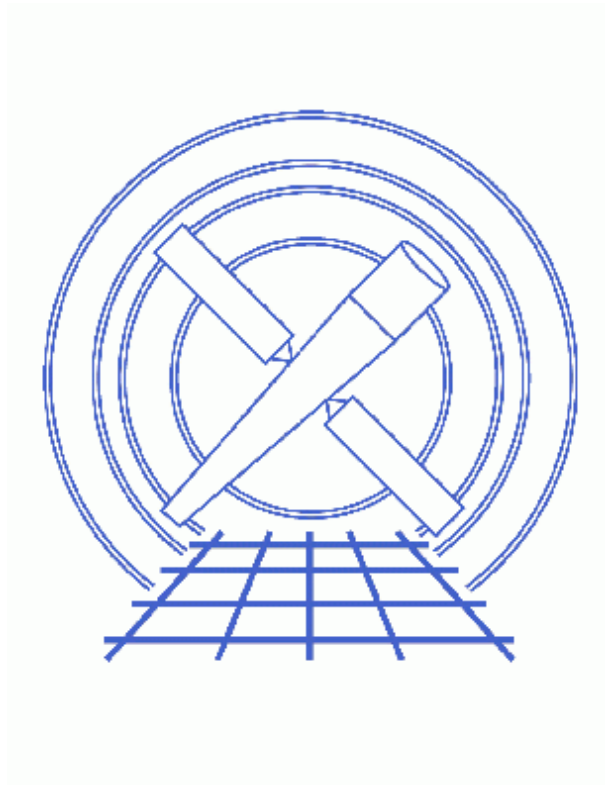


# Introduction to the Sherpa S–Lang Module



## Sherpa Threads (CIAO 3.4)

# Table of Contents

- ***Introduction***
- ***Fitting a PHA Spectrum***
  - ◆ Loading Data and Responses
  - ◆ Filtering Data and Subtracting Background
  - ◆ Defining a Source Model
  - ◆ Fitting
  - ◆ Examining Fit Results
  - ◆ Scripting the Procedure
- ***Fitting and Plotting ASCII Data with Errors***
  - ◆ Loading Data and Errors
  - ◆ Plotting the Data
  - ◆ Defining a Source Model and Fitting
  - ◆ Plotting the Fit
  - ◆ Expanding the Possibilities with a Script
- ***A Note on sherpa eval***
- ***History***
- ***Images***
  - ◆ Plot of data retrieved by "get" functions
  - ◆ Plot of data and fit retrieved by "get" functions
  - ◆ Plot created by "polyfit.sl"

---

# Introduction to the Sherpa S–Lang Module

*Sherpa Threads*

## Overview

*Last Update:* 1 Dec 2006 – reviewed for CIAO 3.4: no changes

### *Synopsis:*

The [Sherpa S–Lang module](#) is an extension to *Sherpa* that allows one to employ its full capabilities from within a [S–Lang](#) script or another S–Lang–enabled application (such as [ChIPS](#)). This thread provides an introduction to the module via some example applications.

### *Read this thread if:*

You want to be able to customize and extend *Sherpa* via S–Lang functions and scripts or use *Sherpa*'s functionality in another S–Lang–enabled application.

### *Related Links:*

- The [sherpa-module](#) ahelp page
- [Sherpa and Scripts](#)
- [Accessing fit results using S–Lang](#)
- [A Guide to the S–Lang Language](#)

Proceed to the [HTML](#) or *hardcopy* ([PDF: A4](#) | [letter](#)) version of the thread.

---

## Introduction

The [Sherpa S–Lang module](#) allows the user to employ *Sherpa*'s full functionality from within a [S–Lang](#) script or another S–Lang–enabled application (such as [ChIPS](#)). This permits one to enhance other applications with *Sherpa*'s features and (more importantly) to extend the capabilities of *Sherpa* beyond its design without altering *Sherpa* itself. Hence, the *Sherpa* S–Lang module lets the user break free of the "black box" model and enhance *Sherpa* to meet his or her specific needs.

This thread introduces the *Sherpa* S–Lang module by guiding the user through some basic examples of its use. It also provides example scripts that users may download and modify to suit their own needs. For more examples of scripts that use the *Sherpa* module, see the [Sherpa](#) section of the [CIAO scripts page](#). For an introduction to the S–Lang programming language, see [A Guide to the S–Lang Language](#).

## Sherpa S–Lang Module – Sherpa

The *Sherpa* S–Lang module is available automatically within a *Sherpa* session and in scripts run by the *sherpa* executable. To use the module in external applications, one must load it using the `import` function. For example, within *ChIPS*:

```
chips> import("sherpa")
```

Or, within an `slsh` script:

```
unix% more myscript
#!/usr/bin/env slsh

import("sherpa");
...
```

For more information on driving *Sherpa* with S–Lang scripts, see the [Sherpa and Scripts](#) thread.

---

## Fitting a PHA Spectrum

As our first example, we will perform a basic fit to a PHA spectrum. The data and procedure are identical to those used in the [Introduction to Fitting PHA Spectra](#) thread. However, we will perform the fit using *only* functions from the *Sherpa* S–Lang module.

### Loading Data and Responses

To begin, we load the spectrum file using `load_dataset`:

```
sherpa> load_dataset("3c273.pi")
1
```

Notice that "1" was printed to the screen. This is the return value of the function; "1" indicates success, while "0" would indicate failure (e.g. if the file did not exist). In scripts, you should usually check a function's return value, as a failure condition generally means that a script must abort or enter an error–handling mode. Here, we will simply discard return values using the syntax "`() = ...`". For example, we could have discarded the "1" returned by `load_dataset` as follows:

```
sherpa> () = load_dataset("3c273.pi")
```

Note that the type and meaning of return values differ between functions, and some functions return nothing at all. For more information on return values for a particular function, see the function's [ahelp](#) file.

Since header keywords in the file `3c273.pi` specify associated [RME](#), [ARF](#), and background files, `load_dataset` automatically loads these files and establishes an appropriate instrument model. We can verify this by issuing the `SHOW` command:

```
sherpa> SHOW

Optimization Method: Levenberg-Marquardt
Statistic:           Chi-Squared Gehrels

-----
Input data files:
-----
```

```

Data 1: /data/sherpa/pha_intro/3c273.pi.
Total Size: 46 bins (or pixels)
Dimensions: 1
Total counts (or values): 736
Exposure: 38564.61 sec
Count rate: 0.019 cts/sec
  Backscal: 2.526436e-06

Background 1: /data/sherpa/pha_intro/3c273_bg.pi.
Total Size: 1024 bins (or pixels)
Dimensions: 1
Total counts (or values): 216
Exposure: 38564.61 sec
Count rate: 0.006 cts/sec
  Backscal: 1.872535e-05

The data are NOT background subtracted.

-----
Defined analysis model stacks:
-----

instrument source 1 = respo2
instrument back 1 = respo2

-----
Defined instrument model components:
-----

rsp1d[respo2]
  Param   Type      Value      Min      Max      Units
  -----
  1   rmf string: "3c273.rmf" (N_E=1090,N_PHA=1024)
  2   arf string: "3c273.arf" (N_E=1090)

```

## Filtering Data and Subtracting Background

Next, we establish an energy filter (selecting energies in the range 0.1–6.0 keV) using the `set_notice` function and subtract the background data using `set_subtract`:

```

sherpa> () = set_notice(,0.1,6.0)
sherpa> () = set_subtract

```

Note that in the `set_notice` call, a comma appears before 0.1. This is necessary because `set_notice` actually takes three parameters: dataset number, lower bound, and upper bound. If the first parameter is empty (as above), the dataset number defaults to 1.

## Defining a Source Model

To establish a source model, we use the function `set_source_expr`:

```

sherpa> () = set_source_expr("xsphabs[abs]*powlaw1d[p1]")

```

This creates the specified source model using the default parameter values for each model component. Note that `set_source_expr` does *not* prompt for parameter values, regardless of the setting of `PARAMPROMPT`.

Next, we set the hydrogen column density in the `abs` model component (with `set_par`) and freeze the

component (with `set frozen`):

```
sherpa> () = set_par("abs.nh","value",0.07)
sherpa> () = set_frozen("abs")
```

We can verify that the source model has indeed been established using `SHOW`:

```
sherpa> SHOW source
(abs * p1)
xsphabs[abs] (XSPEC model name: phabs) (integrate: off)
  Param  Type      Value      Min      Max      Units
  ----  -
  1      nH frozen    7e-02    1e-07    10      10**22 atoms/cm**2
powlaw1d[p1] (integrate: on)
  Param  Type      Value      Min      Max      Units
  ----  -
  1      gamma thawed    1         -10     10
  2      ref frozen    1         0.1248  5.9057
  3      ampl thawed   7.9256e-06 7.9256e-08 7.9256e-04
```

## Fitting

We now fit the model using the function `run_fit`:

```
sherpa> good = run_fit
LVMQT: V2.0
LVMQT: initial statistic value = 355.854
LVMQT: final statistic value = 37.9079 at iteration 10
      p1.gamma  2.1585
      p1.ampl   0.000224838
```

## Examining Fit Results

After performing the fit, `run_fit` calls `get_goodness`, which returns a `S-Lang structure` that contains information about the quality of the fit. We have stored this structure in a `S-Lang variable` named `good` (the name is arbitrary) and can display its contents using the `Varmm` function `print`:

```
sherpa> print(good)
dataset      = 1
datatype     = source
stat         = 37.9079
numbins     = 44
dof          = 42
rstat       = 0.902569
qval        = 0.651155
```

We can also access individual fields of the `good` structure using the syntax `<structname>.<fieldname>`. For example, we can store the statistic value in a `S-Lang variable` named `statval` and save it for future use:

```
sherpa> statval = good.stat
sherpa> print(statval)
37.9079
```

Since `run_fit` returns goodness-of-fit information automatically, there is no need to issue the `GOODNESS` command. For more information on using fit results, see the thread "[Accessing fit results using S-Lang](#)".

## Sherpa S-Lang Module – Sherpa

We can obtain confidence intervals for our model parameters using `run_cov`, the S-Lang equivalent of the `COVARIANCE` command:

```
sherpa> conf = run_cov

Computed for sherpa.cov.sigma = 1

-----
Parameter Name      Best-Fit Lower Bound      Upper Bound
-----
p1.gamma            2.1585  -0.0827851  +0.0827851
p1.ampl             0.000224838  -1.48256e-05  +1.48256e-05
```

`run_cov` returns an array of structures, which we have stored in the variable `conf`. Each element of the array contains the confidence interval for one parameter, which we can display using `print`:

```
sherpa> print(conf[0])
name      = p1.gamma
val       = 2.1585
vlo       = 2.07571
vhi       = 2.24128
sigma     = 1
sherpa> print(conf[1])
name      = p1.ampl
val       = 0.000224838
vlo       = 0.000210012
vhi       = 0.000239663
sigma     = 1
```

---

## Scripting the Procedure

Although it is entirely valid to use *Sherpa* module functions from the *Sherpa* command line (as we have done so far in this thread), they provide few advantages over traditional *Sherpa* commands during interactive use. The real benefit of the *Sherpa* S-Lang module is that it allows one to harness the full capabilities of *Sherpa* from within a S-Lang script.

The S-Lang script `phafit.sl` contains all the commands used above to fit our example spectrum:

```
unix% more phafit.sl
() = load_dataset("3c273.pi");
() = set_notice(,0.1,6.0);
() = set_subtract();
() = set_source_expr("xsphabs[abs]*powlaw1d[p1]");
() = set_par("abs.nh","value",0.07);
() = set_frozen("abs");
variable good = run_fit();
variable conf = run_cov();
```

Note that within a S-Lang script, each statement must end with a semi-colon, and variables must be declared (via the `variable` keyword) before use. (These requirements, which are a standard part of the S-Lang language, are relaxed at the *Sherpa* command line.)

To run this script from the *Sherpa* command line, use the `evalfile` function. Note that if you are still working in the same *Sherpa* session as above, you will have to start a new session or issue the command "`ERASE ALL`" before running the script:

```
sherpa> () = evalfile("phafit.sl")
```

For more information on running S–Lang scripts, see the [Sherpa and Scripts](#) thread.

## Fitting and Plotting ASCII Data with Errors

As our second example, we will demonstrate more of the *Sherpa* module's [set](#) and [get](#) functions by reproducing part of the [Introduction to Fitting ASCII Data with Errors](#) thread. Then, we will show how the *Sherpa* module can be used to extend the capabilities of *Sherpa* by providing a S–Lang script that displays three different fits to a dataset on a single plot.

### Loading Data and Errors

The ASCII data file we want to use contains three columns. The first is the independent variable (x), the second is the dependent variable (y), and the third is error in the dependent variable ( $y_{\text{err}}$ ):

```
unix% more data1.dat
0.5    1.6454  0.04114
1.5    1.7236  0.04114
2.5    1.9472  0.04114
3.5    2.2348  0.04114
...
```

None of the *Sherpa* module's [load](#) functions can handle a file in this format, so we will have to use [set](#) functions to load the data.

First, we read in the file using the [Varmm](#) function [readfile](#):

```
sherpa> dat = readfile("data1.dat")
```

[readfile](#) returns a S–Lang structure that holds both "metadata" about the file (e.g. file name and format) and arrays containing the actual data columns:

```
sherpa> print(dat)
_filename      = data1.dat
_path          = /data/sherpa/basic/
_filter        = NULL
_filetype      = 1
_header        = NULL
_ncols         = 3
_nrows        = 11
col1           = Float_Type[11]
col2           = Float_Type[11]
col3           = Float_Type[11]
```

For ASCII files, the column array names correspond to the order in which the columns appear in the file (`col1` is the first column, `col2` is the second column, etc.). We can examine the contents of a column array using the [print](#) function:

```
sherpa> print(dat.col1)
0.5
1.5
2.5
3.5
...
```



## Sherpa S-Lang Module – Sherpa

It is also possible to select and use individual array elements using the syntax `<arrayname>[<index>]`. Since S-Lang array indices start at zero, we can print the third element of the `col1` array as follows:

```
sherpa> print(dat.col1[2])
2.5
```

We now have to load the data arrays into *Sherpa*. To do this, we use three functions: `set_axes` to load the independent-variable column, `set_data` to load the dependent-variable column, and `set_errors` to load the errors:

```
sherpa> () = set_axes(dat.col1)
sherpa> () = set_data(dat.col2)
sherpa> () = set_errors(dat.col3)
```

To confirm that the data have been loaded, we can issue the `SHOW` command:

```
sherpa> SHOW
...
-----
Input data files:
-----
# dataset 1 loaded via S-Lang module
```

---

## Plotting the Data

We now want to plot the data. The simplest way to do this using a *Sherpa* module function is to call the `LPLOT` command via `sherpa_eval`:

```
sherpa> () = sherpa_eval("LPLOT DATA")
```

However, a more flexible and potentially powerful approach is to use `get` functions to obtain the data and the `curve` function to plot it. First, we obtain the dataspace, data, and errors using `get_axes`, `get_data`, and `get_errors`, respectively:

```
sherpa> x = get_axes
sherpa> y = get_data
sherpa> y_err = get_errors
```

`get_data` and `get_errors` return simple arrays containing the relevant data. However, `get_axes` returns a structure:

```
sherpa> print(x)
axistype      = Channels
axisunits     = bin
lo            = NULL
hi            = NULL
mid           = Double_Type[11]
```


If we were using binned data (e.g. from a PHA spectrum), the `mid` field would be `NULL`, and the `lo` and `hi` fields would be arrays containing the lower and upper boundaries, respectively, for each bin. However, since we are using unbinned data, `lo` and `hi` are `NULL`, and `mid` is an array containing the independent-axis gridpoints. Since this array is all we want, we store it in the variable `x`, overwriting the structure returned by `get_axes`:

```
sherpa> x = x.mid
```

We are now ready to plot the data. To do this, we use three functions from the *ChIPS* S-Lang module:

`chips_clear` to clear the plot window, `curve` to plot the curve, and `chips_redraw` to draw the plot:

```
sherpa> chips_clear
sherpa> () = curve(x,y,y_err)
sherpa> chips_redraw
```

The resulting plot  shows the datapoints and error bars. Although it is possible to customize the output of curve using the *ChIPS* state object, for now we accept the defaults.

---

## Defining a Source Model and Fitting

Next, we define our source model and fit the data. This time, we will create the desired model component first using `create_model` and then make it our source model using `set_source_expr`:

```
sherpa> () = create_model("polynom1d", "modell")
sherpa> () = set_source_expr("modell")
```

Then, we thaw the `c1` parameter (first–order coefficient in our polynomial) and fit, discarding the goodness–of–fit information that `run_fit` returns:

```
sherpa> () = set_thawed("modell.c1")
sherpa> () = run_fit
LVMQT: V2.0
LVMQT: initial statistic value = 2815.14
LVMQT: final statistic value = 151.827 at iteration 5
modell.c0  1.58227
modell.c1  0.198455
```

---

## Plotting the Fit

Finally, we wish to plot the resulting fit. First, we retrieve the data to plot using the `get_mcounts` function, which returns an array containing the y–values of the predicted data (i.e. an evaluation of the source model at every point on the independent axis):


```
sherpa> y = get_mcounts
```

Next, we customize the plot by setting the appropriate fields in the *ChIPS* state object. We choose to connect the data points with a simple, red line and remove the symbols marking individual datapoints:

```
sherpa> chips.curvestyle = _chips->simpleline
sherpa> chips.curvecolor = _chips->red
sherpa> chips.symbolstyle = _chips->none
```

Finally, we plot the curve and redraw the plot window:

```
sherpa> () = curve(x,y)
sherpa> chips_redraw
```

As expected, the plot window  now displays both the original data and the red best–fit line corresponding to our linear source model.

---

## Expanding the Possibilities with a Script

Although the plotting techniques described in the above sections are effective, *Sherpa* provides much simpler ways to create basic plots. For example, since the `sherpa_eval` function processes strings as if they were entered at the *Sherpa* command line, the easiest way to plot the above fit is to call the `L_PLOT` command:

```
sherpa> () = sherpa_eval("L_PLOT FIT")
```

However, the methods we have discussed enable us to perform more complicated tasks that are not possible with the standard *Sherpa* command set. Herein lies the real value of the *Sherpa* S-Lang module: It allows the user to extend *Sherpa*'s functionality as needed, without having to alter *Sherpa* itself.

The S-Lang script `polyfit.sl` provides a simple example of such an extension. Building on the methods used in this section, it performs three fits to the example dataset (using polynomials of order 1, 2, and 3) and plots the data and fits in a single pane, using a different color for each fit. The script uses two functions we have not yet discussed: `chips_color_value`, which takes a color name (e.g. "red") and returns the numeric value associated with that color (in this case, 6), and `chips_label`, which draws a label on the plot window:

```
unix% more polyfit.sl
% Load ASCII data and errors
variable dat = readfile("data1.dat");
() = set_axes(dat.col1);
() = set_data(dat.col2);
() = set_errors(dat.col3);

% Plot data and errors
variable x = get_axes().mid;
variable y = get_data();
variable y_err = get_errors();
chips_clear();
() = curve(x,y,y_err);

% Create source model
() = create_model("polynom1d","modell");
() = set_source_expr("modell");

% Make future curves simple lines without markers
% for individual data points
chips.curvestyle = _chips->simpleline;
chips.symbolstyle = _chips->none;

% Create array containing names of colors for fit plots
variable colors = [ "red", "green", "yellow" ];

% Store initial polynomial order
variable i = 1;

% Run fits for orders 1, 2, and 3 polynomials, and
% plot each fit in a different color

loop (3) {
  % Thaw coefficient for order i
  () = set_thawed("modell.c" + string(i));

  % Fit to order i polynomial
  () = run_fit();

  % Plot fit results, using color specified by element
```

## Sherpa S-Lang Module – Sherpa

```
% i-1 of "colors" array
y = get_mcounts();
chips.curvecolor = chips_color_value(colors[i-1]);
() = curve(x,y);

% Add a label for this fit, in the same color as the fit
% curve
() = chips_label(1.0, (3.6 - 0.15*i), "Order " + string(i),
chips.curvecolor, 1.5);

% Use order i+1 polynomial in next iteration
i++;
}


% Draw the plot
chips_redraw();
```

You can run the script as follows (after starting a new session or issuing the command "ERASE ALL"):

```
sherpa> () = evalfile("polyfit.sl")
LVMQT: V2.0
LVMQT: initial statistic value = 2815.14
LVMQT: final statistic value = 151.827 at iteration 5
  modell.c0  1.58227
  modell.c1  0.198455

LVMQT: V2.0
LVMQT: initial statistic value = 151.827
LVMQT: final statistic value = 59.0027 at iteration 4
  modell.c0  1.30826
  modell.c1  0.347303
  modell.c2  -0.0135317

LVMQT: V2.0
LVMQT: initial statistic value = 59.0027
LVMQT: final statistic value = 30.8491 at iteration 5
  modell.c0  1.49843
  modell.c1  0.1447
  modell.c2  0.0322936
  modell.c3  -0.00277729
```

The plot created  shows the data, the three fit curves, and three labels whose colors match the corresponding curves. Although this is a relatively simple application of the *Sherpa* module, it would *not* be possible using only standard *Sherpa* commands.

---

## A Note on `sherpa_eval`

The function `sherpa_eval`, which is part of the *Sherpa* S-Lang module, can be very useful in S-Lang scripts. It takes a string as its argument and interprets the string as a *Sherpa* command entered at the *Sherpa* prompt. This allows an application or script that imports the *Sherpa* module to execute *any* *Sherpa* command, regardless of whether it has a S-Lang equivalent.

However, when a *Sherpa* command *does* have a S-Lang equivalent, it is almost always preferable to use the S-Lang version, rather than passing the command string to `sherpa_eval`. In addition to being more efficient, *Sherpa* S-Lang functions generally provide S-Lang-scope output data (e.g. fit results) that are impossible to obtain when using `sherpa_eval`.

## Sherpa S–Lang Module – Sherpa

Also, note that `sherpa_eval` differs from the actual *Sherpa* command line in that one may execute only *Sherpa* commands, not *ChIPS* commands or S–Lang statements. To execute a *ChIPS* command in a S–Lang script, use `chips_eval`.

---

## History

14 Jan 2005 reviewed for CIAO 3.2: no changes

21 Dec 2005 reviewed for CIAO 3.3: no changes

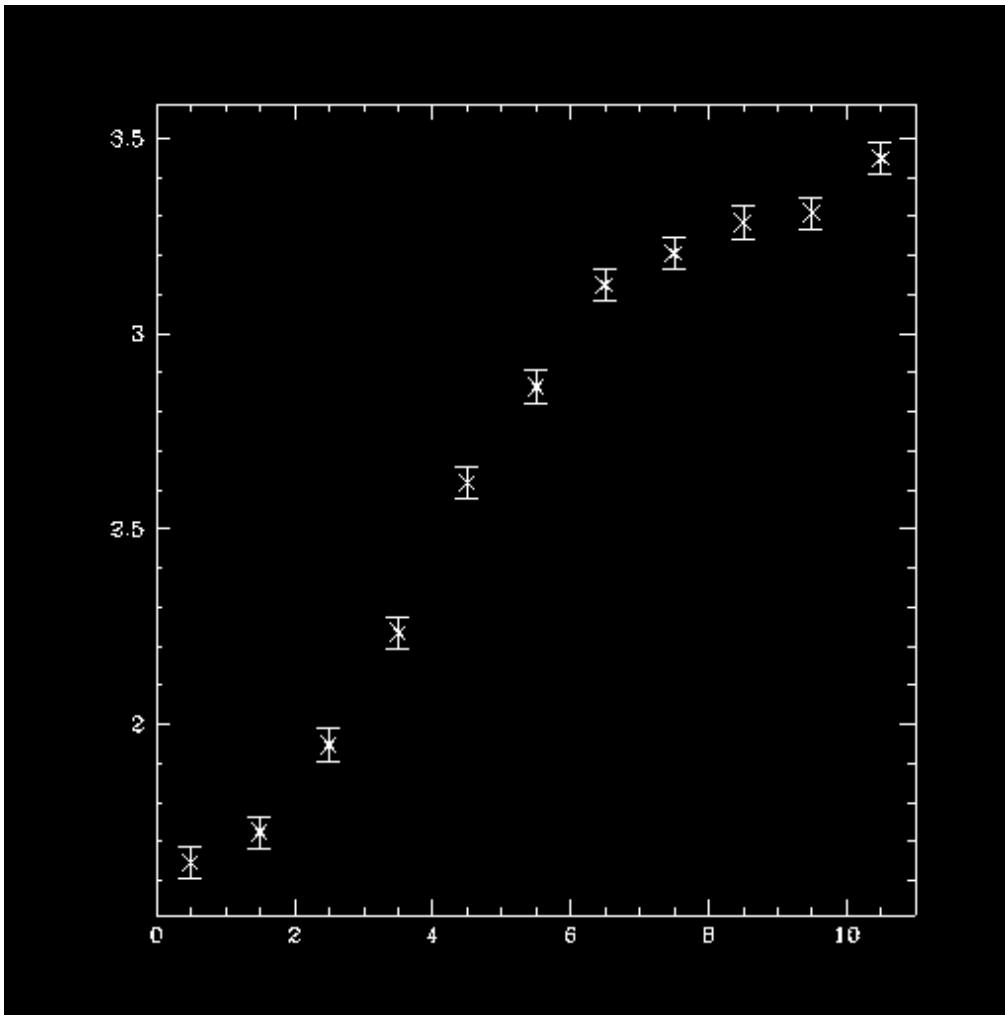
01 Dec 2006 reviewed for CIAO 3.4: no changes

---

URL: [http://cxc.harvard.edu/sherpa/threads/module\\_intro/](http://cxc.harvard.edu/sherpa/threads/module_intro/)

Last modified: 1 Dec 2006

Image 1: Plot of data retrieved by "get" functions



**Image 2: Plot of data and fit retrieved by "get" functions**

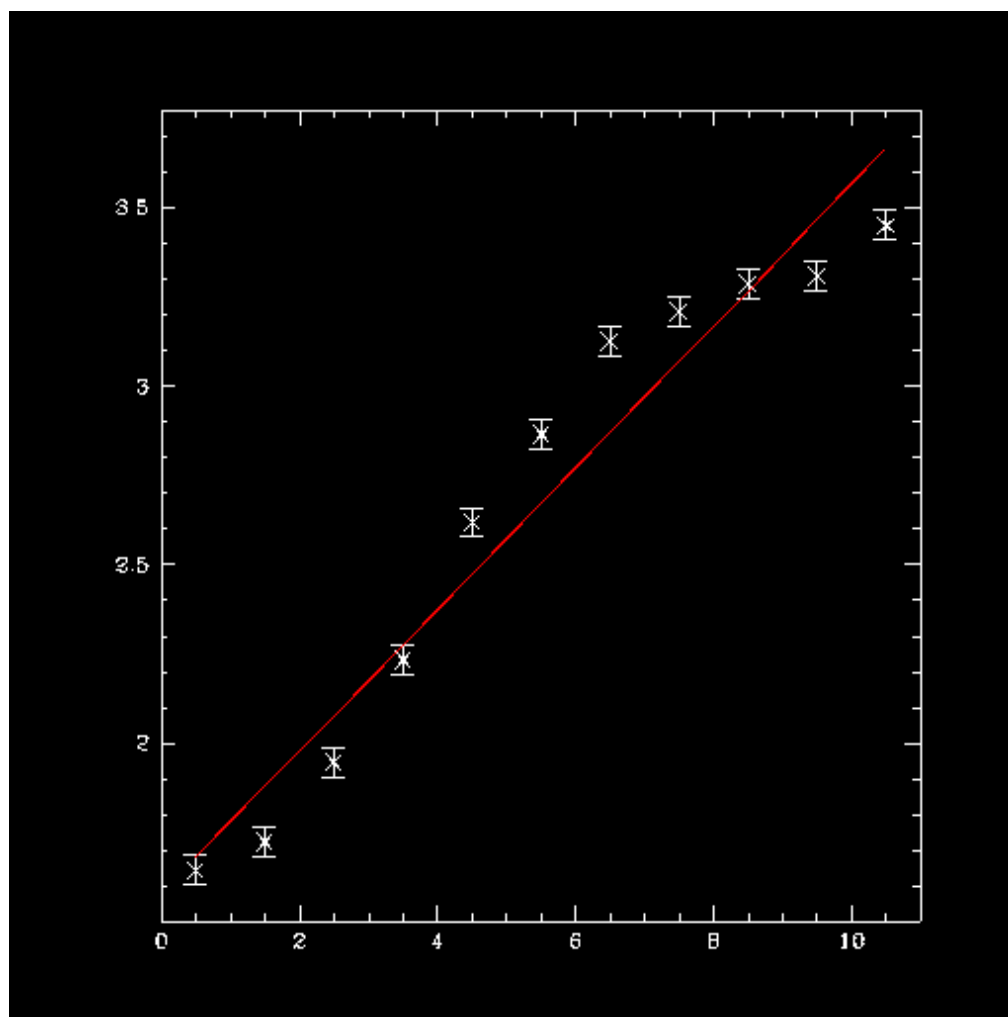


Image 3: Plot created by "polyfit.sl"

