ISIS 1.0
Technical Manual

John C. Houck
WWW: `http://space.mit.edu/cxc/isis/`
Mailing List: isis-users@space.mit.edu

Revision 1.0

April 20, 2018

Chandra X-Ray Observatory Center
MIT Center for Space Research
One Hampshire St.
Building NE80
Cambridge, MA 02139–4307
USA

This document is part of ISIS, the Interactive Spectral Interpretation System
Copyright (C) 1998-2018 Massachusetts Institute of Technology

This software was developed by the MIT Center for Space Research under contract SV1-61010 from the Smithsonian Institution.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Overview

Contents

List of Figures

List of Tables

Preface

This manual describes ISIS, the MIT/CXC **I**nteractive **S**pectral **I**nterpretation **S**ystem which is designed to facilitate interpretation and analysis of high resolution X-ray spectra. It is being developed as programmable, interactive tool for studying the physics of X-ray spectrum formation, supporting measurement and identification of spectral features, and interaction with a database of atomic structure parameters and plasma emission models.

ISIS is written entirely in ANSI-C (with a few POSIX extensions) and is intended to be portable across most Unix operating systems. It has been used successfully on a variety of 32-bit and 64-bit processors and under a variety of Unix operating systems including Linux, FreeBSD, Solaris, SunOS, IBM/AIX, DEC/OSF1 and DEC/Ultrix systems. ISIS requires subroutine libraries from CFITSIO, PGPLOT and S-Lang, all of which are widely available as free software.

This manual is designed to serve as an introduction, user's guide, and definitive reference manual to the ISIS package. Please address any comments or suggestions to the ISIS user's mailing list `isis-users@space.mit.edu`.

- John Houck, April 20, 2018

We are continuing to develop ISIS *– more current information can be found at:*

`http://space.mit.edu/cxc/isis/`

Part I

Introduction to ISIS

Section 1

Introduction

1.1 ISIS in Context

High resolution X-ray spectra can contain a vast amount of information on the physical conditions in an emitting plasma. The richness of these data often lead the observer to examine plasma emission models and their fundamental atomic data in some detail. Until recently, the voluminous atomic data used to generate plasma emission models was often poorly documented and nearly inaccessible to most X-ray observers. Recent efforts by Smith, Brickhouse, Liedahl & Raymond (2001) are helping to improve this situation by making plasma emission models and atomic data available in a form which is more portable and accessible to observers.

ISIS is intended to support the analysis of high resolution X-ray spectra by combining in one package tools to query a database of atomic data and plasma emission models (e.g. that of Smith et al. 2001) with tools to manipulate and measure high resolution spectral data. These tools (called functions in the rest of this document) simplify low level operations such as file input, data plotting and database search and retrieval, allowing users to concentrate on higher level analysis issues. ISIS is also programmable and extensible, meaning that users can write scripts to simplify repetitive analysis tasks and can extend the ISIS command language by adding those scripts (or even user-supplied C programs) as new commands. S-Lang, the interpreted language which provides these features, also provides IDL-like array-based mathematical functions which greatly simplify common analysis operations.[1]

The primary purpose of ISIS is support of high resolution X-ray spectroscopy, but an important secondary goal in the design was to maximize the usefulness of the software by maximizing its portability. For this reason, ISIS is written in ANSI C and uses only widely available, free-software components. CFITSIO and PGPLOT have been widely used by the astronomy community for a number of years and are thoroughly tested and known to run on a wide range of computer systems. S-Lang is currently less well known but is fast, portable and provides an easy-to-use C-like syntax which we believe has a number of clear advantages over other scripting languages such as Perl, Python and Tcl.

[1] S-Langwas created by John E. Davis and has not been developed or maintained by contract funding; it is free software, available under the GNU Public License.

1.2 The Guilty Parties

The ISIS development team currently consists of the following individuals:

John C. Houck	MIT/CXC	Lead Scientist and Software Engineer
John E. Davis	MIT/CXC	Software Engineering Support
David Huenemoerder	MIT/CXC	Science Support
Dan Dewey	MIT/HETG	Science Support
Mike Nowak	MIT/CXC	Science Support
David S. Davis	MIT/CXC	Science Support

1.3 Acknowledgements

We gratefully acknowledge the aid and support of various members of the Chandra X-ray Observatory project including the Mission Support Team, the Calibration group, and the Data Systems group.

Section 2

Quick Start

2.1 Obtaining ISIS Source Code, Documentation and Help

ISIS source code and documentation may be obtained from the ISIS web page at

`http://space.mit.edu/cxc/isis/`

Please send comments, questions and bug reports to the mailing list `isis-users@space.mit.edu`.

2.2 Installing ISIS

Once source code is obtained from the web, detailed instructions on how to install the software may be found in the `INSTALL` file included with the distribution. Section 5 of this manual describes how to set up the spectroscopy database.

Note that only a single ISIS installation is required at a given site; there is no need to install it for each individual user. This also simplifies upgrades and minimizes use of disk space.

2.2.1 Spectroscopy Database Configuration

To access the spectroscopy database, ISIS must obtain the full path to the database directory and the names of all relevant database files (see §5). The person who installs ISIS should also ensure that a spectroscopy database is installed and that at least one configuration script is edited to accurately describe the database location and contents. See §5 for a detailed discussion.

2.3 Running ISIS

Assuming the ISIS executable exists somewhere on your the command search path, you can run ISIS by simply typing **isis** at the Unix prompt. The interactive help system provides documentation on ISIS intrinsic functions. Use `apropos` to find function names containing a certain substring:

```
isis> apropos("data");
```

and use `help` to obtain more detailed documentation on function usage:

```
isis> help("load_data");
```

Because these commands are used quite often, shortcuts are available (see §6.4).

About the semicolon (;) line-separator: The S-Lang scripting language requires lines to end with a semicolon (;). However, for interactive use, some people find it annoying to have to type a semicolon at the end of every line. Using the intrinsic variable `Isis_Append_Semicolon`, one can indicate whether or not ISIS should automatically supply this semicolon in interactive mode. By default `Isis_Append_Semicolon=0`, meaning that, in interactive mode, the user must end each line with a semicolon. To have isis automatically append the semicolon to each command line in interactive mode, set `Isis_Append_Semicolon=1`. The best way to make sure this value is automatically set every time you start ISIS is to set it in your `~/.isisrc` file:

```
Isis_Append_Semicolon = 1;
```

Although some users find it convenient to have isis automatically append the semicolon in interactive mode, this has the important drawback that one cannot reliably cut and paste into the interactive command line scripts which have multi-line S-Lang constructs such as loops or function definitions.

2.3.1 A Simple Example: Measuring the Flux in an Emission Line

For this example, we will use the sample data available from the ISIS source code distribution (see the `isis/test/data` subdirectory).

First, load the spectra:

```
isis> load_data ("acisf01318N003_pha2.fits.gz");
Reading: ............
Integer_Type[12]
isis> list_data;
Current Spectrum List:
 id    instrument part/m  src    use/nbins   A   R     totcts   exp(ksec)  target
  1     HETG-ACIS  heg-3   1    8192/ 8192   -   -  2.1000e+01    26.701  CAPELLA
  2     HETG-ACIS  heg-2   1    8192/ 8192   -   -  2.7000e+02    26.701  CAPELLA
  3     HETG-ACIS  heg-1   1    8192/ 8192   -   -  5.8410e+03    26.701  CAPELLA
  4     HETG-ACIS  heg+1   1    8192/ 8192   -   -  5.0670e+03    26.701  CAPELLA
  5     HETG-ACIS  heg+2   1    8192/ 8192   -   -  1.0300e+02    26.701  CAPELLA
  6     HETG-ACIS  heg+3   1    8192/ 8192   -   -  8.0000e+00    26.701  CAPELLA
```

```
  7     HETG-ACIS  meg-3   1    8192/ 8192   -   -  9.0000e+02    26.701  CAPELLA
  8     HETG-ACIS  meg-2   1    8192/ 8192   -   -  8.6200e+02    26.701  CAPELLA
  9     HETG-ACIS  meg-1   1    8192/ 8192   -   -  2.5683e+04    26.701  CAPELLA
 10     HETG-ACIS  meg+1   1    8192/ 8192   -   -  2.0369e+04    26.701  CAPELLA
 11     HETG-ACIS  meg+2   1    8192/ 8192   -   -  6.3900e+02    26.701  CAPELLA
 12     HETG-ACIS  meg+3   1    8192/ 8192   -   -  5.7000e+02    26.701  CAPELLA

isis>
```

ISIS prints a dot (.) as each spectrum is read from the data file. By default, ISIS loads all the data in the specified file. By specifying the row index, it is also possible to load a single spectrum from a Type II PHA file. Once the data is loaded, `list_data` shows a list of data sets. Each data set is identified by an index (**id**); as we will see below, this index is used to refer to an individual data set. The columns labeled **part/m**, and **src** indicate the spectrum "part" (e.g. the HEG is `tg_part=1` and the MEG is `tg_part=2`), the dispersion order (**m**), and the source index (**src**). The next two columns show the number of bins in the data set (**nbins**) and the number currently noticed for fitting (**nbins**). The columns labeled **A** and **R** indicate the index of the assigned ARF and RMF respectively; a dash (-) indicates that no response function has been assigned. The last four columns give the total number of counts in the spectrum (**totcts**), the exposure time (**exp**), a plot color index (**clr**) and the target name.

For brevity, the above example listing was generated with the global variable `Isis_List_Filenames=0`. By default, this variable is non-zero, causing the names of associated spectrum and background files to be printed along with the other information for each dataset.

The line `Integer_Type[12]` indicates that the **load_data** function returned a 12 element integer array (containing the indices of the data sets loaded). Because this array was not stored in a variable and was not explicitly ignored, ISIS handled the return value automatically. This is an important and useful feature of S-Lang which will be discussed in more detail later on.

To plot the counts histogram from the minus first-order MEG spectrum, use **plot_data_counts**:

```
isis> plot_data_counts (9);
 Graphics device/type (? to see list, default /NULL): /xw
isis>
```

PGPLOT prompts the user for a plot device, defaulting to the `/NULL` device if no default has been specified using `PGPLOT_DEV` environment variable variable. Here, we selected **/xw** to plot to the screen using the X-windows.

Because the plot command `plot_data_counts` is somewhat cumbersome to type, you may wish to define a shorter alias:

```
isis> alias ("plot_data_counts", "pdc");
```

Now clear the plot and re-draw it using this new command alias:

```
isis> clear;
isis> pdc (9);
```

Next we load the ARF and assign it to the appropriate data set:

```
isis> () = load_arf ("acisf01318_000N001MEG_-1_garf.fits.gz");
isis> list_arf;
Current ARF List:
 id grating detector part/m  src   nbins  exp(ksec)  target
  1    HETG     ACIS meg-1    0     8192    28.12  CAPELLA
file:  acisf01318_000N001MEG_-1_garf.fits.gz
isis> assign_arf(1,9);
isis>
```

Note that here we have explicitly ignored the return value from the **load_arf** function and that the **assign_arf** function did not return anything; function return values are normally used either to return requested information or to provide a way of testing whether or not the function call succeeded. The latter values are especially useful in scripts but may often be ignored in interactive mode. Once the ARF is loaded into the internal list, the updated list of ARFs is displayed showing information similar to that in the list of data sets.

Now load the RMF and assign it to the same data set:

```
isis> () = load_rmf ("acismeg1D1999-07-22rmfN0002.fits.gz");
isis> list_rmf;

Current RMF List:
 id grating detector  type   file
  1     MEG   ACIS-S  file:  ... acismeg1D1999-07-22rmfN0002.fits.gz

isis> assign_rmf(1,9);
isis>
```

Now, let's focus on a single emission line and measure the flux in the line. First, change the X-axis range to display the 12-13 Åregion and re-draw the plot:

```
isis> xrange(12.0,13.0);
isis> pdc (9);
```

When fitting models to data, ISIS fits all the currently loaded data sets simultaneously. Because we want to fit only a single Gaussian to a single spectrum, we must first ignore all the other data sets and notice only the wavelength range we want to fit:

```
isis> ignore ([ [1:8], [10:12] ]);
isis> xnotice (9, 12.05, 12.2);
```

The **ignore** function takes an array of data-set indices to ignore. This function can also take an optional wavelength range which applies to all the indicated data sets; if no wavelength range is specified, the entire data set is ignored. The **xnotice** function e**x**clusively-**notice**s the specified range of dataset 9.

To fit a single Gaussian, define the fit-function as

```
isis> fit_fun ("gauss(1)");
```

The integer index indicates which instance of the Gaussian is used; one could fit three Gaussians using `fit_fun("gauss(1)+gauss(2)+gauss(3)");`. To list the fit parameters, use **list_par**

```
isis> list_par;
gauss(1)
 idx  param          tie-to  freeze     value      min      max
  1  gauss(1).area     0      0             0        0        0
  2  gauss(1).center   0      0             0        0        0
  3  gauss(1).sigma    0      0             0        0        0
isis>
```

This listing shows the function definition on the first line and then lists the fit parameters, indicating which parameters are linked or frozen, the parameter values and the allowed parameter ranges. Setting the `min` and `max` values to zero indicates that the corresponding parameter value is unconstrained.

To set the parameter values from the command line, one could use either **set_par** or **edit_par**. Although it is a matter of personal preference, the **set_par** is generally more useful in scripts and **edit_par** is often more useful in interactive mode. The **edit_par** function loads the parameter list into the editor specified by your `EDITOR` environment variable (or `vi` by default). After using the editor to enter the necessary information, save the file and exit the editor. Here, so that we can show the screen interaction explicitly, we use **set_par** to provide initial parameter values:

```
isis> set_par(1,100);
isis> set_par(2,12.15);
isis> set_par(3,0.03);
```

Several different versions of the syntax are supported; which is most useful depends on the circumstance.

For example, we could have used

```
isis> set_par ("gauss(1)", [100, 12.15, 0.03]);
```

This form is useful because it takes only one line and doesn't require knowing the parameter indices, but it requires you to remember the order in which the parameters are defined. Alternatively, we could have used this syntax:

```
isis> set_par ("gauss(1).area", 100);
isis> set_par ("gauss(1).center", 12.15);
isis> set_par ("gauss(1).sigma", 0.03);
```

This doesn't require knowledge of the parameter indices, but it requires more typing and requires correct spelling of the parameter names.

Once the parameters have been set, list the current values using `list_par`:

```
isis> list_par;
gauss(1)
 idx  param          tie-to  freeze         value          min          max
  1  gauss(1).area     0      0               100            0            0
  2  gauss(1).center   0      0             12.15            0            0
  3  gauss(1).sigma    0      0              0.03            0            0
```

In interactive mode, one can do this in yet a different way by using the function **ifit_fun** which reads the plot cursor to initialize the fit parameters (see §7).

Because the specified normalization is probably not even close to the correct value, we'll use **renorm_counts** to obtain a better initial value before doing the fit:

```
isis> renorm_counts;
 Parameters[Variable] = 3[3]
            Data bins = 30
           Chi-square = 1215
   Reduced chi-square = 45
isis>
```

The screen output shows how many parameters are being fit, how many are variable, the number of data bins being fit and gives the resulting χ^2 fit-statistic. Now find the best fit, allowing all the parameters to vary:

```
isis> fit_counts;
 Parameters[Variable] = 3[3]
            Data bins = 30
  Best fit chi-square = 28.86
   Reduced chi-square = 1.069
isis>
```

To overlay the best-fit model, use the **oplot_model_counts** function. As before, we can define an alias to save typing:

```
isis> alias ("oplot_model_counts", "opmc");
isis> opmc (9);
```

To save the fit-results in a file one can either use the default ISIS parameter file format or one can make use of the low-level S-Lang functions to save the values of interest in a different, user-defined format. To save using the default parameter file format:

```
isis> save_par("gfit.p");
```

The default format is exactly the same as the screen listing generated by `list_par`. The main advantage to saving the fit-function and its parameters in this format, is that the values can be restored using **load_par**.

To show how to save the fit-results using a user-defined format, suppose we want to save the best-fit flux along with 90% confidence limits and that we want the wavelengths listed in the first column and the line width in the second column. Here's how to do that:

First, compute the necessary confidence limits using **vconf**:

```
isis> (amin, amax) = vconf(1);
Finding best fit
Best fit par=  1.04961248e-03: chisqr= 2.88578026e+01
Allowed parameter range is currently unrestricted.
Please specify finite min/max values for parameter confidence limit search.
```

Oops – we forgot to specify the range of parameter values for the confidence limit search. Use the optional arguments of **set_par** for that:

```
isis> a = get_par(1);
isis> set_par(1, a, 0, 0.9*a, 1.1*a);
```

Here, we used the **get_par** function to get the exact parameter value rather than typing it again from the screen or from the saved parameter file. We then chose parameter value ranges of $\pm 10\%$. Now compute the confidence limits, saving the values in variables `amin` and `amax`:

```
isis> (amin, amax) = vconf(1);
Finding best fit
Best fit par=  1.04961248e-03: chisqr= 2.88578026e+01
Minimizing for par=  9.97151856e-04
par=  9.97151856e-04  chisqr=  3.0389e+01  dchisqr=  1.5316e+00
Minimizing for par=  9.70921544e-04
par=  9.70921544e-04  chisqr=  3.2304e+01  dchisqr=  3.4462e+00
Minimizing for par=  9.87731365e-04
par=  9.87731365e-04  chisqr=  3.0989e+01  dchisqr=  2.1311e+00
Minimizing for par=  9.80331268e-04
par=  9.80331268e-04  chisqr=  3.1529e+01  dchisqr=  2.6711e+00
Minimizing for par=  9.79859377e-04
par=  9.79859377e-04  chisqr=  3.1565e+01  dchisqr=  2.7076e+00
limit found:  9.79859377e-04  dchisqr=  2.7076e+00
Minimizing for par=  1.39913896e-03
par=  1.39913896e-03  chisqr=  9.6803e+01  dchisqr=  6.7945e+01
Minimizing for par=  1.11941726e-03
par=  1.11941726e-03  chisqr=  3.1568e+01  dchisqr=  2.7104e+00
limit found:  1.11941726e-03  dchisqr=  2.7104e+00
isis>
```

Note that the screen output from **vconf** is verbose; use **conf** to do a "quieter" confidence limit search.

Now that we have our confidence limits, we can save these values in a custom file format:

```
isis> fp = fopen ("gfit.txt", "w");
isis> () = fprintf (fp, "Center  Sigma   Flux (F_lower, F_upper)\n");
isis> () = fprintf (fp, "%10.7f %12.4e  %12.4e  (%9.4e, %9.4e)   \n",
              get_par(2), get_par(3), get_par(1), amin, amax);
isis> fclose(fp);
```

The first line opens the file `"gfit.txt"` for writing (that's what the `"w"` is for). The second line prints some informative column headers (not aligned with the data columns) and the third prints the data values using a syntax which should be very familiar to C programmers; e.g. `"10.7"` indicates a field of width 10 with 7 digits of precision and the `f` and `e` format specifiers indicate decimal and scientific notation formats respectively. Note that we have explicitly ignored the status return values from the **fprintf** function. Similarly, one can write a short function to load values from ascii columns, but it may be simpler to use the built-in functions `readcol` and `writecol` to read and write ascii files with columns of numbers. We could also have stored our fit results in a FITS binary table using the `cfitsio` module.[1]

Plots of 2D confidence contours may be generated using `conf_map_counts` or `conf_map_flux` and `[o]plot_conf`.

[1] For information on the various modules available, see `http://space.mit.edu/cxc/software/slang/modules/`

Section 3

Programming ISIS

3.1 Interactive Scripting

A convenient way to use ISIS for spectral analysis is to write one or more scripts, usually in the form of several useful functions, and to iteratively edit and run these scripts to achieve the desired result. This may sound cumbersome, but in practice often works quite efficiently, because the ISIS intrinsic functions already handle a number of tedious tasks such as manipulating complex data files and searching the spectroscopy database.

For example, one might write one script to load several available data sets and to set up the spectroscopy database. Once that script exists, one can start off later analysis sessions by just running the setup script.

Similarly, one might write another script to carry out a few standardized analysis steps, independent of which data set is loaded. Repeating those analysis steps for several data sets might then be a matter of running the analysis script several times.

Some example scripts may be found on the ISIS web page; several usage examples are discussed in §4.

In cases where a large, specialized task is currently unsupported in ISIS, it is possible for users to extend ISIS by linking in other subroutine libraries, *without* editing *any* of the ISIS source code (see §3.3). This allows large specialized tasks to be handled in compiled code, yet still take advantage of the existing ISIS functions.

3.2 Scripts

To run an ISIS script from the Unix shell, supply the path to the ISIS script on the Unix command line:

```
unix> isis path/script.sl
```

when the script finishes, control will return to the ISIS prompt. To have control return to the

Unix shell when the script finishes, use the `--batch` option or use `exit(num)` or `quit` as the last line of the script.

It is also possible to create executable scripts similar to Unix shell scripts. If the first line of the script contains something like

```
#! /usr/bin/env isis
```

and the script is made executable, e.g.

```
chmod +x path/script.sl
```

then the script can be executed like any other unix command, by simply typing the name of the script at the unix prompt:

```
unix>  path/script.sl
```

In this case, the operating system uses the program specified on the first line of the script, namely ISIS, to interpret the rest of the file.

To run a script during an interactive ISIS session, use

```
isis> .load path/script.sl
```

When the script finishes, control will return to the ISIS prompt (unless `script.sl` contains the `quit` command). Note that this shortcut syntax works only in interactive mode; in a script one must use

```
()=evalfile("path/script.sl");
```

The `.source` short-cut is similar to `.load` except that the script is executed exactly as though each line were entered in interactive mode. This provides a useful mechanism for replaying ISIS log-files.

At startup, ISIS searches for the file `$HOME/.isisrc` (where `HOME` is the path to the user's home directory). If this file exists, ISIS attempts to interpret it as a S-Lang script. Any functions which should run at ISIS startup for, *e.g.* initialization and user-customization should be invoked in this file. For example, to have the spectroscopy database loaded automatically, one could add this line to the `.isisrc` file:

```
plasma(aped);
```

As written, this example assumes the existence of the `aped` data structure. Alternatively, one may define convenient aliases for often used functions (see `alias()`). Site-wide ISIS customizations may be added and documented using the files `etc/local.sl` and `doc/local_help.txt` in the ISIS source code directory (see §2).

If it exists, the configuration file `$HOME/.isisrc` is automatically loaded before executing any scripts specified on the ISIS command line. To prevent loading the configuration file, use the `-n` option on the command line:

```
unix> isis -n path/script.sl
```

To load an alternate configuration file, use the `-i` option:

```
unix> isis -i ~user/.isisrc
```

Scripts may also take command-line arguments using a mechanism familiar to C programmers. This feature provides a convenient way for a script to obtain needed parameters from the command line (e.g. the names of input and output files), allowing scripts to be run without ever entering the ISIS interactive mode. The command-line is available as an array of strings (`__argv`) of length `__argc`. The command line arguments are most conveniently handled when the first line of the script is

```
#! /usr/bin/env isis
```

For example, consider this script:

```
#! /usr/bin/env isis
define isis_main ()
{
   vmessage (" _debug_info = %d", _debug_info);
   vmessage ("__argc = %d", __argc);
   message ("__argv:");
   print (__argv);
}
```

as command-line arguments (note that `_debug_info` is a S-Lang intrinsic variable). Invoking this script (named 'doit') with two command-line arguments yields

```
unix> ./doit a.fits b.txt
 _debug_info = 0
__argc = 3
__argv:
"/tmp/doit"
"a.fits"
"b.txt"
```

As expected, the command-line array has size `__argc=3`.

3.3 Extending ISIS

It is possible to add functionality to ISIS without modifying the ISIS source code. This can be accomplished in several ways:

One way is to define new S-Lang functions using the existing ISIS and S-Lang intrinsics. If these S-Lang function definitions are placed in a file named e.g. `myfunctions.sl`, they can easily be made available at the ISIS prompt

```
isis>  ()=evalfile("myfunctions.sl")
isis>  .load myfunctions.sl                 % simpler equivalent
```

In fact, much of the ISIS command line interface is implemented this way.

Another way to add new functions is to define new intrinsics in C, compile that code into a dynamically linked library (named e.g. `myfunctions-module.so`) and then import that library at ISIS run-time using the S-Lang `import` function:

```
import ("myfunctions")
```

This method is necessarily restricted to machines which support ELF standard dynamic linking (e.g. Solaris 2.x and Linux provide support, but SunOS 4.x does not). Most XSPEC functions can already be imported in this way – see `modules/xspec/src/xspec-module.c`. See the S-Lang documentation for more information on dynamic linking.

Finally, several parts of ISIS support user-defined alternatives. In particular, in fitting models to data, ISIS provides a number of user-definable operations including user-defined fit-functions, fit-kernels, instrument responses (ARFs and RMFs), fit-statistics and minimization algorithms. Some of these may be implemented in both C (or other compiled language) and S-lang while others support only compiled-language or only S-Lang implementations. For details on adding user-defined fit-functions, see `add_slang_function` or `add_compiled_function`. To add a user-defined fit-kernel, see `load_kernel`. To add a user-defined minimization algorithm or fit-statistic, see `load_fit_method` and `load_fit_statistic`, respectively. Although S-Lang is very efficient, computationally-intensive extensions may be best implemented in C or Fortran.

The SLIRP code generator makes it very easy to add arbitrary compiled codes to ISIS, including new models. It can dramatically reduce the time and effort required to make C, C++, or FORTRAN code callable directly from ISIS, automatically vectorize functions to take advantage of the powerful numerical and array capabilities native to S-Lang, and can even generate Makefiles to automate the build process.

3.4 Language Syntax

The ISIS command language syntax is the syntax of S-Lang, its C-like embedded scripting language. Although ISIS can be used without becoming an expert with S-Lang, it will be helpful to understand something of how the scripting language works and, in particular, how to use its array-manipulation syntax.

All command lines must end with a semicolon (;) (but see the discussion in §2.3 about their use in interactive mode). Multiple commands are allowed on the same line and command phrases may also extend over multiple lines. For example, try:

```
isis> message("testing...." +
      string(1 + 2 + 3));
```

In this example, the first line was ended with a carriage return.

The comment character is the percent symbol (%); all characters following a percent symbol (%) on a given line are ignored.

Arrays and complex data structures may be defined and manipulated (see the S-Lang documentation for details). S-Lang array indexing syntax is similar to that used in IDL. Array elements are indexed starting at zero; `x = [ 1, 2, 3, 4 ]` defines `x` as array of 4 values with indices 0, 1, 2, 3 so that, e.g. `x[1] = 2`. The colon shorthand notation is used to specify ranges, *e.g.* `[ 2:7 ]` = `[ 2, 3, 4, 5, 6, 7 ]`. This notation is quite powerful and allows many array manipulation tasks be to be expressed simply and clearly. For example, to reverse the order of elements in an array x:

```
x = [2:7];          <---- this means x = [2, 3, 4, 5, 6, 7]

n = length(x);      <---- n=6  (because x has 6 elements)

reversed_x = x[[n-1:0:-1]];

        % result is --->  reversed_x = [x[n-1], x[n-2], ..  x[0]]
        %                              = [   7,     6,    ..   2  ]
```

this order reversal arises e.g. when converting from a wavelength grid in increasing order to an energy grid in increasing order.

An unusual aspect of the syntax involves functions that return multiple values. For example, if a function returns three values, the return values can be assigned to specific variables using this syntax:

```
variable x,y,z;
(x,y,z) = some_function ();
```

Functions may also return values through the function argument list, using syntax which should be familiar to C programmers:

```
variable x,y,z;
a_function (&x, &y, &z);
```

In scripts, variables must be declared before use as above, but in interactive mode, variables are declared automatically.

Return values may also be discarded. To save only the second return value from the above function while discarding the first and third, use this syntax:

```
(,y,) = some_function();
```

For functions which return only one value, the syntax is more familiar:

```
x = returns_one_value();
```

To explicitly discard the single return value, use

```
() = returns_one_value();
```

In interactive mode, it rarely matters to the user that S-Lang is a stack-based language. This is primarily because handling of function return values is simplified in interactive mode: the stack is automatically emptied after each function executes and any return values not explicitly handled are simply popped off the stack and printed on the screen. For example:

```
isis> x = [47, 12, 13, 92, 46, 12];        % define an array
isis> y = max(x);                          % now y = 92
isis> max(x);               % When the return value isn't "caught",
```

```
92                          % it is popped off the stack automatically.
```

This is true *only* in interactive mode, however.

When writing scripts, one must be more careful to explicitly handle all function return values. When a function is invoked without explicitly handling all of its return values in some way, the remaining return values are placed on the stack. The next function call which pops arguments off the stack will get those values as input. This feature can be exploited to improve the efficiency of S-Lang scripts, but can also lead to unexpected results if incorrect arguments are unintentionally passed to a function because the return value from a previous function call was not handled correctly. Using a somewhat pathological example, the operation of the stack can be seen in interactive mode

```
isis> [47, 12, 13, 92, 46, 12]; max();
92
```

In this example, an array is first pushed onto the stack, then the `max()` function pops an array argument off the stack, finds the largest array element, and pushes that value back onto the stack. Because this is an interactive mode example, the value remaining on the stack is then automatically printed on the screen.

Don't be concerned if this sounds a little complicated. As stated above, in interactive mode, the presence of the stack is usually an unimportant detail. The main reason for mentioning it here is to provide a short introduction to the language structure and suggest ways in which it can be exploited.

For full details, see the S-Lang documentation at

`http://www.jedsoft.org/slang/`

or in the S-Lang source directory.

Section 4

Common ISIS Tasks

This section provides several examples to illustrate how to carry out common analysis tasks using ISIS. These examples are intended to provide a starting point and to give the user an idea of how ISIS can be used. They do not illustrate everything ISIS can do and do not provide a complete description in any sense. For full details on all ISIS functions, see the ISIS Function Reference in chapter 7.

The following examples use sample data files which are available from the ISIS source code distribution (see the `isis/test/data` subdirectory).

1. a Type II pha file called `acisf01318N003_pha2.fits` which contains Chandra/HETG counts spectra for several plus and minus diffraction orders of both HEG and MEG.
2. an Effective Area function (ARF) called `acisf01318_000N001MEG_-1_garf.fits`
3. a Redistribution Matrix function (RMF) called `acismeg1D1999-07-22rmfN0002.fits`
4. a spectral model definition file called `model.dat` (see `load_model` for a description of the format).
5. a S-Lang script (`aped.sl`) describing the layout of the spectroscopy database (see §5).
6. the spectroscopy database itself.

The S-Lang scripts used to generate the figures in this manual are in the `doc/scripts` directory. For instructions on how to obtain and install the spectroscopy database, see §5

4.1 Reading and displaying spectral data

ISIS can read data in either FITS (Type II pha) or ASCII format. To read a Type II pha file and display selected spectra (see Figures 4.1 and 4.2):

```
variable dir = _isis_srcdir + "/../isis-examples/data";

% load data file containing more than 9 spectra
```

```
() = load_data (dir + "/acisf01318N003_pha2.fits.gz");
() = load_arf (dir + "/acisf01318_000N001MEG_-1_garf.fits.gz");
assign_arf(1,9);
flux_corr(9,2);   % flux-correct down to 2-sigma significance level

variable id=open_plot ("data_cts.ps/vcps");
resize(15);
plot_data_counts (9);          % plot spectrum #9
close_plot (id);

id=open_plot ("data_flx.ps/vcps");
resize(15);
xrange(8,25);
plot_data_flux (9);          % plot spectrum #9
close_plot (id);
```

Data plots use a linear axis scale by default; to make a log plot, use `xlog` and/or `ylog` – `xlin` and `ylin` will restore the linear axis scaling.

```
() = open_plot("/xwin");
xlog; ylog;
plot_data_counts (sp);       % plot log-log
```

To change the axis ranges, use the `xrange` and `yrange` commands. Continuing the example:

```
erase;                       % clear the plot window
xlin; ylin;                  % use a linear scale
xrange (11,12);              % set X-range to 11-12 angstrom
yrange;                      % take Y ranges from the data
plot_data_counts (sp);       % plot data
```

The plot coordinates can be changed (e.g. to plot vs energy in keV instead of wavelength in Å) using the `plot_unit` command.

```
erase;                      % clear the plot window
plot_unit ("kev");          % plot in energy units
plot_data_counts (sp);      % plot data
```

Although the current version of ISIS supports analysis of a flux-corrected spectrum, it does not automatically compute the flux-corrected spectrum from the counts histogram. The flux-corrected spectrum may be provided in the input data file or may be computed within ISIS after loading the appropriate effective area (ARF) file:

```
% load one ARF, return value is arf=1
arf = load_arf ("acisf01318_000N001MEG_-1_garf.fits.gz");

assign_arf (arf, sp);          % assign the ARF to the current data set
flux_corr (sp, 2);             % flux-correct, ignoring
                               % bins with S/N < 2
```

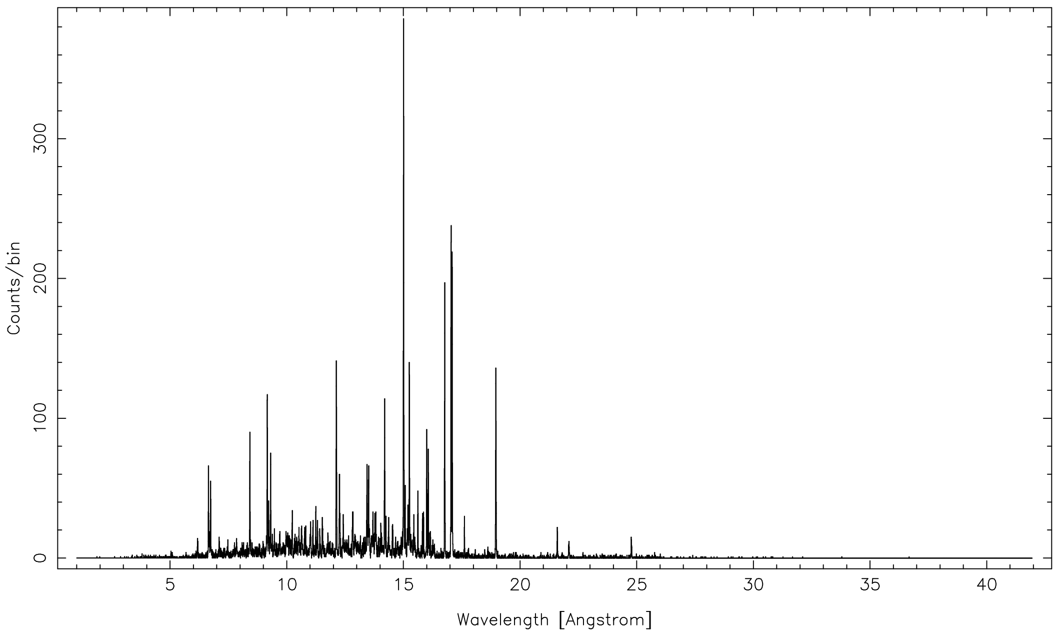

Figure 4.1: Example plot of observed counts per bin vs. wavelength

The flux-corrected result is stored separately and does not over-write the counts histogram. Note that because the S/N threshold was set to a non-zero value (the second argument to `flux_corr`), bins with very few counts above 25 Å are assigned a flux of zero in Figure 4.2; see the description of `flux_corr` in the reference section for more detail.

4.2 Initializing the spectroscopy database

To initialize the spectroscopy database, ISIS must first obtain the location and contents of the database. The person who installs ISIS should provide a S-Lang script (perhaps by editing one of the included example files) which describes the local version of the spectroscopy database. After installation, it is not necessary for casual users to access this database configuration script directly.

See the `etc/local.sl` file in the ISIS source distribution for an example of how the necessary information can be provided by a configuration script which is automatically loaded when an associated initialization function is referenced. For example, to load the spectroscopy database described by configuration script `etc/aped.sl`, calling the `aped` function as an argument to `plasma`:

```
plasma (aped);
```

causes the associated configuration script (`aped.sl`) to load and execute automatically, providing paths to the atomic data and the emissivity tables (if available). The `plasma` function then loads these database tables into ISIS. Similarly, one can load the atomic data tables only using the `atoms` function:

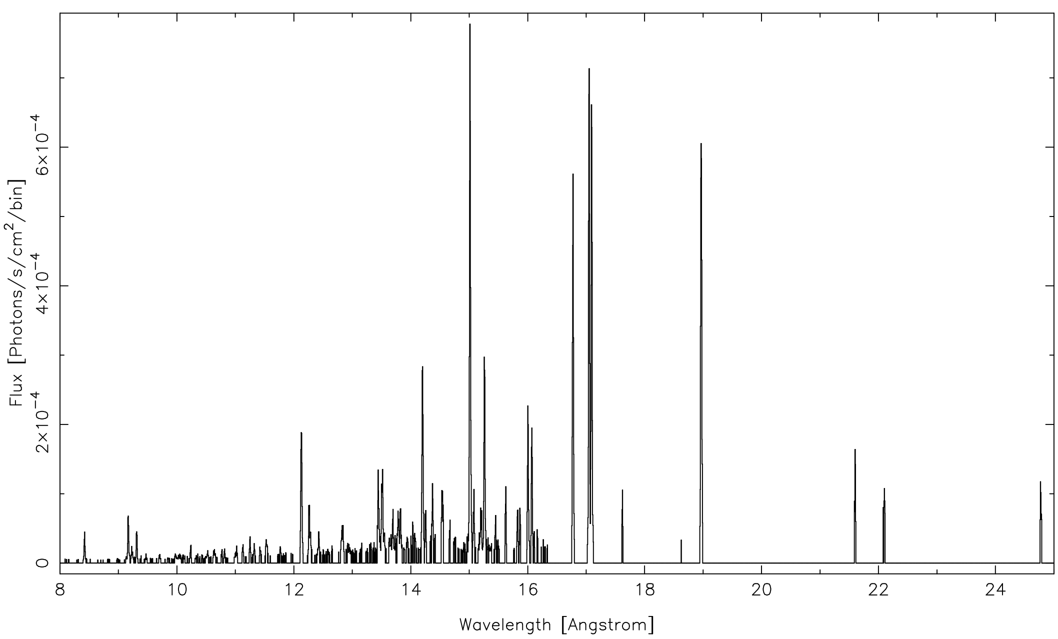

Figure 4.2: Example plot of a flux corrected spectrum

```
atoms (aped);
```

If the atomic data tables are loaded first, loading the emissivity tables will not unload them – any "new" emission lines encountered in the emissivity tables (ie not already listed in the wavelength tables) will be merged into the internal wavelength list.

By default, when ISIS loads the database, it reads all the specified files into memory, including tables of emission line wavelengths, atomic energy levels, ionization balance and line emissivities. With a large spectroscopy database (or a computer with a relatively small memory) this behavior might tax the system memory resources. To encourage ISIS to use less memory, the user can specify that it access large files by reading the necessary pieces from disk as needed rather than reading the entire file into memory at startup (see `Use_Memory`). Because of its large size, the continuum emissivity file is never loaded entirely; instead, the structure of this file is "mapped" so that run-time access for specific items in the file is as efficient as possible.

Important technical note: Although CFITSIO is capable of reading gzip-compressed FITS files transparently, the CFITSIO implementation first uncompresses the *entire* FITS file into memory before the read operation begins. This presents a severe problem if the *uncompressed* FITS file is larger than the computer memory(!) To avoid problems of this nature, it is best to uncompress the individual database files before reading them into ISIS. Reading uncompressed files has the added benefit that the files can be read in more quickly and run-time memory usage is minimized.

Note that even if the atomic database (wavelength tables, energy levels, etc.) is unavailable, the line and continuum emissivity tables may still be used to generate spectrum models. In this case, the ISIS functionality which requires the atomic data tables will be unavailable.

4.3 Identifying emission lines using a spectral model

Emission line identification is simplified by the ability to plot the locations of lines expected from a particular ion or lines which are expected to be bright at a given temperature. This process may be complicated by the fact that some database wavelengths may be incorrect – an important contribution of high-resolution observations will be in providing improved measurements of many line wavelengths. ISIS provides a mechanism to incorporate improved wavelength measurements without changing the database files directly (see `change_wl`).

The process of line identification might start by identifying the brightest lines in the spectrum (since they are likely to have reasonably well measured wavelength values). One approach is to compute a model spectrum and try to match the brightest model lines to those that have been observed. To do this, load a data set and then compute a model spectrum on the same grid using the spectroscopy database:

```
% reset global program state (unloads all data sets, etc.)
reset;

% load the entire type II pha file
load_data ("acisf01318N003_pha2.fits");

plot_data_counts (9);             % plot counts spectrum

plasma (aped);
load_model ("model.dat");         % or use edit_model
use_thermal_profile;              % thermal + turbulence
d = get_data_counts(9);           % use the data grid (Angstroms)

flux = model_spectrum (d.bin_lo, d.bin_hi);
```

Now, select the brightest 10 lines in a particular wavelength range in the model spectrum and over-plot those lines on the observed spectrum (e.g. Figure 4.3):

```
g = brightest(10, where (wl(10,12)));    % get indices of bright lines
                                         % between 10-12 Angstrom

plot_group(g);                     % overlay lines on an existing plot
page_group(g);                     % browse the data on those lines.
save_group (g, "bright_10.out");   % save the list in a file
```

In this case, note that the brightest lines in the model don't accurately predict all of the brightest lines in this region of the spectrum.

For a more direct comparison, the model intensities should be compared to the flux-corrected spectrum so that the variation of effective area with wavelength is removed from the observed spectrum (see `flux_corr`). This is especially important when identifying fainter lines as is the effect of contamination from higher-order features. Locations of features in other diffraction orders can be plotted using `lambda_mth_order`.

Even if collisional ionization equilibrium is not an appropriate model for the data, one may still make use of the wavelength tables in the spectroscopy database. For example, to plot the locations of all the Fe XVI and XVII lines between 10-10.5 Å:

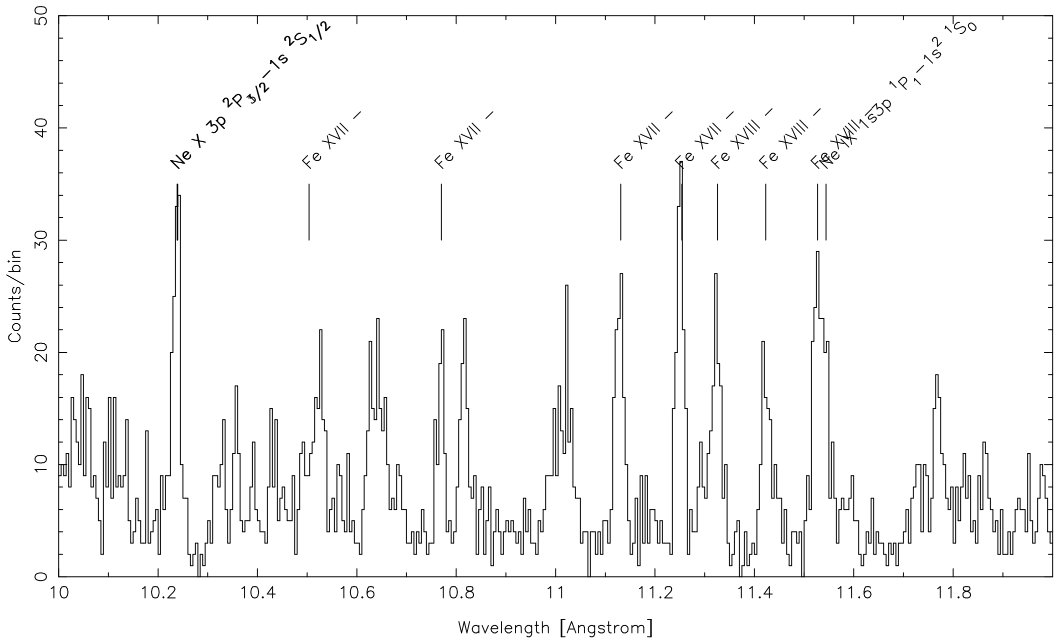

Figure 4.3: Example of emission line labeling

```
fe_lines = where (wl(10,10.5) and el_ion(26,[16,17]));
plot_group (fe_lines);

  OR

plot_group(where (wl(10,10.5) and el_ion(26,[16,17])));
```

Note that the output of the `where` command can be used directly without first being stored in a S-Lang variable.

By defining symbols like `Fe = 26` in the `.isisrc` file, one can avoid having to remember the proton numbers of the various elements (a sample `.isisrc` file is included in the ISIS distribution):

```
fe_lines = where (wl(10,10.5) and el_ion (Fe,[16,17]));
```

Having identified lines of interest, one might want to look at an energy level diagram to better understand how those lines are formed. Naturally, this requires that the spectroscopy database contain energy level information. If LS coupling energy level data are available, a simple energy level diagram can be generated (e.g. Figure 4.4), over-plotting lines transitions listed in a S-Lang array

```
plot_elev (Fe, 16);                 % Fe XVI
oplot_lines (Fe, 16, g);            % using an array g as defined above
list_elev (Fe, 16);                 % list the energy level table
```

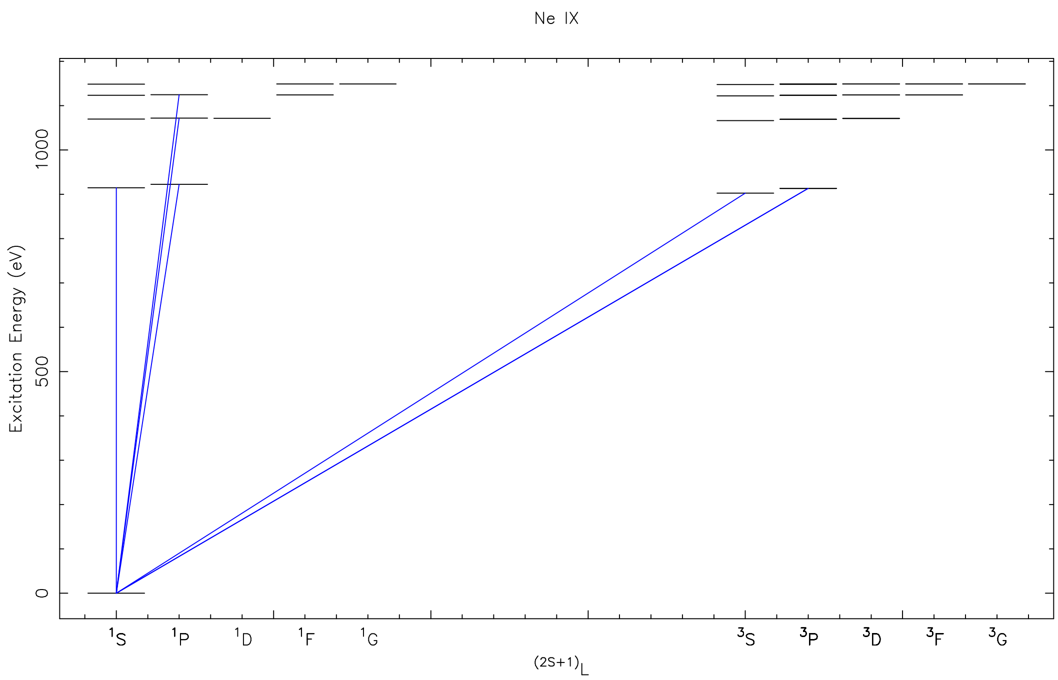

Figure 4.4: Example energy level diagram with over-plotted line transitions

When transition probabilities are available, the energy level listing also includes the total downward spontaneous radiative transition rate for each level, simplifying identification of meta-stable states.

4.4 Measuring line intensities

Observed line intensities may be measured using several different techniques with varying accuracy and sophistication.

The simplest measurement is a sum over the bins which make up a particular feature; because this approach neglects the effect of the line response function it probably should not be used to measure the intensities of individual lines in a blend.

To examine different methods of making line measurements, first load a data set and plot a spectrum, perhaps zooming in to a region of interest.

```
% load spectrum number 9
% variable sp = 1 if this is the only data currently loaded
sp = load_data ("acisf01318N003_pha2.fits", 9);

xrange(10,11);
yrange;
plot_data_counts (sp);            % plot counts spectrum
```

1. Summing bin values:

```
p = region_counts (sp, 10.2, 10.4);
print(p);                        % get struct field names
p.sum;                           % isis will echo the value
```

OR

```
(xmin, xmax) = xinterval;        % mark the interval using the
                                 % plot cursor
p = region_counts (sp, xmin, xmax);
```

OR

```
cursor_counts (sp);               % another cursor-based option
```

Notice that the region of interest is defined by specifying a *wavelength* range in Angstrom (Å) regardless of the units in the input data. Also, notice that the `region_counts` function returns a S-Lang *structure* rather than a simple scalar or array value. The fields of this structure are accessible using the dot notation which will be familiar to C and IDL programmers.

2. Fitting line profile functions:

When measuring the intensities of blended lines, one may account for the effect of the line response function by fitting the line blend with a sum of suitably chosen line profile functions plus a continuum function.

In interactive mode it may be most convenient to obtain initial estimates of the fit parameters by marking data regions with the plot cursor. To do that, start by plotting the data in appropriate bin-density units (e.g. counts/Å rather than counts/bin):

```
open_plot;
plot_bin_density;      % required for using ifit_fun() below
plot_data_flux (sp);   % assumes flux_corr() has been run
```

Now, narrow the plot range and `notice` the bins to be fit:

```
xrange(15.9,16.1); yrange;
errorbars (1);                % turn on error-bars

plot_data_flux (sp);

xnotice(sp,15.9,16.1);
```

As above, the bin-coordinates are always specified in Angstrom (Å) units, regardless of the physical units in the input data file.

One can initialize the fit parameters by clicking the mouse cursor on the screen plot – to do that, define the fit function using `ifit_fun`; here the "i" is for "interactive":

```
ifit_fun("Lorentz(1) + Lorentz(2) + poly(1)");
```

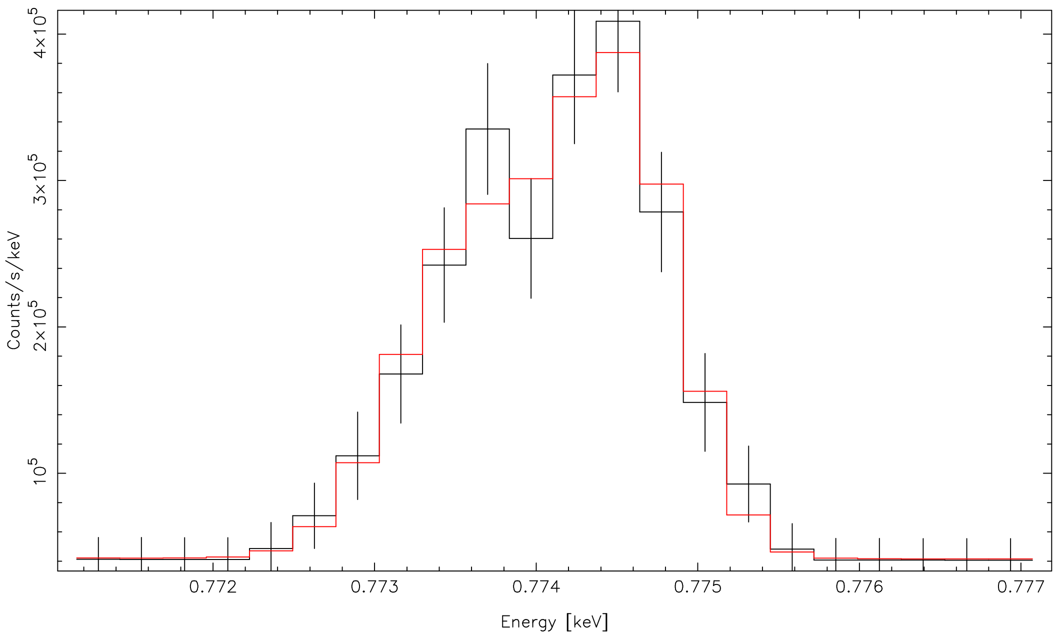

Figure 4.5: Example of fitting a line-blend with over-plotted error-bars.

The indices in the fit function components are used to label multiple occurrences of a particular component type. Here, we have two Lorentzian components labeled as number 1 (`Lorentz(1)`) and number 2 `Lorentz(2)`. After interactively assigning fit parameters, check the quality of the initial values, then do the fit and over-plot the result:

```
renorm_flux;                        % automatically adjust normalization
oplot_convolved_model_flux (sp);    % plot to check the initial params
fit_flux;
oplot_convolved_model_flux (sp);
```

It is important to remember that when fitting, all noticed bins in *all currently loaded spectra* are fit simultaneously. To fit a few line profiles to a small region of a single loaded spectrum, it is essential to first ignore all other spectra and all bins outside the region of interest

```
                             % If 12 spectra are loaded, and you want
                             % to fit only spectrum #9:
ignore([[1:8], [10:12]]);    % or ignore([1:8]); ignore([10:12]);
xnotice (9, 15.9, 16.1);
```

The unbinned fit function, if available for the chosen model, may also be over-plotted as in Figure 4.5:

```
x = [15.95:16.1:7.5e-4];            % generate some X coordinates
y = get_cfun(x);                    % evaluate the function Y(X)
oplot(x,y);
```

For further comparison, one may wish to plot the bin residuals or the ratio between model and data. The simplest way to do that is to use the `rplot_counts` function. Alternatively, one can exercise more control over the plot by manipulating the data and model values directly. For example:

```
d = get_data_flux (sp);
m = get_convolved_model_flux (sp);

limits;
hplot (d.bin_lo, d.bin_hi, d.value - m.value);        % residuals

% avoid division by zero
i = where (m.value != 0)

% plot the ratio
limits;
hplot (d.bin_lo[i], d.bin_hi[i], d.value[i]/m.value[i]);
```

Other common tasks are:

```
list_par;                        % list the fit parameters
conf(2);                         % compute confidence limits
save_par("lorentz2.fit");        % save the fit parameters
value = get_par (idx);           % get a fit parameter value
```

If a better fit is encountered while computing confidence limits, try re-fitting the data until a stable minimum is found.

3. Forward folding:

   In principle, the most accurate approach to measuring line intensities requires explicitly accounting for the effect of the line response function and for variations in the total effective area across the line blend. The sequence of commands is identical to that required to fit line profiles, except that additional information on the line response function (the RMF) and the total effective area (the ARF) is required. In addition to loading the spectral data, one must provide the necessary RMF and ARF files and indicate which response goes with which spectrum (see `load_rmf`, `assign_rmf`, `load_arf` and `assign_arf`). A detailed example is given in §2.3.1.

   By default, whenever response functions (ARF/RMF) are assigned to a spectrum, they are applied when fitting a model to that data set (e.g. forward-folding is the default). Alternatively, the `fit_*` and `renorm_*` functions accept flags which may be used to apply an ideal ARF and/or RMF even when a realistic instrument response has been assigned. This mechanism provides an easy way to isolate the effect of the instrument response.

4.5 Automatically finding, identifying and fitting emission lines

Although ISIS does not currently have intrinsic functions for automatic spectrum processing of this sort, it is possible to do this kind of automatic processing with a script. The advantage of this approach is that users can modify the script without having to recompile the code, thereby customizing the automatic processing for particular cases. An extended example is given in the examples tar file available from the ISIS web page; see the files

```
scan/scan.sl
scan/scan-demo1.sl
scan/scan-demo2.sl
```

The first of these files contains some useful and fairly generic S-LANG functions whose usage is demonstrated in the other two files.

Here is a simple example of how one might use the function `find_lines` (defined in `scan/scan.sl`) to quickly find and plot the brightest lines in a spectrum. First, copy the contents of the examples directory a working directory, then run ISIS:

```
% load and plot an example spectrum
isis> load_data("obsid_1103_Sky_MEGm1_isis.dat");
isis> plot_data_counts (1);

% load the scan functions
isis> .load scan.sl

% read the Usage message
isis> find_lines;

% find strong lines in spectrum 1
isis> (flam, fbin) = find_lines (1, 10);

% how many lines were found?
isis> n = length (flam);

% label plot using data values
isis> d = get_data_counts(1);
isis> for (i=0; i < n; i++) {
          xylabel(flam[i], d.value[fbin[i]], string(i));
      }
```

Although the last line is rather long, there's no need to type it every time. Instead, one could use it to define a S-LANG function which could then be customized to suit a specific application.

4.6 Comparing line measurements with theory

Measurements of line intensity ratios are often used to infer physical conditions in the emitting plasma. For example, the intensity ratio of bright lines from different ions of the same element may be used to infer the relative populations of the different ions, providing a measure of the ionization balance. Similarly, pairs of lines which arise from the same upper energy level in a single ion may be used to infer the optical depth; if one of the lines is optically thick, the branching ratio should differ from the value expected using the spontaneous transition rates (Einstein A values) alone. Radiative transitions involving meta-stable states often provide useful density diagnostics; such states may be identified using the energy level listing (see `list_elev`) which includes the total downward spontaneous radiative transition rate for each level.

Bright, relatively unblended lines may be selected using the line grouping functions discussed above (e.g. `brightest`, `unblended`). Lines which have the same upper energy level may be identified using the function `list_branch`; for example,

```
atoms(aped);              % First load the atomic data
list_branch (Fe, 18);     % Fe = 26
```

lists branching ratios for Fe XVIII:

```
Fe XVIII
upper level =    3
lower        lambda       branch        A         index
    2   1.039370e+02   2.6669e-01   3.3400e+10   14751
    1   9.392300e+01   7.3331e-01   9.1840e+10   14752
 ....
```

This listing indicates that, of downward transitions from the third excited state of Fe XVIII (see the corresponding energy level table), 73% produce a line at 93.923 Å and 27% produce a line at 103.937 Å.

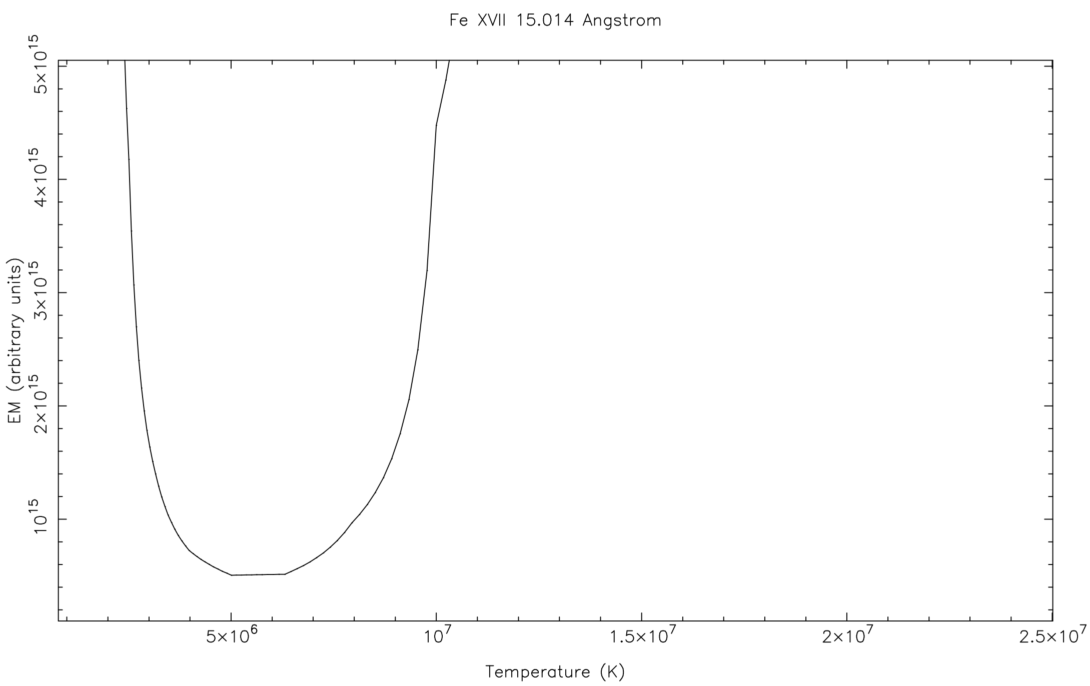

Figure 4.6: Example of estimating the emission measure implied by the flux in a single line.

Having identified line ratios which may be of interest, one can plot the predicted ratio of two lines with indices `n1` and `n2` vs temperature:

```
plasma(aped);                  % first load the emissivity tables
t = 10.0^[6.0:7.4:0.05];            % define a log spaced temperature grid
r = ratio_em (n1, n2, t);           % compute the ratio
plot(t,r);                          % plot the ratio vs. temperature
```

For a given *measured* line ratio, one could read off an estimate of the plasma temperature. A thorough analysis of a spectrum would normally use several, or even all, of the observed features to infer a distribution of physical conditions; for obvious reasons, relying on a single line ratio is not recommended.

Alternatively, because the ratio of the measured flux (F_λ) to the computed emissivity ($\eta_\lambda(T)$) may be expressed as

$$\frac{F_\lambda}{\eta_\lambda(T)} = \kappa(T)\frac{\int n_e n_H \, \mathrm{d}V}{4\pi D^2}. \tag{4.1}$$

one can generate a crude estimate of the differential emission measure near a given temperature using the measured flux (`flux_obs`) in a single line (`"k"`). To plot the measured flux divided by the theoretical emissivity vs. temperature (Figure 4.6):

```
  t = 10.0^[6.0:7.9:0.01];             % define a Temperature grid
  em = line_em (k,t);                  % get emissivity vs. temperature for
                                       % line k
  i = where(em > 0.0);
  plot (t[i], flux_obs / em[i]);      % plot the ratio
```

From this curve, one can read off values proportional to the emission measure at a given temperature.

In general, the emitting plasma will occupy a range of temperatures. To infer the temperature *distribution*, more sophisticated analysis techniques may be required (e.g. differential emission measure analysis). Although such algorithms are not currently implemented in ISIS, many of the required building blocks are provided.

4.7 Computing a spectral model using the XSPEC module

On systems which have XSPEC (Arnaud 1996) installed, the XSPEC module provides convenient access to selected functions in the XSPEC function library. On ELF systems, this module may be accessible by dynamic linking at ISIS run-time using the S-Lang `import` function. On such systems, one can explicitly `import` the module, or it will be automatically imported when any of the XSPEC functions is used for the first time. Alternatively, the XSPEC module may be also be statically linked to the ISIS executable so that calling `import` is not required at all.

Using this module, most XSPEC library routines are available as fit-functions. Selected XSPEC routines are also available as S-Lang functions which take the spectrum grid and other parameters directly as function arguments. Here is an example showing one way to compute and plot an XSPEC model spectrum (Figure 4.7):

```
variable e, s, n, lo, hi, t;
require ("xspec");

e = 10.0^[-1:1:0.005];                 % make a log energy grid
s = _mekal (5.0, e);                   % 5 keV, solar abundance
                                        % MEKAL spectrum

 n = length (s);                        % The e array has n+1 elements
lo = e[[0:n-1]];                       % low edge of histogram bin
hi = e[[1:n]];                         % high edge of histogram bin

variable id = open_plot ("xspec.ps/vcps");
resize(15);
```

```
xlog; ylog;
label ("log E (keV)", "log Flux", "");
hplot (lo, hi, s);                        % plot log-log

t = _wabs (0.01, e);                      % include Nh = 1.e20 cm^-2
ohplot (lo, hi, t * s);                   % overplot absorbed spectrum

close_plot (id);
```

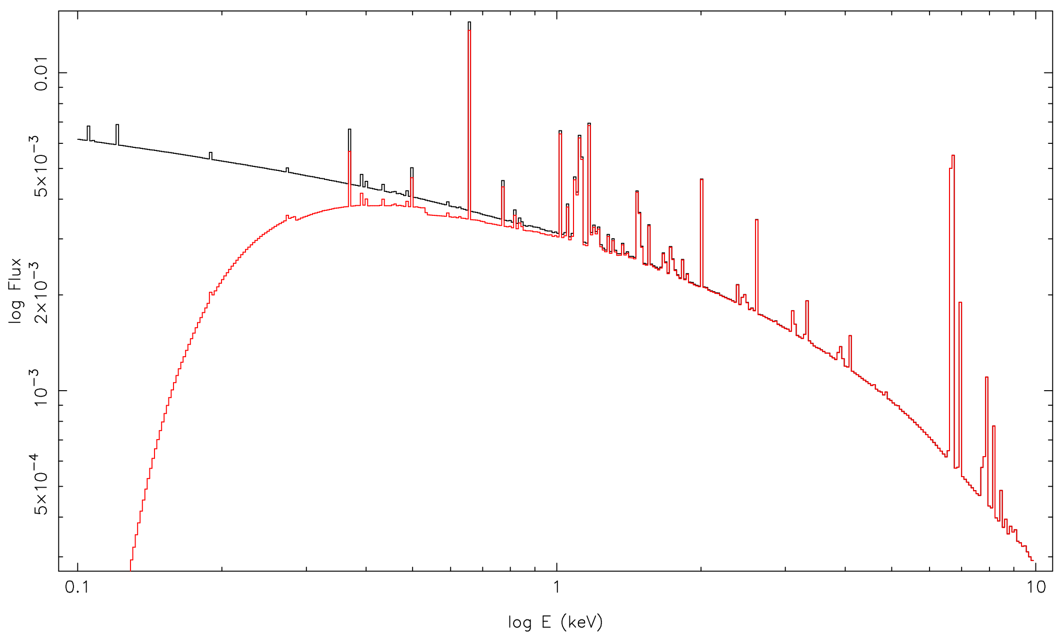

Figure 4.7: Example XSPEC model plot, illustrating the effect of an $N_H = 1.0 \times 10^{20}$ cm^{-2} absorbing column on a 5.0 keV "solar" abundance MEKAL plasma model.

Some array-index manipulation was required in this example because the ISIS convention for representing histograms differs from the XSPEC convention. If plotting the model flux vs. the low-edge of each bin would have been good enough, typing

```
require ("xspec");                    % (this is optional)
e = 10.0^[-1:1:0.005];                % make a log spaced energy grid
s = _mekal (5.0, e);                  % 5 keV, solar abundance
plot (e[[0:n-1]], s);
```

would have produced a similar plot without introducing the additional variables `lo` and `hi`.

4.8 Global fitting

In this section, the term *global fit* refers to fitting the entire range of input data with a plasma model which has a relatively small number of variable parameters (e.g. a one or two temperature CIE plasma model). Although global fits to high-resolution spectra are unlikely to accurately reproduce the wealth of observed features, a global fit may e.g. provide a quick estimate of the dominant plasma temperature(s) which are present. The current version of ISIS supports global fitting with XSPEC models (Arnaud 1996) (see §8), with the ISIS built-in fit-functions and with user-supplied fit functions. Global fits using spectral models generated using the spectroscopy database are supported via the `create_aped_fun` function which can create a custom fit-function with fit-parameters defined by relatively simple data structure (see `create_aped_fun` for further details).

After loading the data and assigning the appropriate instrument responses, a background component can be included from a file (`define_back`), from S-Lang variables (`_define_back`) or from a user-defined function (`back_fun`). Background subtraction may also be done "by hand" using a user-defined function such as this:

```
public define backsub (s_idx, b_idx)
{
   variable s, b;
   variable s_backscale, s_exposure, b_backscale, b_exposure;
   variable scale;

   s = get_data_counts (s_idx);
   b = get_data_counts (b_idx);

   s_backscale = get_data_backscale (s_idx);
   b_backscale = get_data_backscale (b_idx);

   s_exposure = get_data_exposure (s_idx);
   b_exposure = get_data_exposure (b_idx);

   scale = (s_backscale * s_exposure) / (b_backscale * b_exposure);
   s.value -= b.value * scale;
   s.err = sqrt (s.err^2 + (b.err * scale)^2);

   put_data_counts (s_idx, s.bin_lo, s.bin_hi, s.value, s.err);
}
```

Fitting is then carried out using the same sequence of steps described in §2.3.1.

4.9 Examining ionization balance curves

Much of the temperature dependence of emission lines in collisional ionization equilibrium comes from the temperature dependence of the ion concentration.

To examine ionization balance curves, first initialize the spectroscopy database so that an ionization balance table is loaded

```
  plasma (aped);
```

Then plot the ionization fraction of Fe XVII vs. temperature (Figure 4.8):

```
   t = 10.0^[6.0:7.2:0.05];          % define a log spaced temperature grid (K)
   limits;                           % reset the plot limits
```

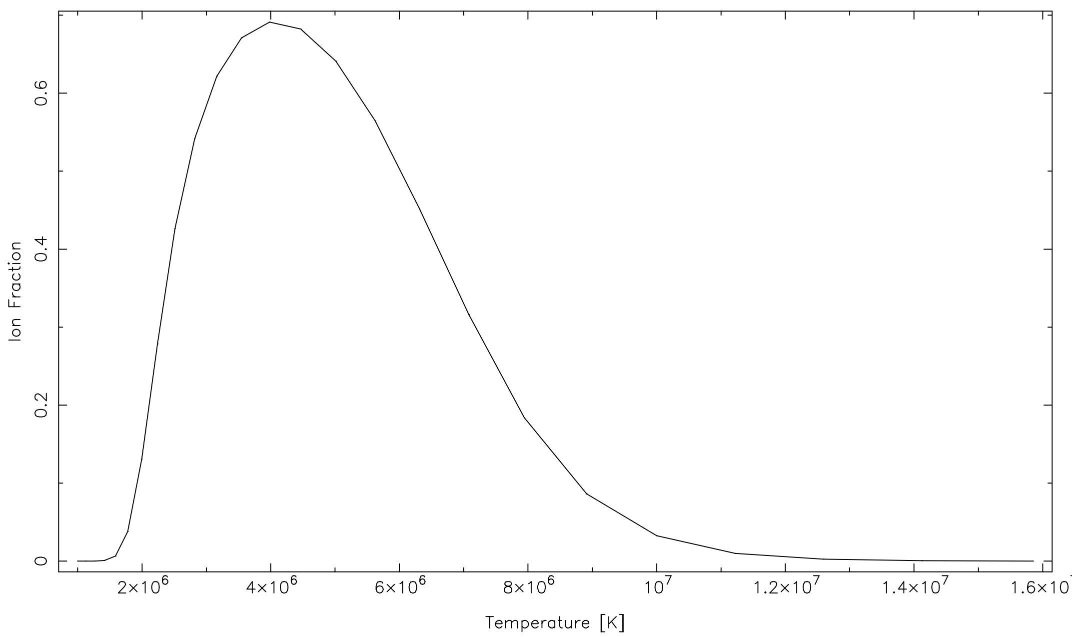

Figure 4.8: Example plot of Fe XVII ion fraction vs. temperature.

```
frac = ion_frac(Fe, 17, t);       % assuming Fe = 26 defined in ~/.isisrc
plot (t,frac);

label ("Temperature [K]", "Ion Fraction", "");     % label axes
```

The `plot()` function used in this example allows plotting any set of X-Y points held in S-Lang arrays; a similar function `hplot()` is provided for plotting arbitrary histograms. As a trivial example of the usefulness of this feature, the above plot could be regenerated vs. temperature in keV by typing

```
limits;
K_per_keV = 1.16e7;                     % conversion factor
plot ( t / K_per_keV, frac);
```

to apply the conversion factor to the temperature array before plotting.

In these example, the array of ion fractions returned by `ion_frac` is first stored in a variable before being passed to the plot function; this is especially useful if the array values will be used in further calculations. The intermediate variable is not necessary however:

```
plot (t, ion_frac(Fe, 17, t));
```

produces the same result – here, the array returned by `ion_frac` is passed directly to the `plot` function.

To compare the relative concentrations of ions of different elements as a function of temperature (Figure 4.9):

```
(i_fe, f_fe) = ion_bal (Fe, 1.e7);         % Fe at T = 1.e7 K
```

```
(i_ca, f_ca) = ion_bal (Ca, 1.e7);         % Ca at T = 1.e7 K
limits;                                    % reset the plot limits
hplot (i_ca-0.5, i_ca+0.5, f_ca);          % plot as histograms
ohplot (i_fe-0.5, i_fe+0.5, f_fe);         % over-plot Fe on Ca
xrange(14,26);                             % pick a better plot range
hplot (i_ca-0.5, i_ca+0.5, f_ca);          % re-plot
ohplot (i_fe-0.5, i_fe+0.5, f_fe);         %
```

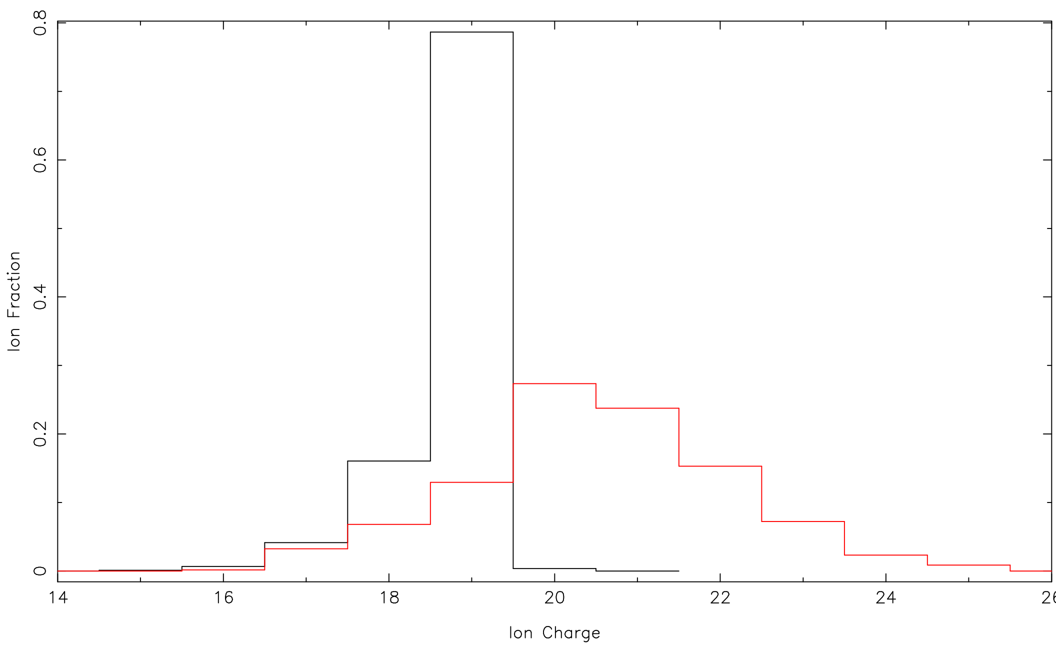

Figure 4.9: Example plot comparing Fe and Ca ionization balance at $T = 10^7$ K.

If the current plot device supports it, over-plots are automatically done in a different color to help simplify complex plots; this feature can be disabled using `plot_auto_color ()`.

4.10 Examining continuum emission components using the spectroscopy database

First, initialize the spectroscopy database to scan the continuum emissivity tables:

```
reset;              % reset the global state
plasma (aped);
```

then, generate a wavelength grid and retrieve the continuum components for a particular temperature

```
(lo, hi) = linear_grid(1, 20, 3000);
 p = get_contin (lo, hi, 3.e6);         % use T = 3.e6 K

 who;                                   % how many variables are defined?
```

```
print (p);                             % print struct components
```

`true` is the sum of true continuum components including continua due to bremsstrahlung, radiative recombination, and two-photon emission and `pseudo` is the pseudo-continuum of weak emission lines.

Now, plot the components for comparison

```
ymax = max ([p.true, p.pseudo]);           % set the plot range
yrange (ymax*1.e-4, ymax);                 % so all components fit
ylog;

hplot (lo, hi, p.true);
label ("Wavelength [Angstrom]",
       "log Flux [photon cm\\u3\\d s\\u-1\\d]",
       "T = 3.0e6 K");
ohplot (lo, hi, p.pseudo);

q = get_contin (lo, hi, 3.e6, 1.0e-3, O);       % Contribution from Oxygen
ohplot (lo, hi, q.true);
ohplot (lo, hi, q.pseudo);
```

The fields of struct `p` can be used in further calculations. For example, one can integrate the various emission components over the wavelength range of interest to determine the various continuum contributions:

```
isis> e_erg = 1.988e-8/(0.5*(lo+hi));         % photon energy at bin center
isis> tot_true = sum(e_erg * p.true);         % sum over bins
isis> tot_true;                               % print the value
```

4.11 Modeling Event Pileup in CCD Detectors

For a detailed discussion of the pileup model see Davis (2001), available on the web from

`http://space.mit.edu/~davis/pileup2001.html`

See §7.7 for a suggestion on how to speed up the pileup model computations. This section provides a brief example showing how to use the model to fit CCD data.

For an on-axis point source, extract counts in a circular region with a radius of about four pixels; this will enclose about 95% of the encircled energy. A larger extraction region will enclose more source counts but will also increase the background level.

Then, in ISIS issue the appropriate commands to load the data, e.g.,

```
() = load_data ("data.fits");
() = load_arf ("arf.fits");
assign_arf (1,1);
() = load_rmf ("rmf.fits");
assign_rmf (1, 1);
```

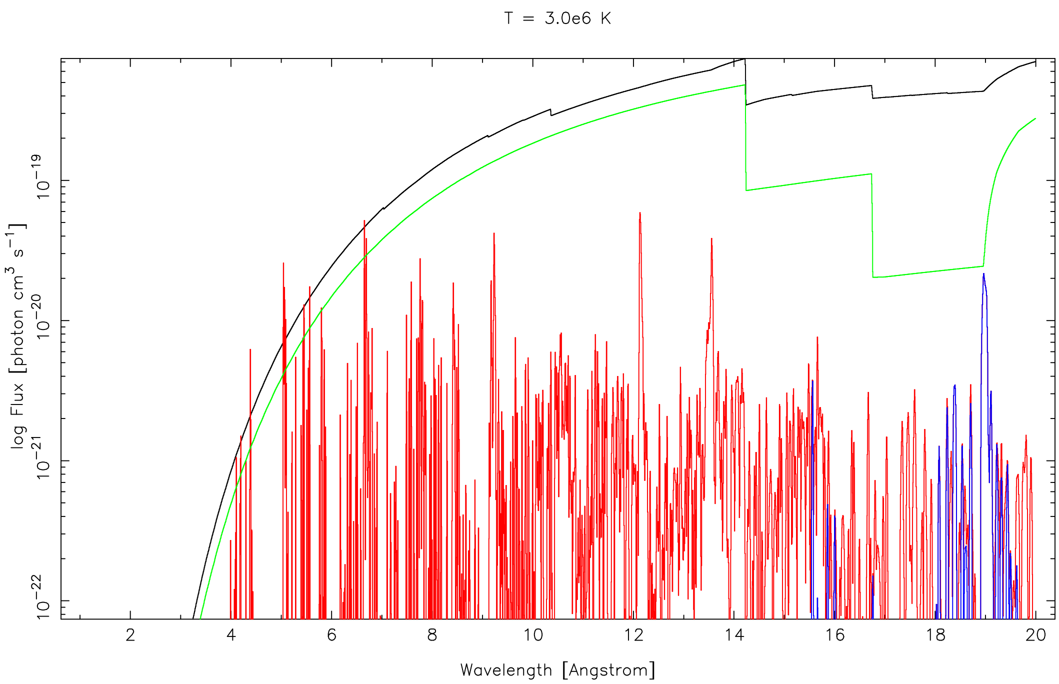

Figure 4.10: Example of plotting continuum emission components for a solar abundance plasma at $T = 3 \times 10^6$ K.

Equivalently, one could use:

```
load_dataset ("data.fits", "rmf.fits", "arf.fits");
```

Now group the data bins, e.g. use

```
group_data (1, 12);
```

to group every 12 pha channels together.

Then specify a model, e.g.,

```
require ("xspec");                % this is optional
fit_fun ("wabs(1)*powerlaw(1)");
```

Now indicate that you want to fit this data set using pileup:

```
set_kernel (1, "pileup");
```

Finally, edit the parameters:

```
edit_par;
list_par;
```

This will produce something like:

```
wabs(1)*Powerlaw(1)
 idx  param            tie-to  freeze    value      min    max
  1  wabs(1).NH_22       0      0        0.01      1e-5   0.1
  2  Powerlaw(1).norm    0      0       0.003      1e-5   0.01
  3  Powerlaw(1).alpha   0      0           1         0   0
  4  pileup.<1>nregions  0      1           1         1   1.5
  5  pileup.<1>g0        0      1           1         0   1
  6  pileup.<1>alpha     0      0         0.5         0   0.75
  7  pileup.<1>psffrac   0      1        0.95       0.9   1
```

The `pileup<1>` parameters are specific to the pileup model and the number in angle-brackets (`"<>"`) is the data-set index, e.g. here indicating that the pileup model is being applied to data set 1. For an on-axis point source, allow only `alpha` to vary, leaving `psffrac` fixed at 0.95. This indicates that 95 percent of the encircled energy in the extraction region fell within the central 3x3 pixel.

Finally, run the fit:

```
() = fit_counts;
```

Because the chi-square space is quite complicated, it is advisable to repeat the fit several times, starting from random parameter values. To simplify this process, ISIS provides a `randomize` function to automatically randomize all free fit-function parameters within the currently specified `min-max` parameter ranges. It may also be helpful to fit first using `subplex` to locate the neighborhood of the global minimum and then use `mpfit` to refine the answer. For example:

```
set_fit_method ("subplex");
loop (3)
  {
     () = fit_counts;
  }
set_fit_method ("mpfitx");
() = fit_counts;
```

Part II

ISIS Reference Manual

Section 5

The Spectroscopy Database

This section discusses the database content, layout and parameter file definition in some detail and may be of interest primarily to the person installing ISIS and maintaining the local spectroscopy database.

5.1 Obtaining Spectroscopy Database Files

ISIS should be able to read files from the spectroscopy database assembled by Smith, Brickhouse, Liedahl & Raymond (2001). These files may be downloaded from the web at

`http://www.atomdb.org/`

The minimum ISIS configuration requires only the line emissivity tables available from from this web page (see §5.2.1).

5.2 Organization

The structure of the local database is specified to ISIS by providing directory paths and file names via a S-Lang structure. Perhaps the most convenient way to specify this structure is by creating and initializing it using a short S-Lang script; for example, the following script is included in the ISIS distribution as `etc/aped.sl`:

```
public define aped ()
{
   variable db = struct
     {
        dir, atomic_data_filemap,
           abundance, ion_balance,
           line_emissivity, continuum_emissivity
     };
```

```
   db.dir = "/nfs/atum/d2/houck/isis/data/aped/atomdb";
   db.atomic_data_filemap = "filemap";

   db.abundance = "APED/misc/Abundances.fits";
   db.ion_balance = "APED/ionbal/MM98_ionbal.fits";

   db.line_emissivity = "apec_line.fits";
   db.continuum_emissivity = "apec_coco.fits";

   return db;
}
```

This script generates an instance of the necessary S-LANG`Struct_Type` containing the path to the spectroscopy database and the names of files containing the line and continuum emissivities (`apec_line.fits` and `apec_coco.fits`), the ionization balance tables (`MM98_ionbal.fits`), elemental abundance tables (`Abundances.fits`) and a filemap listing the associated atomic-data files containing, among other things, tables of energy levels and wavelengths of X-ray line transitions.

In general, such an initialization script is defined only once, perhaps when ISIS or when a new database is installed, and is not accessed by the casual user afterward.

Also, perhaps it is an obvious point, but please keep in mind that the functionality of ISIS depends on the content of the spectroscopy database provided by the user. Importantly, this database is *not part of* ISIS . If the database contains little or no information on the wavelength range of interest, the problem is in the database and not in ISIS.

In general, the spectroscopy database is logically divided into two sections: 1) an emissivity database containing ionization balance tables and tables of line and continuum emissivities as a function of plasma state parameters (e.g. temperature and density and 2) a set of atomic data (energy levels, emission line wavelengths and transition probabilities for selected ions of astrophysically abundant elements – quantities generally independent of the plasma physical state). This organization is mirrored in the two commands used to load data from the database (see `atoms` (§7.3) and `plasma` (§7.4) below). The database files are in FITS format and are described in detail in the CXC Spectroscopy Database ICD.

Although the atomic data tables are optional, when both data sets are used together, it is assumed that the emissivity database has been derived from the atomic database. Depending on the assumed emission model, different emissivity databases may be derived from the same atomic data set.

The standard emissivity database assumes collisional ionization equilibrium as a function of electron temperature T and electron density n. ISIS supports reading several types of files from a collisional ionization equilibrium spectroscopy database:

- A line emissivity file containing the line spectrum for a solar abundance plasma as a function of temperature and density. Each extension contains the line spectrum at a single temperature, density combination.
- A continuum emissivity file containing continua for a solar abundance plasma as a function of temperature and density. Continua are available for individual elements, with true-continuum and pseudo-continuum (weak lines) stored separately; optionally, continuum components broken down by ion may also be available.
- A set of cosmic elemental abundance tables in a single FITS file.

- An ionization balance file containing tables of ionization balance as a function of temperature.

ISIS also supports reading these files from the atomic database:

- Wavelength tables – one per element – with each extension containing a list of wavelengths, A-values and energy-level indices for a single ion.
- Energy level tables – one per element – with each extension containing a list of energy levels for a single ion.

The spectroscopy database may be configured in several different ways, depending on which of these files are available.

Important technical note: Although CFITSIO is capable of reading gzip-compressed FITS files transparently, the CFITSIO implementation first uncompresses the *entire* FITS file into memory before *any* read operation begins (even reading a single keyword from the header is enough). This presents a severe problem if the *uncompressed* FITS file is larger than the computer memory(!). To avoid problems of this nature, it is best to uncompress the individual database files before reading them into ISIS. Reading uncompressed files has the added benefit that the files can be read in more quickly and run-time memory usage is minimized.

5.2.1 Minimal Configuration

As mentioned above, the contents of the spectroscopy database must be specified using a S-Lang structure of the form:

```
variable db = struct { dir, atomic_data_filemap,
                       abundance, ion_balance,
                       line_emissivity, continuum_emissivity
                     };
```

(the name of the structure is arbitrary – for this example, we have chosen to call it `db`) The fields of this structure are strings which provide the spectroscopy database directory and the names of selected files.

dir	database directory
atomic_data_filemap	APEC-format atomic-data filemap
abundance	Table of elemental abundances
ion_balance	Ionization balance table
line_emissivity	Table of line emissivities
continuum_emissivity	Table of continuum emissivities

Note that it is not necessary to have all of these files available in order to use the spectroscopy database. The minimal database configuration requires e.g. only a single line-emissivity table. In this case, the required database configuration structure may be initialized by defining only two of the struct fields:

```
db.dir = "/data/apec";
db.line_emissivity = "apec_line.fits";
```

We point out that the database files can be placed in any convenient directory – for the purpose of this example, we assume the database is in "/data/apec".

Once the database configuration structure is defined, the indicated files are loaded using the functions `plasma` and/or `atoms`. To load both the atomic data and emissivity tables, use

```
plasma (db);
```

where `db` is the name of the database configuration structure. Similarly, to load only the atomic-data files, use

```
atoms (db);
```

Note that the various database files need not all lie in the same directory tree. If the files lie in different directories, one can define the directory to be an empty string (`dir = ""`) and then use the other struct fields to specify the full path to each file.

To simplify creating the database configuration structure and to provide greater versatility, we recommend defining the structure in a S-Lang function. For example, the full APEC database might be defined using a function like this:

```
public define aped ()
{
   variable db = struct
     {
        dir, atomic_data_filemap,
        abundance, ion_balance,
        line_emissivity, continuum_emissivity
     };

   db.dir = getenv("ATOMDB");
   if (db.dir == NULL) db.dir = "./";

   db.atomic_data_filemap = "filemap";

   db.abundance = "APED/misc/Abundances.fits";
   db.ion_balance = "APED/ionbal/MM98_ionbal.fits";

   db.line_emissivity = "apec_line.fits";
   db.continuum_emissivity = "apec_coco.fits";

   return db;
}
```

Because this subroutine initializes and returns the necessary structure, the subroutine itself can be used as the argument of the `plasma` or `atoms` functions. For example,

```
plasma (aped);
```

is exactly equivalent to

```
db = aped();
plasma (db);
```

Section 6

Interactive Mode Features

6.1 Command Line Editing

If ISIS was compiled with GNU `readline`, the command line editing features are those provided by GNU `readline`; see the GNU `readline` documentation for details. The editing features discussed below are those provided by the native S-Lang interpreter. To select the command-line behavior, use `set_readline_method`.

`emacs`-style command line editing is supported. The cursor movement keys can be used to navigate the command history as can the following the control sequences:

ctrl-a	beginning of line
ctrl-e	end of line
ctrl-b	cursor left
ctrl-f	cursor right
ctrl-p	previous command
ctrl-n	next command
ctrl-d	delete the character under the cursor (also exits ISIS if it is the only character typed on the command line)
ctrl-h	delete the character to left of the cursor (or use the delete key)
ctrl-k	delete to end of line

Command line editing works well for browsing and editing the history of single-line commands, but doesn't fully support editing multi-line commands.

6.2 Control Sequences

Several control sequences are available for controlling ISIS in interactive mode:

ctrl-c	Halts execution of an ISIS command, returning to the ISIS prompt. Use `quit` to exit from ISIS.
ctrl-z	Suspend ISIS and return the Unix system prompt.
ctrl-s	XON/XOFF flow control; suspends the terminal
ctrl-q	XON/XOFF flow control; un-suspend the terminal

6.3 Unix Shell Escapes

If the first character on the command line is an exclamation point(!), the remainder of the line is passed to the Unix Bourne shell `/bin/sh`. For example, the following commands are equivalent:

```
isis> !ls ..
isis> ()=system("ls ..");
```

Because the Unix Bourne shell parses the command string, csh-like special characters are not understood:

```
isis> !ls ~             % fails;  Bourne shell doesn't understand '~'
isis> !ls src/*.fits    % works;  Bourne shell understands '*'
isis> !ls $HOME         % works;  Bourne shell understands '$HOME'
```

Unix shell aliases are also unavailable because shell configuration files (e.g. `~/.cshrc`) are *not* invoked.

The `cd` (change directory) command is an exception. Because the string argument of the `cd` command is *not* interpreted by a Unix shell, the specified directory path must be given explicitly without making use of any environment variables or wildcard characters:

```
isis> !cd ..            % works; '..' is a real directory
isis> !cd $HOME         % fails; unless there is a directory named $HOME
```

6.4 Command Shortcuts

In interactive-mode, if the first character on the command line is a period (.) the next non-blank characters are interpreted as a shortcut command name and the rest of the line supplies whitespace-delimited arguments for that command shortcut (if any).

.apropos

Purpose: Shortcut for the apropos() function
Usage: `.apropos string`
See Also: `apropos`

See the documentation for the S-Lang `_apropos` intrinsic function for details; the only difference between the shortcut and the intrinsic function is that intrinsic function provides namespace and filter arguments which are not supported by the shortcut command. In particular, both support S-Lang regular expressions.

```
For example:

isis> apropos data

Found 39 function matches in namespace Global:
__data_weights            __datatype                __is_datatype_numeric
all_data                  combine_datasets          copy_data_keywords
delete_data               get_data                  get_data_backscale
get_data_counts           get_data_exposure         get_data_flux
get_data_info             get_dataset_metadata      get_rmf_data_grid
group_data                have_data                 list_data
load_data                 load_dataset              match_dataset_grids
oplot_data                oplot_data_counts         oplot_data_flux
plot_data                 plot_data_counts          plot_data_flux
put_data                  put_data_counts           put_data_flux
rebin_data                rebin_dataset             set_data_backscale
set_data_color            set_data_exposure         set_data_info
set_dataset_metadata      uncombine_datasets        unset_data_color

Found 6 variable matches in namespace Global:
DataError                 DataType_Type             ENODATA
Isis_Active_Dataset       Isis_Data_Info_Type       _FITS_BAD_DATA_FILL
isis>
```

.cd

Purpose: change ISIS working directory while in interactive-mode
Usage: `.cd path`
See Also: `chdir`

This is the same as using the Unix shell-escape (see below) to change directories:

```
isis> !cd ..
```

Because the argument is not evaluated by the Unix shell, wildcards and environment variables may not be used.

.help

Purpose: Shortcut for the help() function
Usage: `.help topic`
See Also: `apropos, help`

See the documentation for the `help` intrinsic function.

.load

Purpose: run a S-Lang script in interactive-mode
Usage: `.load filename`
See Also: `.source, evalfile`

This is an interactive-mode shortcut for the S-Lang intrinsic function `evalfile()` which interprets a file containing S-Lang code. For example, the following commands are almost equivalent:

```
isis> .load script.sl
isis> ()=evalfile("script.sl");
```

The only difference is that `evalfile()` returns an integer status code to indicate whether or not the file was interpreted successfully (1 for success, 0 for failure); the `.load` command does not return a status code and is only available in interactive-mode.

The "`.sl`" extension on the script name is optional.

.source

Purpose: read S-Lang commands from a file
Usage: `.source filename`
See Also: `.load, evalfile`

This function is very similar to the `.load` shortcut except that lines from the file are interpreted as though they had been typed at the command line – all command shortcuts are available and variables need not be declared before use.

Section 7

ISIS Function Reference

This chapter documents all parameters and functionality of all ISIS intrinsic functions; related functions are grouped together in separate sections. Valid functions may be ISIS intrinsics (e.g. `load_data`) or S-Lang intrinsics (e.g. `message`). Only ISIS intrinsic functions are described here; see the S-Lang documentation for a description of the available S-Lang functions.

Many ISIS functions take a variable number of arguments. In the following descriptions, optional arguments are enclosed in brackets ([]). Similarly, the notation `[o]plot` is intended to stand for either `plot` or `oplot`.

The ISIS interactive help system supplies function name lookup (`apropos`) as well as more extensive documentation ((`help`). For quick-reference, most ISIS functions display a usage message when invoked with no arguments (S-Lang intrinsics do not).

7.1 Utility Functions

A number of utility functions are provided for execution control (`quit`, `reset`), customization (`alias`), and help (`apropos`, `help`). Repetitive tasks may be simplified by first logging an interactive session (see `start_log`, `stop_log`, `save_input`) and then editing the log-file to produce one or more scripts to handle subsequent repetitions.

_A

Purpose: Convert grids between
Angstrom and keV
Usage: `_A (lo[, hi])`
See Also:

This simple function can convert either one or two numerical arguments from
Angstrom wavelength units to keV energy units or the reverse. For array input, the returned arrays have the same numerical sort order as the input array, making it convenient to convert an array of energies to an array of wavelengths, each in increasing order.

```
Example:

isis> e_kev = [1,2,3];
isis> print(_A(e_kev));
4.132806e+00
6.199209e+00
1.239842e+01
isis> lo_kev = [1,2,3];
isis> hi_kev = [2,3,4];
isis> (lo_angstrom, hi_angstrom) = _A(lo_kev, hi_kev);
isis> print(lo_angstrom);
3.099605e+00
4.132806e+00
6.199209e+00
isis> print(hi_angstrom);
4.132806e+00
6.199209e+00
1.239842e+01
```

When the function argument is a `Struct_Type` of the form

```
struct {bin_lo, bin_hi, value, err},
```

the arrays `bin_lo, bin_hi` are converted as above and the `value` and `err` arrays are reversed. This allows one to use the `_A` function in conjunction with the various routines that handle these structures. For example:

```
d = get_data_counts(1);
hplot(_A(d));
```

_featurep

Purpose: Test whether or not a feature is present
Usage: `Int_Type _featurep (String_Type feature)`
See Also: `require, provide`

The `_featurep` function returns a non-zero value if the specified feature is present. Otherwise, it returns 0 to indicate that the feature has not been loaded.

add_help_file

Purpose: Add a documentation file to the list searched by help
Usage: `add_help_file (file)`
See Also: `help, apropos, add_help_hook`

When search for documentation on a given symbol, isis searches a list of ascii-format files and can also call a number of user-supplied functions which also search for help.

`add_help_file` prepends a file to the list of ascii-format documentation files which are searched to find help on a specified topic.

add_help_hook

Purpose: Add a function to display user-supplied help information
Usage: `add_help_hook (name)`
See Also: `help, apropos, add_help_file`

When search for documentation on a given symbol, isis searches a list of ascii-format files and can also call a number of user-supplied functions which also search for help.

`add_help_hook` adds to the list of functions which search for help and display the search results.

add_to_isis_load_path

Purpose: Add a directory to the file search path
Usage: `add_to_isis_load_path ("dir")`
See Also: `get_isis_load_path`

`add_to_isis_load_path` prepends a directory to the script-search path.

Related functions with fairly obvious definitions are: `set_isis_load_path`, `prepend_to_isis_load_path`, `append_to_isis_load_path`.

add_to_isis_module_path

Purpose: Add a directory to the module search path
Usage: `add_to_isis_module_path ("dir")`
See Also: `get_isis_module_path`

`add_to_isis_module_path` prepends a directory to the path searched for dynamically linked libraries (.so files) containing, e.g. compiled modules or user-defined fit-functions.

For example, if a user-defined function is contained in a dynamically linked library at `/home/joe/modules/libmodel.so`, one could add the directory `/home/joe/modules/` to the module search path using

```
add_to_isis_module_path ("/home/joe/modules");
```

Related functions with fairly obvious definitions are: `set_isis_module_path`, `prepend_to_isis_module_path`, `append_to_isis_module_path`.

alias

Purpose: define an alternate name for an intrinsic function
Usage: `alias ("oldname","newname")`
See Also: `who`

After defining an alias for an intrinsic function, the function can be accessed using either the new name or the original name.

apropos

Purpose: recall object names satisfying a regular expression
Usage: `apropos("s")`
See Also: `.apropos, help, who`

The apropos function may be used to get a list of all defined objects whose name matches the regular expression `"s"`.

```
For example:

isis> apropos load

Found 23 function matches in namespace Global:
add_to_isis_load_path      append_to_isis_load_path   autoload
get_isis_load_path         get_slang_load_path        load_alt_ioniz
load_arf                   load_conf                  load_data
load_dataset               load_fit_method            load_fit_statistic
load_kernel                load_line_profile_function load_model
load_par                   load_rmf                   mt_load_model
prepend_to_isis_load_path  prepend_to_slang_load_path rline_load_history
set_isis_load_path         set_slang_load_path
isis>
```

Because all strings generate a match with the empty string (`""`), this can be used to obtain a list of almost all available ISIS intrinsic functions and variables.

array_struct_field

Purpose: Make an array from one field of an array of Struct_Type
Usage: `a[] = array_struct_field (Struct_Type[], "field_name")`
See Also: `print`

For example:

```
isis> d=load_data ("pha2.fits");
Reading: ............
isis> info = get_data_info(d);
isis> part = array_struct_field(info, "part");
isis> part;
Integer_Type[12]
```

delete

Purpose: un-initialize a variable
Usage: `delete (["pattern"])`
See Also: `who`

All variables whose names contain `"pattern"` are un-initialized; if if `"pattern"` is absent, all currently defined variables are un-initialized.

```
For example:

    isis> x=[1,2,3,4];        % define 3 variables
    isis> y=&x;
    isis> z=x*3.0;
    isis> who;                % verify they exist
    x: Integer_Type[4]
    y: &x
    z: Double_Type[4]
    isis> delete;             % delete all 3
    isis> who;
    x: *** Not Initialized ***
    y: *** Not Initialized ***
    z: *** Not Initialized ***
```

atexit

Purpose: Register a function to be called when ISIS exits
Usage: `atexit (&fcn)`
See Also:

Use this function to indicate that a particular function should be called when ISIS exits. The function will be called with no arguments and should return nothing.

If several such functions are specified, the functions will be called in reverse order. If a slang error occurs when one of the functions is called and if that error is not cleared, the remaining functions registered with `atexit` will not be called.

For example:

```
isis> define cleanup1 () {message ("called cleanup1");}
isis> define cleanup2 () {message ("called cleanup2");}
isis> atexit (&cleanup1);
isis> atexit (&cleanup2);
isis> exit;
called cleanup2
called cleanup1
```

chdir

Purpose: Change directories
Usage: `s = chdir (dir)`
See Also:

If successful, this function returns `s=0`, otherwise it returns `s=-1`.

debug

Purpose: Debugging in ISIS
Usage: `isis> help debug`
See Also: `help, apropos, isis --help option`

There are several ways ISIS users may debug or fine-tune the performance of analysis sessions and scripts. We summarize them here, and refer the reader to the S-Lang user manual for more details.

Interactive Debugging:

The `sldb` command will invoke the interactive S-Lang debugger and change the prompt accordingly. Type `help sldb` for more information.

Function and Variable Tracing:

`_traceback` is an intrinsic integer variable whose bitmapped value controls the generation of the call-stack traceback upon error. When set to 0, no traceback will be generated. Otherwise its value is the bitwise-or of the following integers:

```
1        Create a full traceback
2        Omit local variable information
4        Generate just one line of traceback
```

In batch mode, the default value of this variable is 4; in interactive mode, the default value is 0.

Apropos:

The `apropos` and `_apropos` functions may be used to find function and/or variable definitions which match a given expression. For more details consult the help content for each function.

Command Line Options:

Invoking isis on the command line with the `--help` option displays several options which can assist debugging. Among these are:

```
--sldb [FILE]  Invoke S-Lang debugger, optionally on the given FILE

--sldb-isis    Invoke S-Lang debugger on isis internals

--prof [options] script args...

               Invoke S-Lang profiler, optionally on the given FILE
               The profiler can be helpful to increase the performance of
               your scripts by enabling one to see which functions are
               called most often and for how long.

-g             Include debugging information when byte-compiling scripts
               (this automatically sets _traceback to 1)

-v             Show verbose loading messages, which can be helpful to
               discern where scripts are being loaded from, especially
               if multiple copies of them exist on your system.
```

sldb

Purpose: Initiate interactive debugging
Usage: `sldb`
See Also: `_traceback, help debug, isis --sldb and --help options`

This command will invoke the interactive S-Lang debugger and change the prompt accordingly. The `sldb` debugger is loosely patterned after `gdb` and similar debuggers, and allows one to set breakpoints, step through S-Lang statements line by line, inspect or change variable values, and so forth. Type `help sldb` at the ISIS prompt for more information, or `help` once `sldb` is invoked, or consult the sldb chapter of the S-Lang user manual.

fft

Purpose: Compute the discrete Fourier transform of a complex array
Usage: `X[] = fft (x[], sign)`
See Also: `fft1d`

```
  fft(x, sign)[k] = sum_j( x[j] * exp(sign* 2*PI*i *j*k / length(x)) )
                    /sqrt(length(x))
```

where `sign` is `+1` or `-1`.

fft1d

Purpose: Compute the discrete 1D Fourier transform
Usage: `(R, I) = fft1d (re, im, sign)`
See Also: `fft`

This function is deprecated: `fft1d` is essentially the same function as `fft` except that it handles the real and imaginary parts separately. See `fft` for details.

get_isis_load_path

Purpose: Get the current script load path
Usage: `value = get_isis_load_path ()`
See Also: `add_to_isis_load_path`

When loading scripts, ISIS searches directories specified in the current load path which, by default, is initialized using the ISIS_LOAD_PATH and SLANG_LOAD_PATH environment variables. `get_isis_load_path` retrieves this search path.

get_isis_module_path

Purpose: Get the value of the current module search path
Usage: `value = get_isis_load_path ()`
See Also: `add_to_isis_load_path`

When loading scripts, ISIS searches directories specified in the current module path which, by default, is initialized using the ISIS_MODULE_PATH and SLANG_MODULE_PATH environment variables. `get_isis_module_path` retrieves this search path.

grand

Purpose: Generate Gaussian-distributed random numbers
Usage: `nums = grand ([n [,m ...]])`
See Also: `urand, prand, seed_random`

If no arguments are given, a single random number will be generated. If N integer arguments are provided, they are interpreted as defining the dimensionality of an array which is to be populated with random numbers.

For example:

```
x = grand();          % returns Double_Type
a = grand(10);        % returns Double_Type[10]
b = grand(100,200);   % returns Double_Type[100,200]
```

help

Purpose: Retrieve help on ISIS and S-Lang intrinsics
Usage: `help ("string")`
See Also: `apropos, who`

If the string argument matches the name of a documented function, the documentation for that function is displayed.

If no documentation is found, the function prints a list of S-Lang and ISIS intrinsic function names containing the string argument. The function list is the same as would be obtained using the apropos function with `flags = 0xF`. Because all strings generate a match with the empty string (`""`), an empty topic string can be used to obtain a list of all available intrinsic functions.

A usage message is generated if no topic string is specified.

histogram

Purpose: Bin scatter data into a histogram
Usage: `nx = histogram(x, lo [, hi[, &rev]])`
See Also: `hplot, histogram2d, linear_grid, make_hi_grid`

Given M values, x_m, and a set of K bins, $[x_k^{\rm lo}, x_k^{\rm hi})$, this function computes the number of values, n_x, falling within each bin, such that $x_k^{\rm lo} <= x < x_k^{\rm hi}$. It is assumed that $x_{k-1}^{\rm hi} = x_k^{\rm lo}$, e.g. the grid has no holes.

If this function is called with only 2 arguments, a default `hi` grid is constructed such that

```
hi = [lo[[1:K-1]], DBL_MAX].
```

This ensures that the last bin is an "overflow bin" containing the number of values $x_m >$ `lo[K-1]`.

If present, the last optional argument is used to return an array of arrays. Each array element is an array containing the indices of the values falling into the corresponding bin. In other words, using the above notation, `rev[k]` is an array listing the members of bin `k`, so that `length(rev[k]) = nx[k]`.

For example:

```
% Bin some Gaussian distributed random values:
(lo,hi) = linear_grid (-5,5,1024);
x = grand (10000);
nx = histogram (x, lo, hi);
hplot (lo, hi, nx);

% Demonstrate the reverse-index array:
x = grand(10);
(lo, hi) = linear_grid (-2,2,5);
variable rev;
nx = histogram (x, lo, hi, &rev);
m = array_map (Int_Type, &length, rev);
```

```
print(m);
=> 1
   1
   6
   2
   0
```

histogram2d

Purpose: Bin scatter data into a 2-D histogram
Usage: `num[,] = histogram2d (x[], y[], xgrid[], ygrid[] [, &rev])`
See Also: `histogram, plot_contour, plot_image`

Given M points, $(x_m,\ y_m)$, and a grid for each coordinate axis $(xgrid_j, ygrid_k)$, this function computes the number of points, $N(j,k)$, falling within each bin, such that $xgrid_j <= x < xgrid_{j+1}$ for $j = 0, N_j - 1$ and $ygrid_k <= y < ygrid_{k+1}$ for $k = 0, N_k - 1$. The last bin in each row or column is an overflow bin such that its upper limit is at infinity.

If present, the optional argument is used to return an array of arrays such that `rev[i,j]` contains a list of the indices of the values that went into bin `[i,j]`.

howmany

Purpose: Count non-zero array elements
Usage: `n = howmany (x[])`
See Also: `where, any`

This function returns the number of non-zero elements in the specified array.

isis_get_pager

Purpose: Retrieve the current pager definition string
Usage: `s = isis_get_pager ()`
See Also: `isis_set_pager`

isis_set_pager

Purpose: Specify how extensive text output should be displayed
Usage: `isis_set_pager ("pager_command")`
See Also: `isis_get_pager`

By default, ISIS uses the program specified by the `PAGER` environment variable to display extensive text output such as ISIS documentation and tables of emission line parameters. This behavior can be changed by providing a different pager definition. For example, to have text information displayed in a separate window which is opened automatically, do something like this (assuming your favorite pager program is `most`)

```
isis_set_pager ("cat > /tmp/isis.help; xterm -e most /tmp/isis.help &");
```

To simply dump text output to the screen, use

```
isis_set_pager ("cat");
```

To revert to the default behavior, do

```
isis_set_pager (NULL);
```

linear_grid

Purpose: generate a linear histogram grid
Usage: `(binlo[], binhi[]) = linear_grid (min, max, nbins);`
See Also: `make_hi_grid`

This function generates a linear grid of histogram bins such that

```
          binlo[0] = min
      binhi[nbins-1] = max
 binhi[j] - binlo[j] = (max - min) / nbins
 binhi[j] = binlo[j+1]                     for j=0,1,...nbins-2
```

For example, to generate a wavelength grid with 1000 bins extending from 1-20 Å, type

```
(lo, hi) = linear_grid (1, 20, 1000);
```

make_hi_grid

Purpose: Use a single grid array to define a histogram grid
Usage: `hi[] = make_hi_grid (lo[]);`
See Also: `linear_grid`

Given a grid which provides the lower bin edges of a histogram grid, the high edges are defined as

```
hi = [ lo[[1:n-1]], 2*lo[n-1] - lo[n-2] ];
```

so that the last two bins have the same width.

mean

Purpose: Find the average value of an array
Usage: `avg = mean (array)`
See Also: `median, moment`

median

Purpose: Find the median value of an array
Usage: `m = median (array)`
See Also: `mean, moment`

moment

Purpose: Generate statistics for an array
Usage: `Struct_Type = moment (array)`
See Also: `median, mean`

For example:

```
isis> x=grand(1000);         % Gaussian random numbers
isis> s = moment(x);
isis> print(s);
    num = 1000               % number of values
    ave = -0.00860764        % average
    var = 0.918971           % variance
    sdev = 0.95863           % standard deviation
    sdom = 0.0303145         % std dev. of the mean
    min = -2.8576            % smallest
    max = 2.82887            % largest
```

prand

Purpose: Generate Poisson-distributed random values
Usage: `x = prand (rate [,num])`
See Also: `grand, prand, seed_random`

If called with a single argument, this function returns an equal-sized array of Poisson-distributed random values.

```
Example:
   isis> print(prand([3,3,3]));
   1.000000e+00
   0.000000e+00
   8.000000e+00
```

If called with two scalar arguments, the specified number of random values are chosen from the Poisson distribution corresponding to the given rate.

provide

Purpose: Declare that a specified feature is available
Usage: `provide (String_Type feature)`
See Also: `require, _featurep`

The `provide` function may be used to declare that a "feature" has been loaded. See the documentation for `require` for more information.

quit

Purpose: exit ISIS
Usage: `quit`
See Also: `exit`

Use this function to exit ISIS. It is sometimes useful to have a script exit and return an error code to the Unix shell; for this, use `exit (err)`, where `err` is the integer error code.

readcol

Purpose: Read columns from an ASCII file
Usage: `(a, b, ....) = readcol (file, [c1, c2, ....])`
See Also: `writecol`

The values `c1`, `c2`, etc. give the column numbers to be read. For example, to read columns 3 and 6:

```
(x, y) = readcol ("ascii.dat", 3, 6);
```

Lines beginning with a "`#`" character are ignored.

reset

Purpose: reset ISIS
Usage: `reset([force])`
See Also: `quit`

This function frees all allocated memory, closes all plot windows and resets most internal status variables to their internal defaults, but does *not* reload the user's `.isisrc` file.

If invoked with no arguments, `reset` prompts the user to confirm the reset. If invoked with a non-zero value for `force`, no user confirmation is requested.

require

Purpose: Make sure a feature is present, and load it if not
Usage: `require (String_Type feature [,String_Type file])`
See Also: `provide, _featurep, evalfile`

The `require` function ensures that a specified "feature" is present. If the feature is not present, the `require` function will attempt to load the feature from a file. If called with two arguments, the feature will be loaded from the file specified by the second argument. Otherwise, the feature will be loaded from a file given by the name of the feature, with "`.sl`" appended.

If after loading the file, if the feature is not present, a warning message will be issued.

Note that "feature" is an abstract quantity that is undefined here.

A popular use of the `require` function is to ensure that a specified file has already been loaded. In this case, the feature is the filename itself. The advantage of using this mechanism over using `evalfile` is that if the file has already been loaded, `require` will not re-load it. For this to work, the file must indicate that it provides the feature via the `provide` function.

save_input

Purpose: save commands to a disk file
Usage: `save_input [("filename" | File_Type)]`
See Also: `start_log, stop_log, .source`

If no file name is specified, the command log is saved in the file `isis.log`. If the log file already exists, the log is appended to the existing file.

If passed a `File_Type` pointer, the associated file will not be closed by `save_input`.

seed_random

Purpose: Seed the random number generator
Usage: `seed_random (int)`
See Also: `urand, grand, prand`

set_readline_method

Purpose: Select the command prompt readline method
Usage: `set_readline_method ("method");`
See Also:

Use this function to select how the command-line interface should behave.

If ISIS was compiled with GNU `readline`, then GNU `readline` will manage the command line, providing command-line editing and other features.

To use the command-line editing features provided by the S-LANGinterpreter, use

```
set_readline_method ("slang");
```

To turn off the command-line editing features, use

```
set_readline_method ("stdin");
```

start_log

Purpose: save commands to a disk file
Usage: `start_log [("filename")]`
See Also: `stop_log, save_input, .source`

If no file name is specified, the log is saved in the file `isis.log`. If the log file already exists, the log is appended to the existing file.

stop_log

Purpose: turn off command logging
Usage: `stop_log`
See Also: `start_log, .source`

urand

Purpose: Generate Uniformly-distributed random numbers
Usage: `nums = urand ([n [, m ...]])`
See Also: `grand, prand, seed_random`

If no arguments are given, a single random number will be generated. If N integer arguments are provided, they are interpreted as defining the dimensionality of an array which is to be populated with random numbers.

For example:

```
x = urand();         % returns Double_Type
a = urand(10);       % returns Double_Type[10]
b = urand(100,200);  % returns Double_Type[100,200]
```

who

Purpose: list currently defined variables and functions
Usage: `who (["pattern"])`
See Also: `apropos, print`

All variables and functions hose names contain `"pattern"` are listed; if if `"pattern"` is absent, all symbols are listed.

```
For example:

     isis> x=[1,2,3,4];
     isis> y=&x;
     isis> z=x*3.0;
     isis> who;
     x: Integer_Type[4]
     y: &x
     z: Double_Type[4]
     isis>
```

writecol

Purpose: Write arrays to an ASCII file
Usage: `writecol (fp, a, b, ....);`
See Also: `readcol`

The first argument may be either a filename or a file pointer. For example:

```
   % write to a file:
isis> writecol ("ascii.dat", x, y);

   % write to the screen
isis> x=[1:10];
isis> writecol (stdout, x,x,x,x,x,x);
1       1       1       1       1       1
2       2       2       2       2       2
3       3       3       3       3       3
4       4       4       4       4       4
5       5       5       5       5       5
6       6       6       6       6       6
7       7       7       7       7       7
8       8       8       8       8       8
9       9       9       9       9       9
10      10      10      10      10      10
```

7.2 Handling High Resolution Spectra

ISIS manipulates histogram data in which both bin edges are explicitly specified, along with data values and uncertainties in each bin. The input data values in each bin must represent the integral of an underlying continuous distribution over the bin-width. In other words, the input data values must be in bin-integral units like *counts* or *counts/bin*, rather than bin-density units like *counts/Å*. Although the *input* data values must be in bin-integral units, the data can be *displayed* in either bin-integral or bin-density units; see `plot_bin_density`, `plot_bin_integral`.

For each input data set, ISIS maintains storage to hold the associated counts histogram, the flux-corrected data and model spectra in both flux and counts units; the flux-corrected version of the data is optional. When folding a spectral model through the instrument response, ISIS stores both the input spectral model in flux units and the counts histogram obtained by folding that model through the instrument response. Either version of the model or data is accessible through commands such as `get_data_counts`, `plot_model_flux`, etc.

For each dataset, the counts histogram, $D(h)$, is referred to as `data_counts`. The corresponding predicted counts histogram, $C(h)$, is referred to as `model_counts` and is computed according to equation (7.35). Flux-corrected data, $\bar{S}(h)$, is derived from the counts histogram, $D(h)$, and the instrument responses according to equation (7.4). The term `model_flux` refers to the bin-integrated spectral model, $S(E)dE$. The term `convolved_model_flux` refers to the spectrum model obtained by folding the model, $S(E)dE$ through the instrument response, $RMF(h, E)$, with unit effective area, $A(E)$.

ISIS also maintains a separate, rebinned version of the data to support fitting rebinned data "on-the-fly". With ISIS, it is not necessary to run a separate program to generate a rebinned spectrum file. Because the rebinned data is stored separately from the input data, reverting to the input grid is a trivial operation.

When folding data through the instrument response, one can also rebin the instrument response matrix (RMF) using `rebin_rmf`. More often, it may be useful to simultaneously rebin the RMF and the corresonding dataset using `rebin_dataset`. See `rebin_rmf` for details.

Histograms may be loaded from FITS Type I or Type II pha files or from ASCII files (see `load_data`, `define_back`). ISIS can handle multiple spectra simultaneously; reading successive files appends the contents of each file to the internal list of data sets. `list_data` displays a list of the currently loaded spectra and `delete_data` removes items from the list. Type II pha files can contain both count histograms and flux-corrected spectra; both are accessible.

Spectrum plots are generated using `[o]plot_data_counts` and `[o]plot_data_flux`; model fits computed on the same grid as the data may be displayed using `[o]plot_model_counts` and `[o]plot_model_flux`. The data values may be accessed using the functions `get_data_counts` and `put_data_counts`, and similar functions for flux-corrected data. Similar functions are available for obtaining model spectrum values, either in flux-units or in counts (as usually determined by folding a source spectrum through the instrument response).

Effective area functions (ARFs) in both Type I and Type II formats may be manipulated in a similar way using `load_arf`, `list_arf`, `delete_arf`, `get_arf` and `put_arf`. Event Redistribution Matrix Functions (RMFs) are accessible using using `load_rmf`, `list_rmf`, `assign_rmf`, `unassign_rmf`, and `delete_rmf`. Observed spectra may be "flux-corrected" using `flux_corr`.

Positions of higher-order spectral features may be displayed using `lambda_mth_order`.

all_arfs

Purpose: Get a list of indices for all currently loaded ARFs
Usage: `ids = all_arfs();`
See Also: `load_arf, delete_arf, all_rmfs, all_data`

This function is useful when you want to do something to all currently loaded ARFs. For example

```
isis>  delete (all_arfs);
```

all_data

Purpose: Get a list of data-set indices for all currently loaded data-sets
Usage: `ids = all_data([noticed]);`
See Also: `load_data, exclude, include, ignore, notice`

This function is useful when you want to do something to all currently loaded data sets. For example

```
% to ignore all data sets
isis>  ignore (all_data);

% to get info on all % data sets
isis>  info = get_data_info (all_data);
```

If the optional argument (`noticed`) is non-zero, the function returns only the indices of data sets which have noticed bins.

all_rmfs

Purpose: Get a list of indices for all currently loaded RMFs
Usage: `ids = all_rmfs();`
See Also: `load_rmf, delete_rmf, all_arfs, all_data`

This function is useful when you want to do something to all currently loaded RMFs. For example

```
isis>  delete (all_rmfs);
```

assign_arf

Purpose: Assign an ARF to one or more spectra
Usage: `assign_arf (arf_index[], hist_index[])`
See Also: `load_arf, list_arf, unassign_arf, flux_corr, assign_rsp`

After loading an ARF, it may be associated with one or more histograms by specifying the index of the ARF and the histograms. This indicates which ARF function should be used when computing flux-corrected spectra and for fitting. If an RMF is applied, the ARF and RMF grids must match exactly.

```
Example:
        assign_arf (2, 1);             % ARF 2 goes with spectrum 1

        assign_arf (2, [3:6]);         % Spectra 3, 4, 5 and 6 should
                                       % use ARF 2
```

When a dataset has been assigned an RMF that includes an ARF factor, assigning another ARF will generate a warning message, but the ARF will be assigned anyway. To prevent assigning such additional ARF factors, set the intrinsic variable `Allow_Multiple_Arf_Factors` to a negative value. To allow additional ARF factors to be assigned without complaint, set `Allow_Multiple_Arf_Factors=1`. The default behavior corresponds to `Allow_Multiple_Arf_Factors=0`.

assign_back

Purpose: Define the background counts of one dataset using the model counts spectrum from another
Usage: `assign_back (source_index[], destination_index[])`
See Also: `_define_back, define_back, unassign_back`

Use this function to copy the model counts spectrum from one dataset to the background counts spectrum of another dataset. The `BACKSCALE` and `EXPOSURE` values from the source dataset are also copied to the destination dataset.

```
Example:
   % Use model counts spectrum from dataset #2
   % to define the background for dataset #1
        assign_back (2, 1);

   % Use model counts spectrum from dataset #2
   % to define the background for datasets 3,4,5,6
        assign_back (2, [3:6]);

   % Remove the background assigned to dataset #3
   % (same as unassign_back(3))
        assign_back (NULL, 3);
```

If an ARF has been assigned to the source dataset, the ARF exposure time is used to define the background exposure time, otherwise, the data exposure time is used. If multiple ARFs have been assigned to the source dataset, the exposure time of the first ARF is used.

assign_rmf

Purpose: Assign an RMF to one or more spectra
Usage: `assign_rmf (rmf_index[], hist_index[])`
See Also: `load_rmf, list_rmf, unassign_rmf, assign_rsp`

After loading an RMF, it may be associated with one or more histograms by specifying the indices of the RMF and the histograms. This indicates which RMF functions should be used when fitting models to data.

```
Example:
        assign_rmf (2, 1);            % RMF 2 goes with spectrum 1

        assign_rmf (2, [3:6]);        % Spectra 3, 4, 5 and 6 should
                                      % use RMF 2
```

Note that the ARF and RMF grids must match exactly; relative tolerances on the accuracy of the grid mismatch are controlled by the intrinsic variable `Rmf_Grid_Tol`. If the RMF grid match is inexact but "close enough" one can suppress grid-mismatch errors by setting `Rmf_Grid_Tol` to a small positive value. For example, `Rmf_Grid_Tol=0.001` would indicate that acceptable mismatches must be smaller than 1 part in 1000.

assign_rsp

Purpose: Assign one or more ARF/RMF pairs to one or more spectra
Usage: `assign_rsp (arf_list, rmf_list, hist_index_list)`
See Also: `load_arf, load_rmf, assign_arf, assign_rmf`

Although `arf_list` and `rmf_list` can refer to a single ARF or RMF, this function is primarily intended to support assigning multiple responses to one or more datasets (e.g. for fitting LETG data which must include contributions from higher orders). The responses listed are applied pairwise (e.g. `arf_list[k]` goes with `rmf_list[k]`).

```
For example:
   % Assuming ARFs 1-10 and RMFs 1-10 correspond to
   % responses for dispersed orders 1-10:
        assign_rsp ([1:10], [1:10], 1);

   % Assign ARF #2 and RMF #3 to dataset 1.
        assign_rsp (2, 3, 1);
```

The wavelength grid for the data is taken from the first RMF in the `rmf_list`. Therefore, in analyzing data containing multiple dispersion orders, the first element of `arf_list` and the first element of `rmf_list` should normally correspond to first-order.

Generating plots which compare the contribution from different dispersion orders can be somewhat tricky to generate. Here's how to examine the 3rd order contribution to LETG/HRC data using a given spectral model:

```
   % Assuming ARFs 1-10 and RMFs 1-10 correspond to
   % responses for dispersed orders m=1-10:

        assign_rsp ([1:10], [1:10], 1);

   % First overplot the data with a model including
   % orders m=1-10.  Note that this plot uses the
   % 1st order wavelength grid.

        plot_data_counts (1);
```

```
      () = eval_counts;
      oplot_model_counts (1);

   % Now evaluate the 3rd order contribution and
   % overplot it using the 1st order wavelength grid

      assign_rsp (3,3, 1);
      () = eval_counts;
      assign_rsp (1,1, 1);
      oplot_model_counts (1);
```

Values of `arf_index`=0 or `rmf_index`=0 imply the corresponding identity response (e.g. ARF=1 or RMF=1).

combination_members

Purpose: Get a list of combined datasets
Usage: `list = combination_members (gid)`
See Also: `combine_datasets, match_dataset_grids, uncombine_datasets, get_combined, get_combined2, rebin_combined, set_pre_combine_hook`

combine_datasets

Purpose: Combine several datasets to improve the fit-statistics
Usage: `gid = combine_datasets (list [, weights])`
See Also: `combination_members, match_dataset_grids, uncombine_datasets, get_combined, get_combined2, set_eval_grid_method, rebin_combined, set_pre_combine_hook`

This function may be used to label several datasets which should be combined to improve statistics during a model fit. This function may also be used to help solve coupled systems of equations describing multiple sources which are only marginally resolved.

Datasets to be combined must have identical grids; all spectral bins must be the same and the same bins must be noticed in each. Use `match_dataset_grids` to put several datasets onto a common spectral grid.

The optional `weights` argument specifies weights which are used when combining the datasets. In particular, the value of bin k in the combined dataset is

$$D_k = \sum_i f_i D_{k,i} \tag{7.1}$$

where the sum extends over all datasets and where f_i is the weight corresponding to dataset i.

The return value is the index of the dataset combination. Multiple dataset combinations are supported.

Combining datasets in this way is conceptually equivalent to summing datasets, but is somewhat more consistent because the models for the individual datasets are treated consistently for any fit-kernel.

For example, the χ^2 fit statistic for the combined datasets is computed as

$$\chi^2 = \sum_k W_k \left(\sum_i f_i D_{ki} - M_{ki} \right)^2 \tag{7.2}$$

where $D_{k,i}$ and $M_{k,i}$ are the data and model values, respectively, for bin k of dataset i and where W_k is the statistical weight for bin k in the combined dataset. For Poisson statistics,

$$W_k = \frac{1}{\sum_i f_i D_{ki}}. \tag{7.3}$$

When working with combined datasets, one can use the `ignore/notice` functions as long as each member is treated the same way. For example:

```
% create a dataset group:
 g = [1, 2, 3, 4];
 match_dataset_grids (g);
 gid = combine_datasets (g);

% ignore the same data-range in every group member
 ignore (g, 13.4, 14.2);

% fit models in the usual way
 () = fit_counts;
```

One may also combine datasets to analyze data for coupled sources. For example, consider an observation of 2 sources (`a,b`) which are only marginally resolved. Suppose the spectrum model for source (`a`) is `S_A` and the model for source (`b`) is `S_B`. Separate spectra (`D_a, D_b`) may be extracted, but each spectrum is contaminated by the other so that, in general, the 2 datasets are represented by a system of equations of the form

```
D_a = R_Aa * S_A + R_Ba * S_B;
D_b = R_Ab * S_A + R_Bb * S_B;
```

in which both source models contribute to each dataset through a set of responses `R_xy`. In this expression, the products `R*S` are intended as a shorthand notation to represent folding the source model `S` through the (possibly nonlinear) instrument response `R`.

To solve this system of coupled equations, one can do the following:

```
load_dataset (D_a, R_Aa);    % dataset #1
load_dataset (D_a, R_Ba);    %         #2
load_dataset (D_b, R_Ab);    %         #3
load_dataset (D_b, R_Bb);    %         #4

weights = [0.5, 0.5];

combine_datasets (1,2, weights);
combine_datasets (3,4, weights);

define coupled_sources_model()
{
    switch (Isis_Active_Dataset)
```

```
        { case 1 or case 3:  return S_A(); }
        { case 2 or case 4:  return S_B(); }
    }

    fit_fun ("coupled_sources_model()");
```

Consider how this works for the combination of datasets 1 and 2. Evaluating the coupled source model for dataset 1 yields the product `R_Aa*S_A` and evaluating it for dataset 2 yields the product `R_Ba*S_B`. Combination of the models for datasets 1 and 2 then yields

```
R_Aa*S_A + R_Ba*S_B
```

while the weighted combination of the datasets themselves yields

```
0.5*D_a + 0.5*D_a = D_a.
```

(And similarly for the combination of datasets 3 and 4). It follows that minimizing the fit-statistic for all 4 datasets simultaneously yields the solution to the system of 2 coupled equations.

copy_data_keywords

Purpose: Copy keywords values between two data sets
Usage: `copy_data_keywords (to_id, from_id)`
See Also: `load_data`

cursor_counts

Purpose: Compute statistics for a given wavelength region
Usage: `cursor_counts (hist_index [, out_file [, flag]])`
See Also: `cursor_flux, region_counts, region_flux`

.

This function is analogous to the `region_counts` function except that 1) the input `xmin`, `xmax`, `ymin`, `ymax` values are taken from reading the cursor position on a data plot and 2) the computed statistics are automatically saved to a file. If `flag` is one or is not set, the statistics are continuum subtracted; if `flag` is zero, the continuum is assumed to be zero.

cursor_flux

Purpose: Compute statistics for a given wavelength region
Usage: `cursor_flux (hist_index [, out_file [, flag]])`
See Also: `cursor_counts, region_counts, region_flux`

. See `cursor_counts`.

define_arf

Purpose: Define an ARF using S-Lang arrays
Usage: `id = define_arf (Struct_Type | binlo, binhi, arf, arf_err)`
See Also: `set_arf_exposure, set_arf_info, get_arf_info`

This function provides a way to define a new ARF using S-Lang arrays. As input, it accepts 1) a Struct_Type with fields `bin_lo, bin_hi, value, err` or 2) a list of four equal-length arrays with the same data. The `bin_lo`, `bin_hi` arrays provide a wavelength grid in Angstrom units, sorted in ascending order. The new ARF is added to the internal list just as though the data had been loaded from a FITS data file. Normally, the function returns the integer index of the new data-set. If the function fails, the return value is -1.

Note that when an ARF is defined in this way, one must set the associated exposure time explicitly using either `set_arf_exposure` or `set_arf_info`.

define_back

Purpose: Define a background spectrum file
Usage: `status = define_back (index, "file")`
See Also: `define_counts, define_flux, load_data, back_fun, _define_back, get_back`

Use this function to specify a background spectrum for a data set (e.g. the $B(h)$ term in equation (7.35)). The background spectrum file format may be either ASCII or OGIP/FITS Type-I PHA. If the FITS format is used, the BACKSCAL and EXPOSURE keywords are used to scale the background relative to the data spectrum. If the ASCII format is used, no re-scaling is performed.

To unassign the background, use `define_back(index,NULL)`.

For example:

```
() = define_back (1, "background.pha");
```

_define_back

Purpose: Define a background spectrum using S-Lang variables
Usage: `status = _define_back (index, bgd [, area [, exposure]])`
See Also: `define_counts, define_flux, load_data, back_fun, define_back, get_back`

Use this function to specify a background spectrum for a data set (e.g. the $B(h)$ term in equation (7.35)). The optional values of `area` and `exposure` are used to scale the background relative to the data spectrum. The array `bgd` containing the background values must be on an ascending wavelength grid and must match the data set grid; the `rebin` function may be used to match the grids if necessary).

To unassign the background, use `_define_back(index,NULL)`.

For example:

```
bgd_area = 400.0;     % extraction region [pixels^2]
bgd_exposure = 4.e4;  % exposure time [sec]
() = _define_back (1, bgd_array, bgd_area, bgd_exposure);
```

define_counts

Purpose: Define a counts-histogram using S-Lang arrays
Usage: `s = define_counts (Struct_Type | bins | [lo, hi,] counts [, err])`
See Also: `define_flux, define_back, Minimum_Stat_Err`

This function provides a way to define a new data-set using S-Lang arrays. As input, it accepts 1) a Struct_Type with fields `bin_lo, bin_hi, value, err` or 2) a list of four equal-length arrays with the same data or 3) a single array containing only the bin values. The new data-set is added to the internal list just as though the data had been loaded from an ascii or FITS data file. Normally, the function returns the integer index of the new data-set. If the function fails, the return value is -1.

The wavelength grid arrays (`bin_lo, bin_hi`) and the uncertainty (`err`) arrays are optional. If the wavelength grid arrays are shorter than the `counts` array (or are missing), they are ignored, and the data grid is assumed to be supplied by an RMF. If the uncertainty array is shorter than the `counts` array (or is missing), a default uncertainty array will be supplied assuming Poisson statistics.

```
Qualifiers
----------
 min_stat_err     require stat_err >= min_stat_err
```

By default, any input uncertainty values smaller than `min_stat_err` are reset according to

```
   stat_err = max[ sqrt(counts), min_stat_err ]
```

where `min_stat_err` is the smallest acceptable (positive) uncertainty value. When the `min_stat_err` qualifier is not present, the value of the intrinsic variable `Minimum_Stat_Err` is used if positive, otherwise, the default minimum uncertainty is 1. To shut off warnings about invalid uncertainties being replaced, set `Warn_Invalid_Uncertainties=0`.

define_flux

Purpose: Define a flux-corrected histogram using S-Lang arrays
Usage: `s = define_flux (Struct_Type | lo, hi, flux, err)`
See Also: `define_counts`

This function is similar to `define_counts` except that it is used to define a flux-corrected histogram.

delete_arf

Purpose: Delete one or more ARFs from the internal table
Usage: `delete_arf (arf_index_list)`
See Also: `list_arf`

This function removes the indicated ARFs from the internal list; it does *not* affect the FITS file containing the ARF.

```
Example:
      isis> delete_arf (3);
      isis> delete_arf ([4,8,9]);
```

delete_data

Purpose: delete spectra from the internal list
Usage: `delete_data (hist_index_list)`
See Also: `list_data`

This function removes the indicated spectra from the internal list; it does not affect the disk files containing the spectra.

```
Example:
      isis> delete_data (3);
      isis> delete_data ([4,8,9]);
```

delete_rmf

Purpose: Delete one or more RMFs from the internal table
Usage: `delete_rmf (rmf_index_list)`
See Also: `list_rmf`

This function removes the indicated RMFs from the internal list; it does *not* affect the file containing the RMF definition.

```
Example:
      isis> delete_rmf (3);
      isis> delete_rmf ([4,8,9]);
```

factor_rsp

Purpose: Factor a response matrix (RSP) into an ARF and a normalized RMF
Usage: `arfs = factor_rsp (rmfs)`
See Also: `load_rmf, list_arf, flux_corr, eval_flux`

Some response matrices are defined as the product of the instrument effective area (the ARF) and the instrument redistribution function (the RMF).

After loading such a response matrix with `load_rmf`, this function factors out the effective area and appends the corresponding ARF function to the internal list of effective area functions. In the process, the response matrix is renormalized such that the redistribution function for each incident photon energy is unit normalized.

If the function succeeds, it returns the indices of the ARFs appended to the internal list. If an error occurs, the function returns -1.

```
EXAMPLE:

 % load a response matrix which includes the
 % effective area and the redistribution function

 rsp = load_rmf ("rsp.fits");

 % factor out the effective area function
 % and normalize the redistribution function

 arf = factor_rsp (rsp);

 % assign the ARF and RMF to a dataset of interest

 assign_rmf (rsp, data_index);
 assign_arf (arf, data_index);
```

fakeit

Purpose: Generate fake data using a given model, ARF and RMF
Usage: `fakeit ([&noise_fun])`
See Also: `load_arf, load_rmf, fit_fun, set_frame_time, set_arf_exposure, define_back, set_fake`

This function may be used to generate fake data with user-defined uncertainties for a list of ARF, RMF pairs (any real datasets which happen to be loaded at the time will not be overwritten). The exposure time is taken from the ARF. If a noise function (`noise_fun`) is not provided, Poisson statistics are assumed; to generate noiseless data, use `fakeit (NULL)`.

For example, to generate a single fake data set, first load a matching ARF and RMF and assign them to a non-existent data set index, causing ISIS to generate an empty data set. For example:

```
load_arf ("arf.fits");
load_rmf ("rmf.fits");

assign_arf (1,1);
assign_rmf (1,1);
```

Having created an empty data set, the next step is to populate it with fake data.

To do that, first define a spectral model using `fit_fun`. For example, one might use the XSPEC module:

```
require ("xspec");                % this is optional
fit_fun ("phabs(1)*mekal(1)");
```

Adjust the model parameters to the values desired for the fake data set.

Now, use `fakeit` to populate the counts vector for this data set, plus Poisson errors:

```
fakeit;
```

```
rplot_counts (1);
```

Here, we've also used `rplot_counts` to plot the data, model and residuals.

A user-defined noise function should take the model value for a given bin as an argument and return the noise-added value. For example, to add uniformly distributed noise with a 10% amplitude use:

```
define my_noise (model_in)
{
    variable r = 2.0 * (urand(1) - 0.5);
    return model_in * (1.0 + 0.1 * r);
}

fakeit (&my_noise);
```

Each call to 'fakeit' regenerates all fake spectra. To keep a particular fake spectrum from being over-written by subsequent calls to 'fakeit', use `set_fake` to mark that spectrum as 'real'.

flux_corr

Purpose: Compute the flux-corrected spectrum
Usage: `flux_corr (hist_index [,threshold])`
See Also: `flux_corr_model_counts, load_arf, assign_arf, unassign_arf, back_fun`

This function computes the "flux-corrected" spectrum, $\bar{S}(h)$, defined by the expression

$$\bar{S}(h) \equiv \frac{C(h) - B(h)}{t \int_{\Delta E(h)} \mathrm{d}E \; \mathcal{F}\Big(R(h,E), A(E), 1\Big)} \tag{7.4}$$

where $A(E)$ is the effective area (the ARF) at energy E, $R(h,E)$ is the redistribution function (the RMF), $C(h)$ is the number of source counts in detector bin h and t is the exposure time (from the ARF EXPOSURE keyword). In this expression, the fit "kernel", $\mathcal{F}(R,A,s)$ is evaluated for a constant spectrum, $s(E) = 1$, and defaults to

$$\mathcal{F}\,(R, A, 1) = R(h,E)A(E) \tag{7.5}$$

for the standard kernel. The integral spans the energy range, $\Delta E(h)$, that contributes to detector bin h (e.g. all energies represented by the response).

In general, the degree to which this transformation produces a useful result depends on the condition of the data, the characteristics of the instrument response and the effect of the relevant kernel, $\mathcal{F}$. For the standard kernel, in the limit that $R(h,E)$ approaches a delta-function, the flux-corrected spectrum, $\bar{S}(h)$, approaches the model spectrum, $s(E)$. For example, flux-correcting high-resolution grating spectra unaffected by photon pileup often provides a good estimate of the incident spectrum (but uncorrected for blurring due to the line spread function). With moderate levels of pileup, the flux-corrected spectrum may also yield a good estimate of the incident spectrum. For CCD resolution spectra which are unaffected by photon pileup, the flux estimate may be reasonably good for energies above ~ 1 keV and may be useful for visualization purposes. But for $E \lesssim 1$ keV, the CCD RMF becomes rather broad and the resulting estimate $\bar{S}(h)$ may differ significantly from the incident spectrum $s(E)$.

Note that the relevant instrumental background, $B(h)$, is automatically subtracted (if available) to compute the number of source counts $C(h)$. The flux result is stored separately and does not over-write the counts histogram. If the counts histogram is re-grouped or rebinned after being flux-corrected, `flux_corr` should be re-run to compute the flux values on the new grid.

For the standard fit-kernel, uncertainties on the flux-corrected spectrum are computed directly from the counts so that

$$\bar{S}_k = \frac{C_k - B_k}{I_k} \tag{7.6}$$

$$\delta\bar{S}_k = \frac{\sqrt{C_k + B_k}}{I_k} \tag{7.7}$$

where k is the bin index and

$$I_k \equiv t \int_{\Delta E(k)} \mathrm{d}E \; R(k,E)A(E) \tag{7.8}$$

The optional argument `threshold` (= 0.0 by default) specifies the detection limit in terms of the minimum acceptable signal-to-noise ratio ($S/N = C_k/\delta C_k$); bins which fall below this limit are assigned a flux of zero.

flux_corr_model_counts

Purpose: Compute a flux-corrected model spectrum
Usage: `flux_corr_model_counts (hist_index [, threshold])`
See Also: `flux_corr, load_arf, assign_arf, unassign_arf, back_fun`

This function computes the "flux-corrected" model spectrum, $\bar{S}^{(m)}(h)$, defined by the expression

$$\bar{S}^{(m)}(h) \equiv \frac{\int_{\Delta E(h)} \mathrm{d}E \; \mathcal{F}\Big(R(h,E), A(E), s(E)\Big)}{\int_{\Delta E(h)} \mathrm{d}E \; \mathcal{F}\Big(R(h,E), A(E), 1\Big)} \tag{7.9}$$

where $A(E)$ is the effective area (the ARF) at energy E, $R(h,E)$ is the redistribution function (the RMF) mapping into detector bin h. In the numerator of this expression, the model counts are predicted using the fit "kernel" $\mathcal{F}(R,A,s)$ as applied to the model spectrum, $s(E)$. In the denominator, this "kernel" is evaluated for a constant spectrum, $s(E) = 1$. Recall that the standard kernel is

$$\mathcal{F}\,(R,A,s) = R(h,E)A(E)s(E). \tag{7.10}$$

The integrals span the energy range, $\Delta E(h)$, that contributes to detector bin h (e.g. all energies represented by the response).

The result, $\bar{S}^{(m)}(h)$, over-writes the internal array used to store the convolved model flux; to retrieve the numerical values, use `get_convolved_model_flux()`.

Because the current implementation works only on an unbinned model spectrum, the model must be computed on the same grid as the unbinned data. See `flux_corr` for further details.

get_arf

Purpose: Get numerical values from an ARF
Usage: `Struct_Type = get_arf (arf_index)`
See Also: `put_arf, list_arf`

.

This function retrieves an ARF function from the internal list and loads the data into the fields of a S-Lang structure.

```
arf_index = integer index of ARF in internal list
 s.bin_lo = bin left edge [Angstrom]
 s.bin_hi = bin right edge [Angstrom]
  s.value = ARF value [cm^2 counts/photon]
    s.err = ARF uncertainty [cm^2 counts/photon]
```

get_arf_info

Purpose: Get ARF information
Usage: `Struct_Type = get_arf_info (id)`
See Also: `set_arf_exposure, set_arf_info, get_arf_info`

This function returns a structure which contains the values of auxiliary ARF parameters:

```
isis> s=get_arf_info(1);
isis> print(s);
    order = -1
    part = 2
    srcid = 0
    exposure = 28123.4
    fracexpo = 1
```

get_combined

Purpose: Retrieve the combined dataset or corresponding model
Usage: `Struct_Type = get_combined (gid, &get_function)`
See Also: `get_combined2, combination_members, combine_datasets, uncombine_datasets, match_dataset_grids, rebin_combined, set_pre_combine_hook`

Use this function to obtain the specified dataset combination or the corresponding model combination. The return value is a `Struct_Type` of the form

```
    struct {bin_lo, bin_hi, value, err}
```

where the (`bin_lo, bin_hi`) fields give the wavelength grid, the (`value`) field gives the sum of the specified histograms and the (`err`) field gives the corresponding uncertainty. These sums are computed using the combination weights specified via `combine_datasets`:

$$D_k = \sum_i f_i D_{ik} \delta D_k^2 = \sum_i (f_i \delta D_{ik})^2 \tag{7.11}$$

where D_k is the summed value in bin k, f_i is the weight for dataset i and δD_k is the uncertainty of D_k. Because models are assumed to be exact, models usually have a `NULL` value in the `err` field.

Note that it is assumed that the function used as the second argument (e.g. `get_function`) returns spectra with matching grids. Any function returning a result which uses the data grid is acceptable because `match_dataset_grids` will ensure that the specified data grids all match.

In contrast, one cannot in general use `get_model_flux` here because the model flux is computed on the ARF grid and the ARF grids need not match.

EXAMPLE:

```
% To plot the sum of the combined datasets
% and over-plot the model for the combination:

match_dataset_grids (3,4,9,10);
gid = combine_datasets (3,4,9,10);
() = eval_counts ();

d = get_combined (gid, &get_data_counts);
m = get_combined (gid, &get_model_counts);

hplot(d);
ohplot(m);
```

Keep in mind that `get_combined` always computes the sum of vectors provided by the second argument. If those vectors represent counts, then the computed sum yields the total counts. However, if those vectors represent flux, then the computed sum is probably not what was intended – in this case, an exposure weighted mean is more likely to be useful. To use `get_combined` to generate an exposure weighted mean, the second argument might point to a custom function of this form:

```
define exposure_weighted_model_flux (i)
{
   variable f, info, exposure, weight;
   f = get_model_flux (i);
   info = get_data_info (i);
   exposure = get_arf_exposure (info.arfs[0]);
   weight = exposure / Total_Exposure_Time;
   f.value *= weight;
   return f;
}
mean_flux = get_combined (g, &exposure_weighted_model_flux);
```

In this example, it is assumed that the global variable `Total_Exposure_Time` has been previously computed.

get_combined2

Purpose: Retrieve the combined dataset or corresponding model
Usage: `Struct_Type[] = get_combined2 (gid[])`
See Also: `get_combined, combination_members, combine_datasets, uncombine_datasets, match_dataset_grids, rebin_combined, set_pre_combine_hook`

Use this function to retrieve dataset combination(s) from the internal table. If the list of combination ids is `NULL`, then the entire list is returned. The return value is a structure or an array of structures with struct fields:

```
__Field__     __Meaning__
combo_id     combination id number
bin_lo       histogram bin lower edge [Angstrom]
bin_hi       histogram bin upper edge [Angstrom]
model        summed model spectrum
data         summed data spectrum
err          uncertainty on summed data values
indices      <internal use only>
```

get_convolved_model_flux

Purpose: load spectral model into a S-Lang structure
Usage: `Struct_Type = get_convolved_model_flux (hist_index)`
See Also: `flux_corr_model_counts, get_model_counts, get_model_flux`

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
   s.value = bin value [photons/sec/cm^2]
     s.err = bin uncertainty [photons/sec/cm^2]
```

This function retrieves the specified convolved flux histogram from the internal list and loads the data into a structure. Similar functions are available to retrieve counts data and model values.

The convolved model flux is computed using the expression

$$F(h) = \frac{\int dE R(h,E) A(E) s(E)}{\int dE R(h,E) A(E)}. \tag{7.12}$$

get_data_backscale

Purpose: Retrieve the background scaling for a given spectrum
Usage: `area = get_data_backscale (hist_index)`
See Also: `set_data_backscale`

Returns BACKSCAL keyword from the header for histogram `hist_index`. Grating data may have a BACKSCAL vector of the same length as the data array; note that in this case, some BACKSCAL values may be zero, corresponding to wavelength values which fall off of the detector.

get_data_counts

Purpose: load spectral data into a S-Lang structure
Usage: `Struct_Type = get_data_counts (hist_index)`
See Also: `put_data_counts, get_model_counts, get_data_info`

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
```

```
   s.value = bin value [counts]
     s.err = bin uncertainty [counts]
```

This function retrieves the specified counts histogram from the internal list and loads the data into a structure. Similar functions are available to retrieve flux-corrected data and model values.

get_data_exposure

Purpose: Retrieve the exposure time for a given spectrum
Usage: `t = get_data_exposure (hist_index)`
See Also: `set_arf_exposure`

Returns exposure time in seconds from the header for histogram `hist_index`.

get_data_flux

Purpose: load spectral data into a S-Lang structure
Usage: `Struct_Type = get_data_flux (hist_index)`
See Also: `put_data_flux, get_model_flux, flux_corr, get_data_info`

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
   s.value = bin value [photons/sec/cm^2]
     s.err = bin uncertainty [photons/sec/cm^2]
```

This function retrieves the specified flux-corrected histogram from the internal list and loads the data into a structure. See `flux_corr` for details on flux-correcting counts spectra.

get_data_info

Purpose: load spectrum parameters a S-Lang structure
Usage: `Struct_Type[] = get_data_info (index_list)`
See Also: `set_data_info, load_data, list_data, array_struct_field`

This function returns an array of structures whose fields contain the `spec_num, order, part` and `srcid` keywords for each data set listed in the `index_list` along with the `target` name string and the observation start time, `tstart` and the frame time, `frame_time`. This structure also contains `notice` and `notice_list` arrays indicating which data bins are currently noticed (for model fitting) and a flag array, `rebin`, which indicates which of the original bins have been grouped together (see `rebin_data` for details). In addition, the returned structure contains the indices for the ARF(s) and RMF(s) assigned to the data set. The `exclude` field indicates whether or not the dataset is currently excluded from the fit. The names of the spectrum file (`file`) and associated background file (`bgd_file`) are also provided.

```
Example:
    isis> id = load_data ("o1318_heg+1_pha.fits");
    isis> s=get_data_info(id);
    isis> print(s);
        spec_num = 1
```

```
        order = 1
        part = 1
        srcid = 1
        exclude = 0
        combo_id = 0;
        combo_weight = 1;
        target = CAPELLA
        tstart = 7.5667e+07
        frame_time = 3.2
        arfs = Integer_Type[1]
        rmfs = Integer_Type[1]
        notice = Integer_Type[8192]
        notice_list = Integer_Type[8192]
        rebin = Integer_Type[8192]
        file = o1318_heg+1_pha.fits
        bgd_file =
```

To extract a specific field from a returned array of structures, use `array_struct_field`.

```
Example:
   isis> d=load_data ("pha2.fits");
   Reading: ...........
   isis> info = get_data_info(d);
   isis> part = array_struct_field(info, "part");
   isis> part;
   Integer_Type[12]
```

get_dataset_metadata

Purpose: Retrieve user-defined meta data associated with a dataset
Usage: `meta = get_dataset_metadata (hist_index)`
See Also: `set_dataset_metadata`

Use this function to retrieve arbitrary (user-defined) metadata associated with a particular dataset. The ability to assign arbitrary metadata to a dataset and later retrieve it can be useful to support analysis techniques unforseen during isis development.

get_flux_corr_weights

Purpose: Retrieve the weights used to perform flux-correction
Usage: `wt[] = get_flux_corr_weights (hist_index)`
See Also: `flux_corr`

Define

$$W(h) = t \int dy R(h, y) A(y). \tag{7.13}$$

This function returns the array $W(h)$ at full resolution. In the linear regime, the flux-corrected data and uncertainties are defined to be

$$f(h) = (C(h) - B(h))/W(h) \tag{7.14}$$
$$df^2(h) = (C(h) + B(h))/W^2(h) \tag{7.15}$$

Summing over several bins h, to make a wider bin H, a rebinned version is:

$$f(H) = \frac{\sum_h C(h) - \sum_h B(h)}{\sum_h W(h)} \tag{7.16}$$

$$df^2(H) = \frac{\sum_h C(h) + \sum_h B(h)}{(\sum_h W(h))^2} \tag{7.17}$$

If we define weights

$$w(h) = \frac{W(h)}{\sum_h' W(h')} \tag{7.18}$$

we can rewrite the rebinned version, $f(H)$ and $df(H)$, as:

$$f(H) = \sum_h w(h) f(h) \tag{7.19}$$

$$df^2(H) = \sum_h w^2(h) df^2(h) \tag{7.20}$$

This approach allows one to perform arbitrary rebinning after computing the $f(h)$, $df(h)$ and $W(h)$ values only once.

get_frame_time

Purpose: Get the CCD frame-time for a data set
Usage: `frame_time_sec = get_frame_time (hist_index)`
See Also: `set_frame_time, get_data_info`

This function is normally used in conjunction with the CCD photon pileup model. The frame time is specified in units of seconds.

get_back

Purpose: Retrieve the background for a given spectrum
Usage: `b = get_back (hist_index)`
See Also: `define_back, _define_back, back_fun, get_back_data, get_back_model, get_back_data_scale_factor`

Returns an array of instrumental background values. If no `back_fun` has been specified, the scaled instrumental background is returned, otherwise the result of the specified background model is returned. These values are on the same wavelength grid as that returned by e.g. `get_data_counts`.

get_back_data

Purpose: Retrieve the un-scaled instrumental background for a given spectrum
Usage: `b = get_back_data (hist_index)`
See Also: `define_back, _define_back, get_back_data_scale_factor, get_back_model, get_back`

```
Qualifiers
----------
 rebin        If present, return rebinned array.
              Otherwise, return full resolution.
```

Returns an array of instrumental background values with no scaling applied.

get_back_data_scale_factor

Purpose: Retrieve the background scale factor for a given spectrum
Usage: `scale = get_back_data_scale_factor (hist_index)`
See Also: `define_back, _define_back, get_back_data, get_back_model, get_back`

```
Qualifiers
----------
 rebin        If present, return rebinned array.
              Otherwise, return full resolution.
```

Returns the factor used to scale the instrumental background. The scale factor is defined as

```
  scale = (data_backscale * exposure) / (back_backscale * back_exposure)
```

The `exposure` time value used for the data is normally taken from the ARF. If the ARF exposure is unspecified or invalid, the data exposure value is used. Either `backscale` value may be a vector quantity, defined on the same wavelength grid as the data.

get_back_model

Purpose: Retrieve the instrumental background model for a given spectrum
Usage: `b = get_back_model (hist_index)`
See Also: `back_fun, define_back, _define_back, get_back_data_scale_factor, get_back`

```
Qualifiers
----------
 rebin        If present, return rebinned array.
              Otherwise, return full resolution.
```

Returns the value of the current instrumental background model, specified by `back_fun`.

get_rmf_arf_grid

Purpose: Get ARF grid from an RMF
Usage: `Struct_Type = get_rmf_arf_grid (rmf_index)`
See Also: `list_rmf, get_rmf_data_grid`

.

This function retrieves the ARF grid from a specific RMF, returning a S-Lang structure containing a wavelength grid (angstrom units) in monotonic increasing order. The structure returned has the form

```
 s.bin_lo = bin left edge [Angstrom]
 s.bin_hi = bin right edge [Angstrom]
```

get_rmf_data_grid

Purpose: Get the data grid from an RMF
Usage: `Struct_Type = get_rmf_data_grid (rmf_index)`
See Also: `list_rmf, get_rmf_rmf_grid`

.

This function retrieves the data grid from a specific RMF, returning a S-Lang structure containing a wavelength grid (angstrom units) in monotonic increasing order. The structure returned has the form

```
s.bin_lo = bin left edge [Angstrom]
s.bin_hi = bin right edge [Angstrom]
```

get_sys_err_frac

Purpose: Get the fractional systematic error
Usage: `sys_err_frac = get_sys_err_frac (hist_index)`
See Also: `set_sys_err_frac, load_data`

.

See `load\_data` for a definition of fractional systematic error, e.g. `SYS_ERR`.

set_sys_err_frac

Purpose: Set the fractional systematic error
Usage: `set_sys_err_frac (hist_index, sys_err_frac[])`
See Also: `get_sys_err_frac, load_data`

.

See `load\_data` for a definition of fractional systematic error, e.g. `SYS_ERR`.

group

Purpose: Group spectral bins using S/N ratio and/or channel
Usage: `group (datasets[] [; qualifiers])`
See Also: `group_bin, group_data, rebin_data, use_file_group, regroup_file, rebin_dataset, set_rebin_error_method, rebin, rebin_array`

Rebin data sets using a combination of minimum signal-to-noise (S/N) ratio and minimum number of channels in each grouped bin. If multiple datasets are to be grouped, their grids must match exactly.

The background data set, if it exists, is included in the S/N calculation. `group` presumes that the total noise goes as:

```
sqrt( (Total_Counts) + (Back_Counts)*(Back_Scale*Exposure_Ratio)^2 )
```

where the totals represent sums over all datasets (or those specified by the `sn_data` qualifier, if it is present).

The last channels are binned into a single group, regardless of whether or not they meet the binning criteria.

Different rebinning criteria may be applied simulataneously to different parts of the spectrum. Spectrum subintervals are specified using the `bounds` and `unit` qualifiers, and grouping criteria are specified as arrays with one entry for each subinterval.

```
Qualifier    Default     Meaning
---------    -------     -------
min_sn        0          Final bin will have S/N ratio >= min_sn
sn_data       all        List of datasets to be used for S/N calculation
                           (by default, all specified datasets are used)
min_chan      1          Final bin will contain at least min_chan bins
bounds        NULL       Endpoints of spectrum subintervals that
                            will be grouped differently
unit          Angstrom   Physical units of ‘bounds’ coordinates
```

For example,

```
group ([1,2,3]; sn_data=[1,2],
                bounds=[0.5,2], unit="kev",
                min_chan=[2,4], min_sn=5);
```

will group data sets 1,2, and 3, starting at 0.5 keV, to minimum S/N of 5 for the combination of data sets 1 and 2 only, and to a minimum of 2 channels between 0.5-2 keV, and 4 channels above 2 keV.

```
group ([1:4]; bounds=[1.5,6,12,18], unit="a",
              min_chan=[2,4,8,16], min_sn=0);
```

will group data sets 1-4 to 2 channels per bin between 1.5-6 Angstroms, to 4 channels per bin between 6-12 Angstroms, to 8 channels per bin between 12-18 Angstroms, and to 16 channels per bin above 18 Angstroms.

```
\end{isisfunction}

\begin{isisfunction}
{group\_bin}
{Group data to approximately match a specified grid}
{group\_bin (datasets[], lo, hi [; qualifiers])}
{group, group\_data, rebin\_data, use\_file\_group, regroup\_file, rebin\_dataset, set\_rebin\_error\_m

Regroup the specified datasets to match a particular histogram
grid as closely as possible without changing the underlying
spectral grids. If multiple datasets are to be grouped, their
grids must match exactly.

The physical units of the input grid may be specified using the
```

```
\verb|unit| qualifier. The current default physical units may
be obtained or changed using the \verb|unit_default| intrinsic.

To regrid a dataset to exactly match a particular grid, use
\verb|rebin_dataset|.
\end{isisfunction}

\begin{isisfunction}
{group\_data}
{Group spectral bins by an integer factor}
{group\_data (hist\_index\_array, factor)}
{group, group_bin, rebin\_data, use\_file\_group, regroup\_file, rebin\_dataset, set\_rebin\_err
\index{Rebinning!integer factor}

The count data of each histogram in {\tt hist\_index\_array} is
rebinned by summing the contents of the original input data bins
and the associated bin uncertainties ({\tt stat\_err}) are
recomputed assuming Poisson statistics.  Use
\verb|set_rebin_error_method| to change the way bin uncertainties
are recomputed. Note that this rebinning does not involve an event
list.

\begin{verbatim}
Example:

  group_data (1, 4);   % group data set 1 by a factor of 4
```

back_fun

Purpose: Specify instrumental background function for a data-set
Usage: `back_fun (idx, "function")`
See Also: `fit_fun`, `set_par`, `flux_corr`, `define_back`, `_define_back`, `get_back_model`, `get_back`

This function allows one to define the component of the background which is not folded through the ARF and RMF (e.g. the $B(h)$ term in equation (7.35)). The syntax of the function string is the same as that used by `fit_fun`.

To eliminate the instrumental background term for a data-set, use `NULL` or an empty string (`""`) as the second argument.

For example:

```
  % use a sum of power-law functions to model
  % the instrumental background for data-set 1:
  back_fun (1, "Powerlaw(1) + Powerlaw(2)");

  % Turn off the instrumental background term for
  % data-set 2
  back_fun (2, NULL);
```

It is important to note that the specified background function will be evaluated on the wavelength grid associated with the data (at full resolution). If a different wavelength grid is required, it may be necessary to implement the background function using `eval_fun2` to ensure that the correct grid is used.

Here is one way to implement the background function using `eval_fun2`:

```
private variable First_Order_Grid = get_rmf_data_grid (1);
define backfun_fit (_l,_h,_p)
{
   variable l = First_Order_Grid.bin_lo;
   variable h = First_Order_Grid.bin_hi;
   variable p = eval_fun2 ("poly", l, h, _p[[:2]]);
   variable g = eval_fun2 ("gauss", l, h, _p[[3:]]);
   variable t = get_data_exposure (Isis_Active_Dataset);

   return t * (p + g);
}
add_slang_function ("backfun",
                     ["a0", "a1", "a2",
                      "area", "center", "sigma"]);

back_fun (h, "backfun");
set_par ("backfun", [0.030, -7.26e-6, 1.73e-6,
                     -0.32, 48.7, 16.4]);
```

The most important aspect of this implementation is that it ignores the data grid passed in the variables (`_l, _h`), instead computing the background contribution explicitly using the first-order spectral grid contained in the structure, `First_Order_Grid`.

The distinction between the data grid and the background grid can be important in the analysis of dispersed spectra containing contributions from multiple dispersion orders and with a significant background contribution(e.g. Chandra LETG/HRC-S data).

interpol

Purpose: Interpolate a function y(x) onto a new grid
Usage: `new_y[] = interpol (new_x[], old_x[], old_y[])`
See Also: `rebin, interpol_points`

The input function `y(x)` specified by the S-Lang arrays (`old_x,old_y`) is linearly interpolated onto the grid (`new_x`) to determine the corresponding interpolated values `new_y`. Both new and old grids must be in monotonic increasing order.

```
Qualifiers:
name          description
----          -----------
extrapolate   Specify extrapolation method, if any.
                extrapolate = "none"|"linear"|"logx"|"logy"|"logxy"
                     default:  extrapolate="linear"
        "linear" means perform linear extrapolation on old_y vs. old_x
          "logx" means perform linear extrapolation on old_y vs. log(old_x)
          "logy" means perform linear extrapolation on log(old_y) vs. old_x
         "logxy" means perform linear extrapolation on log(old_y) vs. log(old_x)
```

```
null_value    Y value used for off-grid X values when extrapolate="none"
                     default:  null_value=NULL
```

interpol_points

Purpose: Interpolate a function y(x) onto a new grid
Usage: `new_y[] = interpol_points (new_x[], old_x[], old_y[])`
See Also: `rebin, interpol`

This function is identical to `interpol` except that the array of X values, `new_x`, need not be ordered.

is_flux_mode

Purpose: Determine the current data mode
Usage: `flag = is_flux_mode()`
See Also: `use_flux, use_counts`

Use of this function and the ISIS *data modes is not recommended.* Experience has shown that use of these data modes leads to confusion and unnecessary complications in analysis scripts. This function is still available primarily to provide compatibility with earlier versions of ISIS – it may be removed in a subsequent release.

Use this function to determine whether ISIS is currently dealing with flux data or count data. If the return value is one, commands such as `get_data`, `put_data` and `plot_data` all refer to *flux-corrected* data (e.g. the `FLUX` column in the Type II PHA file). If the return value is zero, these commands all refer to *count* data (e.g. the `COUNTS` column in the PHA file)

lambda_mth_order

Purpose: Plot the location of mth order given that the nth order appears at λ_n.
Usage: `lambda_mth_order (m, lambda_n, [n])`
See Also: `plot_data_counts`

If n is not specified, the default value is n=1. lambda_n is the wavelength in Angstroms (Å).

$$\lambda_m = \lambda_n \left(\frac{m}{n}\right) \tag{7.21}$$

Non-integer orders are allowed.

list_arf

Purpose: Display a list of currently loaded ARFs
Usage: `list_arf ([arg])`
See Also: `get_arf, put_arf, delete_arf`

.

The optional argument is used to redirect the output. If `arg` is omitted, the output goes to `stdout`. If `arg` is of type `Ref_Type`, it the output string is stored in the referenced variable.

If `arg` is a file name, the output is stored in that file. If `arg` is a file pointer (`File_Type`) the output is written to the corresponding file.

The currently loaded list of effective area functions (ARFs) is displayed. The indices of the ARFs in this list (`id` column) are used in other commands to refer to individual ARFs; e.g. `get_arf (2)` refers to ARF number 2. The ARF list looks like this:

```
isis> list_arf;

Current ARF List:
 id grating detector  part/m src   nbins  exp(ksec)  target
  1    HETG     ACIS  heg+1   1     8192    89.88     mysrc
  2    HETG     ACIS  heg+2   1     8192    89.88     src2
```

where the columns are defined as

```
      id = integer id number (arf_index)
 grating = e.g. HETG or LETG
detector = e.g. ACIS-S or HRC-S
       m = diffraction order (TG_M)
    part = e.g. HEG or MEG, (TG_PART)
     src = source index (TG_SRCID)
   nbins = total number of bins
     exp = Exposure time [ksec]
  target = target name
```

If the intrinsic variable `Isis_List_Filenames` is non-zero, the name of the ARF file (if any) will be displayed on the line following each list entry.

list_data

Purpose: display the currently loaded list of spectra
Usage: `list_data ([arg])`
See Also: `delete_data, load_data, get_data_info, get_dataset_metadata`

The optional argument is used to redirect the output. If `arg` is omitted, the output goes to `stdout`. If `arg` is of type `Ref_Type`, it the output string is stored in the referenced variable. If `arg` is a file name, the output is stored in that file. If `arg` is a file pointer (`File_Type`) the output is written to the corresponding file.

The currently loaded list of spectra is displayed. The indices of the spectra in this list (`id` column) are used in other commands to refer to individual spectra; e.g. `plot_data (2)` refers to spectrum number 2. The spectrum list looks like this:

```
id instrument  part/m src  use/nbins A  R     totcts exp(ksec) target
 1  HETG-ACIS  heg-3   1  6972/ 6972 -  - 4.8000e+01    0.00   src1
 2x HETG-ACIS  heg-2   1  6972/ 6972 1  1 2.2700e+02    0.00   src1
 3  HETG-ACIS  meg-1   1  6972/ 6972 -  - 6.2870e+03    0.00   src1
```

where the columns are defined as

```
       id = integer data set id number ['x' => excluded from fit]
  grating = e.g. HETG or LETG
 detector = e.g. ACIS-S or HRC-S
     part = e.g. HEG or MEG, (TG_PART)
        m = diffraction order (TG_M)
      src = source index (TG_SRCID)
use/nbins = number of noticed bins/ (total number of bins)
        A = index of assigned ARF ("-" if none assigned)
        R = index of assigned RMF ("-" if none assigned)
   totcts = total counts summed over all bins
      exp = Exposure time [ksec]
   target = target name
```

If the intrinsic variable `Isis_List_Filenames` is non-zero, the names of the spectrum file and background file (if any) will also be displayed on lines following each list entry.

If the function `list_data_hook` is defined in the Global namespace, it will called for each dataset. Among other things, this hook can be used to display user-defined metadata. For example, suppose you want to associate a string and a floating point number with each dataset and you want these values printed out whenever you call `list_data`. One way to implement that is to store your metadata in a structure along with a pointer to a function to do the printing. For example, consider this structure definition:

```
define printm (m)
{
   vmessage ("s=%s, x=%g", m.s, m.x);
}
variable metadata = struct {s, x, printm};
metadata.s = "Hello World!";
metadata.x = 3.1415;
metadata.printm = &printm;
```

Use the `set_dataset_metadata` function to associate this structure with dataset 1:

```
set_dataset_metadata (1, m);
```

Now, to have this metadata printed out whenever `list_data` is run, provide a `list_data` hook like this:

```
public define list_data_hook ()
{
   variable m = get_dataset_metadata (Isis_Active_Dataset);
   if (m != NULL)
    (@m.printm)(m);
}
```

list_rmf

Purpose: Display a list of currently loaded RMFs
Usage: `list_rmf ([arg])`
See Also: `assign_rmf`, `unassign_rmf`

.

The optional argument is used to redirect the output. If `arg` is omitted, the output goes to `stdout`. If `arg` is of type `Ref_Type`, it the output string is stored in the referenced variable. If `arg` is a file name, the output is stored in that file. If `arg` is a file pointer (`File_Type`) the output is written to the corresponding file.

The list of currently loaded Redistribution Matrix Functions (RMFs) is displayed.

```
Example:

isis> list_rmf;

Current RMF List:
 id grating detector  type   file
  1    HETG   ACIS-S  file:  hetg_rmf.fits
```

load_arf

Purpose: Load an effective area (ARF) file
Usage: `status = load_arf ("filename")`
See Also: `load_dataset, list_arf, delete_arf, assign_arf, unassign_arf`

This function loads either a FITS Type I or Type II ARF file; the updated list of currently loaded ARFs is automatically displayed. On return, `status` is equal to the integer index of the ARF just loaded (`status` > 0); a return value of `status = -1` is used to indicate failure. (For Type II ARF input, a return value of zero indicates success).

load_data

Purpose: read a spectrum from an ASCII file or FITS Type I or II PHA file
Usage: `status = load_data("pha_filename" [, rows] [; qualifiers])`
See Also: `load_dataset, define_counts, define_flux, define_back, get_data_*, put_data_*, plot_unit, set_dataset_metadata, get_dataset_metadata, get_sys_err_frac, Minimum_Stat_Err, set_min_stat_err, get_min_stat_err`

The format of the Type II PHA file is defined in the CXCDS Level 2 Data Products ICD. An ASCII-format file should contain the histogram data in 4 columns: `bin_lo`, `bin_hi`, `bin_value`, `bin_uncertainty`. By default, all spectra in a Type II pha file are loaded at once; to load a particular list of spectra, supply an array of row numbers as the (optional) second argument.

```
  Example:
          % to load spectra 9 and 10 from a standard
          % Chandra Type II pha file
          % (usually the MEG +1 and -1 order spectra):

        isis> id = load_data ("hetg_pha2.fits", [9,10]);
```

```
Qualifiers:
-----------
```

```
with_bkg_updown  if present, load BACKGROUND_UP/DOWN columns
min_stat_err     require stat_err >= min_stat_err
```

When the with_bkg_updown qualifier is present, the background is defined using the BACKGROUND_UP and BACKGROUND_DOWN columns using this definition:

```
back = (BACKGROUND_UP + BACKGROUND_DOWN)
       /(BACKSCUP + BACKSCDN)
```

This background is subsequently scaled in the usual way using the relevant BACKSCAL and EXPOSURE values.

Although a Type I PHA file can be loaded without first specifying the RMF, an RMF is required to use the data. The RMF may be specified either by using the RESPFILE keyword in the FITS header or by loading the RMF file separately (see load_rmf and assign_rmf). The ARF may be specified either by using the ANCRFILE keyword in the FITS header or by loading the ARF file separately (see load_arf and assign_arf). To ignore the ANCRFILE and RESPFILE values when loading the PHA file, set Ignore_PHA_Response_Keywords=1.

Similarly, the BACKFILE keyword in the FITS header can be used to specify the name of the file containing the background spectrum. To ignore this keyword, set Ignore_PHA_Backfile_Keyword=1.

By default, isis ignores the GROUPING column of the input PHA file. To automatically apply the grouping on input, set the intrinsic variable Isis_Use_PHA_Grouping to a non-zero value.

Input data values must be bin-integral quantities rather than bin-densities; uncertainty values should be positive. By default, any input STAT_ERR uncertainty values smaller than min_stat_err are reset according to

```
stat_err = max[ sqrt(counts), min_stat_err ]
```

where min_stat_err is the smallest acceptable (positive) uncertainty value. When the min_stat_err qualifier is not present, the value of the intrinsic variable Minimum_Stat_Err is used if positive, otherwise, the default minimum uncertainty is 1. To shut off warnings about invalid uncertainties being replaced, set Warn_Invalid_Uncertainties=0.

If the SYS_ERR column or header keyword is present, a systematic error is always added in quadrature so that the uncertainty on each data bin becomes

```
sigma = sqrt ( sigma_stat^2 + (D * sys_err)^2 ),
```

where sigma_stat is the statistical uncertainty, and D is the corresponding data value. This default behavior can be altered by using the set_sys_err_frac function to set the systematic errors to 0.

When the data are rebinned, isis uses the mean systematic error in each new bin. The mean systematic error in each new bin is computed using

```
syserr =  (\sum_k syserr_k  dy_k ) / (y_hi - y_lo)
```

where the new bin spans the wavelength interval [y_lo,y_hi), and includes contributions from full-resolution bins with systematic error syserr_k and width dy_k.

If the input data are in flux units (photons/cm**2/sec/bin) and non-positive uncertainties are encountered, the uncertainty is set equal to 1.0. In each case a message is printed to warn the user. To reset the uncertainties to another value, see `get_data_*` and `put_data_*`.

Input bin coordinates may be given as Angstrom (Å), nm, eV, keV or Hz; if the bin coordinates are unspecified, the default is Angstrom units. Although all internal calculations are done in Angstrom units, plots may be generated in any of the supported physical units. See `plot_unit`.

In the ASCII format, lines with a # symbol in column 1 are ignored and may be used for comments.

The ASCII file may also contain keywords analogous to the header keywords in FITS files. All keywords must be grouped together at the top of the file (possibly with interspersed comment lines). Each keyword line must have a semicolon (;) in column 1 and may contain only one keyword name/value pair; a maximum of 1024 characters will be scanned on each such line. Only the first 8 characters of each keyword name are significant; the keyword name and keyword value must be separated by at least one space or tab character. Keyword values may be of type int, float, double or string; string keyword values may contain any printable characters including embedded whitespace characters (except newline). The supported keyword names are

```
___Keyname____Type____Definition_______________________

    object   string   source name
instrument   string   e.g. ACIS-S or HRC-S
   grating   string   e.g. HETG or LETG
  exposure   double   exposure [sec]
      tg_m   int      diffraction order
   tg_part   int      e.g. HEG or MEG
  tg_srcid   int      source id number
     xunit   string   physical units of bin coordinates
   bintype   string   bin-value units; [counts|flux]
```

This function returns `status` equal to the integer index of the data set(s) loaded; for Type I pha files, this is a single positive integer (> 0) and for Type II pha files, this is an array of positive integers. Otherwise, it returns `status = -1` to indicate failure.

```
Example ASCII format file:

  # this is a comment line
  #
  ;     Object   test src 1
  #
  ; Instrument   acis-s
  ;    Grating   heg
  ;   Exposure   1.e6
  ;      xunit   angstrom
  ;    bintype   counts
  #
  #   bin_lo           bin_hi    counts stat_err
    1.5955326e+01   1.5960888e+01   4.0    2.0
    1.5960888e+01   1.5966450e+01   4.0    2.0
  #   another comment line
    1.5966450e+01   1.5972012e+01   4.0    2.0
    1.5972012e+01   1.5977573e+01   4.0    2.0
```

If the function `load_data_hook` is defined in the Global namespace, it will be called after the data is successfully loaded as

```
load_data_hook (file, id);
```

where `file` is the name of the file just loaded and where `id` is an array of `Integer_Type` which gives the indices of the datasets just loaded. Among other things, this function may be used to automatically associate user-defined metadata with each dataset.

For example, suppose you want each dataset to carry along the `RA_NOM`, `DEC_NOM`, values from the FITS header. Consider the following function

```
public define load_data_hook (file, id)
{
    variable m = fits_read_key_struct (file, "RA_NOM", "DEC_NOM");
    set_dataset_metadata (id, m);
}
```

load_dataset

Purpose: Load a spectrum, ARF and RMF
Usage: `load_dataset ("data-file", "rmf-file", "arf-file")`
See Also: `load_data`, `define_counts`, `define_flux`, `load_arf`, `load_rmf`, `assign_arf`, `assign_rmf`

This function is essentially equivalent to the sequence:

```
d = load_data ("data.fits");
r = load_rmf ("rmf.fits");
a = load_arf ("arf.fits");
assign_rmf (r,d);
assign_arf (a,d);
```

If either the RMF or ARF names are missing or NULL, the corresponding response is not assigned and defaults to an ideal response.

load_rmf

Purpose: Load an RMF
Usage: `status = load_rmf ("filename[:init_name[;options]]")`
See Also: `load_slang_rmf`, `load_dataset`, `list_rmf`, `assign_rmf`, `unassign_rmf`

An RMF is usually a FITS file which conforms to the OGIP standard RMF format.

ISIS also supports RMFs which are defined in software either in S-Lang (see `load_slang_rmf`) or in a compiled language such as a C. In the latter case, the RMF is specified by giving the name of a shared library (.so file) which provides the software implementation conforming to the interface defined in the ISIS source code in `src/isis.h`. See `test/rmf_user.c` for an example implementation.

The name of the initialization function for the user-defined RMF module should be included in the string specifying the name of the shared library; the two names fields should be separated by a colon (:). Additional RMF-specific options may be specified by adding semicolon-delimited arguments; these options are passed to the RMF module initialization function when the RMF is initialized for a specific histogram. User-defined RMFs may parse this string to obtain additional useful parameters, e.g. perhaps an auxiliary data file name or specific values for user-defined RMF parameters.

```
Examples:
        % load an OGIP FITS-format RMF file
        () = load_rmf ("heg_rmf.fits");

        % load a user-defined RMF module from librmf.so,
        % initialized by calling my_rmf_init_function().
        () = load_rmf ("librmf.so:my_rmf_init_function");

        % here, additional option strings are used to supply
        % two parameters to the RMF function when it is initialized.
        () = load_rmf ("libotherrmf.so:init_function ;sigma=4.32;a=4");
```

By default, isis will generate an error if asked to load a FITS RMF file that does not adhere closely to the OGIP standard format. Setting the global variable `Rmf_OGIP_Compliance=0` will reduce the required level of standards compliance for all input RMF files. Alternatively, one can use the `strict` qualifier to change the setting for a specific file. For example:

```
rmf_index = load_rmf ("non_ogip_compliant.rmf;strict=0");
```

If isis is still unable to read the file, the best approach may be to modify the RMF file to adhere more closely to the standard format. Usually this is just a matter of adding a few keywords and making sure that important keywords have correct values. In particular, isis looks for

```
EXTNAME = EBOUNDS
EXTNAME = MATRIX | SPECRESP MATRIX
HDUCLAS2 = RSP_MATRIX
HDUCLAS3 = REDIST | DETECTOR | FULL
```

The `HDUCLASn` keywords are required for full OGIP compliance, but will be ignored if `Rmf_OGIP_Compliance=0`, or if the qualifier `strict=0` is present.

load_slang_rmf

Purpose: Define an RMF using a S-Lang function
Usage: `id = load_slang_rmf (&func, h_bin_lo, h_bin_hi, arf_bin_lo, arf_bin_hi)`
`id = load_slang_rmf (&func, hist_index, arf_index)`
See Also: `load_rmf, load_dataset, list_rmf, assign_rmf, unassign_rmf`

A S-Lang function can be used to define a redistribution function (RMF) with the effective area (ARF) and data (EBOUNDS) grids specified either as S-Lang arrays:

```
id = load_slang_rmf (&func, h_bin_lo, h_bin_hi, arf_bin_lo, arf_bin_hi)
```

or by referring to the indices of a dataset and an ARF:

```
  id = load_slang_rmf (&func, hist_index, arf_index)

Qualifiers:
  parms=value        (optional parameter passed to func)
  threshold=double   (default: 1e-6)
  grid="en|wv"       (default: "en")
```

The function that computes the RMF profile must be of the form:

```
    define func (h_bin_lo, h_bin_hi, en_or_wv [,parms])
```

It must compute the RMF profile integrated over the `h_bins` at the energy or wavelength value `en_or_wv` (depending upon the grid qualifier).

```
EXAMPLE:

  define rmf_profile (bin_lo, bin_hi, x, parms)
  {
     variable resolution = parms;
     variable rmf = Double_Type[length(bin_lo)];

     if (x < bin_lo[0] || bin_hi[-1] <= x)
       return rmf;

     variable i = where (x * (1.0 - resolution/2.0) <= bin_lo
                        and bin_hi < x * (1.0 + resolution/2.0));

     rmf[i] = 1.0 / length(i);
     return rmf/sum(rmf);
  }

  rmf_id = load_slang_rmf (&rmf_profile,
                           linear_grid (0.1, 10, 1024),
                           linear_grid (0.1, 10, 2048); parms=0.02);
  assign_rmf (rmf_id, dataset_id);
```

match_dataset_grids

Purpose: Put a list of datasets onto the same grid
Usage: `match_dataset_grids (list[])`
See Also: `combination_members`, `combine_datasets`, `uncombine_datasets`, `get_combined`, `get_combined2`, `rebin_combined`, `set_pre_combine_hook`

This function uses `rebin_dataset` to put all the listed datasets onto the original (ungrouped) grid from the first dataset in the list. All spectrum bins are noticed.

Note that each dataset in the provided list should have its own set of responses; in other words, no single response function (ARF or RMF) should be assigned to more than one dataset. This restriction is necessary to avoid unintended side-effects when the responses are interpolated onto the new grid.

The main purpose of this function is to support combining datasets to improve statistics – see combine_datasets.

plot_data

Purpose: plot spectral data (counts) using the current plot format
Usage: `[o]plot_data (hist_index [,style])`
See Also: `plot_data_counts`

This is an alias for `plot_data_counts`.

plot_data_counts

Purpose: plot spectral data (counts) using the current plot format
Usage: `[o]plot_data_counts (hist_index [,style])`
See Also: `rplot_counts, set_data_color, [o]plot_model, title`

This function plots the counts histogram of the specified data set. Other similar functions are available to plot the flux-corrected data set (See `[o]plot_data_flux`) and to plot the corresponding model values (See `[o]plot_model_counts` and `[o]plot_model_flux`)

The spectrum to plot is specified using its integer index (`hist_index`) in the current internal list (see `list_data`). The destination plot window if different from the current default, are specified using `window`.

The plot data coordinates are in Ångstrom units by default, but may be changed using the `plot_unit` function. Bin values may be plotted in either bin-integral units (the default) or bin-density units (see `plot_bin_integral` and `plot_bin_density`).

Axis ranges default to the full range of the data, but may be specified using `[xy]range`. Both linear or logarithmic axis scales are available; see `[xy]log`. Errorbars are also available (see `errorbars`).

Plot axes are labeled automatically by default, to turn off this behavior, set `Label_By_Default = 0`; setting this variable to a non-zero value will restore the default behavior.

Repeated invocations of `oplot_data_counts` will automatically switch line colors or styles to help distinguish the over-plotted curves. Depending on the current setting [see `style`] either colors or line styles (e.g. solid vs. dashed) are used to distinguish over-plots. To override the automatic color/style changes, the `style` index can be specified explicitly for each plot (see also `set_data_color`). Alternatively, the automatic color-changes can be disabled using `plot_auto_color`.

See §7.6 for a general discussion of ISIS plotting.

plot_convolved_model_flux

Purpose: plot convolved spectral model flux for a specific data spectrum
Usage: `[o]plot_convolved_model_flux (hist_index [,style])`
See Also: `flux_corr_model_counts, [o]plot_data_flux, [o]plot_model_flux`

This function plots the model for the specified data set broadened using the assigned RMF. The convolved model flux is computed using the expression

$$F(h) = \frac{\int dER(h,E)A(E)s(E)}{\int dER(h,E)A(E)}. \tag{7.22}$$

See `get_convolved_model_flux` and `plot_model_counts` for details.

plot_data_flux

Purpose: plot spectral data flux using the current plot format
Usage: `[o]plot_data_flux (hist_index [,style])`
See Also: `set_data_color, [o]plot_model_flux, title`

This function plots the flux histogram of the specified data set. The flux histogram may be computed from the counts spectrum by `flux_corr`, loaded from a data file by `load_data` or defined via S-Lang arrays using `define_flux`. For details, see `plot_data_counts`.

plot_model_counts

Purpose: plot spectral model counts for a specific data spectrum
Usage: `[o]plot_model_counts (hist_index [,style])`
See Also: `[o]plot_data_counts, [o]plot_model_flux, [o]plot_convolved_model_flux`

This function plots the model for the counts histogram of the specified data set. The model counts, $C(h)$, are computed using the expression

$$C(h) = B(h) + t \int \mathcal{F}\Big(R(h,E), A(E), S(E)\Big) dE \tag{7.23}$$

where $B(h)$ is the background spectrum, $R(h,E)$ is the RMF, $A(E)$ is the ARF and $S(E)$ is the model for the incident photon spectrum.

See `plot_data_counts` for details.

plot_model_flux

Purpose: plot spectral model flux for a specific data spectrum
Usage: `[o]plot_model_flux (hist_index [,style])`
See Also: `[o]plot_data_flux, [o]plot_model_counts, [o]plot_convolved_model_flux`

This function plots the model for the specified data set. Normally, the model represents the incident flux, $S(E)$, integrated over the width of each spectral bin, and has units of photons/sec/cm^2.

See `plot_model_counts` for details.

put_arf

Purpose: Change ARF grid and/or values
Usage: `put_arf (arf_index, arf_struct | bin_lo, bin_hi, arf, arf_err)`
See Also: `get_arf, list_arf`

If invoked with two arguments, the second argument should be a struct with fields `bin_lo, bin_hi, value, err`. If invoked with five arguments, the last four arguments should provide the equivalent values:

```
arf_index = integer index of ARF in internal list
 a.bin_lo = bin left edge [Angstrom]
 a.bin_hi = bin right edge [Angstrom]
  a.value = bin value [cm^2 counts/photon]
    a.err = bin uncertainty [cm^2 counts/photon]
```

This command replaces internal ARF data with values from several S-Lang array-variables. Only the internal values are changed; no disk files are affected.

The only restrictions are that 1) the dimensionality of the arrays cannot be changed, 2) the input histogram grid must be consistent (Angstrom units in monotonic increasing order with `bin_lo` < `bin_hi`) and 3) the uncertainty values must be positive. Any non-positive input values of `uncertainty` are reset to 1 and a message is printed to warn the user.

```
Example:
      a = get_arf(1);
      put_arf(1, a.bin_lo, a.bin_hi, a.value, 2 * a.err);
```

put_data_counts

Purpose: Change the counts histogram for a data set
Usage: `put_data_counts (hist_index, struct | bin_lo, bin_hi, value, uncertainty)`
See Also: `get_data_counts, list_data, define_counts, define_flux`

If invoked with two arguments, the second argument should be a struct with fields `bin_lo, bin_hi, value, err`. If invoked with five arguments, the last four arguments should provide the equivalent values:

```
hist_index = integer index of spectrum in internal list
  d.bin_lo = bin left edge [Angstrom]
  d.bin_hi = bin right edge [Angstrom]
   d.value = bin value [counts]
     d.err = bin uncertainty [counts]
```

This command replaces internal histogram data with values from several S-Lang array-variables. Only the internal values are changed; no disk files are affected. A similar function is provided to modify the flux-corrected data values (`put_data_flux`).

The only restrictions are that 1) the dimensionality of the arrays cannot be changed, 2) the input histogram grid must be consistent (Angstrom units in monotonic increasing order with `bin_lo` < `bin_hi`) and 3) the uncertainty values must be positive. Any non-positive input values of `uncertainty` are reset (e.g. according to counting statistics for `counts` $= N$, `uncertainty` $= \sqrt{N}$) and a message is printed to warn the user.

```
Example:
      c = get_data_counts (1);
      put_data_counts (1, c.bin_lo, c.bin_hi, c.value, sqrt(c.value));
```

put_data_flux

Purpose: Change the flux-corrected histogram for a data set
Usage: `put_data_flux (hist_index, struct | bin_lo, bin_hi, value, uncertainty)`
See Also: `get_data_flux, list_data, define_counts, define_flux`

This function is analgous to `put_data_counts`, except that it overwrites the appropriate internal array containing flux-corrected data. See `put_data_counts` for details.

put_model_counts

Purpose: Replace the counts model for a data set
Usage: `put_model_counts (hist_index, counts[])`
See Also: `put_model_flux, put_convolved_model_flux, eval_counts, eval_flux`

Use this function to replace the model counts array for a particular data set. Note that the dimension of the input array must match the current, possibly rebinned, data set including both noticed and ignored bins.

put_model_flux

Purpose: Replace the flux model for a data set
Usage: `put_model_flux (hist_index, counts[])`
See Also: `put_model_counts, put_convolved_model_flux, eval_counts, eval_flux`

Use this function to replace the model flux array, `S(E)`, for a particular data set. Note that the dimension of the input array must match the ARF (or the unbinned data, if no ARF has been assigned).

put_convolved_model_flux

Purpose: Replace the convolved flux model for a data set
Usage: `put_convolved_model_flux (hist_index, counts[])`
See Also: `put_model_counts, put_model_flux, eval_counts, eval_flux`

Use this function to replace the convolved model flux array for a particular data set. Note that the dimension of the input array must match the current, possibly rebinned, data set.

rebin

Purpose: Rebin a histogram
Usage: `newval = rebin (new_lo, new_hi, lo, hi, value)`
See Also: `rebin_data, group_data, rebin_dataset, rebin_array, interpol`

Using linear interpolation where necessary, the input histogram specified by the S-Lang arrays (`lo, hi, value`) is mapped onto the specified grid (`new_lo`, `new_hi`) to produce the new bin-value array `newval`. Both input and output grids must be in monotonic order. Note that the input histogram is assumed to be a bin-integrated quantity such as `counts/bin` and that no unit conversions are performed; to rebin a bin-density such as `counts/Å`, the user must explicitly handle the unit conversion, e.g.

```
x = density * (hi - lo);                      % convert to bin-integral
new_x = rebin (new_lo, new_hi, lo, hi, x);
new_density = new_x / (new_hi - new_lo);      % convert to bin-density
```

This function can be used to simplify adding spectra together (e.g. summing plus and minus order diffracted spectra, and summing the associated ARFs):

```
Example:
   m = get_data_counts (megm1);   % get MEG m=-1 order
   h = get_data_counts (hegm1);   % get HEG m=-1 order

   mcts = m.value;
   hcts = h.value;

   % map HEG onto MEG grid:
   new_hcts = rebin (m.bin_lo, m.bin_hi, h.bin_lo, h.bin_hi, hcts);

   total = mcts + new_hcts;       % total counts in -1st order MEG+HEG
                                  % using MEG -1 wavelength grid.
```

rebin_array

Purpose: Rebin a S-Lang array to match a rebinned spectrum
Usage: `result[] = rebin_array (array[], rebin_flags[])`
See Also: `rebin_data, group_data, rebin_dataset`

It is sometimes useful to rebin a S-Lang array so that it matches a rebinned dataset. For example, suppose we have a dataset with `N` bins and a matching model array with the same number of elements. If we rebin the data, the model array no longer matches:

```
% rebin dataset 1 to have at least 30 counts per bin
rebin_data (1, 30);
```

To generate a new, matching model array, do the following:

```
% retrieve the flag array used to rebin the data ..
s = get_data_info (1);
% .. and rebin the model the same way
m = rebin_array (model, s.rebin);
```

rebin_combined

Purpose: Rebin a combination of binned spectra
Usage: `rebin_combined (gid[], min_counts_per_bin | index_array)`
See Also: `combine_datasets, group_data, rebin_dataset, set_rebin_error_method, rebin, rebin_array`

This function is the same as `rebin_data` except that it operates on dataset combinations. For example, one can use this function to rebin a combination of datasets so that the sum has a specified minimum number of counts per bin. See `rebin_data` and `combine_datasets` for details.

rebin_data

Purpose: Rebin a binned spectrum
Usage: `rebin_data (hist_index_array, min_counts_per_bin | index_array)`
See Also: `group_data, rebin_dataset, set_rebin_error_method, rebin, rebin_array`

The count data of each histogram in `hist_index_array` is rebinned by summing the contents of the original input data bins and the associated bin uncertainties (`stat_err`) are recomputed assuming Poisson statistics. Use `set_rebin_error_method()` to change the way bin uncertainties are recomputed. Note that this rebinning does not involve an event list.

The second argument to `rebin_data` may be either a positive scalar value or an integer array of the same size as the original input histogram.

If the second argument is a scalar, it is interpreted as the minimum desired number of counts per bin; e.g. `rebin_data(1,25)` will rebin spectrum 1 so that each bin contains at least 25 counts.

If the second argument is an integer array the same length as the original input histogram, it defines the scheme for summing over the data bins. The integer values of this array should be either -1, 0 or 1; neighboring bins with the same sign (-1 or +1) are summed together, while bins with a zero are ignored.

For example, if the original data has 10 bins, the index array should have 10 bins. These commands will:

```
isis> i = [1, 1, -1, 1, 0, 1, -1, -1, -1, -1];
isis> rebin_data (meg, i);
```

will generate a 4 bin version of spectrum `meg` with this grouping:

```
result_bin[0] = original_bin[0] + original_bin[1]
result_bin[1] = original_bin[2]
result_bin[2] = original_bin[3] + original_bin[5]
result_bin[3] = original_bin[6] + original_bin[7]
                + original_bin[8] + original_bin[9]
```

Note that zero values mean that the associated bin should be ignored (`original_bin[4]` in this example).

The original input histogram may be restored using

```
isis> rebin_data (idx, 0);
```

This also restores the ignore/notice values in effect before the data was first rebinned; this feature provides a mechanism to excise bad bins in the input spectrum.

The flux-columns must be recomputed after rebinning the count data. Similarly, any associated fit-models must be recomputed.

rebin_dataset

Purpose: Rebin an RMF and its assigned spectrum
Usage: `rebin_dataset (dataset, bin_lo, bin_hi)`
See Also: `rebin_rmf, rebin_data, group_data, set_rebin_error_method, rebin`

This function rebins a counts spectrum and its assigned instrument response matrix (RMF) so that the RMF maps onto the new instrument grid `bin_lo`, `bin_hi`. The RMF normalization is preserved. See `rebin_data` and `rebin_rmf` for details.

rebin_rmf

Purpose: Rebin an RMF
Usage: `rebin_rmf (dataset, bin_lo, bin_hi)`
See Also: `rebin_dataset, rebin_data`

This function rebins an instrument response matrix (RMF) so that it maps onto a new instrument grid (`bin_lo`, `bin_hi`). The RMF normalization is preserved.

RMF rebinning makes it possible to apply an RMF to a dataset with a grid different from that for which the RMF was originally constructed.

As one application, RMF rebinning provides a consistent way to apply the instrument response to CCD spectra in PI space. The instrument response is normally measured in PHA space but, to account for gain variations over large areas of the detector, CCD spectra from extended sources are often analyzed in PI space. When constructing PI-RMFs, these gain variations are usually accounted for by simply shifting the locations of Gaussian peaks in the PHA-RMF. However, this process does not account for the distortion in the Gaussian profile shape introduced by the nonlinear transformation from PHA to PI space. In contrast, constructing PI-RMFs by rebinning PHA-RMFs automatically includes any such distortions.

```
Example:

   % This function rebins a PHA-RMF onto a PI-RMF grid
   define make_pi_rmf (k)
   {
      variable lo, hi;

      % Standard PI energy grid [keV]
      hi = 0.0146 * [1:1024];
      lo = hi - 0.0146;

      % tweak low end to E>0
      lo[0] += 1.e-3 * hi[0];

      return rebin_rmf (k, _A(lo, hi));
   }

   pi_data = load_data ("ccd_pi.fits");
   rmf = load_rmf ("pha_rmf.fits");
```

```
make_pi_rmf (rmf);
assign_rmf (rmf, pi_data);
```

regroup_file

Purpose: Apply grouping to PHA file
Usage: `regroup_file (grp[], file) OR regroup_file (id [,file])`
See Also: `group_data, use_file_group, i2x_group, x2i_group`

Use this function to regroup a `PHA` file using an ISIS grouping array. The grouping array, `grp` should be a standard, wavelength-ordered, isis grouping array as defined in the documentation for `rebin_data`. The length of the grouping array should match the length of the spectrum stored in the PHA file. This function will convert the grouping array to an OGIP-standard, energy-ordered grouping array and will write that array to the `GROUPING` column in the specified `PHA` file. If the first argument is an integer dataset index, the grouping array associated with that dataset will be used (e.g. `get_data_info(id).rebin`). If the file name is omitted, the file name associated with the dataset will be used (e.g. `get_data_info(id).file`).

For example:

```
% regroup a PHA file to match a given dataset
 grp = get_data_info (3).rebin;
 regroup_file (grp, "pha.fits");

% update the input PHA file to match the current grouping:
 id = load_data ("pha_x.fits");
 group_data (id, 4);
 regroup_file (id);
```

region_counts

Purpose: Compute count statistics for a given wavelength region
Usage: `Struct_Type = region_counts (hist_index, xmin, xmax [,ymin, ymax])`
See Also: `region_flux`

A data structure is returned for the histogram indicated by `hist_index`. This structure contains a number of statistics for the interval `[xmin, xmax)`. Individual fields are accessible using the S-Lang structure syntax; e.g. `s.sum` or `s.centroid_err` (see below).

If `ymin` and `ymax` are not specified, no continuum subtraction is performed (it is assumed that the continuum level is zero). If `ymin` and `ymax` values are specified, they are used to define a linear continuum $c(\lambda) = a * \lambda + b$ passing through the two points `(xmin, ymin)` and `(xmax,ymax)`. The continuum value in bin `k` between $[\lambda_{\rm lo}^k, \lambda_{\rm hi}^k]$ is then computed using

$$C_k = \left(\lambda_{\rm hi}^k - \lambda_{\rm lo}^k\right) \left[a \frac{\left(\lambda_{\rm hi}^k + \lambda_{\rm lo}^k\right)}{2} + b\right] \qquad (7.24)$$

Note that `ymin` and `ymax` represent the continuum *density* and have units of e.g. counts per Angstrom or flux per Angstrom.

With this continuum level, the structure fields are:

```
              min, max = wavelength limits defining this region [Angstrom]
                 nbins = number of noticed bins in this region
          sum, sum_err = sum of noticed bins and RMS uncertainty
          net, net_err = continuum subtracted sum of noticed bins
                         and RMS uncertainty
 centroid, centroid_err = centroid of continuum subtracted emission
                         in noticed bins and RMS uncertainty [Angstrom]
   eqwidth, eqwidth_err = equivalent width (positive/negative for
                         emission/absorption) of noticed bins and
                         RMS uncertainty [Angstrom]
         contin, slope = continuum density and slope at the
                         centroid position
```

region_flux

Purpose: Compute flux statistics for a given wavelength region
Usage: `Struct_Type = region_flux (hist_index, xmin, xmax [,ymin, ymax])`
See Also: `region_counts`

See `region_counts`.

rplot_counts

Purpose: Plot counts data and model with residuals
Usage: `rplot_counts (hist_index)`
See Also: `rplot_flux, plot_data_counts, title`

This function generates a two-paned plot, the upper pane showing the counts data with model overlaid and the lower pane showing the residuals. The type of residuals plotted is determined by the value of the intrinsic variable `Isis_Residual_Plot_Type`; supported values are:

```
___Value____Definition_________
   STAT     value provided by current fit-statistic
   DIFF     (data-model)
   RATIO    (data/model)
```

The default is `Isis_Residual_Plot_Type=STAT`. When the fit-statistic is `chisqr`, the plotted residual is $\Delta\chi$.

rplot_flux

Purpose: Plot flux-corrected data and model with residuals
Usage: `rplot_flux (hist_index)`
See Also: `rplot_counts, plot_data_flux, title`

This function generates a two-paned plot, the upper pane showing the flux-corrected data with model overlaid and the lower pane showing the residuals. See `rplot_counts` for details.

_[o]rplot_counts

Purpose: Plot residuals for counts data
Usage: `_[o]rplot_counts (hist_index[, style])`
See Also: `rplot_counts, _[o]rplot_flux`

This function plots residuals of the type specified by the intrinsic variable `Isis_Residual_Plot_Type` See `rplot_counts` for details.

_[o]rplot_flux

Purpose: Plot residuals for flux-corrected data
Usage: `_[o]rplot_flux (hist_index[, style])`
See Also: `rplot_flux, _[o]rplot_counts`

This function plots residuals of the type specified by the intrinsic variable `Isis_Residual_Plot_Type` See `rplot_counts` for details.

set_arf_exposure

Purpose: Set the exposure time for a given ARF
Usage: `set_arf_exposure (arf_index, exposure_sec)`
See Also: `get_data_exposure`

Use this to set exposure time in seconds for a particular effective area function (ARF).

set_arf_info

Purpose: Set ARF information
Usage: `set_arf_info (id, Struct_Type)`
See Also: `set_arf_exposure, set_arf_info, get_arf_info`

This function provides access to auxiliary ARF parameters through a structure:

```
isis> s=get_arf_info(1);
isis> print(s);
    order = -1
    part = 2
    srcid = 0
    exposure = 28123.4
    fracexpo = 1;
isis> s.exposure=3.e4;       % change the exposure time
isis> set_arf_info(1,s);     % update the internal value
isis> list_arf;

Current ARF List:
 id grating detector part/m  src   nbins  exp(ksec)  target
  1                   meg-1   0     8192    30.00
```

set_data_color

Purpose: Specify a plot color for one or more data sets
Usage: `set_data_color (hist_index_list, color_index)`
See Also: `unset_data_color, color, plot_data_counts, plot_auto_color`

If a plot color is specified for a data set, that color will always be used when plotting that data set. Unless this command is used to specify a particular color, the data set will be plotted using the default color in the corresponding plot window pane at the time the plot is created.

```
 Example:
  red = 2;
  blue = 4;

  set_data_color (2, blue);        % always plot spectrum 2 in blue.

  set_data_color ([1,4,5], red);   % always plot spectra 1,4 and 5 in red
```

set_data_backscale

Purpose: Set the background scaling for a given spectrum
Usage: `set_data_backscale (hist_index, area)`
See Also: `set_arf_exposure`

Set the BACKSCALE value associated with for histogram `hist_index`. Normally, this is the area of the spectral extraction region; the same area units must be used for both the source and background spectra.

set_data_exposure

Purpose: Set the exposure time for a given spectrum
Usage: `set_data_exposure (hist_index, exposure_sec)`
See Also: `set_arf_exposure`

Set the exposure time in seconds for histogram `hist_index`. Note that the ARF exposure time is the important quantity for fitting data.

set_data_info

Purpose: Change spectrum parameters using a S-Lang structure
Usage: `set_data_info (index_list, Struct_Type)`
See Also: `get_data_info, load_data, list_data`

This function changes selected data parameters using a structure whose fields are integers which define the `spec_num, order, part, srcid` and `exclude` keywords along with the `target` name string.

```
Example:
```

```
isis> load_data ("acisf01318N003_pha2.fits.gz");
Reading: ............
Integer_Type[12]
isis> s = get_data_info(9);
isis> print(s);
    spec_num = 9
    order = -1
    part = 2
    srcid = 1
    exclude = 0
    combo_id = 0;
    combo_weight = 1;
    target = CAPELLA
    tstart = 7.5667e+07
    frame_time = 3.2
    notice = Integer_Type[8192]
    notice_list = Integer_Type[8192]
    rebin = Integer_Type[8192]
    arfs = Integer_Type[1]
    rmfs = Integer_Type[1]
isis>
isis> s.order = 2;          % change #9 to 2nd order
isis> set_data_info (9, s);
```

set_dataset_metadata

Purpose: Associate user-defined meta data with a particular dataset
Usage: `set_dataset_metadata (hist_index, meta)`
See Also: `get_dataset_metadata`

Use this function to associate arbitrary (user-defined) metadata with a particular dataset. For example:

```
set_dataset_metadata (1, "SKY_X=4013.42; SKY_Y=3987.4");
```

The metadata may be retrieved using the `get_dataset_metadata` function. The ability to assign arbitrary metadata to a dataset and later retrieve it can be useful to support analysis techniques unforseen during isis development.

set_eval_grid_method

Purpose: Specify the model evaluation grid method for a dataset
Usage: `set_eval_grid_method (method, datasets[][, hook [,cache]])`
See Also: `combine_datasets`

Supported grid methods are `MERGED_GRID`, `SEPARATE_GRID` and `USER_GRID`

`SEPARATE_GRID` (the default) means that, when the model is evaluated for each dataset, it is evaluated on the wavelength grid defined by the ARF. If no ARF is specified, the data grid is used.

`MERGED_GRID` means that, for the specified list of datasets, the model is evaluated once at the highest resolution needed and then the resulting model histogram is rebinned onto the model grid associated with each dataset (usually the associated ARF grid). This method is automatically selected when datasets are combined using `combine_datasets`. When computation of the model is more expensive than rebinning and when several datasets with extensive wavelength overlap are being fitted simultaneously, this feature should save a considerable amount of CPU time. Because model computation usually dominates the CPU usage, merging grids is often a valuable optimization.

For example:

```
set_eval_grid_method (MERGED_GRID, [1:10]);
```

means that a single wavelength grid will be derived by merging the wavelength grids assigned to datasets 1-10. When fitting these datasets, the model will be evaluated once on the merged grid and then rebinned onto the grid of each individual dataset. This means that on each iteration, the model will be evaluated only once and not 10 times.

To revert to the default mode, use:

```
set_eval_grid_method (SEPARATE_GRID, [1;10]);
```

`USER_GRID` means that, when the model is evaluated for each dataset, it is evaluated on the wavelength grid supplied by a user-provided S-Lang function. This function must be of the form

```
Struct_Type = grid_hook (hist_index, Struct_Type);
```

where the `Struct_Type` has fields `bin_lo` and `bin_hi`. The grid returned by this function must define a wavelength grid in Angstrom units, in ascending order. The optional fourth argument is a boolean value which indicates whether or not the model value computed on the specified grid should be cached and subsequently rebinned onto all other grids in the specified group.

For example, suppose that 10 datasets have been loaded and we want to extend the wavelength grid of 5 of them beyond the range covered by the ARF grid. First, define a function to provide the necessary grid:

```
define extender (id, s)
{
   % <generate s.bin_lo, s.bin_hi here>
   return s;
}
```

We can then impose this model grid using:

```
set_eval_grid_method (USER_GRID, [3:8], &extender);
```

With this grid definition, ISIS will use the extended grid when evaluating the model, S(E), for datasets 3-8. Note that the model will be evaluated once for each dataset. If all these datasets have the same wavelength grid and if the same wavelength ranges are noticed in each, considerable computational work may be saved by specifying that the model be evaluated for one of these datasets and then simply rebinned for the others in this group. To turn on model-caching for the specified datasets, the optional fourth argument should be non-zero:

```
set_eval_grid_method (USER_GRID, [3:8], &extender, 1);
```

IMPORTANT: Note that, by default, the standard fit kernel evaluates the fit model only on those wavelength ranges that contribute to noticed data bins, as determined from the instrumental response. When fitting data with operators that represent convolutions, this behavior may not be desirable. In such cases, it may be necessary to compute the model and to apply the convolution operator over a much broader wavelength range. To ensure that the model is computed over the entire wavelength range specified by the user-defined grid, use the `eval` option on the standard kernel. For example,

```
set_kernel ([3:8], "std;eval=all");
```

To revert to the default behavior, use

```
set_kernel ([3:8], "std;eval=noticed");
```

set_fake

Purpose: Specify whether or not a dataset should be considered fake
Usage: `set_fake (hist_index, 0|1)`
See Also: `fakeit`

This function is used to change the default status of a given dataset. By default, spectral data generated by `fakeit` is considered to be fake and will be over-written by subsequent calls to `fakeit`. All other spectral data is considered to be "real" and will not be over-written by calls to `fakeit`.

For example, to over-write a real dataset with fake data:

```
% load ascii-format spectral data
  load_data ("ascii.dat");

% mark dataset 1 as fake data
  set_fake (1, 1);

% over-write dataset 1
  fakeit;
```

Alternatively, to keep a particular fake spectrum from being over-written by subsequent calls to 'fakeit', one could change the status of that dataset using

```
set_fake (id, 0);
```

set_frame_time

Purpose: Specify the frame-time for a data set
Usage: `set_frame_time (hist_index, frame_time_sec)`
See Also: `set_kernel, load_kernel, set_data_info`

This function is normally used in conjunction with the CCD photon pileup model. The frame time is specified in units of seconds.

get_min_stat_err

Purpose: Get the minimum acceptable counts uncertainty for a dataset
Usage: `min_stat_err = get_min_stat_err (hist_index)`
See Also: `Minimum_Stat_Err, set_min_stat_err, load_data, define_counts`

Retrieve the value of `min_stat_err` for a particular dataset. If `min_stat_err` has not been set, the return value is `-1`. For details, see `Minimum_Stat_Err`.

set_min_stat_err

Purpose: Modify the minimum acceptable counts uncertainty for a dataset
Usage: `set_min_stat_err (hist_index, min_stat_err)`
See Also: `Minimum_Stat_Err, get_min_stat_err, load_data, define_counts`

See `load_data` for details on how the `min_stat_err` value is defined and used. Normally, the `min_stat_err` value is specified on input, and it is rarely useful to modify the threshold value afterward.

When `min_stat_err` is modified using this function, the dataset's uncertainty values are not updated automatically. The uncertainty values are updated when the counts data are rebinned.

Note that because the imposition of the `min_stat_err` floor is intended to validate the input data, any below-threshold uncertainty values are replaced by the floor values and the original input values are not retained. To revert to the original input uncertainties, it may be necessary to reload the original input data.

set_post_model_hook

Purpose: Modify the computed model before computing the fit-statistic
Usage: `set_post_model_hook (id[], &func)`
See Also: `fit_fun, fit_counts`

Use this function to modify the computed model after the separate source and background contributions have been computed, but before the fit-statistic is computed. One can define a function of the form

```
m = func (lo, hi, c, b)
```

which will be called during fitting for the specified dataset. This function will be passed the original (ungrouped) wavelength grid of the detector (`lo`, `hi`) along with the predicted source (`c`) and background counts (`b`). It should return the predicted total counts in each detector bin; by default, it returns the sum of the source and background counts:

```
define post_model (lo, hi, c, b)
{
   return (c + b);
}
set_post_model_hook (id, &post_model);
```

This function may be used for a variety of other purposes, e.g. to plot the model at each iteration of a fit, to treat systematic errors or to examine alternate fitting techniques.

Note that this function may also include parameters which are varied during the fit. Suppose the desired hook-function depends on two parameters which we'd like to vary during the fit. In the following example, we show how to use a do-nothing function to introduce the two parameters:

```
define dummy_fit (l,h,p)
{
   return 1;
}
add_slang_function ("dummy", ["a", "b"]);

fit_fun ("dummy(1)*powerlaw(1)");

define tweak_model (lo, hi, counts, bgd)
{
   variable a, b, x, y;
   x = counts;
   a = get_par ("dummy(1).a");
   b = get_par ("dummy(1).b");

   y = x*(1 + a*x)*exp(-b*x) + bgd;

   return y;
}
set_post_model_hook (id, &tweak_model);
```

With this definition, the computed model has two additional parameters:

```
isis> list_par;
dummy(1)*powerlaw(1)
 idx  param              tie-to  freeze  value   min     max
  1  dummy(1).a              0     0         0     0       0
  2  dummy(1).b              0     0         0     0       0
  3  powerlaw(1).norm        0     0         1     0   1e+10
  4  powerlaw(1).PhoIndex    0     0         1    -2       9
```

which will vary during the fit, controlling the action of the hook-function provided by `set_post_model_hook`.

set_pre_combine_hook

Purpose: Modify the data and computed model spectra before computing spectral sums
Usage: `set_pre_combine_hook (id[], &func)`
See Also: `combine_datasets, set_eval_grid_method`

A pre-combine hook is a function that is called on behalf of a particular dataset immediately before that dataset (or its associated model) is added to one of the spectral sums accumulated by `combine_datasets`. Each dataset may be assigned a pre-combine hook. A pre-combine hook has the form:

```
define pre_combine_hook (id, y)
{
   % compute new_y
   return new_y;
}
```

where id is a dataset index and y and new_y are Double_Type arrays of the same length. To assign a pre-combine hook to one or more datasets, do:

```
set_pre_combine_hook (id, &pre_combine_hook);
```

where id is an integer dataset index or an array of indices. To disable the pre-combine hook, do:

```
set_pre_combine_hook (id, NULL);
```

For consistency when using a pre-combine hook, it is necessary for isis to separately evaluate the spectral model for each associated dataset instead of using the MERGED_GRID method which is the default for combine_datasets. For this reason, it is an error to assign a pre-combine hook to a dataset without also specifying an evaluation method different from MERGED_GRID. The most common evaluation method choice is likely to be SEPARATE_GRID:

```
gid = combine_datasets (id);
set_pre_combine_hook (id, &pre_combine_hook);
set_eval_grid_method (SEPARATE_GRID, id);
```

For details on different evaluation grid methods, see set_eval_grid_method.

As an example application of set_pre_combine_hook, consider a set of observed spectra in which each spectrum has a different Doppler shift, but you'd like to shift all the spectra into the rest frame before combining. In general, because the responses aren't easily shifted, you can't easily shift the data spectra before performing the fit. Assuming that the dataset-specific redshift is available using get_dataset_metadata, the necessary Doppler shift can be applied with the following pre-combine hook:

```
define shift_to_emitter_frame (y, lo, hi, z)
{
   variable
     new_lo = lo / (1.0 + z),
     new_hi = hi / (1.0 + z);

   return rebin (new_lo, new_hi, lo, hi, y);
}

define pre_combine_hook (i, y)
{
   variable
     z = get_dataset_metadata (i),  % redshift
     x = get_data_info (i),
     d = get_data_counts (i);

   variable
     lo = d.bin_lo[x.notice_list],
```

```
     hi = d.bin_hi[x.notice_list];

   return shift_to_emitter_frame (y, lo, hi, -z);
}

...
gid = combine_datasets (ids);
set_eval_grid_method (SEPARATE_GRID, ids);
set_pre_combine_hook (ids, &pre_combine_hook);
```

get_rebin_error_hook

Purpose: Retrieve error propagation hook
Usage: `Ref_Type = get_rebin_error_hook (hist_index)`
See Also: `set_rebin_error_method, set_rebin_error_hook`

Use this function to retrieve a reference to a dataset's error propagation hook, if any. The function returns `NULL` if the Poisson statistics are in effect (or if the dataset does not exist).

set_rebin_error_hook

Purpose: Change the way rebinning propagates uncertainties
Usage: `set_rebin_error_hook (hist_index, &hook)`
See Also: `rebin_data, group_data, rebin_dataset, set_min_stat_err`

Use this function to provide a S-Lang function to define how uncertainties are propagated when a counts spectrum is rebinned. By default, Poisson uncertainties are assumed. To change this default, provide a S-Lang function of the form

```
  define hook (orig_cts, orig_stat_err, grouping)
```

where

```
        orig_cts = the unbinned counts array
   orig_stat_err = the unbinned uncertainties
        grouping = index array specifying which bins are to
                   be grouped (see rebin_data() for details)
```

This function should return the array of uncertainties corresponding to the specified grouping. To indicate an error, the function should return `NULL`. Because ISIS validates the uncertainties returned by this routine, it may also be necessary to change the value of `min_stat_err` to ensure that the uncertainties computed by the error-hook are not modified (see `set_min_stat_err`). Uncertainties must always be positive – negative or zero values are not allowed.

For example, to add bin uncertainties in quadrature when dataset 1 is rebinned:

```
  define rebin_error_hook (orig_cts, orig_stat_err, grouping)
  {
     return sqrt(rebin_array (orig_stat_err^2, grouping));
  }
  set_rebin_error_hook (1, &rebin_error_hook);
```

Note that this error hook is equivalent to the built-in `quadsum` method supported by `set_rebin_error_method`.

To delete the error-hook for a particular dataset (reverting to Poisson statistics), use

```
    set_rebin_error_hook (index, NULL);
```

set_rebin_error_method

Purpose: Change the way rebinning propagates uncertainties
Usage: `set_rebin_error_method (hist_index, "method_name" [, &method_hook])`
See Also: `rebin_data, group, group_bin, group_data, rebin_dataset, set_min_stat_err`

Use this function to define the way statistical uncertainties are propagated when a counts spectrum is rebinned. Supported error propagation methods are:

```
 __Method_Name__      __Definition__
   poisson (default)  new_stat_err = sqrt (grouped_counts)

   quadsum            new_stat_err = sqrt (sum (stat_err^2))

   user               defined by 'method_hook' function reference.
                      (see set_rebin_error_hook for details)

EXAMPLES:
    set_rebin_error_method (1, "quadsum");
    set_rebin_error_method (1, NULL);     % equivalent to "poisson"

    set_rebin_error_method (1, "user", &my_rebin_method);
```

unassign_arf

Purpose: Undo the assignment of an ARF to a data set
Usage: `unassign_arf (hist_index_array)`
See Also: `load_arf, list_arf, assign_arf`

```
Example:
        unassign_arf (1);             % unassign the ARF for spectrum 1
        unassign_arf ([3:6]);         % unassign the ARF for spectra 3, 4, 5 and 6
```

unassign_back

Purpose: Undo the assignment of a background spectrum to a dataset
Usage: `unassign_back (id[])`
See Also: `assign_back, define_back, _define_back`

```
Example:
```

```
% unassign the background for spectrum 1
     unassign_back (1);

% unassign the background for spectra 3, 4, 5 and 6
     unassign_back ([3:6]);
```

unassign_rmf

Purpose: Undo the assignment of an RMF to a data set
Usage: `unassign_rmf (hist_index_array)`
See Also: `list_rmf`, `assign_rmf`

```
Example:
        unassign_rmf (1);             % unassign the RMF for spectrum 1
        unassign_rmf ([3:6]);         % unassign the RMF for spectra 3, 4, 5 and 6
```

uncombine_datasets

Purpose: Turn off dataset combination
Usage: `uncombine_datasets (gid[])`
See Also: `combine_datasets`, `match_dataset_grids`, `combination_members`, `get_combined`, `get_combined2`, `rebin_combined`, `set_pre_combine_hook`

unset_data_color

Purpose: Revert to using the current default plot color when plotting a data set
Usage: `unset_data_color (hist_index_list)`
See Also: `color`, `plot_data_counts`, `plot_auto_color`

This removes a color specification set by `set_data_color` and reverts to the default plot behavior.

use_file_group

Purpose: Apply PHA file grouping to a specified dataset
Usage: `use_file_group (id, file)`
See Also: `group_data`, `regroup_file`, `i2x_group`, `x2i_group`

Use this function to group a dataset to match the grouping in a specified `PHA` file. The length of the spectrum stored in the PHA file should match the length of the ungrouped dataset. This function will convert the OGIP-standard, energy-ordered grouping array from the file to an array that matches the isis convention (see `rebin_data`) and will apply that grouping to the specified dataset. If the file name is omitted, the file name associated with the dataset will be used (e.g. `get_data_info(id).file`).

For example:

```
% regroup a dataset to match a given PHA file
 use_file_group (id, "pha.fits");
```

i2x_group

Purpose: Convert isis-convention grouping array to OGIP-convention grouping array
Usage: `xg[] = i2x_group (ig[])`
See Also: `x2i_group, regroup_file, use_file_group`

The isis grouping array convention is described in the help page for `rebin_data`. See the xspec documentation for details on the OGIP grouping array convention.

x2i_group

Purpose: Convert OGIP-convention grouping array to isis-convention grouping array
Usage: `ig[] = x2i_group (xg[])`
See Also: `i2x_group, regroup_file, use_file_group`

The isis grouping array convention is described in the help page for `rebin_data`. See the xspec documentation for details on the OGIP grouping array convention.

7.3 Access to the Atomic Database

Because wavelength tables may be unavailable or may not provide a complete listing of all emission lines present in the line emissivity tables, the run-time atomic database is populated using *both* wavelength and energy level tables *and* the line-emissivity tables. For this reason, some of the functions described in this section may be useful even when atomic data files are unavailable.

To simplify access to a large database of emission line wavelengths, ISIS provides a number of functions to define, track and manipulate *groups* of lines defined by various criteria. Over-plotting line groups on observed spectra (`plot_group`) provides an important aid to line identification.

A line group is defined by a list of integers which represent the index of each group member in the internal wavelength table. This integer list may be held in a S-Lang array-variable or (optionally) stored in an internal table. A group may be registered in the internal table using `define_group`; groups may be deleted from this internal list using (`delete_group`). Each group in the internal list can be given a mnemonic name (`name_group`).

The user can browse the contents of a line group using `page_group`. Among other things, this output provides line indices which may be used to refer to lines individually. Given the line index, the `line_info` function can retrieve information on a single line. `group` returns an array containing the indices of all lines in a given group. Line groups may be saved to a disk file using `save_group`.

Lines can be grouped by element, ion (`el_ion`), wavelength (`wl`) and predicted flux in some specified spectral model (`flx`) using the `where` command which is a S-Lang intrinsic. The brightest lines in the current model (see `create_aped_fun` in §7.5) can be selected using the function `brightest`. Groups may be defined using logical expressions as arguments to the `where` command:

```
lines = brightest(20, where(el_ion([8,12],[3:6]) and wl(1,15)));
```

uses the current spectral model to select the brightest 20 lines in the range 1-15 Å which arise from O and Mg atoms ionized 3,4,5 or 6 times (e.g. O III, O IV, O V, OVI, Mg III, Mg IV, Mg V and Mg VI). Logical operators `not`, `and` and `or` and may be used in these expressions.

Because some emission line wavelengths may not be accurately known, high-resolution spectral data may provide better values than those listed in the atomic database. `change_wl` allows the user to provide "corrected" wavelength values for use in ISIS run-time analysis (the wavelength values in the FITS files are *not* affected).

Energy level diagrams can be plotted using `plot_elev` and `[o]plot_elev_subset`; line transitions can be over-plotted on energy level diagrams using `oplot_transitions`

add_abundances

Purpose: Add a new abundance table
Usage: `id = add_abundances (Struct_Type | name, abun[], Z[]);`
See Also: `set_abund, list_abund, get_abundances`

Use this function to define a new abundance table. The abundance table may be defined by a structure with fields `Z[]`, `abun[]` and `name` containing the proton number of each element, its abundance and the name of the abundance table, respectively. Alternatively, these values may be provided as separate arguments. The abundance values must be given as log cosmic abundance by number, relative to Hydrogen = 12.0. The return value is the index of the new table.

aped_bib

Purpose: Retrieve bibcodes for APED database sources
Usage: `Struct_Type = aped_bib (Struct_Type db, Integer_Type[] lines)`
See Also: `aped_bib_query_string, el_ion, wl, flx, trans, page_group`

```
EXAMPLE
    % Retrieve references for emission lines between 1-20 Angstrom
    ref_struct = aped_bib (aped, where(wl(1,20)));
```

aped_bib_query_string

Purpose: Make a URL to retrieve APED bibcodes from the ADS database
Usage: `String_Type = aped_bib_query_string (Struct_Type refs)`
See Also: `aped_bib`

```
EXAMPLE
    % Retrieve references for emission lines between 1-20 Angstrom
    ref_struct = aped_bib (aped, where(wl(1,20)));
    qs = aped_bib_query_string (ref_struct);

    () = system ("lynx -dump '$qs' > /tmp/qs.bib"$);
```

aped_fun_details

Purpose: Retrieve detailed information (e.g. emission line fluxes) on a computed APED spectral model
Usage: `List_Type = aped_fun_details ("name(id)")`
See Also: `create_aped_fun, aped_bib, aped_bib_query_string`

A multi-temperature plasma emission model with a custom parameterization can be constructed using `create_aped_fun`. A particular dataset may be modeled by a sum of such multi-temperature models – for example, consider a three component model:

```
xaped(1) + xaped(2) + xaped(3)
```

where each `xaped` component introduces N temperatures-density components,

```
T[i],  n[i], i=0,1,2,...N-1.
```

In the resulting model spectrum, any given emission line represents the sum of 3N contributions – 3 model components, with N temperature-density components each.

The `aped_fun_details` function can be used to retrieve all of these separate contributions to each emission line. For example, the contribution to emission line, k, from temperature/density component, i, of model component, j, can be retrieved like so:

```
   create_aped_fun ("xaped", p);
   fit_fun ("xaped(1) + xaped(2) + ... + xaped(N)");

 %  let 1 <= j <= N

   info = aped_fun_details ("xaped($j)"$);
   flux = info[i].line_flux[k];
```

The single required input parameter is a string specifying a particular instance of an APED model created by `create_aped_fun`.

The return value is a `List_Type` quantity with one `Struct_Type` entry for each temperature-density component of the APED model. Each structure contains selected model parameters:

```
   struct {density, temperature, line_flux,
           norm, redshift, vturb, metal_abund}
```

The `line_flux` field is an array of length equal to the number of emission lines in the APED database. For this reason, the line indices returned by other isis functions, e.g. `brightest`, may be used to index into this array.

```
EXAMPLE

   % define a spectral model
   create_aped_fun ("xaped", default_plasma_state());
   fit_fun ("xaped(1) + xaped(2)");

   % line index for O VIII Ly alpha 2p~^2P_{3/2} - 1s~^2S_{1/2}
   id = where(trans(O,8,4,1))[0];

   % view database parameters for this emission line
   print(line_info (id));

   % evaluate the spectral model (computing all the line fluxes)
   variable lo, hi, f;
   (lo, hi) = linear_grid (1, 20, 2000);
   f = eval_fun (lo, hi);

   % retrieve flux contribution to this line from the first
   % temperature-density component of xaped(1)
   info = aped_fun_details ("xaped(1)");
   flux0 = info[0].line_flux[ id ];
```

atoms

Purpose: load the atomic database
Usage: `atoms (Struct_Type)`
See Also: `plasma, db_push, db_list, db_select`

This function loads all the FITS-format atomic data files listed in the "filemap" file specified by the `atomic_data_filemap` field if the database configuration structure.

```
Example:
      isis>  s.atomic_data_filemap = "data/filemap";
      isis>  atoms (s);
```

Although it is not necessary to load the atomic data files, they may provide more information than is contained in the line and continuum emissivity tables (e.g. energy level data, transition probabilities, identification labels for the emission lines, etc.).

brightest

Purpose: get indices of the k brightest lines in a given list
Usage: `indices = brightest (k, line-list)`
See Also: `line_em, ratio_em, create_aped_fun`

A plasma emission model must be defined and a model spectrum computed *before* using this function. For the purpose of selecting the brightest lines, the total flux from each line is summed over all model components without regard to the relative Doppler-shifts or line-profile shapes involved.

```
Example:
                        % define and compute a model spectrum
load_model("model_dat");
(lo, hi) = linear_grid (1, 25, 6000);
flux = model_spectrum (lo, hi);

                        % find the indices of the
                        % 20 brightest Fe lines
                        % in that model spectrum

b = brightest(20, where(el_ion(Fe)));    % Fe = 26
```

change_wl

Purpose: Change internal wavelength values for specific lines
Usage: `change_wl("filename")`
See Also: `plasma, atoms`

This function loads a file containing a list of emission lines specified by element, ion, and upper and lower energy level indices along with values for the line wavelength and its uncertainty. The wavelengths and uncertainties in the file over-write those in the ISIS memory at run time (the wavelengths in the atomic database files on disk are *not* changed).

This function can be used to correct line wavelengths in the atomic database using wavelengths measured from observational data.

The wavelength file is ASCII-formatted with these columns in order from left to right:

```
       lambda [Angstrom]
   lambda_err [Angstrom]
proton_number
    ion state
  upper_level
  lower_level
```

Lines with a # in column 1 are ignored and may be used for comments.

db_indices

Purpose: Get indices of currently loaded spectroscopy databases
Usage: `Integer_Type[] = db_indices ()`
See Also: `db_push, db_pop, db_select, db_list, plasma, atoms`

db_list

Purpose: list currently loaded spectroscopy databases
Usage: `db_list ([Ref_Type | File_Type])`
See Also: `db_push, db_pop, db_select, db_indices, plasma, atoms`

This function produces a numbered list of spectroscopy databases. Each database is identified by the path to its filemap file. The currently selected database is indicated by an asterisk.

```
EXAMPLE:
  isis> db_list;
  Current database list:
   0 * /nfs/blackhole/d1/ji/ATOMDB/filemap_v2.0.0.rc7
   1   /nfs/cxc/a1/share/atomdb/filemap
```

If called with a `Ref_Type` argument, the `Ref_Type` will be assigned a string containing the database list. If called with a `File_Type` argument, the database list will be written to the indicate `File_Type`.

db_pop

Purpose: Remove a spectroscopy database from the internal list
Usage: `db_pop ([index])`
See Also: `db_push, db_select, db_list, db_indices, plasma, atoms`

This function removes a spectroscopy database from the internal database list. If no database index is specified, the operation applies to the currently selected database.

db_push

Purpose: load another spectroscopy database
Usage: `db_push (Struct_Type)`
See Also: `db_pop, db_list, db_select, db_indices, plasma, atoms`

Use this function to load another spectroscopy database by reading its contents from files. The `Struct_Type` argument should have the same form as the one used as input to the `plasma` function.

db_select

Purpose: select a spectroscopy database
Usage: `db_select (Int_Type k)`
See Also: `db_push, db_list, db_pop, db_indices, plasma, atoms`

Use this function to select a spectroscopy database from the currently list displayed by `db_list`. Subsequent calls to database access functions (e.g. `create_aped_fun`, etc.) will use atomic data and emissivities from the selected database.

define_group

Purpose: register a line group in the internal group list
Usage: `define_group (group_index, line-list)`
See Also: `delete_group, name_group, save_group, brightest, trans, el_ion, wl, flx`

.

Note: The main purpose of this internal list is bookkeeping. Because essentially all line-group functionality is available using line groups specified by S-Lang array-variables, there is no requirement to register a group in the internal list. Because it is not clear that this additional functionality is useful, it may not be supported in future releases.

A line group (`line-list`) is specified by an integer array of line indices. This array may be specified explicitly (e.g.

$$27, 39, 415$$

) or may be generated via a logical expression to the `where` command (a S-Lang intrinsic function). This function registers the specified group using the specified integer `group_index`, replacing any group already registered with that index.

```
Example:
  % group 2 = brightest 20 lines from Fe XVI, Fe XVII, Ni XVI, Ni XVII
  %      with wavelengths between 10-12 angstroms or 14-15 angstroms

      define_group(2, brightest(20,
          where( el_ion([Fe, Ni], [16,17])
                           and (wl(10,12) or wl(14,15)))));

  %  The same effect can be achieved using S-Lang array
  %  operations:

  % 'all' and 'bright' are arrays of integer line-indices:

      all = where( el_ion([Fe, Ni], [16,17])
                           and (wl(10,12) or wl(14,15)));
      bright = brightest(20, all);
      define_group(2,bright);
      define_group(3,all);      % retains the complete list,
```

delete_group

Purpose: delete a line group
Usage: `delete_group (group_index)`
See Also: `define_group`

The line group corresponding to `group_index` is deleted from the internal list of defined groups.

el_ion

Purpose: flag line transitions from selected elements and ions
Usage: `flag = el_ion ([proton_number_list], [ion_list])`
See Also: `define_group, trans, wl, flx, where, and, or, not`

```
Examples:

                      % indices of lines from Oxygen and Magnesium
o_and_mg = where( el_ion([O,Mg]));  % O = 8, Mg = 12

                      % group 2 = lines from ions VII, IX and XX
idlist = where( el_ion(, [7, 9, 20]));
define_group (2, idlist);

                      % group 1 = lines from Fe XVII, Fe XVIII, Fe XIX
define_group (1, where( el_ion(26, [17:19])));

                      % indices of lines from all elements except Iron
not_fe = where(not(el_ion(Fe)));    % Fe = 26
```

This function is intended to be used in conjunction with the S-Lang *intrinsic function* `where`. It returns a character array of length equal to the number of emission lines in the current database line list. If a given line belongs to one of the listed elements/ions, the corresponding array element is set equal to one, otherwise the array element is zero.

flx

Purpose: flag line transitions by which occupy a given flux range
Usage: `flag = flx ([flx_min], [flx_max])`
See Also: `define_group, where, and, or, not`

```
Examples:
                    % indices of lines brighter than f photons/cm^2/s
above_f = where ( flx(f) );

                    % indices of lines between f1 and f2 photons/cm^2/s
between_f1_f2 =  where ( flx(f1, f2));
```

This function is intended to be used in conjunction with the S-Lang *intrinsic function* `where`.

It returns a character array of length equal to the number of emission lines in the current database line list. If a given line falls within the specified flux range (photons $\mathrm{cm}^{-2}\ \mathrm{s}^{-1}$ for the currently defined model) ($F_{\min} \leq F < F_{\max}$) the corresponding array element is set equal to one, otherwise the array element is zero.

isis_linelabel_hook

Purpose: Reformat line label strings for plotting
Usage: `new_label = isis_linelabel_hook (label)`
See Also: `plot_group, line_label_default_style, latex2pg`

The energy level labels in the APED spectroscopy database are formatted using LaTeX-style syntax which must be reformatted for plotting purposes so that subscripts and superscripts are handled properly.

If defined in the Global namespace, the function `isis_linelabel_hook` is used to handle this reformatting. By default, `isis_linelabel_hook` is defined as

```
public define isis_linelabel_hook (s)
{
   return latex2pg (s);
}
```

where `latex2pg` is a simple parser which can handle most of the APED database energy-level labels.

To change the way line labels are reformatted, simply provide an alternate definition for `isis_linelabel_hook`.

lines_in_group

Purpose: get indices of lines in a given group
Usage: `indices = lines_in_group (group_index)`
See Also: `line_em, ratio_em`

This function retrieves indices of lines belonging to a group registered in the internal group-list. It simplifies the use of functions which require a list of line indices as input.

```
Example:

  % define a group by a list of line indices
  all = where( el_ion([Fe, Ni], [16,17]) );
  define_group (2, all);

  % retrieve a list of line indices from a group:
  list = lines_in_group(2);
```

line_info

Purpose: Get data for a single emission line
Usage: `Struct_Type s = line_info (line_index)`
See Also: `page_group`

This function returns a S-Lang structure defined as

```
struct
{
   id, lambda, A, flux, Z, ion,
   upper, upper_g, upper_E, upper_L, upper_S,
   lower, lower_g, lower_E, lower_L, lower_S,
   up_name, lo_name
 }
```

The structure fields give the line index (id), wavelength (lambda) and Einstein A-value (A), and identify the ion and energy levels involved (proton number (Z), ionization state (ion), upper and lower energy level indices (upper/lower) and labels (up_name/lo_name) if available. For each energy level, the level degeneracy (g), excitation energy (E) and LS coupling quantum numbers are also provided. The ionization state is the integer equivalent of the roman numeral; for example, an Fe XVII line would have `Z=26`, `ion=17`. The total flux from this line in the currently defined model (photons cm^{-2} s−1) is returned in `flux`.

line_label_default_style

Purpose: Get a structure with the default line labeling style parameters
Usage: `s = line_label_default_style ()`
See Also: `plot_group, isis_linelabel_hook`

The returned structure has the following fields:

```
___Field______Type____Default___Definition________________________________
 label_type   int      0        0 for short labels (e.g. Ca XI);
                                1 for long labels (e.g. Ca XI 2p 1s)
char_height   float    1.0
    justify   float    0        0=right, 0.5=centered, 1.0=left
      angle   float   90.0      [degrees]
  top_frac    float    0.6      coordinate of upper endpoint of indicator
                                line, as a fraction of the Y plot-range
bottom_frac   float    0.7      coordinate of lower endpoint of indicator
                                line, as a fraction of the Y plot-range
    offset    float    0.1      spacing of text above upper endpoint of
                                indicator line as a fraction of the
                                indicator line length
```

list_branch

Purpose: List radiative transition branching ratios for an ion
Usage: `list_branch (proton_number, ion)`
See Also: `list_elev`

For each energy level having more than one downward radiative transition, this function lists the fraction of spontaneous decays going into each downward transition along with the transition wavelength (Å), A-value (s^{-1}), lower level index and line list index.

```
For example:

isis> list_branch (He, 2);                 % He II

He II
upper level =    5
lower         lambda       branch        A         index
    3   1.640375e+03   3.3337e-01   3.3690e+07      14
    4   1.640533e+03   6.6663e-01   6.7370e+07      15
 ....
```

The value b_{ul} given in the `branch` column is defined by

$$b_{ul} = \frac{A_{ul}}{\sum_{k=1}^{u-1} A_{uk}} \tag{7.25}$$

where A_{uk} is the Einstein A value for the transition from upper level u to lower level k and where the ground state is $k = 1$.

list_elev

Purpose: browse the energy level structure of an ion
Usage: `list_elev (proton_number, ion)`
See Also: `plot_elev`

This command displays a listing of the energy level data for the specified ion using a user defined pager. The energy level data includes the level index, excitation energy (eV), statistical weight, the total spontaneous downward transition rate (s^{-1}), the number of downward radiative transitions, nLS quantum numbers, and a name string.

The pager is specified by the PAGER environment variable. If the PAGER variable is not set, `more` is the default. If `more` is not found on the command search path, the function will fail with an error message.

list_group

Purpose: list the currently defined line groups
Usage: `list_group`
See Also: `define_group, name_group`

This function lists the number of lines in each group registered in the internal group-list along with the group-name, if one has been defined using `name_group`.

name_group

Purpose: Define a mnemonic name for a group of lines
Usage: `name_group (group_index, "group-name-string")`
See Also: `define_group, list_group`

This function is used to provide a mnemonic name for a group of lines registered in the internal group-list.

```
Example:
    name_group (3, "Bright Fe XVII lines");
```

oplot_lines

Purpose: over-plot line transitions on an energy level diagram
Usage: `oplot_lines (proton_number, ion, line_list, [style])`
See Also: `plot_elev, [o]plot_elev_subset`

.

This function over-plots the line transitions in `line_list` on an existing energy level diagram; `line_list` may be a single line index or an array of line indices.

```
Example:
       % plot an energy level diagram showing how the 10 brightest
       % Fe XVI lines form (assuming a spectral model has been computed)

     Fe = 26;
     plot_elev (Fe,16);
     bright_fe16 = brightest (10, where(el_ion(Fe,16)));
     oplot_lines(Fe,16, bright_fe16);
```

Repeated invocations of `oplot_lines` will automatically switch line styles to help distinguish the overlaid line transitions. Depending on the current setting [see `style`] either colors or line styles (e.g. solid vs. dashed) are used to distinguish overlaid data sets. To override the automatic style changes, the style index can be specified explicitly for each plot. Alternatively, `plot_auto_color` can be used to disable the automatic style changes.

page_group

Purpose: browse the data for lines in a given group
Usage: `page_group (group_index or line_list)`
See Also: `plot_group, trans, el_ion, wl, flx`

Line data for the specified lines is presented using a user-defined pager program. For each line, the listing provides these values:

```
Line index
Wavelength [Angstrom]
Element name
Ion state (by charge or Roman numeral)
Einstein A [1/s]
Flux [photons/s/cm^2]
Energy level names and indices
```

The integer line index can be used to refer to a specific line in other commands e.g. `plot_group`

An asterisk (*) by the wavelength indicates that emissivity data for that line is loaded. The flux is computed using the currently defined spectral model. If no model is defined, the flux is listed as zero.

The pager is specified by the PAGER environment variable. If the PAGER variable is not set, `more` is the default. If `more` is not found on the command search path, the function will fail with an error message.

plot_elev

Purpose: plot the energy level diagram of an ion
Usage: `plot_elev (proton_number, ion)`
See Also: `list_elev`

An LS coupling energy level diagram is plotted using the current format settings for color, line style, etc. This function requires that LS coupling quantum numbers be available for all energy levels.

plot_elev_subset

Purpose: plot a subset of the energy level diagram of an ion
Usage: `[o]plot_elev_subset (proton_number, ion, line_list)`
See Also: `plot_elev, oplot_lines`

This function [over]plots an energy level diagram consisting of all energy levels involved in the given list of line transitions (`line_list`). It can be used to examine a subset of the available energy levels or to generate a color-coded energy level diagram.

plot_group

Purpose: over-plot a spectrum with all lines belonging to a particular group
Usage: `plot_group(group [, color_index [,label_style [, redshift]]])`
See Also: `plot_linelist, line_label_default_style, isis_linelabel_hook, page_group, save_group`

All lines in the specified group are displayed on the current plot, meaning that a vertical line is plotted at the line wavelength along with a text label. A line group may be specified by an array of line indices or by giving the index of the group in the internal group-list.

This assumes that the currently active plot window already contains a spectrum plot.

If `color_index` is not specified, the current plot-color is used; if `color_index` is specified, the color applies only to the line labels and is not retained in the internal plot format parameter structure. The third argument is used to specify formatting information for the line labels. If this argument is an integer, it specifies whether short (0) or long (1) text labels should be used. This argument may also be a structure containing multiple style parameters; see `line_label_default_style` for details.

To plot lines at some Doppler shifted wavelength, specify the redshift value defined as

$$\text{redshift} = z = \frac{\lambda_{\text{observed}}}{\lambda_{\text{emitted}}} - 1 \tag{7.26}$$

```
Example:

plot_group (2);          % plot group 2 with short labels

plot_group (where(el_ion(Mg)), 2, 1);   % plot Mg lines using long
                                        % labels in color 2
```

Similarly, to obtain red line labels which are 25% smaller and tilted at a 45 degree angle with an indicator line extending all the way to the plot border, use

```
s = line_label_default_style ();
s.char_height = 0.75;
s.angle = 45;
s.bottom_frac = 0.0;
plot_group (g, red, s);
```

plot_linelist

Purpose: over-plot a spectrum with a user-provided line list
Usage: `plot_linelist (lambdas, labels, [, color_index [,label_style [, redshift]]]])`
See Also: `line_label_default_style, isis_linelabel_hook, page_group, save_group`

This function is essentially the same as `plot_group` except that the line wavelengths and labels are provided explicitly by the user rather than being drawn from the spectroscopy database.

save_group

Purpose: save line parameters to an ASCII file
Usage: `save_group (group_index or line_list, filename)`
See Also: `page_group`

A list of lines may be specified using a S-Lang array-variable or by specifying the index of a group in the internal group-list. Repeated saves to the same filename append data to the file rather than overwriting it.

```
Example:
   fe17 = where (el_ion(Fe,17));
   save_group (fe17, "fe17_lines.txt");
```

trans

Purpose: flag line transitions from a given ion
Usage: `flag = trans ([proton_number[, ion_number [, upper_list[, lower_list]]]])`
See Also: `define_group, el_ion, wl, flx, where, and, or, not`

```
Examples:

            % g = all lines
g = where( trans());
            % g = Neon line list
g = where( trans(Ne));
            % g = Ne IX lines
g = where( trans(Ne, 9));
            % g = Ne IX lines from levels 20-30 downward
g = where( trans(Ne, 9, [20:30]));
            % g = Ne IX lines from levels 20-30 down to levels 1-5
g = where( trans(Ne, 9, [20:30], [1:5]));
            % g = Ne IX lines down to levels 1-5
g = where( trans(Ne, 9, , [1:5]));
```

This function is intended to be used in conjunction with the S-Lang *intrinsic function* `where`. It returns a character array of length equal to the number of emission lines in the current database line list. If a given line belongs to one of the listed elements/ions, the corresponding array element is set equal to one, otherwise the array element is zero.

unblended

Purpose: get indices of unblended lines in a given list
Usage: `indices = unblended (frac, wl_sep, allowed_type, line-list)`
See Also: `page_group, el_ion, line_em, ratio_em`

A line with wavelength λ_k is defined to be "unblended" if the flux from the line of interest dominates the total flux from nearby *contaminating* lines. More precisely, the criterion is that $\sum_{j\neq k} F(\lambda_j) < \texttt{frac} \times F(\lambda_k)$ where the summation extends over all *contaminating* lines with wavelength λ_j within $\pm\texttt{wl\_sep}\lambda_k$ of the line of interest (λ_k). The blend type `allowed_type` specifies which lines are to be considered contaminants.

`allowed_type = 0` indicates that wavelength spacing is the only determining factor; any other line falling within the stated wavelength band limit is considered a contaminant. `allowed_type = SAME_ELEM` means that lines from the same element are not considered contaminants and are excluded from the summation over j. Similarly, `allowed_type = SAME_ION` means lines from the same ionization state are not considered contaminants. To find lines free of blends with any ion of any element, use `allowed_type = SAME_ELEM & SAME_ION`.

Note that, because this function requires line fluxes, a spectrum model must be computed *before* using it.

```
Example:
                      % define and compute a model spectrum
```

```
load_model("model_dat");
(lo, hi) = linear_grid (1, 25, 6000);
flux = model_spectrum (lo, hi);

                    % find the indices of the
                    % 100 brightest lines between 1-100 Angstrom
                    % in that model spectrum

b = brightest (100, where(wl(1,100)));

                    % lines with < 10% "contamination"
                    % from other elements
                    % within +/- 0.001*lambda

ue = unblended (0.1, 0.001, SAME_ELEM, b);

                    % lines for which > 90%
                    % of the flux within +/- 0.001*lambda
                    % comes from a single ion of a single
                    % element:

uei = unblended (0.1, 0.001, SAME_ELEM & SAME_ION, b);
```

For the purpose of selecting unblended lines, the total flux from each line is summed over all model components without regard to the relative Doppler-shifts or line-profile shapes involved. In particular, the current implementation of this function does not support searching for unblended lines in models which have components at significantly different redshifts; wavelength separation is always determined using the rest-wavelength.

wl

Purpose: flag line transitions from a given wavelength range
Usage: `flag = wl ([min_wavelen], [max_wavelen])`
See Also: `page_group, where, and, or, not`

```
Examples:

% indices of all lines longward of 20 angstroms
  above_20 = where(wl(20));

% indices all lines shortward of 30 angstroms
  below_30 = where(wl(,30));

% indices Fe XVI lines between 10 and 15 angstroms
  fe_16_subset = where( el_ion(26,16) and wl(10,15));
```

This function is intended to be used in conjunction with the S-Lang *intrinsic function* `where`. It returns a character array of length equal to the number of emission lines in the current database line list. If a given line's rest wavelength lies inside the specified wavelength range ($\lambda_{\min} \leq \lambda < \lambda_{\max}$) the corresponding array element is set equal to one, otherwise the array element is zero.

7.4 Access to the Plasma Emissivity Database

The emissivity database contains theoretical models of the state of an emitting plasma as a function of its physical state variables. The standard emissivity database is generated for conditions of collisional ionization equilibrium. Although ISIS is designed to make it possible to work with other databases (e.b. photoionized plasmas), at this writing the collisional ionization database is the only one available.

In collisional equilibrium, the primary physical state variables are the electron temperature T and the electron density n. Observed line intensities and line ratios may be used to infer the temperature and density (or distribution of temperatures and densities) in the emitting plasmas. To facilitate this analysis, ISIS provides several functions to manipulate the line emissivity tables.

`line_em` retrieves the emissivity of a single line (or the sum over lines in a group) as a function of density and/or temperature. `ratio_em` generates line (or line group) ratios as a function of density and/or temperature. Details of the ionization balance as a function of temperature can be examined using `ion_bal` and `ion_frac`. Each of these functions returns arrays of emissivities or ion-fractions when can then be plotted (`[o]plot`) or used in computations via the S-Lang array-oriented mathematical operators.

For information on computing spectral models see §7.5.

db_grid

Purpose: Returns the density and temperature grid used by the spectroscopy database
Usage: `Struct_Type = db_grid ([file | Struct_Type])`
See Also: `plasma, [v]list_db`

`db__grid` returns a structure of the form

```
s = struct {temp, dens}
```

whose fields are arrays giving the temperature [Kelvin] and electron density [cm^{-3}] grids used to compute emissivities in the spectroscopy database.

If no argument is given, the grid used by the currently loaded database is returned.

free_alt_ioniz

Purpose: Free memory for alternate ionization balance table
Usage: `free_alt_ioniz ()`
See Also: `load_alt_ioniz`

If an alternate ionization balance table is loaded, it is used to re-scale the line and continuum emissivities. Freeing the alternate ionization balance table reverts to the ionization balance that was used to generate the emissivity tables (no re-scaling is then performed).

get_abundances

Purpose: Get abundance table values
Usage: `Struct_Type = get_abundances ([id | name]);`
See Also: `set_abund, list_abund, add_abundances`

This function returns a structure with fields `Z[]`, `abun[]` and `name` containing the proton number of each element, its abundance and the name of the abundance table, respectively. The abundance table may be specified either by the name or numerical index shown in the listing produced by `list_abund`. If no table is specified, the current default table is returned. The abundance values are normally given as log cosmic abundance by number, relative to Hydrogen = 12.0.

get_contin

Purpose: get continuum contributions by ion
Usage: `p = get_contin (lo, hi, temp, [dens], [Z], [ion])`
See Also: `plasma, line_em`

This function provide access to the continuum emissivity tables in the spectroscopy database.

```
  lo = bin low edge [Angstrom]
  hi = bin high edge [Angstrom]
temp = electron temperature (K)
dens = electron density [cm^(-3)] [default=1.0]
   Z = proton number
 ion = ion state (1, 2, ... Z+1)
```

For the specified electron temperature, this function returns the continuum spectral contribution (photon $\mathtt{cm}^3$ $\mathtt{sec}^{-1}$) vs. wavelength (Å) for the true continuum (`true`), which is the sum of Bremsstrahlung, radiative recombination and two-photon emission, and the pseudo-continuum due to weak line emission (`pseudo`) using the input wavelength grid.

These spectra are organized in a structure defined by

```
struct
{
   true, pseudo
}
```

(each structure component is a S-Lang floating point array of length N).

If the electron density is not specified, the default value is for a low-density coronal plasma (or whatever is available in the database files). If a particular element is specified (by proton number `Z`) the result contains the continuum contributions for that element only (summed over ions), otherwise, the result will be summed over all available elements. Similarly, if a specific ion is specified (by proton number `Z` and ionization state `j`), the result contains the contribution for that ion only.

```
Example:
```

```
plasma (aped);                        % initialize the database

(lo, hi) = linear_grid(1,20,5000);    % define a wavelength grid
p = get_contin (lo, hi, 2.e6);        % get continua for T=2.0e6 K
hplot(lo, hi, p.true);                % plot the true continuum
```

ion_bal

Purpose: get ion fractions at a specified temperature
Usage: `(ion, frac) = ion_bal (proton_number, temp, [ioniz_table_id])`
See Also: `ion_frac, load_alt_ioniz`

The ion fractions are normalized so that the sum over ions of a given element is unity.

```
          temp = input array of temperatures [K]
          frac = output array of ion fractions
           ion = output array of ion states
                 (1 = neutral, 2 = once ionized, etc.)
ioniz_table_id = index of ionization table (0=std or 1=alt)
```

The ionization table index is used only if more than one ionization table is loaded (see also `load_alt_ioniz`); the default is to use the standard table (`ioniz_table_id = 0`).

ion_frac

Purpose: get ion fraction vs. temperature
Usage: `frac = ion_frac (proton_number, ion, temp, [ioniz_table_id])`
See Also: `ion_bal, load_alt_ioniz`

The ion fractions are normalized so that the sum over ions of a given element is unity.

```
            temp = input array of temperatures [K]
            frac = output array of ion fractions
  ioniz_table_id = index of ionization table (0=std or 1=alt)
```

The ionization table index is used only if more than one ionization table is loaded (see also `load_alt_ioniz`); the default is to use the standard table (`ioniz_table_id = 0`).

line_em

Purpose: Compute line-group emissivity vs. temperature and/or density
Usage: `emis = line_em (l_array, [temp] [,dens])`
See Also: `group, ratio_em, line_em1, get_contin`

```
_Variable_____Size______Value____________________________________
  l_array      nl      Input 1-D array of line indices
     temp      nt      Input 1-D array of temperatures [K]
     dens      nd      (optional) Input 1-D array of densities [cm^(-3)]
     emis   (nt)x(nd)  Output 1-D or 2-D emissivity array.
```

This function computes

$$\texttt{emis} = \varepsilon\left(T,\rho\right) = \sum_k \varepsilon\left(T,\rho;k\right) \tag{7.27}$$

where $\varepsilon\left(T,\rho;k\right)$ is the emissivity of emission line k at temperature T and density ρ with elemental abundance and ionization balance factors included. When the density array is absent (the lowest available density data is used as the default) and when either $nt = 1$ or $nd = 1$, `emis` is a 1-D array of size $\max(nt, nd)$. When both $nt > 1$ and $nd > 1$, `emis` is a $nt \times nd$ 2-D array with the first index corresponding to density and the second to temperature. For example,

```
t=10.0^[6.0:7.0:0.05];      % temperature array, nt=21
                            % log-spacing
d=10.0^[11.0:13.0:0.25];    % density array, nd=9
                            % log-spacing

k = lines_in_group(2);      % list of lines in group 2

em2 = line_em(k,t,d);       % em2 = 2-D emissivity array
                            % with dimension [9,21]

y = em2[3,7];               % line emissivity at
                            % density d[3], temperature t[7],
                            % summed over lines in group 2.

emt = line_em(k[2], t, d[4]);  % emt = 1-D emissivity array
                               % with dimension [21]

z = emt[9];                 % z = emissivity line line k[2] at
                            % density d[4], temperature t[9]
```

line_em1

Purpose: Retrieve line emissivity table
Usage: `(emis, temp, dens) = line_em1 (line_index)`
See Also: `group, ratio_em, line_em, get_contin`

This function is similar to `line_em` except that it returns the line emissivity values tabulated in the spectroscopy database rather than interpolating those values onto a user-specified temperature/density grid. For a particular line, it returns three arrays ε_i, n_i, T_i, of size N, (where N is the number of (n, T) points available in the database) giving the emissivity as a function of density and temperature:

$$\varepsilon_i(n_i, T_i), i = 1, 2, ..N \tag{7.28}$$

list_abund

Purpose: List the available cosmic abundance tables
Usage: `list_abund ([verbose]);`
See Also: `set_abund, get_abundances`

The abundance table in the spectroscopy database may contain several sets of elemental abundances. This function lists the available abundance tables providing an integer index which is used in function arguments to select a particular table. By default, the optional argument is `verbose=0` so that only the names of the tables are listed. Set `verbose=1` to see the abundance values listed as well.

list_db

Purpose: List density and temperature range covered by the spectroscopy database
Usage: `list_db ([file | Struct_Type])`
See Also: `plasma, vlist_db, db_grid`

`list_db` generates a short listing showing only the available range of density and temperature; `vlist_db` generates a more verbose listing showing all available density-temperature points.

If no argument is given, the grid used by the currently loaded database is listed.

vlist_db

Purpose: List density and temperature range covered by the spectroscopy database
Usage: `vlist_db ([file | Struct_Type])`
See Also: `plasma, list_db, db_grid`

Similar to `list_db`, but provides a more detailed listing.

load_alt_ioniz

Purpose: Load an alternate ionization balance table
Usage: `load_alt_ioniz (file)`
See Also: `free_alt_ioniz`

Use this function to specify an alternate ionization balance table. Such a table may be used to re-scale the plasma emissivities as a simple way of seeing the effect of an alternate ionization balance calculation without repeating a complete spectrum synthesis calculation.

By default, when two ionization balance tables are loaded, all emissivities are re-scaled using the alternate ionization balance table. The emissivities are re-scaled so that

$$\varepsilon_1 = \frac{X_1(T; Z, q)}{X_0(T; Z, q)}\varepsilon_0 \tag{7.29}$$

where ε_0 is the tabulated emissivity computed using ionization fraction $X_0(T; Z, q)$ for ion (Z, q) at temperature T and where $X_1(T; Z, q)$ is the ionization fraction from the alternate ionization balance table.

To revert to the standard ionization balance table, delete the alternate table using `free_alt_ioniz`.

plasma

Purpose: load the emissivity database
Usage: `plasma (Struct_Type [,density_range [, temp_range]] )`
See Also: `atoms, db_push, db_list, db_select, [v]list_db`

This function is similar to the `atoms` function and uses the same database configuration structure. It causes ISIS to load the atomic data files and the line and continuum emissivity tables into memory (actually, because of their size, the continuum emissivity tables are just scanned so that continua may be quickly retrieved from disk as needed).

The optional arguments `density_range` and `temp_range` are two element floating point arrays specifying a subset of densities (cm^{-3}) and temperatures (Kelvin) for which line and continuum emissivities are to be loaded from the database.

Access to the emissivity tables is usually most efficient if the all of the tables fit in memory at once (the default mode assumes that this is the case). For large databases, however, this may not be practical.

To support very large line lists, it may be necessary to increase the size of an internal hash table. To increase the size of the table, set the intrinsic variable `EM_Hash_Table_Size_Hint` to a value at least 25% larger than the number of emission lines in your database.

The `Use_Memory` variable is used to indicate whether the emissivity tables should be loaded into memory all at once or should be accessed from disk files as needed:

```
  __Use_Memory__     ____Behavior____
     0                Read lines and continua on-demand
     1                Read lines up front, continua on-demand
     2                Read lines on-demand, continua up front
     3                Read lines and continua up front
```

As long as sufficient memory is available, model computations will be fastest if all the data is loaded into memory at once (`Use_Memory=3`, the default), thereby minimizing the number of relatively slow disk accesses. However, when not making use of the continuum tables, one might prefer `Use_Memory=1`; by not loading the continuum tables, database input is faster and the run-time memory footprint is minimized. On small memory machines, one might prefer `Use_Memory=0`.

ISIS maintains a lookup table containing a complete list of all lines in both the atomic database and in the emissivity database.

In order to ensure that this table is complete, ISIS normally scans the line emissivity tables on input and merges any "new" lines into the internal tables. However, this step is necessary only if the spectroscopy database wavelength tables are incomplete (in the sense that some lines in the emissivity tables are not listed in the wavelength tables loaded by ISIS). To stop ISIS from carrying out this (sometimes slow) scanning step, set `Incomplete_Line_List = 0`; when this variable is non-zero, input line emissivity tables will be scanned automatically.

Note that if the scan is not performed and "new" lines are present in the line emissivity tables, those lines will be ignored and an error message will be generated.

ratio_em

Purpose: compute a line-group emissivity ratio vs. temperature and density
Usage: `ratio = ratio_em (l1_array, l2_array [,temp [,dens]])`
See Also: `line_em`

This function computes

$$R(T,\rho) = \frac{\varepsilon_1(T,\rho)}{\varepsilon_2(T,\rho)} \tag{7.30}$$

where $\varepsilon_g(T,\rho)$ is the result of `line_em` for line group g.

set_abund

Purpose: Choose a cosmic abundance table
Usage: `set_abund (id | name);`
See Also: `list_abund, get_abundances, atoms, plasma`

The abundance table in the spectroscopy database may contain several sets of elemental abundances. This function specifies an abundance table either by name or by index in the list of tables generated by `list_abund`. If the specified abundance table is different from the one used to generate the line and continuum emissivity tables, the emissivities are re-scaled using the specified table.

7.5 Defining a Plasma Emission Model

The section describes how to define a plasma emission model by summing line and continuum emissivity components tabulated in the spectroscopy database. Plasma models may also be obtained using the `XSPEC` module (see §8).

After initializing the spectroscopy database (see §7.3 and §7.4), multicomponent plasma models may be generated using the function `create_aped_fun()` (See §7.7).

By default, spectral line profiles are delta-functions; `use_thermal_profile` specifies that a thermal line profile should be used instead.

append_model

Purpose: Append model components using S-Lang arrays
Usage: `append_model (state[]);`
See Also: `default_plasma_state, model_spectrum, define_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`*.*

This function is identical to the `define_model` function except that it appends an array of components to an existing model. If no model has been defined, a new model is created. See `define_model` for details.

create_aped_fun

Purpose: Create a custom fit-function using a multi-component APED spectrum
Usage: `create_aped_fun (name, Struct_Type [, hook_ref])`
See Also: `plasma, fit_fun, eval_fun2, aped_fun_details, create_aped_line_modifier, create_aped_line_profile, aped_line_modifier_args, aped_line_profile_args, aped_hook_args`

This function creates a custom fit-function which computes a multi-component APED model. The first argument gives the name of the new fit-function and the second argument is a `Struct_Type` which defines the number of spectrum components in the function's fit-parameter list. The values contained in this structure provide default values for the associated fit parameters. The optional third argument is used to select specific lines and continua and is discussed further below. Further customization can be achieved by making use of one or more modifier functions as described in the examples below.

The `Struct_Type` argument has the form of the structure returned by the `default_plasma_state()` function:

```
struct { norm, temperature, density,
         elem, elem_abund, metal_abund,
         vturb, redshift }
```

where

```
        norm[] = float array of size Num_Components
```

```
temperature[] = float array of size 1 or Num_Components
    density[] = float array of size 1 or Num_Components
       elem[] = NULL or int array of size Num_Elements
 elem_abund[] = NULL or float array of size Num_Elements
metal_abund = float
      vturb = float
   redshift = float
```

All components in the resulting model have the same values of `metal\_abund`, `vturb`, `redshift`, `elem`, `elem\_abund`. For detailed definitions of these struct fields, see `default_plasma_state()`.

EXAMPLE 1:

In the simplest case, a new fit-function can be generated with one line:

```
create_aped_fun ("xaped", default_plasma_state());
```

This creates a fit-function called `xaped` which allows fitting a single temperature APED spectrum. This function can be used like any other fit-function; the resulting fit-parameter list looks like this:

```
isis> create_aped_fun ("xaped", default_plasma_state());
isis> fit_fun ("xaped(1)");
isis> list_par;
xaped(1)
 idx  param              tie-to  freeze  value     min   max
  1  xaped(1).norm          0     1          1       0     0
  2  xaped(1).temperature   0     1      1e+07       0     0
  3  xaped(1).density       0     1          1       0     0
  4  xaped(1).vturb         0     1          0       0     0
  5  xaped(1).redshift      0     1          0       0     0
  6  xaped(1).metal_abund   0     1          1       0     0
isis>
```

Any parameters not explicitly specified will receive reasonable defaults. Note that by default, all parameters are frozen and the parameter ranges are unbounded. To fit such a function to data, first thaw a few selected parameters and optionally provide reasonable parameter ranges.

EXAMPLE 2:

Generating multicomponent models is not much more complicated. To define a model which corresponds to an APED spectrum with 3 temperature components:

```
variable n = 3;
variable t = default_plasma_state ();

t.norm = Double_Type[n];
t.norm[*] = 1.0;
t.temperature = 1.e6 + (1.e7 * [1:n])/n;
create_aped_fun ("xaped", t);
```

This yields a function with the following parameter list:

```
isis> fit_fun ("xaped(1)");
isis> list_par;
xaped(1)
idx  param              tie-to  freeze  value    min   max
 1  xaped(1).norm1          0     1          1      0     0
 2  xaped(1).norm2          0     1          1      0     0
 3  xaped(1).norm3          0     1          1      0     0
 4  xaped(1).temperature1   0     1    4333333      0     0
 5  xaped(1).temperature2   0     1    7666667      0     0
 6  xaped(1).temperature3   0     1    1.1e+07      0     0
 7  xaped(1).density        0     1          1      0     0
 8  xaped(1).vturb          0     1          0      0     0
 9  xaped(1).redshift       0     1          0      0     0
10  xaped(1).metal_abund    0     1          1      0     0
```

EXAMPLE 3:

As a more complex example, one could specify an overall metal abundance of 0.5 solar and specify e.g. Ne, Mg and Fe explicitly:

```
t.norm = Double_Type[n];
t.norm[*] = 1.0;
t.temperature = 1.e6 + (1.e7 * [1:n])/n;
t.density = 1.0;
t.metal_abund = 0.5;
t.elem = [Ne, Mg, Fe];
t.elem_abund = [1.0, 3.1, 0.0];
t.redshift = 0.0;

create_aped_fun ("xaped", t);
```

This produces the following parameter list:

```
xaped(1)
idx  param              tie-to  freeze value   min    max
 1  xaped(1).norm1          0     1         1     0      0
 2  xaped(1).norm2          0     1         1     0      0
 3  xaped(1).norm3          0     1         1     0      0
 4  xaped(1).temperature1   0     1   4333333     0      0
 5  xaped(1).temperature2   0     1   7666667     0      0
 6  xaped(1).temperature3   0     1   1.1e+07     0      0
 7  xaped(1).density        0     1         1     0      0
 8  xaped(1).vturb          0     1         0     0      0
 9  xaped(1).redshift       0     1         0     0      0
10  xaped(1).metal_abund    0     1       0.5     0      0
11  xaped(1).abund_Ne       0     1         1     0      0
12  xaped(1).abund_Mg       0     1       3.1     0      0
13  xaped(1).abund_Fe       0     1         0     0      0
```

EXAMPLE 4:

The optional `hook_ref` argument can be used to indicate that the model should compute only a specified combination of lines and continua. It provides a reference to a function defined as

```
Struct_Type f = hook (id)
```

where the `id` argument provides the instance of the current fit-function (e.g. 1 for `xaped(1)` and 4 for `xaped(4)`) and the returned structure looks like

```
variable f = struct {contrib_flag, line_list}
```

These struct fields carry the corresponding optional arguments which are accepted by `mt_calc_model` and `model_spectrum` (see `model_spectrum` for details).

For example, to generate a function which can compute the line-spectrum of a single ion (with model continuum excluded), we can do the following: In addition to the above `Struct_Type` parameter definitions, we first provide an appropriate hook function:

```
% we want to look at Ne and Mg lines:
variable Ne_Lines = where (el_ion(Ne,10));
variable Mg_Lines = where (el_ion(Mg,12));
variable Contrib_Flag = MODEL_LINES;

define xaped_hook (id)
{
   variable f = mt_model_qualifier();
   f.contrib_flag = Contrib_Flag;

   if (id == 1) f.line_list = Ne_Lines;
   else if (id == 2) f.line_list = Mg_Lines;
   else f = NULL;

   return f;
}

create_aped_fun ("xaped", t, &xaped_hook);
```

With this definition, `xaped(1)` will compute the Ne X line spectrum and `xaped(2)` will compute the Mg XII line spectrum. Note that the global variable `Contrib_Flag` defined in this example provides a mechanism to alter the behavior of both functions. without changing `xaped_hook` or defining a new function via `create_aped_fun`.

EXAMPLE 5:

In some applications, it may be useful to modify selected line emissivities according to a user-defined function. When this user-defined function has adjustable parameters, it may be useful to allow those parameters to vary during a fit.

For example, suppose we've defined a custom APED spectral model as defined above:

```
create_aped_fun ("xaped", template_struct);
```

To adjust the emissivity of selected spectral lines, we can write a S-Lang function of the form:

```
define line_emis_modifier (params, line_id, state, emis)
{
   variable info = line_info(line_id);
```

```
      %
      % ... modify the emissivity of selected emission lines ...
      %
     return emis;
  }
```

This function accepts an array of parameters, an integer line index, a structure containing plasma state information, and the value of the line emissivity for the current plasma state. The function should return a new value for the line emissivity. The plasma state structure has the form form:

```
state = struct {temperature, ndensity}
```

where `temperature` is the Kelvin temperature and `ndensity` is the number density in `cm^-3`. This function will be called while the `xaped` function is being computed and its adjustable parameters will be included in the fit.

To enable this function to perform these tasks, we must register it as a line-modifier function:

```
create_aped_line_modifier ("modifier", &line_emis_modifier, ["a", "b", "c"]);
```

Note that we have provided a name for the function and each of its parameters. These names will appear in the output of `list_par`.

To invoke this helper function, we specify a fit-function using the syntax:

```
fit_fun ("xaped(1, modifier(1))");
```

This syntax is a generalization of the "operator function" syntax (see `set_function_category`). At run-time, the helper function `modifier(1)` is evaluated and returns values that are interpreted by the `xaped` model function. With this model definition, the list of fit parameters looks something like:

```
xaped (1, modifier(1))
 idx  param              tie-to  freeze   value     min     max
  1  modifier(1).a          0     0           0       0       0
  2  modifier(1).b          0     0           0       0       0
  3  modifier(1).c          0     0           0       0       0
  4  xaped(1).norm          0     1           1       0       0
  5  xaped(1).temperature   0     1       1e+07       0       0
  6  xaped(1).density       0     1           1       0       0
  7  xaped(1).vturb         0     1           0       0       0
  8  xaped(1).redshift      0     1           0       0       0
  9  xaped(1).metal_abund   0     1           1       0       0
```

The `modifier` function may require additional user-defined data to perform the necessary computations. One solution to this problem is to store the additional data in global variables, but this is sometimes inconvenient because the implementation of the `modifier` function then depends on symbols with global scope.

An alternative is to have additional user-defined arguments passed to the modifier function when it is called. The advantage to this approach is that the line modifier function can be defined without referring to global variables.

For example, suppose we want to pass 2 additional arguments to the line modifier function. First, write a line modifier function that handles the additional arguments:

```
define my_mod (params, line_id, state, emis, extra1, extra2)
{
   %... perform calculation...
   return emis;
}
```

Then, declare the number of extra arguments (2) in the call to `create_aped_line_modifier`:

```
create_aped_line_modifier ("mod", &my_mod, ["X", "Y"], 2);
```

Now, simply use the extra arguments in the fit-function definition:

```
fit_fun ("xaped (1, my_mod(1, Value1, Value2))");
```

When this fit-function is evaluated, the extra arguments provided will be passed to `my_mod`. In this case, `my_mod` will be called with `extra1=Value1` and `extra2=Value2`.

EXAMPLE 6:

In some applications, it may be useful to introduce a user-defined line profile function. When this user-defined function has adjustable parameters, it may be useful to allow those parameters to vary during a fit.

User-defined line profile functions can be introduced using a helper function in a manner similar to that described in the previous example. Because line profile functions are evaluated a large number of times and are not easily vectorized, these functions are best implemented in a compiled language. A user-defined line profile function can be compiled as a shared library and imported at run-time using the function `load_line_profile_function`.

To register the line profile function as a helper function, use

```
fptr = load_line_profile_function ("example-profile.so", "square");
create_aped_line_profile ("square_profile", &fptr, ["width"]);
```

Note that we have provided a name for the function and each of its parameters. These names will appear in the output of `list_par`.

To invoke this helper function, we specify a fit-function using the syntax:

```
fit_fun ("xaped(1, square_profile(1))");
```

This syntax is a generalization of the "operator function" syntax (see `set_function_category`). At run-time, the helper function `square_profile(1)` is evaluated and returns values that are interpreted by the `xaped` model function. With this model definition, the list of fit parameters looks something like:

```
xaped (1, square_profile(1))
 idx  param                 tie-to  freeze  value     min      max
  1  square_profile(1).width   0     0         0       0        0
```

```
  2  xaped(1).norm              0     1          1       0         0
  3  xaped(1).temperature       0     1      1e+07       0         0
  4  xaped(1).density           0     1          1       0         0
  5  xaped(1).vturb             0     1          0       0         0
  6  xaped(1).redshift          0     1          0       0         0
  7  xaped(1).metal_abund       0     1          1       0         0
```

Note that multiple "helper" functions maybe used simultaneously by providing a comma-separated list of helper functions in the model definition. For example:

```
fit_fun ("xaped(1, modifier(1), square_profile(1))");
```

EXAMPLE 7:

In some applications, it may be useful to introduce a user-defined ionization balance. This can be achieved by using an ionization balance modifier function (see `create_aped_ionpop_modifier`). When this user-defined function has adjustable parameters, it may be useful to allow those parameters to vary during a fit.

User-defined ionization balance functions can be introduced using a helper function in a manner similar to that described in the previous examples. The help function should have the form:

```
define ionpop_modifier (params, state, last_ionpop [, args])
{
   variable n = _isis_max_proton_number+1;
   variable new_ionpop = Float_Type[n,n];

   return new_ionpop;
}
```

To register the ionization balance function as a helper function, use

```
create_aped_ionpop_modifier ("ionpop", &ionpop_modifier, param_names
                  [,num_extra_args]);
```

Note that we have provided a name for the function and each of its parameters. These names will appear in the output of `list_par`.

To invoke this helper function, we specify a fit-function using the syntax:

```
fit_fun ("xaped(1, ionpop(1))");
```

This syntax is a generalization of the "operator function" syntax (see `set_function_category`). At run-time, the helper function `ionpop(1)` is evaluated and returns values that are interpreted by the `xaped` model function. With this model definition, the list of fit parameters looks something like:

```
xaped (1, ionpop(1))
 idx  param                  tie-to  freeze  value     min      max
  1  ionpop(1).time             0     0          0       0         0
  2  xaped(1).norm              0     1          1       0         0
  3  xaped(1).temperature       0     1      1e+07       0         0
  4  xaped(1).density           0     1          1       0         0
```

```
  5  xaped(1).vturb            0     1          0        0         0
  6  xaped(1).redshift         0     1          0        0         0
  7  xaped(1).metal_abund      0     1          1        0         0
```

Note that multiple "helper" functions maybe used simultaneously by providing a comma-separated list of helper functions in the model definition. For example:

```
fit_fun ("xaped(1, ionpop(1), line_profile(1))");
```

EXAMPLE 8:

Models created by `create_aped_fun` can also be evaluated using the `eval_fun2` interface, without using `fit_fun` at all. When one or more "helper" functions are used, the `eval_fun2` arguments associated with each helper can be obtained by calling the corresponding `aped_${helper}_args` function.

For example, the model computed using

```
   fit_fun ("xaped (1, modifier(1))");
   () = eval_counts;
   f = get_model_flux(1).value;
```

is identical to the model computed using:

```
  f = eval_fun2 ("xaped", lo, hi, xaped_pars,
                 aped_line_modifier_args ("modifier", modifier_pars))
```

assuming matching wavelength grids, `lo`, `hi`, and parameter arrays, `xaped_pars` and `modifier_pars`. Any additional arguments to be passed to the modifier should be supplied after the `aped_line_modifier_args` call, e.g.:

```
  f = eval_fun2 ("xaped", lo, hi, xaped_pars,
                 aped_line_modifier_args ("modifier", modifier_pars),
                 mod_arg1, mod_arg2).
```

Similarly, when the newly created function uses a "hook", as described above, e.g.:

```
   create_aped_fun ("xaped", t, &xaped_hook);
```

the `eval_fun2` arguments associated with that hook can be obtained by calling `aped_hook_args`:

```
  f = eval_fun2 ("xaped", lo, hi, xaped_pars,
                 aped_hook_args (&xaped_hook, xaped_instance));
```

where `xaped_instance` is the integer argument expected by `xaped_hook`.

When multiple helper functions are used, all the associated `eval_fun2` arguments are provided in the obvious way. For example, the model computed using:

```
   fit_fun ("xaped (1, modifier(1), square_profile(1))");
```

is identical to the model computed using:

```
f = eval_fun2 ("xaped", lo, hi, xaped_pars,
               aped_line_modifier_args ("modifier", modifier_pars),
               aped_line_profile_args ("square_profile", profile_pars),
               aped_hook_args (&xaped_hook, xaped_instance));
```

assuming `xaped` was defined to take a hook argument and assuming matching grids and function parameters in each case.

aped_ionpop_modifier_args

Purpose: Return extra arguments needed for `eval_fun2` call.
Usage: `args = aped_ionpop_modifier_args (modifier_name, modifier_params[][, num_extra_args])`
See Also: `create_aped_fun, create_aped_ionpop_modifier, eval_fun2`

See `create_aped_fun` for details.

aped_line_modifier_args

Purpose: Return extra arguments needed for `eval_fun2` call.
Usage: `args = aped_line_modifier_args (modifier_name, modifier_params[] [, num_extra_args])`
See Also: `create_aped_fun, create_aped_line_modifier, eval_fun2`

See `create_aped_fun` for details.

aped_line_profile_args

Purpose: Return extra arguments needed for `eval_fun2` call.
Usage: `args = aped_line_profile_args (profile_name, profile_params[])`
See Also: `create_aped_fun, create_aped_line_profile, eval_fun2`

See `create_aped_fun` for details.

aped_hook_args

Purpose: Return extra arguments needed for `eval_fun2` call.
Usage: `args = aped_hook_args (Ref_Type hook, Integer_Type aped_fun_instance)`
See Also: `create_aped_fun, eval_fun2`

See `create_aped_fun` for details.

create_aped_ionpop_modifier

Purpose: Register an ionization balance modifer function as an APED model helper function
Usage: `create_aped_ionpop_modifier (name, Ref_Type [, param_name_array [,num_extra_args]])`
See Also: `aped_line_modifier_args, create_aped_line_profile, fit_fun, eval_fun2`

See `create_aped_fun` for details.

create_aped_line_modifier

Purpose: Register a line-emissivity modifier function as an APED model helper function
Usage: `create_aped_line_modifier (name, Ref_Type [, param_name_array [,num_extra_args]])`
See Also: `aped_line_modifier_args, create_aped_line_profile, fit_fun, eval_fun2`

See `create_aped_fun` for details.

create_aped_line_profile

Purpose: Register a line profile function as an APED model helper function
Usage: `create_aped_line_profile (name, Line_Profile_Type [, param_name_array])`
See Also: `load_line_profile_function, aped_line_profile_args, create_aped_line_modifier, fit_fun, eval_fun2`

See `create_aped_fun` for details.

default_plasma_state

Purpose: Return a structure describing the default CIE plasma state
Usage: `Struct_Type = default_plasma_state ()`
See Also: `create_aped_fun`

This function is provided to simplify using the `define_spectrum` function by returning a template structure describing the default plasma state. The user can modify the fields of the structure before using it, or an array of such structures, as an argument to the `define_spectrum` function.

The struct fields are defined as follows:

```
       norm = [1.0e-14/(4\pi D^2)] \int n_e n_H dV
              where D is the source distance [cm]
temperature = electron temperature (K)
    density = electron density [cm^(-3)]
metal_abund = metal abundance relative to solar
 elem_abund = (optional) array of elemental abundances relative to solar
       elem = (optional) list of element proton numbers;
              size must match the elem_abund array
      vturb = turbulent velocity component [km/s]
   redshift = the redshift
```

To define the elemental abundances, the value of `metal_abund` is first applied to all of the elements heavier than helium. If present, the abundances in the `elem_abund` array are assigned to the corresponding element listed in the `elem` array.

For example:

```
isis> s=default_plasma_state();
isis> print(s);
    norm = 1
    temperature = 1e+07
    density = 1
    metal_abund = 1
    elem_abund = NULL
    elem = NULL
    vturb = 0
    redshift = 0
isis>
```

define_model

Purpose: Define a theoretical spectrum model using S-Lang arrays
Usage: `define_model (state[]);`
See Also: `default_plasma_state`, `model_spectrum`, `append_model`, `mt_def_model`, `create_aped_fun`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`*.*

Analogous to `load_model`, this function defines a multi-component spectrum model using parameters stored in an array of S-Lang structures, `state[]`.

See `default_plasma_state` for the structure field definitions.

For example:

```
% Choose the grid to match a data set
d = get_data_counts (1);

% Use the default plasma state except with
% a different temperature, and metal abundance
% [and with enhanced neon and low iron abundances]
s = default_plasma_state ();

s.metal_abund = 0.7;
s.temperature = 2.4e7;
s.elem = [Fe, Ne];
s.elem_abund = [0.1, 3.0];

define_model (s);
flux = model_spectrum (d.bin_lo, d.bin_hi);
```

edit_model

Purpose: edit current model parameters
Usage: `edit_model (["filename"]);`
See Also: `load_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided *by* `create_aped_fun`*.*

This function uses a text editor to simplify entering and modifying parameters of a multi-component spectral model. If no model is currently specified, it starts the text editor with a blank form containing only a header to indicate the parameter columns (see `load_model` for a detailed description of the input format). If a model is already specified, the model parameters are presented for editing in the format described.

As long as the specified format is adhered to, model components may be added or removed by adding and deleting lines in the parameter file.

The text editor specified by the EDITOR environment variable; if the environment variable is not set, `vi` is used. When using `emacs`, the `emacsclient` feature (of `emacs`) can be used to avoid invoking a new `emacs` process for each edit.

If a filename is specified, the model is saved in that file, otherwise, a temporary file is generated and is deleted when editing is finished. If the TMPDIR environment variable is set, the temporary file created for editing will be placed in the indicated directory. Otherwise, the temporary file will be placed in the current directory.

list_model

Purpose: list parameters defining the current spectral model
Usage: `list_model`
See Also: `edit_model, save_model, load_model, mt_list_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`*.*

This function lists the parameters defining the current spectral model using the format described in `load_model`.

load_line_profile_function

Purpose: load a user-defined line profile function
Usage: `Line_Profile_Type = load_line_profile_function ("library.so", "name")`
See Also: `create_aped_line_profile`

Here is an example implementation of a user-defined line profile function that provides a square line profile:

```
#include <stdlib.h>
#include <math.h>

#include "isis.h"

/* gcc -shared -fPIC -o example-profile.so example-profile.c */

ISIS_LINE_PROFILE_MODULE(square,g,flux,wl,atomic_weight,mid,
                         params,num_params,options)
{
   double width, lo_edge, hi_edge, frac, xl, xh;
```

```
   double *lo, *hi, *val;
   int i, n;

   (void) atomic_weight; (void) num_params; (void) options;

   /* params[0] = \Delta\lambda/\lambda */
   width = params[0] * wl;
   if (width <= 0.0)
     return 0;

   lo_edge = wl - 0.5*width;
   hi_edge = wl + 0.5*width;

   lo = g->bin_lo;
   hi = g->bin_hi;
   val = g->val;
   n = g->nbins;

#define INCREMENT_BIN(i) do { \
        if ((hi_edge < lo[i]) || (hi[i] < lo_edge)) \
          break; \
        xl = (lo_edge < lo[i]  ) ? lo[i] : lo_edge; \
        xh = (  hi[i] < hi_edge) ? hi[i] : hi_edge; \
        frac = (xh - xl) / width; \
        val[i] += flux * frac; } while (0)

   for (i = mid; i >= 0; i--)
     {
        INCREMENT_BIN(i);
     }

   for (i = mid+1; i < n; i++)
     {
        INCREMENT_BIN(i);
     }

   return 0;
}
```

The `ISIS_LINE_PROFILE_MODULE` macro is used both to ensure that the correct interface is implemented and to provide a way for isis to check interface compatibility at run-time.

load_model

Purpose: load model parameters from a file
Usage: `load_model ("file");`
See Also: `edit_model`, `save_model`, `define_model`, `mt_def_model`, `mt_load_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`.

For a collisional ionization equilibrium plasma, the file has this format (numerical values given here are for illustration only):

```
# id   Temp     Density    Abund    Norm    Vturb    redshift
#      (K)      (cm^-3)                     (km/s)
  1    2.e6     1.e-3      1.0      1.0     200.0    0.0
#
# abundances for component 1:
     Si = 0.8   Na = 0.9   Fe = 0.3
#
  2    3.e6     1.e00      0.4      1.5     150.0    0.0
#
# abundances for component 2:
      Si = 0.8   S = 0.9   Ca = 2.1
#
  3    4.e6     1.e12      1.0      1.5     150.0    0.0
```

where

```
    Temp = electron temperature [K]
 Density = electron density [cm^(-3)]
   Abund = metal abundance relative to solar
    Norm = [10^(-14)/(4\pi D^2)] \int n_e n_H dV
           where D is the source distance [cm]
   Vturb = turbulent velocity component [km/s]
redshift = the redshift
```

Lines with a # symbol in column 1 are ignored and may be used for comments. For a more precise definition of `Vturb`, see `use_thermal_profile`.

For each model component, any elemental abundances which differ from the value given in the `Abund` column may be listed separately on lines immediately following the line defining the temperature and density. Individual elements are specified using their 2 character chemical abbreviation; unrecognized abbreviations are ignored. For example, to set the Iron abundance to 0.25 and all the other elements to 0.6, specify `Abund = 0.6` with `Fe = 0.25` on the following input line.

model_spectrum

Purpose: generate a theoretical spectrum model on a given wavelength grid
Usage: `flux = model_spectrum (binlo, binhi [, contrib_flag [, line_list]]);`
See Also: `load_model, edit_model, linear_grid, use_thermal_profile, create_aped_fun, mt_calc_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`.

If present, the `contrib_flag` argument may have one of the following values:

```
___Value________________Meaning________________________
MODEL_LINES_AND_CONTINUUM  model includes lines and continuum
MODEL_LINES                model includes line emission only
MODEL_CONTIN               model includes continuum emission only
MODEL_CONTIN_PSEUDO        model includes pseudo-continuum only
MODEL_CONTIN_TRUE          model includes true continuum only
```

If a `line_list` is specified the computed spectrum includes line emission only from the specified lines.

The model spectrum is generated by loading emissivity data from the spectroscopy database, interpolating to the temperature-density grid specified by the model (see `load_model` and `edit_model`) and computing the emission-measure weighted sum. More precisely, each bin $[\lambda_{\rm lo}, \lambda_{\rm hi})$ in the result has flux

$$\begin{aligned} F(\lambda_{\rm lo}, \lambda_{\rm hi}) &= \sum_m N_m \sum_Z X_m(Z) \sum_{q=0}^{Z-1} X_m(Z, q; T_m, n_m) \times \\ & \sum_k^{N(Z,q)} \varepsilon_{Zqk}(\lambda_{\rm lo}, \lambda_{\rm hi}; T_m, n_m, z_m) \text{ photon cm}^{-2}\text{s}^{-1} \end{aligned} \quad (7.31)$$

where

$$\varepsilon_{Zqk}(\lambda_{\rm lo}, \lambda_{\rm hi}; T, n, z) = \varepsilon_{Zqk}(T, n) \int_{\lambda_{\rm lo}}^{\lambda_{\rm hi}} {\rm d}\lambda \; \phi(\lambda - (1+z)\lambda_{Zqk}). \quad (7.32)$$

In the expression for the flux in each bin, we have summed over model components m, each with normalization $N_m = \int n_e n_H \; {\rm d}V/(4\pi D^2)$, temperature T_m, density n_m, redshift z_m, elemental abundances $X_m(Z)$ and ionization balance fraction $X_m(Z, q; T_m, n_m)$. Each ion q of element Z has $N(Z, q)$ emission lines λ_{Zqk}. The contribution of each emission line to a given spectral bin is determined by integrating the appropriate (Doppler shifted) line profile function $\phi(\lambda)$ over the width of the specified bin. The emissivity values $\varepsilon_{Zqk}(T, n)$ are computed by interpolating the emissivities contained in the spectroscopy database files.

To save the computed spectrum to a file,

```
flux = model_spectrum (lo, hi);
writecol ("spectrum.out", lo, hi, flux);
```

mt_calc_model

Purpose: Compute the spectrum for a given `Model_Type`
Usage: `emis[] = mt_calc_model (Model_Type, lo, hi[, contrib_flag [,line_list]])`
See Also: `mt_def_model, mt_list_model,mt_load_model, mt_save_model,create_aped_fun`

This function is almost identical to `model_spectrum` except that the model is specified by a `Model_Type` instance instead of the internal model component table.

See `model_spectrum` for details.

mt_create_from_struct

Purpose: Create a `Model_Type` object from an appropriate `Struct_Type`
Usage: `Model_Type= mt_create_from_struct (Struct_Type)`
See Also: `mt_def_model, mt_list_model,mt_load_model, mt_save_model, create_aped_fun`

This function is similar to `mt_def_model` except that it works with a single structure whose fields are arrays.

mt_def_model

Purpose: Define a `Model_Type` using an array of structures
Usage: `Model_Type = mt_def_model (Struct_Type[])`
See Also: `mt_create_from_struct, mt_calc_model, mt_list_model,mt_load_model, mt_save_model,create_aped_fun`

This function is almost identical to `define_model` except that the model definition creates a `Model_Type` instance instead of being stored in the internal model component table.

See `define_model` for details.

mt_list_model

Purpose: List components of a `Model_Type`
Usage: `mt_list_model (Model_Type)`
See Also: `mt_calc_model, mt_def_model, mt_load_model, mt_save_model,create_aped_fun`

This function is almost identical to `list_model` except that the model is specified by a `Model_Type` instance instead of the internal model component table.

See `list_model` for details.

mt_load_model

Purpose: Load an ASCII model file into a `Model_Type`
Usage: `Model_Type = mt_load_model (file)`
See Also: `mt_calc_model, mt_def_model, mt_list_model, mt_save_model, create_aped_fun`

This function is almost identical to `load_model` except that the model is loaded into a `Model_Type` instance instead of the internal model component table.

See `load_model` for details.

mt_save_model

Purpose: Save a `Model_Type` in an ASCII model file
Usage: `mt_save_model (Model_Type, file)`
See Also: `mt_calc_model, mt_def_model, mt_list_model, mt_load_model, create_aped_fun`

This function is almost identical to `save_model` except that the model is specified by a `Model_Type` instance instead of the internal model component table.

See `save_model` for details.

save_model

Purpose: save model parameters to a file
Usage: `save_model ("file");`
See Also: `load_model`, `mt_save_model`

This interface is obsolete and is no longer supported. A much more versatile interface is provided by `create_aped_fun`.

The model is saved in an ASCII file using the format described in `load_model`.

use_delta_profile

Purpose: Use a delta-function line profile shape in the theoretical spectrum
Usage: `use_delta_profile`
See Also: `use_thermal_profile`, `create_aped_fun`

When a delta-function line profile shape is specified, all of the flux from a spectral line centered at wavelength λ_0 is added to the single spectral bin k which contains λ_0 ($\lambda_{\rm lo}^k \leq \lambda_0 < \lambda_{\rm hi}^k$).

use_thermal_profile

Purpose: Use a thermal line profile shape in the theoretical spectrum
Usage: `use_thermal_profile`
See Also: `use_delta_profile`, `create_aped_fun`

When a thermal line profile shape is specified, the contribution to each spectral bin $[\lambda_{\rm lo}, \lambda_{\rm hi})$ from a spectral line with intensity I_0 centered at wavelength λ_0 is

$$\mathrm{I}(\lambda_{\rm lo}, \lambda_{\rm hi}) = \frac{I_0}{\sigma\sqrt{2\pi}} \int_{\lambda_{\rm lo}}^{\lambda_{\rm hi}} \mathrm{d}\lambda \; \exp\left[-\frac{(\lambda-\lambda_0)^2}{2\sigma^2}\right] \tag{7.33}$$

where

$$\sigma \equiv \frac{\lambda_0}{c}\left(\frac{1}{2}v_{\rm turb}^2 + \frac{kT}{m(Z)}\right)^{1/2} \tag{7.34}$$

T is the temperature and $m(Z)$ is the atomic mass of the emitting ion (which has proton number Z). $v_{\rm turb}$ is the RMS turbulent velocity width in the emitting plasma.

7.6 Generic Plot Functions

ISIS provides common plotting functionality through intrinsic functions which try to relieve some of the burden of selecting plot ranges, colors, axis labeling, etc. These intrinsic functions also provide access to most selected plot customization features (colors, etc.) with additional support to track details of multiple plots which may be open simultaneously.

A plot device can be opened using `open_plot`, `rplot_counts`, `multiplot` or one of the other `*plot*` functions such as `plot_data_counts` or `hplot`. Several plot devices are supported (see the PGPLOT documentation for details). The default plot device may be specified using either the PGPLOT_DEV environment variable or the `plot_device()` function. If no default plot device is specified, the user will be prompted to select a device. Note that some plot devices have limited functionality; some devices do not support color, others do not support selective erasure of sub-regions of the plot window, etc. See the PGPLOT documentation for information on the supported plot devices and their capabilities.

ISIS supports use of multiple, simultaneous plot windows and devices and multiple panes per plot window. If a plot window contains multiple panes, the plot focus automatically moves to the next pane each time a new plot is created. Each plot window maintains its own format parameters, initialized using generic default values *once* when the plot device is opened. These format parameters and some of the commands used to set them are:

```
______Function(s)_________________Purpose____________
     [xy]range, limits     Set axis ranges
            xlin, xlog     Set axis type (log or linear)
                 label     Set axis labels
                 title     Set plot title
 color, set_data_color     Set plot line color
             linestyle     Set line style (solid, dashed, etc)
            pointstyle     Set point style (dot, circle, etc)
        connect_points     Set point connect style
                           (points only or points + line)
             plot_unit     Select histogram plot x-axis units
             errorbars     Toggle plotting of histogram errorbars
     plot_bin_density,     Select histogram plot type
     plot_bin_integral
```

All format parameter values are retained for the life of the associated plot window.

Axis ranges in a given plot default to display the full range of the first data set plotted. Axis ranges may be set explicitly using the `xrange` and `yrange` functions; the `limits` command forces the next plot to display the full range of the data.

The size and shape of the plot can be controlled either by using using the `resize` function or, in X-windows, by interactively resizing the plot window as you would any other window. Plot annotations are set using `xylabel`. Plot text appears in Roman font by default. Within a text string, the font can be changed (temporarily) by using the escape sequences `\fn` ("normal"), `\fr` (roman), `\fi` (italic), and `\fs` (script). Additionally, plot annotations may be written in a convenient LaTeX-style format (see `latex2pg`)

Histogram data from the internal list of data-sets is plotted using the functions `[o]plot_data_*` and `[o]plot_model_*` described in §7.2 (see also `rplot_counts`). By default, over-plots automatically change the color or line-style. The user can over-ride this feature by specifying the plot style explicitly or by disabling the feature using `plot_auto_color`. Data-sets can be plotted with or without errorbars (see `errorbars`. `[o]hplot` provides an alternate interface for plotting histograms defined by a set of S-Lang variables.

Arrays of points may be plotted with or without connecting lines and with or without different symbols for individual data points (see `[o]plot` and `connect_points`). Images may be plotted using `plot_image`.

For low-level plotting functionality not otherwise provided by ISIS intrinsics, direct interactive access to most of the PGPLOT subroutine library is available (see §7.9).

The default plot format may be controlled by modifying fields of the intrinsic structure `_isis_plot` defined in Table 7.1: For example, to change the plot defaults to have spectra plotted as bin-density values in keV energy units, add the lines

Table 7.1: ISIS Default Plot Format

Field Name	Default Value	Definition
x_unit	U_ANGSTROM	Energy or wavelength unit used for plotting spectra; supported values are U_ANGSTROM, U_NANOMETER, U_KEV, U_EV, U_HZ
use_bin_density	0	(boolean) 0/1 means spectra are plotted as bin-integral/bin-density values
use_errorbars	0	0 means errorbars off, N means plot every Nth errorbar
logx/logy	0	(boolean) 0/1 means linear/log axis
pointstyle	-1	pointstyle; see `pointstyle()`
linestyle	1	line style; see `linestyle()`
color	1	line color; see `color()`
char_height	1	character height
ebar_term_length	0	length of errorbar terminals

```
_isis_plot.use_bin_density = 1;
_isis_plot.x_unit = U_KEV;
```

to your `.isisrc` file so that your selected plot format defaults will be automatically applied when ISIS starts.

charsize

Purpose: change the character size used in plot labels
Usage: `charsize (size)`
See Also: `color, label, xylabel`

`size` is the normalized character height (1.0 is the default).

Because of an apparent bug in the PGPLOT library, setting the character size to a value significantly larger than 1.0 may cause the plot axis labels to fall off the edge of the plot. To workaround this bug, adjust the plot viewport to create wider plot borders:

```
% The plot device should be open before the viewport
% size is changed
() = open_plot (device);
```

```
% Make the viewport somewhat smaller than the default
% The numbers give xmin, xmax, ymin, ymax in normalized
% device coordinates which range from 0-1.
variable v = get_outer_viewport();
v.xmin = 0.2;
v.xmax = 0.8;
v.ymin = 0.2;
v.ymax = 0.8;
set_outer_viewport(v);

% Now create a plot using larger characters:
charsize(2);
plot_data_counts(1);
```

close_plot

Purpose: Close a plot window
Usage: `close_plot[(window_id)]`
See Also: `open_plot, window`

The window index is the value returned when a new plot device is opened with `open_plot`; also, in X-windows, each window frame is normally labeled with the window index. If window_id is absent, the currently active plot window is closed. Note that when generating a Postscript file, the output file must be explicitly closed in order to generate output, otherwise an empty file may be generated.

color

Purpose: change the plot color
Usage: `color (color_index)`
See Also: `line_or_color`

With the default 16-color configuration of PGPLOT, the color indices correspond to `red=2, green=3, blue=4, light_blue=5, purple=6, yellow=7, orange=8, grey=15`. This mapping may be redefined using functions in the supplied PGPLOT module; see the PGPLOT documentation for further details.

connect_points

Purpose: Specify whether or not data points are connected by a line
Usage: `connect_points (flag)`
See Also: `pointstyle, linestyle`

By default, data points are connected by a line (`flag` = 1; set `flag` = 0 to omit the connecting line; set `flag` = -1 to omit the points and draw the line only. The default is a solid line; an alternate line style may be selected using `linestyle`.

cursor

Purpose: read plot coordinates from the cursor
Usage: `cursor( [ &x, &y [, &ch]])`
See Also: `xylabel`

With no arguments present, the cursor coordinates are printed to the screen each time the mouse cursor is clicked in the plot window. Press 'q' to quit.

The cursor movement keys may be used for finer positioning control.

The coordinates of the first mouse click or keypress and may also be returned in variables specified as function arguments. Because the *addresses* of the variables x,y and ch are used, it is necessary to declare the variables before using them with the `cursor` function, even in interactive mode.

Caveat: *This function does not work as expected if the plot window is divided into sub-panes.*

```
cursor;        % print cursor coordinates on each mouse-click

variable x, y, ch;             % declare variables;
cursor (&x, &y);               % assign x, y the coordinates of
                               %     each mouse-click.
```

cursor_box

Purpose: read corners of a box from plot cursor
Usage: `(xmin, xmax, ymin, ymax) = cursor_box ();`
See Also: `cursor`

This function prompts the user to draw a box with the mouse cursor in a currently open plot window. The box is be oriented so that its sides are parallel to the edges of the plot window. The x-y coordinates of the box corners are returned.

dup_plot

Purpose: open a new plot device using parameters copied from an existing plot
Usage: `new_id = dup_plot (["device" [,window_id]])`
See Also: `open_plot, window`

This function initializes a new plot device with formatting parameters taken from an existing plot (e.g. number of plot panes, axis ranges, character size, line style, etc.). If no plot device is specified, a default device is used or prompted for (see `open_plot` for details). If no plot window is specified (`window_id`) , the parameters of the current plot device are duplicated. This is sometimes useful in generating a hardcopy version of a screen plot which required extensive formatting.

erase

Purpose: erase a plot window
Usage: `erase[(window_id)]`
See Also: `open_plot, window`

This function is also aliased to `clear`. The plot window to erase is specified by giving the integer index of the plot window. The window index is the value returned when a new plot device is opened with `open_plot`; also, in X-windows, each window frame is normally labeled with the window index. If window_id is absent, the active pane of the currently active plot window is erased. If the current device does not support selective erasure of a sub-pane, the entire window will be erased (probably - it depends on the device).

errorbars

Purpose: Specify the plotting of histogram errorbars
Usage: `errorbars (N [,length])`
See Also: `plot_data_counts`

Error bars are not plotted by default ($\mathtt{N} = 0$); to plot every Nth errorbar, set `N > 0`. The optional second argument controls the length of errorbar terminals. The default is 0.0 (no terminals); a value of `length=1.0` gives reasonably sized terminal bars. Currently, error bars are available only on histogram plots; the PGPLOT module functions may be used to get error bars on other kinds of plots.

get_outer_viewport

Purpose: Get current viewport location
Usage: `Struct_Type = get_outer_viewport ()`
See Also: `set_outer_viewport, multiplot`

Use this function to get the current location of the outer viewport in normalized device coordinates. See `set_outer_viewport` for details.

get_plot_info

Purpose: Get information about the current plot format
Usage: `Struct_Type = get_plot_info ()`
See Also: `plot_open, plot_close, get_plot_options, set_plot_options`

This function returns a structure of the form

```
variable p = struct
  {
     xmin, xmax, xlog,
     ymin, ymax, ylog,
     line_width,
     line_color
  };
```

which provides the current plot limits (`xmin, xmax`, `ymin, ymax`) and the current line width and line color. The boolean values `xlog` and `ylog` are non-zero if the corresponding axis is log-scale.

If no plot device is open, the function returns NULL.

get_plot_options

Purpose: Get information about the current plot format
Usage: `Struct_Type = get_plot_options ()`
See Also: `plot_open, plot_close, get_plot_info, set_plot_options`

Most plot options that are used internally by isis are returned as fields of a structure.

```
isis> p = get_plot_options;
isis> print(p);
    xmin = -3.40282e+38         % axis ranges
    xmax = 3.40282e+38
    ymin = -3.40282e+38
    ymax = 3.40282e+38
    xlabel =                    % axis labels
    ylabel =
    tlabel =                    % title
    xopt = BCNST                % axis label styles
    yopt = BCNST
    logx = 0                    % boolean
    logy = 0
    color = 1
    line_style = 1
    line_width = 1
    frame_line_width = 1
    point_style = -1
    connect_points = 1         % boolean
    char_height = 1
    point_size = 1
    ebar_term_length = 0       % length of error-bar terminator
    use_errorbars = 0          % boolean
    use_bin_density = 0        % boolean
```

hplot

Purpose: [o]plot a histogram defined by S-Lang arrays
Usage: `[o]hplot (Struct_Type | binlo, binhi, value [,line_style])`
See Also: `plot, plot_data_counts, window`

This function plots or over-plots a histogram described by three 1-D S-Lang arrays of size N.

```
Example:
       edit_model;               % specify spectrum model parameters

       (lo, hi) = linear_grid (1.0, 20.0, 6000);       % generate a grid
       flux = model_spectrum (lo, hi);                 % compute bin values
```

```
hplot (lo,hi,flux);                              % plot
```

When a single `Struct_Type` argument is given, it should have fields `bin_lo`, `bin_hi`, `value`. For example:

```
d = get_data_counts (1);
hplot (d);
```

label

Purpose: label plot axes
Usage: `label ("xlabel", "ylabel", "title")`
See Also: `xlabel, ylabel, title, xylabel, latex2pg`

Specifies text strings to be used for the plot title and axis labels. To add a plot title only, use `title`.

latex2pg

Purpose: Translate strings from LaTeXformat to PGPLOT format
Usage: `String_Type = latex2pg (String_Type)`
See Also: `xylabel, xlabel, ylabel, title, label`

This function simplifies creating plot annotations which contain Greek symbols, superscripts and subscripts. It translates strings written in a convenient LaTeX-style format into a format which is understood by PGPLOT. For example:

```
____LaTeX_format_____      ____PGPLOT_format____
 1.24 \\x 10^{-14}          1.24 \x 10\u-14\d
 \\A^{-1}                   \A\u-1\d
 2s^^2S_{1/2}               2s \u2\dS\d1/2\u
 e^{i\\pi} + 1 = 0          e\ui\gp\d + 1 = 0
```

Note that two backslash '\' symbols must be used to denote font changes, Greek symbols and other special symbols.

limits

Purpose: Use the plot data to determine axis limits for the next plot
Usage: `limits`
See Also: `xrange, yrange`

This function re-sets the current pane of the current plot window so that the XY ranges of the next plot will cover the full-range of data. This function is equivalent to `xrange; yrange;`

line_or_color

Purpose: control interpretation of line style indices
Usage: `line_or_color (choice)`
See Also: `color, linestyle`

If `choice =0`, line style indices indicate line *type* (e.g. solid, dashed, dotted, etc.). If `choice !=0`, line style indices indicate line *color* (this is the default).

linestyle

Purpose: change the plot linestyle
Usage: `linestyle (linestyle_index)`
See Also: `line_or_color`

The available line-style indices are (1) full line, (2) dashed, (3) dot-dash, (4) dotted, (5) dash-dot-dot-dot. See the PGPLOT documentation for more information.

multiplot

Purpose: Subdivide the current plot-device window into horizontal panes
Usage: `multiplot (relative_sizes)`
See Also: `open_plot, mpane, window`

This function supports subdividing a plot window into horizontal panes which share a common X-axis. The argument to `multiplot` is an integer array which gives the relative sizes of the plot panes in order, from top to bottom.

Example:

```
multiplot ([3,1]);   % top pane is 3x as wide as the bottom pane
multiplot (1);       % Restore subdivided plot to a single pane.
```

mpane

Purpose: Select a specific pane in a multiplot
Usage: `mpane (pid)`
See Also: `multiplot`

This function selects a specific pane within a multiplot (a set of plot panes sharing a common X-axis).

Example:

```
x=[1:100];
multiplot ([1,1,2]);   % a 3-pane multiplot
plot(x,x);             % plot in pane 1 (the top one)
plot(x,x);             % plot in pane 2
```

```
mpane(1);              % move back to the top pane
oplot(x,x^2);          % over-plot in pane 1

mpane(3);              % move to the bottom pane
plot(x,x);
oplot(x,x^2-x);
multiplot(1);          % revert to a single pane format
```

open_plot

Purpose: open a new plot window
Usage: `window_id = open_plot (["device" [, nxpanes, nypanes]])`
See Also: `close_plot, window, dup_plot, plot, hplot, plot_data, plot_model, plot_elev, multiplot`

The first argument specifies the name of plot device. If device = "?" or is blank or absent, the user is prompted to supply the device name. The default device is that specified by the `PGPLOT_DEV` environment variable; if that environment variable is not set, the default device is `NULL`. See the description of the PGPLOT command `pgopen` for full details on the device specification string.

The remaining two optional arguments (`nxpanes`, `nypanes`) give the number of X and Y subdivisions for the window. The return value gives the integer index of plot device just opened.

Postscript:
A good way to generate postscript plots is to write a S-Lang script using an `open_plot` command which specifies the output postscript file name and the postscript device using the device string (see the examples below).

GIF and other formats:
If the appropriate drivers are included when PGPLOT is installed, plots may be generated in a wide range of formats. For example, to generate a GIF image, open the appropriate plot device (e.g. `open_plot("plot.gif/gif")`), generate the plot as usual and then close the plot device (`close_plot`).

X-windows:
Under X-windows, the default initial size of the plot window is controlled by the Xresource

```
pgxwin.Win.geometry.
```

If ISIS is run on a machine with a small screen, the default plot window size should probably be set to something smaller than the PGPLOT default size of 867 × 669 pixels. Otherwise, when a plot window is opened, it might fill the entire screen. To make the default window half as large, insert the line

```
pgxwin.Win.geometry:      432x333
```

in `~/.Xdefaults`, then use

```
xrdb ~/.Xdefaults
```

to have the X window manager read the modified .Xdefaults file. It may also be necessary to kill and re-start the existing pgxwin server (e.g. by re-opening the plot window) before the geometry change will take effect.

Another useful Xresource is `pgxwin.Win.maxColors`. Because each plot window reserves 100 colors, opening several plot windows may quickly exhaust the available colors on the display, leading to color flashing or causing the entire screen to turn black upon opening a plot window. This problem may be alleviated by setting `pgxwin.Win.maxColors` to a smaller value, e.g.

```
pgxwin.Win.maxColors:  16
```

See the PGPLOT documentation for full details:

```
http://astro.caltech.edu/~tjp/pgplot/xwdriv.html
```

If the display turns black because it ran out of colors, one can usually recover by moving the mouse cursor to a different window and perhaps clicking the mouse in that window (if the X-window configuration uses click-to-focus). Another symptom of running out of colors is the notorious color-flashing which occurs as the mouse cursor moves across window boundaries. These problems are sometimes less severe with newer systems that have larger color-spaces (e.g. 24 bit monitors).

```
Examples:

% a single pane X-window
() = open_plot;              % assuming PGPLOT_DEV = "/xwin"

% a color postscript plot with 2 panes in a file "myfile.ps"
() = open_plot("myfile.ps/cps",2,1);

% a different style of plot with 2 sub-panes
% the top pane is 3X as wide as the bottom pane
() = open_plot ("/xw");
multiplot ([3,1])

% A function like this can be used to simplify plotting
% data and residuals in the XSPEC style:
% [the ISIS function rplot_counts() does this]

public define plot_count_residuals (h, dev)
{
   multiplot ([3,1]);

   errorbars (1);
   plot_data_counts (h);
   oplot_model_counts (h);

   errorbars (0);
   yrange;
   _rplot_counts(h);
}
```

```
%  e.g. To use this function to plot data set 1 in an X-window
%  (see the figure) type

   isis> plot_count_residuals (1, "/xw");
```

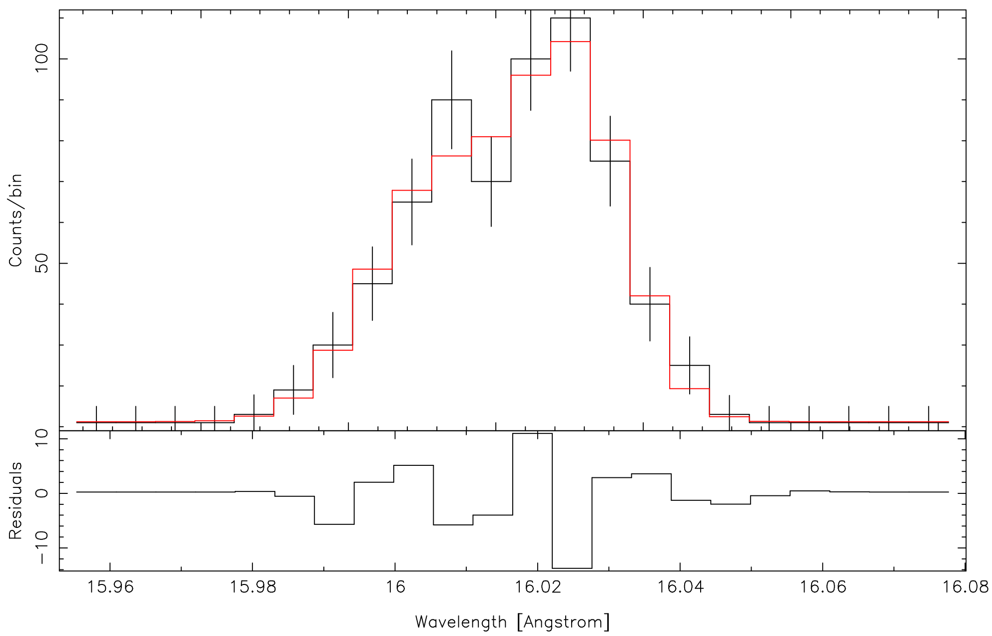

Figure 7.1: Example of plotting count data and fit-residuals.

plot

Purpose: [o]plot x-array vs. y-array
Usage: `[o]plot (x, y, [line_style | symbol_array])`
See Also: `open_plot, window, pointstyle, connect_points`

Given two 1-D S-Lang arrays of size N, this function will plot or over-plot x vs. y. The oplot function behaves like plot if no plots have already been drawn. If the third argument is a single integer, it is interpreted as the the line-style `see line_or_color`. If the third argument is an integer array of length N, it is interpreted as a list of plot symbols, one for each point.

```
% Example:           plot the line y=x for 1 <= x <= 10, labeling the points
%                    with the lower case letters 'a' through 'j'
      x = [1:10];
      y = x;
      sym = ['a':'j'];          % sym[0] = integer ASCII code for 'a', etc.
      plot (x,y,sym);
```

plot_auto_color

Purpose: Control automatic line-color/line-style changes
Usage: `plot_auto_color (flag)`
See Also: `set_data_color, line_or_color`

If `flag` is non-zero, overplots automatically change line color or line style, cycling through the available colors or styles (this is the default). If `flag` is zero, color or style changes must be specified explicitly.

plot_bin_density

Purpose: Plot data values as a density in each bin
Usage: `plot_bin_density`
See Also: `plot_bin_integral`

This function alters the plotted values for histogram plots only. By default, histograms are plotted as bin-integral quantities. This mode specifies that the input data bin values should be divided by the bin width and exposure time for plotting (recall that the input data values are assumed to be bin-integral quantities)

plot_bin_integral

Purpose: Plot data values integrated over each bin width
Usage: `plot_bin_integral`
See Also: `plot_bin_density`

This function alters the plotted values for histogram plots only. If the format type is not specified, it is assumed to be bin-integral. Because the input data values are assumed to be bin-integral quantities, this mode specifies that the input data bin-values are to be plotted unchanged.

plot_contour

Purpose: display image contours
Usage: `plot_contour (img[] [, aspect [,x[], y[] [, c[] ]]])`
See Also: `plot_image, histogram2d`

The image is a 2D array, `img[y,x]`. When the optional argument, `aspect`, is non-zero, the image dimensions determine the aspect-ratio of the plot. Otherwise, the plot fills the plot window and the image may be distorted. The optional 1D array arguments `x,y` provide the coordinate grid for the X,Y axes. The remaining optional argument, `c[]`, provides the contour levels to be plotted.

plot_device

Purpose: specify default plot device
Usage: `plot_device (device_string)`
See Also: `open_plot, window`

This default over-rides the current value of the PGPLOT_DEV environment variable; `device_string` can be any valid PGPLOT device string; the list of available devices is determined by the configuration of the PGPLOT installation. See the PGPLOT documentation for details.

plot_image

Purpose: display an image
Usage: `plot_image (img[][, aspect [,x[], y[] [, amin, amax]]])`
See Also: `plot_contour, histogram2d, set_palette, plot_image_ctrl`

The image is a 2D array, `img[y,x]`. When the optional argument, `aspect`, is non-zero, the image dimensions determine the aspect-ratio of the plot. Otherwise, the plot fills the plot window and the image may be distorted. The optional 1D array arguments `x,y` provide the coordinate grid for the X,Y axes. The two remaining optional arguments, `amin, amax`, specify the range of array values to be mapped onto the chosen color table. Use `set_palette()` to select from a set of predefined color tables; `plot_image_ctrl` supports interactive control over the color table, brightness and contrast.

plot_image_ctrl

Purpose: Adjust image color table, brightness and contrast
Usage: `plot_image_ctrl ()`
See Also: `histogram2d, set_palette, plot_image`

This function supports interactive control over the color table, brightness and contrast used to display an image on the current plot device.

plot_quit

Purpose: close all plot windows
Usage: `plot_quit`
See Also: `open_plot, close_plot, window`

This function closes all currently open plot devices

plot_unit

Purpose: specify X-coordinate units for plotting data histograms
Usage: `plot_unit (x_unit)`
See Also: `plot_data, plot_bin_density`

The default X-coordinate units are Ångstrom units. Supported alternatives are selected by specifying one of the following strings (case-insensitive):

```
Angstrom, A, nm, mm, cm, m
eV, keV, MeV, GeV, TeV,
Hz, kHz, MHz, GHz
```

Note that altering the X-coordinate of the plot will also change the Y-values plotted if the Y-axis is in units of bin-density (because those values are being divided by the bin-width). If plotting bin-integral values, the Y-axis values are independent of the X-coordinate.

pointstyle

Purpose: change the symbol used to plot data points
Usage: `pointstyle (symbol)`
See Also: `[o]plot`

```
-1,-2     a single dot
-3..-31   a regular polygon with abs(symbol)
          edges (style set by current fill style)
0..31     standard marker symbols
32..127   ASCII characters (in current font)
   >127   a Hershey symbol number
```

For further details, including lists of the standard marker symbols and the Hershey plot symbols, see the PGPLOT documentation. Note that the ASCII value of a character is easily obtained in S-Lang:

```
ch = 'b';                              % set ch to the ASCII code for 'b'
sym = Integer_Type [length(array)];    % make a symbol array
sym[*] = 'Q';                          % and set all symbols to 'Q'
```

resize

Purpose: resize the plot window
Usage: `resize[ ([width_in_cm], [aspect_ratio])]`
See Also: `open_plot, window`

The aspect_ratio is defined as height/width. aspect=1.000 gives a square plot
aspect=0.618 gives a horizontal rectangle
aspect=1.618 gives a vertical rectangle

If width=0.0, the window will assume the largest available view surface consistent with the argument aspect_ratio. If no arguments are given, the result is equivalent to `resize (0.0, 0.618)`. Note that the resize takes effect only when something is plotted in the specified window and not at the moment the resize function is called.

set_frame_line_width

Purpose: Set line width for plot axes and axis grid labels
Usage: `set_frame_line_width (width)`
See Also: `set_line_width`

Use this function to set the line width used to draw the plot axes, the tick-mark labels, the axis labels and the plot title. By default, `width=1`.

set_line_width

Purpose: Set line width
Usage: `set_line_width (width)`
See Also: `set_frame_line_width`

Use this function to set the line width used to plot points, curves and error-bars. By default, `width=1`.

set_outer_viewport

Purpose: Set viewport location
Usage: `set_outer_viewport (Struct_Type v)`
See Also: `get_outer_viewport, multiplot`

Use this function to specify the outer viewport of the plot in normalized device coordinates. The coordinates are specified using a structure of the form

```
v = struct {xmin, xmax, ymin, ymax}
```

For a simple plot, the "outer viewport" has the same meaning as the viewport discussed in the PGPLOT documentation for e.g. PGSVP. For a multiplot containing several plot panels sharing a common X-axis, the "outer viewport" is the outermost box that contains all of the plot panels.

set_palette

Purpose: Select image color table
Usage: `set_palette (id)`
See Also: `plot_image, plot_image_ctrl`

Use this function to choose from five predefined color tables:

```
___id______Color_Table___
    1       grey scale
    2       rainbow
    3       heat
    4       iraf
    5       aips
```

set_plot_options

Purpose: Modify the current plot format
Usage: `set_plot_options (Struct_Type)`
See Also: `plot_open, plot_close, get_plot_info, get_plot_options`

title

Purpose: Set plot title
Usage: `title ("title string")`
See Also: `xlabel, ylabel, label, xylabel, latex2pg`

Specifies a string to use as the plot title. To simultaneously label both axes and provide a title use the `label` function.

window

Purpose: specify the active plot device
Usage: `window (window_id);`
See Also: `open_plot`

Plotting commands refer to the active plot device. Use this function to specify which device is active. This function is useful primarily during an interactive session for switching between different plot windows within X-windows. This functionality is not necessarily available with all plot devices or all combinations of plot devices.

xinterval

Purpose: read endpoints of x-interval from plot cursor
Usage: `(xmin, xmax) = xinterval ();`
See Also: `cursor`

This function is useful for selecting X-intervals from a plot in interactive mode. `yinterval` is the analogous command for the y-axis.

xlabel

Purpose: Set plot x-axis label
Usage: `xlabel ("string")`
See Also: `ylabel, label, title, xylabel, latex2pg`

Specifies a string to use as the X-axis label. To simultaneously label both axes and provide a title use the `label` function.

xlin

Purpose: Change the x-axis to linear scale
Usage: `xlin`
See Also: `xlog`

`ylin` is the analogous command for the y-axis

xlog

Purpose: Change the x-axis to log scale
Usage: `xlog`
See Also: `xlin`

`ylog` is the analogous command for the y-axis When log scale is specified for a given coordinate axis, points with non-positive values of that coordinate are ignored when computing the full range of the data. For example, consider a function defined on the set of x coordinates [-5, -3, -2, 0, 2, 4, 5, 10, 15]. When plotted using a log x-axis, the full range of data is considered to extend from log10(2) to log10(15); the non-positive coordinate points are ignored. This command affects the plot appearance only; it does *not* replace the internal data values with their logarithms.

xrange

Purpose: set the X-axis plot limits for the next plot
Usage: `xrange[ ([xmin], [xmax])]`
See Also: `xlin, xlog`

If either argument is absent, the corresponding value will be derived from the data when the next plot is generated. `yrange` is the analogous command for the Y-axis, except that `yrange` sets the Y-axis plot limits to the range of data inside the current X-axis plot limits rather than the full range of the entire data set.

xylabel

Purpose: place a text label at an arbitrary x,y plot location
Usage: `xylabel (x,y,label, [angle [,justify]])`
See Also: `xlabel, ylabel, title, label, latex2pg, cursor`

The optional `angle` argument gives the rotation angle in degrees CCW [default = 0.0]. The optional `justify` argument positions the text relative to the cursor location; `justify`=0.0 for text to the right of the cursor [default], `justify`=0.5 for text centered on cursor, `justify`=1.0 for text to the left of the cursor.

In interactive mode, it may be helpful to read the (x,y) coordinates with the mouse via the `cursor` command.

```
Example:
    variable x,y;
    cursor (&x, &y);
    xylabel (x, y, "Fe XVII");
```

A more general S-Lang script which uses the `cursor` command to select points and then writes appropriate `xylabel()` commands to a file (with explicit X-Y coordinates filled in) might greatly simplify some tedious plot annotation tasks, such as labeling emission lines in an observed spectrum.

7.7 Fitting Functions to Data

To use high resolution spectral data to infer physical conditions in an emitting plasma, measurements of the strength and shape of observed features are often compared to theoretical predictions. To facilitate measurement of the position, strength, size and shape of spectral features, ISIS supports fitting models to data. A few simple functions such as Gaussians and polynomials are provided by ISIS and many others are available from the XSPEC module (§8). ISIS also provides the ability to automatically generate customized multi-component spectral models using the spectroscopy database (see `create_aped_fun`). Multiple user-defined fit-functions are also supported (see `add_compiled_function`, `add_slang_function`). Furthermore, using the ISIS intrinsic variable `Isis_Active_Dataset`, one can write user-defined functions which evaluate differently for different datasets. For example, its possible to simulataneously fit a Gaussian to one dataset and a Lorentzian to another. For another way to do this, see `assign_model`.

Models may be fit to either flux-corrected data or count-data. Extending the standard forward-folding fit algorithm used in XSPEC, the ISIS implementation supports the treatment of some classes of non-linear instrument response (such as event pileup – see §4.11) and also supports application of a user-defined photon redistribution function (RMF) implmemented in software (see `load_rmf` and, for an example implementation, see `src/rmf_delta.c` in the ISIS distribution).

The ISIS implementation of forward-folding "solves" equations of the form

$$C(h) = B(h) + t \int \mathrm{d}E \ \mathcal{F}\Big(R(h,E),\ A(E),\ s(E)\Big) \tag{7.35}$$

where $C(h)$ is the number of counts obtained in detector-bin h, t is the effective exposure time, E is the incident photon energy, and $s(E)$ is the incident source flux at energy E. The instrument response is represented by an effective-area function $A(E)$ (e.g. the ARF) and a redistribution function $R(h,E)$ (e.g. the RMF). The instrumental background in detector-bin h is given by $B(h)$.

All the functions on the right-hand side of equation (7.35) may be user-defined. In particular, the function $\mathcal{F}$ may be user-defined; for brevity, we refer to this function as the *kernel*, although this usage is somewhat inconsistent with standard mathematical terminology. In the standard implementation of forward-folding,

$$\mathcal{F}(R,\ A,\ s) = R(h,E)A(E)s(E). \tag{7.36}$$

This is the form used in XSPEC and is the default kernel used in ISIS . When multiple responses are assigned to a single data set (using `assign_rsp`), ISIS uses

$$\mathcal{F}(R,\ A,\ s) = \sum_i R_i(h,E)A_i(E)s(E), \tag{7.37}$$

where the sum extends over all responses. This latter form is suitable for analysis of HRC/LETGS data, where multiple dispersion orders are summed together in the data, but where responses for individual dispersion orders are available. This form is also applicable to the analysis of CCD data where significant response variation is present across the spectral extraction region (e.g. for ACIS-I observations of extended sources); in this case, the ARFs must be weighted according to the surface-brightness distribution of the source. Davis (2001) showed that event pileup in CCD detectors is a non-linear process which can be described by a considerably more complex function $\mathcal{F}$. Davis's implementation of this pileup kernel is available in ISIS and may be accessed using the `set_kernel` and `print_kernel` functions (see §4.11). Similarly, user-defined fit-kernels may be accessed using the `load_kernel` function.

`ignore`, `notice` and `xnotice` are used to indicate which data bins are to be included in the fit. `ifit_fun` or `fit_fun` define and modify the form of the fit function. `list_par` displays a list of current fit parameters. Parameter values can be set using `edit_par`, `set_par` or `load_par`; allowed ranges for each parameter value are enforced during the fitting iteration. The `renorm_*` functions simplify finding the initial parameter values by automatically adjusting the normalization of the model. Parameters may be fixed at known values or allowed to vary to improve the model fit (see `freeze, thaw`). Parameter values may also be tied to together (see `tie, untie`). Parameter values may even be defined as functions of other parameters and, optionally, as functions of user-defined functions (see `set_par_fun`).

The `eval_*` functions evaluate the model using the current parameter set. Once the fit data is specified and the model and initial parameter values are defined, the `fit_*` functions compute the best fit model parameters and `save_par` saves the parameter values to an ASCII file. `[o]plot_model` displays plots of the best fit model. Single parameter confidence limits can be computed using `[v]conf`.

`get_cfun` and `get_cfun2` evaluate the underlying differential (unbinned) fit function at a specified set of points; these values may then be plotted using `[o]plot` (See §7.6).

While the bin coordinates in the input data may use any of several supported physical units, all fit parameters are defined using Angstrom bin coordinates and *all internal calculations are done in Angstrom bin coordinates*. Function values are interpreted as the integral, over the bin width, of an underlying "differential" function.

The intrinsic variable `Fit_Verbose` controls the amount of information printed to the screen during the fit iterations. In verbose mode (`Fit_Verbose` > 0), the fit routine prints out the values of the variable parameters at each iteration along with the current χ^2 value and convergence tolerance. In silent mode (`Fit_Verbose` < 0), only fatal error messages are printed. The default verbose level is `Fit_Verbose`=0, but this may be set in the `$HOME/.isisrc` configuration file.

By default, the instrumental response (RMF/ARF) is applied when fitting, if these functions are available.

Functions may be fitted to "scatter data" [randomly ordered sets of (X,Y) pairs] using the `array_fit` function.

_par

Purpose: get the value of a fit parameter
Usage: `par = _par (idx)`
See Also: `get_par, set_par, set_par_fun`

This function is identical to `get_par` but is normally used only when defining fit-parameters as functions of other parameters.

add_compiled_function

Purpose: Add a user-defined fit function
Usage: `add_compiled_function (library_name, function_name [, option_string])`
See Also: `del_function, list_functions, add_slang_function, set_function_category`

Note: this function is available only on systems with ELF support. See the `userfun` example from the ISIS web page and the `modules/xspec/src` directories in the ISIS distribution for detailed examples of how to implement user-defined fit functions in C; S-Lang fit-functions are also supported (See `add_slang_function`). An unlimited number of user-defined functions are supported and fit functions may have an unlimited number of parameters.

The first argument, `library_name`, gives the name of the shared library file (.so). The section argument, `function_name`, gives the name of the subroutine that evaluates the fit-function (string `<31` characters). The optional third argument, `option_string`, is a string which is passed to the user-module initialization routine. To avoid future problems, parameter names should avoid using S-Lang special characters.

For example, suppose the user-defined fit-function is a power law ($f(x) = Ax^p$) and is defined by a subroutine called `plaw`, with parameters `norm` and `power`). Further, suppose this subroutine is contained in a dynamically linked library file called `mylib.so`.

The `plaw` function may be accessed at run-time using

```
isis>  add_compiled_function ("mylib.so", "plaw");
```

assuming that `mylib.so` exists somewhere in the ISIS module search path (see e.g. `ISIS_MODULE_PATH`).

For user-defined operator models implemented in C, the `Isis_User_Source_t` structure should be initialized with the `category` field set to `ISIS_FUN_OPERATOR`.

add_slang_function

Purpose: Add a user-defined fit function
Usage: `add_slang_function (name, par_name_array [, norm_indexes])`
See Also: `del_function, list_functions, add_compiled_function, set_function_category, set_param_default_hook`

The first argument, `name`, is the name that will be used in the expression supplied to `fit_fun`; it should be `<= 31` characters long. The corresponding S-Lang function name must have a suffix '`_fit`' appended. The second argument, `par_name_array`, is an array of strings, each `<= 31` characters in length, containing the function parameter names. The optional third argument, `norm_indexes`, is an integer array specifying the zero-based array indices of the parameters which should be interpreted as normalization coefficients. Note that parameter names should be valid S-Lang identifiers.

Physical units for each fit parameter may be specified by appending the units to the parameter name string, enclosed in square brackets ("`[]`"). For example,

```
add_slang_function ("foo", ["kT [keV]", "Mdot [Msun/yr]"]);
```

defines a function of two parameters, `kT` measured in units of keV and `Mdot` measured in units of solar masses per year.

Defining new fit-functions in S-Lang is simpler than in C but, because the function evaluations are performed in an interpreted language, the execution time may be noticeably slower, depending on the implementation. Remember that, in the interpreter, implicit array operations are *much* faster than explicit loops over array indices.

For example, suppose the user-defined fit-function is a power law ($f(x) = ax^b$) and is defined by a S-Lang function called `plaw_fit`, with adjustable parameters `norm` = "a" = `par[0]` and `power` = "b" = `par[1]`. The S-Lang function "`plaw_fit`" which computes the bin-values might look like this:

```
define plaw_fit (lo, hi, par)
{
   variable norm, p, result;

   norm = par[0];
   p    = par[1] + 1;
   result = norm * (hi^p - lo^p) / p;       % integral over bin-width

   return result;
}
```

All user-defined, additive and multiplicative S-Lang fit-functions must follow this basic form; the function interface for operator models is slightly different (see below). The three input array arguments may have any valid variable names, but the arguments will be supplied in the order indicated; bin lower-edge first, bin upper-edge second and parameter array third. The user-defined function must return a single array providing the bin-integrated function value in each bin. Note that, for clarity, error-checking has been omitted from this example.

Thus defined, this `plaw` function may be made available to the fit-engine at run-time using

```
add_slang_function ("plaw", ["norm","power"]);
```

Parameter names must be listed in the order they appear in the parameter array which is passed to the subroutine. The S-Lang function may also be specified using a reference:

```
add_slang_function ("plaw", &plaw_fit, ["norm","power"]);
```

In this case, the function may have any name. To provide a differential version of the same function (as opposed to the usual binned version), provide an array of two function references:

```
add_slang_function ("plaw", [&plaw_binned, &plaw_diff], ["norm","power"]);
```

The differential version should have an interface of the form

```
define plaw_diff (x, p)
{
    % compute the differential value f(x)
    return f;
}
```

To use this new S-Lang function in fitting data, one should use the fit invocation-name with the usual syntax: here, one might use

```
isis>  fit_fun ("plaw(1) + plaw(2)");
```

to sum two instances of the `plaw` function.

When one of the parameters is called `"norm"` (case-insensitive), it will be treated as the only normalization parameter. Alternatively, the zero-based array indicies of those parameters to be treated as normalizing coefficients should be listed using the optional third parameter `norm_indexes`. If no parameter is called `"norm"` and no list is given, none of the function parameters will be adjusted by `renorm_*`.

Default parameter values and ranges may be specified using a separate S-Lang function (see `set_param_default_hook`).

User-defined fit-functions may also access global data, allowing one to write functions which require a significant amount of data to generate the desired result. In particular, one might load a table model from a file, and then use those data to compute model values. For example, consider a file containing the following S-Lang code:

```
require ("xspec");
static variable X_lo, X_hi, Value;

define fast_wabs_init (x)
{
   x = _A(x);
   X_lo = x[[:-2]];
   X_hi = x[[1:]];
   variable y = eval_fun2 (&wabs, X_lo, X_hi, 1.0);

   % rebin assumes bin-integrated quantities
   Value = y * (X_hi - X_lo);
}

fast_wabs_init ( 10.0^[-3:1.3:0.001] );

define fast_wabs_fit (lo, hi, par)
{
   variable y = rebin (lo, hi, X_lo, X_hi, Value);
   y /= hi - lo;

   return y^(par[0]);
}

add_slang_function ("fast_wabs", ["NH_22"]);

provide ("fast_wabs");
```

When this file is loaded, it evaluates the XSPEC absorption model `wabs` for a unit absorbing column and a fixed standard grid, storing the result in the global variables `X_lo`, `X_hi` and `Value`. It also defines a new fit-function called `fast_wabs`. When called, this fit-function interpolates the saved `wabs` values onto the specified grid and rescales the result to match the current value of absorbing column. This is considerably faster than the current XSPEC default implementation which repeatedly recomputes the optical depth contribution from each element, even though the abundances of the absorbing gas remain fixed.

Operator models differ from additive and multiplicative models in that they operate on the result of another function. For example, the Gaussian smoothing operator `gsmooth`, can be invoked using the syntax

```
  fit_fun ("gsmooth (1, mekal(1))");
```

This example computes a spectrum using the `mekal` model and then convolves it with a Gaussian using the `gsmooth` operator. Although isis fit-functions can take an arbitrary number of additional arguments of any type, “operator” models represent an important special case because they correspond closely to the “convolution” models long provided by XSPEC. For historical reasons, ISIS handles operator models in a rather special way.

To implement an operator model in S-Lang, the S-Lang function should be defined with an interface of the form

```
define my_operator_fit (lo, hi, par, fun_value)
```

where the first three arguments are the same as for additive and multiplicative functions and the last argument `fun_value` is a S-Lang array containing the computed function to which the operator will be applied. Before use, operator models implemented in S-Lang must be labeled as operator models by calling the `set_function_category` function:

```
add_slang_function ("my_operator", ["a", "b"]);
set_function_category ("my_operator", ISIS_FUN_OPERATOR);
```

In general, ISIS fit-functions may take an arbitrary number of parameters. This feature makes it possible to encode quite complicated functional relationships into one or more suitably defined ISIS fit-functions that work cooperatively. The usefulness of this feature is perhaps best illustrated by a concrete example.

This example corresponds closely to the design of the nonthermal spectral models for ISIS[1]. In that module, the goal is to compute a variety of nonthermal photon emission spectra that arise from a given nonthermal particle distribution function. Several parameterizations of the particle distribution function are available and each photon emission process carries additional process-specific parameterizations. A natural way to express this relationship is to specify a `pdf` as an argument to each spectral model:

```
sync (1, pdf(1))
invc (1, pdf(1))
ntbrem (1, pdf(1))
```

This syntax means that the `sync` function will be computed for the distribution function specified by `pdf`, and so on.

The implementation is complicated by the fact that, for performance reasons, the particle distribution function is best implemented in a compiled language such as C. Without getting into the details of the implementation, the important point is that one can select the appropriate C subroutine to compute the PDF by creating a S-Lang function `pdf` that returns the name of the PDF that is to be evaluated, along with the parameters for that PDF.

To make this connection, the fit-function `pdf` should return two objects – a string specifying the name of the particle distribution function plus the associated parameter vector:

```
private define pdf (l,h,p)
{
   return ("pdf", p);
}
add_slang_function ("pdf", &pdf, ["a", "b", "c"]);
```

[1] http://space.mit.edu/home/houck/software/slang/modules/nonthermal/

Note that the wrapper function does nothing with the spectral grid arrays `(l,h)`. The only purpose of this model is to enable passing a parameter array on to the named PDF. Each different particle distribution should have a similar wrapper that returns the relevant model name and list of parameter values.

The `sync` function can then be defined to make use of these additional parameters:

```
private define sync (l,h,p)
{
   variable pdf_name, pdf_pars;
   (pdf_name, pdf_pars) = ();
   return compute_sync (l, h, p, pdf_name, pdf_pars);
}
add_slang_function ("sync", &sync, ["norm", "B"]);
```

Note that the `sync` function could also be defined like so:

```
   define sync (pdf_name, pdf_pars, l, h,p)
   {
     ...
   }
```

This definition is functionally identical to the previous one.

For a similar example, see *e.g.* `create_aped_line_profile` and `create_aped_line_modifier`.

add_slang_statistic

Purpose: Add a user-defined S-Lang fit-statistic
Usage: `add_slang_statistic (name, &stat, &report [; <qualifiers>])`
See Also: `set_fit_statistic, load_fit_statistic`

The first argument is the name of the fit-statistic. The next two arguments are addresses of S-Lang functions to compute the fit-statistic and to print the statistic value, respectively.

```
Qualifiers:
  delta_is_chisqr    If present, indicates that the delta-statistic
                     is chi-square distributed (e.g. so that the
                     statistic may be used by 'conf')
```

The function which computes the fit-statistic named `"stat"` should be named `stat_function` and should have the interface

```
    (vec, stat) = stat_function (data, model, weights)
```

where `data` and `model` and arrays of equal length containing the data and model values and where `weights` is an array of the same size containing the statistical weight of each data point. If σ is the uncertainty in the i^{th} data point, the weight is usually $1/\sigma^2$. The function returns the value of the fit-statistic, `stat`, as well as a vector, (`vec`), containing the contribution to the statistic from each bin.

The function which prints the fit-statistic named "stat" should be named stat_report and should have the interface

```
report_string = stat_report (stat, npts, nvpars)
```

where stat is the value of the fit-statistic, npts is the number of data points being fitted, and nvpars is the number of variable fit-parameters. The function returns a string which will be printed at the appropriate time. Note that this string may contain newline characters (\n) allowing it to span multiple lines of output.

```
For example:

% First, define the two functions:
static define mychi_function (y, fx, w)
{
   variable v = (y - fx)^2 * w;
   return (v, sum (v));
}

static define mychi_report (stat, npts, nvpars)
{
   variable s = sprintf ("  My Chisqr = %0.4g\n", stat);
   return s;
}

% Add the statistic to the internal list:
add_slang_statistic ("mychi", &mychi_function, &mychi_report);

% Tell ISIS to use the new fit statistic:
set_fit_statistic ("mychi");
```

alias_fun

Purpose: Derive a new fit-function from an existing one
Usage: alias_fun (name, new_name [; __qualifiers])
See Also: add_slang_function, cache_fun

To derive a new fit-function with the same properties as an existing fit-function, but with a different name, just do

```
alias_fun (old_name, new_name);
```

Several qualifiers support changing the parameter names or their default settings:

```
 names = array of parameter name strings
values = array of default parameter values
freeze = array of integer flags indicating the default freeze state
   min = array of default minimum values
   max = array of default maximum values
params = array of {name, value, freeze, min, max} lists

EXAMPLE:
```

```
alias_fun ("egauss", "FeKa";
            names= ["area [ph/s/cm^2]", "E [keV]", "sigma [keV]"],
            values=[ 0.01,              6.4,      0.5         ],
            freeze=[ 0,                 1,        0           ],
            min=   [ 0,                 5.8,      1e-6        ],
            max=   [ 1,                 7,        2           ]);
```

array_fit

Purpose: Fit a function y=F(x) to data consisting of (X,Y) pairs
Usage: `(pars, stat) = array_fit (x, y, wt, pars, par_min, par_max, &fun)`
See Also: `set_fit_method, set_fit_statistic, fit_counts, fit_flux`

Use this function to fit a function y = F(x) to data consisting of (X,Y) pairs.

```
__Input_Parameters__  __Definition___
   x, y               Input data arrays of equal length.

   wt                 Optional weights for the input Y values;
                      If wt = NULL, all Y values receive wt=1.

   pars               Input array of initial parameter values

   par_min, par_max   Input arrays of allowed parameter ranges.
                      If either value is NULL, the corresponding
                      bound extends to infinity.

   fun_ref            Reference to a S-Lang function with
                      interface  y = f(x, pars)
```

If the fit succeeds, the function returns an array (`pars`) best-fit parameter values and the corresponding value of the fit-statistic (`stat`). If the fit fails, the return values are `pars=NULL` and `stat=0`.

For example, to fit a line to scatter data using equal weights on the data points and with unlimited ranges for the fit-parameters:

```
%  supply a linear fit-function
define fun (x, pars)
{
   return pars[0] + x * pars[1];
}

% fit the data (providing initial parameter values in ‘pars’)
(best, stat) = array_fit (x, y, NULL, pars, NULL, NULL, &fun);
```

assign_model

Purpose: Assign a model to an specific dataset
Usage: `assign_model (id[], model_ref [, arg1 [, arg2, ...]]);`
See Also: `fit_fun, Isis_Active_Dataset`

This function may be used to associate a model with a particular dataset, overriding any model that may have been defined using `fit_fun`.

The `model_ref` argument may be defined in one of two ways:

```
* a String_Type containing a valid model expression
      e.g. "phabs(1) * (mekal(1) + powerlaw(1))"
* a Ref_Type that refers to a S-Lang function:
      e.g. &some_function
```

When the model is defined using a `Ref_Type`, any additional arguments provided to `assign_model` will be passed along to the referenced function when it is evaluated. For example, the following setup:

```
define example_model (i1, i2)
{
    return phabs(i1) * (mekal(i2) + powerlaw(1));
}

assign_model (2, &example_model, 1, 2);
```

is equivalent to:

```
assign_model (2, "phabs(1) * (mekal(2) + powerlaw(1))");
```

To unassign a dataset's model, use `model_ref=NULL`:

```
assign_model (id, NULL);
```

Note that if every dataset has a model specified by `assign_model`, then a default model specified by `fit_fun` will not be evaluated during a fit, even though the parameters of the `fit_fun` model still appear in the `list_par` output and, for the purposes of optimization, may still be treated as variable parameters. In this situation, it may be advisable to use `fit_fun ("null")` to eliminate the extraneous parameters.

bin_center

Purpose: Bin-center utility fit-function
Usage: `bin_center(id)`
See Also: `bin_center_en, bin_width, gauss, Lorentz, poly, delta`

Utility fit-function which evaluates to the wavelength [Ångstrom] at bin-center.

bin_center_en

Purpose: Bin-center utility fit-function
Usage: `bin_center_en(id)`
See Also: `bin_center, gauss, Lorentz, poly, delta`

Utility fit-function which evaluates to the energy [keV] at bin-center.

bin_width

Purpose: Bin-width utility fit-function
Usage: `bin_width(id)`
See Also: `bin_width_en, bin_center, gauss, Lorentz, poly, delta`

Utility fit-function which evaluates to the bin-width in Ångstrom.

bin_width_en

Purpose: Bin-width utility fit-function
Usage: `bin_width_en(id)`
See Also: `bin_width, bin_center, gauss, Lorentz, poly, delta`

Utility fit-function which evaluates to the bin-width in keV.

blackbody

Purpose: Blackbody fit-function
Usage: `blackbody(id)`
See Also: `gauss, Lorentz, poly, delta`

The blackbody function has two variable parameters, the temperature (kT) in keV and the normalization (N).

$$B(E_i^{\mathrm{lo}}, E_i^{\mathrm{hi}}) = 8.0525 \, \frac{N}{(kT)^4} \int_{E_i^{\mathrm{lo}}}^{E_i^{\mathrm{hi}}} \mathrm{d}E \, \frac{E^2}{e^{E/kT} - 1} \tag{7.38}$$

The normalization is the same as that used by XSPEC, with $N = L_{39}/D_{10}^2$ where L_{39} is the source luminosity in units of 10^{39} erg s^{-1} and D_{10} is the source distance in units of 10 kpc.

cache_fun

Purpose: Create a caching fit-function
Usage: `caching_name = cache_fun (name, lo, hi [; __qualifiers]);`
See Also: `add_slang_function, alias_fun`

In some fitting applications (e.g confidence limit searches), computationally expensive model components may be repeatedly evaluated for a single set of parameters. In these circumstances, it may be more efficient to temporarily cache the most recent model result to avoid the expense of re-computing it.

Use the `cache_fun` intrinsic to create a caching version of any fit-function that has at least one parameter. The caching version computes the associated function on the provided wavelength grid and saves the result until the next time a parameter value changes. When the cacheing version is called N times with the same parameters, the underlying function will be evaluated only once, on the first call – the next N-1 calls will be handled by rebinning the cached result.

Note that the caching version will not extrapolate the model beyond the bounds of the specified wavelength grid; any attempt to do so will generate an error or a warning.

```
Qualifiers:
   suffix=STR   Identifier string for the caching model
                (to label different instances).
   mult         Use this qualifier when caching multiplicative
                models.  It ensures that the bin-averaged model
                value is used.  Without this qualifier, the
                rebinned model is bin-integrated.
   warn_grid    Print a warning message when the model is evaluated
                on a grid that extends beyond the cached grid;
                (the default behavior is to throw an exception).
```

```
EXAMPLE:

isis> (lo,hi) = linear_grid(1,20,2000);
isis> variable caching_tbabs_name = cache_fun ("tbabs", lo, hi; mult);
isis> fit_fun ("$caching_tbabs_name(1) * mekal(1)"$);
isis> list_par;
tbabs_cache(1) * mekal(1)
 idx  param          tie-to  freeze   value         min         max
  1  tbabs_cache(1).nH   0     0          1           0      100000
  2  mekal(1).norm       0     0          1           0       1e+10
  3  mekal(1).kT         0     0          1      0.0808        79.9  keV
  4  mekal(1).nH         0     1          1       1e-05       1e+19  cm-3
  5  mekal(1).Abundanc   0     1          1           0        1000
  6  mekal(1).redshift   0     1          0           0          10
  7  mekal(1).switch     0     1          1           0           1
isis>
```

conf

Purpose: Compute single-parameter confidence limits
Usage: `(low, high) = conf(param_index [, level [, tolerance]])`
See Also: `fconf, vconf, conf_loop, fit_counts, fit_flux, conf_joint`

By default, 90% confidence limits are computed. The optional second argument may be used to specify the confidence level. It may be set to one of the values 0, 1 or 2 indicating 68%, 90% or 99% confidence levels respectively. Use `fconf` to specify a particular value of $\Delta\chi^2$.

The tolerance parameter may be used to control how precisely the confidence limit is to be determined; the search for the parameter value at the specified confidence limit will continue until

$$\chi^2(x_{\rm limit}) - (\chi^2(x_{\rm best-fit}) + \delta\chi^2_{\rm limit}) < {\rm tolerance} * \delta\chi^2_{\rm limit} \qquad (7.39)$$

where $\delta\chi^2_{\rm limit}$ is the change in χ^2 corresponding to the specified confidence limit. The default `tolerance` is 10^{-3}.

If an improved fit is found during the confidence limit search, the function updates the internal parameters with the new best-fit parameter values and returns `low = high`. Otherwise the internal parameter table is not modified.

One can also refer to parameters by name:

```
(lo, hi) = conf ("gauss(2).area");
```

If it has been defined, the function `isis_fit_improved_hook` is called immediately before each fit during the confidence limit search. This function must be of the form

```
Integer_Type status = isis_fit_improved_hook ();
```

A non-zero return value causes the algorithm to behave as though a new best-fit parameter value has been detected. If the return value is zero, the algorithm proceeds normally. This hook was added to support distributed computation of single-parameter confidence limits. It provides a mechanism to signal slave processes that the confidence limit search should be re-started from the beginning using a new initial set of parameters.

```
Qualifiers:
   response     Specify responses to be used, if any.
                (default = Assigned_ARFRMF)
                Ideal_ARF | Ideal_RMF | Ideal_ARFRMF
                Assigned_ARF | Assigned_RMF | Assigned_ARFRMF

   flux        If present, use flux-corrected data
               If absent, use counts data
```

conf_grid

Purpose: Generate a parameter grid for computing confidence contours
Usage: `Struct_Type = conf_grid (index, min, max, num)`
See Also: `conf_map_counts`, `conf_map_flux`, `plot_conf`, `save_conf`, `load_conf`

For example:

```
px = conf_grid ("gauss(1).center", 11.43, 12.95, 64);
py = conf_grid ("gauss(1).sigma", 0.01, 0.04, 64);
s = conf_map_counts (px, py);
```

or, using parameter indices:

```
px = conf_grid (2, 11.43, 12.95, 64);
py = conf_grid (3, 0.01, 0.04, 64);
s = conf_map_counts (px, py);
```

The parameter grid points then correspond to the array of values `[min:max:#num]`.

conf_joint

Purpose: Compute two-parameter joint confidence limits
Usage: `Struct_Type = conf_joint (Struct_Type[, delta_chisqr])`
See Also: `conf_map_counts`, `conf_map_flux`, `plot_conf`, `save_conf`, `load_conf`

By default, this routine extracts 68% joint confidence limits for two degrees of freedom, corresponding to a chi-square difference of 2.30. Use the optional argument to supply an alternate delta-chisqr value.

The confidence limits are extracted by interpolating values from the 2D chi-square map generated by `conf_map_counts` or `conf_map_flux`. Therefore, a finer chi-square map grid will usually allow extracting more accurate confidence limit values. The documentation for `conf_map_counts` specifies the form of the input structures.

The return value is a structure which contains the X and Y parameter ranges corresponding to the specified joint confidence limit.

```
Example:
        s = conf_map_counts (px, py);
        jlim = conf_joint (s);

        => jlim = struct {xmin, xmax, ymin, ymax}
```

conf_loop

Purpose: Generate single-parameter confidence limits for specified parameters
Usage: `(min[], max[]) = conf_loop (params[] [, level [, tolerance]][; qualifiers])`
See Also: `conf`, `parallel`

This function calls `conf` to determine single parameter confidence limits for a specified list of fit parameters; for convenience, `params=NULL` indicates that the loop should include all free parameters. If a new best-fit solution during the search for the confidence limits, a new confidence limit search is begun, starting at the new best-fit solution.

```
Qualifier      Default        Meaning
---------      -------        -------
flux           <empty>        If present, perform search by fitting flux-
                              corrected data.
cl_verbose          0         Control verbosity; values < 0 are quieter.
max_param_retries   0         The number of times a slave process
                              should restart its assigned confidence
                              limit search after finding a new best-fit.
prefix         <empty>        Prefix for output files generated by
                              the save option.  If the specified
                              prefix has the form DIR/string,
                              then a subdirectory named DIR will
                              be created to receive output files
```

```
                              prefixed with 'string'.
save                          If present, save intermediate results
                              and generate a parameter file containing
                              the computed confidence limits.
serial                        If present, perform computations on
                              a single CPU.
```

See `parallel` for more details on controlling parallel processes.

For example, this:

```
   (pmin, pmax) = conf_loop ([2,3,6,7] ; save,
                             prefix="/tmp/conf_loop/pars");
```

will compute single-parameter confidence limits for 4 parameters, saving output in the specified directory.

conf_map_counts

Purpose: Generate a 2D chi-square map for counts data
Usage: `Struct_Type = conf_map_counts (Struct_Type x, Struct_Type y [, info])`
See Also: `conf_grid, plot_conf, save_conf, load_conf, conf_map_flux, parallel`

This function maps the chi-square space by stepping two fit-parameters over a specified range. In combination with `set_par_fun`, one can also generate confidence contour plots which include fairly arbitrary coordinate transformations (e.g. contours drawn vs. the log of the parameter rather than the parameter value itself).

By default, the computations are done in parallel; use the `serial` qualifier to force the computations to be performed on a single CPU. See `parallel` for more details on controlling parallel processes.

The parameter indices and an associated uniform grid of values should be specified using a structure of the form:

```
   Struct_Type = struct {index, min, max, num};
```

where `index` gives the parameter index and the remaining three fields specify the grid. The simplest way to generate this struct is to use `conf_grid`.

The optional third argument is a structure whose fields provide references to functions which can be used during confidence contour calculation to customize recovery from failed fit attempts, save parameters in a custom format and to mask out parameter regions to avoid during the calculation. This argument may have the form

```
  info = struct {fail, save, mask}
```

Note that the struct need only include those fields which are actually used.

The return value is a structure containing a 2D array of chi-square values and other information. It has the form:

```
  s = struct {chisqr, px, py, best, px_best, py_best};

Qualifiers:
----------
   flood       If present, visit the map pixels by expanding
               outward from the best-fit, taking the initial
               parameters from the nearest completed fit.

   num_xsub    Number of sub-arrays into which the 2D array
   num_ysub    should be partitioned for parallel computations.
               By default, num_xsub=1, num_ysub=num_slaves.
               The optimal partitioning scheme depends on the
               problem at hand and the number of available CPUs.

   response    Specify responses to be used, if any.
               (default = Assigned_ARFRMF)
               Ideal_ARF | Ideal_RMF | Ideal_ARFRMF
               Assigned_ARF | Assigned_RMF | Assigned_ARFRMF

  try_global   If present, and if the the ‘flood’ qualifier is
               also present, fit each pixel using two different
               sets of initial parameters, one from the nearest
               completed fit and one from the global best-fit.
               Keep the result that gives the lowest fit-statistic.
```

Because confidence maps can be quite cpu-intensive to compute, it may be useful to save the resulting map to a FITS file using `save_conf`. It can then be reloaded later using `load_conf`.

Use `[o]plot_conf` to plot and over-plot the confidence contour map.

```
Example:

   px = conf_grid (2, 11.96, 12.04, 64);
   py = conf_grid (3, 0.015, 0.035, 64);
   s = conf_map_counts (px, py);

% This yields:

print(s);
    chisqr = Double_Type[64,64]
    px = Struct_Type
    py = Struct_Type
    best = 11.0525
    px_best = 12.0046
    py_best = 0.0216003
```

To customize the computation of the confidence contour map, perhaps by writing out additional information for each map element, one can provide a definition for the `save` field of the `info` structure mentioned above. The save hook should be a function of the form

```
   define save_hook (p)
```

where the `p` argument passed to this function is the information structure returned by the corresponding fit-function, e.g. `fit_counts` or `fit_flux`. The hook will be called once each time a new chi-square value is saved in the confidence contour map. For example, to write out an ASCII table containing the chi-square value at every map element plus all the fit parameters, one could do the following:

```
% First, open the output file, saving the file pointer
% in a global variable

variable fp = fopen ("contour_info.txt", "w");
if (fp == NULL)
  exit(1);

%  Define the hook function so that it writes out the
%  current value of the fit-statistic and all the fit-parameters:
define save_hook (p)
{
   variable pars = get_params();

   () = fprintf (fp, "%15.6e", p.statistic);
   foreach (pars)
     {
        variable x = ();
        () = fprintf (fp, "  %15.6e", x.value);
     }
   () = fputs ("\n", fp);
}

% set the save hook to point to your function:
variable info = struct {fail, save, mask}
info.save = &save_hook;

% compute the confidence map, simultaneously generating
% the ASCII file opened above
variable map = conf_map_counts (px, py, info);

%  close the ASCII file
() = fclose(fp);
```

In mapping out the behavior of chi-square it may happen that the chosen fit algorithm fails to converge at some points in the parameter space. Sometimes this is merely a reflection of the fact that the chi-square space is complex and finding the minimum at any given point may be difficult. Unfortunately, such convergence failures may cause the routine to take an inordinately long time to map out the behavior of chi-square and may also degrade the quality of the resulting map.

To provide a way to recover from these convergence failures without restarting the chi-square mapping process, ISIS provides a failure recovery hook via `info.fail`, analogous to the above `save_hook` example. The `fail` field should provide a reference to a function of the form

```
  define fail_hook (p1, p2, best_pars, try_pars, fit_info);
```

where

```
      p1, p2 = the indices of the 2 parameters being mapped
```

```
best_pars = an array of parameter-info structures
            defining the current best-fit parameter set.
            (see get_par_info() for details)
 try_pars = an array of parameter-info structures
            defining where a fit failed to converge.
            (see get_par_info() for details)
 fit_info = the fit_info structure returned
            by the failed call to fit_counts() or fit_flux().
```

The referenced function will be called whenever `fit_counts` or `fit_flux` fails. This function should attempt to determine the best fit given the array `try_pars` as the initial parameter state. On return, it should update the `statistic` field of the `fit_info` structure.

For example, in using the pileup model, it often happens that, the `minim` algorithm is good at getting close to the best fit even when given a relatively poor initial guess, but is relatively slow to improve on a "close" solution. In contrast, although `marquardt` may require a relatively good initial guess, it excels at efficiently optimizing a good initial guess. To use the strengths of both methods, one might use `minim` as the primary algorithm when mapping the chi-square space, but use `marquardt` to recover if `minim` fails.

To do that, one could use a failure-recovery hook like this:

```
define fail_hook (p1, p2, best_pars, try_pars, fit_info)
{
     variable save_method = get_fit_method ();

     set_fit_method ("marquardt");
     () = fit_counts (&fit_info);

     set_fit_method (save_method);
}

info.fail = &fail_hook;
```

One way to speed computation of confidence contour maps using computationally expensive models is to mask out regions of the parameter space which can be ignored. The `mask` hook provides this capability by providing a reference to a function of the form

```
define mask_hook (p1, p2)
```

As arguments, this function should take the coordinates of a point in the parameter space of interest. If a fit should be done at this point, the function should return a non-zero value. If a fit should not be done, the function should return zero.

To compute confidence contours for one or more derived quantities, one can use `set_par_fun` to define the appropriate transformation. Here is an unrealistic example which serves to illustrate the idea.

To compute confidence contours on a log-log plot, we introduce a function to supply the coordinate transformation:

```
define transform_fit (l,h,p)
{
   return 1;
```

```
}
add_slang_function ("transform", ["log_norm", "log_kT"]);

fit_fun ("mekal(1)*transform(1)");

set_par("transform(1).log_norm", -3.0);
set_par("transform(1).log_kT", 0.5);
set_par_fun ("mekal(1).norm", "10^transform(1).log_norm)");
set_par_fun ("mekal(2).kT", "10^transform(1).log_kT");
```

The fit-parameter table is then:

```
mekal(1)*transform(1)
 idx  param              tie-to  freeze    value     min    max
  1  mekal(1).norm           0     1        0.001       0  1e+10
#=>  10^transform(1).log_norm
  2  mekal(1).kT             0     1     3.162278  0.0808   79.9
#=>  10^transform(1).log_kT
  3  mekal(1).nH             0     1            1   1e-05  1e+19
  4  mekal(1).Abundanc       0     1            1       0   1000
  5  mekal(1).Redshift       0     1            0       0     10
  6  mekal(1).Switch         0     1            1       0      1
  7  transform(1).log_norm   0     0           -3       0      0
  8  transform(1).log_kT     0     0          0.5       0      0
```

With this definition, one can now compute confidence contours using the logarithmic parameters (of `transform`) rather than the linear parameters (of `mekal`).

conf_map_flux

Purpose: Generate a 2D chi-square map for flux-corrected data
Usage: `Struct_Type = conf_map_flux (Struct_Type x, Struct_Type y [, info])`
See Also: `conf_grid, plot_conf, save_conf, load_conf`

This function is identical to `conf_map_counts` except that it applies to flux-corrected data. See `conf_map_counts` for details.

vconf

Purpose: Compute single-parameter confidence limits
Usage: `(low, high) = vconf(param_index [, level, [, tolerance]])`
See Also: `conf, fconf, fit_counts, fit_flux, conf_joint`

This is the verbose form of `conf`: on each iteration it prints the current parameter value, chi-square and the change in chi-square away from the initial value.

vfconf

Purpose: Compute single-parameter confidence limits
Usage: `(low, high) = fconf(param_index [, dchisqr, [, tolerance]])`
See Also: `conf, fconf, fit_counts, fit_flux, conf_joint`

This is the verbose form of `fconf`: on each iteration it prints the current parameter value, chi-square and the change in chi-square away from the initial value.

del_function

Purpose: Delete a user-defined fit function
Usage: `del_function ("function")`
See Also: `add_compiled_function, add_slang_function,list_functions`

Note: this function is available only on systems with ELF support. Only user-defined functions may be deleted.

delta

Purpose: Delta-function line profile (for fitting)
Usage: `delta(id)`
See Also: `gauss, Lorentz, poly, bin_center, bin_width`

This fit-function corresponds to a delta-function in that it contributes flux only to the single spectral bin which contains the specified wavelength.

edit_par

Purpose: edit fit parameters
Usage: `edit_par (["filename"])`
See Also: `load_par`

This function allows the user to edit the current set of fit parameters in a text editor. The text editor is specified by the EDITOR environment variable; if the environment variable is not set, `vi` is used. When using `emacs`, the `emacsclient` feature (of `emacs`) may be used to avoid invoking a new `emacs` process for each edit.

If a filename is specified, the model is saved in that file, otherwise, a temporary file is generated and is deleted when editing is finished. If the TMPDIR environment variable is set, the temporary file created for editing will be placed in the indicated directory. Otherwise, the temporary file will be placed in the current directory.

diffevol

Purpose: Differential Evolution optimization algorithm
Usage: `set_fit_method ("diffevol")`
See Also: `optimization, set_fit_method`

Differential Evolution is a very simple population based, stochastic function minimizer. For details, see `http://www.icsi.berkeley.edu/~storn/code.html`.

For help, use:

```
set_fit_method ("diffevol;help");
```

egauss

Purpose: Gaussian line profile function [energy grid]
Usage: `egauss(id)`
See Also: `gauss`

Because multiple Gaussians are allowed in a single fit, the `id` parameter is used as a label to distinguish multiple instances of a particular function type. The function value assigned to each bin is the area under the Gaussian curve which lies inside the bin:

$$\mathrm{gauss}(E_a, E_b) = \frac{A}{\sigma\sqrt{2\pi}} \int_{E_a}^{E_b} \mathrm{d}E \ \exp\left[-\frac{(E-E_0)^2}{2\sigma^2}\right] \tag{7.40}$$

where A is the total area (e.g. photons/s/cm^2) under the Gaussian centered at E_0 with width σ.

eval_counts

Purpose: evaluate the fit-model using the current parameters
Usage: `s = eval_counts([&info_struct])`
See Also: `fit_counts, renorm_counts, ignore, notice, rebin, [un]assign_arf, [un]assign_rmf, eval_stat_counts`

This function evaluates the fit-model and compares it with the counts data to compute the fit-statistic. This is often useful for checking the accuracy of initial parameter values before searching for the best fit values.

See `fit_counts` for details.

eval_flux

Purpose: evaluate the fit-model using the current parameters
Usage: `s = eval_flux([&info_struct])`
See Also: `fit_flux, factor_rsp, renorm_flux, ignore, notice, rebin, [un]assign_arf, [un]assign_rmf`

Analogous to `eval_counts` except that the model is compared with the flux-corrected histogram (see `flux_corr`). See `fit_flux` for details.

eval_fun

Purpose: Evaluate the fit-function on a user-defined grid
Usage: `y = eval_fun (lo, hi)`
See Also: `get_cfun, get_cfun2, fit_fun, eval_fun2, assign_model`

This function evaluates the current fit-function on the specified histogram grid `(lo, hi)`.

```
Example:
         (lo, hi) = linear_grid (1,20,2000);   % define a grid
```

```
y = eval_fun (lo, hi);                % get function values
```

Note that `eval_fun` returns the fit-function integrated over the width of the specified bins.

If the current fit-function evaluates differently for different datasets, it is necessary to specify the dataset index to use when evaluating the fit-function. Do this by setting the `Isis_Active_Dataset` index before calling `eval_fun`. For another way to do this, see `assign_model`.

eval_fun2

Purpose: Evaluate a fit-function on a user-defined grid
Usage: `y = eval_fun2 (handle, lo, hi [, params [, args...]])`
See Also: `get_cfun, fit_fun, eval_fun, fitfun_handle, list_par`

Use `eval_fun2` to evalute a particular fit-function on a given histogram grid `(lo, hi)` using a given vector of function parameters. The fit-function to be evaluated may be specified by name (`"gauss"`), by reference (`&gauss`) or by giving the handle returned by `fitfun_handle`.

```
Example:
          (lo, hi) = linear_grid (1,20,2000);   % define a grid

          % define parameters [area, center, sigma]
          pars = [100.0, 12.0, 0.025];

          % evaluate the function giving its name
          y = eval_fun2 ("gauss", lo, hi, pars);

          % evaluate the function giving a reference
          y = eval_fun2 (&gauss, lo, hi, pars);

          % evaluate the function giving a reference
          handle = fitfun_handle ("gauss");
          y = eval_fun2 (handle, lo, hi, pars);
```

For repetitive function evaluations it is somewhat more efficient to refer to the function using a handle.

The function parameters should be given in the order in which they are listed by `list_par`. If the specified fit-function has no parameters, the fourth argument may be omitted or may be either `NULL` or an array of length zero.

Any additional arguments will be passed on to the fit-function. For example, if the specified fit-function is an operator function the operator will be applied to the vector given in the last argument.

```
Example:
      y = eval_fun2 ("gsmooth", lo, hi, pars, arg);
```

Note that `eval_fun2` returns the fit-function integrated over the width of the specified bins.

eval_stat_counts

Purpose: Evaluate the (counts) fit-statistic using the current parameters
Usage: `Struct_Type = eval_stat_counts()`
See Also: `eval_stat_flux, fit_counts`

This function recomputes the fit-statistic for the counts data without re-evaluating the current model.

This necessarily assumes that the model has already been computed on the correct grid. Note that the model stored internally will be inconsistent if the data have been modified since the last model evaluation (e.g. if the data were rebinned or if different data bins have been ignored or noticed). If the stored model is inconsistent, the computed statistic value will also be inconsistent.

eval_stat_flux

Purpose: Evaluate the (flux) fit-statistic using the current parameters
Usage: `Struct_Type = eval_stat_flux()`
See Also: `eval_stat_counts, fit_counts`

This function recomputes the fit-statistic for the flux-corrected data without re-evaluating the current model.

This necessarily assumes that the model has already been computed on the correct grid. Note that the model stored internally will be inconsistent if the data have been modified since the last model evaluation (e.g. if the data were rebinned or if different data bins have been ignored or noticed). If the stored model is inconsistent, the computed statistic value will also be inconsistent.

exclude

Purpose: Exclude datasets from the fit
Usage: `exclude (data_list)`
See Also: `include, ignore, notice`

This function is similar to `ignore` except that it allows one to exclude a dataset from a fit without changing which bins are currently noticed.

For example, suppose you want to fit a narrow wavelength range in 3 datasets simultaneously. After noticing those wavelength ranges and ignoring everything else, you might want to try excluding one or more datasets from the fit, but you don't want to lose the noticed wavelength ranges. This interaction would look something like

```
% ... try fitting all 3 at once ...
xnotice ([1:3], 12.4, 13.5);

%  -- now fit dataset 2 alone --
exclude (1,3);
```

```
% -- re-include dataset 1, with 12.4-13.5 angstroms
%    still noticed, etc.
include (1);
```

fconf

Purpose: Compute single-parameter confidence limits
Usage: `(low, high) = fconf(param_index [, dchisqr, [, tolerance]])`
See Also: `conf, fconf, fit_counts, fit_flux, conf_joint`

This form of `conf` allows specifying a particular value of $\Delta\chi^2$ appropriate for the desired confidence limit. By default, `dchisqr = 2.71`, and 90% confidence limits are computed.

One can also refer to parameters by name:

```
(lo, hi) = fconf ("gauss(2).area");
```

fit_counts

Purpose: search for best fit parameters
Usage: `s = fit_counts ([&info_struct])`
See Also: `eval_counts, renorm_counts, ignore, notice, freeze, thaw, rebin, [un]assign_arf, [un]assign_rmf, fit_flux, fit_search, set_post_model_hook`

By default, this function fits the current model to the counts data by folding the model through the instrument response. If no instrument response has been assigned, an ideal instrument is assumed. If no errors occurred during the fit, the return status is zero, otherwise the return value is -1.

The optional `Struct_Type` argument should provide the address of a structure:

```
variable info_struct = struct {statistic, num_variable_params, num_bins}
```

On return, the struct fields contain the value of the χ^2 fit-statistic (`statistic`), the number of variable fit parameters (`num_variable_params`) and the number of data bins (`num_bins`).

The `response` qualifier specifies what instrument responses should be applied. Supported qualifier values are `Ideal_ARF`, `Ideal_RMF`, `Ideal_ARFRMF`, `Ideal_RMFARF`, `Assigned_RMF`, `Assigned_ARFRMF`, `Assigned_RMFARF`. As indicated above, the default qualifier value is `Assigned_RMFARF`.

When folding models through the instrument response, the spectral model will be computed on the ARF grid. Furthermore, the model will normally be evaluated only over those wavelength ranges that can contribute to a noticed data bin, as determined by the available RMF (user-defined fit-kernels may change this behavior). For example, consider the case of a dispersed spectrum produced by a diffraction grating. If a single small wavelength range in the data is noticed for fitting, higher order contributions from outside this range will also be included in the predicted counts as long as the RMF includes the higher order contributions. Note that this means that the spectral model will be computed for all wavelength ranges which contribute to higher-order contamination, even if those wavelength ranges correspond to regions of the data that are currently being ignored for purposes of finding the best fitting model.

Several minimization algorithms are available – see `optimization`. Aside from cases where the fit-function evaluation itself might fail, it is possible for the fitting algorithm to fail either because a minimum fit-statistic was not found within a reasonable number of iterations (see Bevington & Robinson (1992) for more details). If the fit algorithm fails, make sure that the initial parameter values are reasonably close to a good fit and that the current parameter values haven't run into the specified upper/lower range limits. Then, try repeating the fit with fewer variable parameters (see `freeze`, `thaw`). See also `Fit_Verbose`.

When the S-Lang function `isis_prefit_hook` is defined in the Global namespace, isis will execute this function immediately before the fit-function is evaluated for the noticed datasets. For example, one might use this hook together with `set_eval_grid_method` to specify a default method for constructing the grid upon which the model will be evaluated:

```
public define isis_prefit_hook ()
{
   message ("called prefit hook");
   set_eval_grid_method (MERGED_GRID, all_data);
}
```

With this definition of the hook, the `MERGED_GRID` method will be used whenever multiple datasets are fitted simultaneously. See `set_eval_grid_method` for more details.

Use the `fit_verbose` qualifier to provide a verbose level that overides the current setting of the intrinsic variable `Fit_Verbose`.

fit_flux

Purpose: search for best fit parameters
Usage: `s = fit_flux ([&info_struct])`
See Also: `eval_flux, renorm_flux, factor_rsp, ignore, notice, freeze, thaw, rebin, [un]assign_arf, [un]assign_rmf, fit_counts, fit_search`

Analogous to `fit_counts` except that the model is compared with the flux-corrected histogram (see `flux_corr`) using the current fit-statistic. By definition, the model for the flux-corrected histogram is

$$F(h) = \frac{\int dER(h,E)A(E)s(E)}{\int dER(h,E)A(E)}, \tag{7.41}$$

where E is the incident photon energy, $s(E)$ is the model spectrum, $R(h,E)$ is the redistribution function or RMF and h is the detector channel. This form ensures that the predicted counts are flux-corrected in the same manner as the observed, background-subtracted counts. Note that this definition is independent of the assigned fit-kernel. When a fit-kernel supports flux-correction, that operation should be consistent with this definition for $F(h)$.

fit_fun

Purpose: define a fit function without prompting for parameters
Usage: `fit_fun("function_string")`
See Also: `assign_model, get_fit_fun, ifit_fun, Lorentz, gauss, load_par, set_par, set_par_fun`

The function definition string must be a valid S-Lang expression. For example, to fit a sum of two Gaussians use

```
isis> fit_fun("gauss(1) + gauss(2)");
```

The integer indices specify different instances of the Gaussian fit-function. By default, a single fit-function is applied to all currently noticed bins in all loaded datasets.

Fit-function parameters may be defined as functions of other fit-parameters (see `set_par_fun`).

Fit-functions may take additional arguments (see `add_slang_function`), and may even return additional arguments, with the restriction that the model string must ultimately evaluate to a `Double_Type` array.

A fit-function which evaluates differently for different datasets may be defined using the ISIS intrinsic variable `Isis_Active_Dataset` or by using `assign_model`.

As an example of how to use `Isis_Active_Dataset`, consider the following:

```
public define my_complicated_function ()
{
   if (Isis_Active_Dataset == 1)
     {
        return gauss(1) + gauss(2);
     }
   else if (Isis_Active_Dataset == 2)
     {
        return Lorentz(1) + Lorentz(2);
     }
}

fit_fun ("my_complicated_function()");
```

When `my_complicated_function()` is evaluated for dataset 1, it will return a sum of Gaussians; when evaluated for dataset 2, it will return a sum of Lorentzians. In effect, each dataset has been assigned a different fit-function. Here, we have assumed that exactly two datasets are loaded and are listed as dataset 1 and dataset 2 in the internal table (see `list_data`).

The ability to retain parameter values from previous fits is a useful feature which may help minimize re-entering parameter values. For example, suppose the initial fit-model is defined as the sum of two Lorentzians:

```
isis> fit_fun("Lorentz(1) + Lorentz(2)");
```

After seeing the resulting fit, one might decide to try a Lorentzian plus a Gaussian,

```
isis> fit_fun("Lorentz(1) + gauss(2)");
```

Seeing that result, one might decide the first fit was better; on returning to that fit-function:

```
isis> fit_fun("Lorentz(1) + Lorentz(2)");
```

the coefficients for Lorentz(2) will have retained their values from the first attempt. This feature can simplify trying different combinations of functions to produce an acceptable model fit.

It is important to remember that, because the fit-function value is interpreted as the value of an integral over the width of each bin, some syntactically correct combinations of intrinsic fit-functions are not analytically consistent. For example, the function definition

```
"Lorentz(1) * Lorentz(2) + 4.0 * poly(1)",
```

is syntactically correct, but analytically inconsistent because

$$\begin{aligned}\text{"Lorentz(1) * Lorentz(2)"} &\equiv \left(\int_{\lambda_{\rm lo}}^{\lambda_{\rm hi}} L(1)\ {\rm d}\lambda\right) \times \left(\int_{\lambda_{\rm lo}}^{\lambda_{\rm hi}} L(2)\ {\rm d}\lambda'\right) \\ &\neq \int_{\lambda_{\rm lo}}^{\lambda_{\rm hi}} L(1)L(2)\ {\rm d}\lambda\end{aligned}$$

Utility functions `bin_width` and `bin_center` are available to support fitting and evaluating bin-averaged and bin-centered functions.

If the model string is either `NULL` or the empty string, then the current model definition, if any, is deleted.

fitfun_handle

Purpose: Obtain a handle for a given fit-function
Usage: `handle = fitfun_handle ("name")`
See Also: `eval_fun2`

See `eval_fun2` for a usage example.

fit_search

Purpose: Search the fit-parameter space
Usage: `best = fit_search (num, &ref [; qualifiers])`
See Also: `fit_counts, eval_counts, fit_flux, eval_flux, save_par, fit_search_info, parallel`

Use this function to automate the process of exploring the fit-parameter space. The first argument specifies the number of Monte-Carlo trials. The second argument provides a reference to a function which will be used to test each randomly generated parameter set. This function may simply evaluate the model for that parameter set (e.g. `&eval_counts`) or it may search for the best fit given that starting point (e.g. `&fit_counts`).

For each Monte-Carlo trial, random parameter values are selected from within the [`min`, `max`) range for each parameter. The test function, `ref`, is then applied using those parameter values.

Intermediate results of the search may be saved in a subdirectory specified by the `dir` qualifier. If the qualifier `save_all` is present, a parameter file will be saved for every trial. Otherwise, a parameter file will be saved each time a new best-fit is encountered.

Each saved parameter file will include a comment line giving the associated fit-statistic. Additional output information may recorded in a user-defined format by defining an `isis_save_par_hook` which uses the `fit_search_info` function (See `save_par` and `fit_search_info` for more information).

For example, to carry out 100 Monte-Carlo trials fitting counts data, saving each successive best-fit in a sub-directory called "trials":

```
stat = fit_search (100, &fit_counts; dir="trials");
```

Each improved parameter set will be saved in files named

```
trials/best.$pid.0
trials/best.$pid.1
trials/best.$pid.2
etc.
```

where `$pid` is the process id.

To carry out 50 Monte-Carlo trials simply evaluating flux-corrected data (not attempting to find the best fit for each trial), and without saving any parameter files to disk:

```
stat = fit_search (50, &eval_counts);
```

On multi-core computers, this function runs in parallel on all available compute cores. To force serial execution, use the `serial` qualifier. See `parallel` for more details on controlling parallel processes.

fit_search_info

Purpose: Retrieve results of the most recent fit
Usage: `Struct_Type = fit_search_info ()`
See Also: `save_par, fit_search, fit_counts`

As `fit_search` continues, the info structure for the most recent fit is saved internally. This structure may be retrieved by calling `fit_search_info`. This function is primarily intended for use in conjunction with `isis_save_par_hook`. See `save_par` for more information.

fit_verbose_info_hook

Purpose: Print verbose status information during a fit
Usage: `fit_verbose_info_hook (statistic, params[], param_names[])`
See Also: `set_post_model_hook, fit_counts`

When `Fit_Verbose` is positive, the S-Lang functions `open_fit_verbose_hook`, `fit_verbose_info_hook` and `close_fit_verbose_hook` are called. No arguments are passed to `open_fit_verbose_hook` and `close_fit_verbose_hook` and none are returned from either function. Before the first function evaluation, `open_fit_verbose_hook` is called to perform any required initializations (such as opening an output file). After each function evaluation, `fit_verbose_info_hook` is called with the names and values of all the parameters and the value of the fit-statistic.

By default, the fit statistic and the parameter names and values are printed to `stdout`.

Users may provide alternate definitions of these functions in the `Global` namespace to print the parameter values to a file or to perform any other suitable task.

For example, to print the parameter values into a file, one could provide the following function definitions (which, for brevity, omit all error checking):

```
% Use a private, file-global variable to hold the file pointer.
private variable Fp = NULL;

public define open_fit_verbose_hook ()
{
   Fp = fopen ("fit_verbose_log.txt", "a");
}

public define close_fit_verbose_hook ()
{
   () = fclose (Fp);
}

define write_par (name, value)
{
   () = fprintf (Fp, "%15.8e = %s\n", value, name);
}

public define fit_verbose_info_hook (stat, values, names)
{
   () = fprintf (Fp, "%11.4e = %s\n", stat, Fit_Statistic);
   array_map (Void_Type, &write_par, names, values);
}
```

freeze

Purpose: freeze one or more fit parameters
Usage: `freeze (par_list)`
See Also: `thaw, tie, untie`

`par_list` may be either a single parameter index or an integer array of indices.

```
    freeze(3);           % freeze param 3
    freeze([1:4]);       % freeze params 1,2,3,4
```

One can also refer to parameters by name:

```
 freeze ("gauss(2).area");
```

gainshift

Purpose: Kernel for introducing a gain shift
Usage: `set_kernel (data_index, "gainshift")`
See Also: `set_kernel, fit_fun`

This kernel applies a linear transformation of the form

```
   E'(c) = E(c)/slope - intercept
```

to the energy grid of the model counts spectrum, `E(c)`, i.e. the energy of channel c from the original EBOUNDS array. The parameters of the linear transformation are fittable. Aside from the energy grid transformation, this kernel performs the same forward-fold computation as the standard fit kernel. The energy grid of any associated background spectrum is not modified; the background associated which each detector channel remains fixed. Flux-correction is not supported.

gauss

Purpose: Gaussian line profile function [wavelength grid]
Usage: `gauss(id)`
See Also: `Lorentz, voigt, poly, delta, bin_width, bin_center`

Because multiple Gaussians are allowed in a single fit, the `id` parameter is used as a label to distinguish multiple instances of a particular function type. The function value assigned to each bin is the area under the Gaussian curve which lies inside the bin:

$$\text{gauss}(\lambda_a, \lambda_b) = \frac{A}{\sigma\sqrt{2\pi}} \int_{\lambda_a}^{\lambda_b} \mathrm{d}\lambda \; \exp\left[-\frac{(\lambda-\lambda_0)^2}{2\sigma^2}\right] \tag{7.42}$$

where A is the total area (e.g. photons/s/cm^2) under the Gaussian centered at λ_0 with width σ.

get_convolved_model_flux

Purpose: load convolved model values into a S-Lang structure
Usage: `Struct_Type = get_convolved_model_flux (hist_index)`
See Also: `get_model_flux, get_data_flux, factor_rsp`

This function returns a structure with three array fields, `bin_lo`, `bin_hi` and `value`:

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
   s.value = bin value [photons/sec/cm^2]
```

Use this function to load the convolved model flux for data set `hist_index` into S-Lang array variables.

The value returned by `get_convolved_model_flux` is

$$F(h) = \frac{\int dER(h,E)A(E)s(E)}{\int dER(h,E)A(E)}, \tag{7.43}$$

where the integral extends over the full energy range, ensuring that the instrumental redistribution is fully accounted for.

get_cfun

Purpose: get the y-values of the differential fit-function
Usage: `y = get_cfun (x_array)`
See Also: `eval_fun, fit_fun, assign_model`

Given an array of x-coordinate values `x_array`, this function returns an array containing the corresponding (unbinned) values of the currently defined fit-function. See also `[o]plot`

```
Example:
          x = [8.5 : 10.2 : 0.001];      % define an x-grid
          y = get_cfun (x);              % get function values

          oplot(x, get_cfun(x));         % another usage example
```

Note that, if the current fit-function evaluates differently for different datasets, it is necessary to specify the dataset index to use when evaluating the fit-function. Do this by setting the `Isis_Active_Dataset` index before calling `eval_fun`. For another way to do this, see `assign_model`.

get_cfun2

Purpose: Evaluate a differential fit-function on a user-defined grid
Usage: `y = get_cfun2 (handle, x, [, params])`
See Also: `get_cfun, fit_fun, eval_fun2, fitfun_handle, list_par`

Use `get_cfun2` to evalute a particular fit-function on a particular wavelength grid using a given vector of function parameters. The fit-function to be evaluated may be specified by name (`"voigt"`), by reference (`&voigt`) or by giving the handle returned by `fitfun_handle`.

```
Example:
          e = [0.9:1.1:1.e-3];   % define a grid

          % define parameters [norm, energy, fwhm, vtherm]
          pars = [1.0, 1.0, 0.02, 1.e3];

          % evaluate the function giving its name
          y = get_cfun2 ("voigt", _A(e), pars);

          plot(e, reverse(y));
```

For repetitive function evaluations it is somewhat more efficient to refer to the function using a handle (e.g. see `eval_fun2`).

The function parameters should be given in the order in which they are listed by `list_par`. If the specified fit-function has no parameters, the corresponding argument may be omitted or may be either `NULL` or an array of length zero.

get_fit_fun

Purpose: Get the current fit-function definition
Usage: `s = get_fit_fun ()`
See Also: `fit_fun, get_par_info, get_par, get_params, get_fun_components`

If no fit-function has been defined, this function returns NULL.

```
For example:
```

```
isis> get_fit_fun;
NULL
isis> fit_fun ("gauss(1) + poly(1)");
isis> s = get_fit_fun();
isis> print(s);
gauss(1) + poly(1)
isis>
```

get_fun_components

Purpose: Get the names of the components of the current fit-function
Usage: `names[] = get_fun_components ()`
See Also: `get_fit_fun, fit_fun, get_par_info, get_par, get_params, get_fun_params`

If no fit-function has been defined, this function returns NULL. For example:

```
isis> fit_fun ("gauss(1) + wabs(1)*mekal(1) + gauss(1)");
isis> s=get_fun_components();
isis> print(s);
"gauss(1)"
"mekal(1)"
"wabs(1)"
```

This function can be used in scripts together with other other information retrieval functions to automatically access all details about the current fit function. Continuing the example, one might do something like:

```
isis> p = get_params (get_fun_params(s[0]));
isis> p;
Struct_Type[3]
isis> print(p[0]);
 name = gauss(1).area
 index = 1
 value = 1
 min = -1.79769e+308
 max = 1.79769e+308
 freeze = 0
 tie = 0
 is_a_norm = 1
 fun = NULL
```

get_fun_params

Purpose: get the parameter indices for a given fit-function
Usage: `y = get_fun_params ("gauss(1)")`
See Also: `fit_fun, get_params, set_params`

For example:

```
fit_fun ("gauss(1) + poly(1)");
```

```
isis> p = get_fun_params ("poly(1)");
isis> print(p);
4
5
6
isis> v = get_params(p);
isis> print(v[0]);
    name = poly(1).a0
    index = 4
    value = 1
    min = -1.79769e+308
    max = 1.79769e+308
    freeze = 0
    tie = 0
    is_a_norm = 1
    fun = NULL
```

get_kernel

Purpose: Determine the fit-kernel assigned to a dataset
Usage: `s = get_kernel (hist_index)`
See Also: `load_kernel, print_kernel,set_kernel, list_kernels`

Note that the value returned by this function is accurate only after the kernel has been assigned and the current fit-function has been evaluated at least once.

If the fit-kernel is uninitialized, an error will be generated. If the fit-kernel has been initialized, but has been changed since the last time the fit-function was evaluated, the value returned by this function will not reflect the change.

get_model_counts

Purpose: load binned model values into a S-Lang structure
Usage: `Struct_Type = get_model_counts(hist_index)`
See Also: `get_data_counts`

This function returns a structure with three array fields, `bin_lo`, `bin_hi` and `value`:

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
   s.value = bin value [counts]
```

Use this function to load the binned model counts for data set `hist_index` into S-Lang array variables.

The model counts are computed using

$$C(h) = B(h) + t \int dE \mathcal{F} \left(R(h,E), A(E), S(E) \right) \tag{7.44}$$

where $B(h)$ is the background spectrum, t is the exposure time, $\mathcal{F}$ is the fit-kernel, $R(h,E)$ is the RMF, $A(E)$ is the ARF and $S(E)$ is the model for the incident photon spectrum. Note that the model includes any background contribution which has been assigned.

get_model_flux

Purpose: load binned model values into a S-Lang structure
Usage: `Struct_Type = get_model_flux (hist_index)`
See Also: `get_data_flux, factor_rsp`

This function returns a structure with three array fields, `bin_lo`, `bin_hi` and `value`:

```
hist_index = integer index of spectrum in internal list
  s.bin_lo = bin left edge [Angstrom]
  s.bin_hi = bin right edge [Angstrom]
   s.value = bin value [photons/sec/cm^2]
```

Use this function to load the binned model flux for data set `hist_index` into S-Lang array variables.

The value returned by `get_model_flux` is

$$F(h) = \int dES(E) \tag{7.45}$$

where the integral extends over the energy width of bin h.

get_num_pars

Purpose: get the number of parameters in the current fit model
Usage: `n = get_num_pars ()`
See Also: `get_par, set_par, list_par, save_par, get_num_pars, get_par_info`

This function is useful in scripts which may need to loop over all parameters of the current model. For example, to print information for each fit parameter, use:

```
variable n, i;

n = get_num_pars ();
_for (1, n, 1)
   {
       i = ();
       print(get_par_info (i));
   }
```

get_par

Purpose: get the value of a fit parameter
Usage: `par = get_par (idx)`
See Also: `set_par, list_par, save_par, , get_num_pars, get_par_info, get_fit_fun`

Given the index (`idx`) of a fit parameter, one can retrieve its value in a S-Lang variable. The indices of current fit parameters may be determined using `list_par`.

One can also refer to parameters by name:

```
x = get_par ("gauss(1).area");
```

get_params

Purpose: Copy fit-parameter information into an array of structs
Usage: `Struct_Type[] = get_params ([list])`
See Also: `set_params, set_par, get_fit_fun`

This function returns an array of structs:

```
isis> fit_fun ("gauss(1)");
isis> list_par;
gauss(1)
 idx  param         tie-to  freeze     value      min       max
  1  gauss(1).area     0     0             1        0         0
  2  gauss(1).center   0     0            12        0         0
  3  gauss(1).sigma    0     0         0.025        0         0
isis> x=get_params;
isis> print(x[2]);
    name = gauss(1).sigma
    index = 3
    value = 0.025
    min = -1.79769e+308
    max = 1.79769e+308
    freeze = 0
    tie = 0
    is_a_norm = 0
    fun = NULL
```

get_par_info

Purpose: get all information on a fit parameter
Usage: `s = get_par_info (idx)`
See Also: `get_num_pars, get_par, set_par, list_par, save_par, get_fit_fun`

Given the index (`idx`) of a fit parameter, one can retrieve the associated information from the internal parameter table; this information is returned as fields of a S-Lang structure. The indices of current fit parameters may be determined using `list_par`. For example:

```
isis> s=get_par_info(2);
isis> print(s);
{name="gauss(1).center",
 index=2,
 value=12.0,
 min=-1.7976931348623157e+308,
 max=1.7976931348623157e+308,
 hard_min=-inf,
 hard_max=inf,
 step=0.0,
 relstep=0.0,
```

```
freeze=0,
tie=NULL,
units="A",
is_a_norm=0,
fun=NULL}
```

If non-NULL, the `fun` parameter contains the expression string specified by `set_par_fun`.

One can also refer to parameters by name:

```
x = get_par_info ("gauss(1).area");
```

ifit_fun

Purpose: define a fit function with interactive prompting for parameters
Usage: `ifit_fun ("function_string")`
See Also: `fit_fun, plot_bin_density, get_cfun`

This function reads cursor positions from an existing bin-*density* plot of a data-set and uses those coordinates to compute initial guesses for parameters of the specified function.

```
Example:
      plot_bin_density;            % MUST plot bin-density to use ifit_fun
      ignore ([1:12]);             % ignore all data, then next step will
                                   % notice a single region for fitting
      xnotice(1, 6,7);             % select the data interval for fitting
      xrange(6,7);                 % set the plot x-limits to the same interval
      yrange(y1,y2);               % set the plot y-limits appropriately
      plot_data_flux (1);
                                  % point and click to define fit params
      ifit_fun("Lorentz(1) + Lorentz(2) + poly(1)");

      eval_flux;                  % evaluate the model using those params
      oplot_convolved_model_flux(1);        % over-plot the model
```

ignore

Purpose: Ignore a wavelength range when fitting
Usage: `ignore (hist_index_list [, lambda_lo, lambda_hi])`
See Also: `ignore_en, notice, xnotice, exclude, include`

`hist_index_list` may be either a single histogram index or an integer array of indices. All data bins are noticed by default. If either of the wavelength range arguments are missing, the limiting value is taken from the input data. Therefore, omitting both range arguments ignores the entire wavelength range.

```
Example:
      ignore ([1:4]);               % ignore data sets 1,2,3,4
      ignore ([2:4], 10, 11);       % ignore 10 <= lambda < 11 for
                                    % data sets 2,3 and 4.
      ignore ([ [1:4], [6:8] ]);    % ignore data sets 1-4 and 6-8
```

Note that when fitting data using an ARF and RMF the RMF is used to determine which model bins contribute to the noticed data bins. Use `ignore_en` to ignore an energy range in keV.

ignore_en

Purpose: Ignore an energy range when fitting
Usage: `ignore_en (hist_index_list [, E_lo, E_hi])`
See Also: `ignore, notice_en, xnotice_en, exclude, include`

This is an alternate form of the `ignore` function that takes an energy range in keV instead of a wavelength range. See `ignore` for details.

ignore_list

Purpose: Ignore a list of bins when fitting
Usage: `ignore_list (datasets[], list)`
See Also: `ignore, notice_list, xnotice, exclude, include`

This is an alternate form of the `ignore` function that takes a list of bin indices instead of a wavelength range. For example:

```
% to ignore bins with < 10 counts
d = get_data_counts(1);
ignore_list (1, where(d.value < 10.0));
```

Note that the list of bin indices refers to the internal data which is stored in increasing wavelength order.

See `ignore` for details.

ignore_values

Purpose: Ignore bins with values exceeding a threshold
Usage: `ignore_values (datasets[], lo1, hi1 [,lo2, hi2...]  ; qualifiers)`
See Also: `ignore, ignore_list, notice, notice_values, notice_list, xnotice, exclude, include`

Ignore bins in specific wavelength or energy intervals that also meet criteria specified by the supported qualifiers. If multiple datasets are specified, their spectral grids should match exactly.

```
Qualifier    Default     Meaning
---------    -------     -------
unit          Angstrom   physical units of (lo, hi)
min_sum       NULL       if defined, ignore only bins for which
                            (sum over datasets) >= min_sum
min_val       NULL       if defined, ignore only bins for which
                            (value in every dataset) >= min_val
```

For example,

```
ignore_values ([2,4,5], 1.0, 1.5 ; min_sum=20, min_val=5, unit="kev");
```

will ignore bins falling entirely within the range 1-1.5 keV and which also have more than 20 counts when summed over datasets 2,4 and 5 and which have at least 5 counts in each of those datasets.

include

Purpose: Include a list of datasets in the fit
Usage: `include (data_list)`
See Also: `exclude, ignore, notice_en, xnotice_en`

See `exclude` for details.

list_fit_methods

Purpose: List the currently defined fit methods
Usage: `list_fit_methods`
See Also: `load_fit_method, load_fit_statistic, add_slang_statistic`

list_free

Purpose: List all free parameters
Usage: `list_free ([arg])`
See Also: `list_par, set_par, get_par`

The optional argument is used to redirect the output. If `arg` is omitted, the output goes to `stdout`. If `arg` is of type `Ref_Type`, it the output string is stored in the referenced variable. If `arg` is a file name, the output is stored in that file. If `arg` is a file pointer (`File_Type`) the output is written to the corresponding file.

This function lists all free fit parameters; it is equivalent to `list_par(1)`. See `list_par` for details.

list_functions

Purpose: List the currently defined fit functions
Usage: `list_functions`
See Also: `add_compiled_function, add_slang_function, del_function`

The generated listing gives the name of each function.

```
Example:
isis> list_functions;
Lorentz         bin_width       egauss          pileup
Powerlaw        blackbody       gauss           poly
isis>
```

See §8) for information on accessing XSPEC source models.

list_kernels

Purpose: list the available fit-kernels
Usage: `list_kernels`
See Also: `set_kernel, get_kernel, load_kernel`

```
Example:
isis> list_kernels;
"pileup"
"std"
```

list_par

Purpose: list current fit function and parameters
Usage: `list_par ([arg])`
See Also: `list_free, edit_par, set_par, get_par, save_par, set_par_fun`

The optional argument is used to redirect the output. If `arg` is omitted, the output goes to `stdout`. If `arg` is of type `Ref_Type`, it the output string is stored in the referenced variable. If `arg` is a file name, the output is stored in that file. If `arg` is a file pointer (`File_Type`) the output is written to the corresponding file.

The parameter listing looks like this:

```
gauss(1) + poly(1)
 idx  param        tie-to  freeze    value      min          max
  1  gauss(1).area     0      0       103.6        0            0
  2  gauss(1).center   0      0        12.1       10           13
  3  gauss(1).sigma    0      0       0.022    0.001          0.1
  4  poly(1).a0        0      0       1.2e4        0            0
  5  poly(1).a1        0      1           0        0            0
  6  poly(1).a2        0      1           0        0            0
```

The first line defines the form of the fit-function. The parameter index `idx` may be used to refer to individual fit parameters (see `set_par`). `freeze = 1 (0)` indicates that the corresponding parameter value is frozen (variable). If two parameter values are tied together, the connection is indicated in the `tie-to` column. For example, if parameter 1 has `tie-to = 5`, that means the value of parameter 1 is tied to the value of parameter 5; if parameter 5 changes, parameter 1 will follow the change exactly. If `min=max=0`, the corresponding parameter value is unconstrained.

In input parameter files (see `load_par`), lines beginning with a '`#`' are mostly ignored and may be used to include comments. Exceptions to this rule are "special" comment lines which are used to support additional functionality such as, e.g. writing some parameters as functions of other parameters (see `set_par_fun`). Note that, aside from these special cases, comment lines are not loaded by `load_par` and will not be preserved if file is later overwritten by `save_par`.

lmdif

Purpose: Variant of Levenberg-Marquardt minimization algorithm
Usage: `set_fit_method ("lmdif")`
See Also: `optimization, set_fit_method, mpfit`

This interface invokes `mpfit` with a limited set of options and is provided primarily for backward compatibility with isis versions 1.5 and earlier. See `mpfit` for details.

For help on lmdif options, use:

```
set_fit_method ("lmdif;help");
```

The `lmdif tol` option defines both the `ftol` and `xtol` parameters of `mpfit`.

load_conf

Purpose: Load a 2D chi-square map from a FITS file
Usage: `Struct_Type = load_conf (file)`
See Also: `conf_map_counts, save_conf, plot_conf`

This function reads a 2D confidence map from a FITS file previously created by `save_conf`.

For example,

```
% Create and save a confidence map:
   map = conf_map_counts (px, py);
   save_conf (map, "map.fits");
% reload the map:
   copy = load_conf ("map.fits");
```

See `conf_map_counts` for details.

load_fit_method

Purpose: load a user-defined fit-method
Usage: `status = load_fit_method ("library.so", "name")`
See Also: `register_slang_optimizer, set_fit_method, add_to_isis_module_path`

See `math/marq.c` in the ISIS distribution for an example of how to implement a user-defined fit-method. To add a new method called `newfit` which has been compiled into a shared library (e.g. `libnewfit.so`), use

```
ret = load_fit_method ("libnewfit.so", "newfit");
```

The return value is (0/-1) to indicate success/failure.

load_fit_statistic

Purpose: load a user-defined fit-statistic
Usage: `status = load_fit_statistic ("library.so", "name")`
See Also: `set_fit_statistic, add_to_isis_module_path, add_slang_statistic`

To add a new method called `newstat` which has been compiled into a shared library (e.g. `libnewstat.so`), use

```
ret = load_fit_statistic ("libnewstat.so", "newstat");
```

The return value is (0/-1) to indicate success/failure.

load_kernel

Purpose: load a user-defined fit-kernel
Usage: `ret = load_kernel ("library_name", "init_name" [, "init_args"])`
See Also: `set_kernel, print_kernel, list_kernels, add_to_isis_module_path`

See `src/pileup_kernel.c` in the ISIS distribution for an example of how to implement a user-defined fit-kernel. Assuming such a fit-kernel has been implemented with an initialization function called `my_kernel_init` and compiled into a shared library (e.g. `libmy_fit_kernel.so`), the user-defined fit-kernel may be dynamically linked to ISIS by using

```
ret = load_kernel ("libmy_fit_kernel.so", "my_kernel_init");
```

The return value is (0/-1) to indicate success/failure. The optional string argument provides a mechanism to supply parameters which may be required at the time of the kernel's initialization.

load_par

Purpose: load fit function and parameters from a file
Usage: `load_par ("filename")`
See Also: `list_par, save_par, edit_par, set_par, get_par`

Use this function to define a fit-function and its parameter values by reading an ASCII file. See `list_par` for file format details.

Lorentz

Purpose: Lorentzian line profile function
Usage: `Lorentz(id)`
See Also: `gauss, voigt, poly, delta, bin_width, bin_center`

The `id` parameter identifies a particular instance of a Lorentzian profile; multiple instances are allowed in a single fit. The function value assigned to each bin is the area under the Lorentzian curve which lies inside the bin:

$$\mathrm{Lorentz}(x_a, x_b) = \frac{A}{\pi} \int_{x_a}^{x_b} \mathrm{d}x \, \frac{\Gamma/2}{(x - x_0)^2 + (\Gamma/2)^2} \tag{7.46}$$

where A is the total area (e.g. photons/s/cm^2) under the Lorentzian centered at x_0 with full-width at half-maximum Γ.

marquardt

Purpose: Levenberg-Marquardt minimization algorithm
Usage: `set_fit_method ("marquardt")`
See Also: `optimization, set_fit_method`

This is an alternate implementation of the Levenberg-Marquardt algorithm. This algorithm was derived under the assumption that the function to be minimized is the Chi-square fit-statistic. For details on the derivation, see e.g. Bevington and Robinson (1992).

The algorithm supports several optional parameters:

```
 __Option__ __Default__  __Purpose__
  max_loops  50          Max number of iterations
        tol  1.e-4       Fractional chisqr acceptance tolerance
      delta  1.e-6       Initial numerical derivative step size
jump_factor  10.0        Lambda adjustment factor
```

To see a list of optional parameters, use

```
  set_fit_method ("marquardt;help");
```

mpfit

Purpose: Variant of Levenberg-Marquardt minimization algorithm
Usage: `set_fit_method ("mpfit")`
See Also: `optimization, set_fit_method`

The `mpfit` Levenberg-Marquardt algorithm is the default minimization algorithm used in fitting models to data. This algorithm was derived under the assumption that the function to be minimized is the Chi-square fit-statistic. For details on the derivation, see e.g. Bevington and Robinson (1992).

For help, use:

```
set_fit_method ("mpfit;help");
```

`mpfit` computes numerical derivatives of the chi-square function with respect to each free parameter. Computation of these derivatives requires choosing a step size for each parameter. This step size may be specified using either an absolute measure, `step`, or a relative measure, `relstep` (see `set_par`). The numerical derivative step size, `h`, is determined by the following algorithm:

```
    eps = sqrt(max([epsfcn, machine_epsilon]));
```

```
    h = eps * abs(param);
    if (step > 0) h = step;
    if (relstep > 0) h = relstep * abs(param);
    if (h == 0) h = eps;
```

where `param` is the parameter value, `epsfcn` is a parameter of the method, and `machine_epsilon=2.2204460e-16`.

optimization

Purpose: Optimization methods provided with isis
Usage: -
See Also: `set_fit_method, list_fit_methods`

Isis provides a variety of optimization methods suitable for wide variety of fitting applications. Each method has its own strengths and weaknesses. Because no single method works well on every problem, it is often necessary to give some thought to which optimization method is likely to be effective for a given problem.

When the problem of interest is well behaved (e.g. the statistic is a smoothly varying function of a small number of uncorrelated parameters and has a unique minimum whose location can be guessed with reasonable accuracy), a local optimization method such as Levenberg-Marquardt is most likely to perform well, in the sense of accurately locating the minimum after a relatively small number of model evaluations (see `mpfit`, `marquardt`, `plm`). The Powell method may also work well in such cases (see `powell`).

For more difficult problems, the simplex method (see `simplex`, `subplex`) or Powell method (see `powell`) may be more effective, although significantly more model evaluations may be required. Global methods such as differential evolution (`diffevol`) or simulated annealing (`simann`) may produce results even when other methods make little or no progress. However, the large number of model evaluations sometimes required by these methods means they can be very CPU intensive.

plm

Purpose: Parallel Levenberg-Marquardt optimization method
Usage: `set_fit_method ("plm")`
See Also: `optimization, set_fit_method, parallel`

This is a parallelized version of the Levenberg-Marquardt optimization method. The basic algorithm was derived under the assumption that the function to be minimized is the Chi-square fit-statistic. For details on the derivation, see e.g. Bevington and Robinson (1992).

The algorithm supports several optional parameters, with default values shown in parentheses:

```
          tol (1.e-4)  Chi-square (X) has converged when dX/X <= tol
       lambda (0.1)    Levenberg-Marquardt parameter
  grow_factor (10)     lambda increases by lambda *= grow_factor
shrink_factor (0.1)    lambda decreases by lambda *= shrink_factor
    max_loops (100)    Maximum number of trials
        delta (1.e-4)  param delta for numerical derivative is
                          (|p| + sqrt(delta))*sqrt(delta)
```

The number of compute slaves may be specified using the `num_slaves` option. For example:

```
set_fit_method ("plm;num_slaves=4;max_loops=200");
```

See `parallel` for more details on controlling parallel processes.

To see a list of optional parameters, use

```
set_fit_method ("plm;help");
```

simann

Purpose: Variant of Simulated Annealing minimization algorithm
Usage: `set_fit_method ("simann")`
See Also: `optimization, set_fit_method`

To see a list of optional parameters, use

```
set_fit_method ("simann;help");
```

The following description is an excerpt from the documentation included with the code.

This routine implements the continuous simulated annealing global optimization algorithm described in Corana et al.'s article *Minimizing Multimodal Functions of Continuous Variables with the Simulated Annealing Algorithm* in the September 1987 (vol. 13, no. 3, pp. 262-280) issue of the ACM Transactions on Mathematical Software.

A very quick (perhaps too quick) overview of SA: SA tries to find the global optimum of an N dimensional function. It moves both up and downhill and as the optimization process proceeds, it focuses on the most promising area.

To start, it randomly chooses a trial point within the step length VM (a vector of length N) of the user selected starting point. The function is evaluated at this trial point and its value is compared to its value at the initial point.

In a maximization problem, all uphill moves are accepted and the algorithm continues from that trial point. Downhill moves may be accepted; the decision is made by the Metropolis criteria. It uses T (temperature) and the size of the downhill move in a probabilistic manner. The smaller T and the size of the downhill move are, the more likely that move will be accepted. If the trial is accepted, the algorithm moves on from that point. If it is rejected, another point is chosen instead for a trial evaluation.

Each element of VM periodically adjusted so that half of all function evaluations in that direction are accepted.

A fall in T is imposed upon the system with the RT variable by T(i+1) = RT*T(i) where i is the ith iteration. Thus, as T declines, downhill moves are less likely to be accepted and the percentage of rejections rise. Given the scheme for the selection for VM, VM falls. Thus, as T declines, VM falls and SA focuses upon the most promising area for optimization.

The importance of the parameter T: The parameter T is crucial in using SA successfully. It influences VM, the step length over which the algorithm searches for optima. For a small intial

T, the step length may be too small; thus not enough of the function might be evaluated to find the global optima. The user should carefully examine VM in the intermediate output (set IPRINT = 1) to make sure that VM is appropriate. The relationship between the initial temperature and the resulting step length is function dependent.

To determine the starting temperature that is consistent with optimizing a function, it is worthwhile to run a trial run first. Set RT = 1.5 and T = 1.0. With RT > 1.0, the temperature increases and VM rises as well. Then select the T that produces a large enough VM.

For modifications to the algorithm and many details on its use, (particularly for econometric applications) see Goffe, Ferrier and Rogers, "Global Optimization of Statistical Functions with Simulated Annealing," Journal of Econometrics, vol. 60, no. 1/2, Jan./Feb. 1994, pp. 65-100.

Minimum_Stat_Err

Purpose: Defines the default minimum valid uncertainty for counts data
Usage: `(intrinsic global variable)`
See Also: `load_data, define_counts, fit_counts, set_min_stat_err, get_min_stat_err`

When input data values are to be treated as counts, isis attempts to validate the corresponding uncertainty values, requiring them all to be greater than or equal to some minimum positive value, `min_stat_err`. The minimum valid uncertainty may be specified when the data is loaded using (e.g. see `load_data`), otherwise, if `Minimum_Stat_Err` is positive, its value is used, otherwise, the minimum value is 1.

For example, defining a single-bin counts dataset with has bin value `1.e-5` and uncertainty `1.e-8`:

```
isis> define_counts (1.0, 2.0, 1.e-5, 1.e-8);
1
isis> print(get_data_counts(1).err);
1
```

Note that isis replaced the `1.e-8` uncertainty value because it was inconsistent with default expectations for this kind of data.

To force isis to retain positive counts-uncertainty values smaller than unity, define a minimum (positive) allowable uncertainty value using the `min_stat_err` qualifier:

```
isis> define_counts (1.0, 2.0, 1.e-5, 1.e-8; min_stat_err=1.e-20);
2
isis> print(get_data_counts(2).err);
1e-08
```

Here, isis kept the `1.e-8` uncertainty value.

When input data is to be treated as flux, this kind of filtering does not take place:

```
isis> define_flux (1, 2, 1.e-5, 1.e-8);
1
isis> print(get_data_flux(1).err);
1e-08
```

Uncertainties for flux values must be positive.

To shut off warnings about invalid uncertainties being replaced, set `Warn_Invalid_Uncertainties=0`.

notice

Purpose: Notice a wavelength range when fitting
Usage: `notice (hist_index_list [, lambda_lo, lambda_hi])`
See Also: `notice_en, ignore, xnotice, exclude, include`

Because all data bins in all data sets are noticed by default, this function is most useful when some datasets or wavelength ranges have been previously `ignore`ed. One can notice a particular wavelength range and simultaneously ignore wavelengths outside that range using `xnotice` (for "exclusive-notice"). If either of the wavelength range arguments are missing, the limiting value is taken from the input data; therefore, omitting both range values is equivalent to noticing the entire wavelength range.

Note that when fitting data using an ARF and RMF, the RMF is used to determine which model bins contribute to the noticed data bins.

notice_en

Purpose: Notice an energy range when fitting
Usage: `notice_en (hist_index_list [, E_lo, E_hi])`
See Also: `notice, ignore_en, xnotice_en, exclude, include`

This is an alternate form of the `notice` function that takes an energy range in keV instead of a wavelength range. See `notice` for details.

notice_list

Purpose: Notice a list of bins when fitting
Usage: `notice_list (datasets[], list)`
See Also: `notice, ignore_list, xnotice, exclude, include`

This is an alternate form of the `notice` function that takes a list of bin indices instead of a wavelength range. For example:

```
% to notice bins with > 10 counts
d = get_data_counts(1);
ignore(1);
notice_list (1, where(d.value > 10.0));
```

Note that the list of bin indices refers to the internal data which is stored in increasing wavelength order.

See `notice` for details.

notice_values

Purpose: Notice bins with values exceeding a threshold
Usage: `notice_values (datasets[], lo1, hi1 [,lo2, hi2...]  ; qualifiers)`
See Also: `ignore_values, ignore, ignore_list, notice, notice_list, xnotice, exclude, include`

Exclusively notice bins in specific wavelength or energy intervals that also meet criteria specified by the supported qualifiers. If multiple datasets are specified, their spectral grids should match exactly.

```
Qualifier    Default     Meaning
---------    -------     -------
unit          Angstrom   physical units of (lo, hi)
min_sum       NULL       if defined, notice only bins for which
                            (sum over datasets) >= min_sum
min_val       NULL       if defined, notice only bins for which
                            (value in every dataset) >= min_val
```

For example,

```
notice_values ([2,4,5], 1.0, 1.5 ; min_sum=20, min_val=5, unit="kev");
```

will exclusively notice bins falling entirely within the range 1-1.5 keV and which also have more than 20 counts when summed over datasets 2, 4 and 5 and which have at least 5 counts in each of those datasets.

open_fit

Purpose: Open fit object
Usage: `Struct_Type = open_fit ([qualifiers])`
See Also: `register_slang_optimizer, fit_fun, load_data, set_par, set_kernel`

This function is primarily useful for implementing optimization methods in S-Lang. It returns a structure of the form

```
s = struct
{
   object, close, eval_statistic,
   status, statistic, num_vary, num_points,
   response_type, data_type
};
```

Two qualifiers are recognized. If present, the '`flux`' qualifier indicates that the fit statistic should be computed for flux-corrected data, otherwise the statistic will be computed for counts data. The '`response`' qualifier may be used to control which instrument response is applied to the spectral model; valid values are:

```
response=Ideal_ARF
         Ideal_RMF
```

```
Ideal_ARF | Ideal_RMF
Ideal_ARFRMF
Assigned_ARFRMF
```

The values of the 'flux' and 'response' qualifiers are stored in the `data_type` and `response_type` fields of the fit object structure.

The `object` field of this structure is an opaque `Fit_Object_Type` pointer to an internal data structure that represents the current state of the fit engine, including the current fit-function and all noticed datasets. To generate this opaque object pointer, isis performs all the initialization normally associated with each call to `eval_counts` or `eval_flux`.

The `eval_statistic` field provides a function that may be used to compute the current fit statistic for a given vector of free parameters. For example:

```
s = open_fit ();
stat = s.eval_statistic (pars);
```

Performing this statistic computation using the fit object is more efficient than performing the same computation using `eval_counts` or `eval_flux` because the initialization associated with the `Fit_Object_Type` pointer in the `object` field has already been performed. If the `eval_statistic` method is called with the `nocopy` qualifier, some internal copying of computed model values will be omitted. While skipping this internal copying provides a useful optimization during the intermediate steps of a fit, the final statistic computation should be performed without the qualifier so that the final computed model spectra are saved internally.

The remaining fields of the fit object structure are scalar values containing results of the fit statistic computation. The `statistic` field contains the value of the fit statistic. The `num_vary` field contains the number of variable fit parameters. The `num_points` field contains the number of data points involved in the statistic computation. The `status` field is negative if any error occurred, and zero otherwise.

pileup

Purpose: Pileup kernel used to model photon pileup in CCDs
Usage: `set_kernel (data_index, "pileup[;option=value;...]")`
See Also: `set_kernel, fit_fun`

The pileup kernel includes the effects due to event pileup within a single CCD frame-time and is described in detail in Davis (2001).

Valid options for the pileup kernel include:

```
  nterms=value     Max number of piled photons
fracexpo=value     Fraction exposure for ARF
 verbose=value     verbose level
```

See the printed manual for a brief usage example.

The pileup kernel provides an implementation of the `print_kernel` function to provide statistics on the nature of the pileup fit.

For example:

```
isis> print_kernel(1);
1: 0.211913     0.929297
2: 0.029725     0.0673295
3: 0.00277968   0.00325211
4: 0.000194952  0.000117811
5: 1.09384e-05  3.41426e-06
*** pileup fraction: 0.0707029
```

This says that 21% of the frames contained a single photon in the pileup region, 3 percent were contained 2 photons, etc. The 3rd column indicates that 93 percent of the events were single photon events, 7 percent were due to 2 photons, etc.

The ACIS frame time in seconds is determined as follows. If the `EXPTIME` keyword is present in the spectrum file header, its value is taken as the frame time; otherwise, if the `TIMEDEL` keyword is present, the frame time is taken as `TIMEDEL-0.04104`. If neither of these keywords is present, the frame time is assumed to be 3.2 sec. Use `set\_frame\_time` to specify the frame time explicitly.

The most computationally expensive aspect of the pileup model involves convolution of real-valued sequences using the FFT. By default, ISIS uses an FFT algorithm which is very general but not particularly fast. If you plan to make heavy use of the pileup kernel, you may wish to obtain the specialized and highly optimized FFT library, `djbfft`, available from `http://cr.yp.to/djbfft.html`. To use this library, first download and install it, then install ISIS using the `--with-djbfft` configure option to supply the path to the `djbfft` library. Informal benchmarks on an AMD Athlon cpu show that the pileup model using the `djbfft` library runs about five times faster than with the default FFT.

poly

Purpose: polynomial fit function
Usage: `poly(id)`
See Also: `gauss, Lorentz, delta, bin_width, bin_center`

The `id` parameter identifies a particular instance of a polynomial; multiple instances are allowed in a single fit. The function value assigned to each bin is the area under the polynomial curve (of order ≤ 2) which lies inside the bin:

$$\begin{aligned} \mathrm{poly}(x_a, x_b) &= \int_{x_a}^{x_b} \mathrm{d}x \, \left(a_0 + a_1 x + a_2 x^2\right) \\ &= \delta x \left[a_0 + \frac{a_1}{2}\left(x_b + x_a\right) + \frac{a_2}{3}\left(x_b^2 + x_b x_a + x_a^2\right)\right] \end{aligned} \tag{7.47}$$

where $\delta x \equiv x_b - x_a$. Notice that $\mathrm{poly}(x_a, x_b)$ has units of area (e.g. photons/s/cm^2/*bin*), consistent with the definition of the other fit functions `gauss` and `Lorentz`), so that a_0 has units of a density (e.g. photons/s/cm^2/Å). By setting the coefficients appropriately, this function can also serve as both a linear function and a constant.

powell

Purpose: Powell minimization algorithm
Usage: `set_fit_method ("powell")`
See Also: `optimization, set_fit_method, set_fit_constraint`

The powell algorithm is an implementation of Powell's direction set method described in the paper: "An efficient method for finding the minimum of a function of several variables without calculating derivatives", The Computer Journal 1964. This implementation also makes use of modifications suggested by W.I. Zangwill, "Minimizing a function without derivatives", Computer Journal 1967

For help, use:

```
set_fit_method ("powell;help");
```

Powerlaw

Purpose: power-law fit function
Usage: `Powerlaw (id)`
See Also: `gauss, Lorentz, add_slang_function, add_compiled_function`

The `id` parameter identifies a particular instance of a power-law; multiple instances are allowed in a single fit. The function value assigned to each bin is the area under the power-law curve which lies inside the bin:

$$\text{Powerlaw}(E_1, E_2) = \int_{E_1}^{E_2} \text{d}e \; Ae^{\alpha} \tag{7.48}$$

for e in keV. Notice that Powerlaw(E_1, E_2) has units of area (e.g. photons/s/cm^2/bin), consistent with the definition of the other fit functions `gauss` and `Lorentz`).

plot_conf

Purpose: Plot 2D chi-square confidence contours
Usage: `[o]plot_conf (Struct_Type[, line[, dchisqr_array]])`
See Also: `save_conf, load_conf, conf_map_counts, conf_map_flux`

By default, this function plots contours at $\delta\chi^2 = 2.30$, 4.61 and 9.21, corresponding to 1-sigma (68.3% confidence), 90% confidence, and 3-sigma (99% confidence) respectively. To plot different contours, supply their delta-chisqr values using the optional array argument. The first argument is the return value of `conf_map_counts` or `conf_map_flux`. If not otherwise specified, the axis limits of these confidence contour plots are determined by the `Struct_Type` arguments used to generate the confidence map. The current implementation does not support plotting sub-regions of confidence maps.

The first optional argument is a structure which specifies the line style and has the form

```
line = struct {width, type, color};
```

where each struct field may be either a scalar or an array of length equal to the number of plot contours; the contours are in order from lowest to highest confidence. See the `linestyle` for a list of supported values of `line.type`.

To overlay contours on an existing confidence contour plot, use `oplot_conf`.

```
Example:
     % generate conf. contours for dataset 1
     s1 = conf_map_counts (px, py);

     % generate conf. contours for dataset 2
     s2 = conf_map_counts (px2, py2);

     % provide X-Y ranges big enough to span both sets
     % of contours
     xrange (xmn, xmx);
     yrange (ymn, ymx);

     % overlay conf. contours:
     plot_conf (s1);
     oplot_conf (s2);
```

print_kernel

Purpose: Print kernel parameters
Usage: `print_kernel (hist_index_list)`
See Also: `load_kernel, set_kernel`

This function prints the current value of parameters associated with the fit-kernel for each dataset in `hist_index_list`. Although the standard fit-kernel has no parameters, the pileup-kernel does have a number of parameters. User-defined fit kernels may provide this functionality as well – see `src/pileup_kernel.c` in the ISIS distribution for an implementation example. See also §4.11.

randomize

Purpose: Randomize variable fit-parameters
Usage: `randomize ([params])`
See Also: `set_par, get_par, fit_counts, fit_flux, fit_search`

For each variable fit-parameter, this function randomly selects a new value from within the specified `[min, max)` range. Note that if no parameter range has been specified, the parameter is unconstrained and a random value will be selected within the range $(-\infty, +\infty)$ – this is probably not what you want.

register_slang_optimizer

Purpose: Register an optimization method implemented in S-Lang
Usage: `register_slang_optimizer (name, &method [;qualifiers])`
See Also: `load_fit_method`

Use this function to register an optimization method implemented in S-Lang. For example:

```
private variable Qualfiers;
define set_options (options)
{
   Qualfiers = options;
}

define some_function (obj, params, params_min, params_max)
{
   %... search for optimal parameters...
        statistic = __eval_stat (obj, pars);
   %...
   return new_params;
}

register_slang_optimizer ("name", &some_function;
                                   set_options=&set_options);

set_fit_method ("name; option1; option2=2.4;  option3=astring");
```

The first argument to `register_slang_optimizer` provides a name for the optimization method. The second argument provides a reference to the S-Lang function implementing the method.

Two qualifiers are supported. The '`stat_name`' qualifier gives the name of the fit statistic that will be used by default. If not specified, the default fit-statistic is `chisqr`.

If present, the '`set_options`' qualifier should provide a reference to a function that takes one argument, which is a `Struct_Type` containing options from the `set_fit_method` call. In the above example, the options struct will have the form

```
options = struct {option1, option2=2.4, option3="astring"};
```

renorm_counts

Purpose: Automatically adjust fit normalization
Usage: `s = renorm_counts ()`
See Also: `renorm_flux`

This function automatically adjusts the normalization of the model to improve the fit to the counts data. See `fit_counts` for details.

renorm_flux

Purpose: Automatically adjust fit normalization
Usage: `s = renorm_flux ()`
See Also: `renorm_counts`

This function automatically adjusts the normalization of the model to improve the fit to the flux-corrected data. See `fit_flux` for details.

save_conf

Purpose: Save a 2D chi-square map as a FITS image
Usage: `status = save_conf (Struct_Type, file)`
See Also: `conf_map_counts, load_conf, plot_conf`

This function saves a 2D confidence map as a FITS image with WCS coordinates defined using the corresponding grid of fit-parameter values. The return value is 0 for success, < 0 for failure.

For example,

```
% Create and save a confidence map:
   map = conf_map_counts (px, py);
   status = save_conf (map, "map.fits");
```

See `conf_map_counts` for details.

save_par

Purpose: save fit function and parameters to a file
Usage: `save_par ("filename")`
See Also: `load_par, list_par, edit_par, set_par, get_par, fit_search`

The current fit function and parameter values may be saved in an ASCII file. The allowed fit-ranges, and freeze/thaw/tie state is also saved. See `list_par` for format details. `load_par` may be used to re-load the file.

If the function `isis_save_par_hook` is defined in the Global namespace, it will be called after the parameters have been written out, but before the file is closed. The name of the parameter file will be passed to `isis_save_par_hook`, which should return a string. If non-empty, this string will be appended to the parameter file.

This trivial example would record time of day in the parameter file:

```
public define isis_save_par_hook (filename)
{
    return time();
}
```

`isis_save_par_hook` is probably most useful in conjunction with `fit_search` and `fit_search_info`. In that case, it might be used to automatically collect fit results into a table with a custom format.

For example:

```
define value_string (p)
{
   return sprintf ("%13.6e", p.value);
}

define param_values_string ()
```

```
{
   variable p, s;

   p = get_params ();
   if (p == NULL) return "";

   s = array_map (String_Type, &value_string, p);

   return strjoin (s, " ");
}

public define isis_save_par_hook (fname)
{
   variable s, fp, info, stat = 0.0;

   % retrieve info on current best fit.

   info = fit_search_info();
   if (info != NULL)
     stat = info.statistic;

   % generate a string 's'

   variable v = param_values_string ();
   s = sprintf ("%s %12.4e  %s\n", fname, stat, v);

   % append 's' to a log file
   variable dir = path_dirname (fname);

   fp = fopen (dir + "/save_par.log", "a");
   if (fp == NULL) return NULL;
   () = fputs (s, fp);
   () = fclose (fp);

   % the returned string will be appended to the param file.

   return "";
}
```

set_fit_constraint

Purpose: Impose a fit-constraint
Usage: `set_fit_constraint (&ref [, param_names[])`
See Also: `set_fit_method, set_fit_statistic, fit_fun`

In order to impose constraints on a fit in addition to the usual bound-constraints on the fit-parameters (e.g. to impose Lagrange multiplier constraints), one can provide a function that modifies the fit-statistic computed during the fit. This function may depend upon the fit statistic, any of the data values, model parameters, or computed model values and may also depend upon additional fit-parameters associated with the constraint function itself.

Use `set_fit_constraint` to provide a reference to the constraint function and, if necessary, to provide an array of strings giving the names of fit-parameters that are associated with the constraint function. The parameters named in this array will appear as fit-parameters of the form `constraint(1).name`.

The constraint function itself should accept two parameters and should return a scalar-valued penalty. The first input parameter is the scalar-valued fit statistic. The second input parameter is an array of parameter values. If the constraint function has no parameters, the second input parameter will be set to `NULL`.

For example:

```
define a_constraint (stat, pars)
{
    variable lam1, lam2, penalty;

    lam1 = par[0];
    lam2 = par[1];

    penalty = lam1 * fcn1() + lam2 * fcn2();

    return stat + penalty;
}
set_fit_constraint (&a_constraint, ["lambda1", "lambda2"]);
```

Note that, if the constraint function is defined in the `Global` namespace or in the `isis` namespace, it may not have the name '`constraint`'. The name '`constraint`' may be used only if the function is declared to be private:

```
private define constraint (stat, pars)
{
   % ... compute the penalty ...
   return stat + penalty;
}
```

To remove the fit-constraint use

```
set_fit_constraint (NULL);
```

set_fit_method

Purpose: select a fit-method
Usage: `set_fit_method ("name[;options]")`
See Also: `load_fit_method`, `register_slang_optimizer`, `set_fit_statistic`, `set_fit_constraint`, `randomize`, `optimization`

The ISIS distribution includes a number of optimization methods (see `optimization`). Control parameters associated with the fit methods (such as convergence tolerances) may be adjusted by supplying qualifers in the method name string. For example, one might set the maximum number of function evaluations performed by `subplex` by setting the `maxnfe` option using

```
set_fit_method ("subplex;maxnfe=1000");
```

Similarly, one can limit the maximum number of iterations performed by `marquardt` using

```
set_fit_method ("marquardt;max_loops=100");
```

For a given fit method, the list of available control parameters may be obtained using the `help` qualifier:

```
set_fit_method ("marquardt;help");
```

Default parameter values may be restored using the `default` qualifier:

```
set_fit_method ("minim;default");
```

Given a reasonably good initial guess and a reasonably smooth χ^2 space, `mpfit` generally converges to the best fit fairly quickly. In the ideal case, its convergence is quadratic. In cases where the χ^2 parameter space is relatively complex (e.g. pileup), with many local minima, `subplex` may be helpful because it is somewhat less likely to become stuck in a local minimum. One possible strategy may be to use `subplex` to find the neighborhood of the global minimum and then use `mpfit` to quickly find the best fit.

In general, it is a good idea to thoroughly search the parameter space to find the best fit. The `randomize` function may be helpful as part of a scripted Monte-Carlo search of the likely parameter space (see also `fit_search`).

set_fit_range_hook

Purpose: Control allowed fit-parameter value range
Usage: `set_fit_range_hook (&function)`
See Also: `set_fit_method, set_fit_constraint`

When fitting models to data, fit-parameters may be constrained to fall within a specified range. Each fit-method may supply its own algorithm for enforcing these parameter ranges. To supply an alternative algorithm, one can supply a S-Lang "range-hook" function of the form

```
(new_par, new_min, new_max) = range_hook (par, min, max, idx)
```

The arguments `par`, `min` and `max` give the parameter values and min/max ranges while the last argument, `idx`, gives the index of each parameter as shown by `list_par`. The `idx` argument is necessary because the range-hook is passed only those parameters which are allowed to vary during the fit. Frozen and tied parameters are not passed.

To switch to the new range function, use e.g.

```
set_fit_range_hook (&range_hook);
```

to revert to the default fit-range algorithm, use

```
set_fit_range_hook (NULL);
```

Note that the range-hook function can also adjust the allowed parameter value range (as well as the parameter value itself) during the fit iteration.

For example, the following function enforces the allowed parameter range by setting out-of-range parameters to a randomly chosen value from within the allowed range:

```
define enforce_ranges (p, pmin, pmax, idx)
{
   variable i, out_of_range;
   out_of_range = where (p < pmin or pmax < p);

   foreach (out_of_range)
   {
       i = ();
       p[i] = pmin[i] + urand() * (pmax[i] - pmin[i]);
   }

   return (p, pmin, pmax);
}

set_fit_range_hook (&enforce_ranges);
```

set_fit_statistic

Purpose: select a fit-statistic
Usage: `set_fit_statistic ("name")`
See Also: `load_fit_statistic, set_fit_constraint, set_fit_method`

ISIS includes three built-in fit statistics, `chisqr`, `cash` and `ml`. User-defined fit-statistics are also supported and may be implemented in `C` or S-Lang. Control parameters associated with the fit statistics (such as the chi-square weighting) may be adjusted by supplying qualifers in the method name string. For a given fit statistic, the list of available control parameters may be obtained using the `help` qualifier:

```
set_fit_statistic ("chisqr;help");
```

For example, one might set the variance used with the chi-square statistic `chisqr` by setting the `sigma` option using

```
set_fit_statistic ("chisqr;sigma=gehrels");
```

Given data values $C_i \pm \sigma_i$ and model values M_i, the chi-square statistic is defined to be

$$\chi^2 = \sum (C_i - M_i)^2 / \sigma_i^2 \tag{7.49}$$

where the definition of σ_i is one of the following:

```
   data    \sigma_i [default]
gehrels    1 + \sqrt(C_i + 0.75)
  model    \sqrt(M_i)
    lsq    1
```

The Cash statistic is defined to be

$$S_{\text{cash}} \equiv 2 \sum_i (M_i - C_i) + C_i \ln \left(\frac{C_i}{M_i} \right). \tag{7.50}$$

See Cash (ApJ 228, 939) and the XSPEC manual for details. The max-likelihood (ML) statistic is defined to be

$$S_{\rm ml} \equiv 2 \sum_i \left(M_i + \ln \Gamma(C_i + 1) - C_i \ln M_i \right) \tag{7.51}$$

set_function_category

Purpose: Specify the category to which a fit-function belongs
Usage: `set_function_category (name, category)`
See Also: `add_slang_function, add_compiled_function`

Two categories of fit-functions are currently supported:

```
___Category____       __Definition__
 ISIS_FUN_ADDMUL      additive and multiplicative models
 ISIS_FUN_OPERATOR    operator or ``convolution'' models
```

The default category is `ISIS_FUN_ADDMUL`.

See `add_slang_function` and `add_compiled_function` for details on how to define the various types of fit-functions.

set_kernel

Purpose: Specify fit-kernel to use in forward-folding
Usage: `set_kernel (hist_index_list, "kernel_name[;option=value;..]")`
See Also: `load_kernel, print_kernel, get_kernel, list_kernels`

The default fit kernel, equivalent to applying the ARF and RMF in the usual way, is called "std". If the ARF and RMF are unavailable, the response defaults to an identity matrix. The pileup kernel, which includes the effects due to event pileup within a single CCD frame-time and described in detail in Davis (2001), is called "pileup" (See also §4.11).

The first argument, `hist_index_list`, is a list of data set indices which should use this kernel. The second argument, `kernel_name`, is the name of the kernel, which should be `<= 31` characters. This argument may also contain values for kernel-specific options; for example:

```
set_kernel (3, "pileup;nterms=20;fracexpo=0.9");
```

To determine what kernel options are supported, use the `help` option. For example:

```
isis> set_kernel (1, "pileup;help");
Valid options for subsystem "pileup" include:
    nterms=value     Max number of piled photons
  fracexpo=value     Fraction exposure for ARF
   verbose=value     verbose level
isis>
```

Alternate fit-kernels may be defined by the user by creating a shared library (.so file) with the necessary interface. This shared library may be loaded using the `load_kernel` function.

```
Example:
   % use the pileup kernel for data sets 4-6:
   set_kernel ([4,5,6], "pileup");

   % revert to the standard kernel for data set 5
   set_kernel (5, "std");
   set_kernel (5, NULL);     % this form is also valid
```

__set_fitfun_post_hook

Purpose: Set a hook to be called after evaluating a model component
Usage: `__set_fitfun_post_hook (String_Type fitfun_name, Ref_Type hook)`
See Also: `__set_fitfun_trace_hook`

```
Example:
   __set_fitfun_post_hook ("warmabs", &post_hook);
```

The hook function must be of the form:

```
  define post_hook (fitfun_name, id)
  {
  }
```

where `fun_name` is the name of the fit-function just evaluated and `id` is the instance of that fit-function. For example, if the hook is called after `warmabs(2)` is evaluated, then `fun_name` will have the value `"warmabs"` and `id` will have the value 2.

__set_fitfun_trace_hook

Purpose: Set a hook to be called before evaluating a model component
Usage: `__set_fitfun_trace_hook (String_Type fitfun_name, Ref_Type hook)`
See Also: `__set_fitfun_post_hook`

```
Example:
   __set_fitfun_trace_hook ("warmabs", &trace_hook);
```

The hook function must be of the form:

```
  define trace_hook (id, params)
  {
  }
```

where `id` is the instance of the fit-function that is about to be called. For example, if the hook is called before `warmabs(2)` is evaluated, then `id` will have the value 2 and `params` will contain a `Double_Type` array of all the parameters being passed to the function.

__set_hard_limits

Purpose: Adjust the hard limits constraining a fit parameter's value
Usage: `__set_hard_limits (fun_name, par_name, hard_min, hard_max)`
`__set_hard_limits (index, hard_min, hard_max)`
See Also: `set_par`

Because hard limits of a parameter are intended to provide a way for the author of a model to define the full range of parameter values for which the model is applicable, it should not be necessary for the user to modify the hard limits at all.

Unfortunately, hard limits are sometimes carelessly set to an overly restrictive range. For example, hard limits may constrain a Doppler shift parameter to always correspond to a redshift even though the same code can perfectly well handle a blueshift.

Use `__set_hard_limits` to change the hard limits for a given parameter.

This change can be applied either to a specific instance of a function or to the default configuration of a function. If the default configuration is modified, all subsequent instances of the function will be affected.

To change the hard limits for a specific instance of a function:

```
__set_hard_limits ("mekal(1).redshift", -10, 10);
```

To change the hard limits for the default configuration of a function:

```
__set_hard_limits ("mekal", "redshift", -10, 10);
```

If the specified hard limits do not contain the current parameter value and its soft limits, then consistent values for the parameter and its soft limits must also be provided using the `parm` qualifier. The `parm` qualifier is a numerical array of length 3. If present, the array will be sorted in increasing order and then the three elements will be used as the new `min`, `value`, `max`. For example:

```
__set_hard_limits ("mekal(1)", "redshift", -10, 0; parm=[-3, -2, -1]);
```

set_par

Purpose: set the value of a fit parameter
Usage: `set_par (idx [, value [, freeze, [ min, max]]])`
See Also: `edit_par, load_par, get_par, set_par_fun, __set_hard_limits, tie, set_params`

```
      idx    parameter index or string identifier
    value    parameter value
   freeze    [optional] 1(0) if parameter is(is not) frozen
 min, max    [optional] allowed range for parameter value
```

```
Qualifiers:
     step=VAL    initial parameter absolute step size
  relstep=VAL    initial parameter relative step size
      min=VALUE  parameter min value
      max=VALUE  parameter max value
```

If allowed ranges are not specified by the user or if `min`=`max`=0, the parameter value range is unlimited; this is the default. If the function provides hard limits on the parameter value range, then the `min`, `max` values must be consistent with those hard limits (see `set_hard_limits`).

One can also refer to parameters by name:

```
  set_par ("gauss(1).area", 47.0);
```

Parameters may also be identified using expressions with embedded wildcard characters. Both S-Lang regular expressions and globbing expressions are supported. If a regular expression string is to be used, then the string must either begin with '`^`' or end with '`$`'. A globbing expression is frequently used for matching filenames where '`?`' represents a single character and '`*`' represents 0 or more characters. For example, consider a model with many instances of similar parameter names, such as:

```
 "xpileup_n(1).G0_0"
 "xpileup_n(1).G0_1"
 "xpileup_n(2).G0_0"
 "xpileup_n(2).G0_1"
     .
     .
```

Using a globbing expression, one can use `set_par` to set all the `G0_0` parameters to a specified value, e.g.,

```
 isis> set_par ("xpileup_n(?).G0_0", 0.5);
 isis> set_par ("*.G0_1", 2.1);
```

S-Lang regular expression equivalents of the above are:

```
 isis> set_par ("^xpileup_n(.)\\.G0_0$", 0.5);
 isis> set_par (".*G0_1$", 2.1);
```

(Note the presence of `^` or `$` to indicate the string represents a regular expression).

Array arguments are supported. Consider:

```
isis> fit_fun ("gauss(1)");
isis> list_par;
gauss(1)
 idx  param         tie-to  freeze   value          min          max
  1  gauss(1).area     0     0           1            0            0
  2  gauss(1).center   0     0          12            0            0
  3  gauss(1).sigma    0     0       0.025            0            0
isis> set_par ("gauss(1)", [ 2, 8,  0.03], [ 1,  0,   1],
                           [-1, 3, 0.004], [10, 20, 0.3]);
isis> list_par;
```

```
gauss(1)
 idx  param         tie-to  freeze   value         min          max
  1  gauss(1).area     0     1          2           -1           10
  2  gauss(1).center   0     0          8            3           20
  3  gauss(1).sigma    0     1       0.03        0.004          0.3
isis>
```

The function value may be specified as a function of other fit-parameter values. For example:

```
  set_par ("gauss(1).center", "gauss(2).center-3.0");
```

For details, see `set_par_fun`.

If parameter ranges have been specified and a value is provided that falls outside those ranges, the default behavior is to interrupt the S-Lang interpreter and print an error message. For example:

```
isis> set_par (1, 1, 0, 0, 1);
isis> set_par (1,3);
*** Error:  param 1 not set:  value 3 lies outside [min, max] interval [0, 1]
S-Lang Error: Intrinsic Error: Error while executing _set_par
isis>
```

In long-running scripts this behavior may be undesirable. An alternative behavior may be specified by defining a function named `isis_set_par_hook` in the Global namespace. When the value range error occurs, if this function exists, it is passed a struct containing information on the relevant parameter. This function may then return the struct after modifying the range or value fields so that the `set_par()` operation can complete successfully. For example, one might adjust the parameter range as follows:

```
public define isis_set_par_hook (p)
{
   if (p.value <= p.min) p.min = 0.5 * p.value;
   if (p.value >= p.max) p.max = 2.0 * p.value;
   return p;
}
```

Depending on which optimization algorithm is used, the initial parameter `step/relstep` value may or may not be used.

set_param_default_hook

Purpose: Specify a function to set default parameter values and ranges
Usage: `set_param_default_hook (function_name, hook [,args])`
See Also: `add_slang_function, set_function_category`

For a fit-function implemented in S-Lang, one can define parameter defaults by providing an associated hook function. The hook function should be a S-Lang function of the form

```
    define param_default (i [,args])
```

where the first parameter is the zero-based array index and `args` is an optional list of user-defined arguments. The optional arguments are specified when the hook function is defined. The function should return the default values in a structure of the form

```
struct {value, freeze, min, max, hard_min, hard_max, step, relstep}
```

where `value` is the default parameter value and `min` and `max` define the user-adjustable parameter range. The user-adjustable parameter range must fall within the hard limits defined by `hard_min` and `hard_max`. The initial parameter absolute step size, `step`, and relative step size, `relstep`, may used by some optimization methods. `freeze` is a boolean value, non-zero if the parameter is frozen by default.

To specify the hook function to be used with a given fit-function, provide either a reference to the hook function (a S-Lang `Ref_Type`)

```
set_param_default_hook ("plaw", &plaw_param_default [,args]);
```

or the name of the hook function (a S-Lang `String_Type`)

```
set_param_default_hook ("plaw", "plaw_param_default" [,args]);
```

For a power-law fit-function, one might construct such a parameter-default function as follows:

```
define set_default (value, freeze, lim, hard_lim, step)
{
   variable x = struct {value, freeze, min, max,
                        hard_min, hard_max, step, relstep};
   x.value = value;
   x.freeze = freeze;
   x.min = lim[0];
   x.max = lim[1];
   x.hard_min = hard_lim[0];
   x.hard_max = hard_lim[1];
   x.step = step;
   x.relstep = 1.e-5;

   return x;
}

variable Defaults = Struct_Type[2];
Defaults[0] = set_default (1.0, 0, [0.0, DOUBLE_MAX], [0.0, _Inf], 0);
Defaults[1] = set_default (  0, 0, [-5.0, 1.0], [-10.0, 10.0], 0);

define plaw_default_hook (i)
{
   return Defaults[i];
}
set_param_default_hook ("plaw", &plaw_default_hook);
```

In this example, the normalization is constrained to be positive, but has no default upper-limit. The power-law exponent is, however, constrained to fall in the range [-5,1].

set_params

Purpose: Reset all fit-parameter information using an array of structs
Usage: `set_params (Struct_Type[])`
See Also: `get_params, set_par, tie`

See `get_params` for more information.

set_par_fun

Purpose: Define a fit-parameter value using a function
Usage: `set_par_fun (idx, expression_string)`
See Also: `set_par, get_par_info, _par, fit_fun, conf_map_counts`

Fit parameters may be defined as functions of other fit-parameters and may also depend on arbitrary functions. If the computed parameter value falls outside that parameter's `min/max` range, then the out of range value will be ignored; a warning message may be printed, depending on the context and the current verbosity level (see `Isis_Verbose`).

This has a number of applications and can be used to couple fit-parameters in complex ways, derive quantities of interest from fit-parameters, compute confidence limits on derived quantities and also to introduce arbitrary coordinate transformations in e.g. plots of confidence contours.

For example, to fit a sum of two Gaussians centered 4.3 Angstroms apart use:

```
isis> fit_fun ("gauss(1) + gauss(2)");
isis> set_par_fun ("gauss(2).center", "gauss(1).center + 4.3");
isis> list_par;
gauss(1) + gauss(2)
 idx  param        tie-to  freeze     value        min       max
  1  gauss(1).area     0     0            1          0         0
  2  gauss(1).center   0     0           12          0         0
  3  gauss(1).sigma    0     0        0.025          0         0
  4  gauss(2).area     0     0          2.5          0         0
  5  gauss(2).center   0     1         16.3          0         0
  #=>  gauss(1).center + 4.3
  6  gauss(2).sigma    0     0        0.025          0         0
```

The constraint expression may also contain user-defined functions. For example:

```
  public define offset ()
  {
     if (Isis_Active_Dataset == 2)
       return 3.0;
     else
       return 2.5;
  }
  set_par_fun ("gauss(2).center", "gauss(1).center + offset()");
```

Here, the ISIS intrinsic variable `Isis_Active_Dataset` is used to define an offset which has the value 3.0 for dataset 2, but is 2.5 for all other datasets.

To delete the parameter-function definition, use

```
set_par_fun (index, NULL);
```

Note that `list_par` shows that all parameters defined as functions have `freeze=1`. This does not mean that the parameter's value is fixed during fit iteration. It merely indicates that the parameter's value is derived from other parameters and is not an independent variable during the fit.

One can also use `set_par_fun` to enforce inequality constraints. The following example shows how to constrain one fit parameter to be smaller than another:

```
% Define a do-nothing fit function with a single parameter
define one_fit (lo, hi, par)
{
    return 1.0;
}
add_slang_function ("one", "c");

% Include this do-nothing function in your spectral model
fit_fun ("one(1) * phabs(1) * (mekal(1) + mekal(2))");

% With this model, we have
% param 1  = c (the 'dummy parameter')
% param 4  = kT_1
% param 10 = kT_2

% Now, introduce the inequality constraint
% to require param 4 <= param 10:

set_par_fun ("mekal(2).kT", "mekal(1).kT + one(1).c^2");
```

With this definition, the value of parameter 10 will be computed by adding a non-negative value, `c^2`, to parameter 4, thereby enforcing the constraint.

Although one can also refer to parameters by index:

```
set_par_fun ("gauss(1).area", "_par(4) + _par(1)^2");
```

using names of the form `fname(i).parname` is more robust, because such names are unaffected by changes in the parameter indexing.

simplex

Purpose: Simplex minimization algorithm
Usage: `set_fit_method ("simplex")`
See Also: `optimization, set_fit_method, set_fit_constraint`

The simplex algorithm is an implementation of the Nelder-Mead simplex method, a general method for solving unconstrained optimization problems.

For help, use:

```
set_fit_method ("simplex;help");
```

subplex

Purpose: Subplex minimization algorithm
Usage: `set_fit_method ("subplex")`
See Also: `optimization, set_fit_method, set_fit_constraint`

The subplex algorithm is a minimization algorithm used in fitting models to data. A variant of the Nelder-Mead simplex method, it is a general method for solving unconstrained optimization problems. It is well suited for optimizing objective functions that are noisy or are discontinuous at the solution. The implementation used in ISIS came from the Netlib repository; for details, see `www.netlib.org`.

The algorithm has a number of options:

```
   __Option__ __Default__  __Purpose__
       maxnfe   128N       Max number of function evaluations
          tol   1.e-4      Relative error tolerance
 scale_factor   1.0        Parameter scale factor
```

By default, the `maxnfe` is set to `128N` where `N` is the number of number of fit parameters.

`scale_factor` is used to scale and provide initial stepsizes for the simplex. For each parameter, the scale is defined to be `scale_factor` multiplied by the initial parameter value.

`tol` is the relative error tolerance on the parameter value.

thaw

Purpose: thaw one or more fit parameters
Usage: `thaw (par_list)`
See Also: `freeze, tie, untie`

`par_list` may be either a single parameter index or an integer array of indices. Thawing a parameter allows the parameter to vary when searching for the best fit to the data.

```
    thaw(3);           % thaw param 3
    thaw([1:4]);       % thaw params 1,2,3,4
```

One can also refer to parameters by name:

```
  thaw ("gauss(1).area", "gauss(2).area");
```

tie

Purpose: tie one or more fit parameters to a single parameter
Usage: `tie (par, par_list)`
See Also: `untie, freeze, thaw, set_params, set_par`

`par_list` may be either a single parameter index or an integer array of indices. Tieing one or more parameters to a variable parameter causes the tied parameters to vary as one. Tieing to a frozen parameter has the opposite effect.

```
tie(2,3);           % tie param 3 to the value of param 2
tie(5, [1:4]);      % tie params 1,2,3,4 to the value of
                    % param 5
```

One can also refer to parameters by name:

```
tie ("gauss(1).area", "gauss(2).area");
```

The values of tied parameters are updated only when the model is evaluated.

If `set_par` is used to modify a tied parameter, the parameter value is not changed and no warning or error message is generated. `set_params` can be used tie or untie parameters.

untie

Purpose: untie one or more fit parameters
Usage: `untie (par_list)`
See Also: `tie, freeze, thaw, set_params, set_par`

`par_list` may be either a single parameter index or an integer array of indices.

```
untie(2);          % untie param 2
untie([1:4]);      % untie params 1,2,3,4
```

One can also refer to parameters by name:

```
untie ("gauss(2).area");
```

voigt

Purpose: Voigt profile function
Usage: `voigt(id)`
See Also: `gauss, Lorentz`

The Voigt profile is the convolution of the natural resonance profile of an emitted line with the Maxwellian speed distribution. The natural resonance profile is a Lorentzian with full-width at half maximum, $\Gamma/4\pi$. The Maxwellian speed distribution is a Gaussian with velocity width $v_0 = (2kT/m)^{1/2}$, where m is the ion mass and T is the temperature.

The fit parameters are

```
norm      [photons/s/cm^2]    Normalization
energy    [keV]               Line-center energy
fwhm      [keV]               Gamma
vtherm    [km/s]              Maxwellian velocity width=sqrt(2kT/m)
```

The Voigt profile is unit-normalized so that the value of the `norm` parameter gives the area under the profile.

Note that the `fwhm` parameter name is misleading – `fwhm`= Γ but the true FWHM is $\Gamma/2\pi$.

xnotice

Purpose: Notice a wavelength range when fitting
Usage: `xnotice (hist_index_list [, lambda_lo, lambda_hi])`
See Also: `ignore, notice`

`hist_index_list` may be either a single histogram index or an integer array of indices. This command is equivalent to first ignoring the entire wavelength range and then noticing the specified range.

```
% first ignore all bins, then notice the specified range.
   isis> ignore(idx);
   isis> notice(idx, lambda_lo, lambda_hi);
```

Note that only the histograms explicitly mentioned in `hist_index_list` are affected; to ignore other histograms, use `ignore`.

Note that when fitting data using an ARF and RMF, the RMF is used to determine which model bins contribute to the noticed data bins.

xnotice_en

Purpose: Notice an energy range when fitting
Usage: `xnotice_en (hist_index_list [, E_lo, E_hi])`
See Also: `xnotice, ignore_en, notice_en`

This is an alternate form of the `xnotice` function that takes an energy range in keV instead of a wavelength range. See `xnotice` for details.

yshift

Purpose: Kernel for introducing a wavelength shift
Usage: `set_kernel (data_index, "yshift")`
See Also: `set_kernel, fit_fun`

This kernel adds a wavelength shift (not a Doppler shift!) to the model counts spectrum. Any associated background spectrum is not shifted. The size of the shift is a fittable parameter. Aside from the wavelength shift, this kernel performs the same forward-fold computation as the standard kernel. Flux-correction is not supported.

This kernel was added primarily to support analysis of dispersed spectra of marginally resolved point sources using `combine_datasets`.

7.8 Mult-Core Parallel Programming

ISIS provides a simple interface to support parallel programming on multi-core computers. This interface is specifically intended to support coordinating multiple instances of isis running on a single multi-core computer. To coordinate multiple isis processes running on different computers, use the `PVM` module instead (see `http://space.mit.edu/cxc/software/slang/modules/pvm/`).

Scripts using this parallel programming interface will normally be structured as follows:

```
variable s, slaves = new_slave_list ();
loop (num_slaves)
  {
     s = fork_slave (&task, args);
     append_slave (slaves, s);
  }
manage_slaves (slaves, &message_handler);
```

Interface details for the functions `task` and `message\_handler` are defined below.

`parallel`

Purpose: Controlling multi-core parallel processes
Usage: :
See Also: `parallel_map, fork_slave, manage_slaves`

A number of aspects of multi-core parallel processing can be controlled by setting fields in the `Isis_Slaves` structure. The fields of this structure are interpreted as follows:

```
__name__          __meaning__
num_slaves        Default number of slave processes to create
num_slave_slaves  Default maximum number of slave processes that a
                  slave process can create (default = 0).
nice              Default nice level or execution priority, in the
                  range 0-19, for slave processes (default = 0).
serial            Force computations to be performed on a single cpu.
```

These control values apply to all multi-core parallel functions.

Some of these parameters may be set temporarily by using qualifiers with individual function calls. For example, this:

```
(pmin, pmax) = conf_loop ([1:4]; num_slaves=2, nice=10);
```

runs `conf_loop` with two slaves at `nice=10` without modifying the values of the `Isis_Slaves` structure.

parallel_map

Purpose: Parallel analogue of array_map
Usage: [(Return_Values, ...) =] parallel_map ([Return_Types...,] &func, args, ... [; qualifiers])
See Also: fork_slave, manage_slaves, parallel

The interface for parallel_map is almost identical to the interface for array_map; see array_map for details. The primary difference between array_map and parallel_map is that parallel_map performs the computations in parallel using a number of CPUS.

```
EXAMPLE:
  define slow_function (x, y)
  {
     % ... perform expensive computation
     % yielding 2 results, a(x,y), q(x,y)...
     return a, q;
  }
  (A, Q)= parallel_map (Double_Type, Complex_Type, &slow_function, X, Y);

EXAMPLE:
  define compute_and_write_file (file, a, b)
  {
      % ...perform expensive computation
      % and save results to 'file'...
  }
  parallel_map (Void_Type, &compute_and_write_file,
                filename_array, A_array, B_array)
```

_num_cpus

Purpose: Determine the number of available CPUs
Usage: Integer_Type = _num_cpus ()
See Also: parallel_map, append_slave, fork_slave, manage_slaves

new_slave_list

Purpose: Create an empty list to manage slave processes
Usage: List_Type = new_slave_list ()
See Also: parallel_map, append_slave, fork_slave, manage_slaves

fork_slave

Purpose: Start a new isis processes running a specified function
Usage: Struct_Type = fork_slave (Ref_Type task [, args] [; qualifiers])
See Also: parallel_map, append_slave, manage_slaves, _num_cpus, send_msg, recv_msg, send_objs, recv_objs

```
Qualifiers:
```

```
  ctrl_c_kills_slaves   If present, then hitting ctrl-c will
                        immediately kill all slave processes
                        (even if the ctrl-c occurs outside of
                        the manage_slaves call).
```

This function creates a copy of the current isis process and executes the function referenced by `task` using the remaining parameters, if any. The `task` function should have the form

```
define task (Struct_Type slave_info [, arg1, arg2, ...] [; qualifiers])
{
   % ...
   return status;
}
```

The `slave_info` parameter has the form:

```
slave_info = struct {pid, sock, fp, status, data};

where:
 pid = slave process id (from getpid)
sock = a socket file descriptor (FD_Type) that may be
       used to communicate with the master process
  fp = a file pointer (File_Type) that may be used to
       communicate with the master process
data = a pointer to user-defined data.
```

Any additional arguments and qualifiers passed to `fork_slave` will also be passed to the `task` function. When this function returns, it should return an integer status value to indicate success (`status=0`) or failure (`status!=0`).

To run the new ISIS processes at lower priority, use the `nice` qualifier to provide a priority value in the range 0-19, with 19 being the lowest priority.

Note that the `manage_slaves` function should always be called at some point after slave processes are created by calls to `fork_slave`.

append_slave

Purpose: Append a slave structure to the list of managed slaves
Usage: `append_slave (List_Type, Struct_Type)`
See Also: `parallel_map, fork_slave, manage_slaves`

manage_slaves

Purpose: Manage a list of running slave processes
Usage: `manage_slaves (List_Type, Ref_Type message_handler [; qualifiers])`
See Also: `parallel_map, append_slave, _num_cpus, fork_slave, manage_slaves`

```
Qualifiers:
  while_=Ref_Type   Provide a reference to a boolean function that
                    will return zero to indicate when manage_slaves
```

```
                      should return, leaving all slave processes running.
                      Note that the condition tested normally requires
                      input from the slave processes.

  timeout=time_sec    Specify a maximum time-out interval for the
                      master process to use when waiting for input
                      from the slave processes.
                      By default, timeout=-1, meaning that the
                      master should wait indefinitely until input
                      from a slave appears. timeout=0 means
                      don't wait for input from the slaves.
                      timeout=t means wait at most `t'
                      seconds for input from the slaves.
                      For details, see the man page for `select'.
```

The `manage_slaves` function uses the provided message handler to manage communications between the master and slave processes until all the slave processes exit. If non-`NULL`, the message handler function should have the form:

```
define message_handler (slave, message)
{
}
```

The `slave` parameter is a `Struct_Type` like that returned by `fork_slave`. The `message` struct has the form:

```
message = struct {pid, type};
```

where `pid` is the process id of the slave that sent the message and `type` is an integer "message".

For example, a practical message handler might look like this:

```
define message_handler (s, msg)
{
   switch (msg.type)
   {case READY:         send_task (s); }
   {case HAVE_RESULT:   recv_result (s); }
   {case HAVE_PROBLEM:  handle_problems (s);}
}
```

The `send_task` function represents a user-defined function that might use the ISIS intrinsics `send_msg` and `send_objs` to send the parameters of a task to a slave process. Similarly, the `recv_result` function represents a user-defined function that might use the ISIS intrinsic `recv_msg` and `recv_objs` to receive data structures sent to the master by a slave process.

The message handler function reference may be `NULL` in a case where no master-slave communication is required. For example, the slave process may perform some computation and simply write the results out to disk. Note that the `manage_slaves` function should always be called after `fork_slave` is called, even if the message handler function reference is `NULL`. Otherwise, system data structures associated with the exiting slave processes will not be promptly released and the defunct slave processes will linger on as "zombies" (and you don't want that to happen, do you?).

In some situations, it is desirable to cause the `manage_slaves` function to return without waiting for all the slaves to exit. For example, suppose the computation of interest can be naturally partitioned into two parallel sections separated by a serial section, and that these three sections occur within a loop so that it would be useful to avoid the overhead of forking slaves numerous times.

The `while_` qualifier provides a reference to a function of the form

```
define while_function ()
{
   return boolean_flag;
}
```

This function is called by the master process each time it loops over the list of slave processes listening for messages. That loop will continue as long as slave processes are running and the `while_` function returns non-zero. When the `while_` function returns zero, that loop will end and the `manage_slaves` function will return, leaving the slave processes running.

For a detailed example of how the `while_` qualifier can be used to implement the three-section loop described above, look at the implementation of the `plm` fit-method (`isis/share/plm.sl`).

send_msg

Purpose: Send an integer message code to a slave process
Usage: `send_msg (Struct_Type slv, msgid)`
See Also: `recv_msg, send_objs, recv_objs, fork_slave, manage_slaves`

recv_msg

Purpose: Receive an integer message code from a slave process
Usage: `Integer_Type = recv_msg (Struct_Type slv)`
See Also: `send_objs, recv_msg, recv_objs, fork_slave, manage_slaves`

send_objs

Purpose: Send S-Lang objects to a slave process
Usage: `send_objs (Struct_Type slv, obj1 [, obj2, ...])`
See Also: `recv_objs, recv_msg, send_msg, fork_slave, manage_slaves`

Most S-Lang objects can be sent transparently to another process. However, a few S-Lang objects are not yet supported (arrays of `List_Type` and `Assoc_Type`, objects of `Ref_Type`).

recv_objs

Purpose: Receive S-Lang objects from a slave process
Usage: `List_Type = recv_objs (Struct_Type slv [, num])`
See Also: `send_objs, send_msg, recv_msg, fork_slave, manage_slaves`

A few examples may be sufficient to illustrate how to use this function. If one process executes:

```
send_objs (s, "a string", PI, 3, [1,2,3], {["a", "b"], urand(5)});
```

then, in the receiving process,

```
x = recv_objs (s);
```

will define x as a List_Type object with 5 entries:

```
x[0] = "a string";
x[1] = PI;
x[2] = 3;
x[3] = [1,2,3];
x[4] = List_Type {["a", "b"], Double_Type[5]}
```

The optional argument num is used to specify the number of objects to return. For example, in the above example,

```
x = recv_objs (s, 3);
```

would have yielded a List_Type with only 3 entries. The remaining two entries would remain in the communications buffer and could be retrieved by subsequent calls to recv_objs.

7.9 Low-level PGPLOT Interface

ISIS provides provides interactive access to most of the functions in the PGPLOT subroutine library. Using PGPLOT functions directly provides finer control over plot formatting and makes it possible to create complex, specialized plots.

7.10 Custom Plot Examples

The following script generates a plot called custom1.ps (see Figure 7.2)

```
variable pha2_file;
pha2_file = _isis_srcdir + "/../isis-examples/data/acisf01318N003_pha2.fits.gz";

variable id = 10;
() = load_data (pha2_file);
variable pid = open_plot ("custom1.ps/vcps");
resize(15);

xrange(10,20);
plot_data_counts (id);

% coordinates of region for inset plot:

variable xmin, xmax, ymin, ymax;
xmin = 12.35;
xmax = xmin + 0.2;
```

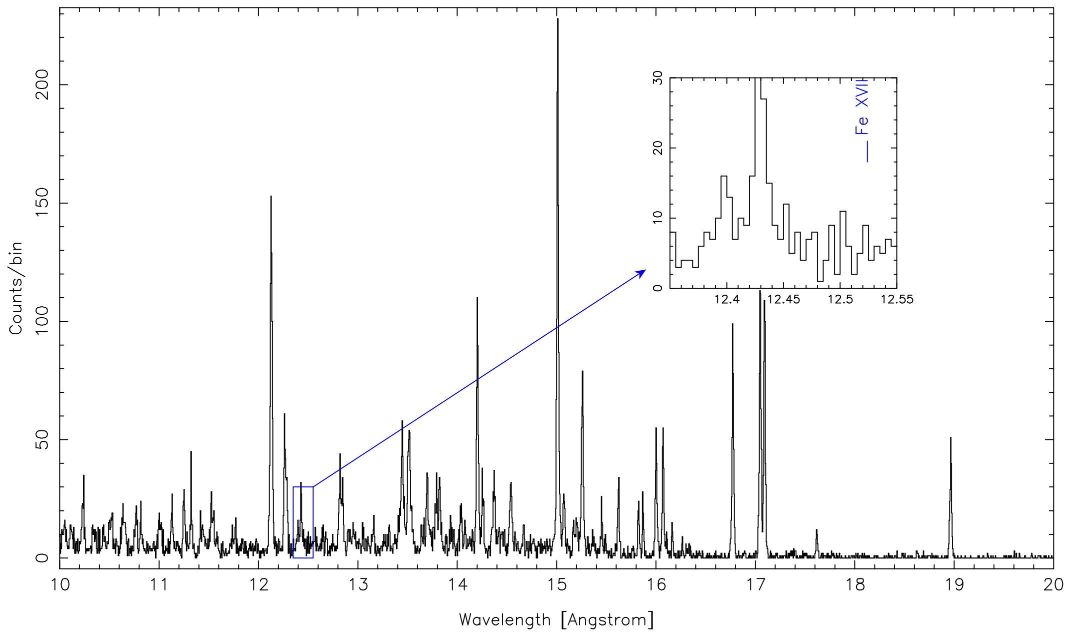

Figure 7.2: An example of plot customization using PGPLOT functions with ISIS intrinsic functions

```
ymin = 0.0;
ymax = 30.0;

_pgsci (4);                       % Draw blue rectangle:
_pgsfs (2);
_pgrect (xmin, xmax, ymin, ymax);
_pgsfs (1);                       % restore default fill-area style

_pgsci (4);                       % Draw blue arrow:
_pgsch (0.7);
_pgsah (1, 45.0, 0.3);            % specify arrow-head shape
_pgarro (xmax, ymax, 15.9, 122);  % from (x1,y1) to (x2,y2)
_pgsci (1);
                                  % New viewport in normalized
                                  % device coordinates:
_pgsvp (0.6, 0.8, 0.5, 0.8);      % xleft, xright, ybot, ytop
_pgswin (xmin, xmax, ymin, ymax);

                                  % Insert smaller plot
_pgsch (0.7);
_pgbox ("BCNST", 0.0, 1, "BCNST", 0.0, 1);
_pgsch (1.0);

variable d;
d = get_data(id);
_pgbin (d.bin_lo, d.value, 0);    % 0 means x value is bin left edge

plasma(aped);

charsize(0.7);
```

```
variable Fe = 26;
plot_group(where(wl(xmin, xmax) and el_ion(Fe,17)),4);
charsize(1.0);

close_plot(pid);
```

The following script generates a plot called `contour.ps` (see Figure 7.2)

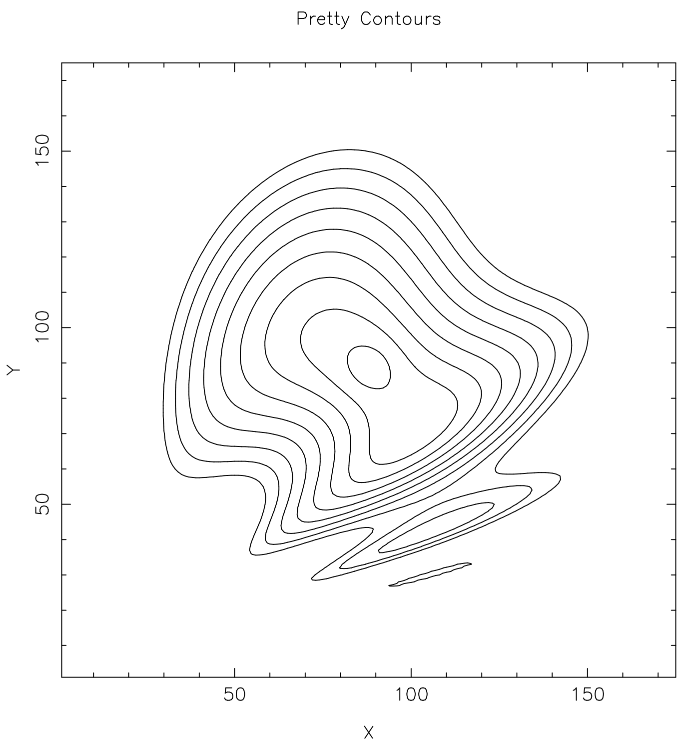

Figure 7.3: An example of contour plotting using PGPLOT functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PGPLOT has several contouring functions:
%
% PGCONB -- contour map of a 2D data array, with blanking
% PGCONF -- fill between two contours
% PGCONL -- label contour map of a 2D data array
% PGCONS -- contour map of a 2D data array (fast algorithm)
% PGCONT -- contour map of a 2D data array (contour-following)
% PGCONX -- contour map of a 2D data array (non rectangular)
%
%
% the corresponding ISIS wrappers are
% _pgconb, _pgconf, _pgconl, _pgcons, _pgcont
%
% At the moment, _pgconx isn't supported.

typedef struct
{
   min, max, num
}
Param_Type;

public define plot_contours (a, px, py, levels)
{
   % 1. define the plot coordinate system and draw the box
   % 2. plotting contours one at a time allows coloring
```

```
   %    them individually (see _pgsci()).

   _pgenv (px.min, px.max, py.min, py.max, 0, 0);
   _pgswin (1, px.num, 1, py.num);

   variable tr = [0, 1.0, 0, 0, 0, 1.0];
   variable i, n = length(levels);

   _for (0, n-1, 1)
     {
        i = ();
% negative number of contours means contour lines
% are drawn with the current line attributes
% (color index, style, width)
_pgcont (a, 1, 1, px.num, py.num, levels[i], tr, -1);
     }
}

public define fake_data (px, py)
{
   variable a = Float_Type [px.num, py.num];
   variable i, j, r, f;

   f = PI * py.max / px.max;
   for (i = 0; i < px.num; i++)
     {
for (j = 0; j < py.num; j++)
  {
     r = sqrt ( (i - px.num/2.0)^2 + (j - py.num/2.0)^2 );
     a[i,j] = r - 0.1 * px.num * (sin(i*f/(j+1.0)))^2;
  }
     }
   return a;
}

public define do_demo ()
{
   variable px = @Param_Type;
   variable py = @Param_Type;

   % Define axis ranges and resolution
   px.min = 1.0;
   px.max = 175.0;
   px.num = 175;

   py.min = 1.0;
   py.max = 175.0;
   py.num = 175;

   % Define contour levels
   variable levels = [0.0:50.0:5.0];

   plot_contours (fake_data (px, py), px, py, levels);
   _pglab ("X", "Y", "Pretty Contours");
```

```
}

() = _pgopen ("contour.ps/vcps");
_pgpap (4.0, 1.0);      % 4 inches, aspect ratio = 1.
do_demo ();
_pgclos();
```

7.11 Supported Functions

The PGPLOT functions are described in detail in the PGPLOT documentation; see

```
http://astro.caltech.edu/~tjp/pgplot/index.html
```

Here, we provide a condensed listing of the supported functions and the S-Lang syntax with which they may be invoked.

Almost all of the functions implemented here have exactly the same number of arguments as the PGPLOT library functions. The main difference is that array size arguments are not needed. Another difference is that temporary workspace arguments are handled transparently. For example, the library function `PGHI2D` takes 12 arguments. The corresponding S-Lang intrinsic _pghi2d takes only 9 arguments since the 2 of the 12 specify the dimensions of a 2D array, and 1 is a temporary workspace argument.

Finally, some functions that return values via the argument list are implemented as S-Lang functions that return multiple values. An example of this is _pgcurs which is prototyped in Fortran as:

```
PGCURS (X, Y, CH)
```

but used in S-Lang as:

```
(x,y,ch) = _pgcurs ();
```

Also see _pgband for another example.

Name	Description	S-Lang Syntax
PGARRO	draw an arrow	_pgarro (xfrom, yfrom, xto, yto)
PGASK	control new page prompting	_pgask (flag)
PGAXIS	draw an axis	_pgaxis (opt, x1, y1, x2, y2, v1, v2, step, nsub, dmajl, dmajr, fmin, disp, orient)
PGBAND	read cursor position, with anchor	_pgband (mode, posn, xref, yref, x, y, &xout, &yout, &ch)
PGBBUF	begin batch of output (buffer)	_pgbbuf ()
PGBIN	histogram of binned data	_pgbin (x, data, center)
PGBOX	draw labeled frame around viewport	_pgbox (xopt, xtic, nx, yopt, ytic, ny)
PGCIRC	draw a circle, using fill-area attributes	_pgcirc (x, y, a)
PGCLOS	close the selected graphics device	_pgclos ()
PGCONB	contour map of a 2D data array with blanking	_pgconb (data, i1, i2, j1, j2, c, tr, blank)
PGCONF	fill between two contours	_pgconf (data, i1, i2, j1, j2, c_lo, c_hi, tr)
PGCONL	label contour map of a 2D data array drawn by PGCONT	_pgconl (data, i1, i2, j1, j2, c, tr, label, intval, minint)
PGCONS	contour map of a 2D data array (fast algorithm)	_pgcons (data, i1, i2, j1, j2, c, tr, nc)
PGCONT	contour map of a 2D data array	_pgcont (data, i1, i2, j1, j2, c, tr, nc)
PGCTAB	install the color table to be used by PGIMAG	_pgctab (l, r, g, b, contra, bright)
PGCURS	read cursor position	(x, y, ch) = _pgcurs ()
PGDRAW	draw a line from the current pen position to a point	_pgdraw (x,y))
PGEBUF	end batch of output (buffer)	_pgebuf ()
PGEND	close all open graphics devices	_pgend ()
PGENV	set window and viewport and draw labeled frame	_pgenv (xmin, xmax, ymin, ymax, just, axis)
PGERAS	erase all graphics from current page	_pgeras ()
PGERR1	horizontal or vertical error bar	_pgerr1 (dir, x, y, e, t)
PGERRB	horizontal or vertical error bar	_pgerrb (x, y, e, t)
PGERRX	horizontal error bar	_pgerrx (x1, x2, y, len)
PGERRY	vertical error bar	_pgerry (x, y1, y2, len)
PGETXT	erase text from graphics display	_pgetxt ()
PGGRAY	gray-scale map of a 2D data array	_pggray (data, i1, i2, j1, j2, fg, bg, tr)
PGHI2D	cross-sections through a 2D data array	_pghi2d (data[][], ix1, ix2, iy1, iy2, x[], ioff, bias, center)
PGHIST	histogram of unbinned data	_pghist (dmin, dmax, nbins, flag)
PGIDEN	write username, date, and time at bottom of plot	_pgiden ()
PGIMAG	color image from a 2D data array	_pgimag (data, i1, i2, j1, j2, a1, a2, tr)
PGLAB	write labels for x-axis, y-axis, and top of plot	_pglab (xlabel, ylabel, toplabel)
PGLCUR	draw a line using the cursor	_pglcur (x, y)
PGLDEV	list available device types on standard output	_pgldev ()
PGLINE	draw a polyline (curve defined by line-segments)	_pgline (x, y)
PGMOVE	move pen (change current pen position)	_pgmove (x, y)
PGMTXT	write text at position relative to viewport	_pgmtxt (s, f, g, h, t)
PGNCUR	mark a set of points using the cursor	_pgncur (x, y, symbol)

Name	Description	S-Lang Syntax
PGOLIN	mark a set of points using the cursor	_pgolin (x, y, symbol)
PGOPEN	open a graphics device	id = _pgopen (dev)
PGPAGE	advance to new page	_pgpage ()
PGPANL	switch to a different panel on the view surface	_pgpanl (ix, iy)
PGPAP	change the size of the view surface	_pgpap (w, a)
PGPNTS	draw several graph markers, not all the same	_pgpt (x, y, symbol)
PGPOLY	draw a polygon, using fill-area attributes	_pgpoly (x, y)
PGPT	draw several graph markers	_pgpt (symbol)
PGPTXT	write text at arbitrary position and angle	_pgptxt (x, y, angle, just, text)
PGQAH	inquire arrowhead style	(fill_style, angle, barb_fraction) = _pgqah ()
PGQCH	inquire character height	h = _pgqch ()
PGQCIR	inquire color index range	(lo, hi) = _pgqcir ()
PGQCLP	inquire clipping status	clp = _pgqclp ()
PGQCOL	inquire color capability	(ci1, ci2) = _pgqcol ()
PGQHS	inquire hatching style	(angle, sep, phase) = _pgqhs ()
PGQITF	inquire image transfer function	itf = _pgqitf ()
PGQCF	inquire character font	cf = _pgqcf ()
PGQCI	inquire color index	ci = _pgqci ()
PGQCS	inquire character height in a variety of units	(xch, ych) = _pgqcs (units)
PGQFS	inquire fill-area style	fs = _pgqfs ()
PGQID	inquire current device identifier	id = _pgqid ()
PGQINF	inquire PGPLOT general information	value = _pgqinf (item)
PGQLS	inquire line style	ls = _pgqls ()
PGQLW	inquire line width	lw = _pgqlw ()
PGQNDT	inquire number of available device types	n = _pgqndt ()
PGQPOS	inquire current pen position	(x, y) = _pgqpos ()
PGQTBG	inquire text background color index	tbci = _pgqtbg ()
PGQVP	inquire viewport boundary coordinates	(xmin, xmax, ymin, ymax) = _pgqvp (units)
PGQVSZ	inquire size of view surface	(x1, x2, y1, y2) = _pgqvsz (units)
PGQWIN	inquire window boundary coordinates	(xmin, xmax, ymin, ymax) = _pgqwin ()
PGRECT	draw a rectangle, using fill-area attributes	_pgrect (xmin, xmax, ymin, ymax)
PGSAH	set arrow-head style	_pgsah (fs, angle, style)
PGSAVE	save PGPLOT attributes	_pgsave ()
PGSCF	set character font	_pgscf (ci)
PGSCH	set character height	_pgsch (ch)
PGSCI	set color index	_pgsci (ci)
PGSCR	assign RGB color to color index	_pgscr (ci, r, g, b)
PGSCL	scroll window	_pgscrl (dx, dy)
PGSCRN	assign RGB color to color index by name	ier = _pgscrn (ci, name)
PGSFS	set fill-area style	_pgsfs (fs)
PGSHLS	assign HLS color to color index	_pgshls (ci, h, l, s)
PGSHS	set hatching style	_pgshs (ang, s, p)
PGSITF	set image transfer function	_pgsitf (i)
PGSLCT	select an open graphics device	_pgslct (id)

Name	Description	S-Lang Syntax
PGSLS	set line style	_pgsls (ls)
PGSLW	set line width	_pgslw (lw)
PGSTBG	set text background color index	_pgstbg (ci)
PGSUBP	subdivide view surface into panels	_pgsubp (nx, ny)
PGSVP	set viewport (normalized device coordinates)	_pgsvp (xmin, xmax, ymin, ymax)
PGSWIN	set window	_pgswin (xmin, xmax, ymin, ymax)
PGTBOX	draw frame and write (DD) HH MM SS.S labeling	_pgtbox (xopt, tic, nx, yopt, ytic, ny)
PGTEXT	write text (horizontal, left-justified)	_pgtext (x, y, text)
PGUNSA	restore PGPLOT attributes	_pgunsa ()
PGUPDT	update display	_pgupdt ()
PGVSIZ	set viewport (inches)	_pgvsiz (xmin, xmax, ymin, ymax)
PGVSTD	set standard (default) viewport	_pgvstd ()
PGWNAD	set window and adjust viewport to same aspect ratio	_pgwnad (xmin, xmax, ymin, ymax)

Section 8

The XSPEC Module

ISIS is distributed with a module[1](see `modules/xspec`) which provides access to all of the additive, multiplicative and convolution models in XSPEC (Arnaud 1996). Table models and XSPEC-style local models are also supported. This module makes it possible to fit data using familiar XSPEC models and to easily manipulate the output values (e.g. computed spectra) using S-Lang array-based mathematical operators.

Because most of the code for this module is automatically generated from files in the XSPEC distribution, the model parameter names and the order of parameters should exactly match the XSPEC interface (although the ISIS `norm` parameter always comes first in the parameter list). For details on how to automatically generate new module source code in order to match an upgrade of XSPEC that introduces new source functions, see `modules/xspec/src/README.code`.

8.1 Installation and Setup

See `INSTALL` and `modules/xspec/README` for detailed instructions on how to compile and install the module.

If the module is dynamically linked, it will be automatically imported when any of the module functions is used for the first time. One can also import it explicitly using

```
require ("xspec");
```

When the module is statically linked, the S-Lang preprocessor symbol `XSPEC_IS_STATIC` is defined. To write scripts which work independent of how the XSPEC module is linked, one can use

```
#ifdef XSPEC_IS_STATIC
   require ("xspec");
#endif
```

[1]On systems with ELF support this module may be included via dynamic linking, making it possible to install and use ISIS without first installing XSPEC. For convenience, the module may also be statically linked, eliminating the need to first `import` the module before using the XSPEC fit-functions.

This will insure that no attempt is made to import the XSPEC module on systems where the module is unavailable.

8.2 Imported Functions

This section is incomplete – many more functions are included in the XSPEC module than are described here. Use `list_functions` *to get a list of function names. The XSPEC documentation provides parameter definitions and more complete documentation.*

See the XSPEC documentation for references and a more complete description of these functions.

add_atable_model

Purpose: Define an additive table-model
Usage: `add_atable_model ("filename", "modelname")`
See Also: `add_etable_model, add_mtable_model`

This function loads an XSPEC-format additive table model and defines a fit-function based on that table, using parameter names defined in the `NAME` column of the file. Multiple instances of each table-model may be fitted simultaneously.

```
Example:
   add_atable_model ("atable.fits", "bshock");
   fit_fun ("bshock(1) + bshock(2)");
```

add_etable_model

Purpose: Define an exponential table-model
Usage: `add_etable_model ("filename", "modelname")`
See Also: `add_atable_model, add_mtable_model`

This function loads an XSPEC-format exponential table model and defines a fit-function based on that table, using parameter names defined in the `NAME` column of the file. Multiple instances of each table-model may be fitted simultaneously.

```
Example:
   add_etable_model ("etable.fits", "my_exp");
   fit_fun ("my_exp(2) * mekal(1)");
```

add_mtable_model

Purpose: Define an multiplicative table-model
Usage: `add_mtable_model ("filename", "modelname")`
See Also: `add_atable_model, add_etable_model`

This function loads an XSPEC-format multiplicative table model and defines a fit-function based on that table, using parameter names defined in the `NAME` column of the file. Multiple instances of each table-model may be fitted simultaneously.

```
Example:
   add_mtable_model ("mtable.fits", "my_mul");
   fit_fun ("my_mul(1)*mekal(1)");
```

build_xspec_local_models

Purpose: Compile XSPEC local models (xspec 12+ only)
Usage: `build_xspec_local_models ([dir [, pkg_name]][;lmodel=filename])`
See Also: `;`

load_xspec_local_models

If no arguments are provided, this function will compile the XSPEC local models in the directory specified by the `LMODDIR` environment variable. Optionally, the directory path and a package name may be provided. By default, model parameter names are loaded from a text file named `lmodel.dat`; an alternate file may be specified using the `lmodel` qualifier.

Use `load_xspec_local_models` to import the spectral models into isis.

load_xspec_local_models

Purpose: Load XSPEC local models
Usage: `load_xspec_local_models ( [dir [, pkg_name]][;lmodel=filename])`
See Also: `build_xspec_local_models`

ISIS can use local models compiled for XSPEC. Any local models available along path specified by the `LMODDIR` environment variable are automatically loaded at the time the XSPEC module is imported. Local models may also be loaded by specifying the shared library directory and, optionally, the associated package name. By default, model parameter names are loaded from a text file named `lmodel.dat`; an alternate file may be specified using the `lmodel` qualifier.

Defining this global variable:

```
  public variable _xspec_module_verbose_link_errors=1;
```

will cause `load_xspec_local_models` to generate more verbose error messages that may help diagnose certain linking errors that may occur.

xspec_abund

Purpose: Specify the abundance table used by XSPEC models
Usage: `status = xspec_abund (name | abund_array)`
See Also: `xspec_xsect`

If this function is called with no name parameter, the name of the current abundance table is returned.

See the XSPEC documentation for details on the accepted values of the `name` parameter.

An abundance table may also be specified by providing a 30 element S-Lang array of abundance values relative to Hydrogen.

xspec_config_hook

Purpose: Perform xspec related customizations
Usage: `xspec_config_hook()`
See Also: `xspec_rename_model_hook`, `__set_hard_limits`

If a function named `xspec_config_hook` is defined when the XSPEC module is loaded, then that function will be called just after XSPEC module initialization finishes. This feature is intended to provide a simple way to perform any customizations that may be needed before using XSPEC models.

For example, the default hard limits on some XSPEC model parameters may be set to inconvenient values. In particular, the hard limits of Doppler shift parameters often restrict them to positive values even though negative values are perfectly well supported by the implementation. To automatically change such hard limits, one can insert a function definition of the form:

```
define xspec_config_hook ()
{
   __set_hard_limits("compth","redshift",-4.,4.);
}
```

into either a user's `isisrc` file or into a site-customization file. When isis starts, the `xspec_config_hook` function will be defined. Then, whenever the xspec module is loaded, the hook will automatically change the hard limits to the desired values.

xspec_elabund

Purpose: Retrieve XSPEC element abundances
Usage: `abundance = xspec_elabund (elname)`
See Also: `xspec_xsect`

The element name should be specified using its chemical abbreviation. Use `xspec_abund` to specify an abundance table.

xspec_get_cosmo

Purpose: Specify the cosmological parameters used by XSPEC models
Usage: `(H0, q0, lambda0) = xspec_get_cosmo();`
See Also: `xspec_set_cosmo`

The cosmological parameters are the Hubble constant (`H0`), the deceleration parameter (`q0`) and the cosmological constant (`lambda0`). See the XSPEC documentation for details

xspec_gphoto

Purpose: Compute mean photoelectric absorption cross-sections (Verner)
Usage: `s = xspec_gphoto (E1_kev, E2_kev, Z)`
See Also: `xspec_phfit2, xspec_photo`

This function computes the mean photoelectric absorption cross section (in cm^2) for an element with atomic number Z, by averaging the value at energies `E2_kev` and `E2_kev`.

xspec_help

Purpose: Display documentation for an XSPEC model
Usage: `xspec_help (model_name [; method=<method-string>])`
See Also: `apropos, help`

The spectral model documentation for xspec12 is locally available in either `html` or `pdf` format. The `method` qualifier may be used to select the preferred file format and reader. This string variable must have the structure

```
"<file-format> ; <display-command>"
```

where `file-format` is either `pdf` or `html`. The `%s` directive must appear in the `display-command` substring to indicate where the name of the documentation file should be inserted.

The environment variable `XSPEC_MODULE_HELP` may also be used to select the preferred file format and reader.

```
EXAMPLE

% To view html documentation with the web browser ‘firefox’:
  xspec_help ("powerlaw", method="html; firefox %s");
or
  setenv XSPEC_MODULE_HELP "html; firefox %s"

% To view pdf documentation with the pdf reader ‘acroread’:
  XSPEC_MODULE_HELP = "pdf; acroread %s"
```

xspec_ionsneqr

Purpose: Compute non-equilibrium ionization for a given Te, tau structure
Usage: `Struct_Type = xspec_ionsneqr (T[], tau[])`
See Also: `xspec_abund`

This function provides an interface for the XSPEC subroutine `ionsneqr`. The input arrays should have the same length. The return value is a structure of the form

```
struct {fout, ionel, ionstage}
```

From the `ionsneqr` source code documentation:

```
Calculates ionization fractions fout at electron temperature
tmp(n) and ionization parameter tau(n), for electron
temperatures tmp given in a tabular form as a function of
ionization parameter tau.

Input:  tmp      - temperatures (K)
        tau      - ionization timescales  (cm^-3 s)
Output: fout     - ionic concentrations
        ionel    - the element for each fout entry
        ionstage - the ion stage for each fout entry
                   (ionel+1 = fully stripped)
```

xspec_phfit2

Purpose: Compute partial photoelectric absorption cross-sections (Verner)
Usage: `s = xspec_phfit2 (nz, ne, is, e)`
See Also: `xspec_gphoto, xspec_photo`

This function provides an interface for Dima Verner's phfit2 subroutine. From the `phfit2` source code:

```
This subroutine calculates partial photoionization cross sections
for all ionization stages of all atoms from H to Zn (Z=30) by use of
the following fit parameters:
Outer shells of the Opacity Project (OP) elements:
   Verner, Ferland, Korista, Yakovlev, 1996, ApJ, in press.
Inner shells of all elements, and outer shells of the non-OP elements:
   Verner and Yakovlev, 1995, A&AS, 109, 125
Input parameters:  nz - atomic number from 1 to 30 (integer)
                   ne - number of electrons from 1 to iz (integer)
                   is - shell number (integer)
                   e - photon energy, eV
Output parameter:  s - photoionization cross section, Mb
Shell numbers:
1 - 1s, 2 - 2s, 3 - 2p, 4 - 3s, 5 - 3p, 6 - 3d, 7 - 4s.
If a species in the ground state has no electrons on the given shell,
the subroutine returns s=0.
```

xspec_photo

Purpose: Compute mean photoelectric absorption cross-sections (BCMC)
Usage: `photo = xspec_photo (keV1, keV2, Z [, versn])`
See Also: `xspec_gphoto`

This function provides an interface to the `photo.f` subroutine distributed with xspec. From the source code:

```
Cross-section data from Henke etal, Atomic Data and Nuclear Data Tables
vol 27, no 1, 1982. Fits mainly by Monika Balucinska-Church and Dan McCammon
"Photoelectric Absorption Cross Sections with Variable Abunances"
Ap.J. 400, 699 (1992)
```

```
Arguments :
     keV1    r       i: Lower energy of bin in keV.
     keV2    r       i: Upper energy of bin in keV.
     Z       i       i: Atomic number of element
     versn   i       i: 2 == old Marr & West He x-section
                        3 == new Yan et al. x-section
     photo   r       r: Cross-section in cm**2
```

xspec_rename_model_hook

Purpose: Define a mapping from XSPEC model to ISIS fit-function names
Usage: `xspec_rename_model_hook (isis_model_name_assoc_array)`
See Also: `xspec_config_hook`

If a function named '`xspec_rename_model_hook`' is defined when the XSPEC module is loaded, then that function will be called with an associative array as a parameter. The hook can be used to define different fit-function names for XSPEC models that are provided to ISIS – e.g., to avoid name clashes with its own intrinsic fit-functions.

Unlike `xspec_config_hook`, `xspec_rename_model_hook` is called before the XSPEC module initializes its fit functions.

For example, in order to rename XSPEC's `voigt` model to `VOIGT`, one can define the following function in the installation's `isis/etc/local.sl` or even one's personal `.isisrc` file:

```
define xspec_rename_model_hook (isis_model_name)
{
   isis_model_name["voigt"] = "VOIGT";
}
```

xspec_set_cosmo

Purpose: Specify the cosmological parameters used by XSPEC models
Usage: `xspec_set_cosmo(H0, q0, lambda0)`
See Also: `xspec_get_cosmo`

The cosmological parameters are the Hubble constant (`H0`), the deceleration parameter (`q0`) and the cosmological constant (`lambda0`). See the XSPEC documentation for details

```
For example:
   xspec_set_cosmo (75.0, 0.5, 0.7);
```

xspec_xsect

Purpose: Specify the photoionization cross-section table used by XSPEC models
Usage: `status = xspec_xsect (name)`
See Also: `xspec_abund`

The available cross-section tables are `bcmc`, `vern`, `obcm`. If this function is called with no name parameter, the name of the current cross-section table is returned.

See the XSPEC documentation for details

xspec_xset

Purpose: Set various XSPEC parameters
Usage: `xspec_xset (s̈ymbol,̈ v̈alue)̈`
See Also: `xspec_abund, xspec_xsect`

This function is equivalent to the XSPEC `xset` function.

For example, to set the value of the XSPEC parameter `NEIVERS` (which controls the atomic data used by the various non-equilibrium ionization models), do

```
xspec_xset ("NEIVERS", "2.0");
```

See the XSPEC documentation for details.

Section 9

Customizing ISIS Configuration

The ~/.isisrc file allows individual users to customize ISIS at run-time; a template configuration file called isis.rc is included with the distribution. This file is a S-Lang script which is executed at ISIS startup and could, for example, be used to automatically load a particular database or to initialize some constants or other useful data structures. The sample file explains how to set several ISIS intrinsic variables to control some aspects of the program's operation (also see §3.2).

Extensions which are to be made available to all ISIS users at a particular site may be initialized using the file etc/local.sl in the ISIS source directory. If this file exists, it is automatically loaded at ISIS startup (before the user's .isisrc is loaded). Documentation for such local extensions may be provided in an ascii file called local_help.txt in the same directory; the format of this file should be the same as that of the help.txt file included in the ISIS distribution. If the file local_help.txt exists, the interactive help system will search it automatically.

A local customization script may also be specified at configure time using the --with-local-setup=FILE configure option. The specified S-Lang script will be loaded automatically when isis starts.

Note that each individual user's ~/.isisrc file will be loaded last, making it possible to over-ride site-local definitions from either the etc/local.sl or the local setup files mentioned above.

9.1 Environment Variables Affecting ISIS

Although the simplest configuration doesn't require setting any environment variables, several environment variables are available to help customize ISIS (Table 9.1). Note that, because S-Lang provides the ability to change environment variables of the isis process, it is possible to change some of of the following values at run-time (it is not useful to change those environment variables which are examined only at ISIS startup). For example, the default editor can be changed at run-time using the S-Lang function putenv. To change the current editor to emacs without first exiting ISIS, use

```
isis>  putenv ("EDITOR=emacs");
```

Table 9.1: Environment Variables Affecting ISIS

ISIS_SRCDIR	Optional	Provides the full path to the ISIS source code directory. If the ISIS source-code directory is moved after ISIS is compiled, use this environment variable to specify the new path.
ISIS_HISTORY_FILE	Optional	If GNU readline has been enabled (see `set_readline_method`) the cumulative history of interactive commands will be saved in the file specified by this environment variable.
PGPLOT_DIR	*Required by* PGPLOT	Provides the path to the PGPLOTfont files (`grfont.dat`, etc.)
PGPLOT_DEV	Optional	Specifies the default plot device. See the description of the PGPLOT function `pgopen` for details. See also `plot_device()`
PAGER	Optional	Specifies which pager program to use for browsing output text; the default is `more`. See the table notes below for more information.
EDITOR	Optional	Specifies which editor to use for editing parameter tables; the default is `vi`. See the table notes below for more information.
TMPDIR	Optional	If set, ISIS places temporary files in the specified directory, otherwise, all temporary files are placed in the startup directory.
HEADAS	*May be required to build the* XSPEC *module*	Provides the path to the XSPEC installation; usually this is the path to the architecture-specific subdirectory in the HEASARC headas installation.
ISIS_LOAD_PATH	Optional	ISIS searches this colon-separated list of directories when loading S-Lang scripts.
SLANG_LOAD_PATH	Optional	ISIS searches this colon-separated list of directories when loading S-Lang scripts.
ISIS_MODULE_PATH	Optional	ISIS searches this colon-separated list of directories when loading dynamically linked modules (.so files).
SLANG_MODULE_PATH	Optional	ISIS searches this colon-separated list of directories when loading dynamically linked modules (.so files).
LD_LIBRARY_PATH	System specific	This colon-separated list of directories is searched when loading dynamically linked modules (.so files).

Table Notes:

PAGER

Because the standard Unix `more` pager program may not provide sufficient flexibility, we recommend using a high quality pager program such as `most` which allows scrolling files forward, backward and even horizontally; other good choices include `less` and `jed`. If PAGER is not set and `more` is not found on the command search path, some output browsing functions will not work properly.

EDITOR

If EDITOR is not set and `vi` is not found on the command search path, the parameter editing functions will not work properly. If `emacs` is used, the `emacsclient` feature (of `emacs`) may be used to avoid invoking a new `emacs` process for each edit; see the `emacs` documentation for more details.

9.2 ISIS Intrinsic Variables

ISIS also defines a number of S-Lang intrinsic variables which are used to control the behavior of various functions (Table 9.2). Their default values may be changed at any time by changing the value of the variable – such configuration changes always take place instantly and do not require restarting ISIS or re-loading the `~/.isisrc` file.

Variable	Default Value	Purpose
`Ion_Format`	`FMT_ROMAN`	Controls the print format for ion names. Allowed values are `FMT_ROMAN`, `FMT_INT_ROMAN`, `FMT_CHARGE`
`Use_Memory`	1	Controls run-time memory usage (see `plasma` and §4.2). If non-zero, most data tables will be loaded into memory all at once. If zero, needed items from from large tables will be loaded only when actually used.
`Incomplete_Line_List`	1	Controls the treatment of line emissivity tables on input (see `plasma`). The default probably need not be changed.
`Isis_Active_Dataset`	-	This variable contains the index of the dataset for which the fit-function is currently being evaluated. This intrinsic variable is primarily useful for writing user-defined fit-functions which evaluate differently for different datasets. See also `assign_model`.
`Isis_Append_Semicolon`	0	If non-zero, ISIS will automatically append a semicolon (;) to each input command line in interactive mode. Otherwise, the user must type the semicolon to mark the end of the command line.
`Isis_Eval_Grid_Method`	`SEPARATE_GRID`	This sets the default evaluation grid type used when fitting models to data. Supported grid types are `MERGED_GRID|SEPARATE_GRID`. For details, see `set_eval_grid_method`. .
`Isis_List_Filenames`	1	If non-zero, `list_data` will list the spectrum filename and background filename for each dataset, if any.
`Isis_Num_Slaves`		The number of slave processes used for parallel processes.
`Isis_Use_PHA_Grouping`	0	If non-zero, `load_data` will apply the channel grouping specified by the `GROUPING` column in the input PHA file.
`Ignore_PHA_Response_Keywords`	0	If non-zero, `load_data` will ignore the `RESPFILE` and `ANCRFILE` keywords in PHA file headers.
`Ignore_PHA_Backfile_Keyword`	0	If non-zero, `load_data` will ignore the `BACKFILE` keyword in PHA file headers.
`Allow_Multiple_Arf_Factors`	0	See `assign_arf`.
`_num_statistic_evaluations`	0	The number of fit-statistic evaluations performed during the most recent fit. Note that if the fit method is parallelized, this count may represent only those statistic evaluations occurring in the current process (the "master" process). Whether or not the count includes statistic evaluations in other (slave) processes depends on the implementation of the specific parallel fit method.

Variable	Default Value	Purpose
Isis_Residual_Plot_Type	-	This variable controls the type of residuals generated by rplot_counts and rplot_flux. Supported options are STAT, DIFF, RATIO, to generate $\Delta\chi^2$ residuals, (data-model) residuals and (data/model) ratio plots.
Label_By_Default	1	Controls the labeling of spectrum plot axes. If non-zero, plot axes are labeled automatically; otherwise, axes are not labeled.
Fit_Verbose	0	Controls the level of information printed during iterative fitting of models to data. If zero, only the final value of the fit-statistic is printed. If negative, no information is printed, if positive extra information is printed.
Isis_Verbose	0	Controls the general level of verboseness. Larger positive values indicate increased verbosity. A negative value silences all informational output except messages reporting serious errors.
Minimum_Stat_Err	0	Used for validating statistical uncertainties associated with input spectral data. When positive, this value of this variable defines the smallest valid uncertainty for a single data bin. If zero (the default), Poisson errors are assumed.
Warn_Invalid_Uncertainties	1	If non-zero, a warning message will indicate when invalid uncertainty values have been replaced.
Min_Model_Spacing	0.0 Å	Sets the minimum wavelength grid spacing used when computing the spectral model $S(\lambda)$ for multi-order spectral analysis. Use Min_Model_Spacing=0 (the default) to get the finest allowed grid spacing (resolution set by the combination of all the ARF grids).
Remove_Spectrum_Gaps	0	Used for validating the histogram grid of the input spectrum. If zero (the default), ISIS will refuse to read a spectrum which has $\texttt{bin\_hi}[\texttt{i}] \neq \texttt{bin\_lo}[\texttt{i}+1]$ for any spectrum bin "i". If non-zero, such gaps will be automatically removed by merging them into the neighboring bin.
Verbose_Fitsio	0	Controls printing of FITSIO error messages. If non-zero, FITSIO error messages will be printed when they occur.
Rmf_Grid_Tol	10^{-4}	Controls the internal grid validity checking. An error is generated when the fractional error between the data, RMF and ARF grids exceeds this value. To suppress such errors, set the tolerance to a larger value.
Rmf_OGIP_Compliance	2	Controls the level of OGIP standard compliance required of FITS RMF files. Setting this variable to zero means that minimum standard compliance is required.
Xspec_Module_Help	"pdf;xpdf %s"	Indicates the preferred format for viewing XSPEC 12 model documentation. Available formats are pdf and html. The environment variable XSPEC_MODULE_HELP may also be used to provide the same information.

Bibliography

[1] Arnaud, K.A., 1996, Astronomical Data Analysis Software and Systems V, eds. Jacoby G. and Barnes J., p17, ASP Conf. Series volume 101.

[2] Bevington, P.R., and Robinson, D.K., 1992, "Data Reduction and Error Analysis for the Physical Sciences", 2ed, (McGraw-Hill).

[3] Davis, J., 2001, "Event Pileup in Charge Coupled Devices", Astrophysical Journal, submitted.

[4] Smith, R. K., Brickhouse, N. S., Liedahl, D. A., & Raymond, J. C. 2001, Astrophysical Journal, 556, L91.

Index