URL: http://cxc.harvard.edu/ciao3.4/sherpa.html
Last modified: December 2006

AHELP for CIAO 3.4

sherpa

Context: sherpa

Jump to: Description Examples COMMAND−LINE OPTIONS Bugs

Synopsis

Command summary of Sherpa, CIAO's modeling and fitting engine.

Description

Sherpa is the generalized fitting engine of CIAO. Sherpa enables the user to fit models to data, particularly but not exclusively, to data that is being returned by NASA's Chandra X−ray Observatory. Sherpa features syntax that allows the user to construct complex models from simple definitions and to link parameters algebraically.

What commands does Sherpa understand?

As discussed below, Sherpa can be controlled from an interactive prompt, from a command file, or directly from S−Lang. For all but the last case the allowed syntax is given by the following list:

Commands allowed at the Sherpa prompt or in a Sherpa command file.

- Any valid Sherpa command. Note that Sherpa commands are case insensitive, and are generally listed in upper case in the documentation for clarity. For a list of all Sherpa commands, see "ahelp −c sherpa".
- Any valid ChIPS command.
- ahelp and about can be used to access the CIAO on−line help system. By default any help queries will be restricted to the "sherpa" context,
- A one−line S−Lang statement that does not need to end with a semi−colon and in which new variables do not need to be pre−declared. This allows you to enter "a = 23" rather than "variable a = 23;".
- Any line beginning with "$" or "!" is passed through to the shell (after removing this character). This allows you to run shell commands from the Sherpa prompt.
- The shell commands "pwd", "ls", and "cd" are available directly from Sherpa (so you do not need to say "!pwd").
- Any line beginning with the "#" character is ignored.

I. The command line

Sherpa may be launched by typing sherpa on the command line. This brings up the following welcome message −

```
unix% sherpa

------------------------------------------------------
Welcome to Sherpa: CXC's Modeling and Fitting Program
------------------------------------------------------
Version: 3.2

Type AHELP SHERPA for overview.
Type EXIT, QUIT, or BYE to leave the program.

Notes:
    Temporary files for visualization will be written to the directory:
    /tmp
    To change this so that these files are not deleted when you exit Sherpa,
    edit $ASCDS_WORK_PATH in your 'ciao' setup script.

    Abundances set to Anders & Grevesse

sherpa>
```

− and leaves you at an interactive prompt at which you can enter any valid Sherpa, as discussed above.

There are two ways to start Sherpa and get it to execute a pre−determined set of commands. If the ASCII file fit.shp contains text that could be entered at the Sherpa prompt then it can be processed by saying:

```
unix% sherpa fit.shp
```

This will cause Sherpa to evaluate all the commands in fit.shp and then leave you at the interactive prompt. If you want Sherpa to exit after processing the commands either add a QUIT statement to the end of the file or use the "−−batch" command−line option and say

```
unix% sherpa −−batch fit.shp
```

If you want to evaluate a S−Lang script − which can define useful functions or run through a set of commands − then you have to use the "−−slscript" option:

```
unix% sherpa −−slscript fit.s
```

Here the contents of fit.sl must be valid S−Lang code: variables must be pre−declared (unless you have set the _auto_declare variable to 1 at the start of the script) and statements require the closing semi−colon, but S−lang statements can be split across more than one line. After processing the S−Lang file the interactive prompt will be displayed (the "−−batch" option can be used to cause Sherpa to exit after processing the files).

Any Sherpa resource file will be loaded before any command−line file (either Sherpa or S−Lang format) is loaded.

You can load in both Sherpa and S−Lang files at the same time using

```
unix% sherpa −−slscript fit.sl fit.sh
```

This evaluates the file fit.sl and then the file fit.shp. Multiple S−Lang files can be specified using "−−slscript" but only one Sherpa command file. In fact, the command−line processing stops after the first Sherpa command file is found.

Note that the files can be called anything you like: we use the convention that the extensions "shp" and "sl" indicate Sherpa and S−Lang files respectively.

II. Sherpa Recovery

If for any reason, Sherpa exits with a nonzero status, a log file of the session is left behind in the user's home directory to assist in recovering the Sherpa session. This log file contains contains all commands that were successfully executed up to the point that Sherpa failed. In such a case, the name of the file will be $HOME/.sherpa−session−PID, where PID is the process ID of the Sherpa session that failed. For example, if the PID of a failed Sherpa session was 2048, then if the user types:

```
unix% sherpa ~/.sherpa-session-2048
```

then Sherpa will start and execute all the commands from that session, up to but not including the command that caused a failure.

When Sherpa exits normally, with an exit status of zero, no such log file is left behind.

III. Sherpa Module Functions

The scripting language S−Lang has been embedded into the CIAO system. Sherpa functionality can be easily extended with S−Lang scripts. Sherpa also now provides a loadable module, which can be loaded into other S−Lang applications at run−time (e.g., ChIPS, slsh). The Sherpa module has many functions that provide access to Sherpa data sets, and that invoke Sherpa functions. See the documentation on sherpa−module for a list of Sherpa/S−Lang module functions.

The syntax to load the Sherpa module into a S−Lang script is:

```
require("sherpa");
```

Note that this will automatically load in the ChIPS, Varmm, and XPA modules if they have not already been loaded.

IV. Customizing Sherpa

Sherpa can be customised by use of the Sherpa state object (also called customization variable) and the Sherpa resource file. The approach used is similar to that taken by ChIPS.

V. The Sherpa resource file

When a Sherpa session is started − either directly by the "sherpa" command in a S−Lang program − it looks for a Sherpa resource file. If found, the contents are processed at the start of the session.

The resource file must be in one of the following locations:

- the $SHERPARC environment variable
- $PWD/.sherparc
- $HOME/.sherparc

The search stops when the first match is made and Sherpa is launched, even if the chosen resource file contains an error.

The following is an example $HOME/.sherparc file that causes any application that starts Sherpa to print two messages, turns off the prompting for parameter values when a model is created, changes the optimization method to SIMPLEX, and defines a simple S−Lang function called "q" which allows you to exit Sherpa by just entering "q" (or "q()") at the Sherpa prompt.

```
message("Starting to process .sherparc")
paramprompt off
method simplex
define q () { () = sherpa_eval("quit"); }
message("Finished processing .sherparc")
```

Although the first and last lines of the above example create screen output for demonstrative purposes, it is recommended that the .sherparc file does not contain any command that creates either text or graphical output.

Since Sherpa also loads ChIPS, any customization applied using the ChIPS resource file (see "ahelp chips") will also be available. Note that the ChIPS resource file is executed before the Sherpa one.

Va. Format of the resource file

Since the resource file is a Sherpa (not S−Lang) script, it may contain simple, one−line S−Lang statements; this is in contrast to the Varmm resource file, which can contain any set of valid S−Lang statements. The "evalfile" command can be used to embed S−Lang function definitions, or other statements that require more than one line − such as setting up a S−Lang user model (see "ahelp slang usermodel") − into the resource file.

For example, if $HOME/slang/myblackbody.sl contains the slang_blackbody() function from the Examples section of "ahelp slang usermodel", then the following $HOME/.sherparc file will load it and make it available to Sherpa:

```
# find out the location of $HOME
variable home = getenv("HOME")
# load the S−Lang usermodel
() = evalfile( home + "/slang/myblackbody.sl" )
# register the model (this call must all be on one line)
()=register_model("slang−blackbody",["kT","ampl"],1,[1.0,1.0],[0.001,0.0],[100.0,1.e10],[1,1])
print("Loaded and registered slang_blackbody (S−lang usermodel)")
```

One obvious candidate is for setting fields in the Sherpa state object, which is discussed below. Note that the S−Lang function save_state() can also be used to make changes to the state object which will we recognised in new Sherpa sessions (see the discussion below and "ahelp save_state").

VI. Configuration of Sherpa with Sherpa State Objects

Sherpa has several state objects (e.g. configuration files) that control the appearance of plots and floating point numbers on the screen or the standard output, and the execution of the confidence levels calculations. These state objects are S−Lang variables that are created when Sherpa is started. In general the content of the state object can be displayed using print command. The command print(sherpa) lists state objects available in Sherpa.

```
sherpa> print(sherpa)
    plot            =  sherpa_Plot_State
    dataplot        =  sherpa_Plot_State
    fitplot         =  sherpa_FitPlot_State
```

```
    resplot          =  sherpa_Plot_State
    multiplot        =  sherpa_Draw_State
    output           =  sherpa_Output_State
    regproj          =  sherpa_VisParEst_State
    regunc           =  sherpa_VisParEst_State
    intproj          =  sherpa_VisParEst_State
    intunc           =  sherpa_VisParEst_State
    proj             =  sherpa_Proj_State
    cov              =  sherpa_Cov_State
    unc              =  sherpa_Unc_State
    con_levs         =  NULL
    modeloverride    =  0
    multiback        =  0
    deleteframes     =  1
    clobber          =  0
```

Most of the fields in the sherpa state object are actually references to other state objects (in such cases, instead of showing the value of the field, the print() function shows the type of variable to which that field refers). The following table lists each field, a description, and the default value:

Field Name	Description	Default
plot	LPLOT style for plots that are not data, fit or residuals plots	See "ahelp SHERPA.PLOT"
dataplot	Style for LPLOT DATA, LPLOT BACK only	See "ahelp SHERPA.DATAPLOT"
fitplot	Style for LPLOT FIT, LPLOT BFIT only	See "ahelp SHERPA.FITPLOT"
resplot	Style for LPLOT RESIDUALS, LPLOT RATIO, LPLOT BRESIDUALS, LPLOT BRATIO only	See "ahelp SHERPA.RESPLOT"
multiplot	Settings to apply when one or more plots are created in the ChIPS window (e.g., gap width between plots)	See "ahelp SHERPA.MULTIPLOT"
output	Appearance of floating point numbers printed to standard output (e.g., precision)	See "ahelp SHERPA.OUTPUT"
regproj	Region projection settings	See "ahelp SHERPA.REGPROJ"
regunc	Region uncertainty settings	See "ahelp SHERPA.REGUNC"
intproj	Interval projection settings	See "ahelp SHERPA.INTPROJ"
intunc	Interval uncertainty settings	See "ahelp SHERPA.INTUNC"
proj	Projection settings	See "ahelp SHERPA.PROJ"
cov	Covariance settings	See "ahelp SHERPA.COV"
unc	Uncertainty settings	See "ahelp SHERPA.UNC"
con_levs	The command CPLOT creates contour plots of 2D data, models, etc. (e.g., CPLOT DATA creates a contour plot of the data, CPLOT SOURCE creates a contour plot of the source model, and so on). This field allows users to set their own contour levels in such plots. If	NULL

	sherpa.con_levs is NULL, Sherpa automatically calculates default values for the levels plotted. If instead the user sets con_levs equal to an array of values (e.g., "sherpa.con_levs = [1,2,3]") then those levels are instead plotted by the CPLOT command. (The setting of the sherpa.con_levs field has no effect on plots generated with the REG−PROJ and REG−UNC commands.)	
modeloverride	If true, allow a model to be redefined without first having been erased (0 = false, 1 = true)	0
multiback	If true, allow multiple backgrounds per data set (0 = false, 1 = true)	0
deleteframes	The IMAGE command sends 2D images to ds9 for display. If sherpa.deleteframes is true, then Sherpa deletes all ds9 frames before sending the data to a newly created frame. If sherpa.deleteframes is false, then the frames that currently exist are left alone, and the data are sent to a newly created frame. (0 = false, 1 = true)	1
clobber	If true, allow output files to be overwritten if they already exist (0 = false, 1 = true)	0

The values of the fields of a state object can be changed at the command line, just as one would change the value of any S−Lang variable:

```
sherpa> sherpa.clobber = 1
```

Most of the fields of the sherpa state object are themselves references to other state objects. For example, sherpa.plot refers to a variable of type sherpa_Plot_State; this variable affects the appearance of plots created with the LPLOT command. The values of sherpa.plot can be displayed thus:

```
sherpa> print(sherpa.plot)
x_errorbars      =  0
y_errorbars      =  0
errs_style       =  bar
errs_type        =  both
x_log            =  0
y_log            =  0
curvestyle       =  step
curvecolor       =  default
symbolstyle      =  none
symbolcolor      =  default
symbolsize       =  2
xlabel_size      =  1.5
ylabel_size      =  1.5
zlabel_size      =  1.5
title_size       =  1.5
tickvals_size    =  1.5
prefunc          =  NULL
postfunc         =  NULL
```

Changing sherpa.plot fields as shown here:

```
sherpa> sherpa.plot.y_errorbars = 1
sherpa> sherpa.plot.curvestyle="simple"
```

will change the appearance of all subsequent plots created with the LPLOT command.

The state objects sherpa.plot, sherpa.dataplot, sherpa.fitplot, and sherpa.resplot each have fields x_log and y_log, which control whether or not the x− and y−axis are set to a linear or log scale. These fields can be changed on an

individual basis, as shown in the example above. Sherpa also provides functions to change to x_log and y_log fields of sherpa.plot, sherpa.dataplot, sherpa.fitplot, and sherpa.resplot all at once; these functions are called set_xlog(), set_ylog(), set_log(), set_xlin(), set_ylin(), and set_lin(). (See the ahelp files set_log and set_lin for more information.)

Similarly, sherpa.plot, sherpa.dataplot, sherpa.fitplot, and sherpa.resplot each have fields x_errorbars and y_errorbars, which control whether or not error bars are added to plots. The functions set_erron(), set_xerron(), set_yerron(), set_erroff(), set_xerroff(), and set_yerroff() turn error bars on and off, for all state objects at once. (See the ahelp files set_erron and set_erroff for more information.)

The user can also save the current values of all fields of all Sherpa state objects to a file, using the S−Lang function save_state():

```
sherpa> save_state()
sherpa> save_state("state1.shp")
```

In the former case, the values of the state objects are saved to the file $HOME/.sherpa−state−rc (which is overwritten every time save_state() is executed). If $HOME/.sherpa−state−rc exists, then the file is automatically read in whenever the user starts Sherpa. In the latter case, the state is saved in the file state1.shp; this file will not be automatically read in. The user can read in the file at any time during the Sherpa session. This can be useful if, for example, the user wishes to use two or more different plot styles.

VIa. Aliases for Sherpa State Objects

The names of the state objects, and names of their fields, are verbose. This can be good (as the names indicate the functions of the variables) and bad (as the user must type more at the comamnd line). Fortunately, it is possible for users to add aliases to their .sherparc files. Adding lines such as the following:

```
variable sp = sherpa.plot
variable sd = sherpa.dataplot
```

causes Sherpa to create new variables which are references to the state objects. Thus, instead of typing "sherpa.plot.x_log = 1", the user can now type "sp.x_log = 1".

VII. Using S−Lang data in Sherpa

Normally, Sherpa reads data from a file and then uses it for fitting, modelling, or data visualization. However, it is also possible to read data using Varmm S−Lang functions and then tell Sherpa to use these datasets. This is done by using Sherpa module functions to copy data from a S−Lang variable into Sherpa. Note that this behaviour is new to CIAO 3.0; in previous versions of Sherpa a much less flexible technique was used that is no longer supported.

The following example shows an ASCII dataset being read into the Varmm structure AGauss and then loaded into Sherpa:

```
  sherpa> AGauss = readfile("phas.dat")
  sherpa> () = load_dataset(1, AGauss)
  sherpa> lplot data
```

At this point the data is treated as if the dataset had been read in from a file using Sherpa's data command, as shown by the "lplot data" command. Data from S−Lang variables can also be copied into Sherpa as backgrounds or errors applied to the data (e.g., using load_backset(), set_errors() functions). For example, if you wished to use the Varmm "AGauss" as a background dataset you would say:

```
sherpa> () = load_backset(1,AGauss)
```

This ability also extends to response files (ie ARFs and RMFs), using the RSP model:

```
sherpa> spec = readpha("source.pha")
sherpa> rmf = readrmf("source.rmf")
sherpa> arf = readarf("source.arf")
sherpa> () = load_dataset(spec)
sherpa> () = load_arf("response", arf)
sherpa> () = load_rmf("response", rmf)
sherpa> instrument = response
```

If the input PHA file (here source.pha) has the necessary keywords set − ie BACKFILE, RESPFILE, or ANCRFILE − then the background, RMF and ARF will be read in using these files, without you having to specify them.

The examples section below shows how data in an ASCII file can be read into Sherpa using S−Lang.

VIII. Defining a Usermodel with S−Lang

Since CIAO 2.2, Sherpa has the ability to define models written in S−Lang that can then be used in the same manner as any of the in−built Sherpa models (such as bbody or beta2d). The "ahelp slang usermodel" document provides more information on this facility.

IX. Accessing MDLs from S−Lang

Sherpa can store the results of a fitting session in a Model Descriptor List (MDL; "ahelp mdl"). This data can be accessed from S−Lang using the following functions:

- mdl = update_mdl()
- mdl = get_mdl()
- mdl = get_mdl_data()
- mdl = get_mdl_models()

Note that prior to CIAO 3.0 the function names began with "sherpa_".

As indicated, all the routines return a S−Lang structure representing the MDL. If there are line models present, then update_mdl() should be used, to ensure that the flux values are recalculated for the given model parameters. If there are no line models, or the line fluxes are not of interest, then get_mdl() can be used instead, since it doesn't recalculate the line parameters, and so is faster.

The get_mdl_data() and get_mdl_models() functions only return part of the information stored in the MDL: the data "block" (ie the list of data, error, RMF and ARF file names) and the model "block" (ie the list of source models, and the parameter values/ranges) respectively.

If there are no source models defined, then update_mdl() and get_mdl() just return the data "block", as if get_mdl_data() had been called.

X. Memory Allocation

Sherpa itself doesn't have any hard−coded limits on memory allocation; it will use as much memory as the machine makes available. It does, however, inherit all the Data Model limits, since the DM is used for file I/O. Some of the commands − e.g. READ, DATA, BACK, and WRITE − can accept DM virtual file syntax, including "opt mem" which sets the maximum memory used for a virtual image. "ahelp dmopt" and the individual command ahelp files have details.

Example 1

Reading data from an ASCII file

If you wish to read in data from an ASCII file, and it has more than two columns, then you need to tell Sherpa which columns to use. The easiest way is to restrict the columns when reading them in to a Varmm structure using readascii().

The following example shows how this can be done. The test dataset (example.dat, listed below) contains three columns − x, y, and the error on y − which we read in using two calls to readascii(). The load_dataset() and set_errors() functions are then used to tell Sherpa what data and error values should be used. Once this is done, the data can be fit as with any dataset. Here we use the POLYNOM1D model to fit a line of the form "y=c0+c1*x" to the data.

```
sherpa> $cat example.dat
#    x      y  error
 1.000 −0.028  1.000
 2.000  1.581  1.414
 3.000  2.714  1.732
 4.000  4.368  2.000
 5.000  5.686  2.236
 6.000  6.263  2.449
 7.000  7.079  2.646
 8.000  7.898  2.828
 9.000  9.392  3.000
10.000  9.552  3.162
sherpa> dat = readascii("example.dat","1,2")
sherpa> () = load_dataset(1,dat)
sherpa> err = readascii("example.dat","1,3")
sherpa> () = set_errors(1, err.col2)
sherpa> lplot data
sherpa> paramprompt off
Model parameter prompting is off
sherpa> source = polynom1d
sherpa> thaw polynom1d.c1
sherpa> fit
 LVMQT: V2.0
 LVMQT:   initial statistic value = 36.6616
 LVMQT:   final statistic value = 1.29266 at iteration 6
     polynom1d.c0  −0.028
     polynom1d.c1  0.996091

WARNING:
   The value of polynom1d.c0 is equal to the polynom1d.c0.min limit boundary.
   You may wish to consider changing min/max values and refitting.
```

```
sherpa> polynom1d.c0.min = −5
sherpa> fit
 LVMQT: V2.0
 LVMQT:   initial statistic value = 1.29266
 LVMQT:   final statistic value = 0.494812 at iteration 3
     polynom1d.c0  −0.875496
     polynom1d.c1  1.15018

sherpa> lplot fit
```

Example 2

Using MDL files to store fit data

In this example we have already fit a simple model to a dataset. We can use update_mdl() − or get_mdl() in this case since the fit does not contain any line models − to look at the result:

```
...
sherpa> fitmdl = update_mdl()
sherpa> print(fitmdl)
_filename        =  baabaaazm
_path            =  /var/tmp/
_filter          =  NULL
_header          =  NULL
_ncols           =  25
_nrows           =  0
index            =  1
type2row         =  0
data             =  spec.pi
error            =  none
weights          =  none
rr_name          =  AutoReadResponse
rmf              =  spec.rmf
arf              =  spec.arf
back             =  spec_bg.pi
src              =  Integer_Type[8]
comp             =  Integer_Type[8]
sc               =  Integer_Type[8]
model            =  String_Type[8]
parname          =  String_Type[8]
parvalue         =  Float_Type[8]
parmin           =  Float_Type[8]
parmax           =  Float_Type[8]
frozen           =  String_Type[8]
method           =  String_Type[8]
statname         =  String_Type[8]
statval          =  Float_Type[8]
flux             =  Float_Type[8]
fluxerr          =  Float_Type[8]
filt_mod         =  String_Type[8]
lineid           =  String_Type[8]
```

Individual entries of the MDL structure can be examined using standard Varmm functions. Here we look at the list of models:

```
sherpa> printarr(fitmdl.model)
(cluster * galabs)
xsraymond[cluster]
xsraymond[cluster]
```

```
xsraymond[cluster]
xsraymond[cluster]
xsraymond[cluster]
xswabs[galabs]
xswabs[galabs]
```

The first entry gives the source model, the remaining lines are related to the parameters for the individual components, as indicated by the parname variable (we switch to the Varmm writeascii() function so as to print out more than one column per line):

```
sherpa> writeascii(stdout,fitmdl.model,fitmdl.parname)
(cluster * galabs)
xsraymond[cluster]
xsraymond[cluster] cluster.kT
xsraymond[cluster] cluster.Abundanc
xsraymond[cluster] cluster.Redshift
xsraymond[cluster] cluster.norm
xswabs[galabs]
xswabs[galabs]     galabs.nH
```

COMMAND−LINE OPTIONS

The command−line options for Sherpa − which can also be listed by entering "sherpa −help" − are:

Option	Description
−−batch	Runs in batch mode: the Sherpa welcome message is not displayed, any supplied code is run as if the ChIPS command "BATCH ON" were specified, and Sherpa exits after evalulating the code.
−h, −help, or −−help	Lists the command−line options.
−−slscript <filename>	Evaluates the S−Lang code in <filename>.

See the section titled "The command line" in the main discussion for more details on how ChIPS and S−Lang code can be automatically executed by ChIPS using these options.

Bugs

See the Sherpa bug pages online for an up−to−date listing of known bugs.

URL: http://cxc.harvard.edu/ciao3.4/sherpa.html
Last modified: December 2006