Documentation for *SpectFit*

Version 2.0, © October 2002

Steven Andrews

What is *SpectFit*?

SpectFit is a Macintosh program for fitting and manipulating one dimensional scientific data (one independent variable). For the most part, it is controlled through a command line interface, with output sent to a graphics window. Strong features are Fourier data analysis and highly versatile fitting methods. While *SpectFit* was written for infrared spectral anaysis, it is at least as useful for other types of data. It is free, open source, and runs on Macintosh OS X.

A fundamental design concept is that scientific data generally has a discrete number of data points, but is thought of as representing a continuous function (such as an absorption spectrum, a line profile from an image, etc.) As much as possible, *SpectFit* lets the scientist treat the data as a function and not worry about sample spacing and endpoint issues. For example, in *SpectFit* the *x* units of Fourier transforms have correct positive and negative frequency values, rather than the more common range which extends from 0 to *n* data points.

<u>Features</u>

linear and nonlinear fitting
complicated models can be created
multiple model fitting at once
linked fitting parameters
partially constrained parameters
can save and load analytical models
interactive model adjustment
convenient data arithmetic
Fourier filtering
complete documentation
automatic handling of different data spacing
can create data from equations

<u>Notable limitations</u>

mediocre user interface
no multi-dimensional data
cannot print graphics
new fitting functions are added to source code
no online help
only for Macintosh
not designed for large data sets

<u>new features (version 2.0)</u>
improved complex number support
most bugs fixed
undoable fitting
improved model file format
improved error reporting
fast fourier transform
uncertainties allowed for data points
parameter covariance matrix available
models can exist without data

I. Getting Started

Short Tutorial

You probably have some data and want to fit it. This example will show you how to do that. While you could try to follow the example with your own data, its probably easier the first time to use the data set supplied in the file name “sample1”. When you start *SpectFit*, you will see a text window and a blank graphics window; you will type commands in the text window. Your data needs to be in a text file, with the columns separated by spaces or tabs. Put the *x* column first and the *y* column second (other file formats are discussed later) and put the data in the same directory as *SpectFit*.

Type this	What’s happening
`a=load("sample1")`	The data is loaded into the data type variable called `a`.
`print a`	This tells a little about your data set, including the first and last points.
`plot a`	The data is plotted to the graphics window, although most of it is out of the visible region.
`scale`	Autoscale the graphics window to show the data. Note that *x* and *y* positions of the lower left and upper right corners are shown in the corners.
`afit=model(a)`	Define a blank model for the data, called `afit`. Models are analytic functions.
`add gaussian`	Models are composed of a sum of basis functions. These are pre-programmed functions, including linear ones like a quadratic, non-linear ones like a Gaussian, and several speciallized functions.
`plot afit`	Again, it’s partly out of the visible region.
`scale`	Autoscale to show the whole data set and model.
`print afit`	This tells a little about the model and about the basis function that you added. The basis function is named “`gaussian:0`”, where the suffix allows you to add more Gaussians without confusion.
`mean=30`	This changes the mean of the Gaussian from its default value of 0 to 30. (The number 30 was chosen based on the edges of the screen).
`fit`	Find the best fit.
`print afit`	The best fit parameters are displayed, along with their confidence intervals. However, from the graphics window, the model clearly doesn’t capture the data, so we’ll add more basis functions.
`add constant`	Add a constant offset.
`scale`	
`fit`	
`scale`	Clearly, the fit is much better, but now we want to get rid of the overall shape of the baseline.
`add sine`	
`print afit`	

`fit`	This was worth a try, but didn't do what was wanted.
`unfit`	Return to the previous parameters.
`tweak`	We'll change the sine parameters interactively. The upper left corner of the screen shows which parameter is being tweaked. Press the right arrow a couple times to scroll through the parameters until you see one called "sine:0.amp". Then, press the down arrow several times until it's around 0.04; when you overshoot, press the up arrow (repeating an arrow means that a larger step size is used each time, whereas alternating them yields a smaller step size). Press the right arrow to move on to the parameter "sine:0.freq", and adjust that to about 0.1. Then adjust "sine:0.shift to about 2.5 (you will have the press the up arrow lots of times). Press escape to stop tweaking.
`fit`	It should fit well.
`save afit`	Save the analytic model for future use, or to store a record of the best fit parameters. Choose a name, or type "cancel" if you don't want to save it.
`save data(afit)`	Also convert the model to a numerical data set, like `a`, and save that. This way it can be imported to Excel or some graphics program. Again, choose a file name or type "cancel".

Now, we'll clean up the data some to remove the fringes.

`unplot afit`	Remove the model from the graphics window.
`pow=ftpower(a)`	Calculate a frequency power spectrum of the data, called `pow`.
`plot pow`	
`scale pow`	The peak at 0 captures the dominant shape of `a`, while the little peaks on the sides represent the high frequency fringes. The *x* units are the inverse of the *x* units for `a`.
`mouse`	Click the mouse over the little peaks to see where they are. You'll see that they are at about ±4.5 and a bit under 0.5 units wide.
`unplot pow`	
`scale`	
`a2=filter(a,"notch",4.5,0.5)`	This filters the original data with a notch type filter, in which a few freqencies are cut out from the data. In this case, there is a notch centered at ±4.5 and with a width of 0.5, using the numbers we found previously.
`plot a2`	
`a2.color="blue"`	Change the color to make it more visible.
`unplot a`	Now it's obvious how much the data was cleaned up.
`clear pow`	Get rid of the power spectrum since we're done with it.
`exit`	The end.

SpectFit has a lot more capabilities than those shown here, but hopefully you have an idea of how it works at this point. You probably noticed a lot of repetitive typing during the example, such as the words "scale", "print", and "plot". A useful shortcut is that only the first letter or letters are needed, so rather than typing “scale” and “plot”, you can type just “s” and “p”. Also “?” is equivalent to “print”.

Something to watch out for

If you put the text window fully on top of the graphics window, and then select the graphics window, the text window goes behind it. The problem is that it’s impossible to get it out again. So, make sure this doesn’t happen (it would be a lot of work to fix this bug.)

II. Using *SpectFit*

Command interface

SpectFit is driven almost exclusively through a text interface, where the user types in commands, and the program executes them. *SpectFit* also displays data and results to a graphics window. There are two types of commands: procedures and assignments.

Procedures are used to control the program, arrange the graphics window, and manipulate existing variables. Examples of procedures:

```
plot a         print 5/3           scale        fit
```

Assignments either define a new variable or set the value of an existing variable or parameter. Examples of assignments:

```
abs=load("AbsData")         k=31         gft=fourier(g)
```

Data types

SpectFit supports four variable types: numbers, strings, data, and models. Another type is the basis function, but these cannot exist outside of a model, so they don't count as variables. It is not possible to create new variable types, nor is it possible to declare arrays.

Numbers are always unitless floating point numbers. Examples of numbers:

```
  a=5    b=(1+2)*3     size=gaussian:0.area       xlo=scale.xmin
```

Strings are just regular strings of text or numbers. There is no limit to the length of string variables. However, string parameters are limited to 256 characters, where these include things like the name of a data set, an equation used to link fitting parameters together, and the units of the *x* or *y* axis. Examples of strings:

```
s1="hello"     s2=s1+" world"      s3=model.name
```

A data set is a structured type including a name, a description, *x* and *y* units, a list of data, and other things. Data may be loaded, saved, plotted, and manipulated in many ways (smoothed, differentiated, added, subtracted, etc.). While data are interpolated and extrapolated as neccessary, they are fundamentally lists of discrete points. Since *SpectFit* was originally written for analyzing spectra, data sets are frequently referred to as spectra. Examples of data:

```
a=load("sample1")     d1=deriv(a)         res=model-a
```

Models are another kind of structured variable. In contrast to data, a model is an analytic function, defined as the sum of one or more basis functions (such as gaussians, exponentials, polynomials, etc.). The components of a model include its name, the data it describes, the range of *x* values where the model is defined, a list of basis functions, and other things. Because one typically wants to do a good deal of work on a single model, before moving on to another one, the word `model` is used to indicate the current model being modified. Much like a current directory, `model` can be set to other models as desired. Model examples:

```
    afit=model(a)          m=loadmodel("mymodel",a)
```

Basis functions are structured types within models. Each basis function has a name and a set of parameters that depend on the function. For example, a gaussian has three parameters: the area, mean, and standard deviation. A data set may also be used as a basis function, in which case the only variable parameter is the weighting of the data in the model.

Structure elements

Data and model variables are made up of many elements. These elements are referenced with a dot followed by the element name, so "`print a.file`" would return "`sample1`", if that was the file name. Dots are also used to get some useful information about a variable even if it isn't an actual element of the structure. For example the maximum value of a data set is found by "`print a.ymax`". Many elements may be set as well as just looked at, but this is not true for all of them. For example, it is possible to change the domain over which a model is defined since it is an analytic function ("`model.xmin=-10`"); however, it is not possible to change the domain of a data set since it is a data array ("`a.xmin=-10`" returns the error "`can't assign to left side`").

Following is a list of the most useful information that may be referenced. A complete list is included in the reference section. A mark in the changeable column denotes that the element may be set as well as read.

	reference		changeable	description
data	.file		•	file name, if the data has one
	.color		•	color string, only first letters matter
	.xmin			smallest x value
	.xmax			largest x value
	.ymin			smallest y value
	.ymax			largest y value
	.*value*			interpolated y value for the given x value
	.x	.*value*		nearest x data for the point number
	.y	.*value*	•	nearest y data for the point number
basisfn	.name		•	name of basis function
	.n			number of parameters, fixed and free
	.*param*		•	value of *param*
	.*param*	.eqn	•	equation string to link parameters
		.min	•	lower bound of parameter for fitting

	.max	•	upper bound of parameter for fitting
model	.color	•	color string, only the first letter matters
	.sigma	•	model weighting, or actual error size
	.xmin	•	lower end of modeled domain
	.xmax	•	upper end of modeled domain
	.dx	•	model spacing for plotting and saving
	.basisfn	•	a basis function in the model
scale	.xmin	•	left side of graphics window
	.xmax	•	right side of graphics window
	.ymin	•	bottom of graphics window
	.ymax	•	top of graphics window

More complex structures and structures with parentheses can be interpreted as well. As dots bind more tightly than arithmetic symbols, parantheses are sometimes needed to make references meaningful. Here are a couple examples of valid references:

```
model.constant:0.offset.eqn
a.((a.xmax+a.xmin)/2)
```

Operators

Some mathematics is supported by *SpectFit*, which follows the conventional precedence of operators. For virtually all binary operators (operators with two operands), the operands may be any combination of the supported types, other than strings. Also, the result of an operation is typically either a number or a data set. Thus, a model plus data is a data set. In order of decreasing precedence, the operators are:

operator	example	description
""	`"my data"`	delimit strings
()	`5*(3+4)`	force higher precedence
.	`model.line:0.slope`	elements of a structure
^	`spec^2`	raise to a power
* /	`gaussian:0/model`	multiply and divide
+ -	`spec+0.3`	add and subtract
=	`a=b=c`	make assignment to left side
;	`k=1;plot a`	do two commands sequentially

Data file format

SpectFit can read data from tables of text, where the *x* and *y* data are entered in separate columns. A data table may be created directly from Excel, Kaleidagraph, OPUS, MS Word, SimpleText, or any of many more standard software packages. Data columns should be separated by either a single space or a single tab and rows should be separated by carriage returns. While *SpectFit* can typically tell the difference between text in a file header and the data, it may be necessary to count the lines yourself. Here are a couple examples of data files and how to read them:

<u>file: “xydata”</u>

$x_0\ y_0$
$x_1\ y_1$
…
$x_{n-1}\ y_{n-1}$

```
a=load("xydata")
```

<u>file “data table”</u>

This is a 2 line header for the table.
Columns are t,x,y,z; I want t vs. z.

$t_0 \quad x_0 \quad y_0 \quad z_0$
$t_1 \quad x_1 \quad y_1 \quad z_1$
…
$t_{n-1} \quad x_{n-1} \quad y_{n-1} \quad z_{n-1}$

```
a=load("data table",1,4,2)
```

The arguments of the load function are the file name, the *x* data column number, the *y* data column number, and the number of lines to skip. The latter 3 arguments are optional; if they are omitted, then the default values are to assume *x* values are in column 1, *y* values are in column 2, and 0 lines are skipped. If there is a gap in the data table, *SpectFit* skips over it and continues reading. If your columns of data are separated by multiple spaces, then just tell *SpectFit* to load from a higher column number. If each row of the table has a different number of spaces between data points, then *SpectFit* won’t be able to cope and you will have to fix that elsewhere. Complex data cannot be loaded in a single statement, but can be loaded and assembled with a complicated statement like this:

```
a=complex(load("mydata",1,2),load("mydata",1,3))
```

SpectFit saves data using the `save` procedure, resulting in a column of *x* data and a column of *y* data, separated by single spaces, and with no file header. If the data set is complex, there are two columns of *y* data, for the real and imaginary components.

It is generally easiest to put the files to be read or written in the same folder as *SpectFit*, although it is also possible use standard Macintosh path notation. For this notation, a file in the same folder as *SpectFit* needs no prefix. A file on the desktop, called “data”, is accessed as `":data"` (doesn’t apply to OS X), a file on the hard drive called “Mac HD” is accessed as `"Mac HD:data"`, and a file in a folder in the hard drive is accessed as `"Mac HD:data folder:data"`. If the data file is in a folder and the folder is in the same directory as *SpectFit*, the file is at `"::folder:data"`.

Model file format

There are several ways to save a model. If you want to list the best fit parameters in your lab book along with their uncertainties, the easiest thing is to type `print model`, and then copy and paste the results into some other program, such as Microsoft Word. It is also often useful to save a numerical version of the model, which can be plotted in Kaleidagraph or some other graphics program, so people can see how good your fit is. In that case, convert it to a data set and save it: `save data(model)`. Finally, this section is really about saving and loading descriptions of analytic models.

Typing `save model` writes a description of the model to disk as a text file. The file is reasonably self-explanatory, but be aware that uncertainties and other fit statistics are not saved.

Models may be loaded from a text file using either a previously saved model, or one created with a text editor. The ability to define models with an editor rather than with *SpectFit* is a convenient way to save typing for complicated models. The file format is as flexible as possible, but this is still fairly rigid. Attempting to load incorrectly formatted files results in the message "error reading line #", where the bad line number is shown. In some circumstances, an error won't become apparent until later in the loading procedure. Models are loaded with the command:

```
afit=loadmodel("mymodel",a)
```

The second argument is the data set the model is intended to model. It is optional since it is now possible for models to exist without data sets. Clearly they cannot be fit in this case, but it is a useful way to manipulate analytic functions.

Here is an example of a model saved by *SpectFit* (which is the model created in the tutorial introduction):

```
# Model saved by SpectFit 2.0.

name: af
file: samplefit
color: r
# data modeled: a
# uncertainties: none
sigma: 0.000000
xmin: 1.000000
xmax: 100.000000
dx: 0.200000
# rms error: 7.783680e-03
basis: gaussian
 name: gaussian:0
 color: g
 desc: y=area/(std_dev*√2π)*exp(-(x-mean)^2/(2*std_dev^2))
 param: area
  value: 8.052529 ± 0.027272
 param: mean
  value: 26.815847 ± 0.014973
 param: std_dev
  value: 5.184603 ± 0.015257
  min: 0.000000
 endbasis
basis: constant
 name: constant:0
 color: g
 desc: y=offset
 param: offset
  value: 0.108604 ± 0.000445
 endbasis
basis: sine
```

```
name: sine:0
color: g
desc: y=amplitude*sin(frequency*x+shift)
param: amp
 value: 0.036178 ± 0.000597
param: freq
 value: 0.091987 ± 0.000511
param: shift
 value: 2.715276 ± 0.030701
endbasis
```

Clearly, *SpectFit* starts by saving general information about the model, followed by the details of all the basis functions. Each basis function starts with the word `basis:`, followed by the function. Then, the parameters and any constraints are listed, followed by `endbasis`. Several items are saved, although they are not loaded in if the model is reused. These are the lines that begin with `#` and the uncertainties on the fitting parameters. To create your own models, it is helpful to know the complete list of commands available.

<u>File statements</u>

`#` *text*	Comment line
`name:` *name*	Model name.
`color:` *color*	Model color string.
`sigma:` *sigma*	Model weighting or actual error size.
`xmin:` *xmin*	Upper end of modeled domain.
`xmax:` *xmax*	Lower end of modeled domain.
`dx:` *dx*	Model spacing for plotting
`end`	End of definition file (optional).
`basis:` *proc*	Add this basis function to the model, and start loading it.
`name:` *name*	Basis function name.
`color:` *color*	Basis function color string.
`desc:` *desc*	Basis function description.
`spec:` *spec*	Name of a data set that the function depends on. It must exist.
`param:` *param*	Start loading aspects of this parameter.
`value:` *value*	Value of the current parameter.
`min:` *min*	Minimum value of the current parameter.
`max:` *max*	Maximum value of the current parameter.
`eqn:` *eqn*	Equation for the current parameter.
`fix`	Fix the current parameter.
`free`	Free the current parameter.
`endbasis`	Stop loading this basis function, and return to model statements.

All commands are optional, with default values used if they aren't given. However, several things need to be checked. Make sure the model `xmin` value is less than the `xmax` value, and that `dx` is greater than 0. Also, constraints for the parameters of the basis function, the `min` and `max` values, must have `min<max`, if they are given. If the basis function requires a data set, such as the "spectrum" and "diffuse" basis functions, then the

data set needs to have been loaded in beforehand, using the same name that is listed in the model file.

Writing more stuff to disk

Finally, it is possible to write numbers or strings to a file with the write function:

```
ok=write("output",data.name,exp:0.slope)
```

The value returned from the function might as well be ignored since any error codes are also displayed as text.

Tweaking

While the fitting procedures are good at optimizing a fit, they often don't do very well if the parameters are far off initially. The solution is to adjust the model parameters manually before letting the fitting procedure take over, done with the `tweak` command. When you type `tweak`, you will see a parameter name listed in the upper left corner of the graphics window, which can be adjusted. The following keys are used:

↑	increase parameter
□	decrease parameter
□	tweak previous parameter
□	tweak next parameter
`-`	change sign of the parameter
`s`	autoscale to the data set
`m`	autoscale to the model
`a`	autoscale to both the data and the model
escape	stop tweaking

Pressing the up or down arrow several times in a row increases the step size of the adjustment, whereas alternating up and down arrows decreases the step size. This is supposed to allow both the ability to change a parameter rapidly but also to allow precise control.

Fitting

There are several fitting methods. However, all of them that are applicable to the situation should give very nearly the same result. They differ in how they get to the best fit, in their speed and robustness, and in their applicability.

Linear fitting is an exact method rather than an iterative one, so it is the fastest and the most accurate. However, it also requires a well behaved linear model. A linear model is one which can be written as

$$y = a_1f_1(x) + a_2f_2(x) + ... + a_nf_n(x) + f_{n+1}(x)$$

where the a_i coefficients are fittable parameters and there are no fittable parameters within the $f_i(x)$ functions. Well behaved means that the parameters are not highly covariant, as discussed in the next section. This routine uses simple matrix inversion, making it quite simple, but also sensitive to covariant parameters.

Random fitting is an iterative method. Starting with the initial parameter values, it randomly walks about in parameter space, keeping steps that decrease the fit error and backtracking any steps that increase the error. Since it doesn't use basis function derivatives, it works for basis functions that don't have derivatives (none currently) and it works when parameters are linked to each other using the ".eqn" field. For the latter reason, random fitting is the only method currently available that can fit multiple models simultaneously.

Levenberg-Marquardt (LM) fitting is another iterative method. However, instead of walking randomly, it uses the function derivatives to guide the parameter choices, typically leading to a fast attainment of the best fit. Linked parameters may cause a problem in its ability to find the best fit, because the the derivative information doesn't account for these links. This is the default fitting method.

Summary of fitting methods:

Fit method	speed	robust	requires linearity	linked parameters	multiple models	requires derivatives	conf. intervals
linear	best	good	yes	no	no	yes	yes
random	slow	best	no	yes	yes	no	yes
LM	good	good	no	maybe	no	yes	yes

Constrained fitting

Parameters of basis functions may be simple numbers that are free to vary during fitting. This is generally the default situation. Alternatively, they may be fixed to some value, so they won't change during fitting. This is done with the fix command, and undone with the free command,

```
fix quad:0.slope
fix mean
free gaussian:0
free model
```

Another option is to allow them to vary, but be constrained to physically reasonable values. This is done by setting the `.min` and `.max` elements of the parameters.

```
asinh:0.weight.max=30
quad:0.curve.min=0
```

Finally, parameters may be fixed not to a value, but to an equation. That equation may look at anything, including fittable parameters in the same model or fittable parameters in a different model. For example, one could fit a peak with a pair of gaussians with different areas and different standard deviations, but constrained to have the same mean. An expression to do this looks like:

```
    fix gaussian:1.mean
    gaussian:1.mean.eqn="afit.gaussian:0.mean"
```

Note that the equation is entered as a string. If it isn't, then the expression won't be accepted. The parameter needs to be fixed for the equation to be computed. Writing equations for complicated models tends to be tedious, so it might be easier to create the model in a text file, and then load it with the `loadmodel` command, as described above.

Multiple model fitting

Typically, there is no reason to fit two models simultaneously. The exception is if you have two (or more) data sets that are related, and you want to fit both of them separately, but the fit parameters in one need to be match the fit parameters in the other. For example, in Stark effect spectroscopy, one might fit an absorption spectrum with a Gaussian and also fit the corresponding Stark spectrum with the second derivative of the same Gaussian. The free parameters for the absorption spectrum are the Gaussian area, mean, and standard deviation. For the Stark spectrum, the area is a free and independent parameter, but the mean and standard deviation must be the same as those in the absorption spectrum. It is possible to fit one data set first, and then the other using the results from the first fit, but it is preferable to use all the information possible for all parameters. Thus, the models are fit simultaneously.

Models may be fit simultaneously with the `fit all` command. It is not currently possible to just fit a couple models and leave others unaffected, so models that should not be fit need to have all their parameters fixed.

Multiple fitting does not reduce the total error in a pair of fits, but is a way of partitioning it between the fits. Ideally, uncertainties are known for each data set, which are entered into the `model.sigma` element or the `model.uncert` list. However, if absolute uncertainties are not known, but it is known how they compare between the data sets, these relative error sizes may be used as the uncertainties. If no uncertainties are explicitly declared for each data set, *SpectFit* assumes that they are all equal, and the fits are likely to be incorrect.

A way of finding the absolute uncertainty is to run the experiment twice, yielding two data sets that differ only in the noise; the rms difference is the uncertainty. An alternate method is to calculate the rms error in a region away from the part being fit. Finally, the noise may be known from theory, such as shot noise being equal to the square root of the measured number of events.

Error estimates

SpectFit can display four statistics on the quality of the fit: the rms error, the parameter confidence interval, χ^2, and the covariance matrix. The first two are typically the most useful.

The rms error is essentially the average difference between the data and the model, in the same units as are on the *y*-axis. More precisely, it is the square root of the average of the squared differences between the data points and the model,

$$\text{rms error} = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}\left[y_i - f(x_i)\right]^2}$$

n is the number of data points, which are x_i,y_i pairs, and modeled with function $f(x)$. For the most part, the rms error is independent of the number of data points that are fit, making it directly comparable for fits of similar data sets. However, this breaks down when there are not a lot more points than fittable parameters, in which case the rms error underestimates the true error.

Parameter confidence intervals are defined in a round-about way. What they mean to say is that if you did the experiment many times and found the best fit for each data set, the average of each best fit parameter over all the data sets would be the true value, and there would also be a standard deviation for each parameter. This standard deviation is the confidence interval. However, you only have one data set. It turns out that equivalent confidence intervals can be estimated by seeing how much worse the fit gets when each parameter is adjusted slightly, which is quantified with the covariance matrix and is discussed below. The interpretation of the confidence interval is that there is a 68% chance that the true value of a parameter lies within one confidence interval of the best fit value, and a 95% chance that it lies within two confidence intervals. Using the covariance matrix, C_{ij}, the confidence intervals are the just the square roots of the diagonal elements,

$$\delta a_i = \sqrt{C_{ii}}$$

δa_i is the confidence interval for the *i*'th parameter. While each confidence interval is useful and accurate independently, they generally cannot be used meaningfully in combination. For example, the probability that two parameters are both within one confidence interval of their best fit values is typically not $68\%^2$=46%, since their values are often correlated. Also, note that the confidence intervals reported depend on whether you have assigned uncertainties to data points, as discussed below.

The χ^2 statistic is used to quantify how much of the fit error is due to noise in the data and how much is due to the shape of the model being incorrect. Thus, it is meaningless if there is no independent assessment of the noise in the data. There are two ways to tell *SpectFit* what the uncertainties are for the data: if they are the same for all data points, then set the model `sigma` value to the 1 standard deviation error for the *y* values ($\sigma \leq 0$ is the default and implies that σ is unknown). Alternatively, if the uncertainties are known for each data point separately, load those in as a data set and set the model `uncert` structure member to that data set: `model.uncert=aerror`. If you do both at once, the uncertainty values are multiplied. With these uncertainties, χ^2 can be calculated,

$$\chi^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - f(x_i)}{\sigma_i}\right)^2$$

χ^2 is unitless and, typically, in the vicinity of *n*. Ideally, a χ^2 value smaller than *n* implies that the data are better than expected and a larger χ^2 implies either that the data are worse than expected or that the model is inaccurate. However, this is only true if the σ values are accurate.

Finally we come to the covariance matrix, C_{ij}, which is calculated from the curvature matrix, α_{ij}. Continuing the discussion above for the confidence intervals, we can imagine lots of data sets being fit many times. Plotting all the best fit parameters in parameter space results in a cloud of points, centered at the location of the true parameter values. The density of the cloud has a multi-dimensional Gaussian distribution,

$$\text{probability}(a_1, a_2, \cdots) \propto \prod_{i,j} e^{-\frac{1}{2}\alpha_{ij}\left(a_i - a_{i,\text{true}}\right)\left(a_j - a_{j,\text{true}}\right)}$$

a_i is the value of parameter *i* and $a_{i,\text{true}}$ is the true value of parameter *i*. The inverse of the α_{ij} matrix is the covariance matrix C_{ij}. Comparing the equation above to that for the more familiar one dimensional Gaussian, in which the standard deviation is in the denominator of the exponent, it can be seen that the covariance matrix is, in a sense, the standard deviation of the cloud of best fit parameters values. Once again though, we only have one data set. It turns out that the shape of the χ^2 surface in parameter space, using a single data set, is the same as the shape of the cloud of best fit parameters using lots of data sets. Thus, the curvature matrix is the second derivative of the χ^2 function with respect to each pair of parameters,

$$\alpha_{ij} = \frac{1}{2}\frac{\partial^2\chi^2}{\partial a_i \partial a_j}$$

The previous equation for χ^2 relates these derivatives to the derivatives of the basis functions with respect the their parameters. If the uncertainties aren't known, neither χ^2 nor α_{ij} can be calculated. The solution is to use the rms error as the best available estimate of the uncertainties, which is what *SpectFit* does. With this, the best fit χ^2 is exactly equal to the number of data points, and so is meaningless and is not displayed. However, the curvature matrix can be calculated, as can the covariance matrix, where the implied assumption is that all differences between the model and the data are due to noise and not to a bad fit.

$$\mathbf{C} = \boldsymbol{\alpha}^{-1}$$

The covariance matrix can be displayed by typing "`print model.covar`".

For a more thorough understanding, the *Numerical Recipies* books are excellent, and most statistics books should cover this as well.

Fourier analysis

There are several Fourier transform options available. Slow summation with the `fourier`, `invfourier`, `ftpwer`, `realft`, and `realift` functions allow any number of data points. Fast methods with the `fft` and `invfft` functions require that the number of data points is an integer power of two, which is easily achieved with the `zerofill` function. In all cases, Fourier transforms assume the data are evenly spaced.

The forwards and inverse Fourier transform equations are

$$\hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(k) e^{ikx} dk$$

There is no consensus on where the factor of 2π belongs, but this is essentially the applied physics convention, and is what is used in *SpectFit*. As data are discrete points and only extend over a finite domain, transforming is a bit more complicated than these formulas suggest. Due to the discreteness, transforms cover a finite domain, satisfying the Nyquist relation, $k_{max}-k_{min}<2\pi/\Delta x$, where Δx is the data interval. The k_{min} value may be chosen, or the default value may be used, shown below. The finite domain of the data results in a discrete fourier transform, where the transformed data intervals are Δk,

	n even	*n* odd
$x_{max}-x_{min}$	$(n-1)\Delta x$	
Δk	$\frac{2\pi}{n\Delta x}$	
$k_{max}-k_{min}$	$(n-1)\Delta k = \left(1-\frac{1}{n}\right)\frac{2\pi}{\Delta x}$	
k_{min} default	$-\frac{\pi}{\Delta x}$	$-\frac{\pi}{\Delta x}+\frac{\Delta k}{2}$
k_{max} default	$\frac{\pi}{\Delta x}-\Delta k$	$\frac{\pi}{\Delta x}-\frac{\Delta k}{2}$

Rather than using the default limits for k, it is also common to set k_{min} to 0, in which case k_{max} is $(n-1)\Delta k$.

Because of the finite domain of the input data, the procedure needs to make some assumption about the data outside the given domain. The assumption made is that of periodic boundary conditions, meaning that it assumes that the data are repeated indefinitely (no reflections, just repeats). This is a problem for convolutions, and perhaps for other things, so the solution is to append a lot of zeros to the ends of the data using the `zerofill` command.

Command logging

By default, all commands are logged to the disk file "SFlog", which is a way of recording a session. One or multiple lines of the log file may be easily copied and pasted directly into *SpectFit* to repeat earlier commands. Also, the log file is useful for debugging purposes, since it is difficult to fix a program error if the user has forgotten what sequence of commands caused it. Logging can be turned off with the command "`unlog`" and turned back on again with "`log`".

Adding Basis Functions

Ideally, new basis functions could be plugged in trivially. However, they can't be. Instead, it requires writing a small amount of C code and recompiling *SpectFit*. The first thing to do is to look in the source file BasisFns.c, and see how current functions are written, which serve as good templates for new ones. Then go through the following steps:

1) Figure out exactly what the equation should be. Good equations are relatively simple, don't have redundant parameters, and don't have highly covariant parameters. Then figure out the derivatives of the equation with respect to each fitting parameter.
2) Write a routine for the function in BasisFns.c. This routine is given the x value where it's to be evaluated, the fixed and free parameters as an array with `param[0]`, `param[1]`, etc., a data set if one is required for the function, and either a blank vector in which derivative results are to be returned or `NULL` if derivatives aren't wanted. The function is supposed to return the y value and, if requested, derivatives with respect to the parameters. The returned value needs to be valid over all possible x values (for example $\sqrt{x}$ might return 0 for all $x<0$). Derivatives are less critical, but are supposed to be equal to 0 if they can't be computed somewhere.
3) Add a line to the header file BasisFns.h to declare the new routine.
4) Add a couple lines of code to the function `getbasis`, in the file BasisFns.c, so the new basis function will be recognized by the main program. These lines give the name of the function, a short description, the address of the code to be run, the total number of parameters, parameter names, and initial parameter values. After registering the function with `DeclareBasis`, it is also possible to fix or constrain some parameters.
5) Update the documentation below.
6) Recompile and run *SpectFit* and test the function. If it doesn't work right, fix it.

Possible additions

The capabilities of *SpectFit* largely match the needs of my research. The program will likely be updated as my research evolves, but it also may be expanded to be useful for other research applications as well. Features that might be added include:

- Savitzky-Golay smoothing
- mouse driven scaling and panning
- on-line help
- complete arithmetic support
- printing
- multi-dimensional data
- data integration methods
- fourier transforms of irregularly spaced data
- Simplex fitting
- better graphics
- deconvolution
- programming

SpectFit availability and citation

SpectFit, in both the compiled form and as source code, is available for free for all non-commercial uses. The source code, the compiled version of *SpectFit*, and this documentation are copyrighted by myself (Steven Andrews), with the exception of a few small portions of the code that have been copyrighted previously (portions copied from *Numerical Recipies in C*). No warrenty is made for the performance or suitability of either *SpectFit* or for any of the source code. The only portion of the code that may not be modified is the copyright information. Modifications should be noted in the code, and added to the documentation. If improvements are made or bugs are fixed, then I would appreciate a copy of the new source code.

I expect to maintain a working copy of the program indefinitely, and the Boxer lab may have complete copies as well. The current download site for *SpectFit* is http://sahara.lbl.gov/~sandrews/index.html. If *SpectFit* is used to a significant extent, it may be appropriate to cite or acknowledge its use.

Acknowledgements

In many ways, *SpectFit* was inspired by David Lambright's *Juluka* program. *Juluka* was an excellent fitting program, but became obsolete due both to the tremendous technological changes since it was written and to the lack of available documentation. When *SpectFit* was first being written, Steve Dudek worked on a user friendly C++ version of it. While that version was later abandoned, his effort is appreciated. Throughout the development of *SpectFit*, many members of the Boxer group have been supportive of this effort, have made useful suggestions, and inspired many of the program's features. While writing this program was initially discouraged by Steve Boxer, my graduate advisor, I appreciated that he let me write it anyhow. Had I been a couple orders of magnitude closer in estimating how long it would take to write, I probably would have followed his advice.

References

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Second edition. Prentice Hall. Englewood Cliffs, NJ. 1988.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipies in C: The Art of Scientific Computing*. Cambridge University Press. Cambridge. 1988.

III. Reference

Whereas previous sections of the documentation listed only the most useful components of *SpectFit*, this section has complete lists.

Structure elements

reference			changeable	description
spectrum	.name			name of spectrum
	.file		•	file name, if spectrum has one
	.desc		•	description, including list of changes
	.xunit		•	x units (a string)
	.yunit		•	y units (a string)
	.color		•	color string, only first two letters matter
	.n			number of data points
	.cmplx			1 if spectrum is complex, 0 if not
	.xmin			smallest x value
	.xmax			largest x value
	.ymin			smallest y value
	.ymax			largest y value
	.value			interpolated y value for the given x value
	.x	*.value*		nearest x data for the point index
	.y	*.value*	•	nearest y data for the point index; real data
	.yr	*.value*	•	nearest y to index; real for complex data
	.yi	*.value*	•	nearest y to index; imag. for complex data
basisfn	.name			name of basis function
	.proc			name of basis function procedure
	.color		•	color string, only the first letter matters
	.desc		•	description of basis function
	.model			model that owns the basis function
	.n			number of parameters, fixed and free
	.param		•	value of *param*
	.isspec			1 if a spectrum is required, 0 if not
	.spec		•	referenced spectrum, if there is one
	.param	.eqn	•	equation string to link parameters
		.pname	•	name of parameter
		.freeze		1 if parameter frozen, 0 if not
		.min	•	lower bound of parameter for fitting
		.max	•	upper bound of parameter for fitting
param	.eqn		•	equation string to link parameters
	.pname		•	name of parameter
	.freeze			1 if parameter frozen, 0 if not
	.min		•	lower bound of parameter for fitting

	.max	•	upper bound of parameter for fitting
model	.name		name of model
model	.file	•	file name of model
	.color	•	color string, only the first letter matters
	.spec	•	spectrum to be fit by the model
	.uncert	•	list of uncertainties, as a spectrum
	.n		total number of parameters in model
	.covar		covariance matrix from the previous fit
	.sigma	•	model weighting, or actual error size
	.xmin	•	lower end of modeled domain
	.xmax	•	upper end of modeled domain
	.dx	•	model spacing for plotting and saving
	.basisfn		a basis function in the model
scale	.xmin	•	left side of graphics window
	.xmax	•	right side of graphics window
	.ymin	•	bottom of graphics window
	.ymax	•	top of graphics window

Procedural commands

There are fairly few commands, but most of them can do many things. They are all listed below. To save typing, most commands may be entered with only the first letter or letters, if desired. Thus, rather than typing the word "scale", it is completely equivalent to type "s", "sc", "sca", etc. Another shortcut worth knowing is that the command "print" may be entered with a "?", using the BASIC tradition. A couple commands require the entire word to prevent mistakes, including "exit" and "kill".

<u>short commands</u>

help	Does nothing useful.
exit	Exit *SpectFit*. Nothing is saved automatically.
log	Turn on command logging to append to disk file "SFlog".
unlog	Turn off command logging.
/*text*	Remark. Does nothing.
kill *filename*	Kill a disk file.
exec *string*	Execute the string, as though it were a regular command. Example: `exec "a=a+1"`. It's not generally useful.

eval *string*	This is mostly for program development, or for finding why errors occur. It evaluates the string as though it were a regular expression and displays the type of the result.

<u>variable manipulation</u>

clear...	Clear a variable from memory.
spectrum	Clear a spectrum, as well as models and basis functions that use it.
model	Clear a model and its basis functions.
basis	Remove a basis function from the current model. Same as remove *basis*.
number	Clear a number variable.
string	Clear a string variable.
spectra	Clear all spectra.
model	Clear the current model.
models	Clear all models.
numbers	Clear all number variables.
strings	Clear all string variables.
plot	Clears everything from the graphics window. Same as unplot all.
all	Clear all variables.

save...	Save data to disk. In all cases, previously saved versions are overwritten. Spectra and models have a ".file" structure element which is used for the file name, if available; otherwise the user is asked for a file name.
spectrum	Save a spectrum.
model	Save an analytical model.
spectra	Save all spectra.
model	Save the current model.
models	Save all models.
all	Save all spectra and all models.

<u>display</u>

print...	Prints stuff on the screen. A shorthand for the word "print" is "?". If nothing follows print, a blank line is printed.
spectra	Print a list of defined spectra.
basis	Print a list of basis functions that may be added to the model.
models	Print a list of defined models.
numbers	Print a list of defined numbers, along with their values.
strings	Print a list of defined strings, along with their contents.
model	Print the basis functions in the current model, along with their parameters.
params	Print the current model parameters, in pasteable form.
all	Print a summary of all variables.

plot	Print a list of the things that are plotted.
scale	Print the plot scale limits.
spectrum	Print the header of a spectrum, and a couple of the data points.
basis	Print the parameters of a basis function in the model.
model	Print the contents of a model.
number	Print a number.
string	Print a string.

plot...	Plots stuff to the graphics window. The window is not automatically rescaled. If nothing follows plot, then the graphics window is redrawn.
model	Plot the current model, using current parameters.
all	Plot all spectra and all models currently existing.
spectrum	Plot a spectrum.
basis	Plot a basis function in the model.
model	Plot a model.

unplot...	Removes stuff from the graphics window.
model	Remove the model from the plot.
all	Remove everything from the plot.
spectrum	Remove a spectrum.
basis	Remove a basis function.
model	Remove a model.

scale...	Scale the graphics window in one of several ways. If nothing follows scale, then autoscale to show all spectra and all models.
models	Scale to all models currently plotted.
spectra	Scale to all spectra currently plotted.
model	Scale to the current model.
all	Scale to show everything currently plotted.
x	Scale just the x axis.
y	Scale just the y axis.
spectrum	Scale to the size of the spectrum, whether it is plotted or not.
basis	Scale to a basis function, whether it is plotted or not.
model	Scale to a model, whether it is plotted or not.
number	Zoom scale in or out by the value of the number. A number greater than 1 zooms out, less than 1 zooms in.

mouse	Displays the mouse position when it is clicked. Press a key to exit.

<u>model commands</u>

add...	Add an analytic basis function to the current model. It is named the basis function name, concatanated with a :#, where the # is a 0, 1, 2,..., allowing a basis function to be added to the model several

	times, with different parameters in each version. It is generally desirable to set the parameters immediately after adding a basis function.
procedure	Look for a basis function in the master list with that procedure name, and add it to the model using default parameter settings.
basis	Add a copy of a previously declared basis function, but make the name unique.
model	Add copies of all the basis functions from the named model (which may not be the current model).
spectrum	Add a spectrum to the current model. Spectra are interpolated to make them continuous basis functions. If the spectrum variable changes (*e.g.* `a=a*2`), then the basis function changes as well. The new basis function is named the spectrum name, appended with a ":0". Note that a spectrum can only be added to a model once.
remove...	Remove a basis function from the current model.
basis	Remove an analytic basis function. This is the same as clearing it.
tweak...	Tweak parameters to visually fit a spectrum. Left and right arrows are used to choose the parameter for tweaking, which is displayed in the graphics window; up and down arrows increase or decrease its value. Press '-' to change the sign of the parameter, 'm' to autoscale to the model, 's' to autoscale to its spectrum, or 'a' to autoscale to everything. If nothing follows tweak, then the current model is tweaked.
model	Tweak parameters of the current model.
model	Tweak parameters of the named model.
fix...	Fix model parameters so they won't be optimized during fitting. This is also required to make parameter equations active.
model	Fix all parameters of the current model.
models	Fix all parameters of all models.
all	Fix all parameters of all models.
model	Fix all parameters of a model.
basis	Fix all parameters of a basis function.
basis.param	Fix just the parameter listed.
param	Fix just the parameter listed.
free...	Unfix model parameters so they can be optimized during fitting. This also makes parameter equations inactive, so they are ignored for calculating and fitting.
model	Free all parameters of the current model.
models	Free all parameters of all models.
all	Free all parameters of all models.
model	Free all parameters of a model.
basis	Free all parameters of a basis function

basis.param	Free just the parameter listed.
param	Free just the parameter listed.
fit...	Fit a model to its spectrum. If nothing follows fit, then the current model is fit, using Levenberg-Marquardt (LM) fitting.
model	Fit the current model using LM fitting.
LM	Fit the current model using LM fitting.
random	Fit the current model, using a random walk fit method. This is not a very good method, but it does do non-linear fitting and it was easy to program.
linear	Fit the current model with a linear fit. This is a good routine, the most accurate and the fastest, but it requires that all unfrozen parameters be linearly independent. If they aren't, then the program may crash. Parameters are linearly independent if the optimum fit may be expressed as the solution to an overdetermined matrix solution. For example polynomials and sums of spectra are linear, gaussians and exponentials are nonlinear.
all	Simultaneously fit all models, using a random walk method.
model	Fit a model with LM fitting.
unfit...	Undo the previous fitting operation for the given model. Only one level of unfitting is possible; to redo the fit, type "unfit" again. If nothing follows unfit, the current model is unfit.
model	Unfit the current model.
model	Unfit the model listed.
update...	Make sure all models are updated and any errors in parameter equations are reported. This is a temporary fix, since errors aren't being reported otherwise. If nothing follows update, the current model is updated.
model	Update the current model.
model	Update the model listed.

Assignment commands

The following list of assignment commands uses these abbreviations: *S* is a spectrum, *F* is a number (*F* because it's floating point), *M* is a model, *B* is a basis function, and *C* is a string (*C* for characters). Where multiple letters are combined, the function can handle any of them as parameters. Thus, *S*=exp(*SBM*) means that a spectrum, basis function, or model may be exponentiated and the result is a spectrum. Lower case letters indicate optional arguments, where default values are used if the arguments aren't given. As usual, commands may be nested; for example, `a=baseline(load("abs"),"ends")` is legitimite.

An assignment to note is the `spec()` assignment, which creates a new spectrum out of pretty much anything, such as a model, a string, or a number.

Some functions allow either complex or real spectra, some allow only real ones, and some allow only complex ones. The only way to know is to try and see what happens.

Basic assignments and manipulation

S=*S*	Copy of a spectrum, or result of a calculation.
F=*F*	Copy of a number, or result of a calculation.
M=*M*	Copy of a model.
M=model	Copy of a current model.
model=*M*	Set the current model to *M*.
C=*C*	Copy of a string, or result of a calculation.
S=load(*C*,f_1,f_2,f_3)	Load a spectrum from disk, with the given file name. f_1 is the column of the x data, f_2 is the column with the y data, and f_3 is the number of header lines to skip.
M=model(*s*)	Create a model for a spectrum.
M=loadmodel(C,*s*)	Load a model from disk with the given file name. *s* is the spectrum to be modeled.
S=spec(*SBMFC*,f_1,f_2,f_3)	Copy a spectrum, basis function, model, number, or string to a spectrum, where resulting *x* values are from f_1 to f_2 in steps of f_3. If the first term is spectrum, it is copied and interpolated as needed. If the term is a basis function or model, the new spectrum is calculated analytically. If the term is a number, the result is a constant value. If the first term is a string, it is used as an equation. The string may include any numbers or variables or operators, as usual. Also, if it contains "x" and x isn't already declared as something else, then it refers to values along the *x*-axis. For example, `spec("5*x^2+2",-10,10,1)` is a spectrum type in the shape of a parabola, defined from –10 to 10, in steps of 1.
S=merge(*S*,*S*)	Combine two spectra with different domains into a single spectrum, with a smooth transition from one to the other in any overlap region.

Arithmetic

FS=exp(*FSBM*)	Base *e* exponential.
FS=exp10(*FSBM*)	Base 10 exponential.
FS=ln(*FSBM*)	Natural log.
FS=log(*FSBM*)	Base 10 log.
SF=sqrt(*FSBM*)	Square root.
S=timesx(*S*)	Multiply y values by x values.
S=divx(*S*)	Divide y values by x values.
S=expx(*S*)	Exponentiation of x values, y values unchanged.
S=exp10x(*S*)	Base 10 exponentiation of x values, y values unchanged.
S=lnx(*S*)	Natural log of x values, y values unchanged.

S=logx(S)	Base 10 log of x values, y values unchanged.
S=sqrtx(S)	Square root of x values, y values unchanged.
S=shiftx(S,F)	f is added to x values, y values unchanged.
S=scalex(S,F)	x values are multiplied by f, y values unchanged.
S=powx(S,F)	x values are put to the f power, y values unchanged.

<u>Calculus</u>

S=deriv(S)	Take first derivative of a spectrum, without any smoothing. May also be called with S=deriv1(S).
S=deriv2(S)	Take second derivative of a spectrum, without any smoothing.
S=xderiv(S)	Divide spectrum by x values, take derivative, and then multiply by x values again. This is useful for Stark effect fitting. May also be called with S=xderiv1(S).
S=xderiv2(S)	Divide spectrum by x values, take second derivative, and multiply by x values.
S=integral(S,f)	Integrate a curve, with the integration constant chosen so it is 0 at x=f. The x-axis is used as the baseline.
F=integrate(S,f_1,f_2)	Integral of S from f_1 to f_2. Default values of f_1 and f_2 are the ends of the spectrum domain.

<u>Non-fourier signal processing</u>

S=smooth(S,F)	Smooth a spectrum by convolving it with a Pascal's triangle (sort of a discrete Gaussian). The value is the number of points on each side of the triangle.
S=mask(S,"notch",F_1,f_2,f_3,f_4,…)	Spectrum multiplied by transmitting mask with gaussian notches. Notches are centered at f_1,f_3,…, and have standard deviations of f_2,f_4,…
S=mask(S,"gauss",F)	Multiply by gaussian mask centered to pass at zero, with standard deviation F.
S=mask(S,"lowpass",F_1,f_2)	Multiply by trapezoidal mask with cut-off centered at $\pm F_1$ and cut-off region width F_2.
S=mask(S,"highpass",F_1,f_2)	Multiply by inverse trapezoidal mask with cut-on centered at $\pm f_1$ and cut-on region width f_2.
S=baseline(S,"ends")	Baseline correct spectrum with a straight line, such that the ends are set to zero.
S=baseline(S,"left")	Baseline correct spectrum with a constant value, such that the left end is set to zero.
S=baseline(S,"right")	Baseline correct spectrum with a constant value, such that the right end is set to zero.
S=noise(S,f)	Gaussian noise with standard deviation f; S is used only to defined the point positioning.
S=convolve(S_1,S_2)	Convolves S_1 with S_2, where S_2 is best thought of as the convolution kernal. S_1 needs to have uniformly spaced points.

Fourier signal processing

S=complex(S,s)	Converts spectra to real and imaginary parts of a complex spectrum.
S=real(S)	Real part of a complex spectrum.
S=imag(S)	Imaginary part of a complex spectrum.
S=zerofill(S,"ends",f)	Add zeros to both ends of s, expanding domain by factor of f. If $f \leq 0$, expand to next integer power of 2.
S=zerofill(S,"left",f)	Add zeros to left side of s, expanding domain by factor of f. If $f \leq 0$, expand to next integer power of 2.
S=zerofill(S,"right",f)	Add zeros to right of s, expanding domain by factor of f. If $f \leq 0$, expand to next integer power of 2.
S=fourier(S,f)	Fourier transform of a complex spectrum, starting at $k=f$. This function, as well as all fourier methods below, assumes uniformly spaced points. If f is not given, the result is centered about 0.
S=invfourier(S,f)	Inverse fourier transform of a complex spectrum, starting at $x=f$, or centered about 0.
S=ftpower(S)	Power spectrum of fourier transform, centered about zero. Spectrum is automatically baseline corrected first.
S=fft(S,f)	Fast Fourier transform of a complex spectrum, starting at $k=f$, or centered about 0. The number of data points in S needs to be an integer power of 2.
S=invfft(S,f)	Inverse fast Fourier transform of a complex spectrum, starting at $x=f$, or centered about 0. The number of data points in S needs to be an integer power of 2.
S=hankel(Sf)	Hankel transform of a real spectrum, which starts at 0. f is amount of interpolation that should be done (10 is default).
S=realft(S)	Fourier transform of a real spectrum, starting at $k=f$, or centered about 0, of which only the real portion is returned.
S=realift(S)	Inverse Fourier transform of a real spectrum, starting at $x=f$, or centered about 0, of which only the real portion is returned.
S=filter(S,C,f_1,f_2,f_3,f_4,…)	Frequency filter applied to spectrum, with same methods and parameters as for mask command. The fourier components of the baseline are not filtered out.

Programming

$FSBMC$=input(C)	Displays the string, waits for an input from the user, evaluates the input, and assigns that result to the left hand side.
F=write(C,fc,fc…)	Writes either numbers or strings to a single row of a disk file named C. Function returns 0 for successful operation, or disk specific error code (see documentation for DiskIO library).

Useless assignments

S=new(S,"1",F_0,F_1,F_2)	Create a new spectrum from F_0 to F_1 in steps of F_2, with all y values equal to one. S is ignored. The "spec" command is equivalent.
S=new(S,"x",F_0,F_1,F_2)	Create a new spectrum from F_0 to F_1 in steps of F_2, with all y values equal to the x values. S is ignored. The "spec" command is equivalent.
S=copy(S,C,F_0,F_1,f_2)	Copy a spectrum from F_0 to F_1 in steps of f_2. C is ignored. The "spec" command is equivalent.
S=unbaseline(S,"ends",F_1,f_2)	Remove previous baseline correction, making left end equal to F_1 and right end equal to f_2.
S=unbaseline(S,"left",F_1)	Remove previous baseline correction, adding F_1 to the whole spectrum.
S=unbaseline(S,"right",F_1)	Remove previous baseline correction, adding F_1 to the whole spectrum.

Current basis functions

Following is a list of current basis functions, including the basis function name, the name of the function in the source code, a description, and the parameters. Parameters list initial values and a brief description.

constant	`constbasis`	A simple constant offset.
y=offset		
offset	1	The amount of offset.
spectrum	`spectbasis`	The value of a spectrum, interpolated as necessary.
*y=weight**spectrum(*x*)		
weight	1	Weighting factor.
line	`linebasis`	A straight line through (*x0*,0).
*y=slope**(*x–x0*)		
slope	0.001	The slope of the line.
x0	0	*x* position where the line crosses the *x*-axis.
exp	`expbasis`	Exponential function.
*y=factor**exp(*slope***x*)		
factor	1	The pre-exponential factor.
slope	0.1	Exponential slope.
log	`logbasis`	Natural log function.
*y=weight**ln(*slope***x*+*intercept*), or 0 if argument is ≤0		
weight	10^{-4}	Function weight.
slope	1	Slope of argument.
intercept	1	Intercept of argument.

quad `quadbasis` A quadratic in standard format.

$y=\mathit{curve}*(x-\mathit{x0})^2+\mathit{slope}*(x-\mathit{x0})+\mathit{intercept}$

curve	10^{-6}	Curvature.
slope	10^{-5}	Slope.
intercept	0.01	*y*-intercept.
x0	0	*x* shift.

asinh `asinbasis` Inverse hyperbolic sine function.

$y=\mathit{weight}*\mathrm{asinh}(\mathit{slope}*x+\mathit{intercept})$

weight	10^{-4}	Function weight.
slope	1	Slope of argument.
intercept	1	Intercept of argument.

gaussian `guassbasis` A standard Gaussian.

$y=\mathit{area}/(\mathit{std\_dev}*\sqrt{2\pi})*\exp[-(x-\mathit{mean})^2/(2*\mathit{std\_dev}^2)]$, or 0 if *std_dev* is 0

area	1	Total area of Gaussian.
mean	0	Mean of Gaussian.
std_dev	1	Standard deviation of Gaussian, ≥0.

xgauss `xgaussbasis` Gaussian times *x*; useful for heterogeneosly broadened spectral lines.

$y=x*\mathit{area}/(\mathit{std\_dev}*\sqrt{2\pi})*\exp[-(x-\mathit{mean})^2/(2*\mathit{std\_dev}^2)]$, or 0 if *std_dev* is 0

area	0.0002	Similar, but not equal, to the area.
mean	1945	Close to the mean.
std_dev	4	Close to the standard deviation, ≥0.

sine `sinbasis` Sine wave.

$y=\mathit{amp}*\sin(\mathit{freq}*x+\mathit{shift})$

amp	1	Amplitude, baseline to peak.
freq	1	Frequency, in radian units.
shift	0	Phase shift, in radians.

lorentz `lorentzbasis` A standard Lorentzian.

$y=\mathit{max}/\{1+[(x-\mathit{mean})/(\mathit{fwhm}/2)]^2\}$

max	1	Peak height.
mean	0	Peak center.
fwhm	1	Full width at half maximum, ≥0.

peak `peakbasis` An *x*-weighted sum of a Gaussian and a Lorentzian. Useful for spectroscopy, with homogenously and heterogeneously broadened lines.

$y=x/\mathit{position}*[(1-\mathit{shape})*\mathrm{gauss}(x-\mathit{position})+\mathit{shape}*\mathrm{lorentz}(x-\mathit{position})]$

$\mathrm{gauss}(x)=\exp(-4*\ln(2)*x^2/\mathit{width}^2)$

$\mathrm{lorentz}(x)=1/(1+4*x^2/\mathit{width}^2)$

height	1	Maximum peak height.
position	2250	Peak center, ignoring skewing.
fwhm	10	Full width at half maximum, ≥0.

shape	0.5	Fraction of height that is from lorentzian, 0 to 1.

peakd1 `peak1basis` First derivative of a peak function, using *x*-weighted differentiation. Useful for Stark effect fitting.

$y=x*\{\partial/\partial x\ [\mathrm{peak}(x)/x]\}$

peak(x) is defined by the "peak" basis function

height	1	Maximum of the peak that is differentiated.
position	2250	Center of the peak that is differentiated.
fwhm	10	FWHM of the peak that is differentiated, ≥0.
shape	0.5	Shape of the peak that is differentiated, 0 to 1.

peakd2 `peak2basis` Second derivative of a peak function, using *x*-weighted differentiation. Useful for Stark effect fitting.

$y=x*\{\partial^2/\partial x^2\ [\mathrm{peak}(x)/x]\}$

peak(x) is defined by the "peak" basis function

height	1	Maximum of the peak that is differentiated.
position	2250	Center of the peak that is differentiated.
fwhm	10	FWHM of the peak that is differentiated, ≥0.
shape	0.5	Shape of the peak that is differentiated, 0 to 1.

peakz `peakzbasis` Sum of zeroth, first and second derivatives of a peak function. Useful for Stark effect fitting.

$y=z0*\mathrm{peak}(x)+z1*x*\{\partial/\partial x\ [\mathrm{peak}(x)/x]\}+z2*x*\{\partial^2/\partial x^2\ [\mathrm{peak}(x)/x]\}$

peak(x) is defined by the "peak" basis function

z0	0.0001	Zeroth derivative contribution.
z1	0.001	First derivative contribution.
z2	0.01	Second derivative contribution.
height	1	Maximum peak height.
position	2250	Peak center, ignoring skewing.
fwhm	10	Full width at half maximum.
shape	0.5	Fraction of the height that is lorentzian contribution.

diffuse `diffusebasis` A standard gaussian multiplied by a spectrum. Useful for finding diffusion values, using Fourier transforms of the concentrations at two times.

$y=\mathrm{spectrum}(x)*area*\exp(-dt*x^2)$

area	1	Area of Fourier transformed Gaussian.
dt	1	Diffusion constant times time.

diffuse2 `diffuse2basis` A squared error function. Was used for diffusion out of a square 2-D box.

$y=c0*\mathrm{erf}^2\{\sqrt{[td/(x-t0)]}/4\}$, or $c0$ if $x\le t0$

c0	100	Initial value.
td	1	Diffusion time, equal to box area/diffusion const.
t0	0	Time diffusion starts.

decay `convexpbasis` Convolution of a gaussian with an exponential that turns on at *x*=0. Useful for pump-probe spectroscopy, where the pump beam autocorrelation is the Gaussian and the time response is the exponential.

y=convolution of {$height$*exp($-kt$), or 0 if t<0} with $1/[□\sqrt{(2\pi)}]$*exp$[-t^2/(2□^2)]$

=$height$/2*exp($-kt+□^2k^2/2$)*{1+erf[$t-□^2k/(□\sqrt{2})$]}

$k=1/tau$, and is decay rate

$□=fwhm/[2\sqrt{(2 \ln 2)}]$, and is standard deviation of autocorrelation

$t=x-shift$, and is time since start of exponential

height	1	Initial height of exponential.
fwhm	0.16	FWHM of gaussian, with unit area.
tau	1	Time constant of exponential
shift	10	*x* value where exponential turns on.

rational `rationbasis` A rational function, which fits almost anything. Note that there is redundancy in the equation, so some parameters should be fixed.

$y=(n0+n1*x+n2*x^2+n3*x^3+n4*x^4+n5*x^5)/(d0+d1*x+d2*x^2+d3*x^3+d4*x^4+d5*x^5)$

n0	1	Numerator constant coefficient.
n1	0	Numerator linear coefficient.
n2	0	Numerator quadratic coefficient.
n3	0	Numerator cubic coefficient.
n4	0	Numerator quartic coefficient.
n5	0	Numerator quintic coefficient.
d0	1	Denominator constant coefficient.
d1	0	Denominator linear coefficient.
d2	0	Denominator quadratic coefficient.
d3	0	Denominator cubic coefficient.
d4	0	Denominator quartic coefficient.
d5	0	Denominator quintic coefficient.

sigmoid `sigmoidbasis` A generalized sigmoid curve.

$y=min+(max-min)/(1+10\text{^}(slope*(ec50-x)))$

min	0	Minimum value.
max	1	Maximum value.
slope	1	Slope of section where curve rises.
ec50	–1	log10 of ec50, which is position of rise.

IV. Source Code Documentation

This section of the documentation contains a detailed description of the main body of the *SpectFit* code, in the file `SpectFit.c`. While this is hopefully not useful to most program users, it is essential for program maintainance and any significant program updates. Note though, that new basis functions can be added quite easily, as described above, without a knowledge of the rest of the code.

Brief history of the source file: First parts were written 9/98, significant work was done 1/99, and more work 6/99. This largely completed the program, although several minor additions were added afterwards. Code was cleaned up and the documentation was improved 3/02.

Known bugs

None as of 10/28/02.

Required files

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "errors.h"
#include "Rn.h"
#include "RnSort.h"
#include "Spectra.h"
#include "Plot.h"
#include "Set.h"
#include "BasisFn.h"
#include "DiskIO.h"
#include "Utility.h"
#include "math2.h"
#include "VoidComp.h"
#include "SpectFit.h"
```

As much as possible, the code in `SpectFit.c` deals with simple communication with the user and with command parsing, leaving the numerical operations for the libraries. The two libraries that are used most are `BasisFn.c`, which contains code for the basis functions, models, and all fitting, and `Spectra.c`, which contains code for spectral manipulations.

Structures

```
struct result{
   void *val;
   void *valptr;
   char type;
   int temp;
   int var;
```

```
    int er; };
```

The dominant structures in *SpectFit* are spectra, models, and basis functions, all of which are declared in the libraries and described in their documentation. The only structure declared in `SpectFit.c` is a `result`, shown above, which is used for passing around results of parsed expressions.

The `val` member is the result of the evaluation. `valptr` is a pointer to the where the result is stored, if it is stored somewhere. This is for useful for the assignment of parameters where the location of the answer is needed, but not its value. `type` is a one letter capital character that describes the type of the answer. `temp` is 1 if the result was created (and allocated) in the evaluation routine and 0 if it was created elsewhere. If `temp` is 1, then the result should be freed after use. Typically, if `temp` is 1, then `valptr` is `NULL`, and if `temp` is 0, then `valptr` is defined. `var` is 1 if the answer is an actual variable, and 0 if it is not a variable or if it is just a reference to a variable. `er` is a standard error code, where 0 means no error and any other number indicates an error, using the list of standard errors in the library `errors.c`. Typically, errors will be posted when the type is '?' and not otherwise, but this is not necessarily true. Here is a full list of what can be expected:

type	val	valptr	temp	var	meaning	example
`' '`	`NULL`	`NULL`	0	0	blank expression	
`'?'`	-	-	-	-	syntax or other error	`3+(2`
`'S'`	`sptr`	`NULL`	1	0	new spectrum	`load("data")`
	`sptr/NULL`	`sptr*`	0	0	reference to spectrum	`model.spec`
	`sptr`	`NULL`	0	1	spectrum variable	`abs`
`'F'`	`float*`	`NULL`	1	0	new number	`5+3`
	`float*`	`float*`	0	0	float parameter	`scale.xmin`
	`float*`	`NULL`	0	1	float variable	`num`
`'W'`	`char*`	`NULL`	1	0	new word	`vse`
	`char*`	`NULL`	0	0	a word result/parameter	
`'C'`	`char*`	`NULL`	1	0	new string	`"a string"`
	`char*`	`char*`	0	0	string parameter	`abs.xunit`
	`char*`	`NULL`	0	1	string variable	`str`
`'B'`	`basisptr`	`NULL`	0	0	basis function	`gaussian:0`
`'M'`	`modelptr`	`NULL`	1	0	new model	`model(abs)`
	`modelptr/NULL`	`modelptr*`	0	0	model parameter	`line:0.model`
	`modelptr`	`NULL`	0	1	model variable	`afit`

Other facts: String parameters (`valptr` defined) are always allocated and always with `STRCHAR` characters, although other strings may be different sizes. Note that `valptr` for a string is a `char*` and not a `char**`, and similarly, `valptr` for a number is a `float*`.

Constants and global variables

```
set Spectra,Plotlist,Models,Numbers,Strings;
modelptr Model;
int Logcmd;
int Need2Plot,Need2Update;
```

No constants are defined in the SpectFit.h header file, which would be available elsewhere. However, a couple constants that are defined elsewhere are used here, including `MAXARG` and `STRCHAR`. The naming convention is that pre-processor defined constants and macros are in all capitals, global variables start with a capital letter, and local variables use all lower case letters.

`MAXARG` is a pre-processor defined constant, defined in `Spectra.h` and used here as well. It is the maximum number of arguments allowed for a function in an assignment command. For example, `smooth(spectrum,value)` has two arguments. `MAXARG` is not used for anything else.

`STRCHAR` is a pre-processor defined constant, defined in `Strings.h`, allowing the use of a standard length string. Essentially all strings are allocated to this size.

`Spectra` is the set of spectra currently loaded. Note that sets are defined in Set.c, and are basically an unsorted collection of objects, stored with a linked list.

`Plotlist` is a list of everything that should be plotted to the graphics window, including spectra, models, and basis functions.

`Basisfns` are the basis functions that are avaiable for adding to models. They are not added directly, but are copied when a basis function is needed.

`Models` is the list of all the models currently defined.

`Numbers` is a list of all number variables currently defined, which are all floating point real numbers.

`Strings` lists the string variables.

`Model` is used to point to the model which is currently being used, or is `NULL` if no models are defined. The current model is also a member of the `models` set.

`Logcmd` is 1 if commands are to be logged to disk, and 0 if not.

`Need2Plot` is 1 if the graphics window should be refreshed and 0 if not. Because of the way the Sioux text window works, this is a lot more reliable than using the Macintosh event manager routines.

`Need2Update` is 1 if the models need to be updated and 0 if not.

<u>Declarations within the header file</u>

```
int UpdateModel(modelptr m);
sptr Name2Spec(char *str);
```

Since `SpectFit.c` contains the main program, an effort was made to not require, or allow, the libraries to get information from the main file. However, some commumication to the libraries is necessary in a couple situations, dealt with in these routines, both of which are used by `BasisFn.c`.

`UpdateModel` goes through the list of equations for the parameters in the basis functions. It calculates their new values using `evalexpr` and updates the parameters as appropriate. It returns 0 if nothing was changed, –1 if a change was made and no errors were encountered, 17 if an equation returned a value of either `Inf` or `NaN`, or

18 if an equation returned a result other than a number (such as a syntax error). If an error is encountered, all equations are still evaluated. The return value of `UpdateModel` may be ignored, since the model is still updated as best possible.

`Name2Spec` returns a pointer to the spectrum for a given name, where the spectrum is from the `Spectra` set. If the name isn't recognized, the funtion return `NULL`.

Graphical routines

```
void drawplot();
void mouse();
int tweak(modelptr m);
```

These are the only routines of the program that are Macintosh specific.

`drawplot` redraws the entire contents of the output window, using `Plotlist`. In general, the way to add things to the output is add them to `Plotlist` and then set an update event. This is how virtually all routines work. The output is rescaled by calling the Plot.c function `SetScales` and then setting an update event.

`mouse` just waits until the mouse is clicked in the output window, and displays that position to the standard error (usually the text window). A key press ends this routine.

`tweak` allows the user to manually adjust parameters in a model until a reasonable starting fit is achieved. As the model is tweaked, it is continually displayed along with the current value of the parameter being tweaked. This routine draws directly to the screen, rather than setting update events as other graphics commands do. The active keys are the four arrows, the '-' sign, and the escape key. Left and right arrows scroll forward and backward through the model parameters, displaying them on the standard text window, and wrapping around at the ends of the list. Up and down arrows increase or decrease the parameter being adjusted. As they increase or decrease multiplicatively rather than additively, step sizes are generally reasonable and it is possible to tweak both very large and very small parameters. Repeated use of a single direction increases the step size, while alternating between up and down arrows reduces the step size. If a parameter is zero initially, an up or down arrow changes it to ±0.001. The '-' key reverses the sign of a parameter. Escape leaves tweaking mode. It might return an error code.

Set management routines

```
int UpdateModels();
void ClearSpect(sptr s,int fre);
void ClearBasis(basisptr b);
void ClearModel(modelptr m,int fre);
void ClearString(char *name);
int CreateSpect(sptr s);
int CreateModel(modelptr m);
int CreateNumber(char *name,float f);
int CreateString(char *name,char *str);
void ResultFree(struct result ans);
int CheckArgs(struct result *ag,int arg,char *str);
```

These are a collection of low level routines for taking care of the globally defined sets. In particular, if something needs to be cleared or replaced, every pointer to the object needs to be updated. While the clearing routines remove all pointers to the objects, there may be text references that are not cleared, such as equations in basis functions. These will return errors when they are executed.

`UpdateModels` updates all the models in the `Models` set. It returns the first error code it finds, or 0 if there aren't any errors. It calls `UpdateModel` for each one.

`ClearSpect` removes all references to a spectrum from the `Spectra` set, the `Plotlist`, and in `Models`. A model pointing to the spectrum is not cleared, but the pointer is set to `NULL`. However, a basis function that relies on it is cleared. If `fre` is 1 the spectrum is freed, otherwise it isn't.

`ClearBasis` frees a basis function and removes it from its model and the `Plotlist` if appropriate.

`ClearModel` removes a model from the `Models` set and the `Plotlist` if appropriate. If the current model is cleared, then the current model is set to the first model in the `models` list. If `fre` is 1 the model is freed, otherwise it isn't.

`ClearString` removes a string from the set `Strings`, if it was in there and frees it.

`CreateSpect` takes in a spectrum and looks in the `Spectra` set for a previous one with the same name. If a previous one is found, all references to the old one are changed to point to the new one, and the old one is then cleared. The color of the new one is set to the old color. In any case, the new spectrum is added to `Spectra`; this routine should be called any time a new spectrum is declared. An error message may be returned, possibly for lack of memory, but more likely because the new spectrum is incompatible with some of the old references. For example, a model is only allowed to point to real spectra. If a situation like this occurs, the old reference is corrected as appropriate, such as by setting it to `NULL`. Thus, any error besides 1 (out of memory) implies that as much was done as possible; an error of 1 means that nothing was done.

`CreateModel` adds a model to the set `Models`, and, if one was previously declared with the same name, then the old one is replaced and cleared. Pointers to the previous model are moved to the new model. It sets the current model to point to the one just declared. An error implies that nothing was done.

`CreateNumber` adds a number to the set `Numbers`, and, if one was previously declared with the same name, its memory space is used instead. In any case, the string input for the name is not used directly and so may be freed. The function returns 1 if memory could not be allocated.

`CreateString` adds a string to the set Strings, and. if one was previously declared with the same name, its memory space is used if it is big enough. Neither string input is used directly and so both may be freed. It returns 1 if memory could not be allocated.

`ResultFree` frees a `result` structure. If the result was temporary (`temp=1`), then it also frees any value included. It does not look for a value in any sets before freeing it, because, by definition, a temporary result is one which is not in any long term structure. The `valptr` member is ignored (but is supposed to be `NULL`).

`CheckArgs` takes in a list of arguments for a routine, passed in as already evaluated results, and compares their types against those in a string, to see if they match. If they match, the routine returns 0; otherwise it returns an error code to indicate how they did not match. Neither the string nor the arguments are changed. Codes for the string:

string code	argument
`S,F,W,C,B,M`	required, same type
`X`	required, any type besides ‘ ’ or ‘?’
`s,f,w,c,b,m`	optional, if exists needs to be specified uppercase type
`x`	optional, if exists any type besides ‘?’
`.`	any further arguments are optional and of any type besides ‘?’

It is always an error if an argument has an error associated with it or if it specifies a data type but has no value. The list of arguments is assumed to be good up to `arg`, but is not looked at beyond there. Error codes result from too many arguments, too few arguments, empty arguments, argument errors, or arguments that don’t match the string character.

Parsing routines

```
struct result dosymbol(char *expr,int i);
struct result dodots(char *expr);
struct result doword(char *cmd,char *expr);
struct result evalexpr(char *expr);
struct result doassign(char *lhs,char *rhs);
int docommand(char *cmd);
int doline(char *cmd);
```

This is a collection of big routines to pull apart strings into meaningful results. For the most part, the routines are quite straightforward, and only execute a couple dozen lines of code, or less, each time they are run. However, they have long lists of `if…else if…` structures, for each of the input possibilities.

`dosymbol` is a parsing procedure which takes in a string containing an arithmetic symbol at location `at`, evaluates the two halves using `evalexpr`, combines them appropriately, and returns the result. Any input string is permitted, and is unchanged during the procedure. The first line in the following table shows the output type for the input types and symbols listed below; all other permutations result in a ‘?’ type and an error code.

F	F	S	S	S	S	S	S	S	S	S	S	S	S	C
+F	F+F	+S	F+S	S+S	+B	+M	B+S	M+S	B+B	M+M	M+B	B+F	M+F	C+C
-F	F-F	-S	F-S	S-S	-B	-M	B-S	M-S	B-B	M-M	M-B	B-F	M-F	
	F*F		F*S	S*S			B*S	M*S	B*B	M*M	M*B	B*F	M*F	

```
F/F       F/S  S/S            B/S  M/S  B/B  M/M  M/B  B/F  M/F
F^F                                                    B^F  M^F
          S+F                 S+B  S+M            B+M  F+B  F+M
          S-F                 S-B  S-M            B-M  F-B  F-M
          S*F                 S*B  S*M            B*M  F*B  F*M
          S/F                 S/B  S/M            B/M  F/B  F/M
          S^F
```

Default result	Output result
`type=?`	if successful, changed to `F`, `S`, or `C`
`val=NULL`	if successful, value of result
`valptr=NULL`	
`temp=1`	
`var=0`	
`er=0`	if unsuccessful, equals error

The two sides of the symbol are evaluated from left to right, but I think this is never relevant. As an example, `dosymbol` could be called with the string `"(abs+0.2)/abs.ymax"`, and `at`=9, pointing to the '`/`'. First, the `(abs+0.2)` and the `abs.ymax` parts are evaluated with `evalexpr`, getting back, respectively, a temporary spectrum and the address of a float. In the first case, `evalexpr` calls `dosymbol` recursively, but that's unimportant. The spectrum is divided by the float and another temporary spectrum, the result, is returned.

`dodots` is a parsing routine that takes in an expression in which the last dot is at position `at`, interprets it and returns the result. This allows a user to directly access the members of a structure. Any input string is permitted and is unchanged by the procedure. Supported structure types:

```
scale.W
S.W.F
S.W          S.F
B.param.W    param.W
B.W          B.param
M.B          M.W    model.B      model.W
```

Default result	Output result
`type=?`	if successful, changed to `F,S,C,B,M`
`val=NULL`	if successful, value of result (which may be `NULL`)
`valptr=NULL`	if successful, pointer to value if result is changeable
`temp=0`	typically 0 but may be 1
`var=0`	
`er=0`	if unsuccessful, equals error

The text on the left of the dot, the root, generally evaluates to a structured variable. The text on the right is the member name (which is sometimes a basis function name or a number), leading to the result. Of course, the root may be a complicated expression, but that doesn't matter. A more difficult situation is encountered for the following situation: "constant:0.offset.min", because the normal root in this case evaluates to a number, and not a structured variable. The solution is to call "constant:0" the root and the remainder is a pair of members. A better solution from the programming point of view is to not allow this type of structure, but that would be less useful for the user. The full list of what this routine recognizes is given in the reference section above.

doword processes function calls, such as "shiftx(abs,2)". The function name is sent in as *cmd and the text inside the parentheses, with commas still included but parentheses removed, is sent in the *expr field. In the routine, the parameter list is separated into a list of arguments, each of which is evaluated with evalexpr before use.

Default result	Output result
type=S	if successful, changed to F,S,C,B,M; ? if unsuccessful
val=NULL	if successful, value of result (which may be NULL)
valptr=NULL	
temp=1	
var=0	
er=0	if unsuccessful, equals error

All arguments given are checked before use with the CheckArgs function, although often the latter arguments are optional, in which case default values are used as needed. Most function names require no more than a call to SpectMath, BasisMath, or ModelMath, with essentially the same function name. However, some require a bit more manipulation. In particular, the function "spec", which converts pretty much anything to a spectrum with definable endpoints and spacing, is fairly long. This routine recognizes quite a lot of functions, which are all listed in the reference section above.

evalexpr takes in a string expression that is to be evaluated and outputs the result. This routine checks syntax and evaluates expressions with the proper order of operations. It calls, as necessary, dosymbol, dodots, and doword, to take care of most of the actual work. Things it searches for, in order, and what it does with them:

""	return a ' ' type
" x"	remove space and try again
"x "	remove space and try again
number	return a 'F' type
=	doassign
+	dosymbol
-	dosymbol
*	dosymbol
/	dosymbol
^	dosymbol
.	dodots
(x)	remove parentheses and try again
"x"	remove quotes, copy to a new string and return a 'C' type

`x(x)`	remove parentheses and send to `doword`
`x(x)x`	error
`x"x"x`	error
variable name	return pointer to variable
param name	return pointer to parameter in current model
word	return copy of word

Default result	Output result
`type=?`	if successful, changed to `F,S,C,B,M`
`val=NULL`	if successful, value of result (which may be `NULL`)
`valptr=NULL`	if successful, either `NULL` or pointer to result
`temp=0`	0 or 1
`var=0`	0 or 1 (if `var` is 1, `val` is defined and `valptr` is `NULL`)
`er=0`	if unsuccessful, equals error

As usual, the expression sent in is unaffected by the routine. Results do not point to the expression sent in. Free results with `ResultFree`, which looks at the `temp` and `type` members of the result structure to see what needs freeing and how to free it. Note that an evaluation of the word "model" says that it's a word, and not a model.

`doassign` makes assignments of all types. `lhs` and `rhs` are the left and right sides of an equals sign and they are evaluated to result types with `evalexpr`. Then, depending on the two types and what they are (temporary value, variable, parameter, etc.), the appropriate assignment is made. Following are the possible assignments:

`model=M`	set `Model` to point to `M`
`W=S`	create new spectrum variable; use right if it is temporary
`W=F`	create new float variable
`W=C`	create new string variable
`W=M`	create new model variable; use right if it is temporary
`S=S`	copy spectrum if left is a variable; set pointer if it is a parameter
`S=null`	if left is a parameter set it to `NULL`
`F=F`	set old number to equal new one; no memory changes
`C=C`	replace string if left is a variable; copy if left is a parameter
`B=B`	replace left basis function with a copy of right, but keep name
`M=M`	copy model if left is a variable; set pointer if its a parameter
`M=null`	if left is a changeable parameter set it to `NULL` (none exist)

The function returns the result of the evaluated right hand side if it was successful. Otherwise, it returns a result with a '`?`' type and the appropriate error code.

`docommand` processes procedural commands, where the whole command string is sent in as `cmd`. It returns –1 if the program is to stop after that operation, 0 for correct operation, or an error code if an error occured. The command is broken into its first word, which is the operation, and the object which is evaluated with `evalexpr`.

Most operations can handle different kinds of objects but not all. Following are the recognized combinations.

`help`	ignored
`exit`	ignored
`log`	ignored
`unlog`	ignored
`/`	ignored
`kill`	`C`
`exec`	`C`
`eval`	`SFWCBM`
`clear`	`SBMFCW`
`scale`	`_SMBFW`
`save`	`SMW`
`plot`	`_WSBM`
`unplot`	`WSBM`
`print`	`_SMBFCW`
`add`	`BSWM`
`remove`	`BS`
`tweak`	`_MW`
`mouse`	ignored
`fit`	`_MW`
`unfit`	`_MW`
`fix`	`BMWF`
`free`	`BMWF`
`update`	`_MW`

When the type is `W`, this means that one of several special words are permitted, such as "model", "all", "spectra", etc. Also, the type '_' is a space, which is properly ' '.

`doline` takes in a line of input from the user and processes it. A line with an '=' in the first word, or at the beginning of the second word is assumed to be an assignment, and is passed to `doassign`. Otherwise, it's assumed to be a command and is passed to `docommand`. It returns –1 if the program is supposed to stop, 0 if everything went well, or an error code if the operation could not be done.

Top level routines

```
void debug();
int main();
```

`debug` is included for debugging purposes. It is probably most usefully set up as a macro file, in which all the lines are `doline()` calls, which create a known problem.

`main` prepares the graphics window, initiallizes the global variables, and then runs a text sort of event loop. The hope was that Macintosh type events, including primarily window update events, would be sent to `doevent` (now removed) and text entry events to `doline`. In practice, the Sioux window (standard input/output exchange), which is the text window, captures all events. As a result, there is no way to respond appropriately to mouse events while using text input from the Sioux window. At the end, which occurs when `doline` returns a –1, `main` frees the global variables and returns to the shell.