.0]

New Features

- Plot values can now be retrieved from the individual additive components within a model, using the new *Plot.addComp()* method.
- Added Fit.nVarPars attribute to perform the equivalent of standard Xspec's "tclout varpar".
- Added access to standard Xspec's "rerror" command (for running the "error" command on response parameters. This is now available by passing a second (bool) argument to the Fit.error() method. (This has been previously released as an XSPEC patch.)

4.1.8 Version 2.0.2 Oct 2018 [XSPEC 12.10.1]

New Features

- It is now possible to call XSPEC's model functions directly from PyXspec using the new function *xspec.callModelFunction()*. This utility is particularly useful for users who wish to access particular XSPEC model functions within their own local model Python code.

Fixes

- Some XSPEC model parameter names contained parentheses, which were incompatible when used in PyXspec. These parentheses have been replaced with a single underscore.

4.1.9 Version 2.0.1 Mar 2018 [XSPEC 12.10.0]

(*) = previously available as XSPEC patches

New Features

- New methods for *AllChains*: *margin()* and *marginResults()*, *best()* and *dic()*, providing access to equivalent MCMC functionality in standard XSPEC.
- The *Chain* class attribute: *rescale*.
- Additions to *Spectrum* class: *xflt* and *responsesUsed* attributes, *fileinfo()* method.
- *AllModels.identify()* method, wrapping standard XSPEC's 'identify' and 'tclout idline' functionality.
- *Fit.nullhyp* attribute now wraps 'tclout nullhyp'.*
- The *Xset.parallel* object now has additional attributes: *steppar*, *walkers*, and *goodness*, and a new *show()* method.*

Fixes

- The *Model.energies()* function had been returning values as a list of strings. This has been changed to a more convenient list of floats.
- The *Xset.restore()* function was not properly handling 'cd' commands, nor gain fit parameters.*

4.1.10 Version 2.0.0 Nov 2016 [XSPEC 12.9.1]

New Features

- Now compatible with Python 3.x. (No longer compatible with Python versions prior to 2.5.)
- Added *Xset.save()* and *restore()* functions for saving and recovering the state of a PyXspec session.
- New adjustable attributes for the *Chain* class: *walkers* and *algorithm*.*
- *AllModels.tclLoad()* method added to provide a lower-level alternative to the *lmod()* local model loading scheme.*
- *Xset.allowPrompting* attribute provides option for turning on/off XSPEC prompting. This can be useful in iPython environments where certain contexts can lead to infinite prompting loops.
- PyXspec manuals have been converted from Doxygen-generated to Sphinx-generated HTML and PDF.

Fixes

- Was previously unable to unlink a linked parameter in a Python-coded local model.*

4.1.11 Version 1.1.0 Jul 2015 [XSPEC 12.9.0]

New Features

- Local models may now be written in Python and inserted into XSPEC's models library with new *AllModels.addPyMod()* function.
- The plot array retrieval interface (ie. *Plot.x()*, *Plot.y()*) has been expanded to allow retrieval from secondary plot panels in a multi-panel plot.
- New *Parameter.index* attribute.
- New *backscale* attribute for Spectrum and Background classes.
- Added new function *Fit.stepparResults()* for retrieving results of most recent steppar run. (Previously available as a patch)
- New *noWrite* option added to *AllData.fakeit* (Previously available as patch)

Fixes

- The *Model.__call__* function now returns Parameter objects by reference rather than by value. This is to allow the returned object to retain any custom attributes the user may have added.
- Improved handling of *Ctrl-c* breaking in several prompting contexts.

4.1.12 Version 1.0.4 Jul 2014 [XSPEC 12.8.2]

- Added *Fit.testStatistic* attribute for retrieving the test statistic value from the most recent fit.
- Added compiler macros for switching to <Python/Python.h> include paths when building on Mac platforms.
- Bug fix for get/set *Spectrum.correction* files on OS X Mavericks.

4.1.13 Version 1.0.3 Aug 2013 [XSPEC 12.8.1]

- The *Fit.statMethod* and *statTest* attributes can now be set for a range of individual spectra rather than only applying to all.
- Bug fix to the RModel class (ie. the class of the *Response.gain* attribute). If the user assigned multiple RModel objects to point to the same underlying XSPEC response gain, changes made through one object weren't necessarily showing up in the other objects. This also fixes a related bug created in Xspec patch 12.8.0l that caused a list of error messages to appear (only) in Python versions 2.6.x.

4.1.14 Version 1.0.2 Jan 2013

- Added *Xset.parallel* attribute, with options 'leven' and 'error' for setting parallel processes.
- Added *Fit.statTest* attribute for getting/setting the XSPEC test statistic.

4.1.15 Version 1.0.1 Dec 2012 [XSPEC 12.8.0]

- 2 additions to the Spectrum class: an *ignored* attribute and an *ignoredString()* function. The former returns a Python list object containing every ignored channel number. The latter returns the same information in convenient string form, which can be reused as input to a future *ignore* or *notice* command.

4.1.16 Version 1.0 Feb 2012 [XSPEC 12.7.1]

Changes relative to the PyXspec **Beta** version:

> **Warning:** Two Backwards-Incompatible Changes:

- When using **multiple data groups**, the Model objects assigned to the higher-numbered groups now all have their parameters indexed from **1 to nPar**. For example with a 3 parameter model applied to 2 data groups, you would now access the first parameter in the 2nd model object with *mod2(1)* rather than *mod2(4)*.
- The *Model.setPars()* function (introduced with patch 12.7.0f) used the **p`n`** keyword argument syntax to set non-consecutive parameters. This has been replaced. with the use of Python **dictionaries**. For example, *m.setPars(p2=.3, p4=1.1)* should now be *m.setPars({2:.3, 4:1.1})*.

New Features

- Added Standard XSPEC's *gain* command functionality. This is implemented with the new *gain* attribute for Response classes. *Response.gain* is a class of type RModel, and has two Parameter objects: *slope* and *intercept*.
- New *AllModels.setPars()* function for changing multiple parameters in multiple Model objects with a single call.
- Now compatible with Cygwin.

  Features Previously Added As Patches To XSPEC 12.7.0
- *AllModels.initpackage()* for building local models inside the Python shell.
- Bayesian inference provided through the *Fit.bayes* and *Parameter.prior* attributes.
- *Fit.goodness()* and *Fit.improve()* functions.
- *Model.setPars()* function for changing multiple parameters with a single call.
- *AllModels.simpars()* function to do the equivalent of Standard XSPEC's tclout simpars.
- *Fit.covariance* attribute for retrieving the covariance matrix from the most recent fit.
- *Model.expression* attribute which stores the model expression string.
- *AllModels.sources* attribute which stores a map of source number and model name assignments.

Fixes

- All PyXspec bug fixes previously released as patches to XSPEC 12.7.0 are included.
- Now handles model component-by-name access when the component is a table model whose name includes whitespace.

4.2 What's Missing

Python equivalents for these standard XSPEC commands are not yet implemented:

- hardcopy
- Tcl script commands: addline, lrt, modid, multifake, rescalecov, simftest, writefits

The following commands perform functions which are not applicable to the currently intended design and usage of PyXspec, and therefore are not likely to be implemented in the near future:

- addcomp
- autosave
- delcomp
- editmod
- script

CHAPTER

FIVE

AUTHORS

PyXspec was developed by Craig Gordon and Keith Arnaud
HEASARC Software Development, Astrophysics Science Division,
Code 660.1, NASA/GSFC, Greenbelt MD 20771

Please send questions, comments, and bug reports to xspec12@athena.gsfc.nasa.gov

INDEX

Symbols

A

B

C

D

S

T

U

V

W

X

Y

Z