The IFEFFIT Tutorial

Matthew Newville
Consortium for Advanced Radiation Sources
University of Chicago, Chicago, IL

Version 1.2.6
July 07, 2004

Contents

License

1 Introduction

IFEFFIT is an interactive program for XAFS data analysis. The main program runs like a command-line 'shell', in which you enter commands to process and manipulate data. IFEFFIT has a high-level command language, allowing you to do the complex data manipulation needed for XAFS analysis (such as background subtraction and Fourier transforms) with simple commands.

One of the principle features of IFEFFIT is that it's command-line functionality can be run in either interactively, from files of commands (i.e., batch files), or accessed from within other programming and high-level scripting languages like Tcl, Perl, and Python. While the details of how this is done are beyond the scope of this tutorial, most people actually use IFEFFIT through one of the GUI programs written on-top of the basic IFEFFIT engine, such as ATHENA, ARTEMIS, or SIXPACK. In this sense, IFEFFIT is not a single program, but a family of related programs and libraries using a common underlying engine.

This tutorial gives a brief description of the command structure and syntax of the IFEFFIT engine, and an overview of the main IFEFFIT command-line program. Other documentation is available at the IFEFFIT web site: http://cars9.uchicago.edu/ifeffit/

1.1 Running IFEFFIT

Once IFEFFIT has been installed on your computer, typing `ifeffit` at the system command prompt (or double-clicking on the appropriate icon) will start the basic IFEFFIT program. You should get a set of messages and a command prompt that looks something like this:

```
Ifeffit 1.2.6  Copyright (c) 2004 Matt Newville, Univ of Chicago
                 command-line shell version 1.1 with GNU Readline
Ifeffit>
```

At this point, you're ready to start typing IFEFFIT commands at the prompt. Try typing

```
Ifeffit> print '1 + 1  = '  1+1
```

If the result makes sense to you, you're ready to continue. Now that you've started IFEFFIT successfully, the rest of this tutorial describes *what* to type. To exit IFEFFIT, you can type `quit`.

The main IFEFFIT program has a friendly shell environment and allows a simple subset of system-level commands to be executed from within the command-line program. On Unix systems, the commands `ls` and `more` will give a directory listing and show the contents of a file, respectively. On Windows, the command `dir` takes the place of `ls`. Typing `help` will give a brief list of the most common commands, while `help <command_name>` will give a little more of information on the nature and use of the selected command. The command *history buffer* is accessible through the up- and down-arrow keys, so that you can scroll through and edit previously executed commands. Further information about the command-line program on Unix systems is given in section 10.

2 Data, Commands, and Simple Data Manipulation

IFEFFIT has a simple view of data, and gives you access this data through high-level commands. There are three types of data that IFEFFIT distinguishes: *scalars* contain a single floating point number, *arrays* contains a list or vector of floating point numbers, and *strings* contains a set of text characters. If you've done any programming, these data types should be familiar to you. In keeping with the language of computer programming, I'll refer to the data in IFEFFIT as *Variables*. These are not necessarily the quantities varied in a fit you might do with your data: I'll call those *Fitting Variables* when there's room for confusion.

2.1 Data Types and Naming Conventions

Variables in IFEFFIT are named, and you can create, name, and manipulate your own data variables. This makes IFEFFIT a fairly general purpose calculator and data plotter. That is, you can type something like this at the IFEFFIT command line:

```
Ifeffit> a =  5
Ifeffit> phi =  (sqrt(a) + 1 ) /2
Ifeffit> print a, phi
   5.00000000     1.61803399
Ifeffit> print sin(10)
 -0.544021111
```

Well, that only shows how to use scalars, not arrays or strings, but it does show how IFEFFIT allows you to do simple calculations using syntax similar to most procedural programming languages.

IFEFFIT distinguishes its three data types *by name*. This allows both you and IFEFFIT to know exactly what kind of data each variable holds. Here are the naming rules for the variables:

Scalars must have names that begin with a letter, ampersand '&', or underscore '_', and then contain letters, numbers, ampersands, and underscores after that. The names are not sensitive to case: `A` is the same as `a`.

Arrays have names that always have exactly one dot ('.'). This gives array names a prefix and suffix. The prefix of the array name is associated with the array *Group*, which is a simple and effective way to make several arrays related to each other. The naming rules for the prefix or group name are exactly the same as for scalars. The suffix of the array name is associated with the array contents. The naming rules for the suffix are similar to those for scalars, but relaxed to allow it to begin with numbers as well as letters, ampersand '& or underscore '_'.

Thus, 'data.energy' and 'data.xmu' are array names, and are said to be in the group 'data'. When we get to discussing commands for doing background removal, Fourier transforms, and the like, we'll see that arrays created by IFEFFIT commands will use the same group name as the input data, which make it easy to keep a group of data together. Other valid array names are 'cu.1' and '_XX_.001'.

Strings have names that always begin with a dollar sign '$', and contain letters, numbers, ampersands, and underscores after that. You can define a string like this:

```
Ifeffit> $string = Gosh, this is easy!
Ifeffit> print $string
  Gosh, this is easy!
```

In fact, '$1' and '$99" are valid string names, but their use is discouraged: '$1' ... '$9' may be internally overwritten whenever you invoke a macro, so it's not a good idea to rely on their values.

There's one more thing to note on names for variables. By convention, "system variables" begin with an ampersand '&' (or '$&' for system strings). This convention is not enforced in any way, but IFEFFIT tends to use such system variables for things that effect its behavior (such as how output is written to the screen). Unusual behavior may result if you write over system variables without knowing what you're doing.

2.2 Data Manipulation

As shown in the brief example at the beginning of the previous section, working with scalar variables in IFEFFIT is easy. You can create and use variables with normal algebraic syntax:

```
Ifeffit> a   =  5
Ifeffit> phi =  (sqrt(a) + 1 ) /2
Ifeffit> x   = pi / phi
Ifeffit> y   = cos(7*x)
Ifeffit> print a, phi, x, y
 5.00000000    1.61803399    1.94161104    0.519178666
```

All scalars are floating point numbers (16 bits of precision). The usual mathematical operators (sin, tan, log, exp, coth, and so on) are supported, and a few operators not often found are supported as well[1]. The variables you define can be used anywhere in the mathematical expressions and assignment statements for other variables.

Manipulating arrays is just as easy as manipulating scalars, though creating arrays of data is a bit more work. One very common way to create arrays is to read them in from data files – that's covered in the next section3. You can also create arrays from scratch using the built-in functions `indarr()`, `ones()`, and `zeros()`. For example

```
Ifeffit> test.index = indarr(10)
```

will create an array with elements (1,2,3,...,10). You can make other evenly spaced arrays:

```
Ifeffit> npts = 100
Ifeffit> step = 0.01
Ifeffit> test.index = step * indarr(npts)
```

which will create an array with elements (0.01,0.02,...,1.0). (Starting with version 1.0053 you can also say

```
Ifeffit> test.ones = range(0.01,1.0, 0.01)
```

to get the same effect. The range function takes "start, stop, step" as it's arguments). In addition, you can create arrays using

```
Ifeffit> data.ones = ones(10)
Ifeffit> data.null = zeros(1000)
```

which will create first an array with 10 elements, all set to 1: (1,1,1,...,1) and then an array of 1000 zeros.

Once you have arrays created or read in from data files, manipulating them is easy:

```
Ifeffit> test.index = indarr(100)
Ifeffit> test.sqrt  = sqrt( test.index / 10)
```

will fill `test.sqrt` with square-roots of the numbers (0.1, 0.2, ... 10.0). Note that the assignment of `test.sqrt` is automatically done *element-by-element*, without looping over elements needed. In fact, IFEFFIT doesn't even allow looping over the elements of an array.

[1] For a complete list of operators, consult the Reference Guide

2.3 Commands and their conventions

The operations you type at the IFEFFIT command line are interpreted as commands. In general, IFEFFIT commands consist of a name, followed by a set of arguments, usually with a *keyword/value* syntax:

```
Ifeffit> command(key= value, key= value, key= value, ...)
```

The parentheses are optional, but if an opening parenthesis is used just after the command name, the closing one is required even if that means the command has to extend over multiple lines. We'll see that many commands will become quite long, so this ability will become convenient. Though they are optional, I'll use the parentheses when talking about commands, so you can tell I mean a command when I say `print()`. The keywords are usually short descriptive names describing some parameter the command may need. The value is often a number, but can often be a variable, string, or even a mathematical expression – a typical command would look like this:

```
Ifeffit> spline(energy=data.e, xmu =data.xmu, rkbg=1, kweight="2")
```

The keyword/value syntax is not universal, and some commands (like the `print()` command shown earlier) take simple lists of arguments separated by commas. Some commands even mix lists and keyword/value pairs:

```
Ifeffit> command(argument1, argument2, argument2)
Ifeffit> command(argument1, argument2, key= value,
                 key= value, key=value)
```

This may seem a little surprising, especially since in the previous section we just used

```
Ifeffit> a = 2
```

which appears to have no command at all! The truth is that IFEFFIT always expects a command to be the first word typed at the command line, but if the first word is not a known command, it uses the default command `def()` – short for define, not default – to define a variable. That is, the above definition was translated to

```
Ifeffit> def( a = 2 )
```

As we'll see in the next section, this can have some profound consequences, so it is often useful to keep in mind that the `def()` command is the default. I'll continue to use the simpler "`a = 2`" syntax throughout this tutorial, and expect that you will too.

2.4 Storing Definitions of Scalars and Arrays: `show()` and `def()`

Because IFEFFIT is primarily an XAFS modeling program, it is important to be able to set up both simple and complex models for XAFS path parameters that can be adjusted during a fit. To allow a flexible modeling environment, a principle feature of IFEFFIT is to allow you to define Program Variables *by formula* and have the values automatically updated when the values of variables in the formula change.

An example will probably help. Let's consider the case of entering this seemingly innocent set of assignments (which, we now know, will use the `def()` command):

```
Ifeffit> a = 1
Ifeffit> b = a + 1
Ifeffit> a = 2
```

What value should b have: 2 or 3? In most computer languages and programs, b is 2, because the formula for it has not been stored, only the value at the time of its assignment. For IFEFFIT the value will be 3 – the formula is stored, not just the value.

The main advantage for this is that you could tell IFEFFIT that a is a fitting variable (by saying guess a = 1, and use both a and b for parameters in the fitting model. No matter what value the fitting engine decides a should have, b will always obey the formula you specified. When we get to fitting XAFS and non-XAFS data, you'll find (at least, eventually) this somewhat unique behavior to be very useful.

Sometimes, however. you really want b to stay as 2, not be dependent on the future value of a. That is, you sometimes want to turn off the 'store the formula' aspect of IFEFFIT. To do this, all you need to do is use the set() command as an alternative to def():

```
Ifeffit> a = 1
Ifeffit> set(b = a + 1)
Ifeffit> a = 2
```

Now the formula for b will not be saved, and it will remain 2 no matter how a changes. Again, it is often useful to remember:

The default command is def(), which will save a variables definition.

2.5 The show() command

At some point you're going to want to get information back from IFEFFIT such 'what exactly *is* the value of e0, anyway?' and 'what are the names of all the arrays I have?'. There are two main commands for showing such information about program variables, and there's no better place to introduce them then right now. The first command is show(), which will show information about Program Variables and other IFEFFIT objects like macros and paths (which we haven't gotten to yet, but which you'll find useful soon). To follow the example of the previous section, we can see the value and definitions of the scalars a and b like this:

```
Ifeffit> a = 2
Ifeffit> b = a + 1
Ifeffit> show a
 a              =       2.000000000
Ifeffit> show b
 b              =       3.000000000 := a+1
```

Note that not only the values are shown, but also the definitions, where appropriate. Although a wasn't actually set(), it was defined as an obviously constant value that didn't depend on any program variables, so FEFFIT knew to treat it as a set value. The show() command takes a list of things (scalars, strings, arrays, etc) to show. We could have said something like this:

```
Ifeffit> a = 2,  b = a + 1
Ifeffit> $doc_string = "here is a simple definition"
Ifeffit> show $doc_string, a, b
 $doc_string    = here is a simple definition
 a              =       2.000000000
 b              =       3.000000000 := a+1
```

For arrays, show() doesn't show all the data points, but a one-line summary of the data:

```
Ifeffit> test.ones = ones(10)
Ifeffit> test.index = 0.1 * indarr(100)
Ifeffit> show test.ones, test.index
 test.ones      =   10 pts [ 1.000 : 1.000 ] := ones(10)
 test.index     =  100 pts [0.1000 : 10.00 ] := 0.1*indarr(100)
```

which shows `test.index` to have 100 point, with a minimum value of 0.1 and a maximum value of 10, for example. As for defined scalars, the definition is shown after the ":=" characters.

It's often necessary to show *all* the scalars, arrays, or string variables. The `show()` has several modifiers to tell it to show entire classes of program variables. All the modifiers begin with the `@` symbol, so to see the values and definitions of arrays, you'd say `show @arrays`. To see all the scalars and strings, you'd say `show @scalars` and `show @strings`.

If you try `show @scalars` or `show @strings`, you'll notice several scalars and strings that you didn't define, but are loaded in to the program as it starts. These include `pi` and `etok`[2] and several "system variables" that begin with a & (or $& for system string variables). As mentioned at the end of section 2.1, these "system variables" are used internally by IFEFFIT, and though you can change their values, this is not necessarily recommended. For the most part these "system variables" can be ignored during normal use, and won't be discussed into detail in this tutorial.

You may also notice that the order the scalars shown by `show @scalars` is not the same order you put them in. The order may even change over time. This reflects the fact IFEFFIT tries to manage the scalars and definitions for its own internal efficiency, and will attempt to arrange it so that the `set()` values are listed before the `def()` values. Since many commands can change the list of scalars, the order of listing may change at any time. The same behavior applies to arrays: IFEFFIT will rearrange the list of arrays to suit its own needs and to try to list the "constant" arrays before the defined arrays.

Speaking of array data, it's useful to see information about the groups of arrays, as analysis threads are generally done according to group. To show all the arrays in a particular group, you'd say `show @group=data`, which will show all the arrays in the group `data` as if you had said `show data.energy, data.xmu` and so forth. To get a list of the array groups, type `show @groups`.

The `show()` command can take other "@" modifiers for FEFF paths, macros, and fitting variables. These will be discussed when the time comes to discuss `feff` paths, writing macros, and fitting. The `show()` command has a modifier to show a brief description of all the commands: `show @commands` will list of all commands.

2.6 The `print()` command

We used the `print()` command before, but haven't really explained what it's doing. Unlike the `show()` command, which tends to show information about the data *type*, the `print()` command is a more literal command. That is, `print e0` will simply print the values of the scalar `e0`. No definitions will be shown. For array data, the entire array will be printed – hardly ever what you really want, but sometimes it's necessary. At it's simplest, then, the `print()` command prints the values of variables listed:

```
Ifeffit> number = 99.0
Ifeffit> print number
 99.0000000
Ifeffit> print pi, number, $doc_string
 3.14159265 99.0000000 here is a string
```

In addition to this simple behavior, the `print()` command can print literal strings and also evaluate expressions in place. Thus, you can use `print()` to write simple messages:

[2] `etok` is the value of $2m_e/\hbar^2$ in units on $\text{Å}^2/\text{eV}$, and is useful for converting x-ray energy values to photoelectron wavenumber: `set kval = sqrt(etok * (energy-e0))`

```
Ifeffit> print " the square root of ", number, " is ", sqrt(number)
 the square root of    99.0000000  is    9.94987437
Ifeffit> $descrip = " # of seconds per year"
Ifeffit> value    = 365*24*60*60
Ifeffit> print  $descrip, " = ", value
 # of seconds per year =    31536000.0
```

In addition to the `print()` command, there's also an `echo()` command that simply prints a string:

```
Ifeffit> echo "Hi Mom!"
 Hi Mom!
```

This is not incredibly useful when typing at the command line, but does become useful when you load files of IFEFFIT commands, as we'll see next.

2.7 Command Files

Typing at the command line is all well and good until you have to do it the third or fourth time, at which point it becomes pretty tedious. More importantly, it's not convenient for processing lots of data. For that, you'd like to be able to edit text files of IFEFFIT commands and run them all at once. You can. A file of commands can be loaded with the `load()` command, which will run through all the commands in the file. A command file *show_bkg.iff* that looks like this

```
# File  show_bkg.iff
read_data(file=Cu.dat, type=raw, group= cu)
cu.energy  = cu.1 * 1000.0
cu.xmu     = ln(cu.2 / cu.3)
spline(energy = cu.energy, xmu = cu.xmu,
       rbkg=1.1, kweight=1., kmin=0)

plot(cu.energy, cu.xmu)
plot(cu.energy, cu.bkg, xmin=8850, xmax=9300,
               color=red)
#
```

can be loaded as

```
Ifeffit> load show_bkg.iff
```

More than that, the history mechanism of the `ifeffit` command-line program saves a list of the 500 most recent IFEFFIT commands run to the file *.ifeffit_hist* in your home directory. This file can be used as a starting point for creating and editing command files. These topics will be discussed further in section 8

3 Reading Arrays from Data Files

IFEFFIT reads ASCII files with data listed in columns, delimited by whitespace (blanks or tabs). The data are stored in IFEFFIT arrays, so that they are ready for immediate manipulation, plotting, and analysis. The `read_data()` command is used to read data arrays from an ASCII file. A typical use would look like

```
Ifeffit> read_data(file = cu_001.xmu, group= 'cu', type = 'xmu')
```

Because of the naming rules for arrays (see 2.1), `read_data()` needs to assign both a prefix (or group name) and suffix for each array read in from the data file. Since data in a file is usually grouped together logically, `read_data()` will use just one group name for all the arrays in an individual file. The group name used can be specified with the `group` keyword in the `read_data()` command. By default, the prefix of the filename itself is used.

The method for assigning the *suffixes* of the array names is a bit more involved. There are four different ways for IFEFFIT to determine the array suffixes when reading in a file:

From the 'type' argument : In the example above, the `type = 'xmu'` argument tells IFEFFIT to use the suffixes `energy` and `xmu` for the first and second columns, and use `3`, `4`, ... for any remaining columns. Another commonly used *file types* is `chi` for columns of `k` and `chi`. A more complete list of known file types and the suffixes they produce is given in the Reference Guide.

From the 'label' argument : For data that is not in one of the pre-defined types (or if you just want to specify the array suffixes explicitly), then the `label` argument can be used. `label` takes a string that is just the array suffixes listed separated by a space. Using `read_data(file = cu_001.xmu, group='cu', label = 'energy xmu')` would be equivalent to the above `type` version.

From the files own 'label' line : Many files (especially, those written by IFEFFIT, FEFFIT, or AUTOBK) will have a *label line* which contains the column labels (i.e, array suffixes) and appears just before the data and just after a line of minus signs (`#----------`):

```
# Cu foil at 10K
# OFFSETS     51284   50016   48319
#---------------------------------
#     energy          xmu
 .8786204E+04    .1013661E+01
```

If neither the `type` nor `label` keyword are specified, IFEFFIT will look for a label line and use it.

By column index : If none of the above methods are used (that is neither the `type` nor `label` keyword are given, and a label line is not found), IFEFFIT will name the arrays by column number `1`, `2`, `3`, ... Though primitive, this is actually the most predictable behavior, and can be enforced by using "`type = 'raw'`".

4 Plotting Data

This section assumes that IFEFFIT has been built with the PGPLOT plotting package. This is available for both Unix and Win32 systems, though support for Win32 is still experimental. For Unix systems, this library has to be installed prior to installing IFEFFIT, which is fairly easy to do for most systems. IFEFFIT can also be built without this plotting package.

Plotting in IFEFFIT is encapsulated in a handful of commands, the most important of which is the `plot()` command, which takes two required arguments for the ordinate (x-array) and abscissa (y-array) to be plotted, and take a large set of optional arguments. A simple plot can be done like this:

```
Ifeffit> plot(cu.energy, cu.xmu)
```

The `plot` command will overplot , so that a second `plot` command:

```
Ifeffit> plot(cu.energy, cu.bkg)
```

will add a trace of the background to the earlier plot. To force the current plot to be erase before plotting, you'd say

```
Ifeffit> newplot(cu.energy, cu.xmu)
```

Each x-y trace plotted has a *color* and *line style* associated with it. You can set the values in these table explicitly for each particular plot command by specifying the color or linestyle (or both) directly:

```
Ifeffit> plot(cu.energy, cu.xmu, color = blue)
Ifeffit> plot(cu.energy, cu.bkg, color = red, style = dashed)
```

You can also pre-define the color and linestyle for the first, second, ... trace ahead of time

```
Ifeffit> color(1=blue, 2 = red, 3 = black)
Ifeffit> linestyle(1=solid, 2=dashed, 3 = linespoints2)
```

so that the first trace plotted is a solid blue line, and the second is a dashed red line, and the third a black line with a '+' at each data point.

The allowed color names are the 'standard X Windows' colors found in the *rgb.txt* file on your system. Most common color names are supported, as well as the descriptive if ambiguous 'X Windows' names like "lightsalmon3" and "bisque". You may also use the conventional hexadecimal representation of the color with a string of '#RRGGBB'.

Allowed line styles are 'solid', 'dashed', 'dotted', 'points', and 'linespointsN' where N = 1, 2, 3, ... The latter will draw a line connecting symbols at each data point, with symbols '.', '+', '*','o', 'X', squares, and triangles for N=1,2,3,4,5, and 6. Setting the color and linestyle tables like this is often a convenient thing to do in a macro (see section 9) or start-up file.

For Unix using X Windows, the PGPLOT window supports getting the (x,y) coordinates from the plot window using the mouse device. To use this within IFEFFIT, you'd type `cursor` at the command prompt, and then click on the desired point on the plotting window. The IFEFFIT variables `cursor_x` and `cursor_y` will contain the (x,y) coordinates. The `zoom` command will let you view a selected region of the plot by clicking the mouse on the corners of the area to zoom in on. Many more plotting options exist – please consult the Reference Guide.

5 XAFS Data Processing

IFEFFIT's main job is to help you analyze XAFS data, and IFEFFIT tries to make it easy to do simple XAFS analysis tasks. These simple tasks include such things as converting beamline data in $\mu(E)$, doing pre-edge subtraction, determining E_0, fitting a post-edge (spline) subtraction to determine $\mu_0(E)$ and $\chi(k)$, and doing XAFS Fourier transforms to look at data in R-space. These standard tasks of XAFS data manipulation are described in this section.

5.1 Data Reduction

The general algebraic manipulation of array data within IFEFFIT makes the conversion of raw beamline data to XAFS $\mu(E)$ very easy, and also allows the averaging and re-scaling (if necessary) of data. For example, reading raw beamline transmission and fluorescence data and converting this to $\mu(E)$ might look like this:

```
Ifeffit> read_data(file = cu_expt.dat, group= 'cu',
                   label = 'energy i0 i1 if')
Ifeffit> set cu.xmu_t = log(cu.i1 / cu.i0)
Ifeffit> set cu.xmu_f = (cu.if / cu.i0)
```

If you've collected data in fluorescence using a multi-element-detector, you may need to sum several arrays before dividing by I_0, something like this will do the trick:

```
Ifeffit> read_data(file= med_fluor.dat, group='med', type='raw')
Ifeffit> set med.if= (med.4  + med.5 + med.6 + med.7 + med.8 +
                      med.9  +med.10 +med.11 +med.12 +med.13 )
Ifeffit> set med.xmu= (med.if / med.2)
```

5.2 Pre-Edge Subtraction, E0 determination, and Normalization

Pre-edge subtraction removes the baseline from the EXAFS $\mu(E)$, determines the edge energy (used as the origin of *k*), and normalizes the above-edge $\mu(E)$ to 1. All of this is done by the `pre_edge()` command which takes arrays of energy and absorption $\mu(E)$:

```
Ifeffit> pre_edge(cu.energy, cu.xmu)
```

Like most IFEFFIT command, this simple-looking command actually causes a fair amount of data processing behind the scenes. It also creates several program variables, especially arrays and scalars detailing the pre-edge subtraction. Here's a list of the most important program variables that `pre_edge()` sets:

Variable	Description
`e0`	Energy Origin (found near maximum derivative of $\mu(E)$
`edge_step`	Edge Step / normalization constant
`pre_slope`	slope of pre-edge line
`pre_offset`	offset of pre-edge line
`$group.pre`	array of pre-edge subtracted $\mu(E)$
`$group.norm`	array of pre-edge subtracted and normalized $\mu(E)$

where `$group` is the 'group name' taken from the input $\mu(E)$ array - 'cu' in this case. Using `pre_edge()` on another array would generate new arrays of pre-edge subtracted and normalized $\mu(E)$ for the corresponding group. Note, however, that it will overwrite the scalar values from the earlier invocation of `pre_edge()`.

There are several optional arguments to `pre_edge()` not mentioned here. These arguments can be use to set the ranges over which to fit the pre-edge line and post-edge curve, whether or not to perform every part of the calculation, and so forth. These optional arguments would normally be used like this:

```
Ifeffit> pre_edge(cu.energy, cu.xmu, e0=8980.5)
```

which would force the value of E_0 to be 8980.5 and prevent `pre_edge()` from trying to determine E_0 by itself. As for all commands, the complete set of the optional parameters for `pre_edge()` are given in the Reference Guide.

5.3 Post-Edge Background Subtraction

Post-edge background subtraction involves drawing a "smooth background" ($\mu_0(E)$ through the oscillatory part of the XAFS and extracting $\chi(k)$ using this and the formula

$$\chi(E) = \frac{\mu(E) - \mu_0(E)}{\Delta\mu(E_0)}$$

where $\mu_0(E)$ is "the smooth background" of $\mu(E)$ and $\Delta\mu(E_0)$ is the edge-jump. Determining the XAFS background function $\mu_0(E)$ generally receives quite a bit of attention in the XAFS community. IFEFFIT uses the AUTOBK algorithm, which simply asserts and then implements a rather common-sense approach to background subtraction: only the low-frequency components of $\mu(E)$ should make up $\mu_0(E)$.

The implementation of AUTOBK in IFEFFIT is encompassed in the `spline()` command, which takes arrays of energy and μ, and several optional parameters, and writes out $\chi(k)$, $\mu_0(E)$, and several scalars. A basic use of `spline()` would look like this

```
Ifeffit> spline(cu.energy, cu.xmu, rbkg=1.0)
```

Like `pre_edge()`, this simple-looking command does quite a bit of data processing behind the scenes, and creates or writes several variables with IFEFFIT. For one thing, `spline()` will execute `pre_edge()` unless it's obvious that it shouldn't[3]. That means that all the output parameters of `pre_edge()` will be created (or overwritten) by `spline()`, and E_0 and the edge jump $\Delta\mu(E_0)$ will be determined if needed.

To say much more about the `spline()` command, I'd have to discuss the details of the AUTOBK algorithm. I'll try to be brief: `spline()` chooses a smooth background spline such that the low-R portion of the resulting EXAFS χ are minimized. That means that `spline()` needs to do a Fourier transform of the $\chi(k)$ it generates. Because of this, the `spline()` function takes many command arguments that resemble Fourier transform parameters, so that in addition to the arguments of `pre_edge()` the important arguments that `spline()` takes are:

Variable	Description
`rbkg`	$R_{\rm bkg}$, the highest R value to consider background
`kmin`	$k_{\rm min}$, the starting k for the Fourier transform
`kweight`	w, the k-weight factor for the Fourier transform.

Please note that the Fourier transforms appropriate for `spline()` are *not* the same as those appropriate for structural analysis. Usually, values of $k_{\rm min} \approx 0$ and $w = 1$ are appropriate for `spline()`.

[3] where "obvious" means that an array named $group.pre already exists. This really is mediocre definition of obvious, especially if you're writing scripts to automate the processing of lots of data. On the other hand, if you're casually strolling through your data at a command-line prompt, it's probably fine.

Like `pre_edge()`, the `spline()` command also creates new arrays, most importantly:

Variable	Description
`$group.bkg`	$\mu_0(E)$, the background function itself
`$group.k`	k, the array of wavenumbers
`$group.chi`	$\chi(k)$, the (unweighted) EXAFS

5.4 Fourier Transforms

Fourier transforms are an integral part of XAFS analysis, and the ability to perform them quickly and easily is very important. IFEFFIT has different commands for forward ($k \rightarrow R$) and reverse forward ($R \rightarrow q$: I'll use q to refer to back-transformed k-space data) Fourier transforms. They're very similar to one another, so I'll first discuss *forward* Fourier transforms with `fftf()` in detail, then *reverse* Fourier transforms with `fftr()` more quickly.

IFEFFIT use a simple Fast Fourier transform (FFT), which places some demands on the data transformed by these commands. In addition, XAFS analysis usually imposes some conventions on Fourier transforms, so that the commands discussed here are not general purpose Fourier transform functions, but are really *XAFS Fourier transforms*. The most obvious difference between 'normal' and 'XAFS' Fourier transforms is that the latter transforms k to R, while a 'normal' FT would transform k to $2R$. Aside from a scale factor, this changes the normalization constants used. In addition, the XAFS Fourier transform (at least as IFEFFIT implements it) allows a variety of smoothing window functions, and a weighting factor. More details can be found in *XAFS Analysis with* IFEFFIT .

5.4.1 Forward Fourier Transforms

The Forward XAFS Fourier transform command is `fftf()`. It's primary input is an array of $\chi(k)$ data. The requirements of the FFT mean that the input $\chi(k)$ data **must** be an array that is on an even k-grid with grid spacing of $k = 0.05$ Å^{-1}, and starting at $k = 0$.

These are stringent requirements. Fortunately, the `spline()` command of section 5.3 writes its output $\chi(k)$ array according to these rules. If you're importing $\chi(k)$ data written from another program, you'll have to make sure the data is moved to this k-grid. There are two ways to this: either interpolate the data yourself (see the interpolation functions in the Reference Guide) or specify the k-array corresponding to your χ data in the `fftf()` command and let it do the interpolation for you.

Properly aligned $\chi(k)$ data can be transformed to R-space using a command like this:

```
Ifeffit> fftf(cu.chi, kmin=2.0, kmax=17.0, dk=1.0, kweight=2)
```

while data not on the expected grid would be Fourier transformed like this:

```
Ifeffit> fftf(cu.chi, k=cu.k,
              kmin=2.0, kmax=17.0, dk=1.0, kweight=2)
```

The keywords `kmin`, `kmax`, and `dk` help define the window function, and `kweight` sets the k-weighting factor. You can also specify the form of the window function. The full list of window types and their functional form is given in the Reference Guide, but the most useful ones are the Hanning window (`kwindow=hanning` – the default window function) which ramps up to one on either end of the k-range as $\cos^2$, and the Kaiser-Bessel window (`kwindow=kaiser`), which is often thought to give superior peak resolution.

Because it is often necessary to do many Fourier transforms with exactly the same parameters, the `fftf()` command, can also read Fourier transform parameters from appropriately

named program variables. This is actually a feature of many commands, but it seems most useful for the Fourier transform commands. It works like this: setting scalars `kmin`, `kmax`, and so forth will have the same effect as setting those arguments to the `fftf()` command.

```
Ifeffit> kmin=2.0, kmax=17.0, dk=1.0, kweight=2
Ifeffit> fftf(cu.chi)
```

would have the result as

```
Ifeffit> fftf(cu.chi, kmin=2.0, kmax=17.0, dk=1.0, kweight=2)
```

In fact, the `fftf()` command will first read the value for the parameter $k_{\rm min}$ from the program variable `kmin` (and so on for the other Fourier transform parameters), and then from the command argument `kmin`, so that the command argument will always override the program variable value. Furthermore, `fftf()` will itself set the value of the program variable `kmin`, possibly overwriting any value you had previously set. Thus

```
Ifeffit> kmin=2.0, kmax=17.0, dk=1.0, kweight=2
Ifeffit> fftf(cu.chi, kmin=3.)
Ifeffit> fftf(other.chi)
```

will use the same Fourier transform parameters (that is $k_{\rm min} = 3\text{Å}^{-1}$) for both transforms.

A Fourier transform inherently deals with complex data, and a minor complication arises from the inconvenience that the measured XAFS $\chi(k)$ is strictly a real function. In `fftf()` then, there is an ambiguity of whether to use the measured XAFS as the real or imaginary part of the complex XAFS function. Since the measured XAFS is typically described as the imaginary part of a complex fine-structure function $\tilde{\chi}$, one might be tempted to say that the data $\chi(k)$ ought to be set to the imaginary part of $\tilde{\chi}$. IFEFFIT usually assumes that the data $\chi(k)$ is the real part of the complex $\tilde{\chi}$ and sets the imaginary part to zero – this is in keeping with the convention of older programs from the University of Washington, but it is purely a matter of convention. You can explicitly specify the behavior by saying `fftf(imag = data.chi,...)` or `fftf(real = data.chi,...)`. If you don't specify, the data will be taken as the real part. This is almost never important – at least not until you want to compare unfiltered $\chi(k)$ with filtered $\chi(k)$. The output arrays generated by the `fftf()` command are

Variable	Description
`$group.win`	The k-space window function used
`$group.r`	R, the array of distances
`$group.chir_mag`	$\lvert\chi(R)\rvert$, the magnitude of $\chi(R)$
`$group.chir_pha`	the phase of $\chi(R)$
`$group.chir_re`	$\mathrm{Re}[\chi(\mathrm{R})]$, the real part of $\chi(R)$
`$group.chir_im`	$\mathrm{Im}[\chi(\mathrm{R})]$, the imaginary part of $\chi(R)$

It is possible to do "phase-corrected" Fourier transforms with IFEFFIT using the theoretical phases from FEFF calculations, but that is beyond the scope of this tutorial.

5.4.2 Reverse Fourier Transforms

`fftr()` is the Reverse XAFS Fourier transform command, mainly used to filter $\chi(R)$ data to backtransformed $\chi(k)$. Following the convention of FEFFIT, backtransformed k-space is called q to avoid confusion with the original k-space data. The `fftr()` command then transforms $\chi(R)$ to $\chi(q)$. As with the forward Fourier transform, the data is requirements of the FFT mean that the input $\chi(k)$ data *must* be given as an array that is evenly spaced in R with a fixed grid spacing of $R = \pi/1024 \approx 0.03068\,\text{Å}$, and starting at $R = 0$. To further complicate matters,

there is ambiguity as to whether to transform just the real part, just the imaginary part of $\chi(R)$, or both. In general, both parts are transformed to ensure that the overall amplitude scale is preserved.

`fftr()` has an almost identical command set to `fftf()`, with `k` replaced by `r` in the parameter names. That is, the FT window parameters are defined with `rmin`, `rmax`, `dr`, and so on. The window functional form is set by `rwindow`. A typical use might look like this

```
Ifeffit> fftr(real=cu.chir_re, imag=cu.chir_im,
              rmin=1.70, rmax=3.0, dk=0.1)
```

The output arrays generated by the `fftf()` command are

Variable	Description
`$group.rwin`	The R-space window function used
`$group.q`	q, the array of wavenumbers
`$group.chiq_mag`	$\|\chi(q)\|$, the magnitude of $\chi(q)$
`$group.chiq_pha`	the phase of $\chi(q)$
`$group.chiq_re`	$\mathrm{Re}[\chi(\mathrm{q})]$, the real part of $\chi(q)$
`$group.chiq_im`	$\mathrm{Im}[\chi(\mathrm{q})]$, the imaginary part of $\chi(q)$

Before ending this section, let me say a few words about "Fourier filtering". IFEFFIT's approach to Fourier transforms is fairly general, and the use of two Fourier transforms to "isolate a shell" is not a trivial process with IFEFFIT. This partly reflects my experience and belief that "Fourier filtering" can *not* be made a trivial process.

Comparing filtered with unfiltered data is a common desire. Given the ambiguities in how to handle the real and imaginary parts of the data, getting the details right for this are not trivial. Though macros won't be discussed until section 9, using a macro to get the details right is a good idea. Such a macro for Fourier filtering might look like this:

```
macro filter group  "kweight=2,kmin=3"   "dr=0"
  fftf(real=$1.chi, $2)
  fftr(real=$1.chir_re, imag=$1.chir_im,  $3)
  set $1.chik   = $1.chi  * $1.k^kweight
  set $1.chik_w = $1.chik * $1.win
  set $1.chiq   = $1.chiq_re / ($1.k^kweight)
end macro
```

With this macro definition (which takes arguments as group name, k parameters, and R parameters), you could perform a filter, and overplot filtered and unfiltered data with

```
filter data "kweight=2,kmin=3,kmax=15,dk=1" "rmin=1.6,rmax=3."
newplot(data.q, data.chiq_re )
plot(   data.k, data.chik_w )
```

Note that the convention used here is that the data $\chi(k)$ is the real part for the Forward transform, so that the real part of $\chi(q)$ is the appropriate choice for comparison. Of course, that should be compared to the k-weighted, windowed $\chi(k)$.

6 XAFS Analysis

XAFS Analysis in IFEFFIT consists of modeling XAFS $\chi(k)$ in terms of a *sum of paths* with the XAFS contribution from each path being modeled by a calculation from FEFF. A Path is an abstract formalism for breaking the EXAFS into manageable pieces based on the scattering path that a photo-electron takes. Many EXAFS analysis methodologies use concepts such as 'shell' or 'coordination sphere'. To some extent the distinction is only conceptual, and therefore unimportant. In other ways, the path formalism is clearly a superior way to robustly analyze complex EXAFS data that includes the effects of multiple scattering. If you're used to thinking of your EXAFS data as having contributions based on shells or coordination spheres, paths aren't that big of a change.

6.1 Defining Paths

IFEFFIT relies on FEFF for basic information about all of its paths. This means that you need to run FEFF and sort through its results (neither of which are trivial tasks) prior to defining paths with IFEFFIT. The outputs of FEFF that IFEFFIT needs is the set of *feffnnnn.dat* files, where each file represents the result for a single path. Some versions of FEFF can write all of the path information into a single file called *feff.bin* as an alternative to a large set of *feffnnnn.dat* files. IFEFFIT can read some versions of *feff.bin*, but this discussion will focus on using the more universal *feffnnnn.dat* files.

Once FEFF has run and a set of *feffnnnn.dat* files exists, using them within IFEFFIT is fairly easy. The `path()` command is used to define paths within IFEFFIT. The information needed to describe an XAFS path is a bit too complicated to conveniently hold as a set of IFEFFIT scalars and arrays – it could be done, but the bookkeeping would be a nightmare. Instead, paths are defined and stored internally, so that you can refer to a path by an integer index. Though a break from the idea in section 2 that all data is stored in program variables that you can access easily and directly, there are commands to convert the path data to program variables.

The `path()` command takes the name of a *feffnnnn.dat* file and a unique IFEFFIT *path index* as its primary arguments. The IFEFFIT path index is an integer (from 1 to 10000). It does not need to be the same index as FEFF used, but that is often a convenient way to do it. In addition, the `path()` command takes several optional arguments, known as *Path Parameters* that represent the parameters used to modify the EXAFS function for that path. The path parameters include such things as ΔR, the change in path distance (well, half-path-length for multiple scattering paths), σ^2, the mean-square displacement of the bond, and an E_0 shift. The most important Path Parameters are

Path Parameter keyword	Description
`file`	Name of *feffnnnn.dat* file
`index`	path index: a unique integer identification
`label`	path label: a string describing the path
`s02`	S_0^2, amplitude reduction factor
`e0`	E_0, energy shift
`delr`	ΔR
`sigma2`	σ^2
`third`	C_3, third cumulant

a full list is given in the Reference Guide. A simple path definition would look like this:

```
Ifeffit> path(index=1,file = feff0001.dat,
              s02 = 0.9, sigma2 = 0.001)
```

This would define path 1, and assign values for S_0^2 and σ^2 for this path. An important aspect of IFEFFIT is that the Path Parameters are *defined* in the same way as other `defined` scalars (see section 2) – the formulas specified for the path parameters in the `path()` command are remembered. This allows you to define the path like this

```
Ifeffit> s02_001  = 0.9
Ifeffit> e0_001   = -1.
Ifeffit> sig2_001 = 0.001
Ifeffit> path(index=1,file = feff0001.dat,
              s02    = s02_001,
              e0     = e0_001,
              sigma2 = sig2_001 )
```

where the parameter `s02` for path 1 now depends on the definition of the scalar `s02_001`. This is a fairly simple example, and the definition for the `s02` parameter for path 1 could be more complex – say, `min(1.2,s02_001)` to set an upper bound on this parameter. In addition, the `s02_001` scalar here could have a more complex definition or even be a fitting variable defined with `guess s02_001 = 0.9`.

Additional paths are defined in the same way:

```
Ifeffit> path(index=2,file = feff0002.dat,
              s02    = s02_001, e0 = e0_001,
              sigma2 = sig2_001 )
```

The only *required* keyword for a path is the integer index (if the keyword comes first in the list of arguments, the `index` can be dropped, so a completely equivalent definition is

```
Ifeffit> path(2, file   = feff0002.dat)
Ifeffit> path(2, s02    = s02_001  )
Ifeffit> path(2, e0     = 2.0      )
Ifeffit> path(2, sigma2 = sig2_001 )
```

subsequent `path()` commands will set or *overwrite* the definitions for other path parameters. This makes it fairly simple to change the definition of a path parameter half way through an analysis, say between fits, where something like

```
Ifeffit> path(2, s02    = 0.85)
```

will change one path parameter and leave the rest as previously defined.

6.2 Combining Paths

Once defined, combining paths to give a $\chi(k)$ function is very easy with the command `ff2chi()`. `ff2chi()` takes its name from the FEFF module that combine FEFF path files into a single $\chi(k)$ function. Because IFEFFIT allows you to alter the paths through the Path Parameters, and use paths calculated from different runs of FEFF, this version of `ff2chi()` is significantly more flexible and powerful than FEFF's own version.

`ff2chi()` takes a list of path indices, and creates arrays for $\chi(k)$. Assuming that the paths have already been defined, a command like this:

```
Ifeffit> ff2chi(1,2,3, group = ff)
```

will add paths 1, 2, and 3 together and create arrays `ff.k` and `ff.chi` containing the sum of these paths. Each of the paths will be altered according to its own Path Parameters in the sum. The list syntax '`1, 2, 3`' could also be replaced by '`1-3`'. Lists of the form `1-3,5,7-10,23,11` are also allowed. An important note is that (currently), paths listed twice are used twice. This is an easy mistake to make, so beware[4].

Besides the list of paths and group name for the output arrays, `ff2chi()` has a few other arguments, most of which you won't really need. I'll just mention a few convenient ones here. The arguments `kmin` and `kmax` can set the k-range of the output arrays, and the argument `do_real` will cause the "real part" of $\chi(k)$ to be written to the array `$group.chi_real` – the confusion over real and imaginary parts of the complex $\chi(k)$ shows up again! The `$group.chi` array that `ff2chi()` always generates is the imaginary part of the complex $\chi(k)$, and should correspond to the experimentally derived $\chi(k)$.

6.3 Getting and Viewing Path Parameters

Once you've combined a few paths with `ff2chi()` or, as we'll see shortly, done a fit with `feffit()`, you may want to find out some information about the path parameters for a set of paths. There are a few ways to get this information. The `show()` command, first discussed in section 2.5, has a "`@path`" modifier to give some information about a defined path.

```
Ifeffit> show @path=1
 PATH   1
   feff  = feffcu01.dat
   id    = Cu metal first neighbor
   reff  =    2.547800, degen  =   12.000000
   s02   =    0.937373, e0     =    0.510790
   dr    =    0.000525, ss2    =    0.003496
   3rd   =    0.000000, 4th    =    0.000000
   ei    =    0.000000, dphase =    0.000000
```

This shows all the scalar values of the path parameters (`reff` is the half-path length, `degen` is the path degeneracy, and so on). The argument to the `@path` modifier to `show()` can be any valid path list as described in section 6.2 (for example: `1,2,4-9,12`). To see all the defined paths `show @paths` will work.

The `show @path` family of commands only prints the values of the path parameters. Another way to get path information is to convert them to program variables with the `get_path()` command. This command only takes one path at a time (not a path list), but allows you to get the path parameter values into program variables for later manipulation. The syntax for `get_path()` is

```
Ifeffit> get_path(path = 1, prefix = path1)
```

or, equivalently and more simply

```
Ifeffit> get_path(1, path1)
```

This will create scalars `path1_s02`, `path1_e0`, `path1_ei`, `path1_delr`, `path1_sigma2`, `path1_third`, `path1_fourth`, `path1_degen`, `path1_reff` and give the values of the path parameters for path 1. In addition, the strings `$path1_file` and `$path1_id` will contain the values of the path FEFF file name and path identification string, respectively. The `prefix` argument to `get_path()` can be any valid variable name. Of course, array variables for a given path can be formed with the `ff2chi()` command, as described in the previous section. Just to be clear, though

[4] it is on the "TODO list" to check for and warn about this possibility, which is hardly ever intended

```
Ifeffit> ff2chi(1, group = path1)
```

will create arrays of `path1.k` and `path1.chi`. Other arrays for the real part, magnitude, and phase of $\chi(k)$ can be obtained from `ff2chi()` by using arguments `do_real`, `do_mag`, and `do_phase`, respectively.

6.4 `feffit()`: Fitting XAFS Data with Paths

At some point in the analysis of XAFS data, you'll want to refine a set of path parameters so that a sum of paths best matches the data. The `feffit()` command is the basic tool for fitting EXAFS data to a set of paths. In some sense, `feffit()` is just a fancy version of `ff2chi()`. Another view might be that the `feffit()` command is the main point of the IFEFFIT library, so that `ff2chi()` is a very simple version of `feffit()`. In any event, the two functions are closely related and purposefully implemented with as much overlap as possible so that once you feel comfortable with the `paths()` and `ff2chi()` commands, `feffit()` should be a fairly small step.

`feffit()` has four basic requirements:

1. $\chi(k)$ data to fit,
2. a list of paths to sum to make the model XAFS.
3. Fourier transform and fit range parameters
4. one or more variables defined that affect the model XAFS.

First, `feffit()` needs some $\chi(k)$ data to fit. As you can probably guess, this data needs to be a named array for $\chi(k)$ just as you would use for the Fourier transform routines (ie, starting at $k = 0$, and progressing in steps of 0.05 Å^{-1}). You specify the $\chi(k)$ array with `feffit(chi= data.chi,...)`. If the data you have is not on the expected grid, you'll either need to interpolate it on to the expected grid or specify the array of k values, with

```
Ifeffit> feffit(chi= data.chi,k=data.k, ...)
```

Second, `feffit()` needs a list of paths for the sum-over-paths that makes up the model XAFS. This sum is essentially the same as for `ff2chi()`, comprising of a simple list of path indices. Path lists of the '`1, 2, 3`', '`1-3`', and '`1-3,5,7-10,23,51`' are accepted, so that the `feffit()` command would now look like (assuming the data is on the proper k grid)

```
Ifeffit> feffit(chi= data.chi,1-3, ...)
```

Third, `feffit()` needs Fourier transform and fit range parameters defined. This reflects the fact that XAFS is a band-limited phenomenon, with a finite k- and R-range. As mentioned in section 5.4.1, the Fourier transform parameters can be read either from the appropriately named program variables (`kmin`, `kmax`, `kweight`, `dk1`, `dk2`, and so forth) or from the command argument list (using these same identifiers as keywords). In addition to the k-space Fourier transform parameters, `feffit()` needs the fit R-range, specified by `rmin` and `rmax` (this reflects the fact that `feffit()` normally does the fit in R-space – fitting in k-space is possible, as well). Like the Fourier transform parameters, the R-range can be set from program variables (`rmin` and `rmax` or in the `feffit()` command statement itself. We could say either

```
Ifeffit> kmin = 3.0, kmax = 15.0, kweight = 2, dk1 = 2, dk2= 2
Ifeffit> rmin = 1.5, rmax =  3.5
Ifeffit> feffit(chi= data.chi, 1-3 )
```

or

```
Ifeffit> feffit(chi= data.chi,   1-3, rmin=1.5, rmax=3.5,
                kmin=3,kmax=15,kweight=2,dk=2)
```

The fourth requirement for a fit may seem obvious: `feffit()` needs some variables to vary. **Fitting Variables** are like regular scalars, except that they are defined with the `guess()` command instead of `set()`, as in `guess e0 = 1.` . Fitting variables also carry around information about their estimated uncertainty, and correlation with other fitting variables. We'll discuss that in the next section.

Putting it all together, a simple fit of a single FEFF path to $\chi(k)$ data would look like this:

```
Ifeffit> read_data(file = cu_chi.dat, group= data, type = chi)
Ifeffit> set   s02_001  = 0.9
Ifeffit> guess e0_001   = 1.
Ifeffit> guess dr_001   = 0.01
Ifeffit> guess sig2_001 = 0.001
Ifeffit> path(index  = 1,
              file   = feff0001.dat,
              s02    = s02_001,
              e0     = e0_001,
              delr   = dr_001,
              sigma2 = sig2_001 )
Ifeffit> kmin = 3.0, kmax = 16.0, kweight = 2, dk1 = 2
Ifeffit> rmin = 1.5, rmax = 3.2
Ifeffit> feffit(data.chi, 1, group=fit)
```

The `feffit()` command will update the values of all the fitting variables, and also create arrays for the best-fit $\chi(k)$ and $\tilde{\chi}(R)$ using the group name specified (`fit` in this case, `feffit` by default), and also create arrays for the data $\tilde{\chi}(R)$, effectively running

```
Ifeffit> fftf(data.chi)
Ifeffit> fftf(fit.chi)
```

for you. The best-fit values and estimated uncertainties for the variables can be seen with `show @variables`, as will be further discussed in the next section.

In addition to what has been discussed so far, `feffit()` has several optional parameters for a wide range of things such as specifying uncertainty in the input data, whether to refine background-spline parameters with the structural refinement, how to fit in k-space (with or without Fourier filtering), and an IFEFFIT macro to run for each fit iteration. In the interest of brevity, I'll resist discussing any of these fascinating topics in this brief tutorial.

6.5 Uncertainties in Fitted Variables, Fitting Statistics

A fit is not very useful without some idea of the reliability of the fit and some estimate of the uncertainty in the fitted variables. The `feffit()` command automatically calculates the estimated uncertainties and goodness-of-fit parameters for you. The particulars of how these values are determined is outside the scope of this tutorial – the attention here will be on how to view and manipulate the values for these statistics that `feffit()` determines.

The `feffit()` command writes several fitting statistics to help you determine the goodness of fit. First, IFEFFIT will write the number of fit iterations to `&fit_iteration`. Though not necessarily useful as a fitting statistic, this can be valuable to see if a fit is taking far too many iterations, probably indicating a problem with the set up or model. The number of variables is

stored in n_varys. The number of independent points in the band-limited data is estimated as $2\Delta k \Delta R/\pi$, for fit k-range Δk and R-range ΔR and stored in n_idp. The estimated uncertainty in the data itself is stored in the variables epsilon_r for the uncertainty in the data $\tilde{\chi}(R)$ and epsilon_k for the uncertainty in the data $\chi(k)$.

The main goodness-of-fit statistics are r_factor, chi_square, and chi_reduced, which are three different ways of scaling the sum-of-squares of the final misfit. The full definition of these is given elsewhere, but briefly, r_factor is the misfit scaled to the data itself, so it a *relative misfit*, chi_square is scaled to the estimated uncertainty in the data (epsilon_r for data fit in R-space). chi_reduced is chi_square divided by the number of "free parameters in the fit", given by the difference of n_idp and n_varys. More information on these terms and concepts is elsewhere in the IFEFFIT documentation.

For each fitting variable, IFEFFIT will create (or overwrite) a variable named delta_VAR, and write the estimated uncertainty for variable VAR to this variable. For example, the fit shown above would put the uncertainties in the fitting variables in delta_s02_001, delta_e0_001, delta_sig2_001, and delta_dr_001.

The estimated uncertainties are, of course, very valuable for assessing fit results. In addition, it often instructive to know the correlations between pairs of variables resulting from the fit. To extract the correlations from IFEFFIT, you use the correl() command, which simply takes the names of two fitting variables, and the name of an output scalar variable to store the resulting correlation to. Thus,

```
Ifeffit> correl(s02_001, sig2_001, out= cor1)
Ifeffit> print cor1
 0.887130839
```

At this point, it is an error to give correl() the name of any variable that is not a fitting variable.

7 Modeling non-XAFS data

In the course of data analysis, it is often necessary to fit non-XAFS data. This can be any task similar to fitting a line or quadratic function to a set of data (say $\sigma^2 v.T$), fitting a set of Lorentzian, Gaussian and/or arc-tangent functions to a XANES spectra, or fitting some weighted average of model XANES spectra to fit an unknown spectra. Building on the general data-processing and XAFS modeling capabilities, IFEFFIT provides a simple and powerful way to model many kinds of data.

The `minimize()` command implements a simple least-squares minimization routine. It takes as its primary argument a *residual* array to minimize in the least-squares sense. Usually the residual would be "Data - Fit", but you may want to provide some scaling, possibly emphasizing some region of the fit more than others.

The residual array is built just like any other array, and is expected to depend through definitions on a few variables defined with `guess()`. A very simple example would be a linear fit:

```
Ifeffit>read_data(my_data.dat, group = 'data', label = 'x y')
Ifeffit>guess ( m  = 1, b = 0 )
Ifeffit>fit.y     = m * data.x + b
Ifeffit>fit.resid = fit.y  - data.y
Ifeffit>minimize(fit.resid)
```

This will adjust the variables `m` and `b` to best-fit the data. As with `feffit()`, the fitting statistics `&fit_iteration`, `chi_square`, `chi_reduced`, and `r_factor` will be written and uncertainties in the variables `delta_m` and `delta_b` will be created for the uncertainties in the fitted parameters. The `correl()` function described above can be used to retrieve the correlation between 2 fitting variables. Since the general problem of estimating uncertainties in experimental data is difficult, you'll need to be quite mindful of this fact and careful when interpreting these fitting statistics.

In order to specify an estimate of the uncertainty in the experimental data, create an array (of the same size as the residual array) containing the estimated uncertainty at each point, and tell `minimize()` to use this with the `uncertainty` argument:

```
Ifeffit> fit.eps = 0.01 * indarr(npts(fit.resid))
Ifeffit> minimize(fit.resid, uncertainty = fit.eps)
```

If you wish to limit the fit to be over a limited range of the residual array, you need to tell `minimize()` what array to use as the ordinate or "x"-array, and the minimum and maximum "x" values to use:

```
Ifeffit> minimize(fit.resid, x =data.x, xmin= 2, xmax =10)
```

There are several built-in special mathematical functions to aid in the modeling of data – more information can be found in the Reference Guide.

8 Saving data and logging your IFEFFIT session

This section is about getting information back from IFEFFIT, either at the command-line, or written to external files.

8.1 Writing output data files

At some point, you'll want to write out some of IFEFFIT's arrays to a file. This is done fairly simply with the `write_data()` command, which will write a plain ASCII file containing text strings and scalar values at the top of the file as "comment lines", then arrays in column format. The syntax for `write_data()` is fairly simple, consisting of a file name, and a list of arrays, scalars, and strings to write out, as in

```
Ifeffit> $title1 = 'Fit #1, varying 4 variables'
Ifeffit> write_data(file='x_fit.chi', rmin, rmax,
                    fit.k, fit.chi, $title1)
```

which will write the file *x_fit.chi*, which will have header lines from `$title1`, `rmin`, `rmax` looking like this:

```
# Fit #1, varying 4 variables
# rmin =    1.5000000
# rmax =    3.5000000
#------------------------
#   k           chi
  0.0000000        0.40189216E-01
```

followed by columns for the arrays `fit.k` and `fit.chi`. Notice that the file will have a label line making it ready to be read back in with the `read_data()` command using this "label" line, as described in section 3.

The string program variables are always written first, in the order they appear in the argument list, followed by the scalars (as shown) in the order they appear in the argument. Finally, the arrays will be written in the order they appear, with the first one being in the first column.

8.2 Writing log files

While doing a complex analysis, it's often necessary to save values and information into a file for later inspection. While not extremely sophisticated, one simple way to do this is to have everything that would be written to the screen written to a *log file*. This is surprisingly effective way of keeping track of your analysis session, and may well inspire you to insert many more `show()` and `print()` commands in your scripts.

To open a log file, you use the `log()` command, naming the log file to use, and specify the "screen echo" mode:

```
Ifeffit> log(my_test.log, screen_echo = 3)
```

where the "`screen_echo = 3`" selects writing to both the screen and to the log file. Other values for this parameter are discussed in the Reference Guide.

To close a log file, you say

```
Ifeffit> log(close)
```

Only one log file can be open at a time – opening a log file when one is already open will cause the first to be closed. If the program exits with a log file opened, it is not guaranteed that all information will be actually written to the log file. (NB: This is under investigation, and is hoped to be fixed in a future version).

8.3 Saving the state of an IFEFFIT session

It's often desirable to suspend an analysis session and save the current point in the session, either to return to it later or to compare your results with someone else's. To this end, IFEFFIT allows you to save the current state of all program variables into a single file that can be read in later. The saved file is portable across different platforms.

The `save` command will save the current state. It takes a few optional arguments, the most important of which is the file name, which defaults to *`ifeffit.sav`*. That is

```
Ifeffit>save
```

will write *`ifeffit.sav`*, and

```
Ifeffit>save( current.sav)
```

will save the program state to *`current.sav`*. You can also specify what kinds of program variables to save with optional arguments like `with_strings`, `no_strings`, `with_arrays`, `no_arrays`, and so forth. The default is to save everything.

To restore a previously saved session, you use the `restore` command, giving it the name of the save file (again, *`ifeffit.sav`* by default):

```
Ifeffit>restore( my_save_file.sav)
```

This will load all the program variables saved in the given file. Note that it will *not* erase any program variables that may have existed in the current session, but will overwrite any variables with the same name. This means that unless you start with a new session, it's possible that restoring a session may not produce an identical session to the previous one.

9 Defining and Using Macros

Macros are named sequence of IFEFFIT commands which can be run by simply typing the macro name. Macros are primarily designed to cut down on typing for repetitive tasks and to make IFEFFIT easier to use and customize. While not going far enough to making IFEFFIT a real programming language, macros offer some flexibility normally associated with batch processing languages.

A macro is defined with the `macro` keyword, after which comes a series of commands as you would type them to the command line and ending with `end macro`. As a simple example, defining a macro named `make_ps` would look like this would be

```
macro make_ps
  plot(device="/ps",file= "ifeffit.ps")
end macro
```

Now, typing the new command `make_ps` will create a postscript file named *`ifeffit.ps`* of the current plot. This macro is only one line long, which doesn't save much typing, but we'll get to longer examples below. First, let's add an optional argument: what if you want the Postscript file to be named something other than *`ifeffit.ps`*, since multiple executions would just keep overwriting this file? For this, you can supply *macro arguments*, which are variables named `$1`, `$2`,..., `$9` that are expanded in place as text strings by the first, second, ..., ninth arguments when the macro is run. The definition look like this

```
macro make_ps
  plot(device="/ps",file= "$1")
end macro
```

and would be invoked by `make_ps data_01.ps` or `make_ps my_fit.ps`. OK, but now what happens if you *don't* give an argument? Well, the argument `$1` would be "", so that would be the same as saying `...,file="")`, which would create the default file *`ifeffit.ps`*.

You can also define a default parameter for the macro at its definition, by including the default value on the "`macro`" line. That is,

```
macro make_ps  other.ps
  plot(device="/ps",file= "$1")
end macro
```

the default value for $1 will be `other.ps`. The value for this parameter can still be overridden by saying `make_ps data_01.ps`.

Macros can be of arbitrary length (though the total number of macro lines in memory is fixed at 2048 lines). In addition, if the first line of a macro definition is a plain text string, this will be used as the macro documentation, and will be shown with `show @macros`. A more typical macro may look like this

```
macro do_pre_edge  a
   "Read File, Calculate Pre-Edge, Plot, Write File"
   read_data($1.xmu,  type = xmu,  group = my)
   pre_edge(my.energy, my.xmu)
   my.norm = my.pre / edge_step
   $title1 = 'normalized, pre-edge subtracted data'
   write_data(file = $1.pre, $title1,  energy, pre, norm, xmu)
end macro
```

More example macros are included in the main IFEFFIT distribution.

10 The ifeffit command-line program

The main program distributed with IFEFFIT is simply called `ifeffit` (or `ifeffit.exe` on Win32 systems) and runs the basic command interpreter we've been discussing throughout this tutorial. If you've been following along with the examples, you've run this program already. This, the final section of the tutorial, will give a few additional details of using this program. The emphasis here will be on the Unix version.

On Unix and Unix-like systems, the `ifeffit` program uses the GNU Readline library which supports command-line editing, which greatly enhances the usability of the command-line program by letting you 1) recall previous commands, 2) edit them with keyboard commands, and 3) use tab-completion. In short, the up-arrow and down-arrow keys work to scroll through previous commands, you can easily edit the command-line, and you can use the tab key to help you finish typing commands and file names.

Following the usual GNU Readline behavior, 'emacs keybindings' are used, and there is a large number of 'control-keyboard-sequences' to assist editing the current command line. For example Ctrl-a (that's holding the control key and 'a' at the same time) will move to the beginning of the line, Ctrl-e will move to the end of the line. Both Ctrl-f and the right-arrow key will move one character to the right, and Ctrl-b and the left-arrow key will move one character to the left. Ctrl-p is the same as the up-arrow key, and Ctrl-n is the same as the down-arrow key, and will scroll through the previous and next commands in the "history buffer". Ctrl-k will cut the text from the current cursor position to the end of the line, and Ctrl-y will insert that cut text at the current point of the cursor. Following the behavior of many shell programs, Ctrl-d has a dual purpose: on an empty line it will exit the program (essentially the same as typing 'quit'!), while on a non-empty line it will erase one character at a time. Ctrl-h, on the other hand, will erase one character at a time backwards. There are several more control-sequences available for more rapid command-line editing – consult the Readline documentation.

Tab completion means that if you've typed `read` at the command prompt and then hit the tab key, the program will guess that you wanted to type the command name `read_data`, and complete this word for you. If you type `re` and hit tab, you should hear a beep because there are more than one possible completions – hitting tab again will show the three possibilities `read_data`, `rename`, and `restore`. For the first word on a command-line, `ifeffit` will use a restricted set of commands for tab completion (type 'help' or hit the tab key twice at the command prompt). For later words on the command-line, `ifeffit` will use the set of files in the current working directory for tab completion.

In addition to keeping a 'command history' for the current session, `ifeffit` will maintain a recent history of commands between sessions, and re-load the command history on start-up. This allows you to scroll through the commands of earlier sessions. The commands are kept in memory, and re-saved into a*history file* when `ifeffit` exits. This file can be used as a record of your session, and as a starting place for batch command files, scripts, and macros.

The name of the history file and the number of commands saved can be set with the environmental variables `IFF_HISTORY_FILE` and `IFF_HISTORY_LINES`. If these variables aren't set, the history file used will be named *`.ifeffit_hist`* in your home directory and will contain up to 500 command lines.

When `ifeffit` is started, two 'start-up files' will be loaded, if found. These are loaded simply as command files, as if you typed their contents to the command-line. First, *`startup.iff`* in the installation directory (typically */usr/local/share/ifeffit*) is loaded, and then the file *`.ifeffit`* in your home directory is loaded. These start-up files are intended for site-wide or personal definitions for common macros and color-table preferences. In fact, these

start-up files are not only loaded by the command-line program, but are loaded by the underlying library, and so the definitions given in these start-up files can be used in scripts written using the IFEFFIT library in perl or python. This can lead to a portability problem for scripts, however, and it is not necessarily recommended to rely on definitions in start-up files in lengthy scripts.

When starting the`ifeffit` program, you can give a list of IFEFFIT command files and save files that will effectively be `load`ed or `restoreed` as necessary before the command-prompt is given to you. Just to be clear, these are loaded after any start-up files, so definitions in start-up files can be used in "`load`ed" files. If multiple files are listed they will be `load`ed / `restore`ed in the order specified. To run a command file in 'batch mode', or non-interactively, you can use the `-x` switch, which will load all the files on the command line and then exit, without ever showing the command prompt. Note that `pause` statements, which would normally wait for your input are skipped in this 'batch mode'. For example,

```
~> ifeffit  my_file.iff
```

will load *my_file.iff* and then give you the command prompt to continue, while

```
~> ifeffit -x my_file.iff
```

will run *my_file.iff* and then exit. If running in batch mode, you may want to redirect the screen output to a file (`ifeffit -x my_file.iff > my_file.out`), or use the `-q` switch (`ifeffit -q -x my_file.iff`) to "run quietly", suppressing screen output altogether.

While in an `ifeffit` session, there are a few additional commands available besides the standard commands of the IFEFFIT library. The standard Unix "shell commands" `ls`, `cd`, `pwd`, and `more` work to give information about the current directory and files. Additional shell commands can be executed by preceding it with a `!` sign, as in

```
Ifeffit> ! gv  ifeffit.ps
```

which will run `gv` (a popular postscript viewer) for you. Putting an ampersand `&` at the end of this line will run the job in the background so that you can continue typing in the `ifeffit` shell as well. That makes an effective way to edit command files:

```
Ifeffit> ! emacs test_cmnd.iff  &
Ifeffit> load test_cmnd.iff
```

Finally, typing 'help' at the `ifeffit` prompt will give a one-line description of the more common IFEFFIT commands.