

Adding models to XSPEC

XSPEC includes a large collection of standard models that can be fit to data. However, sometimes these are not enough and a new model might be required. In order of increasing complexity the ways to do this are: use the `mdefine` command; create a table model; load a model function created by someone else; create and load your own model function. The `mdefine` command can be used for a model which can be described using a simple formula and is documented under the commands section of the manual so we do not discuss it further. This appendix describes the other three methods then finishes with a note about the more complex issue of mixing models.

Table models

A very simple way of fitting with user-defined models is available for a particular class of models. These are models that can be defined by a grid of spectra, with the elements of the grid covering the range of values of the parameters of the model. For instance, for a one-parameter model, a set of model spectra can be tabulated for different values of the parameter (P1, P2, P3, etc.) The correct model spectrum for a value P is calculated by interpolation on the grid. The generalization to more parameters works in the obvious way. The table is specified in the model command by the special strings `atable`, `mtable`, or `etable` with the filename following in brackets – see the entries in the models section of the manual. Any number of table model components can be used simultaneously.

Table model components can be much slower than most standard models if there are significant numbers of parameters. The memory requirements increase as 2^n where n is the number of parameters in the model. A table model with more than 3 or 4 fitting parameters is not recommended. Additionally, the interpolation is linear, which implies that the second derivatives used by the default Levenberg-Marquadt algorithm may not be accurate. If the fit does not work well it may be worth trying the `migrad` (minuit library) algorithm which makes no assumptions about the second derivative.

As with standard models, the spectra should be in terms of flux-per-bin and not flux-per-keV. Any set of energy bins can be used, and XSPEC will interpolate the model spectra onto the appropriate energy bins for the detectors in use. It is therefore a good idea to choose energy bins such that the spectrum is well-sampled over the range of interest. The file structure for these models is a FITS format described at :

http://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/general/ogip_92_009/ogip_92_009.html

or:

ftp://legacy.gsfc.nasa.gov/fits_info/fits_formats/docs/general/ogip_92_009

Loading a new model function

New model functions either downloaded from the XSPEC additional models webpage at :

<http://heasarc.gsfc.nasa.gov/docs/xanadu/xspec/newmodels.html>

or acquired privately are added using the two commands **initpackage**, which prepares and compiles a library module containing them, and the **lmod** command which actually loads them into the program. These commands are described in the XSPEC commands section of the manual, to which the user is referred. Any number of different user model packages may be added to XSPEC from the user prompt, and the user has control over the directory from which models are loaded. On Cygwin, local model libraries must be built and linked to XSPEC statically prior to runtime, and therefore cannot be loaded with **lmod**. Cygwin users should consult the **static_initpackage** section of the **initpackage** command documentation.

Note that the **lmod** command requires write-access to the particular directory specified. This is because **lmod** uses the Tcl ‘make package’ and ‘package require’ mechanisms for automatic library loads and these require Tcl write an index file (pkgIndex.tcl) to the directory. Consequently we recommend using the Tcl **load** command instead of **lmod** if the library is being used by a number of users on a local network. Note that such a library can be loaded automatically by placing the command in the `global_customize.tcl` script (see the section “Customizing XSPEC”).

Writing a new model function

A model function is a subroutine that calculates the model spectrum given an input array of energy bins and an array of parameter values. The input array of energy bins gives the boundaries of the energy bins and hence has one more entry than the output flux arrays. The energy bins are assumed to be contiguous and will be determined by the response matrix in use. The subroutine should thus make no assumptions about the energy range and bin sizes. The output flux array for an **additive** model should be in terms of photons $\text{cm}^2 \text{s}^{-1}$ (not photons $\text{cm}^2 \text{s}^{-1} \text{keV}^{-1}$) i.e. it is the model spectrum integrated over the energy bin. The output array for a **multiplicative** model is the multiplicative factor for that bin. Convolution models are operators on the output from additive or multiplicative models. Model subroutines can be written in fortran, either in single or double precision, in C++ using either C++-style arguments or C style arguments, and in C.

The “model.dat” entry

In addition to the subroutine, XSPEC requires a text file describing the model and its parameters. The standard models are specified in the `model.dat` file so we usually refer to this text file by that name. A sample `model.dat` entry has the following form:

modelentry	5	0.	1.e20	modelfunc	add	0	0
lowT	keV	0.1	0.0808	0.0808	79.9	79.9	0.001
highT	keV	4.	0.0808	0.0808	79.9	79.9	0.001

```

Abundanc " "      1.      0.      0.      5.      5.      0.01

*redshift " "      0.0

$switch      1

```

The first line for each model gives the model name, the number of parameters, the low and high energies for which the model is valid, the name of the subroutine to be called and the type of model (add, mul, mix, or con, or acn). The final argument two arguments are flags: the first should be set to 1 if model variances are calculated by modelfunc and the second should be set to 1 if model should be forced to perform a calculation for each spectrum. This final flag is necessary because if multiple spectra have the same energy bins, the default behavior is to perform the model calculation for just one spectrum and copy the results for each of the others. However if a model depends on information about the spectrum in addition to its energy ranges, it must be forced to perform a calculation for each spectrum.

The remaining lines in the text file specify each parameter in the model. For regular model parameters the first two fields are the parameter name followed by an optional units label. If there is no units label then there must be a quoted blank (“ ”) placeholder. The remaining 6 numerical entries are the default parameter value, hard min, soft min, soft max, hard max, and fit delta, which are described in the **newpar** command section.

There are three special types of parameter which can be used. If the name of the parameter is prefixed with a “*” the parameter is a “scale” parameter and cannot be made variable or linked to an other kind of parameter other than another scale parameter. Since the parameter value can never vary only the initial value need be given. If the name of the parameter is prefixed with a “\$” the parameter is a “switch” parameter which is not used directly as part of the calculation, but switches the model component function’s mode of operation (i.e. calculate or interpolate). Switch parameters only have 2 fields: the parameter name and an integer value. If a P is added at the end of the line for a parameter then the parameter is defined to be periodic. During a fit, a periodic parameter will not be pegged if it tries to exceed its hard limits. Instead it will be assigned a value within its limits: $f(\text{max} + \text{delta}) = f(\text{min} + \text{delta})$, $f(\text{min} - \text{delta}) = f(\text{max} - \text{delta})$. The soft min and max settings are irrelevant for period parameters and will be ignored.

The model subroutine function

The following table lists the function arguments required for the different language options. The second column is the way the function name should be included in the model.dat entry.

Call Type	Specification	Arguments and Type	Meaning
Single precision fortran	modelfunc	real*4 ear(0:ne)	Energy array
		integer ne	Size of flux array

		real*4 param(*)	Parameter values. (Dimension must be specified inside the function)
		integer ifl	The spectrum number being calculated
		real*4 photar(ne)	Output flux array
		real*4 photer(ne)	Output flux error array (optional)
Double precision fortran	F_modelfunc	real*8 ear(0:ne)	As above
		integer ne	"
		real*8 param()	"
		integer ifl	"
		real*8 photar(ne)	"
		real*8 photer(ne)	"
C/C++, C-style	c_modelfunc	const Real* energy	Energy array (size Nflux+1)
		int Nflux	Size of flux array
		const Real* parameter	Parameter values
		int spectrum	Spectrum number of model component being calculated
		Real* flux	Output flux array
		Real* fluxError	Output flux error array (optional)
		const char* init	Initialization string (see below)
C++, C++ style	C_modelfunc	const RealArray& energy	Energy array
		const RealArray& parameter	Parameter values

		int spectrum	Spectrum number of model component being calculated
		RealArray& flux	Output flux array
		RealArray& fluxError	Output flux error array (optional)
		const string& init	Initialization string (see below)

For example, a model component in double precision fortran is specified by:

```
modelentry      5  0.      1.e20      F_modelfunc      add  0
```

XSPEC then picks out the right function definition, and calls the function `modelfunc` which expects double precision arguments. The C-style call can clearly be compiled and implemented by either a C or a C++ compiler: however we recommend using the C++ call if the model is written in C++ as it will reduce overhead in copying C arrays in and out the XSPEC internal data structures. To prevent unresolved symbol linkage errors, we also recommend prefacing C++ local model function definitions with the **extern "C"** directive.

Example C/C++ function definitions:

```
/* C -style */
```

```
extern "C" void modelfunc(const Real* energy, int Nflux, const Real* parameter, int spectrum,
Real* flux, Real* fluxError, const char* init)
```

```
{
```

```
/* Model code: Do not allocate memory for flux and fluxError arrays. XSPEC's
```

```
C-function wrapper will allocate arrays prior to calling the function (and will free them
afterwards). */
```

```
}
```

```
// C++
```

```
extern "C" void modelfunc(const RealArray& energy, const RealArray& parameter, int
spectrum, RealArray& flux, RealArray& fluxError, const string& init)
```

```
{
```

```

// Model code: Should resize flux RealArray to energy.size() - 1. Do the same for
// fluxError array if calculating errors, otherwise leave it at size 0.

}

```

Note on type definitions for (C and C++): XSPEC provides a typedef for **Real**, in the `xsTypes.h` header file. The distributed code has

```
typedef Real double;
```

i.e. all calculations are performed in double precision. This is used for C models and C++ models with C-style arguments.

The type **RealArray** is a dynamic (resizeable) array of **Real**. XSPEC uses the `std::valarray` template class to implement **RealArray**. The internal details of XSPEC require that the **RealArray** typedef supports vectorized assignments and mathematical operations, and indirect addressing (see C++ documentation for details). However, we do not recommend using specific features of the `std::valarray` class, such as array slicing, in case the typedef is changed in future.

The input energies are set by the response matrices of the detectors in use. IFL is an integer which specifies to which response (and therefore which spectrum) these energies correspond. It exists to allow multi-dimensional models where the function might also depend on eg pulse-phase in a variable source. The output flux array should not be assumed to have any particular values on input. It is assumed to contain previously calculated values only by convolution/pileup models, which have the nature of operators. The output flux error array allows the function to return model variances.

The C and C++ call types allow one extra argument, which is a character string that can be appended to the top line of the model component description. This string is read on initialization and available to the model during execution. An example of its use might be the name of a file with specific data used in the model calculation: this allows different models to be implemented the same way except for different input data by specifying different names and input strings.

Third-Party Libraries In Local Models Build

The Makefile that **initpackage** creates for building your local models library is based on the template file `heasoft-[ver]/Xspec/src/tools/initpackage/xspackage.tmpl`. If you need to add a path to a third-party library's header files, add: `-I/path/to/your/3rdParty/library/include` to the `HD_CXXFLAGS` setting. Then type "hmake" and "hmake install" from the `heasoft-[ver]/Xspec/src/tools/initpackage` directory.

To make sure the linker pulls in the library on **Mac OS X**:

Further edit the `xspackage.tmpl` file by adding a "-l" flag for the library (e.g. `-lgsl`) in the `HD_SHLIB_LIBS` settings. Then reinstall `xspackage.tmpl` as mentioned above.

On Linux/Unix:

The XSPEC executable itself should be relinked with the new library included. So, edit the file `heasoft-[ver]/Xspec/src/main/Makefile` by adding a "-l" flag for the library to the `HD_CXXLIBS` setting. Then from the same directory do:

```
rm xspec
hmake local
hmake publish
hmake install
```

After these modifications, you should be able to use **initpackage** and **lmod** in the normal way to build and load your local models library.

Writing new mixing models

Mixing models are fundamentally different from the other kinds of models since they apply a transformation to a set of modeled fluxes (as enumerated by the spectra in the fit), rather than modify the flux designed to fit a single spectrum. The need to store temporary results, as well as the requirements of the model calculation, lead to many workspace arrays: further, the transformations applied are often fixed during a fit, or can be split to avoid redundant calculations into parts that are fixed and parts that change during iteration in order. XSPEC's internal organization (data structures) can be mapped straightforwardly to the requirements of these models so to implement them efficiently and handle memory allocation, we recommend that mixing models be written in C++ or C. At present only a C++ implementation is available. Users considering adding new mixing model types should contact the developers of XSPEC at xspect12@athena.gsfc.nasa.gov