**1. Multifit methods requirements**

In many cases, the most convenient thing to do is extract the x,y,e arrays from an object and pass those to multifit. This can be done by using a method defined on the object to extract an x,y,e triple:

>> [wout, fitdata] = multifit (xye(w), func, pin,...)

where the method xye must return a structure of the form required by multifit, namely a structure with fields ‘x’,’y’ and ‘e’ ,where ‘x’ is a cell array {x1,x2,…} containing the coordinates of the points along the first, second… axes, and ‘e’ contains the standard deviations. A convenient way to do this is to use the methods sigvar\_get and sigvar\_getx if they have been written to allow the object itself to be passed to multifit (see below).

If multifit is being used to fit functions to objects rather than x-y-e triples, then there are some methods that need to be defined. You might want to fit the objects if their internal structure is more complex, for example if the fitting function depends on fields other than just the x values and parameters being passed to the fit function. Another case is when the masking of points from fitting requires manipulation of fields other than simply removing x-y-e values. [An example is the case of the sqw objects used in Horace. Here the calculation of the intensity at a data point depends on the information of the individual pixels that contribute to that data point. Masking requires that the pixel information of masked bins is removed from the sqw object.]

The methods required for fitting objects with multifit are as follows

**1.1 Fit functions**

The global function, and background function(s) if given, must be methods of the class, with input argument form as described in detail in multifit help. The general format is:

>> wcalc = my\_function (w,p,c1,c2,...)

**1.2 Utility methods**

These are required to enable multifit to work with objects

**1.2.1 wout = mask (win, msk)**

A method that masks data points from further calculation. The output object must be a valid instance of the class in which the masked values have been removed in whatever sense the class requires.

**1.2.2 [y,var,msk] = sigvar\_get(win)**

A method that returns the intensity and variance arrays from the objects, along with a mask array that indicates which elements are to be retained (where elements of msk are true, the corresponding elements of y and var are retained). The output arrays must y and var must have the same size and shape; msk must have the same number of elements (but can be a different shape). The array msk must be understood by the method 'mask' defined below.

**1.2.3 wsum = w1 + w2**

If a background function is provided, addition of objects must be defined as (requires overloading of the addition operator with a method named plus.m)

**1.2.4 x = sigvar\_getx(win) *[optional]***

Get the corresponding x values to the y, var, msk arrays.

-if one dimensional i.e. single x coordinate per point:

x must be a single array, the same size as y and var

-if n-dimensional i.e. n x-values per point:

x must be a cell array of arrays, one per x dimension, each the same size as y and var.

This method replaces the need to have the method 'mask\_points' described below, as 'sigvar\_getx' will enable the masking function built in to multifit to be used. However, if mask\_points exists, then it will have priority over the use of sigvar\_getx.

**1.2.5 [msk, ok, mess] = mask\_points (win, 'keep', xkeep, 'xremove', xremove, 'mask', msk\_in) *[optional]***

Create a mask array given ranges of x-coordinates to keep &/or remove &/or &/or mask array. The mask arrays have true for those elements which are to be retained. Must output a logical flag ok, with message string if ~ok rather than terminate. (Can have it terminate if ok and mess are not given as return arguments; it is the advanced syntax is required within multifit).

**2. Function organisation**

**2.1. User functions**

Multifit provides two user functions for fitting:

* fit

Fitting a single dataset, or independently fitting a collection of datasets in a loop (using the same initial parameter sets for each dataset), to

* + a single function, or
  + a foreground and a background function e.g. a Gaussian as a the foreground, and a straight line as the background.
* multifit

Simultaneously fitting a collection of datasets to one of

* + a single function (‘global foreground’),
  + a single function (‘global foreground’) and independent background functions (‘local backgrounds’),
  + independent functions for each dataset (‘local foreground'),
  + independent functions for each dataset (‘local foreground’) and independent background functions (‘local backgrounds’).
  + independent functions for each dataset (‘local foreground’) and a single background function (‘global backgrounds’).

Multifit is very general; it allows parameters to be bound to each other in fixed ratios, both within a single function, across foreground and background functions, and between foreground functions. In fact, the function ‘fit’ is simply a wrapper to call multifit in a loop over each input dataset.

2.2 Developer functions

Sometimes it is convenient or necessary to overload multifit for a particular class. The following gives some examples.

* Making use of pre-existing functions. We might have a library of functions of a single abscissa of the form y=my\_func(x,p) where p is a vector of parameters e.g. a Gaussian ( where p is the peak area, peak location and peak width), the convolution of Gaussian and Lorentzian, etc. The IX\_dataset\_1d object is unrecognised by these functions, but we can create a multifit method that internally wraps the function passed as an input argument using the IX\_dataset\_1d method called func\_eval.
* Fitting Horace sqw objects to models for S(Q,w) with simple backgrounds. Here multifit can be overloaded to internally wrap the foreground functions using sqw/sqw\_eval, and wrap the backgrounds with sqw/func\_eval.

Note that if multifit is overloaded for a class, then fit will automatically use the overloaded multifit method. However, it might be desirable to overload fit as well.

To make the creation of overloaded multifit easy, there are a number of functions available in the public multifit folder:

**multifit\_gateway\_parse\_func**

Parse input arguments to check that they are consistent, and return the parsed

**multifit\_gateway\_wrap\_functions**

Wrap the foreground and background functions with class method(s)

**multifit\_gateway\_main**

General gateway function to the least-squares fitting function.

**multifit\_gateway\_pbind\_struct\_to\_cell**

Convert the binding structure returned by multifit\_gateway\_parse\_func into cell array suitable as an argument to multifit\_gateway\_main (use this if have to custom construct binding – see Tobyfit in Horace for an example)

**multifit\_gateway\_parameter\_get**

Get the numeric parameter vector from a valid parameter argument for a fit function (i.e. stripped of any function nesting)

**multifit\_gateway\_parameter\_set**

Set the numeric parameter vector in a valid parameter argument for a fit function

**multifit\_gateway\_get\_state**

Get the status of multifit. Useful within the function evaluation function.