Documentation for BasisFn.h and BasisFn.c

Steven Andrews, © 2003

See the document “LibDoc” for general information about this and other libraries.

```
typedef struct modeltype   {
     char name[STRCHAR];
     char file[STRCHAR];
     char color[STRCHAR];
     sptr spec;
     sptr uncert;
     set basis;
     int np;
     float sigma;
     float xmin;
     float xmax;
     float dx;
     float *covar; }* modelptr;

typedef struct basisfn {
     char name[STRCHAR];
     char proc[STRCHAR];
     char color[STRCHAR];
     char desc[STRCHAR];
     float (*addr)(float x,float *param,sptr spec,float *deriv);
     modelptr model;
     int n;
     float *param;
     float *pold;
     float *paramlo;
     float *paramhi;
     char **eqn;
     char **pname;
     int *freeze;
     float *scratch;
     int isspec;
     sptr spec; }* basisptr;

basisptr BasisAlloc(int n);
void BasisFree(basisptr b);
modelptr ModelAlloc(sptr s);
void ModelFree(modelptr m);
int CheckBasis(basisptr b,modelptr m);
int CheckMSpec(modelptr m);
int CheckUncert(modelptr m);
int setupbasis();
void PlotBasis(basisptr b);
void PlotModel(modelptr m);
void BasisRange(basisptr b,float *xa,float *xb,float *ya,float *yb,int fn);
void ModelRange(modelptr m,float *xa,float *xb,float *ya,float *yb,int fn);
int CountFreParam(modelptr m,int **frzptr);
int FindBasisParam(basisptr *bptr,modelptr m,char* str) ;
int FindBasisParam2(basisptr *bptr,modelptr m,float* fltptr) ;
float ModelValue(float x,modelptr m);
float ModelValueD(float x,modelptr m,float *d);
void TypeBasis(basisptr b);
void TypeAllBasis();
void TypeModel(modelptr m);
basisptr BasisCopy(basisptr b);
modelptr ModelCopy(modelptr m);
basisptr AddBasis(basisptr b,char *bproc,sptr s,modelptr m);
void RemoveBasis(basisptr b);
int SaveModel(modelptr m);
int LoadModel(modelptr *mptr,char *name,sptr s);
int BasisMath(basisptr b,void *v,sptr *ansptr,float k,char *fn);
int ModelMath(modelptr m,void *v,sptr *ansptr,float k,char *fn);
void endbasis();
float rmserror(modelptr m);
float chisq(modelptr m,float ch2max);
float ChiSqD(modelptr m,float ch2max,float *alpha,float *beta,int np);
int prefit(modelptr m,float ***pptr,float **ploptr,float **phiptr);
int ModelCovar(modelptr m);
void unfit(modelptr m);
int RandomFit(modelptr m);
int LinearFit(modelptr m);
int LMFit(modelptr m);
int RandMultiFit(set s);
```

Requires: `<string.h>`, `<float.h>`, `<math.h>`, `<stdio.h>`, `"math2.h"`, `"Set.h"`, `"Spectra.h"`, `"BasisFn.h"`, `"Rn.h"`, `"Plot.h"`, `"random.h"`, `"RnLU.h"`, `"RnSort.h"`, `"Utility.h"`, `"SpectFit.h"`, `"DiskIO.h"`

Example program: `SpectFit.c`

Written 9-12/98. Updated 10/99. Moderate testing, thoroughly proofread 6/99. Works with Metrowerks C. Updated and modified 3/3/02. Minor changes were made in model saving and loading formats. Moved code for actual basis functions to BasisFns.c 3/10/02. Significant modifications 3/02, and minimal testing since then.

This library is designed to work closely with `Spectra.c` for the *SpectFit* program. For the most part this library will probably not be generally useful. Many commonly used functions, including models that fit spectra, may be created as the sum of a set of basis functions. This library defines a structure to contain all the elements of a basis function and has several routines for general manipulation of basis functions. The routines that define the basis functions are in a separate library, BasisFns.c. This library also defines a model as a set of basis functions, along with its limits and the spectrum it's associated with, and a few model routines are in here. Basis functions and models represent analytical functions, although a couple model elements describe the points where the model is plotted.

Basis functions

The elements of the `basisfn` structure are moderately self-explanatory. `name` is a string which identifies a basis function. Any string is permitted, and the string may be changed at any time. `proc` is the name of the basis function procedure; for example a basis function named “quad:0” can be renamed to “base”, or whatever else, but the procedure is always “quad”. `color` is also a string, but only the first letter of the string is ever used; this follows the color coding method described in Plot.c. `desc` is a string that describes the basis function, purely to assist the user as to what the parameters are for. `addr` is the address of the routine (in BasisFns.c) that computes the basis function value. `model` is a pointer to the model that owns the basis function. For the routines here, basis functions are allowed to exist without being members of models. `n` is the total number of parameters required for the function (e.g. peak area). `param, pold, paramlo, paramhi, eqn, pname`, `freeze`, and `scratch` are arrays of size `n` which, respectively, store the parameter values, old parameter values for unfitting, lower and upper bounds for the parameters, equations that can relate parameters to other variables, parameter names, parameter frozen status, and scratch space for functions. Parameter names should be single words without punctuation. If a parameter is frozen, indicated by a 1 value, then it won’t be allowed to vary during fitting, whereas unfrozen parameters, with 0 values, are optimized in a fit. The parameter bounds are set to `±FLT_MAX` if they are not used. `isspec` is 1 if the basis function requires a real spectrum (such as if the basis function is a spectrum), 2 if a complex spectrum is required, 3 if either a real or complex spectrum is required, and 0 if no spectrum is needed (which is most common). If a spectrum is required, then it is referenced by `spec`. The scratch space may be used temporarily by any function, but should not be assumed to be unchanged or meaningful after a function is called.

Assuming a basis function is allocated correctly and not ruined thereafter, it is safe to assume that all elements are defined, although any of the strings may be empty (all `eqn` and `pname` values are initiallized to empty strings as well). While some routines assume that the spectrum in `spec` matches what’s specified in `isspec`, most do not and should not. The ones that do assume a correspondence are `ModelValue`, `ModelValueD`, `chisq`, `ChiSqD`.

Models

Model elements include a `name` and a `color`, which are completely analogous to the comparable basis function parameters. `file` is either an empty string or the filename of the analytical model file. `spec` is the spectrum the model is intended to fit. While spectra are often required, their existance is never assumed; however, when spectra do exist, they are assumed to be fully set up. `uncert` is an optional list of uncertainties, which is stored as a spectrum type. Any properly set up spectrum is allowed, but only ones that match the modeled spectrum are used. `basis` is the set of basis functions that compose the model. `np` is the total number of parameters in the model, frozen and not; it is the sum of the basis function `n` members. `sigma` is the uncertainty in the data that the model is intended to represent (`spec`), and is used for fitting. If it is 0 or negative, then the rms difference between the model and the spectrum is used as a best guess of errors. `xmin` and `xmax` and `dx` are the domain over which the model is defined and the spacing over which it is calculated. Typically, the limits match those of the spectrum, although this is not necessary. However, `dx` needs to be greater than 0 and `xmax` needs to be larger than `xmin`. `covar` is the covariance matrix returned by fitting routines, and may be used for finding error ranges on the fit parameters or non-orthogonality of the basis functions.

Again assuming correct allocation, it is safe to assume that elements of models are defined and that `xmin`, `xmax`, and `dx` have acceptable values. Exceptions are the spectrum, which may be `NULL`, and the `covar` matrix, which should be either `NULL` or size `np` by `np` (sized for all parameters, not just fittable ones). Ideally, the spectrum, uncertainties, and basis functions would be certain to be valid if present, but this isn’t the case yet.

Useful additions

Currently, all error estimation is done using analytical derivatives, which is fine if they exist and if parameters aren’t linked. However, it would also be easy to calculate χ^2 derivatives numerically, allowing more general error estimation. This would be a small function that would call `chisq` for a bunch of nearby parameter values. Similar additions would allow LM fitting for linked parameters and for multiple models.

The matrix inversion of the alpha matrix is often problematic due to highly covariant parameters, or round-off error. It would be much more robust with a SVD inversion. Also, an SVD fitting routine would be nice, to complement the linear routine. And, while adding fitting routines, simplex would be good too.

The plotting routines could use PlotData rather than plotting line segments; this would make them more portable. The other non-portability is inkey(), used in the fitting routines.

`BasisAlloc` allocates memory for a basis function structure, returning a pointer to the initiallized structure, with `n` parameters, or `NULL` if it was unable to find the necessary memory. `n` may be any number greater than 0. Most structure elements are allocated but left blank. The color is set to 'g', which is green, and `n` is set properly; the arrays are created and the `eqn` and `pname` strings are set to empty strings with `STRCHAR` characters. All parameters are unfrozen and all parameter bounds are set to `±FLT_MAX`.

`BasisFree` frees a basis function but does not remove it from the model. Partially allocated basis functions may be freed, provided that all unallocated pointers are set to `NULL`.

`ModelAlloc` allocates memory for a model. Send in the spectrum to be modeled, and the limits are automatically set to match the full domain of the spectrum. `dx` is set to have the same number of data points as is in the spectrum, the `color` is set to 'r', which is red, and the `basis` set is allocated but left empty. If no spectrum is sent in, a model is still created, with the *x* range from 0 to 10 in steps of 0.5.

`ModelFree` frees a model and allows partially allocated models, provided that unallocated pointers are set to `NULL`.

`CheckBasis` checks that any spectra in basis function `b` and/or the basis functions in `m` match the conditions specified in the `isspec` basis function element. It also checks that `paramlo` values are less than `paramhi` values. It returns 0 for valid basis functions and 8, 12, 16, or 45 for invalid basis functions, using standard error codes from `errors.c`. `b` and/or `m` may be `NULL`.

`CheckMSpec` makes sure that a spectrum in a model, if given, is real. It returns 0 for a valid model and 8 otherwise.

`CheckUncert` makes sure that uncertainties are valid if they are given. Valid means that *x* values match those of the spectrum and all *y* values are positive. Possible error codes are 0, 8, 11, 14, and 16.

`setupbasis` allocates the set structure `basisfns`, which has the scope of all of `BasisFn.c`, and fills it with all functions listed in getbasis, which is in `BasisFns.c`. It returns 1 if memory could not be allocated.

`PlotBasis` plots a basis function, using Plot.c routines. The basis function needs to be a member of a model.

`PlotModel` plots a model using Plot.c routines.

`BasisRange` returns the domain and range of a basis function. The basis function needs to be a member of a model. `xa`, `xb`, `ya`, and `yb` are returned with the *x* low, *x* high, *y* low, and *y* high values. If `fn`=0, then the input values are ignored. If `fn`<0 , the function compares the domain and range with the values that were sent in, and returns the inside values; if `fn`>0, the outside values are returned.

`ModelRange` returns the domain and range of a model. `xa`, `xb`, `ya`, and `yb` are returned with the x low, x high, y low, and y high values. If `fn`=0, then the input values are ignored. If `fn`<0 , the function compares the domain and range with the values that were sent in, and returns the inside values; if `fn`>0, the outside values are returned.

`CountFreParam` returns the number of parameters in a model that are free. If `frzptr` is `NULL`, it is ignored, otherwise it is sent back with an integer vector, sized for the total number of model parameters, with a 1 for each fixed parameter and a 0 for a free parameter.

`FindBasisParam` returns the index number of a parameter in a basis function, or –1 if it could not be found. `str` is the parameter name to be looked for. If `bptr` points to a basis function, only it is searched. Otherwise, `bptr` should point to `NULL` and the every basis function in the model is searched sequentially; the basis function is returned in `*bptr`. If both `*bptr` and `m` are `NULL`, –1 is returned.

`FindBasisParam2` is similar to `FindBasisParam`. However, it searches for one based exclusively on the address of the parameter value. The input `bptr` is ignored, `m` is the model that is searched, and `fltptr` points to the parameter value. It returns –1 if it could not be found.

`ModelValue` returns the value of model `m`, at position `x`. If it has no basis functions, its value is 0. It does not check that the returned value is valid (i.e. not `NaN`, `Inf`, etc.).

`ModelValueD` is identical to `ModelValue` except that it also returns, in `d`, a vector with the derivatives of the model with respect to the unfrozen parameters. That is, `d[i]` is $\partial y/\partial($`param[i]`$)$. The vector needs to be allocated beforehand to be large enough for all unfrozen parameters.

`TypeBasis` displays essentially all information about a basis function to the standard output.

`TypeAllBasis` displays a list of the all the basis functions currently available, along with their descriptions. getbasis needs to have been called at some point beforehand to initiallize the set of basis functions.

`TypeModel` displays most of the information about a model to the standard output. It doesn't display the covariance matrix but, if there is one, it displays the error ranges for the fit parameters. The unfrozen fitting parameters define a point in a multi-dimensional space. The covariance matrix can be used to locally define an uncertainty ellipse in this space, where the ellipse boundary is at points of constant chi squared. The one standard deviation ellipse is separately projected onto each parameter coordinate and those projection ranges are returned as the error ranges, which are both the formal standard errors, and are the single parameter confidence intervals for one standard deviation. See the *SpectFit* documentaion for a more thorough discussion and *Numerical Recipes* for details. Sometimes, "Nan" is displayed for errors, arising from a negative diagonal element of the covariance matrix. It's not clear why this occurs, but it's probably from highly covariant parameters.

`BasisCopy` returns an identical copy of a basis function. It has the same name as the original, points to the same model, and points to the same spectrum, if there is one. The function returns `NULL` if memory could not be allocated.

`ModelCopy` returns an identical copy of a model. Basis functions are copied and are identical to previous ones, except that they point to the new model. The function returns `NULL` if memory could not be allocated.

`AddBasis` adds a copy of a basis function to a model. If `b` is defined, then it is copied and added to the model; if `b` is `NULL`, then the basis function procedure name is required (it's ignored otherwise). `s` is used for the `spec` member of the new basis function, and so should be `NULL` or a spectrum, as appropriate (any spectrum in `b` is ignored). The function returns the address of the newly created basis function, or `NULL` if an error occured.

`RemoveBasis` removes a basis function from its model, but does not free the basis function. The relevent model and basis function parameters are updated.

`SaveModel` saves an analytical model to disk. The disk file includes all important parameters about the model, but does not include any statistics or information about the fit. This function returns 0 if it executed properly, 2 if the user cancelled the save procedure (due to a denied request to overwrite a previous file), or 3 if a write error occured. These are the same error codes used in DiskIO.c, and some of the code is similar.

`LoadModel` loads an analytical model from disk, from filename `name`, returning it in `mptr`. `mptr` should be a pointer to an unallocated model. `s` is the spectrum to be modelled, which may be `NULL`. The file format may be the same as what is saved by `SaveModel`, or may be somewhat different, since this uses reasonably flexible command parsing. This routine returns 0 for successful operation, or an error code between 1 and 6 (see DiskIO.c documentation), or 1000+ the line number where the file did not make sense. See the *SpectFit* documentation for the details of the file format.

`BasisMath` does assorted arithmetic on basis functions, returning the answer as a spectrum. The basis function input, `b`, and a string specifying the math function, `fn`, are required. As appropriate, `v` is assumed to be a spectrum or a basis function. If `k` is used, then `v` is ignored, and vice versa. The resulting spectrum is returned in `*ansptr`, which needs to be initiallized. If the value of `*ansptr` is not `NULL` initially, the spectrum pointed to it is automatically freed and it is sent back pointing to a new spectrum. The spectrum name is the same as the basis function name, except that a ‘s’ is appended to the name to denote a spectrum. The spectrum *x* values are evenly spaced, rannging from the model `xmin` to `xmax` with the model `dx` spacing. If the function was successful, it returns 0. Otherwise, `*ansptr` is returned as `NULL` and an error code is returned, chosen from one of the ones in the library errors.c (0,1,6,7,8,9). Only real spectra are allowed as inputs and are sent as outputs (a complex input spectrum results in error code 8). In most cases, points as which arithmetic that would return bad values, such as division by 0 or exponentiation that would overflow, are set to 0 and no error flags are set. The exception is division by the constant 0, in which case every point would fail, so error 9 is returned. Allowable function strings are:

```
exp b    b+s    s+b    b+b    b+k    k+b
10^b     b-s    s-b    b-b    b-k    k-b
ln b     b*s    s*b    b*b    b*k    k*b
log b    b/s    s/b    b/b    b/k    k/b
sqrt b                        b^k
```

In these strings, “`b`” represents a basis function, sent in with `b` if one is required and `b` and `v` if two are required, “s” represents a spectrum sent in with `v`, and “`k`” represents a float constant, sent in with `k`. The operations are obvious.

`ModelMath` is nearly identical to `BasisMath`, with the difference that it does arithmetic with at least one model. Allowable function strings are:

```
exp m    m+s    s+m    m+b    b+m    m+m    m+k    k+m
10^m     m-s    s-m    m-b    b-m    m-m    m-k    k-m
ln m     m*s    s*m    m*b    b*m    m*m    m*k    k*m
log m    m/s    s/m    m/b    b/m    m/m    m/k    k/m
sqrt m                                      m^k
```

`endbasis` frees the basis functions in the `basisfns` set, and then frees the set structure.

`rmserror` computes and returns the root mean square difference between a model and its spectrum. Possible error codes are the same as those for `prefit`.

`chisq` computes the chi square difference between a model and a spectrum, using all the spectrum *x* values that fall within the model limits. If `ch2max` is initially set to some value greater than or equal to 0, then the computation stops when chi sqared exceeds that value and returns `ch2max`; otherwise it continues over all relevant x values. Models are assumed to be complete, including spectra. The model `sigma` value is used in the compuation if it greater than 0 and is assumed to be 1 otherwise; uncertainties are also used if given.

`chisqD` is identical to `chisq`, but it also returns a vector and a matrix containing derivative information, with respect to parameters. An error of –1 is returned for failure to allocate memory. The input `np` is the number of unfrozen parameters in the model, `alpha` needs to be allocated to size `np x np`, and `beta` needs to be allocated with `np` elements. `np` should be 1 or greater. Initial `alpha` and `beta` values are ignored. Before returning `alpha` and `beta`, any error values (`Inf`, `NaN`) are replaced with 0's. The model sigma value and uncertainties, if given, are used for the returned value and for derivatives. The `alpha` and `beta` matrices are exactly as given in the *Numerical Recipies* discussion on non-linear fitting.

`prefit` checks that a model is setup correctly for fitting, returning an error code if not (–6 for no model, –16 for no spectrum, –8 for complex spectrum, or an error code from `CheckBasis`). If all is well, it allocates and prepares an array of pointers to fittable parameters for fitting routines, `**pptr`, and also allocates space for a vector of minimum parameter values, `*ploptr`, a vector of maximum values, `*phiptr`, and the covariance matrix, `*cptr`. The range vectors are set to the limits in `paramlo` and `paramhi` and the covariance matrix is cleared. It returns the number of unfrozen parameters, which is also the dimension of the array of pointers, the vectors, and of the matrix. If the function returns 0, then there are no fittable parameters and nothing was allocated. Negative return values, –1 and –6, are for a memory allocation error and an illegal input error.

`ModelCovar` calculates the covariance matrix and puts it in the model. It uses derivative information found with `chisqD`. If the model `sigma` value is ≤0 and there are no uncertainties, it temporarily replaces the `sigma` value with the rms error to yield a meaningful covariance matrix. See the *SpectFit* documentation. It returns 0 for correct operation and a standard error code otherwise.

`unfit` replaces the parameter values with the old parameter values. It also removes any covariance matrix.

`RandomFit` is a simple fitting routine that alters model parameters randomly to fit a spectrum. It starts at current parameter values and ends when several random tries fail to make an improvement. It is slow but robust. Basis function derivatives are not used in the fitting, but are used afterwards by `ModelCovar`. Return values are 0 for good operation, 1 for inability to allocate memory, or 6 for illegal input parameters. `RandomFit` may be stopped at any time by pressing a key.

`LinearFit` fits a model to a spectrum with the assumption that the model is linear with respect to the unfrozen parameters. After finding the optimal parameter values, they are corrected to be within the limits. If the final result is worse than the starting state, the parameters are returned to their initial values. Non-linear models are permissible, but may not yield improved fits. If fixed model parameters are linked to fittable ones, the linking is not accounted for in the fitting. It uses LU decomposition to find the best possible parameters within roundoff error, and then LU improvement to minimize the roundoff error. Return values are the same as for `RandomFit`, but also an error code of 9 is possible for divide by zero, implying a singular matrix.

`LMFit` performs Levenberg-Marquardt fitting, by a routine copied significantly from *Numerical Recipes*. If fixed model parameters are linked to fittable ones, then the derivative information is incorrect and the routine may be led away from the minimum rather than towards it. Thus, this may not work well for linked parameters, although this has not appeared to have been a problem so far. Otherwise, it generally works quite well. `LMFit` may be stopped at any time by pressing a key. Error codes are as for `RandomFit`.

`RandMultiFit` performs random walk type fitting on a set of models, simultaneously.
Error codes are as for `RandomFit`. It may be stopped at any time by pressing a key.
No covariance matrix is returned.