The IFEFFIT Reference Guide

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

Version 1.2.7
Apr 07, 2005

Contents

License

1 Introduction

IFEFFIT is a program and programming library for analyzing x-ray absorption fine-structure (XAFS) data. As its name may suggest, IFEFFIT[1] gives an interactive method for fitting XAFS data using calculations from FEFF, and is based on the fitting program FEFFIT of the UWXAFS3.0 Analysis Package. There have been significant additions to FEFFIT, and the user interface has been completely rewritten.

IFEFFIT is a complete XAFS analysis package, allowing general data manipulation, analysis, and plotting, as well as meeting the unique demands of analyzing XAFS data. For those of you familiar with the UWXAFS3.0 Package, IFEFFIT combines the functionality of AUTOBK and FEFFIT and adds many more features. A major goal of IFEFFIT is to make a graphical user interface (GUI) for XAFS analysis, but IFEFFIT is not simply a GUI program by itself. It is, instead, a command-based library that can be run either as a command-line program or called from other programs, including both graphical and non-graphical interfaces. At this writing, an enhanced command-line program named G.I.FEFFIT is distributed with IFEFFIT, and the GUI programs ATHENA, ARTEMIS, TKATOMS, HEPHAESTUS, and SIXPACK make use of IFEFFIT.

This *Reference Guide* describes the commands and syntax of IFEFFIT, and is intended for people who want direct access to the underlying 'IFEFFIT engine', either directly from the command-line program or through their analysis scripts. While the high-level GUI wrappers such as ATHENA, ARTEMIS, and SIXPACK provide most of the functionality needed for XAFS analysis, some sientists will appreciate a simple, scriptable command-based interface to a core set of XAFS analysis routines. The commands and syntax described here are an attempt to provide such an interface.

This guide starts with an example in Chapter 2 and then discusses the general syntax and structure of the commands in Chapter 3. The next several chapters give more in-depth information about input and output files, plotting, fitting XAFS data with FEFF calculations, and fitting non-XAFS data to simple functions. Chapter 9 lists all the commands and their complete syntax. Chapter 10 discusses writing simple macros for IFEFFIT and more complicated scripts using scripting languages.

In addition to this *Reference Guide*, there is also *The* IFEFFIT *Tutorial* which gives a more gentle introduction to IFEFFIT. More information about IFEFFIT, including installation instructions, further examples, an archived mailing list, and the other documention are available with the source code and at the IFEFFIT web site: http://cars9.uchicago.edu/ifeffit/

Acknowledgements

IFEFFIT was written with helpful suggestions from Julie Cross, Bruce Ravel, and John Rehr. More detailed and up to date acknowledgements can be found in the `THANKS` file distributed with IFEFFIT.

[1] The name was originally intended to stand for INTERACTIVE FEFFIT, but I sort of like the simple self-declarative nature of the name as well as the literary allusion to *I, Claudius* and *I, Robot*. It could also be interpreted as the answer to the question, "How do you analyze your XAFS data?". It is in no way intended as a subliminal insertion of FEFF into that *other* Chicago technological institute known to do some XAFS.

2 The Basics of IFEFFIT

IFEFFIT is a command-based program. That is, you tell IFEFFIT to do something, it does that and then waits for you to tell it what to do next. The commands that IFEFFIT accepts are simple statements (there are no loops or conditional statements) useful for data manipulation, and especially for XAFS analysis. Most commands tell IFEFFIT to manipulate arrays of numerical data. There are commands for reading arrays from files, writing arrays to files, plotting arrays, doing simple mathematical manipulation of arrays, and more XAFS-specific commands such as background-spline removal and Fourier transforms. It can also fit XAFS data using theoretical standards from FEFF with complex modeling abilities and automated error analysis. This chapter gives a quick overview of IFEFFIT with a simple annotated example. Much of this material is also covered in *The* IFEFFIT *Tutorial*.

2.1 Starting the program

Typing `ifeffit` at the system command prompt will start the basic IFEFFIT command-line program. You should get a set of messages and a command prompt that looks like this:

```
Ifeffit 1.2.7  Copyright (c) 2005 Matt Newville, Univ of Chicago
                 command-shell version 1.1 with GNU Readline
Ifeffit>
```

At this point, you're ready to start typing IFEFFIT commands at the prompt. If you don't get such a prompt, the installation on your machine is probably corrupted. Detailed installation instructions are available with the IFEFFIT distribution.

2.2 A Sample Run

We start with a fairly complete example (see the tutorial for a more gentle introduction). Let's say you have some raw data from a beamline in a plain-text column format, and you want to convert it to $\mu(E)$, do a background subtraction, and then a Fourier Transform to see what the data looks like in R-space. Though a very practical request, it's really quite a bit of data processing, so this is a fairly intense example. Here's what the session might look like:

```
Ifeffit> read_data(file=Cu.dat, type=raw, group= cu)
Ifeffit> cu.energy  = cu.1 * 1000.0
Ifeffit> cu.xmu     = ln(cu.2 / cu.3)
Ifeffit> spline(energy = cu.energy, xmu = cu.xmu,
Ifeffit>        rbkg=1.1, kweight=1., kmin=0)
Ifeffit> plot(cu.energy, cu.xmu)
Ifeffit> plot(cu.energy, cu.bkg, xmin=8850, xmax=9300,
Ifeffit>                         color=red)
Ifeffit> kweight = 2.0,  cu.chi_kw = cu.chi * cu.k^kweight
Ifeffit> newplot(cu.k,  cu.chi_kw)
Ifeffit> fftf(real = cu.chi, kmin = 2.0, kmax = 13.0,
Ifeffit>      dk = 1.0, kweight=2)
Ifeffit> newplot(cu.r, cu.chir_mag, xmax=8)
Ifeffit> $title1 = "Test: writing out  k, chi, chi*k"
Ifeffit> $title2 = "   data from Cu.dat,   rbkg = 1.0"
Ifeffit> write_data(file = Out.chi,  cu.k, cu.chi,
Ifeffit>            cu.chi_kw, $title1, $title2)
```

One important aspect of IFEFFIT is that you can save commands into a file and execute all commands in that file at one time. By saving the above commands into the file *process.iff*, we could simply type `load process.iff` at the IFEFFIT command line. These two methods of running these commands are completely equivalent.

We'll now go through each of these lines in detail. At times there may be *too* much detail here. If so, please go through the *The* IFEFFIT *Tutorial* and be patient.

```
Ifeffit> read_data(file=Cu.dat, type=raw, group= cu)
```

This command reads in data arrays from the ASCII column file *Cu.dat*. The arguments `type=raw` and `group=cu` help `read_data()()` name the arrays it reads in. Because arrays are often read in and processed together, it is convenient to give them names that are related. Arrays names always have two parts – a prefix and suffix, with a dot '.' in between. The prefix gives the group name, and the suffix explains what the data contains. Here, `cu` is used as the group name (prefix). The type `raw` is the simplest type, so the suffixes will just be the column index. To make a long story short, we just read in the arrays `cu.1`, `cu.2`, and `cu.3`.

```
Ifeffit> cu.energy  = cu.1 * 1000.
Ifeffit> cu.xmu     = ln(cu.2 / cu.3)
```

Presumably, we know what the contents of our data file. For this *Cu.dat* file, the column contained energy in keV, the second contained I_0 and the third I, for absorption data measured in transmission. There might have been more columns in the file, but this is all we need at this point. IFEFFIT prefers to think about energy in eV not keV, so we make an array `cu.energy` that has energy in eV, and then we calculate $\mu(E)$ and call that `cu.xmu`. Note that the math here is done on all elements of the array.

```
Ifeffit>  spline(energy = cu.energy, xmu = cu.xmu,
Ifeffit>         rbkg=1.1,kweight=1.,kmin=0)
```

This computes the background spline $\mu_0(E)$ for our $\mu(E)$ using the AUTOBK algorithm. The argument `energy = cu.energy` names the array to use as the energy values, and `xmu = cu.xmu` names the $\mu(E)$ array. `rbkg=1.1` sets the value of $R_{\rm bkg}$, while `kweight=1.` sets the k-weighting, and `kmin=0` sets the value of $k_{\rm min}$.

Like many other commands, `spline()` uses, modifies, and (if necessary) creates several arrays and scalars. The complete list of what `spline()` uses is listed in section 9.41. For now a partial list will do: `spline()` sets the arrays `cu.bkg` to contain $\mu_0(E)$, `cu.k` to contain the k values, and `cu.chi` to contain $\chi(k)$. Several scalar values (including `rbkg`, `kweight`, `kmin`, and `e0`) are also set by `spline()`. You can see the values of these variables with the `show()` command – try `show(e0, kmax)` for example.

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

This plots $\mu(E)$. That is a plot window should appear and a trace of $\mu(E)$ should be drawn on it. If this does not happen, please consult the installation instructions. The `plot()` command has many optional arguments, but here we're just giving the array names for the ordinate `cu.energy` and the abscissa `cu.xmu`.

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

This adds a plot of $\mu_0(E)$ to the previous plot. We specify the x range of the plot with `xmin = 8850, xmax = 9300` to look at just the near-edge region, and explicitly give the color to use for $\mu_0(E)$. Note that overplotting is the default behavior. See section 9.27 for all the plotting options, and chapter 5 for even more information about plotting with IFEFFIT.

```
Ifeffit>  kweight = 2.0,  cu.chi_kw = cu.chi * cu.k^kweight
```

This sets the value of `kweight` and uses the new value to create the array `cu.chi_kw` which contains $k^2\chi(k)$. Note that multiple "set variable" commands were put on a single line.

```
Ifeffit>  newplot(cu.k,  cu.chi_kw)
```

This plots the k-weighted $\chi(k)$ that we just calculated. `newplot()` is a variation of `plot()` command that reinitializes the plot, so that it won't be plotted over the current window (which was still showing $\mu(E)$ and $\mu_0(E)$).

```
Ifeffit>  fftf(real = cu.chi, kmin = 2.0, kmax = 13.0,
Ifeffit>       dk = 1.0, kweight=2)
```

This does the forward XAFS Fourier transform of $\chi(k)$. The argument `real = cu.chi` tells `fftf()` to use `cu.chi` (which is the un-k-weighted $\chi(k)$ from `spline()`) as the real part of the function to Fourier transform. We could have said `imag = cu.chi` to use $\chi(k)$ as the imaginary part. This is, to some extent, a matter of convention – see *XAFS Analysis with* IFEFFIT for more details.

The Fourier transform window was specified with the parameters `kmin`, `kmax`, and `dk`. The k-weight parameter for the Fourier transform will be read from the variable `kweight`, which we just defined as 2. `fftf()` creates arrays `cu.r` for R, and `cu.chir_re`, `cu.chir_im`, and `cu.chir_mag` for the real part, imaginary part, and magnitude of $\tilde{\chi}(R)$, among other things.

```
Ifeffit>  newplot(cu.r, cu.chir_mag, xmax = 8)
```

Now we plot $|\chi(R)|$, explicitly limiting the R-range to 8Å. By default, the R-based arrays from `fftf()` will extend to 10Å.

```
Ifeffit>  $title1 = "Test: writing out  k, chi, chi*k"
Ifeffit>  $title2 = "   data from Cu.dat,   rbkg = 1.1"
```

Here we define a pair of *text string* variables, which always have names starting with a dollar sign. These are useful for doing things like

```
Ifeffit>  write_data(file = out.chi,  cu.k, cu.chi,
Ifeffit>             cu.chi_kw, $title1, $title2,e0,rbkg)
```

in which we save the $\chi(k)$ and k-weighted $\chi(k)$ data to the file *out.chi*. The rest of the arguments list the arrays, text strings, and scalars to write to the output file. Text strings will be written first, then the scalars, and finally the data arrays will be written, all given in the order listed. This ends the annotated example.

2.3 The `show` and `print` Commands

Two very important commands that you definitely want to know about were left out of the above example. These are the `show()` and `print()` commands, which will write out information about Program Variables, including their values. The `show()` command takes a simple list of program variables, like this:

```
Ifeffit>  show e0, kmin, $title1, cu.chi
```

and will print something like

```
e0             =  8982.315
kmin           =  2.000000
$title1        = Test: writing out  k, chi, chi*k
cu.chi         = 302 pts [ -0.3232080 : 1.233829 ]
```

`show()` doesn't print out entire arrays but gives just enough information (the number of points, and the minimum and maximum value) to convince you that an array exists. The `show()` command can also be used to show all current scalars, arrays, and strings. You can read more about the `show()` command in section 9.40.

Whereas the `show()` command will only report about existing variables, and will not do any processing, the `print()` command is a bit more literal, printing out the *values* of variables or expressions. That is, `print(e0)` will just print the numerical value of `e0` to the screen:

```
Ifeffit>  print e0
     8982.315
Ifeffit>  print "sqrt(25) + 1.001"
      6.00100
```

Note that in the last example, the math expression was enclosed in double quotes. This (or equivalently, enclosing braces "{}") tells IFEFFIT to evaluate the expression, instead of printing it literally, and tells IFEFFIT where the expression ends. Using single quotes would print the expression literally:

```
Ifeffit>  print 'sqrt(25) + 1.001 = ', "sqrt(25) + 1.001"
  sqrt(25) + 1.001 =  6.00100
```

You can also `print()` out several expressions at once:

```
Ifeffit>  print "sqrt(25) + 1.001", "pi / 2"
      6.001000      1.570796
```

Using `print()` for arrays (or an expression that gives an array) will print all the values of the arrays:

```
Ifeffit>  print indarr(4)/5
    0.2000000     0.4000000     0.6000000     0.8000000
```

Section 3.7 gives a more complete description of all the ways of getting information back from IFEFFIT. We'll come back to `show()` and `print()` in section 3.8, where the ability to change where these outputs are printed will be discussed.

At this point, you may find it useful to repeat the above example session mixing in `show()` or `print()` commands after every line, and plotting some of the other arrays. That should give you enough of a feel for IFEFFIT to be able to use it for simple data processing and allow you to use the rest of this document as a reference guide.

3 Structure and Syntax of IFEFFIT

This chapter takes an in-depth view at IFEFFIT's data and syntax. It is more formal and in-depth than the previous chapter, but most of the topics here were touched on there. For those of you with some programming experience or some familiarity with FEFFIT, nothing in this chapter should be too confusing. In some sense, IFEFFIT is a language for data analysis and this chapter describes the IFEFFIT language. As a programming languages, IFEFFIT is pretty weak, missing many key features. Since IFEFFIT is really XAFS data analysis program, I'll try not to make this sound like a programming text, and keep the examples in the previous chapter close at hand throughout this chapter.

IFEFFIT keeps a common storage areas for all its data, and gives you access to this data through named variables. You are allowed to pick your own variable names for your data and create and do simple algebraic manipulation on your data. In the previous chapter we saw an example of this when we read in data from a file to variables with names corresponding to the column labels in the file, and then constructed $\mu(E)$ ourselves by simple manipulation of this data.

3.1 Commands

IFEFFIT is a command-based program, and every operation you type at the command-line or send to IFEFFIT through a script is a interpreted as a command. Most IFEFFIT commands have syntax like this:

```
Ifeffit> command(argument1, argument2, ... )
Ifeffit> command(keyword= value, keyword= value,
                 keyword= value, ...).
```

That is, there is a unique command name that is given first, then a set of arguments separated by commas. Most arguments are keyword/value pairs, which have the form `keyword= value`, where `keyword` is a predefined string and `value` is the value you wish to assign to it. Some commands use arguments that are not keyword/value pairs, but have some or all of their arguments be lists of strings or program variables, but the majority are keyword/value pairs. We'll come back to these topics in section 3.6.

The enclosing parentheses for commands are optional, so that

```
Ifeffit> command keyword= value, keyword= value
```

is allowed. If you use an open paren "(", you must match it with a close paren ")", even if that means the command extends over several lines. That allows commands to be extended over several lines, which will be necessary in some cases. Commands are not processed until they have matching open and closing parentheses. When using the command-line program, you'll get a prompt of `...>` instead of the usual `Ifeffit>` when a command is partially completed.

As we will saw in the example in the previous chapter, and as we will see in the next few sections, a significant amount of the command-line processing you will do with IFEFFIT is to create and define Program Variables from your data. These procedures are done with the commands `def()` and `set()` which differ slightly but which both have the effect of specifying a value for a Program Variable. That is

```
Ifeffit> def(kweight = 2.0,  kmin = 2.50, kmax = 15.4)
```

will create and define (or redefine, if they already existed) values for the Program Variable `kweight`, `kmin`, and `kmax`.

But, as you have probably noticed, we didn't use the `def()` or `set()` command in the previous example, and simply said

```
Ifeffit> cu.energy  = cu.1 * 1000.0
Ifeffit> cu.xmu     = ln(cu.2 / cu.3)
```

This is because the default command it `def()`. That is, if the first word on the command line is not a known command or macro (which we'll get to eventually), the `def()` command is used.

An important consequence of this is that when you do really want to use the `set()` command, you'll need to specify it explicitly. We'll come back to this topic in section 3.3. For know, you can simply remember that

The default command is `def()`.

3.2 Scalars, Arrays, and Strings

IFEFFIT has three types of variables: numeric scalars, numeric arrays, and text strings. I'll call all of these "Program Variables" throughout this Reference Manual to mean all the named quantities that IFEFFIT knows about – you shouldn't confuse "Program Variables" with values that are adjusted in a fit which I'll call "Fitting Variables" (see section 3.4) when there's room for confusion. If you have some programming experience, IFEFFIT's data types should be familiar: numeric scalars are real numbers, numeric arrays are arrays of real numbers, and text strings are sequences of characters. If that you're not familiar with the ways computers store data, numeric scalars contain a single number, numeric arrays contain a set of numbers that is referred to as a whole, and text strings contain a single line of text. IFEFFIT uses double precision (64bit) floating point numbers for all it's numeric values, and does not have a separate integer type. Due to implementation limitations, the maximum number of elements in an array is 16384, and the maximum length of text strings is 128 characters. These limits are set when IFEFFIT is built. In principle, these limits could be changed, but I wouldn't guarantee it to be easy to do.

All Program Variables are named, and are *global* in the sense that every part of IFEFFIT sees the same variables using the same names. As you use IFEFFIT, you will define variables for your data, and tell the commands which data to use by name. In turn, IFEFFIT's commands will access variables by name, and may even create named variables or overwrite the values of variables already defined. That is, the variables you create and the ones that the IFEFFIT commands use will live in the same "name-space". This gives you and IFEFFIT's commands equal access to the variables. Thought this should definitely be seen as a strength, it also means that you should use some care in naming your variables. Normally, there's not much too worry about, as long as you avoid things like trying to store the value of E_0 in a variable named `kmin`. In general, IFEFFIT expects named variables to store quantities **it** expects. A glossary of the names and meanings IFEFFIT expects is listed at the end of Appendix A.

To create a variable, you would say something like `phi = (1 + sqrt(5))/2`. To set the value of k_{min} (used by several commands for Fourier transforms), you might say `kmin = 1.00`. You can also use any variable to assign to other variables. If you were so inclined, you could have said something like `kmin = phi*(phi-1)`, or possibly something along the lines of `x= exp(0), kmin = 4*atan(x)/pi` is more of your idea of a good time. Actually, these aren't exactly equivalent ways to define variables, but we'll get to that shortly.

IFEFFIT uses a simple and strict naming convention for Program Variables, and distinguishes the 3 variable types *by name*. This means that everyone (and every command) can tell the

variable type from its name alone. The convention may be a somewhat unusual, but shouldn't be very difficult to get used to. The rules are:

1. Text strings have names that begin with a dollar sign ($). Names for numeric scalars and arrays do not begin with a dollar sign.

2. Arrays have names that contain one dot (“.”), with at least one character before and after the dot. This gives arrays a prefix and suffix, which leads to a convention: the *prefix* is associated with the *group* of the array, and the *suffix* describes the contents of the array. Neither scalars nor text strings may have a dot in their name.

3. Scalars, the prefix and suffix of arrays, and the characters in the name of a text string after the dollar sign can contain only letters, numbers, '&', '?', ':', and '_' (underscore). They are limited to 64 characters.

4. Scalars and array prefixes names cannot begin with a numeral. Text strings and the suffixes of array names can begin with a numeral.

5. Variable names are not case-sensitive.

Some examples of the naming convention: '$file' and '$plot_color' are text strings. So is '$20', but '$19.95' is *not* allowed. '$1' is a valid name for a text string, but IFEFFIT uses '$1' ... '$9' as special variables for macro arguments (see chapter 10) and it will likely cause confusion if you actually try to use such variable names, especially in macros. '$test.1' is not allowed, nor is 'dec$window' (it contains a '$' in the middle of it's name, and might inspire a wistful pining for VMS). Array names look like 'my.energy' and 'data.chi'. 'X.11' is allowed, and as we saw in the previous chapter, the common convention for how to name data from column files, but '8.3' is not allowed as a variable (because it's a plain number). Typical scalar names are 'E0' and 'edge_step'. '10th_var' is not allowed, since it starts with a numeral. Neither 'data-10' nor 'the_end.' are allowed (the former contains a minus sign, the latter has a dangling dot '.').

'&' is allowed in variable names, and by convention, several built-in "system" scalar variables begin with '&', notably `&print_level` and `&screen_echo`. The characters '?', '_', and ':' are also allowed in variable names, and are not currently given any special meaning. Speaking of special characters, '@' is not allowed in any variable names, but is used with the `show()` command as a primitive wildcard or glob character. In addition, '*' is used in the `write_data()` command as a glob character, to match multiple string names, but isn't allowed in real string names. We'll come back to these glob characters later.

The naming convention gives a clear distinction between the three types, making life easier on all of us. Text strings will likely be used less than numbers, so the extra $ (with the mnemonics of "string" for English speakers) was chosen for text strings. The '.' in an array name suggests that they are associated with files (or data structures for the programmers out there). It also gives a prefix and a suffix to the array names, which gives a handy convention for grouping them together. Though this convention can be disregarded, it is generally a good idea to use the prefix to associate related arrays (say, from the same file, or along a single line of analysis), and to use the suffix to distinguish the contents of the array. This allows `read_data()` to assign decent array names based either on the column labels in the file or on the `type` and `group` keywords. It also means you can have arrays named `Cu1.xmu` and `Cu2.xmu`, containing $\mu(E)$ data from two different files. A `spline()` command with `Cu1.xmu` would generate `Cu1.bkg`, etc, while a `spline()` on `Cu2.xmu` would generate `Cu2.bkg`, etc.

One more thing on naming conventions: though we haven't discussed macros much yet, macros are named sequences of IFEFFIT commands. They share the same naming rules as plain scalars. In fact, *commands* also share the same naming rules as plain scalars. To avoid confusion, you cannot assign a scalar a name of an existing macro or command. By the same token, you won't be able to create a macro with the name of a known scalar or command.

Before leaving this section I should admit that there actually is some data that you do not have direct access to through the Program Variables. The most notable example of data that is *not* available through Program Variables is the complete set of information for an EXAFS Path. The hiding of this data was a difficult design decision, but I felt it provided a simpler and cleaner interface [2], and one that was more amenable to expanding. For the most part, you probably won't even notice that this path data is missing. Besides, almost all of the information about a Path can be converted into Program Variables.

3.3 Dynamic Variables: `Set`, `Def`, and `Sync`

As seen in the previous chapter, you can define your own variables in IFEFFIT simply like this:

```
Ifeffit>  a  =  1,  b  =  3
Ifeffit>  c  = (a + b)/2
```

When you say something like this, it's pretty clear that the value of `c` should be 2. But a complication can arise when you redefine a variable. For example, if you now say

```
Ifeffit> b  =  5
```

Should `c` be 2 or 3? There are two slightly separate parts to this question, and both are important for understanding how IFEFFIT works: 1) should IFEFFIT store the value or the formula for a defined variable? and 2) if the formula is stored, how often should the value be re-evaluated?

The answer to question 1 is that IFEFFIT can store either the value or the formula for a variable. It stores the formula by default, which means that `c` will be 3. If you don't want the formula stored, but want the current value of an expression *at the time of definition*, use `set()` command should be used:

```
Ifeffit> set (c  = (a + b)/2 )
```

With `set()`, the formula is not stored, so no matter what values `a` or `b` are changed to, `c` will be 2.

This "store the formula" aspect of IFEFFIT is fairly unusual in data-processing programs and languages. IFEFFIT is primarily designed for complex data *modeling*, and this approach is extremely useful for setting up complex models for fitting.

Well, that brings us to the second question: if the formula is stored, when will `c` be re-evaluated? Normally, IFEFFIT automatically re-evaluates the variables for you. As you might imagine, if lots of interrelated formulas are stored, re-evaluating them to make sure they're all consistent can turn become complicated and inefficient. IFEFFIT tries to hide all this from you,

[2] If you have some programming experience, it might help to think of Paths as Objects: A complex data structure which you cannot access directly, but only through the supplied methods. In this view, path() defines and creates a path object, and the commands `show()`, `ff2chi()` and `feffit()`, the functions `debye()` and `eins()`, and special variables like `reff` and `degen` access this data.

and isn't as inefficient as you might guess[3]. It's possible that IFEFFIT can get confused about variables and their interdependencies. You can force the variables to be re-evaluated with the `sync()` command. `sync()` checks the dependencies of each variable (both scalars and arrays), re-orders the list of variables, and makes sure that the values are up-to-date. `sync()` is done internally at the beginning of many IFEFFIT commands. These are listed in chapter 9, and include `def()`, `set()`, `plot()`, `write_data()`, and the fitting commands (`minimize()` and `feffit()`). You should never need to use `sync()` explicitly, but it's there if you think you need it.

You can see all this in action with `show @scalars` and `show @arrays`, which list variable names, values, and *formulas* for scalars and arrays. In fact, the variable list shown is as re-ordered by `sync()` to reflect the order of inter-dependencies of variables, so that the variables can be correctly evaluated in one pass. Trivial formulas like '`x = 1 + 1`' are not stored, since IFEFFIT can easily tell that the value of `x` won't change.

Before moving on, let me try to clarify the distinction between `set()` and `def()` one more time. Everything you type at the command line is interpreted as a command. The default command is `def()`. When you type `a = 1, b = 3`, IFEFFIT first looks at `a` to see if it's a command. Recognizing that it is not, IFEFFIT translates the line to `def(a = 1, b = 3)`. This has the consequences that you don't really need to type `def()`, though keeping in mind that it's really there all the time is generally a good idea.

3.4 Fitting Variables: `Guess`

In the previous section, we discussed two varieties of scalars: those that are `def()`ined as expressions of other scalars and arrays, and those that are `set()` to a static value. There is actually a third category: *fitting variables*. These are a lot like static (`set()`) scalars except that their values will be changed by the fitting commands `minimize()` and `feffit()`. Fitting variables are a little special in that they keep track of their uncertainties and correlations with other variables as well as their value. To define a fitting variable, and to give the initial value for it, you use the `guess()` command, which has a syntax like this:

```
Ifeffit> guess my_age = 19
```

There are no *fitting arrays* or *fitting strings*.

It should be clear that many commands alter the values of scalars. For example, `pre_edge()` can alter the value of `e0`. But in this case `e0` is not considered a fitting variable in `pre_edge()`, because it is not defined as a `guess()`ed scalar. You may, however, define a fitting variable `e0` when doing fitting of the XAFS with the `feffit()` command. That is, a scalar is a fitting variable if it was defined with `guess()`, not just if its value changes.

After a fit is executed (either with the `feffit()` or `minimize()` command), an estimate of the uncertainties in the fitted variables will be determined and stored in scalar variable. The variable will be named with a `delta_` pre-pended to each variable name: `delta_air_lines` will contain the uncertainty in `air_lines`, and so on.

Finally, there is an `unguess()` command that will turn all fitting variables into regular scalars, with their current value. This effectively does a `set()` on all variables, and can be very convenient when changing fitting models, as unused variables can make it impossible to

[3] That is, it's not storing the text string for the formulas and re-using them – that would be far too inefficient for data modeling. Instead, IFEFFIT parses and converts the formula into a sort of 'byte-code' for easy re-evaluation. The `sync()` command inspects this 'byte-code' for each formula to determine the simplest order of re-evaluation of the variables so that all the inter-dependencies are satisfied

determine error bars. For complicated macros, scripts, or programs, it is usually a good idea to execute an `unguess()` before executing your `guess()` commands to clear any unwanted variables.

3.5 Mathematical Syntax and Operations

As mentioned in the previous section, the definitions of numeric scalars and arrays are interpreted as mathematical expressions. These are simple algebraic expressions, using numbers, named variables (scalars and arrays), mathematical operations, and intrinsic functions. The syntax is fairly standard, and the case of the operators and variable names is ignored.

When arrays are used in an expression, the result will typically be an array, with each element of the array being operated on. For example,

```
Ifeffit> my.y =  sin(my.x)
```

will define `my.y` as an array of the same size as `my.x`, with the sine operation done on each element of the array. In addition, arrays can be *built* with IFEFFIT using functions such as `range`,

```
Ifeffit> my.x =  range(1,4,1)
Ifeffit> print my.x
  1.0000  2.0000  3.0000 4.0000
```

Several more functions for creating and manipulating array data, as well as details of built-in math functions are given in the following sections.

3.5.1 Common Math Operations

The supported math operations include `*`, `/`, `+`, `-`, `**`, and `^`, with exponentiation done with either `**` or `^`. Standard math precedence (quantities inside parentheses first, from inner to outer parentheses, then `**` and `^`, followed by `*` and `/`, and then + and -) is obeyed[4], but parentheses are encouraged. If syntax errors like out-of-range arguments or nonsense math operations are give, an error message will be printed, and the value will be zero.

There are two built-in constants, whose values you won't be able to change: `pi` gives the value of π; 3.1415926..., and `etok` gives the value of $2m_e/\hbar^2$ in units of $1/\mathrm{eV\AA^2}$; 0.2624683.... The utility of π should be obvious. For XAFS, converting photo-electron energy E in eV to wavenumber k in $\mathrm{\AA}^{-1}$ can be done as `k = sqrt(E * etok)`. This line just keeps

3.5.2 Common Math Functions

The supported math functions are listed in Table 1, Table 2, and Table 3. You'll find all the usual trigonometric functions there, as well as a few other special functions. Note that both `log` and `ln` give the natural logarithm, base e. All trigonometric functions use radians.

While most functions listed in Tables 1 should be familiar, there are a few here that may need further explanation. Gamma, log-gamma, and error function are supported as `gamma()`, `loggamma()`, and `erf()`, following the usual description found in standard mathematical handbooks.

[4] Actually, IFEFFIT violates standard math precedence by incorrectly associating exponentiation from left to right instead of right to left. That is, it evaluates `4**3**2` as 4096 (= (4**3)**2) instead of the correct value of 262144 (= 4**(3**2)). Fixing this is not high on the priority list.

Table 1: Table of Mathematical Functions, Part I (common functions). All function arguments can be expressions themselves. The listing below indicates the expected type for all these functions as scalar. When applied to arrays, these functions return an array of values with the function applied to each element of the argument.

Function Prototype	Description
y = abs(x)	absolute value
y = min(x1,x2)	smaller of two values
y = max(x1,x2)	larger of two values
y = sign(x)	sign (1.0 or -1.0) of x
y = sqrt(x)	square root
y = exp(x)	exponential, base e
y = log(x)	logarithm, base e
y = ln(x)	logarithm, base e
y = log10(x)	logarithm, base 10
y = sin(x)	sine
y = cos(x)	cosine
y = tan(x)	tangent
y = asin(x)	arc-sine
y = acos(x)	arc-cosine
y = atan(x)	arc-tangent
y = sinh(x)	hyperbolic sine
y = cosh(x)	hyperbolic cosine
y = tanh(x)	hyperbolic tangent
y = coth(x)	hyperbolic cotangent
y = gamma(x)	gamma function
y = loggamma(x)	log of gamma function
y = erf(x)	error function
y = gauss(x,x0,sigma)	Gaussian function
y = loren(x,x0,sigma)	Lorentzian function
y = pvoight(x,x0,fwhm,eta)	pseudo-Voight function

The functions `gauss(), loren(),` and `pvoight()` are available for constructing typical lineshapes to model data, which are especially convenient when used the general least-squares minimization command `minimize()`. The functions are most likely used with the first argument being an array, though this is not required. The versions here are properly normalized so that they integrate to 1. For example, `y = gauss(x,x0,sigma)` gives

$$y = \mathcal{G}(x, x_0, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp[-(x - x_0)^2/2\sigma^2)] \tag{1}$$

where x is `x`, x_0 is `x0`, and σ is `sigma`. Similarly, `y = loren(x,x0,sigma)` gives

$$y = \mathcal{L}(x, x_0, \sigma) = \frac{\sigma/2\pi}{(x - x_0)^2 + (\sigma/2)^2} \tag{2}$$

The pseudo-Voight function is a linear combination of a Gaussian and Lorentzian function with the same full-width-at-half-maximum (FWHM), often used to describe real diffraction

Table 2: Table of Mathematical Functions, Part II (array-specific functions). All function arguments can be expressions themselves. The listing below indicates the expected type (`x` for scalar, `my.x` for array) for arguments and results. Functions with array arguments return a scalar value (`npts()` ...`vprod()`, `nofx()`) will work on scalar arguments, with trivial results. Functions with a scalar arguments that return arrays (i.e., `indarr` ... `range`) will use the first element of an array argument as the scalar used.

Function Prototype			Description
y	=	floor(my.a)	smallest element of array
y	=	ceil(my.a)	largest element of array
y	=	vsum(my.a)	sum of all elements of array
y	=	vprod(my.a)	product of all elements of array
y	=	npts(my.a)	number of elements in array
my.y	=	indarr(n)	generate array 1, 2, ..., n
my.y	=	ones(n)	generate array of n ones
my.y	=	zeros(n)	generate array of n zeros
my.y	=	range(start,stop,step)	generate array over a given range
my.y	=	join(my.x1,my.x2)	concatenate 2 arrays
my.y	=	slice(my.x,n1,n2)	generate sub-array my.x[n1:n2]
n	=	nofx(my.x,x)	index of my.x with value closest to x

lineshapes. Since the same FWHM for the Lorentzian and Gaussian functions imply different values of σ, the result is that `y = pvoight(x,x0,sigma)` gives

$$y = \eta\mathcal{L}(x, x_0, \sigma) + (1 - \eta)\mathcal{G}(x, x_0, \sigma_g) \tag{3}$$

where the specified FWHM = σ and $\sigma_g = \sigma/2\sqrt{2\ln(2)}$.

3.5.3 Array-Specific Operations

As mentioned above, when arrays are used in expression, the result is usually an array. In addition, the functions listed in Tables 2 are designed especially to work with arrays, and may either create arrays given a few scalars or return a scalar given an array expressions. For example, the functions `ceil(), floor(), vsum(), vprod(),` and `npts()` each return a scalar given an array argument. `ceil()` and `floor()` give the maximum and minimum element of the array, respectively. `vsum()` returns the sum of all elements of an array, `vprod()` returns the product of all the elements, and `npts()` returns the number of points in an array. As with all functions, these can be used on expressions as well as on named arrays. Note that `ceil()` is different from `max()`, as `ceil()` returns the single largest value of an array, while `max()` returns the larger of two values. The functions `floor()` and `min()` are not the same.

As mentioned earlier, there are a number of functions to help build arrays from scratch. The function `indarr()` takes one scalar argument as an array length and fills the array with integers: 1, 2, 3, ... For example

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

will fill `my.index` be the array (1, 2, 3, ..., 10). The functions `ones()` and `zeros()` create arrays of a specified length, only the created arrays will have all elements set to 1 and 0, respectively. Of course, these functions can be used anywhere in a math expression:

```
Ifeffit> my.xval  =  1 + indarr( max(x1,1)  ) /100
```

It is an error to give these functions an argument that evaluates to less than 1. If the argument to one of these three functions is itself an array, the first element will be used as the dimension, and the rest of the array will be ignored.

The function `range()` is a more general function for creating evenly spaced arrays. It takes three arguments: a starting value, a stopping value, and a step size for the array. That is

```
Ifeffit> my.x  = range(3,10,1)
```

will create the array (3, 4, 5, ... 10), and

```
Ifeffit> my.x  = range(2,4,0.1)
```

will create an array with values (2,2.1,2.2,..., 3.8,4.0). If the range is not an exact multiple of the step size, `range()` will make sure all elements of the array are within the range specified by the start and stop values. Negative step sizes are allowed, but should be used with care.

Arrays can also be built up and broken apart using the `join()`, `slice()`, and `nofx()` functions. The `join()` function concatenates two arrays, for example

```
Ifeffit> my.x1  = indarr(3)
Ifeffit> my.x2  = range(10,16,1.5)
Ifeffit> my.x   = join(my.x1,my.x2)
Ifeffit> print my.x
1.0000  2.0000  3.0000  10.0000  11.5000  13.0000  14.50000  16.0000
```

The `slice()` function will select a *sub-array*, or portion of the array defined by indices for the first and last point desired. Counting of indices begins with 1, so that

```
Ifeffit> my.x1  = indarr(10)
Ifeffit> my.x2  = slice(my.x1,3,6)
Ifeffit> print my.x2
 3.0000  4.0000  5.000  6.0000
```

The function `nofx()` will return the index of an array nearest to a given scalar value:

```
Ifeffit> my.x1  = range(90,180,9)
Ifeffit> nx     = nofx(my.x1,126)
Ifeffit> print my.x1
 90.0000  99.0000  108.000  117.000  126.000  135.000
144.000  153.000  162.000  171.000  180.000
Ifeffit> print nx
  5.0000
```

If multiple values in the array match (or are equally close to) the requested scalar, the first occurrence of "the closest value" will be reported.

Table 3: Table of Mathematical Functions, Part III (special functions). See notes for Tables 1 and 2 for explanation of arguments and text for explanation of the functions themselves.

Function Prototype			Description
my.y	=	deriv(my.x)	finite-difference derivative of array
my.y	=	smooth(my.x)	three-point smoothing of array
my.y	=	interp(old.x,old.y,my.x)	linear interpolation of arrays
my.y	=	qinterp(old.x,old.y,my.x)	quadratic interpolation of arrays
my.y	=	splint(old.x,old.y,my.x)	spline interpolation of arrays
my.y	=	rebin(old.x,old.y,my.x)	re-binning interpolation of arrays
my.z	=	lconvolve(my.x,my.y,sigma)	convolve data with Lorentzian
my.z	=	gconvolve(my.x,my.y,sigma)	convolve data with Gaussian
y	=	debye(temp,theta)	σ^2 in Debye approximation
y	=	eins(temp,theta)	σ^2 in Einstein approximation

3.5.4 Functions for Smoothing and Interpolating Data

The functions listed in Table 3 do need some more detailed explanation. Many of these functions are for smoothing and interpolating data. The `deriv()` function returns the derivative (in the finite-difference sense) of its argument (a single array). The `smooth()` function returns a three-point smoothing of its array argument. There are three functions for interpolation of arrays, `linterp()` `qinterp()`, do linear and quadratic interpolation, respectively, while `splint()` does a cubic spline interpolation. These routines map an x-y pair onto another set of x-values, with syntax

```
Ifeffit> new.y = linterp(old.x, old.y, new.x)
Ifeffit> new.y = qinterp(old.x, old.y, new.x)
Ifeffit> new.y = splint(old.x, old.y, new.x)
```

It may be a bit confusing to remember the order of arguments, but is important to keep straight. The idea is that you're moving `old.y` from `old.x` onto `new.x`.

The `rebin()` function works in a similar manner to these interpolation schemes, but will average input data onto the output x array. That is, if the input x-array is finer than the output x-ray (over any portion of the two arrays), the corresponding y-values will be *averaged* where it is appropriate to do so. This is especially useful for finely-spaced data, as QEXAFS or continuous scan data. Note that the `spline()` and `bkg_cl()` commands use this algorithm for all XAFS data when converting from $\mu(E)$ to $\chi(k)$.

As with other functions, even though the interpolation functions are intended for working with other arrays, they can be used with some of their arguments as scalars. This can be useful to estimate a single value of an array as a scalar as

```
Ifeffit> y = splint(old.x, old.y, x)
```

which will define `y` to be the scalar value of `old.y` at `old.x = x`.

The functions `lconvolve()` and `gconvolve()` provide ways to convolve a data (given by two arrays: $y(x)$ represented by `my.x` and `my.y` for x and y, respectively) with a Lorentzian or Gaussian function. For example,

```
Ifeffit>  my.y = lconvolve(my.x,my.y,sigma)
```

will convolve $y(x)$ with a Lorentzian function described by `sigma`. Similarly,

```
Ifeffit>  my.y = gconvolve(my.x,my.y,sigma)
```

will convolve $y(x)$ with a Gaussian function described by `sigma`. Convolution can be a slow process, show this should be used with caution, and special care should be given to whether `set()` or `def()` is used with these functions.

3.5.5 XAFS-specific Functions for σ^2

There are two intrinsic functions to calculate σ^2 for an XAFS path. The `debye(temp, theta)` function generates σ^2 for a path given a temperature `temp` and Debye Temperature `theta` using the correlated Debye Model implemented by Rehr, *et al.* in FEFF. Similarly, `eins(temp,theta)` calculates σ^2 for a path given a temperature `temp` and Einstein Temperature `theta` using the Einstein Model.

Of course, these two functions need more information than just external temperature and characteristic temperature. All of the other information depends on which XAFS path is being considered, and is found using the concept of the *Current Path*, further discussed in Section 7.1. Without repeating too much of that discussion here, the basic point is that IFEFFIT keeps track of which path it is considering with the variable `path_index`. Within the functions `ff2chi` and `feffit()`, this variable is looped through the paths you specified, and the path-dependent values like `reff` are automatically updated. Similarly, if a σ^2 Path Parameter depends on `debye()` or `eins()` (even indirectly), it will automatically be updated. From outside `ff2chi()` and `feffit()`, you can set `path_index` yourself, and then both `debye()` and `eins()` will use be evaluated for the specified path.

3.6 Commands, Arguments, and Keyword/Values

As mentioned in section 3.1, every operation you send to IFEFFIT is a command, and commands have the general syntax

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

Here, we'll give a more in-depth discussion of the functionality hinted at in section 3.1.

Generally speaking, arguments (including keyword/value pairs) can be in any order. The rules for dealing with keyword/value pairs are discussed below , but for I'll just say that the keyword itself determines what type of variable will be read, and that the equal sign (=) and comma (,) are the delimiters between the keyword and the value. The value is often treated as a character string, in which case it's often safest to enclose it in quotes.

Many commands have default keywords that are expected in the first few arguments. For example, `spline` has default keywords of `energy` and `xmu` for the first and second arguments, so that

```
Ifeffit> spline( my.x, my.y)
```

is shorthand for

```
Ifeffit> spline(energy= my.x, xmu= my.y)
```

Similarly, `plot()` has default keywords of `x` and `y` for the first two arguments, so that

```
Ifeffit> plot(my.x, my.y)
```

is the same as `plot(x = my.x, y = my.y)`. Let me clarify this. In this example, the first keyword is taken as `x` for `plot()` because `my.x` is not a keyword, and it is the first argument in the list. This is not to say that the first argument without a known keyword will be used as the `x`-array.

```
Ifeffit> plot(color = blue, my.x, y = my.y)     # No!
```

will not work. For several commands, there are keywords that don't really require a value. Such keywords are often used as *flags*, and so just giving the keyword is sufficient. For:

```
Ifeffit> pre_edge(my.energy, my.xmu, e0find)
```

the `e0find` will cause E_0 to be determined, even if it appears to be known already. Similarly, in

```
Ifeffit> plot(my.energy, my.bkg, nogrid, xmin = 8800)
```

the `nogrid` flag will draw the plot without the background grid. Actually, flags like '`nogrid`' don't really ignore their value – they interpret their value as either 'true' or 'false', and are true unless the first character of the value is `f`, `F`, `n`, `N`, or `0`. Specifically, a blank value means 'true', so that a value is not needed.

Some commands don't use keyword/value pairs for all their arguments, but use lists for some of their arguments. The notable examples of this are `write_data()`, `ff2chi()`, and `feffit()`. `write_data()` uses lists of arrays of text strings to write out to a file. Both `ff2chi()` and `feffit()` take a list of FEFF paths to use in the sum-over-paths. `write_data()` is somewhat more complicated, as it interprets a few arguments as keyword/values (notably, the `file` keyword), but expects lists of arrays and strings. Since `write_data()` can even take a glob '*' character, it is definitely the command with the least 'normal' syntax. It may be convenient to think of lists as values without keywords, but it's probably better to not think about it at all. Lists are used when they make sense, which turns out to not be very often.

As mentioned in section 2.2, commands take their input from both the argument list and the global Program Variables. More precisely, many commands read *default* values for command parameters from the global set of Program Variables. In all cases, there is a corresponding keyword (often using the same name as the Program Variable) that can be used to override the value read from the Program Variable. Typically, the Program Variables that are used as default inputs are also written to when the command is finished, so that subsequent executions of that command focus on what to change from the earlier execution.

This mechanism of having default values allows you to use a mixture of explicit arguments and "silent" Program Variables. For example, the `fftf()` command in the initial example took it's value for the k-weight from the Program Variable `kweight` which had been explicitly set a few a lines up. We could have also said

```
Iff> fftf(real = my.chi, kmin = 2.0, kmax = 13.0,
Iff>      dk = 1.0, kweight=2)
```

which is more explicit. Of course, there is plenty of room for confusion in this approach of mixing implicit and explicit parameters for a command. In general, the use of implicit parameters setting should be kept to a minimum. Though it means a bit more typing, you'll probably be happier explicitly specifying all command arguments you care about.

The general execution procedure for all commands is this:

1. read the needed parameters from the list of program variables. If a program variables has not yet been defined, it will be created, and initialized to 0 for numeric variables or blank for strings.

2. read the command parameters from the argument list. This effectively overrides the values read from the program variables.

3. perform its main tasks, possibly altering or creating some program variables.

4. update the output program variables to the new values.

The description in Chapter 9 for each command contains a complete list of the variables read as inputs, the variables set on output, as well as the complete list of keywords and the variables they correspond to.

The procedure listed above may a bit abstract, so an example will probably help. Let's investigate the `spline()` call in the example of the previous chapter in more detail. There, the deceptively simple command

```
Ifeffit> spline(energy = my.energy, xmu = my.xmu,
Ifeffit> rbkg=1.1, kweight=1., kmin=0)
```

was used. Looking ahead to section 9.41, you may notice that `spline()` actually needs the values for several scalar variables: `rbkg`, `e0`,`kmin_spl`, `kmax_spl`, `kweight_spl`, etc[5] to name a few. So how does `spline()` get its values for `rbkg`, `e0`, and `kmax_spl`?

First, `spline()` checks if there are already variables named `rbkg`, `e0`, etc. If these exist, their values are taken. Then `spline()` reads its argument list. Since the keyword `rbkg` is given in this example, that value is used in place of any `rbkg` variable already defined. Now, some values don't have defaults, but are required for `spline()`: the name of the energy and $\mu(E)$ arrays are required. If these aren't found in the argument list, `spline()` will complain and return without calculating anything.

Once `spline()` has determined all the input parameters, it calculates $\mu_0(E)$ based on all the scalar and array parameters. The method used is essentially the same algorithm as AUTOBK, and is described in detail in *XAFS Analysis with* IFEFFIT , but it's sort of secondary at this point. What's important here is that the values of some scalar and array parameters may be changed, or been created if they didn't exist already. For example, `e0` and `kmax_spl` may be changed if their values at the beginning of `spline()` were out of the data range. Because of this possibility, the last thing `spline()` does before returning is to reset the values of the Program Variables it has used. This ensures that both you and all the commands you run later will agree on the values of the program variables like `e0`.

Of course, each command has its own set of program variables that it will look for on input and reset on output. Chapter 9 lists the input and output variables for each command, as well as a bunch of other information about each command.

For the most part, command arguments are interpreted as keyword/value pairs. Keywords are single words (no whitespace), and are not case-sensitive. Generally, a keyword is followed by an equal sign, and the text between the equal sign, and up to the next comma or right parentheses[6] is taken as the value.

[5] Note that all the Fourier transform parameters for `spline()` are stored in variables that end with `_spl`.

[6] Actually, up to the first unprotected comma or right parentheses. A comma or parentheses can be protected by enclosing it in pairs of parentheses, double quotes, or curly braces.

The value is interpreted according to what *type* of value the keyword wants. That is, if the keyword wants an array, the value is read as an array, and if the keyword wants a text string (say, a filename or comment string) the value is read as a string. Most arguments that expect array or scalar values take *expressions* which could be a simple value (as in `e0 = 8979.5`), the name of existing variable (as in `energy = my.energy`) or something more complicated, like

```
Ifeffit> set   rbkg1 = 1.2
Ifeffit> spline(my.energy, "a.xmu+b.xmu", rbkg = rbkg1)
```

Though this may seem unnecessarily extravagant for command-line use, it can be very useful for more elaborate scripts. This feature is also especially convenient with the `plot()` command, as it avoids needing to define intermediate arrays just to plot them:

```
Ifeffit> newplot(my.energy, "my.xmu - my.bkg")
```

Note that when an expression is given, it is usually a good idea to enclose the expression in matched parentheses, double quotes, or curly braces.

3.7 Getting information back from IFEFFIT

At some point, you'll probably want to get information about Program Variables, Paths, and other aspects of IFEFFIT's state. The most basic way to do this is with the `show()` command.

The `show` command will print out information about the program variables, fitting variables, paths, macros, color table, and built-in commands. In it's simplest form, `show()` will print the value and, if appropriate the formula, for a program variable. For scalars, `show()` works like this:

```
Ifeffit> a = 8,   x  = 9 / a
Ifeffit> show a, x
  a             =       8.00000000
  x             =       1.12500000 := 9/a
```

Note that both the value and formula are shown for `x`. For text strings, the results are pretty straightforward – there's no stored formula, so the string is simply printed.

```
Ifeffit> $string  = ' My favorite string '
Ifeffit> show  $string
  My favorite string
```

For arrays, however, the results are a little different. Since it's unlikely you'll want to see every element of an array (and if you do, there's always `print()`), so the number of points, maximum, minimum values, and if appropriate the formula are printed:

```
Ifeffit> my.x = indarr(10)/3
Ifeffit> show my.x
  my.x   =   10 pts  [ 0.33333  : 3.3333 ]  := indarr(10)/3
```

To see the contents of a pre-defined macro, just tell the `show()` command the macro name:

```
Ifeffit> show make_ps
  macro make_ps ifeffit.ps  /cps
    "dump plot to a postscript file"
      plot(device=$2,  file= $1)
  end macro
```

The arguments to `show()` can be of mixed type – you can show scalars, strings, arrays, and macros with a single command. If a program variable is not known, an 'undefined variable' will be printed.

Beyond these simple examples, `show()` can also print out classes of program variables, using the special `@` symbol. That is, to get a listing of all the scalars (with values and definitions, if appropriate), type `show @scalars`. Similarly, `show @arrays` and `show @strings` will show all the arrays and strings, respectively. The listing shown for scalars and arrays will be sorted with "most constant" (*i.e.*, those values that were `set()`) at the top, and the "least constant" (*i.e.*, those that depend on other variables) at the bottom. This means the listing is subject to re-ordering at any time.

All the fitting variables can be shown with `show @variables`: both the current value (which would be the initial value before a fit and best-fit value after a fit) and estimated uncertainty will be shown. `show @colors` will print out the plotting color table (discussed in section 5.3). `show @macros` will print out all the macro names, with description and default arguments, but won't print out the full text for each macro. You can say `show @macro = make_ps` or just `show make_ps` to see the full contents of a macro.

For XAFS paths defined with the `path()` command, `show @paths` will list the values of all the path parameters for all paths. To get a listing for a limited selection of paths, `show @path = 1` or `show @path = 1,2,5` will show just those paths.

In addition to the `show()` command, you can also print out messages with the `echo()` and `print()` commands. `echo()` simply prints its argument, which is of limited utility at the command line, but is often helpful in more complicated scripts:

```
Ifeffit> echo(" I am in macro BKG, about to write outputs")
 I am in macro BKG, about to write outputs
Ifeffit> echo " t"
 t
```

`print()`, on the other hand, is a more general purpose printing command, interpreting its arguments as strings or mathematical expressions where appropriate, and printing out the resulting *values*. For example,

```
Ifeffit> print pi
3.141593
```

You can print out more than one value at a time, and even print out the value of expressions, making `print()` act like a simple calculator:

```
Ifeffit> var = 100.
Ifeffit> $string = 'This is a string'
Ifeffit> print pi/2  sqrt(5*var)  $string
    1.570796    22.36068 This is a string
```

`print()` does a mediocre job of parsing, so it is best to enclose expressions in quotes or double quotes. Enclosing in single and double quotes have different results, though. Double quotes cause evaluation, while single quotes *prevents* evaluation, so the string is written out literally. That is, you can say something like this.

```
Ifeffit>print 'x , sqrt(x) = ' ," x ", " sqrt(x) / 2 "
   x , sqrt(x) =    10.22000    1.598437
Ifeffit>print ' Rbkg  = ',  rbkg, ' Ang '
   Rbkg  =    1.2000000  Ang
```

Table 4: Table of Values for `&screen_echo`

Value	Meaning
0	save message to echo buffer.
1	print message to screen.
2	print message to log file, if open.
3	print message to both screen and to log file.

Finally, unlike `show()`, `print()` will print out all elements of an array. I won't give an example of that – you can try it for yourself.

3.8 Log Files, `echo`, `show`, and `print`

In the previous section we saw how `echo()`, `show()` and `print()` were used to get various information from IFEFFIT printed to the screen during an interactive session. Sometimes, you don't care to have such information printed to the screen, but would like to write it directly to a file. Furthermore, when controlling IFEFFIT from an external program or script, you may want to save the messages that would've been written to the screen and read them into the calling program. All of these things are possible within IFEFFIT and in this section.

IFEFFIT sends all its output messages through a single routine that either writes the message to the screen (technically, standard output) or to an external log file, or saves the message into a buffer that a calling program can read later. Which of these actions is taken depends on the scalar variable `&screen_echo` and whether a log file is actually open and in use. The valid values and meanings for `&screen_echo` are given in Table 4. When IFEFFIT starts, `&screen_echo` is 1, even when run from an external program, so to set up an external program to 'capture the echo buffer', you need to set `&screen_echo` to 0. If doing this, it's probably a good idea to check the echo buffer after each command. Examples of doing this are available.

For normal interactive use, the default `&screen_echo` is probably appropriate until you want to write some of the information from IFEFFIT to a log file. For that, the `log()` command is exactly what you need. Using `log()` is simple:

```
Ifeffit> log(file = my.log)
```

will close any existing log file (there can be only one log file at a time), and open *my.log* as the current log file. It will also set `&screen_echo` to 2, so that any subsequent outputs from IFEFFIT, including from the `show()`, `print()`, and `echo()` commands are sent to this file instead of the screen. Because of system-specific constraints, you can't rely on the log file being completely full until it's closed, which you can do explicitly with

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

Though it's not completely necessary, you can also tell `log()` the level of `&screen_echo` to use as well:

```
Ifeffit> log(file = my.log, screen_echo = 3)
Ifeffit> log(file = my.log, screen_echo = both)
Ifeffit> log(file = my.log, screen_echo = tee)
```

will all tell `log()` to write all text output to both the log file and the screen. You can also change the value of `&screen_echo` yourself while a log file is open.

Note that because a script or program may use the echo buffer for its own purposes, it may not be wise to overwrite the value of `&screen_echo` from *within* an IFEFFIT application. For example, most of the other GUI applications (including G.I.FEFFIT, ATHENA, and ARTEMIS) set `&screen_echo` to 0 and then intercept the output to save in a history buffer or display to some information screen. This means that you don't need to set `&screen_echo` yourself, and that bad things may happen if you do! On the other hand, getting a history of the input or output from these programs is usually a matter of cutting-and-pasting from the displayed output.

4 Input and Output Files

At some point, you'll want to read your data into IFEFFIT. Getting data in the expected format is quite possibly the hardest part of dealing with any data analysis program. Currently, IFEFFIT uses plain text files with data in columns delimited by white space (blanks and/or tab characters). There is also some support for a IFEFFIT-specific file format for storing data to be read back into subsequent IFEFFIT sessions. The 'old-style' UWXAFS RDF binary files are not currently supported.

IFEFFIT expects data to be in plain text (also known as ASCII) files that have some lines of descriptive text followed by some numerical arrays stored in space- or tab-delimited columns. The column-based data files have the feature that only one set of related data can be stored in a given file. In particular, all arrays in an ASCII column file must have the same number of data points. This is not usually a serious problem, but it is something to keep in mind. Another thing to keep in mind is that IFEFFIT does not yet have any special procedures to support beamline-specific data.

The commands for dealing with column files are `read_data()` and `write_data()`. These commands are discussed in this chapter and in sections 9.34 and 9.45.

4.1 Reading ASCII Column Files

When reading data files with `read_data()`, IFEFFIT needs to assign array names to the arrays read from the file. Deciding the arrays names is the only tricky part to `read_data()`. Since all arrays in IFEFFIT have two-part names, with a "group name" for the suffix, and since data in any data file are usually meant to be grouped together, IFEFFIT will use a common group name for all arrays read from a given file. The group name used can be specified explicitly with the `group` keyword to the `read_data()` command. If not specified, the group name will be automatically set based on the file name itself.

Before we get on to the details of naming the arrays, let's discuss the title lines in the file. As said above, these lines contain descriptive text about the data, and not the numerical data itself. Normally, IFEFFIT will try to figure out where the descriptive text ends and the numeric data begins. One simple way to ensure that this is done correctly is to put some non-numeric character in the first column of each line of the header. '#', '%', '*', '!', and ';' are popular choices for such "comment characters", and IFEFFIT will respect them all. Another possibility is to specify the number of title lines explicitly, with the `title_lines` keyword, as in

```
read_data(file=my.chi, title_lines = 3)
```

It is generally easier to arrange for files to have title lines that always begin with some comment character than to have to count the number of title lines for each file.

The first 64 title lines will be stored in strings named according to the "group name" with names like `$GROUP_title_II`. That is, when reading a file with group name "`xfile`", the first comment line will be save in `$xfile_title_01`, and the twelfth in `$xfile_title_12`.

OK, on to the naming of the arrays. There are several ways to specify how the suffixes of the array names will be. The sheer number of options may seem unnecessary at this point, but after using IFEFFIT for a while, you will probably end up using several of these methods depending on the data file you're using. The simple way to name the arrays from a file is to make the suffix of each array name be the integer for that column, by specify `type=raw` in the `read_data()` command. That is,

```
read_data(file=myfile.dat, group = A, type = raw)
```

Table 5: Table of Known Data Types for `read_data()`. Note that if there are more columns in the file, the subsequent arrays will be named by the column index.

Data Type	Array suffixes
xmu	energy, xmu
chi	k, chi
rsp	r, chir_re, chir_im, chir_mag, chir_pha
qsp	q, chiq_re, chiq_im, chiq_mag, chiq_pha
chi_std	k_std, chi_std
xmu.dat	energy, e_wrt0, k, mu, mu0, chi
chi.dat	k, chi, mag, phase
feff.dat	k, cphase, mag, phase, redfactor, lambda, realp

will create arrays `A.1`, `A.2`, `A.3`, and so on. The "raw" names aren't very mnemonic, but they're simple and and very predictable. For analyzing several similar files, you could read in the data using "`type = raw`" and write a macro to rename the "raw" column arrays. For more on macros, see chapter 10.

Often times data comes in fairly standard file types, in which the columns have known arrays. So the second way to name arrays in IFEFFIT is to specify one of several known "type" to `read_data`. Thus,

```
read_data(file=my.chi, group = A, type = chi)
```

will name the array from the first column `A.k`, and that from the second column `A.chi`. Any remaining columns will be `A.3`, and so on. Similarly,

```
read_data(file=cu.xmu,  type = xmu)
```

will name the array from the first column `cu.energy`, and that from the second column `cu.xmu`. (Note here that "`group`" was not specified, and so was taken from the file name itself). Table 5 lists all the recognized file types and the associated column names.

Another common situation is for files to come with column labels already provided in the file. In such a case, the `type = label` can be used to *read* the column labels from the file itself, provided the file has been formatted with this in mind. By that I mean 1) the file has text strings at the top of the file, before the column data, 2) that the next to last text line is a line of minus signs (the important thing is that the third through eight character on the line are minus signs), and 3) the last text line is a label line. Such a file would look like this

```
# Cu XAFS
# data file containing xmu(energy)
#----------------------------
#  energy  xmu    i0
 8760.02   1.313982  60351.3
 8770.01   1.323154  59808.3
 8779.98   1.332213  59290.3
 8790.02   1.344031  58709.3
 8799.98   1.352667  58245.3
 8810.02   1.364248  57719.3
 8819.99   1.375764  57165.3
 8829.98   1.384695  56690.3
```

and be said to have column labels "energy", "xmu", and "i0". All the files distributed with IFEFFIT have columns labeled in this way and IFEFFIT normally write out files with such labels. To read this file, and have the array be named by the given labels, you would say

```
read_data(file=my.xmu, group = A, type = label)
```

which will create the arrays `A.energy`, `A.xmu`, and `A.i0`. There are some minor issues when column labels contain characters ("()./," and so forth) which can't be in the suffix part of an array name, in which case the arrays names will end up a little mangled – usually "_" will be used instead of the offending character.

When all else fails, you'll just want to specify the array names yourself. Keeping in my the previous method, the preferred way to specify the names is with the `label` keyword. This overrides all the above methods, and can be used to override default column names from the file itself. The argument to `label` is a space delimited string with the array suffixes as if it had been the label string in the file. Thus, using

```
read_data(file=my.xmu, group = A, label = 'x y z')
```

would create arrays `A.x`, `A.y`, and `A.z`, even if the arrays are labeled something else in the file.

A note of caution: there is nothing preventing two columns from having the same label. This is currently an unsolved 'feature' that may be fixed in the future.

As we have seen, there are several options for how to name arrays read in with `read_data()`. So, which one is the default if neither `type` or `label` are given, or if both are given? The answer is this: First the string from the `label` keyword is used. If `label` is not given, then the type specified by the `type` keyword is used to generate the array names (with results shown in Table 5). If neither of the `label` or `type` keywords are given, the existing "label line" from the file is used. Finally, if there is no label line in the file, the column indices (*i.e.* `type=raw`) are used.

Given these complicated rules, it might be nice to know for sure what arrays were actually read in. This may not seem so important when using IFEFFIT at the command-line, because you can always do a `show @arrays` or even `show @group=A` and figure it out. But if you're writing a *script*, this can be a serious issue. In all cases, `read_data()` will set the string `$column_label` to a space-delimited string of the array *suffixes* just read in, as if this had been the label string in the file.

4.2 Sorting Data with `read_data()`

In general, IFEFFIT expects data to be well-ordered. For $\mu(E)$ data, it generally expects the data to be in strictly increasing order of energy – and without repeated energy values. Unfortunately, not all data comes like this. Notably, XAFS data collected in "continuous scan mode" or "Quick-EXAFS" is rarely guaranteed to be in strictly increasing order. In addition, data is not always perfectly ordered for some beamlines using encoder-readback for monochromator position even in step-scan mode.

To overcome problems resulting from poorly-sorted data, `read_data()` allows you specify a column *number* to put into strictly non-decreasing order (that is, increasing order but with repeated points retained) and to sort the other columns accordingly. To do this, you would use the "sort" keyword,

```
read_data(file=sro_xafs.dat, group = sro, sort=1)
```

Note that the column number is given, not the column name. The reason for this is that the actual column names are not necessarily known prior to running `read_data()`, and confusion could easily occur between similarly named columns. By default, the data will not be sorted. You can ensure that sorting will not be done by using `read_data(...,no_sort=1,...)`.

The `spline()`, `pre_edge()`, and `bkg_cl()` commands will internally sort $\mu(E)$ data into strictly increasing order (averaging repeated energy points) to avoid such problems. Still, if you may have poorly-sorted data, it is recommended that you sort it with `read_data()` before doing any processing on it.

4.3 Writing ASCII Column Files

To write data with IFEFFIT, you use the `write_data()` command, specifying the name of the output file, and listing the names of the text strings, scalars, and arrays to write to this file. Text strings are written at the top of file, one per line, in the order you specify. The "comment character" is written at the beginning of each of the lines of text (in the first non-blank row). To specify the comment character, either set the Program Variable `$commentchar` or set the `commentchar` keyword to `write_data`.

After the string variables are written, scalars are written, one per line, in the form "¡comment character¿ ¡scalar name¿ = ¡scalar value¿". After the scalars, a line of minus signs is written, and the a label line (suitable for later reading with `read_data(..., type = label)`) is written containing all the array *suffixes*. Both the line of minus signs and the label line will still have the "comment character" at the front of the line). Finally, the arrays are written out in columns, in the order (left-to-right) listed.

`write_data()` will write the same number of points for all arrays, even if the Program Variables have different number of points. In fact, the *minimum* number of points will be written. At this writing, `write_data()` is limited to writing out 64 columns per file.

It's common to want to write out *all* the strings named `$GROUP_title_II` to a file, so as to preserve the comments from the starting file(s). Typing each name individually would be painful, so `write_data()` supports the usual "*" glob character to mean "all that match". So putting `$GROUP_title_*` in the list would cause all strings matching that name to be written. You can use a "*" in this way to match any string names. You could even use it to match array names using something like `*.xmu` or `a.*`, but since the number of points could be different for the different arrays, this should be used with caution.

4.4 IFEFFIT PAD Format for Save and Restore Files

Though ASCII column files are often the most convenient form to use, they are somewhat limited for large amounts of data. The least desirable features of ASCII files are that they take too much space, that they can't store arrays of different sizes, and that they can't hold many arrays. All of these limitations can be overcome by using the PAD (that stands for Packed ASCII Data) file format specially designed for FEFF and IFEFFIT. PAD files are fully portable plain-text files that can store an unlimited number of arrays of different sizes and can be transferred to (and read on!) any machine. The PAD format stores approximately 12 significant digits for each numerical value.

Of course, there are drawbacks to the PAD file system. These are 1) a slight increase in time to read the file compared to regular binary (but much faster than a set of plain ASCII files!!), and 2) these files cannot be used directly in any other program. The second point is a serious problem. Like the UWXAFS RDF files that they're intended to replace, the PAD format is a

home-built file storage system. Moving to one of the accepted scientific data formats such as netCDF may be necessary in the future.

PAD files are especially suited for IFEFFIT because they can store and mix all types of Program Variables. This makes them ideal for saving a full set of IFEFFIT variables. The `save()` command (section 9.38) uses the PAD format to save the current state of IFEFFIT into a save file that can be then read in with `restore()` (section 9.37) to re-create a previous IFEFFIT session. Since all Program Variables are saved and since the PAD files are saved in printable ASCII characters, this `save()`/`restore()` mechanism makes a completely portable way to preserve an IFEFFIT session into a single file for later inspection or to share with a colleague.

5 Plotting with IFEFFIT

Graphical displays of data are essential to data analysis. IFEFFIT uses PGPLOT, a simple graphics library that is fairly well-supported and portable, and can be accessed from fortran, C, and a variety of scripting languages. PGPLOT supports many graphics devices (terminals, graphics files, hardcopies), and works well on Unix, MacOS X using X Windows, and Windows systems. Much of the information in this chapter is adapted from the PGPLOT documentation. That documentation is aimed at the programmer and not the end user, but the concepts discussed are fairly simple. If you have a question about IFEFFIT graphics, you may wish to consult the PGPLOT documentation, which can be found at http://www.astro.caltech.edu/~tjp/pgplot/.

Plotting in IFEFFIT is encapsulated in the commands `plot()`, `newplot()`, `cursor()`, `zoom()`, `color()`, `plot_text()`, `plot_marker()`, and `plot_arrow()`. These commands make respectable looking plots on screen and paper, and provide good flexibility for graphical data analysis. These routines allow different colors and linestyles for each trace on the plot, They allow other symbols, text strings, and arrows anywhere on the plot window, and allow a 'legend' of plotted trace to be easily and automatically built. They also allow you to use the cursor to get x-y positions of particular points on the plot window and to zoom in on particular areas of the plot window.

The hardcopies made by PGPLOT are of reasonable quality, but may not satisfy your criteria of publication quality. Since there are many programs designed especially for these purposes, and since IFEFFIT is intended to be an XAFS analysis program, not a high-quality graphics program or a data visualization tool, the quality of the resulting graphics seems acceptable.

The `plot()` command is the main plotting command in IFEFFIT, making a two-dimensional line-plot or scatter-plot given x and y arrays. There are many optional arguments, most of which will be discussed in this chapter. A complete list is given in section 9.27. The `newplot()` command is a minor variation on `plot()`, that will always erase the current plot before plotting. The rest of this chapter will discuss the details of the various plotting options available in IFEFFIT.

5.1 Specifying Data for Plotting

The `plot()` command plots arrays of data. Typically, you you will specify array names for both the x- and y-arrays, as in

```
Ifeffit> plot(x= my.x, y= my.y)
```

Of course there are many other optional keywords you can give to `plot()`, but here we're just focusing on the `x` and `y` arguments because they are the most important. As you use IFEFFIT, you'll find that a simple `plot()` command such as the one above gets used a lot. Because of this, even this simple `plot()` command can be made easier, more flexible, and more powerful, as will be discussed here.

First, the `plot()` command uses the concept of positional keywords, so that you can drop the `x=` and `y=`, and just use

```
Ifeffit> plot(my.x,my.y)
```

that is, the first argument in the list (i.e., everything between '(' and the the first comma) without an explicit keyword is assumed to be the value for `x`, and the second argument without a keyword is assumed to be the value for `y`. Since a command that fits on a single line can drop the parentheses, this can become

```
Ifeffit> plot my.x, my.y
```

Much, or perhaps most, of the data you'll want to plot will be in a single group. If the `x` and `y` arrays you intend to plot are in the same group, you can specify the group once, and only give the suffixes of the array names:

```
Ifeffit> plot x, y, group=my
```

In fact, if the string variable `$group` is set to the group you want (and it often is set to the "current group" after most data processing commands, and the `plot()` command itself), this group name will be used by default:

```
Ifeffit> $group = my
Ifeffit> plot x, y
Ifeffit> plot x, z
```

which greatly simplifies the task of over-plotting data from the same group.

Since there are two `plot()` commands in the above example, this is a good time to discuss when the plot window gets cleared for a fresh plot. Normally, the `plot()` command adds the specified trace to the existing plot: over-plotting is the normal behavior. To clear the plot, you can use

```
Ifeffit> newplot (my.x, my.z)
```

which will erase the previous plot and start over. You can simply send an empty `newplot()` command and subsequent `plot()` commands will add to this refreshed plot window.

In many cases, you'll want to plot more than one `y` array as a function of the same `x` array. This leads to a slight variation on the rule for positional keywords. I said above that the first argument without a keyword is taken for `x` and the second for `y`, but the full truth is a little more complicated. If exactly one argument has no keyword, and there is no explicitly set `y=...` argument, then the argument without a keyword is actually taken as the y array. In this case, the `x` array is the *previous* `x` array. That is, after

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

these two commands will produce the same effect:

```
Ifeffit> plot my.energy, my.bkg
Ifeffit> plot bkg
```

If you give only one array for the first trace to plot, a simple index array (i.e, 1,2,3,4,...) will be used for `x`, so that

```
Ifeffit> newplot my.energy
```

would be equivalent to

```
Ifeffit> my.tmpx = indarr( npts(my.energy))
Ifeffit> newplot my.tmpx, my.energy
```

That covers how to make the `plot()` command simpler, now let's look at how to make it more complicated. As the last example hints at, sometimes it's desirable to plot arrays that do not yet exist as named arrays in IFEFFIT. For example, to view $\chi(E) = \mu(E) - \mu_0(E)$ after a background subtraction from `spline()`, you might say

```
Ifeffit> cu.chie = cu.xmu - cu.bkg
Ifeffit> plot(x= cu.energy, y= cu.chie)
```

or use one of the simplified forms discussed above. But if you only want to view $\chi(E)$, not do any processing with it, it seems unnecessary to create it. So, in fact you can pass a simple expression as the `y` (or `x`) argument to `plot()`:

```
Ifeffit> plot(x= cu.energy, y=cu.xmu-cu.bkg)
```

In fact, the expression can include any of the data and any of the functions described in section 3.5. This provides a flexible way to plot a variety of functions, but there are some caveats to using expressions instead of existing arrays. First, to be sure the expression is parsed correctly, you should put it in quotes:

```
Ifeffit> plot(cu.energy, y="(cu.xmu - cu.bkg)*(cu.energy-e0)")
```

This is especially true if the value in the keyword/value pair contains spaces: use quotes or you're likely to get several warnings and weird results. Also, when using an expression instead of an existing array, you should not rely on `x` and `y` being default positional keywords, and explicitly use the `y=` as shown above.

5.2 Error Bars

If you have experimental data for which you have estimates of the uncertainties in the data, it is often useful to display these uncertainties using error bars. This can be done using the keywords `dy` and `dx` which name arrays for the point-by-point uncertainties in the `y` and `x` arrays, respectively. That is, if `dat.sigma` is an array of uncertainties in `dat.y`,

```
Ifeffit> plot(dat.x, y=dat.y, dy=dat.sigma)
```

will add vertical error bars extending from `dat.y-dat.sigma` to `dat.y+dat.sigma`.

Similarly, though it somewhat less usual for EXAFS data, if you have an estimate for the uncertainties in the ordinate array, these can be displayed using

```
Ifeffit> plot(dat.x, y=dat.y, dx=dat.delx)
```

will add horizontal error bars extending from `dat.x-dat.delx` to `dat.x+dat.delx`.

Though we'll discuss how to select plot colors and line styles in the next section, the error-bar plots generated with these `plot()` commands will use a single color and style for the main trace and error bars. If you'd like to see the error bars alone, or have thee error bars in a different color from the main trace, the you should use `style=points1` when plotting the error bars:

```
Ifeffit> plot(dat.x, y=dat.y, color=red)
Ifeffit> plot(dat.x, y=dat.y, dy=dat.sigma,
                  color=black, style=points1)
```

5.3 Colors, Line Styles, and Other Attributes

IFEFFIT allows different colors for each trace as well as the background, the foreground (text strings and the box around the plot), and the optionally displayed grid. It also supports different styles of lines (solid, dashed, points, etc.) to be used for different x-y pairs. This section will explain how to use these options in detail.

There are a few different ways to specify colors for x-y traces, background, foreground, and optional grid. First and most simply, you can specify the color for each object directly by name:

```
Ifeffit> plot(my.x,my.y, color=red, bg=white)
```

which will draw the x-y trace in red on a white background. This is probably the easiest way to get the colors you want. The available set of named colors is determined when the PGPLOT routines are installed, and are listed in the file *rgb.txt* in the PGPLOT directory. This file is usually very close to the standard X-Windows set of colors, so you can specify `dodgerblue3` and `bisque`, but not `teal`. For other colors, you can specify the red-green-blue intensities in hexadecimal format, using the somewhat standard '#RRGGBB' format:

```
Ifeffit> plot(my.x,my.y,color='#FF00FF')
```

will be magenta, for example.

If colors are not specified in the `plot()` command, the default values used will be taken from an internal "color table". The color table lists the colors used for background, foreground, grid, and then the traces in order that they are drawn, and can be displayed with either `color(show)` or `show @colors`. A typical output would look like this:

```
Ifeffit> show @colors
plot color table:
   bg   = white
   fg   = black
   grid = #CCBEE0
      1 = blue
      2 = red
      3 = darkgreen
      4 = black
  ⋮
```

which means the first trace drawn would be in blue, the second in red, and so on. You can change the values in the color table using the `color()` command, like this:

```
Ifeffit> color(fg = black, bg = white, grid = '\#AABB99')
Ifeffit> color(1 = yellow, 2 = cyan)
Ifeffit> color(3 = white, 4 = magenta)
```

which will reset the default colors to plot with. Such commands may be placed in a start-up file *.ifeffit*. For postscript output intended for a printer please note that a black background will use a lot of ink – a white background is recommended for hardcopies.

There are a few different line styles available as values for the `style` keyword. The supported line styles are given in Table 6 with examples shown in Figure 1. There is a fair selection of points available for the `points` and `linespoints` styles, which are specified by integer

Table 6: Supported Plotting line styles for the `plot()` command. These can be specified with `plot(..., style=lines)` and so forth. The available point types are show in Figure 2.

Line Style	Description
solid	solid line
dashed	dashed line
dotted	dotted line
dot-dash	mixed dot-dashed line
points	a special marker at each point (see Figure 2)
linespoints	solid line with a special markerd at each point

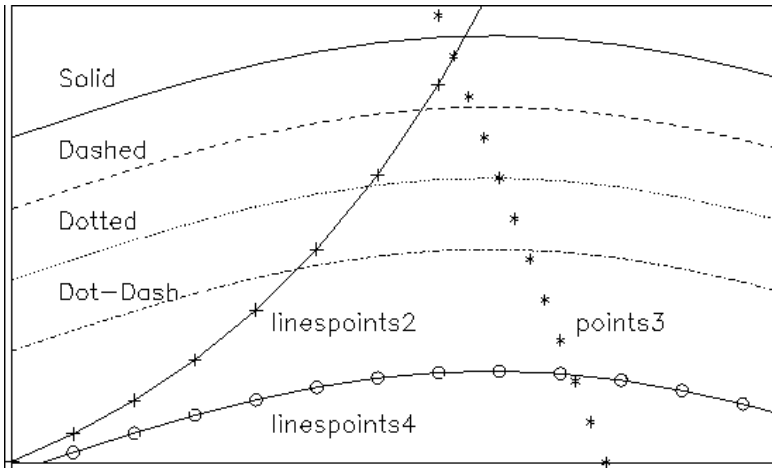

Figure 1: A selection of plotting line and points styles for IFEFFIT.

with `points1`, `points2`, `linespoints12`, etc, with examples of the available point types shown in Figure 2.

The width of the lines for all traces and axes can be set with the `linewidth` keyword, which takes an integer value. Values between 1 to 5 are appropriate, and the default is 2.

5.4 Text Strings and Labels

Text strings can be put on a plot to labels axes, give a title, as a legend for each trace, and to put text at user-specified coordinates. There is some control over the character size and font used, and there is a primitive syntax for non-standard characters that allows Greek letters, the Ångstrom symbol, subscripts, and superscripts. Like many aspects of PGPLOT, the possibilities are limited, but include a reasonable set of functionality needed for most applications. The full list of `plot()` keywords affecting the placement of text strings on the plot window are given in Table 7. In addition to the `plot()` command, the `plot_text()` command can be used to put text strings at selected x-y coordinates.

The keywords `xlabel` and `ylabel` set the labels for the x-axis and y-axis, respectively, while `title` will set the title above the drawing box. Each x-y trace can have a legend that will appear along the right side of the plot, with a short version of the plot line style and a "legend key" – a short text string that you can set with the `key` keyword, as in

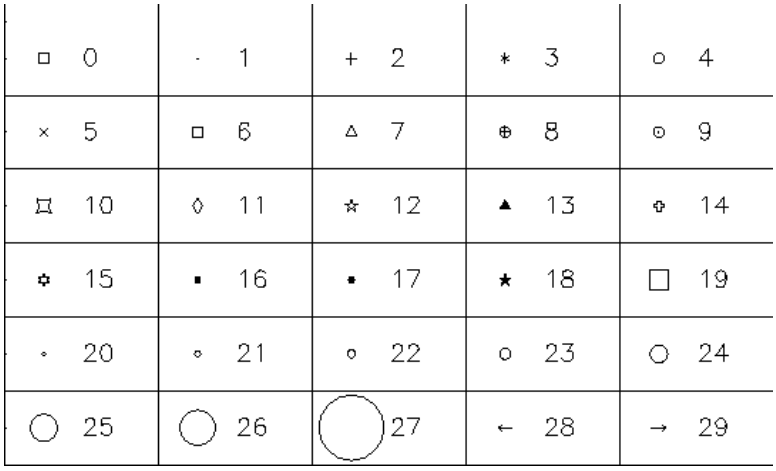

Figure 2: The plotting point types for IFEFFIT, as produced with syntax such as `style=points3`, and so forth. The linespoints types such as `style=linespoints3` will show the same point type, joined with a solid line.

```
Ifeffit> plot(dat.r,dat.chir_mag, color=red, key='data')
Ifeffit> plot(fit.r,fit.chir_mag, color=red, key='fit')
```

Text strings can also be put anywhere on the plot using either the `text` keyword, which will put a label at a coordinates specified by the keywords `text_x` and `text_y` or by the `plot_text()` command.[7]

You can put on multiple labels (up to 32) on a plot, but once on they can only be erased by making a new plot, which will erase all the labels.

```
Ifeffit> plot( text='300 K Data', text_x=7025, text_y=0.2)
```

or

```
Ifeffit> plot_text(x=7025,y=0.3, text='400 K Data')
Ifeffit> plot_text( 7025,   0.5,  '500 K Data')
```

As you can see, the advantage of the `plot_text()` variation is that it has default positional keywords for `x`, `y`, and `text` which greatly simplifies the syntax. Such labels are often useful when coupled with arrows, discussed in section 5.5.

The sizes of the various text strings and plot markers can each be set separately. The text size for the axis, x- and y-labels, and title can be set with the keyword `labelsize`. The text size for legend keys and explicitly-placed text strings are set with the keyword `textsize`. The size of the point markers for point and linespoints line styles are set with the `markersize` keyword. The keyword `charsize` will set all of the sizes to the same value. The value taken is a real number, with appropriate values usually in the range of 1.0 to 3.0. To completely suppress the axis text, you might be tempted to set `labelsize` to zero. This can cause odd results, so using a very small value such as

```
Ifeffit> plot(dat.x, dat.y, labelsize=0.0001)
```

[7] These two variations of `plot(text='300 K Data', text_x=7025, text_y=0.2)` and `plot_text(x=7025,y=0.2,text='300 K Data')` really are equivalent and can be mixed.

Table 7: Arguments for the `plot()` command for putting text strings on the plot window.

Keyword	Meaning
`xlabel`	the label of the x-axis
`ylabel`	the label of the y-axis
`title`	plot title along top
`key`	keyword describing each trace for legend
`charfont`	integer to select font for text
`charsize`	character size for all characters and markers
`labelsize`	character size of axis labels and numbers
`textsize`	character size of text strings and legend keys
`markersize`	size of plotting point markers
`text`	text string for a general plot label
`text_x`	x-coordinate for this text string
`text_y`	y-coordinate for this text string

will effectively suppress the axes from being drawn.

The syntax of the text strings themselves is almost straightforward. The PGPLOT documentation gives a more complete description, but I'll outline the main points here. Most text is displayed as typed, of course. You can also use "control sequences" to control formatting and special characters.

PGPLOT uses Hershey (vector) fonts which are easy to render when rotated but look slightly less than ideal. You can specify one of four fonts as the default font with the `charfont` keyword, which takes an integer 1–4 as its value. `Charfont =1` is the normal typeface – a sans serif font. `Charfont = 2` gives a roman font, `Charfont = 3` gives an italic font, and `Charfont = 4` gives a script font which is pretty difficult to read. You can mix these fonts in a text string with the an escape sequence.

```
\fn The Sans Serif Font    \fi The Italic Font
\fr The Roman Font     \fs The Script Font
```

which will render fonts as shown in Figure 3.

To get Greek characters, escape sequences starting with $\backslash g$ are used: $\backslash gm$ give μ, for example. Subscripts are done with $\backslash d$ and superscripts with $\backslash u$. There are no sub-subscripts, super-superscripts, sub-superscripts, etc. The Ångstrom symbol is $\backslash A$. Examples of these, and some string sequences to get common symbols for XAFS analysis are shown in Figure 3, and are provided in the examples distributed with IFEFFIT.

5.5 Markers and Arrows

The plot points as shown in Figure 2 can be put at any location on the plot window as *markers*. This is done with the o`plot_marker()` command

```
Ifeffit> plot_marker(x=7000,y=4,marker=1)
Ifeffit> plot_marker(7000,2,3)
```

where the value of `marker` gives the integer from Figure 2 for the symbol to use. Like the `plot_text()` command, this command can use default positional keywords for `x`, `y`, and `marker`, greatly simplifying the syntax.

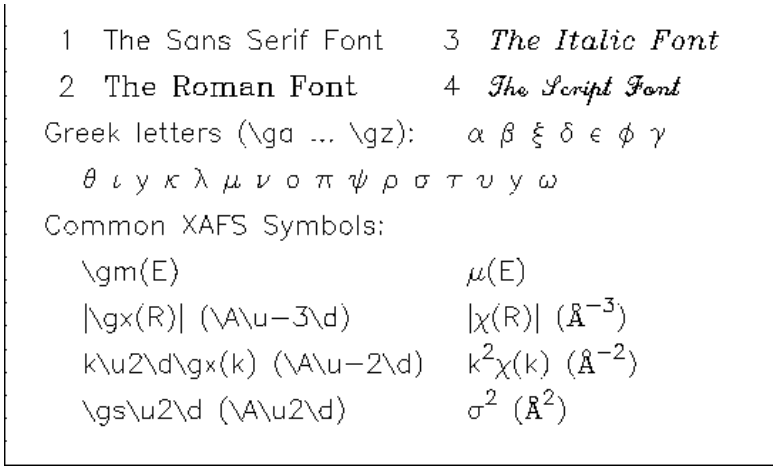

Figure 3: Sample of plotting text strings, fonts, and special characters for IFEFFIT

Arrows and lines can be put anywhere on the plot window, pointing to some spectral feature of the plotted data. This is done with the `plot_arrow()` command, which takes beginning and end points, and parameters describing how draw the arrow head (including whether to have no arrow head at all). An arrow can be placed like this:

```
Ifeffit> plot_arrow(x1=7000,y1=4, x2=7050,y2=3)
```

More complex control over the shape of the arrowhead can be obtained with the keywords `size`, `angle`, and `barb`. The `size` keyword alters the size of the arrowhead, while `angle` gives the angle subtended by the point in degrees. The `barb` keyword controls the shape and concavity of the arrowhead. Examples, with reasonable values for the `size` and `barb` parameters include

```
Ifeffit> plot_arrow(x1=10, y1= 4, x2=25, y2=4,  barb=2)
Ifeffit> plot_arrow(x1=65, y1= 4, x2=90, y2=4,  barb=0,outline=1)
Ifeffit> plot_arrow(x1=65, y1= 0, x2=90, y2=0,  angle=100,barb=0)
Ifeffit> plot_arrow(x1=65, y1=-4, x2=90, y2=-4, size=5)
Ifeffit> plot_arrow(x1=30, y1= 8, x2=90, y2=8,  no_head=1)
```

The result of these and other `plot_arrow()` commands are shown in Figure 4. The arrowhead can be made hollow by setting `outline=1`. To get a line between points (x1,y1) and (x2,y2), the `plot_arrow()` command is used with `no_head=1`, which completely suppresses the arrowhead.

5.6 Cursor and Zooming

For interactive data analysis, it is often desirable to get the x-y coordinates of some particular point on a plot, or to zoom in on a particular region of the plot window. The `cursor()` command allows you get the coordinates of a point on the plot window by clicking on it with the mouse. The x- and y-coordinates are then stored in the IFEFFIT variables `cursor_x` and `cursor_y`. These values can be written to the screen immediately by using the `show` keyword:

```
Ifeffit> cursor(show)
```

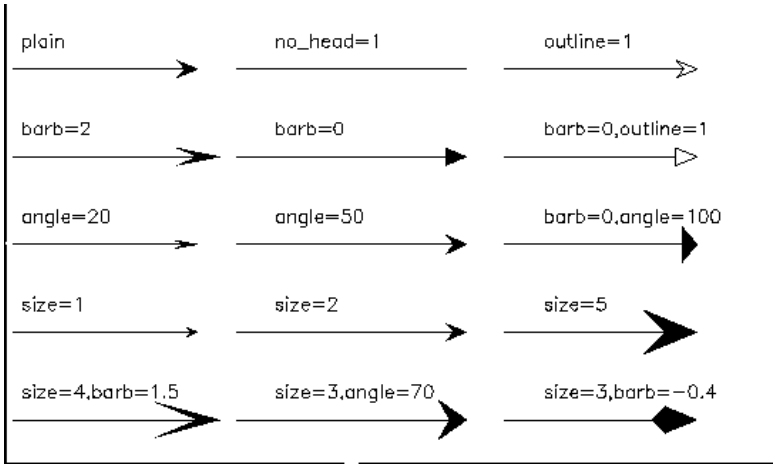

Figure 4: A selection of examples of plotting arrow parameters for IFEFFIT.

This will wait for you to click on the plot window, then print out the result in the form

```
cursor: x =     7.78890    , y =    0.963039
```

It is important to remember that the `cursor()` command blocks all other processing until you click on the plot window. For one thing, this means that putting `cursor()` in a macro should be done with caution.

By default, `cursor()` will show a small cross at the cursor point, but it is possible to change the look of the cross-hair shown while selecting the cursor point. For example,

```
Ifeffit> cursor(crosshair)
```

will show a moving crosshair that extends the full height and width of the plot window. There are a few other variations on the look of the cursor when selecting points. For example, `cursor(vert)` will show just a vertical line that spans the entire length of the plot window, and follows the cursor as you move the mouse. Similarly, `cursor(horiz)` will show a horizontal line that extends over the entire plot window and moves with the cursor position.

For selecting x-ranges (say, for choosing an energy range for the pre-edge region), it is often useful to see the previously chosen cursor position as you select the next one. This can be accomplished with `cursor(xrange)` which will show a stationary vertical line at the previously selected cursor point (or, if you explicitly set `cursor_x` yourself, that value will be used) and also show a vertical line that moves with the mouse. Similarly, for selecting a y-range, `cursor(yrange)` will draw a stationary horizontal line at the value set by `cursor_y` as well as a horizontal line that moves with the mouse.

The `zoom()` command will allow you to select a region on the current plot window to "zoom in" on. This will first show the full-length cross-hair, as from `cursor(crosshair)`, until the mouse is clicked, then show a box with one corner at the first selected point and the opposite corner that follows the cursor until the mouse is clicked again. At that point, the window will be zoomed to this selected box. Using `zoom(nobox)` will suppress the drawing of the cross-hair and "zoom box", showing only the normal cursor plus sign.

You can get the resulting x- and y-coordinates of corners of the selected zoom box by using `zoom(show)`. This will print out two lines of the form

Table 8: Typical PGPLOT device labels and their meaning. Note that several of these are restricted to a single platform, and that other plotting devices may be available on some platforms.

Device Name	Platforms	Description
/null	all	none – no plot will be drawn
/ps	all	black-and-white Postscript file, portrait mode
/vps	all	black-and-white Postscript file, landscape mode
/cps	all	color Postscript file, portrait mode
/vcps	all	color Postscript file, landscape mode
/gif	many	GIF file, portrait mode
/vgif	many	GIF file, landscape mode
/png	many	PNG file
/tpng	many	PNG file with a transparent background
/xwindow	Unix/X	X-window screen
/xserve	Unix/X	X-window screen, persistent plot frame
/gw	Win32	GrWin graphics screen
/aqt	Mac OS X	Mac Aquaterm (non-X) screen

```
cursor: x =     1.03020    , y =    0.001020
cursor: x =     4.30290    , y =    0.963039
```

that will contain the limits of the zoomed plot window.

IMPORTANT NOTE: On Windows systems these variations on the cursor lines and "zoom box" are not working. This is under investigation.

5.7 Graphics Devices

PGPLOT, and therefore IFEFFIT, uses the concept of a plotting *device* to distinguish different output forms. Depending on how PGPLOT was installed, there can be several different devices, including output to the screen (possibly using different screen plotting libraries) and graphic image files suitable for making hardcopies or including in other documents. The list of devices typically available is given in Table 8. To see the available devices for any installation of IFEFFIT, you can inspect the string `$plot_devices`, which will contain a simple space-delimited string of device names as given in Table 8.

```
Ifeffit> print $plot_devices
 /gif /vgif /png /tpng /null /ps /vps /cps /vcps /xwindow /xserve
```

The currently selected device is contained in `$plot_device`. For each platform, at least one interactive plotting screen device is available. Normally one of these is the default plotting device, so that plots are 'live' and interactive. The following sections describe several of the plotting devices in details and how to customize the settings for them.

5.7.1 X-Windows Graphics

For Unix (including Mac OS X), the X Windows library is well supported by PGPLOT and IFEFFIT. To use IFEFFIT on an Unix/X window system, you'll need to set two environmental variables. First, the variable PGPLOT_DIR gives the directory location of the PGPLOT library

(probably /usr/local/pgplot/). Second, the variable PGPLOT_DEV sets the default plotting device. To draw to the X window, this should be set to either '/XSERVE' or '/XWINDOW'. In c-shell and derivatives, this would be done like this (possibly put in the *.cshrc* file):

```
# csh syntax
setenv PGPLOT_DIR /usr/local/pgplot/
setenv PGPLOT_DEV /XSERVE
```

In bash and ksh shells, this would look like this (possibly in the *.profile* file):

```
# bash syntax
 PGPLOT_DIR /usr/local/pgplot/
 PGPLOT_DEV /XSERVE
 export PGPLOT_DIR
 export PGPLOT_DEV
```

The '/XWINDOW' and '/XSERVE' devices look identical, but there is one important difference between them. The '/XWINDOW' plot window will disappear when you leave IFEFFIT, while the '/XSERVE' plot window will remain plotted until explicitly killed by your window manager (say, by clicking the little X button). If after leaving IFEFFIT you start another IFEFFIT session (possibly on a different machine) that same plot window will be used, though two concurrent IFEFFIT sessions will not use the same window.

In addition to the choice of X window type, there are a few other X-window settings for PGPLOT that you may way to customize in your *.Xdefaults* or *.Xresources* file. The two most important settings are:

```
  pgxwin.server.visible:  false
  pgxwin.Win.geometry:    610x377
```

The first of these suppresses a little X-server window that pops up. The second sets the initial window geometry in pixels. Of course, you can resize it, but this will set it to a decent size to begin with. Finally, the X-Windows device has a tendency to use up the X color map, which causes color flashing, especially on machines with older video cards, and especially when other color-intensive programs are running. If this happens, try limiting the number of colors that PGPLOT can set aside:

```
  pgxwin.Win.maxColors:   64
```

did the trick on my old laptop.

5.7.2 GrWin Graphics for Win32 systems

For Win32 users (that is, Microsoft Windows NT 4.0, 95, 98, 2000, ME, and whatever else they come up with), the GrWin graphics used by IFEFFIT are fairly straightforward to use. Cut and Paste to the Windows clipboard works, so importing graphics into documents is easy.

For most users, the GrWin graphics are set-up when the icons to run the command-line program `ifeffit.exe` or one of the GUIs. In general, using the GrWin graphics requires setting the environmental variable PGPLOT_DEV to /GW and the environmental variables PGPLOT_DIR and IFEFFIT_DIR to point to the directory of the IFEFFIT installation, and add this directory to the system PATH. These setting can be done through the "normal setting of environmental variables": autoexec.bat for older versions of Windows, and through the Control Panel or system registry for Windows NT and later.

Alternatively, These settings can be encapsulated into a batch file which runs the IFEFFIT executable program itself. The Windows distribution of IFEFFIT includes such batch files. At this writing, the Windows version of IFEFFIT supports the GrWin, Postscript, and GIF devices.

5.7.3 Aquaterm Graphics for Mac OS X

On Mac OS X, IFEFFIT can be built with X Windows graphics, in which case everything in section 5.7.1 applies. But for Mac OS X users who wish to run IFEFFIT without X Windows, there is an alternative to plot directly to the Aquaterm. This PGPLOT device is still under active development, and though I have seen this work, I have not myself built or used this plotting device. I expect that this will become a fairly standard option, ...

5.7.4 PostScript, GIF and PNG Graphics Files

The plots generated by IFEFFIT can be saved to Postscript, GIF, and PNG files, though these may not all be available on all platforms. The creation of hardcopy as described in this section is an ideal job for IFEFFIT macros, as discussed in chapter 10. You'll probably want to play with the 'hardcopy generation' macros once and then use them extensively without looking at them. As with many aspects of PGPLOT, the quality is not the highest (notably, vector fonts are used, even in the PostScript output), but the results are passable enough for many situations.

To save a plot image to a file, you need to supply a file name and the type of output device. Thuse, to save a black-and-white PostScript version of the current plot to the file *ifeffit.ps*, you would type:

```
Ifeffit> plot(device="/ps",file= "ifeffit.ps")
```

The plot device is automatically reset to the default interactive window after saving the file. Other devices listed in Table 8 will produce different forms of output.

It must be noted that the images generated will preserve the background and foreground color, even for black-and-white output. This means that if you're plotting on a screen with a black background and a white foreground (ie, white text), then the output image file will use a lot of ink when you print it out. You almost certainly want to redraw plots with black foreground and white background for printing:

```
Ifeffit> plot(device="/vps",file= "ifeffit.ps",
              bg=white,fg=black)
```

The size of the output postscript file is set (in units of 0.001" = 25 μm) with the environmental variables PGPLOT_PS_HEIGHT and PGPLOT_PS_WIDTH:

```
# bash syntax: set PGPLOT PostScript size
  PGPLOT_PS_HEIGHT  6000
  PGPLOT_PS_WIDTH   4000
  export PGPLOT_PS_HEIGHT
  export PGPLOT_PS_WIDTH
```

While Postscript files are appropriate for printing and inclusion in papers, GIF and PNG files are more widely used for Web publication. GIF files can be produced with the "/gif" plot device. Landscape mode GIFS are generated with "/vgif". As with the Postscript drivers, the colors generated in the GIF file will be close to those on the screen, including the background and foreground colors. The text strings, on the hand, may be rendered slightly differently than

on the screen. The size of the GIF output file can be set in pixels with the environmental variables PGPLOT_GIF_HEIGHT and PGPLOT_GIF_WIDTH:

```
# cshrc syntax: set PGPLOT PostScript size
  setenv PGPLOT_GIF_HEIGHT  800
  setenv PGPLOT_GIF_WIDTH  1000
```

Due the limited quality of the GIF output, it may be tempting to create very large GIF file and reduce it afterwards. This works reasonably well, though to be honest, I've had better success with doing screen grabs of the PGPLOT window to a native bitmap and converting that to the desired format.

PNG files are similar to GIF, though generally smaller and slightly superior in quality. The PNG files written by PGPLOT are, however, about the same quality as the GIF output. On the bright side, they are not burdened by use of a software patent.

6 Basic XAFS Data Processing

IFEFFIT is designed for processing XAFS data. Though the physical justifications for the analysis procedures are outside the scope of this Reference Manual, some details of how the commands for XAFS data processing are used will be given here. This is not meant as an exhaustive or introductory treatment – the reader is expected to know why these procedures should be done and, to some extent, what pitfalls to avoid.

Many of the simple processes here are easily incorporated into macros, scripts, and other programs. The GUIs for IFEFFIT are all able to automate, or at least provide forms-based interfaces for, the basic XAFS processing described here. The goal here then is not to give exhaustive examples for novice XAFS analysts, but to give the basic ideas of how the commands are operated at the command-line level.

6.1 Data Manipulation and Corrections

XAFS data is generally collected in a small number of data channels (that is, arrays) and either collected at discrete energy values or at least binned in someway into discrete energy values. Typically, there are signals from somewhere between 2 and 20 "detectors" that are collected at discrete energy points. The XAFS $\mu(E)$ can then be determined by simple manipulation of these detector signals.

If that all sounds too abstract, here's a more concrete situation. Usually, one monitors the intensity of the x-ray beam incident on the sample (I_0), and either the intensity of the x-ray beam trasmitted through (I) or fluoresced by (I_f) the sample at several distinct energy values. The XAFS $\mu(E)$ that is analyzed is then given either as $\mu(E) = ln(I/I_0)$ or $\mu(E) = I_f/I_0$. Of course, since I_0, I, and I_f are imperfect measures of x-ray intensities, the validity of using these expressions is only as good as the measure of intensities themselves.

Since much of this Reference Manual discusses the simple manipulation of data, the operations to convert measured intensities to $\mu(E)$ should be straightforward by now:

```
Ifeffit> read_data(file=rb_xafs.dat, group=rb,
                   label='energy i0 it if')
Ifeffit> set rb.xmu_trans = log(rb.it / rb.i0)
Ifeffit> set rb.xmu_fluor =     rb.if / rb.i0
```

When using multiple-element fluorescence detectors, it is usually necessary to add several data channels together to get the total fluorescence intensity. This can be accomplished simply by adding arrays:

```
Ifeffit> read_data(file=med_xafs.dat, group=med,
           label='energy i0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10')
Ifeffit> set med.if = med.f1 + med.f2 + med.f3 + ....
Ifeffit> set rb.xmu = med.if / med.i0
```

When dealing with data files with similar structures, the use of macros (see chapter 10) can make such processing easier.

6.2 De-glitching

It is sometimes necessary to remove data points from the measured $\mu(E)$ spectra because the measurement is obviously dominated by a systematic measurement error. A typical example of

this is a monochromator glitch, where a second diffraction condition is met, lowering the intensity of the desired beam diffracted by the monochromator. Though there may be a temptation to replace the offending points with "smoothed values", it is usually best to simply remove these points from the arrays, reflecting the fact that though you know the measuements were wrong, you don't know what they should have been.

Deglitching by removing a single point is possible with IFEFFIT's syntax, if somewhat klunky. It goes something like this:

```
Ifeffit> new.x = join(slice(old.x,1,badx-1),
                      slice(old.x,badx+1,npts(old.x)))
Ifeffit> new.y = join(slice(old.y,1,badx-1),
                      slice(old.y,badx+1,npts(old.y)))
```

where `badx` is the x location of the bad point (which could be found with the cursor, and then `badx = nofx(old.x,cursor_x)`). This is an excellent task for a macro or script, and can be done by at least some of the data processing programs.

6.3 Pre-Edge Subtraction, Finding E_0, and Normalization

The first step in processing XAFS data is typically to remove the base-line absorption and to normalize $\mu(E)$ so that it has values ~ 0 below the absorption edge and ~ 1 well above the absorption edge. In addition, an estimate of the edge position E_0 is often made at this point.

IFEFFIT has two commands for these steps, both starting with arrays for energy values and a calculation of $\mu(E)$. The `pre_edge()` command will perform all the steps of pre-edge subtraction and normalization given only the input spectra. Alternatively, the `bkg_cl()` command will use the Cromer-Libermann calculations for the pre-edge and normalization processes, and also give a very bad post-edge background subtraction.

The `pre_edge()` command will find E_0 from the maximum of the first derivative. This is usually ends up being a reasonable estimate of the threshold energy, which E_0 is meant to represent. The maximum of the first derivative is also easy to define and so can easily be compared for data from various source.

In many cases, however, you'll want override this default value. This can be especially important when trying to compare many data sets, where the maximum-of-the-first-derivative may move by a point or two even for fairly good data. You can do this by explicitly telling `pre_edge()` what value of E_0 to use, with

```
pre_edge(...,e0=7115.0,...)
```

You can also explicitly set the value of E_0 in the `spline()` command:

```
spline(...,e0=7115.0,...)
```

The normalization

6.4 Simple XANES spectral analysis

Though not necessarily giving a complete understanding of the spectral features, a common approach to XANES analysis is to describe the edge structure as either a sum of pre-defined lineshapes or measured spectra from known "standards". Pre-defined lineshapes typically include a combination of arc-tangents, Gaussian, Lorentzians, or pseudo-Voight functions.

6.5 Post-Edge Background Subtraction: isolating $\chi(k)$

To fully analyze the extended fine-structure, it is traditional to isolate the fine-structure $\chi(k)$ from the slowly-varying "background", $\mu_0(E)$. This is done

6.6 XAFS Fourier Transforms

The Fourier Transform is essential to the understanding of EXAFS.

6.6.1 Forward Fourier Transforms with `fftf()`

6.6.2 BackTransforms with `fftr()`

6.6.3 Phase-Corrected XAFS Fourier Transforms

The XAFS Fourier transform $\tilde{\chi}(R)$ is well-known to have peaks at R values considerably lower than the neighbor distances. The difference between the peak in $\tilde{\chi}(R)$ and the neighbor distance is typically ~ -0.5 Å. This difference is well-understood to be due to the scattering *phase-shift*, the $\delta(k)$ term in the XAFS equation:

$$\chi(k) =$$

7 Fitting XAFS Data with FEFF Calculations

Of course, a major point of IFEFFIT is to be able to manipulate FEFF calculations of XAFS spectra, and especially to fit XAFS data with FEFF calculations. Though many of the details of how this happens is discussed in *XAFS Analysis with* IFEFFIT , this chapter discusses the mechanics of implementing fitting with FEFF calculations in IFEFFIT. Some familiarity with FEFF and FEFFIT will definitely help when reading this chapter.

In order to do a FEFF fit, these things need to be done:

1. run FEFF for a model structure that is expected to resemble your system.
2. read in $\chi(k)$ data to fit to.
3. decide on the fitting space (k, R, or back-transformed k) and Fourier transform parameters to use.
4. define a set of paths, including the FEFF file to use and any numerical path parameters (S_0^2, ΔR, σ^2 and the like).
5. build a fitting model to determine what should be varied in the fit, what should be held constant, and what constraints should be put in place.
6. execute the fit.
7. inspect the results.

If you made it this far in this manual, the first three items should be fairly straightforward. The $\chi(k)$ data can be read-in from a file or generated from the `spline()` command. When using data generated from other programs, an important point is that the IFEFFIT array storing the χ data needs to be on an evenly spaced k-array, with the first point corresponding to $k = 0$ and the k-grid of $0.05\,\text{Å}^{-1}$. The `interp` function can help align data onto this grid, with syntax like

```
read_data(file = my_chi.dat, type = chi, group = unaligned)
new.k    = range(0,ceil(my_chi.k),0.05)
new.chi  = interp(my_chi.k, my_chi.chi, new.k)
```

This use of `range()` and `ceil()` guarantees that `new.k` will start at zero, and goes out far enough in k, both of which are important.

Deciding on the fitting space and Fourier transform parameters depends on what your data looks like, and what you're trying to get from the fit. Those are important topics, but left aside here in favor of getting through the mechanics efficiently.

7.1 Defining and Using Paths

The fit of XAFS data with FEFF calculations uses the idea of a *scattering path* as the basic unit of the XAFS signal. Some approaches to XAFS analysis use the concept of a "shell" or "sphere", which are similar but not identical to the "path". A shell is generally thought of as a group of atoms, usually of the same atomic species, at roughly the same distance from the central atom. A path, on the other hand, represents a set of atoms through which the photo-electron can scatter from before returning to the central atom. For single-scattering XAFS the distinction is subtle, and possibly not worth worrying about – but for multiple-scattering the difference is important,

and the approach of using scattering paths is clearly superior. The total XAFS is then just a sum of these individual paths. A convenient aspect of this approach is that the sum can usually be limited by path distance or to a limited set of "important" paths.

FEFF [8] calculates the XAFS contributions for each path separately. A series of files named *feffnnnn.dat* is written, one for each path with *nnnn* replaced with a 4 digit "path index". Starting with FEFF8, all the scattering information for all the paths is written to the single file *feff.bin*. IFEFFIT can use this file directly, or you can convert it to a series of *feffnnnn.dat* files with the appropriate PRINT flag in FEFF (consult the FEFF documentation for details). FEFF also writes out a single file of the sum-of-paths called *chi.dat*, but IFEFFIT doesn't use this file – it makes it's own sum-of-paths instead.

The `path()` command is used to define paths in IFEFFIT. The path definition consists of a *path index*, the *feffnnnn.dat* file to use for this path, a text-string label, and a set of path parameters which will be used to alter the XAFS for the path. The path index is an integer by which the path will be referred in IFEFFIT. This path index does not need to be the same as the index used by FEFF, but that is often a convenient way to label them. The syntax and keywords for the `path` command are listed in section 9.25. Specifically, the keywords `s02`, `e0`, `delr`, `sigma2`, and `third` give the "standard EXAFS parameters", and should look familiar to those who've used FEFFIT. There are three additional path parameters for the paths in IFEFFIT that are *arrays* to give k-dependent phase-shifts and amplitudes. While these may be necessary for some advanced analyses, or to extend the cumulant expansion, these k-dependent path parameters are infinitely abusable and should be used with caution.

A typical path definition command would look like this:

```
path(index  = 1,
     feff   = feffcu01.dat,
     label  = "Cu metal first neighbor",
     s02    = my_s02,
     delr   = my_delr,
     sigma2 = my_ss2,
     e0     = my_e0  )
```

This defines path #1 to be based on *feffcu01.dat*, and sets the path parameters S_0^2 , ΔR, σ^2, and E_0 to take the values of the IFEFFIT scalars `my_s02`, `my_delr`, `my_ss2`, and `my_e0`, respectively. Actually, since the `path()` command only defines the path parameters, you can use repeated calls instead of long continuation lines:

```
path(1,  feff   = feffcu01.dat)
path(1,  label  = "Cu metal first neighbor")
path(1,  s02    = my_s02,  e0 = my_e0)
path(1,  delr   = my_delr, sigma2 = my_ss2)
```

Note that the path index must be supplied for each execution of `path()`. Abbreviated `path()` commands like this are especially convenient for redefining path parameters.

The *feffnnnn.dat* file is a very important part of the path definition. Each path must have a *feffnnnn.dat* file associated with it. But a *feffnnnn.dat* file can be used for more than one path definition – this is a useful trick in many cases, and very important for multiple data set fits. Because of the likelihood of repeated use of *feffnnnn.dat* files, IFEFFIT keeps an internal list of *feffnnnn.dat* files that have been read in, and will not read in a

[8] IFEFFIT requires version 5 or higher of FEFF. FEFF7 or higher is recommended for use with IFEFFIT.

feffnnnn.dat more than once. In addition, the actual *feffnnnn.dat* file will not be read until it is needed. The file will *not* be read in when you run the `path()` command, but rather when you actually need the information: either when `ff2chi()`, `feffit()` are executed or when you ask to see information about the path with `show @paths` or `get_path()`. So don't be alarmed if the `path()` command executes without seeming to have read the *feffnnnn.dat* file – it's not supposed to.

When summing paths with `ff2chi()` or `feffit()`, IFEFFIT loops through each path, and sets scalars for `path_index`, `reff`, and `degen` corresponding to the "current path". This allows you to refer to `reff` and `degen` in definitions of Path Parameters and be assured that the right value will be used for each path.

As mentioned above, the `show @paths` command is helpful in displaying which paths have been defined, and what their settings are. You may also view individual paths by specifying the path index: `show @path=1` will show information for path 1, while `show @path=1,2,4` will show that for paths 1, 2, and 4. A typical output from executing `show @paths` would look like this:

```
PATH   1
  feff  = ../feff/feffcu01.dat
  id    = Cu metal first neighbor
  reff  =    2.547800, degen  =   12.000000
  s02   =    0.937476, e0     =   -0.867040
  dr    =    0.007575, ss2    =    0.003522
  3rd   =    0.000000, 4th    =    0.000000
  ei    =    0.000000, dphase =    0.000000
```

In addition to the simple `show()` command you can convert path information to Program Variables with the `get_path()` command. This is a very simple command, taking only the path index and a prefix to use for the names of the output scalars:

```
get_path(1, prefix=path1)
```

which will create Program Variables `path1_s02`, `path1_e0`, `path1_delr`, `path1_sigma2`, ..., `path1_reff` from the current values of the Path Parameters for that path.

7.2 Creating $\chi(k)$ data with `ff2chi`

Once a single path or set of paths have been defined, it is often desirable to convert them to $\chi(k)$ data so that they can be plotted and compared to one another or to data arrays. The `ff2chi()` command will do a simple sum-of-paths to give $\chi(k)$ data. This essentially mimics the final step of FEFF, with the additional features like being able to alter any of the path parameters and include *feffnnnn.dat* files from different runs of FEFF in the sum.

`ff2chi()` is a fairly simple command, with most of its arguments describing optional output parameters that are usually not needed. The main argument will be a "path list", which is a list of path indices to sum. Building on the path definition above, the simple

```
ff2chi(1, group=feff)
```

will generate arrays `feff.k` and `feff.chi` from the path #1 defined above. If we then add a second path, with

```
path(2, file = feffcu02.dat, s02 = s02,
  sigma2  = {sqrt(2) * ss2}, e0 = e0)
```

we can add these two paths together, like this,

```
ff2chi(1,2,  group=two_paths)
```

which will generate `two_paths.k` and `two_paths.chi`.

The path list for the `ff2chi()` (and `feffit()`) command can be a simple list like "1,2,3,4", or use a dash like "1-6", or some combination of both, like "1, 3-8, 10, 100-105". Be warned though that a path listed twice will be used twice.

7.3 Building a Fitting Model

Now for the tricky part. Like it's predecessor FEFFIT, the fitting of XAFS data in IFEFFIT is general and flexible, allowing a variety of physical constraints to be imposed on the fit. This is made possible by having the "Path Parameters" be written as a mathematical function of variables and scalars.

In many cases, the fitting model is simple and straightforward to implement. An example would be to use one path, and adjust parameters S_0^2, E_0, and ΔR. To do this, you define variables for each of these parameters:

```
guess (my_s02 = 0.7 , my_ss2 = 0.021)
guess (my_e0= 0.0123, my_delr = 0.1)
```

and then use these variables in the path definition:

```
path(index  = 1,  feff   = feffcu01.dat,
     label  = "Cu metal first neighbor",
     s02    = my_s02,
     sigma2 = my_ss2,
     e0     = my_e0  ,
     delr   = my_delr)
```

In the path definition, the left-hand-side names the path parameter (say, `s02`) and the right-hand-side (`my_s02`) gives the formula used to calculate its value in terms of the variables and other IFEFFIT scalars. Here it's a simple formula, but it could have been more complicated. Using `max(0.5, my_s02)` would put a lower limit on the value of the S_0^2 parameter, for example. An important feature of this approach of having the path parameters be functions of the variables (and not variables themselves) is that any variable like `my_s02` can be used multiple times, say to set the `s02` parameter for different paths. In the most general case, the different path parameters can be quite complex functions of the set of fitting variables.

7.4 Executing a Fit

Most of the hard work (and flexibility) for the user is in building the fitting model. The actual `feffit` command is actually fairly simple to execute. Like its cousin `ff2chi()`, `feffit()` sums a list of paths and generates $\chi(k)$ data. The principle difference is that `feffit()` does a fit – the defined variables are adjusted until a set of data is best-fit. That means that, in addition

to setting up the paths, you also have to set up the $\chi(k)$ data and and Fourier transform and fitting parameters for the `feffit()` command.

A simple example is probably best, so here's a relatively short but complete example of a fit. We read in a $\chi(k)$ data file, 1 feff path, define 4 variables, set up the Fourier transform parameters, and do the fit. It looks like this:

```
# read in chi(k) data file
  read_data(file=../data/cu_chi.dat, type=chi,group=data)

# turn off all previously defined variables
  unguess

# give initial values for fitting variables
  guess s02  = 1.0
  guess ss2  = 0.0
  guess e0   = 0.0
  guess delr = 0.0

# define a scattering path
  path(index  = 1,
       feff   = ../feff/feffcu01.dat,
       label  = "Cu metal first neighbor",
       s02    = s02,
       e0     =  e0,
       sigma2 = ss2,
       delr   = delr)

# define FFT parameters
  set (kmin = 2, kmax =17)
  set (kweight=2, dk1 = 1, dk2 =1)
  set (rmin = 1, rmax = 3)

# keep initial guess , and generate chi(R) for it
  ff2chi(1, group=init)
  fftf(real = init.chi)

# do the actual fit
  feffit(1, chi= data.chi, group = fit)

# show results
  show @variables
  plot(data.r, data.chir_mag, color = blue, xmax = 7, new)
  plot( fit.r,  fit.chir_mag, color = red)
  plot(init.r, init.chir_mag, color = black, style=dashed)
```

We'll refer to this example for the next few sections.

It's important to note that the Fourier transform parameters are set once here, not at each call to `fftf()` and `feffit()`. This relies on the default values for the Fourier transforms being taken from the appropriately named scalars. The values could be set directly in each call to `fftf()` and `feffit()`, but then you have to be more careful that the Fourier transform

parameters are the same for all of the commands. The convention here is a very convenient and reduces chance of typing errors.

The `feffit()` command generates the best-fit values for the `guess`ed variables (`s02`, and so on in this example). It also generates $\tilde{\chi}(R)$ arrays for the data and best-fit – the `data.r`, `data.chir_mag`, `fit.r`, and `fit.chir_mag` arrays here – so that they're immediately ready for plotting or saving to file.

7.5 Estimating the uncertainties in fitted variables

The uncertainties in the fitted variables will be estimated by the `feffit` command immediately after the fit is done. No extra input from the user is required for this automated error analysis. The correlations between pairs of variables will also be calculated. We'll get to those is a bit, after talking about the variable uncertainties.

For each variable `xxx`, the scalar `delta_xxx` will be used to store the estimated uncertainty for that variable. This allows you to see the uncertainties two ways. Either you can either view the set of variables, best fit values and uncertainties together

```
 Iff> show @variables
s02             =      0.93747649 +/-      0.02586825
e0              =     -0.86703986 +/-      0.34801825
delr            =      0.00757485 +/-      0.00153554
ss2             =      0.00352229 +/-      0.00015579
```

or you can select individual variables or uncertainties

```
 Iff> show s02, delta_s02, e0, delta_e0
s02             =       0.937476486
delta_s02       =       0.025868253
e0              =      -0.867039864
delta_e0        =       0.348018253
```

The estimated uncertainties reflect the goodness-of-fit statistics and include the correlations between variables. Of course, the uncertainties are only an estimate. Also, note that if a variable is later set with a `set()` or `def()` command, the scalar `delta_xxx` will remain, probably holding an irrelevant value.

As mentioned above, the correlations between pairs of fit variables are also generated by `feffit()`. Because there are very many possible correlation parameters, many of which are small and uninteresting, these values are *not* automatically converted to Program Variables, but are kept internally (until the next time you execute a `feffit()` or `minimize()` command.) To view the correlations or to convert them to Program Variables, you can use the `correl()` command. A simple way to print out all the correlations is to say

```
Iff> correl(@all,@all,print)
 correl_delr_s02 =    0.115944
 correl_delr_e0  =    0.870971
 correl_ss2_s02  =    0.880360
 correl_ss2_delr =    0.116302
```

The will create the scalars shown (`correl_XX_YY` for variables `XX` and `YY`) and print out their values. The `correl()` command (further discussed in section 9.5) takes its first two

arguments as the name of the variables to find the correlation of (with the special value `@all` meaning to find the correlations with all variables). The keyword `print` means to print out as well as save the correlation values. The minimum correlation (absolute value) to report can be set with the `min` keyword – the default value is 0.05.

7.6 Goodness of Fit Parameters

In addition to doing the fit, the `feffit()` command generates a few scalars for the goodness-of-fit statistics. The scalars `chi_square`, `chi_reduced`, and `r_factor` will contain the values of goodness-of-fit parameters χ^2, χ^2_ν, and $\mathcal{R}$, respectively. The estimated uncertainties in the data in k and R space will be stored in `epsilon_k`, and `epsilon_r`. Details of these calculations are given in *XAFS Analysis with* IFEFFIT . In addition, the number of variables in `n_varys`, the number of fit iterations in `&fit_iteration`, and the number of independent points in the data in `n_idp`, which is defined as

$$N_{\rm idp} = \frac{2\Delta_k\Delta_R}{\pi}$$

where $\Delta k =$ `kmax-kmin` and $\Delta R =$ `rmax-rmin`. This value gives an estimate of the maximum number of parameters that can be determined from the data.

The estimated uncertainties reflect the goodness-of-fit statistics and include the correlations between variables. Of course, the uncertainties are only an estimate, and there are a couple of things you can do to affect these estimates. By far the most important thing you can do is to improve the fit – that's not always that easy.

The goodness-of-fit parameters and, to a very small extent the uncertainties in the fitted parameters, depend on the estimated uncertainty in the data (which can be specified either in k- or R-space). Normally, `feffit()` automatically estimates these for you from the data itself and you don't have to worry about them. If, on the other hand, you want to worry about or change these values, you can use the `chi_noise()` command to do this. `chi_noise()` will estimate the uncertainty in the XAFS data $\chi(k)$ and $\tilde{\chi}(R)$ (`epsilon_k` and `epsilon_r`, respectively), based on the assumption that the noise in the data can be approximated from the high-R components of $\chi(k)$. This can be done explicitly simply as

```
Iff> chi_noise(chi  = data.chi)
```

which will calculate `epsilon_k` and `epsilon_r`, given the current set of Fourier Transform parameters (you can give these with the usual parameters, of course). If this default calculation for the uncertainty in the data is not good enough for your needs, you can explicitly specify the value of `epsilon_k` or `epsilon_r` to use in the `feffit()` command:

```
Iff> feffit(1, chi= data.chi, group = fit, epsilon_k = 0.0008)
```

This will alter the resulting values for `chi_square` and `chi_reduced`, but not `r_factor` or the uncertainties in the fitted variables.

7.7 Post-Fitting Tasks

`feffit()` will generate $\chi(R)$ for the data and total best-fit. It will not, however, generate the back-transforms $\chi(k)$ or the $\chi(R)$ for the individual paths directly. `ff2chi()` will not generate $\chi(R)$ or back-transformed $\chi(k)$ either. So, depending on which set of paths (or partial sums of paths) you'd like to see, you may want to generate the contribution from paths separately by

calling `ff2chi()`, and possibly `fftf()` several times. Such tasks are an ideal job for macros or scripts. As a simple example of a pair of macros I use often, consider

```
macro makepath 1
  "make chi(k) and chi(R) for a single path"
  ff2chi($1, group=path$1)
  fftf(real = path$1.chi)
end macro
macro showpath 1
  makepath $1
  plot(path$1.r, -path$1.chir_mag, $2)
end macro
```

This pair can then be used after a fit like this to show the data and best-fit, and the contributions from each path:

```
feffit(1-3, chi= data.chi, group = fit)
newplot data.r, data.chir_mag, xmax=7
plot    fit.r,  fit.chir_mag
showpath 1  "color=blue"
showpath 2  "color=red"
showpath 3  "color=black,style=linespoints2"
```

You'll also have to manage the writing of output files of the data and fit yourself too, as well as log files. You may want to plot the data and fit first, or look at the variables and then try re-defining some paths or path parameters of variables, or whatever else you can think of to get that perfect fit. Again, using `feffit()` well is generally helped greatly by writing macros for such tasks.

7.8 Additional Fitting Features of `feffit`

Though already somewhat complex and feature-rich, the use of `feffit()` described so far really only shows the basic fitting capabilities of the `feffit()`. Scattering paths are defined, what to vary and what to keep fixed in the fit is described, and the paths are summed together until they match the data, and the results are inspected. It is by no means trivial or easy to come up with a realistic fitting model or assess whether a fit is meaningful, but `feffit()` as described so far gives you all the tools to do these tasks.

In the rest of this chapter, more advanced features of `feffit()` are described. The features include the ability to refine the background ($\mu_0(E)$) parameters at the same time as the structural model, the ability to include additional knowledge about the physical parameters of the systems, and the ability to create and fit a model describing more than one data set at a time. I call these features "advanced", but most of these features are very easy to use, especially when compared to the rather large undertaking of building up a simple fitting model. That is to say that although these features may seem like "advanced topics", and so best left alone by the beginner, they can, in fact, help greatly in assessing the quality and reliability of many fits, and should be kept in mind for many analyses, even by fairly new users. These abilities are being built-in to the ARTEMIS GUI program, and I heartly recommend trying these features.

7.8.1 Including Background Refinement

It is often desirable in XAFS analysis to understand how the background absorption function effects or is affected by the structural fitting parameters. Trationally, this has been difficult to do, as background removal and parameter fitting have been completely separated. Though IFEFFIT still separates these procedures, the `feffit()` command can be used to modify background-like parameters for $\chi(k)$ at the same it modifies structural parameters. This option is very easy to add to a fit: simply add the argument `do_bkg = true` to `feffit()`:

```
Iff> feffit(1, chi= data.chi, group = fit, do_bkg=true)
```

This will automatically add several variables to define a smoothly varying spline $\mu_0(k)$ (now in k-space instead of E-space) that will be added to the model $\chi(k)$ to match your data. The miminum R value used in the fit will be set to 0.0. The *number* of spline parameters will be determined by `rmin`, which now takes the role of `rbkg` in the `spline()` command) so that the spline will only be "free enough" to easily match the low-R portion (ie, those parts below `rmin`) of the spectra.

The main purpose of this feature is to investigate how the background parameters are correlated with the structural (or rather, traditional) fitting parameters. The additional outputs from using this switch are a set of fitting variables `bkg01_01`...`bkg01_NN` for `NN` background variables. The uncertainties in these variables and the correlations between these and the traditional fitting parameters will be available as normal. Note that this feature can greatly increase the number of fitting variables and therefore slow down the fit. In addition to adding the `bkg01_NN` fitting variables, the `do_bkg` switch will also cause the output of additional array, `fit.kbkg` which will contain the additional background function, $\mu_0(k)$ added to the model in order to match the data.

7.8.2 Constraints and Restraints in Fitting

The amount of information available from XAFS is limited, and there is often a fair amount known about the system before you even collect XAFS on it. Because of this, the ability to include some *prior knowledge* about the physical parameters describing the system can be very important to a successful analys. As a first step, you need to be able to impose relationships between path parameters affecting the fit, for example to say that `e0` should be the same for all paths, or that `delr` of one path should be related to that of another path. Up to now, we've only discussed imposing constraints between parameters, using the `def()` command as discussed in section 3.3 to define an exact mathematical relationship between two or more parameters.

Sometimes, however, our prior knowledge is not exact and it desirable to impose *inexact* knowledge or *preferences* on the fit. This can be easily accomplished with a **restraint**. Whereas as **constraint** is a hard, exact relationship imposed on the fit, a **restraint** is a softer relationahip imposed on the fit. I'll avoid an in-depth discussion of restraints in XAFS analysis here, in favor of describing how to do them with IFEFFIT. A restraint is essentially a scalar to be added as an additional element of the vector for the least-squares minimization. To make this happen, you need to define the restraint condition expressed as a scalar value that you would like minimized in the least-squares fit of the data, and then to identify this scalar with the `restraint` keyword in `feffit()`. A simple (if not altogether useful) example would be to impose a restraint that a distance R should be near some value expected from other information:

```
set r_expect = 2.5400
set weight   = 0.01
```

```
def res1     = (reff + delr - r_expect) / (weight)
feffit(1, chi= data.chi, group = fit, restraint=res1)
```

The fit will add the restraint value defined by `res1` to the normal sum-of-squares of the difference between data and fit. An important concept here is the relative weight (represented here with the imaginitively named scalar `weight`) between restraint condition and the normal fit to the data. This topic is left for later discussion.

7.8.3 Multiple-k-Weighting

As we'll see in the next section, IFEFFIT can simultaneously fit more than one data set at a time. In the FEFFIT program, one early and very common use of this feature was to fit the same data set with more than one k-weight at a time. This is an important ability, as simultaneous fits with more than 1 k-weight can significantly reduce the correlation between the EXAFS parameters R and E_0 and between NS_0^2 and σ^2. This ability is so important that it is implemented in IFEFFIT directly, without the need to use the more complicated mechanism of multiple data-set fits.

To fit with only one k-weight at at time, you simply give the k-weighting power with the `kweight` keyword: `feffit(...,kweight=2,...)`, or rely on the previously defined value of the scalar `kweight`. To fit with more than one k-weight at at time, you simply give repeated values of `kweight`, as in

```
Iff> feffit(1, chi= data.chi, kweight=2, kweight=0, kweight=4)
```

which will cause the fit to use k-weights of 0,2, and 4 at the same time. Up to 5 k-weights can be used simultaneously. You must list all the k-weights explicitly, however, and the value in the `kweight` scalar will not automatically be used.

When fitting with multiple k-weights, the order of the listed k-weights only matters for the weight of the automatically created output $\tilde{\chi}(R)$ arrays. These will be formed using the **first** k-weight listed. Of course, to get arrays for the other k-weights, you can also construct your own output arrays using `ff2chi()` and `fftf()` as described in section 7.7.

7.8.4 Simultaneous Fitting of Multiple Data Sets

A very important feature of IFEFFIT is ability to simultaneously fit more than one data set at a time. To be honest, unlike most of the features above, this really is something of an advanced topic, and you should probably be fairly well-acquainted with using the `feffit()` command or with using the older FEFFIT program before trying this yourself. As discussed in the previous section, fitting multiple data sets while only varying the k-weighting is no longer necessary in IFEFFIT.

The key concept in simultaneously fitting multiple sets of data is pose the fitting model to the *set* of spectra, not to individual spectra. This can greatly change the physically model imposed. Nonetheless, even for simple data, the simultaneous analys of multiple data sets can be valuable, as the two key non-structural XAFS Path Parameters (namely, S_0^2 and E_0) can often be asserted to be the same for a set of data that has been measured under similar experimental conditions and that is carefully aligned in energy.

To fit more than one data set at a time, you simply give a series of `feffit()` commands for fits to each of the individual data sets, while identifying each data set as unique, and giving the total number of data sets to be simultaneously fit. The actual fit will not actually be done until all the expected fits to the individual data sets have been defined. Fitting variables, path

indices, and all program variables are globally available, and can be shared between the models for the different data sets [9]. Since all path definitions are visible to all fits, you may have to define different paths that are very similar to one another for fits to different data sets.

A simple example will illustrate this. We'll stick with Cu metal, and fit the data for 3 different temperatures simultaneously. In fact, we'll use the Einstein model for σ^2, and vary the Einstein temperature in the fit instead of the individual `sigma2` parameters. For those with experience using FEFFIT for multiple-data-set fits, the example shown below should look familiar.

The paths for each data set are defined separately: even though they really use the same *`feffnnnn.dat`* file, the values of the Path Parameters `delr` and `sigma2` may be different. In fact, each of these parameters is taken to be temperature dependent: `delr` is linear in temperature and `sigma2` is calculated from the Einstein model. Each `feffit()` command includes `data_set` to identify each data set , and `data_total` to tell how many total data sets there will be. The fit is not really done until `data_total` is equal to the total number of defined data sets – in this case 3.

[9] For those of you with experience with FEFFIT, the concept of the "local" variable does not exist in IFEFFIT. All variables are truly global.

```
# simultaneous fit to 1st shell XAFS for Cu at 10K,
# 50K, and 150K using einstein model for sigma^2

guess (s02     = 0.9, e0   = 3.5)
guess (alpha   = 0.0, beta = 0.00, theta   = 240)

rmin    = 1.60    , rmax    = 2.75
kmin    = 1.5     , kmax    = 18.5
dk      = 1.0     , kweight = 2

read_data(file= ../data/cu10k.chi, group=dat_10, type=chi)
read_data(file= ../data/cu50k.chi, group=dat_50, type=chi)
read_data(file= ../data/cu150k.chi,group=dat_150,type=chi)

path(index  = 101, feff   = ../feff/feffcu01.dat,
     label  = "Cu metal first shell, for 10K",
     s02    = s02,  e0 = e0,
     sigma2 = eins(10, theta),
     delr   = reff * (alpha + 10.0 * beta) )

path(index  = 201, feff   = ../feff/feffcu01.dat,
     label  = "Cu metal first shell, for 50K",
     s02    = s02,  e0 = e0,
     sigma2 = eins(50, theta),
     delr   = reff * (alpha + 50.0 * beta) )

path(index  = 301, feff   = ../feff/feffcu01.dat,
     label  = "Cu metal first shell, for 150K",
     s02    = s02,  e0 = e0,
     sigma2 = eins(150, theta),
     delr   = reff * (alpha + 150.0 * beta) )


feffit(chi = dat_10.chi,  group = fit_10,  101,
       data_set=1, data_total=3)

feffit(chi = dat_50.chi,  group = fit_50,  201,
       data_set=2, data_total=3)

feffit(chi = dat_150.chi, group = fit_150, 301,
       data_set=3, data_total=3)

show @variables, r_factor, chi_square
```

8 Fitting Non-XAFS Data with IFEFFIT

Well, fitting XAFS data to FEFF calculations is OK, but sometimes you just want to fit a simple line, polynomial, or Gaussian to some data. This can be done using the `minimize` command, which gives a simple but powerful interface to a non-linear least-squares fitting routine. There are a few implementation quirks, but this general approach is definitely sufficient to fit simple functions to data, and to add a set of known XANES spectra to fit an unknown spectrum. This chapter will describe the `minimize` command and give a few examples of its use.

When fitting data, the general idea is to minimize some function in the least-squares sense. Usually the function to minimize is the difference between the data and a parameterized model describing the data. This function to be minimized is sometimes called the *residual* of the fit. In keeping with that spirit, the `minimize` function in IFEFFIT takes a residual vector that you define, and adjusts the defined variables until this residual is minimized. An example complete would look like this:

```
read_data(file=my.dat, group= data, label= "x y")
guess (a0 = 1, a1 = 2, a2 = 0.012)
fit.y     = a0 + a1 * data.x + a2 * data.x^2
fit.resid = fit.y - data.y
minimize(fit.resid)
```

Here we read in the data, rename the default array names to more convenient names. The model function `fit.y` is defined as a simple quadratic polynomial, with the three coefficients defined as variables. The residual is then simply the difference of model and data, and the variables are optimized to minimize the sum of the squares of `fit.resid`. Really, that's pretty much all there is to it. `minimize` is remarkably simple and powerful.

There are a few bells and whistles to the `minimize` command. Sometimes you'll want to fit a limited portion of an array, say just over some peak. Of course, you could edit the data to only include the portion of the data you want to fit. But `minimize` gives an alternative to this: you can specify the *ordinate* (or x-array) corresponding to the data you're fitting, and a minimum and/or maximum values for the x-array. In the above example, we could have said

```
minimize(fit.resid, x = data.x, xmin = 3., xmax=10.)
```

to limit the fitting range.

A fit is generally of limited use without some idea of the uncertainties in the fitted parameters. Many otherwise bright people seem to ignore this, and believe they can judge the reliability of fitted parameters by the overall quality of a fit –usually by some visual inspection. Well, a reliable estimate of uncertainties in fitted parameters is a bit more involved than that. In general, it's difficult to get a reasonable estimate without a good estimate of the uncertainties in the data itself. This is further discussed in *XAFS Analysis with* IFEFFIT and in standard data analysis texts. For now, the important point is that if you have a good estimate of the uncertainties in the data, you can use them to determine the uncertainties in the fitted parameters. To do this, you can specify an *array* of uncertainties in the data – the same length as the data and residual itself, of course. To use such an array, the `uncertainty` keyword will be helpful:

```
read_data(file=my.dat, group= data, label= 'x y dy')
guess (a0 = 1, a1 = 2, a2 = 0.012)
fit.y     = a0 + a1 * data.x + a2 * data.x^2
```

```
fit.resid = fit.y - data.y

minimize(fit.resid, uncertainty = data.dy)
```

As with the `feffit` function, this will create scalars for the estimated uncertainties in the fitted variables with names based on the variable name itself. In this case, the created variables `delta_a0`, `delta_a1`, and `delta_a2` will hold the estimated uncertainties.

At this writing, restraints and multiple-data-set fits are not supported in `minimize()`. This will probably change in the future.

9 Commands

This chapter lists all the commands available to IFEFFIT. As discussed in section 3.6, the basic syntax for IFEFFIT commands is

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

That is, each command recognizes a set of command arguments, usually given as keyword/value pairs, where the keyword itself determines what type of variable will be read from the *value* field, and what the command will do with that value. Some commands also use simple lists (comma delimited, but without the form *key* = *value*) for some or all of their input arguments.

Some commands read default values for their command parameters from the global set of Program Variables. Such default values are always read *before* the keyword/value arguments are read, so that explicitly using the corresponding argument will always overwrite the default value. Many commands will write or change output values (scalars, arrays, and/or strings) to the global set of Program Variables.

For each command, the following sections will list these attributes: ‘

Description: gives a brief description of the command. This description does not include much information about the algorithm used or when the use of the command would be appropriate for EXAFS analysis.

Input Program Variables: lists the Program Variables read as the default values of the command parameters.

Keywords/Values: describes the keywords for the command parameters, usually with a table of keywords, default program variable used for input, default value (if not given and default program variable is not set), and a description of the parameter (**Keyword**, **Variable**, **Default**, and **Description** respectively). Many commands have default keywords for the first few keywords – these are indicated by a leading superscript: ^{1}`file` would mean that `file` was the default for the first keyword.

Output Program Variables: lists and describes the program variables that are created or modified. When arrays are created, they are listed with the generic group name `$GROUP`, which will be substituted by the current value of `$group`.

Notes: gives some additional information on the command, typically pointing out the unusual program variables used.

Examples: lists one or more examples of this command, though not necessarily using every feature.

See also: lists similar commands and other places in this *Reference Guide*

Many of the IFEFFIT commands use similar syntax, program variables, and conventions for their command arguments. This is especially true for the input and output of scalars and data arrays, and for parameters used in background removal and Fourier transforms. To this end, and for the sake of brevity and coherence, the glossary in Appendix A contains more detailed description of the common program variables and conventions used. Additional hints can usually be found in the Notes: section.

9.1 bkg_cl

Description: Use values of x-ray scattering factors derived from the Cromer-Libermann tables to estimate the pre-edge and normalization constant for XAFS data. The tabulated values for $f''(E)$ are modified by multiplying by a constant and adding a quadratic polynomial in energy so that they best match the input XAFS data $\mu(E)$.

Input Program Variables: `$group`.

Keywords/Values:

Keyword	Variable	Default	Description
[1]energy			energy array name
[2]xmu			xmu array name
[3]z			atomic number for element
group	`$group`		group name for output arrays
e0			E_0, the energy origin
width			energy convolution width for calculated $\mu(E)$
edge_step			Edge Step
pre1		-200.	pre-edge line lower limit
pre2		-50.	pre-edge line upper limit
norm1		100.	normalization line lower limit
norm2		300.	normalization line upper limit
norm_order		2.	order of normalization polynomial
find_e0		F	flag to force finding E_0
interp		quad	method to use for data interpolation

Output Program Variables: Scalars: `e0`, `edge_step`, `pre1`, `pre2`, `norm1`, `norm2`, `pre_slope`, `pre_offset`, `norm_c0`, `norm_c1`, `norm_c2`, `$group`.

Arrays: `$GROUP.pre`, `$GROUP.norm`, `$GROUP.k`, and `$GROUP.chi` for the data and `$GROUP.f2`, `$GROUP.f2pre`, and `$GROUP.f2norm` for the calculation.

Notes: The Cromer-Libermann calculations have a sharp jump in $f''(E)$ at the absorption edge which generally needs to be broadened (here, by using `width`, in eV) to match $\mu(E)$ data.

The output pre-edge subtracted muE (`$GROUP.pre` and `$GROUP.norm` for the normalized version) are probably the most useful – while `$GROUP.k` and `$GROUP.chi` may be completely useless. The output arrays for the calculation are chosen to match the input $\mu(E)$ data (`$GROUP.f2`) and the pre-edge subtracted (`$GROUP.f2pre`) and normalized (`$GROUP.f2norm`) data.

Examples:

```
Iff>  bkg_cl(data.energy,data.xmu, z = 29)
```

See also: `f1f2`(Section 9.11), `spline`(Section 9.41), `pre_edge` (Section 9.31).

9.2 chi_noise

Description: Estimates the measurement uncertainty of XAFS data. The estimates are made simply from the RMS value of the high-R range of $\chi(R)$, under the assumption that the EXAFS has died out substantially above 15Å or so.

Input Program Variables: FT parameters `kmin`, `kmax`, `kweight`, `dk1`, `dk2`, `$kwindow`, and `$altwindow`.

Keywords/Values:

Keyword	Variable	Default	Description
[1]chi			array of χ data to estimate noise of.
k			array of k data.
group	`$group`		group name for arrays.
kmin	`kmin`	0	FT parameter
kmax	`kmax`	0	FT parameter.
dk1	`dk1`		FT parameter.
dk2	`dk2`		FT parameter.
dk			sets both `dk1` and `dk2`.
kweight	`kweight`		FT parameter.
kwindow	`$kwindow`		FT parameter.
altwindow			FT parameter.
rwgt1		15.	lower R bound of high R range.
rwgt2		25.	upper R bound of high R range.

Output Program Variables: `epsilon_k` will contain the estimated uncertainty in $\chi(k)$, and `epsilon_r` will contain the estimated uncertainty in $\chi(R)$. `kmin`, `kmax`, `kweight`, `dk1`, `dk2`, and `$group` will be updated.

Notes: In practice, `chi_noise` is rarely used directly, because `feffit` will automatically run this for you. It is sometimes useful to understand how the uncertainties are estimated.

Examples:

```
Iff>  chi_noise(data.chi,kmin=2,kmax=15,dk=1,kweight=2)
```

See also: `feffit`(Section 9.12).

9.3 color

Description: Manipulate the plotting color table. The color table is used to set the default colors for the traces (x-y pairs) in a plot as well as the background, foreground and grid colors. Note that colors on screen will not be affected until `plot` is re-executed.

Input Program Variables: None.

Keywords/Values:

Keyword	Variable	Default	Description
show			display color table
fg		black	foreground color
bg		white	background color
grid		#CCBEE0	color of grid displayed on window
1		blue	first x-y trace
2		red	second x-y trace
3		green	third x-y trace
4		black	fourth x-y trace
5		magenta	fifth x-y trace
⋮			⋮

Output Program Variables: None.

Notes: The foreground color will be used for the outer box, tick marks, and all text on plot. The list of named colors can be found in the file *rgb.txt* in the PGPLOT installation directory.

Examples:

```
Iff>  color (fg = white, bg = black)
Iff>  color (1 = red, 2 = '#33EEBB')
```

See also: `linestyle`(Section 9.19),`plot`(Section 9.27), section 5.3.

9.4 comment

Description: Write a comment line to the command history buffer.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None.

Examples:

```
Iff> comment 'this next part requires four hands to play'
```

See also: `history`(Section 9.18).

9.5 correl

Description: Converts selected values from the correlation matrix of fitting variables into named program variables.

Input Program Variables: correl_min.

Keywords/Values:

Keyword	Variable	Default	Description
[1]x			name of first fitting variable
[2]y			name of second fitting variable
out		(see notes)	scalar name for result
min	correl_min	0.05	minimum correlation to report
print		F	flag to print correlation values
save		T	flag to save correlation values to scalars
no_save		F	flag to NOT save correlation values to scalars

Output Program Variables: None.

Notes: If the out argument is not given and if the save option is used, the output scalar containing the correlation will be named correl_xx_yy for variables xx and yy.

The value @all can be used for either or both of x and y to tell the command to extract all the correlations for that variables.

If the correlation value requested is smaller than the minimum reportable correlation, then nothing is printed or saved.

Examples:

```
Iff> correl(x=e0,y=delr_1,print,no_save)
   correl_e0_delr1 =  0.870124

Iff> correl(x=e0,y=delr_1,save,min=0.2)
Iff> print correl_e0_delr_1
   correl_e0_delr1 = 0.870124

Iff> correl(x=e0,y=@all,print,min=0.6)
   correl_e0_delr1  = 0.870124
```

See also: feffit(Section 9.12), minimize(Section 9.23).

9.6 cursor

Description: Select a point on the graphics screen, and get its x and y values. The point is typically selected with a left-click of the mouse button. The program will wait until the point is selected before continuing. Since the program will do nothing until input is given, it is recommended that macros, scripts, and programs inform the user that input is expected.

Input Program Variables: None.

Keywords/Values: The following flags can be given to customize the behavior of `cursor()`:

Keyword	Description
show	print output string after selection
last_pos	start cursor at previously selected position
cross-hair	show cross-hair over full plot window
horiz	show horizontal line over full plot window
vert	show horizontal line over full plot window
xrange	see section 5.6
yrange	see section 5.6

Output Program Variables: `cursor_x`, and `cursor_y` contain the x and y positions of the cursor, respectively.

Notes:

Examples:

```
Iff>  cursor(show)
    cursor_x =     2.05795    , cursor_y =    -3.34442
```

See also: `plot`(Section 9.27), `zoom`(Section 9.46), Section 5.6.

9.7 def

Description: define a Program Variable. In contrast to `set()`, `def()` remembers the definition (ie, the mathematical formula) of numerical Program Variables. This is especially useful for complex math expressions used in non-linear least squares fits.

Input Program Variables: None.

Keywords/Values: The keyword is taken as the name of the variable to be assigned, and the Value is taken as the mathematical expression to use for the definition.

Output Program Variables: None.

Notes: `def()` is the default command and so is optional. That is, simply typing `a = b` is equivalent to `def a = b`.

Examples:

```
Iff> def (b = a + 1,  c = 100 * sqrt(b) )
Iff> def my.chik = my.chi * my.k^kweight
```

Note that both `b` and `c` will change if `a` changes, and that `my.chik` will automatically update when `kweight` changes.

See also: `set`(Section 9.39), `sync`(Section 9.42).

9.8 echo

Description: echo a text string to the screen, without any interpolation of variables. This is mostly useful in macros.

Input Program Variables: `&screen_echo`, which can be used to turn on and off the actual 'echo'ing.

Keywords/Values: None.

Output Program Variables: None.

Notes: Setting `&screen_echo` to zero will suppress the actual 'echo'ing (see example). Note that this will also cause all `pause()` commands to be ignored, which can be useful for batch processing.

In fact, setting `&screen_echo` to an even value will suppress the actual echo, and setting it to an odd number will turn on the echo. This can be used in conjuction with the `log()` command to fully control printing to the screen and/or the log file defined with `log()`. `&screen_echo` of 0 will suppress all printing, 1 will print to the screen but not the log file, 2 will print to the log file but not the screen, and 3 will print to both.

Examples:

```
Iff> echo "here's a comment!"
   here's a comment!
Iff> &screen_echo = 0
Iff> echo "here's a comment!"
Iff>
```

See also: `comment`(Section 9.4), `log`(Section 9.21), `macro`(Section 9.22), `pause`(Section 9.26), `print`(Section 9.32).

9.9 erase

Description: Erase one or more Program Variables. Erasing removes the variable from memory (as opposed to just resetting it). This is not a keyword/value command, but a list-directed command see section 3.6).

Input Program Variables: None.

Keywords/Values: `@arrays`, `@scalars`, `@strings`,will erase every array, scalar, and string. You can also erase an entire group with `@group`, or a path definition with `@path`.

Notes: You cannot currently erase only fitting variables, or individual macros.

Output Program Variables: None.

Examples:

```
Iff> erase kmin, my.energy
Iff> erase @strings
Iff> erase @group = my
```

This first will erase the scalar `kmin`, and the array `my.energy`. The second will erase all text strings. The third will erase all arrays in the `my` group.

See also: `rename`(Section 9.35), `set`(Section 9.39), `show`(Section 9.40), `unguess`(Section 9.43).

9.10 exit

Description: Exit the program.

Keywords/Values: None.

Output Program Variables: None.

Examples:

```
Iff> exit
```

See also: `quit`(Section 9.33).

9.11 f1f2

Description: Get x-ray scattering factors $f'(E)$ and $f''(E)$ derived from the Cromer-Libermann tables over a specified energy range. The tabulated values can be broadened with a Lorenztian function.

Input Program Variables: `$group`.

Keywords/Values:

Keyword	Variable	Default	Description
[1]energy			energy array name
[2]z			atomic number for element
group	`$group`		group name for output arrays
width		0.	energy convolution width.
do_f1		T	flag for calculating $f'(E)$.
do_f2		T	flag for calculating $f''(E)$.

Notes: The width, if supplied, will be used to broaden both $f'(E)$ and $f''(E)$.

The sign convention used for $f'(E)$ and $f''(E)$ are the "conventional", if somewhat internally inconsistent version that is in wide use in crystallography. That is, $f''(E)$ is a positive quantity and $f'(E)$ is negative, and at an absorption edge the change in $f''(E)$ will be positive and the cusp in $f'(E)$ will point down.

Output Program Variables: Arrays `$GROUP.f1` (for $f'(E)$) and `$GROUP.f2` for $f''(E)$ will be generated for each energy of the input energy array.

Examples:

```
Iff> f1f2(energy=cu.energy, z=29)
```

See also: `bkg_cl()` (Section 9.1),

9.12 feffit

Description: Fit XAFS $\chi(k)$ data to a sum of FEFF paths, optimizing a set of fitting variables in the process.

Input Program Variables: rmin, rmax, kmin, kmax, kweight, dk1, dk2, $kwindow, data_set, data_total, $fit_space.

Keywords/Values:

Keyword	Variable	Default	Description
[1]*path list*			list of paths (see note 1)
chi			array of χ data to fit
k			array of k data
group	$group	feffit	group name for created arrays
rmin	rmin	0	$R_{\rm min}$: lower R bound of fit
rmax	rmax	0	$R_{\rm max}$: upper R bound of fit
kmin	kmin	0	FT parameter
kmax	kmax	0	FT parameter
dk1	dk1		FT parameter
dk2	dk2		FT parameter
dk			sets both dk1 and dk2
kweight	kweight		FT parameter
kwindow	$kwindow		FT parameter
altwindow			FT parameter.
epsilon_k	epsilon_k		uncertainty in $\chi(k)$ data
epsilon_R	epsilon_r		uncertainty in $\chi(R)$ data (see note 2)
toler		1.e-8	fitting tolerance
data_set	data_set	1	index of the current data set
data_total	data_total	1	total number of data sets in fit
fit_space	$fit_space		name of space for fit (see note 3)
do_real		F	save real part of best-fit $\tilde{\chi}(k)$
do_mag		F	save magnitude of best-fit $\tilde{\chi}(k)$
do_phase		F	save phase of best-fit $\tilde{\chi}(k)$
do_bkg		F	refine background spline (see note 4)
macro			user macro to run at each iteration (see note 5)
restraint			scalar restraint (see note 6)

Output Program Variables: data_total, chi_square, chi_reduced, r_factor, n_idp, n_varys, epsilon_k, epsilon_r, and the above FT parameters. In addition, the estimated uncertainty for each variable will be stored in variables with names like delta_VAR. Arrays will be created (or overwritten) for the best-fit $\chi(k)$ and $\chi(R)$: $GROUP.k and $GROUP.chi, $GROUP.chir_mag, $GROUP.chir_real, $GROUP.chir_imag.

Optional output arrays can be written for other parts of the complex $\tilde{\chi}(k)$ according to the appropriate keyword: $GROUP.chi_real (do_real), $GROUP.chi_mag (do_mag), and $GROUP.chi_phase (do_phase).

Notes:
1. feffit() uses a list of paths as the default argument.
2. If neither epsilon_k nor epsilon_r arguments are provided, feffit() will execute chi_noise() on the supplied data to get these values.

3. Valid values for `fit_space` are 'k', 'R', and 'Q', with 'Q' meaning backtransformed k-space. The default is 'R'.
4. By using `do_bkg`, the fit will add several variables to define a spline $\mu_0(k)$ that will be added to the model $\chi(k)$, and the miminum R value used in the fit will be set to 0.0. The *number* of spline parameters will be determined by `rmin` (now acting in the role of `rbkg` in the `spline()` command) so that the spline will only be "free enough" to easily match the low-R portion of the spectra.
5. A user-defined macro can be run at each iteration of the fit. This will cause serious problems when `feffit()` is itself called in a macro, but works otherwise. This makes for a convenient way to inspect a fit as it happens.
6. Up to separate 10 restraint conditions can be added.

Examples:

```
Iff> feffit(chi=cu.chi,  1,2,4-6, rmin=1,rmax=4,
            kmin=2, kmax=18, kweight=2, dk=5,
            kwindow='Kaiser')
```

See also: `chi_noise()` (Section 9.2), `ff2chi`(Section 9.13), `minimize`(Section 9.23), `path`(Section 9.25), Chapter 7.

9.13 ff2chi

Description: Sum a set of FEFF paths to generate $\chi(k)$. The paths must be defined by the `path()` command.

Input Program Variables: None.

Keywords/Values:

Keyword	Variable	Default	Description
[1]*path list*			list of paths
group		feff	group name for created arrays
kmin		0.	$k_{\min}$ for output arrays
kmax		20.	$k_{\max}$ for output arrays
s02		1.	overall scale factor for $\chi(k)$
sigma2		0.	overall σ^2 for $\chi(k)$
do_real		F	save real part of $\chi(k)$
do_phase		F	save phase-shift of $\chi(k)$
do_mag		F	save magnitude of $\chi(k)$
do_all		F	save all optional output

Output Program Variables: Arrays for k and $\chi(k)$ will be written to `$GROUP.k` and `$GROUP.chi`.

Optional output arrays can be written for other parts of the complex $\tilde{\chi}(k)$ according to the appropriate keyword: `$GROUP.chi_real` (`do_real`), `$GROUP.chi_mag` (`do_mag`), and `$GROUP.chi_phase` (`do_phase`).

Notes: Like `feffit()`, `ff2chi()` uses a list of paths as the default argument.

Examples:

```
Iff> ff2chi(1,2,4-9, do_phase)
```

See also: `feffit`(Section 9.12), `fftf`(Section 9.14), Chapter 7.

9.14 fftf

Description: Forward XAFS Fourier Transform of an array. Generally used for transforming from $\chi(k)$ to $\tilde{\chi}(R)$. This does a discrete Fourier Transform.

Input Program Variables: kmin, kmax, dk1,dk2,kweight, $kwindow and rmax_out.

Keywords/Values:

Keyword	Variable	Default	Description
[1]real			array for Re[χ(k)].
imag			array for Im[χ(k)].
k			array of k data
group	$group		group name for output arrays
kmin	kmin	0	$k_{\rm min}$ FT parameter
kmax	kmax	0	$k_{\rm max}$ FT parameter
dk1	dk1		FT parameter
dk2	dk2		FT parameter
dk			sets both dk1 and dk2
kweight	kweight		k-weight FT parameter
kwindow	$kwindow		FT window function
altwindow			array for alternate FT window.
phase_array			phase array for phase-correction.
pc_edge			element name and edge symbol for phase-correction.
pc_feff_path			index of path to use for phase-correction.
pc_caps			flag to use central-atom phase-shift in phase-correction
pc_full			flag to use full phase-shift in phase-correction

Output Program Variables: kmin, kmax, dk1, dk2, kweight, and rmax_out will be set on output.

In addition, several arrays will be created: $GROUP.win will contain the k-space window array, $W(k)$; $GROUP.r will contain the array of r values, $GROUP.chir_mag will contain $|\tilde{\chi}(R)|$, $GROUP.chir_re will contain Re[$\tilde{\chi}$(R)], and $GROUP.chir_im will contain Im[$\tilde{\chi}$(R)].

Notes: See Appendix B.

Normally, $ftf_real names the $\chi(k)$ array, and $ftf_imag remain unset.

If the real (or imaginary) part of the input $\chi(k)$ data does *not* start at $k = 0$, the array of k values should be specified with the keyword k. Otherwise, the FT will be inaccurate.

Phase-corrected Fourier transforms can be done by providing a phase array – see section 6.6.3 and Appendix B for details.

Examples:

```
Iff> fftf(real=my.chi, kmin = 1.0, kmax =16.0,
          dk=1.0, kweight=2., kwindow= 'hanning' )
```

See also: fftr(Section 9.15), window(Section 9.44), section 6.6, section 6.6.3, Appendix B.

9.15 fftr

Description: Reverse XAFS Fourier Transform of an array. Generally used for transforming from $\tilde{\chi}(R)$ to $\tilde{\chi}(k)$. This does a discrete Fourier Transform..

Input Program Variables: rmin, rmax, dr1,dr2,$rwindow.

Keywords/Values:

Keyword	Variable	Default	Description
[1]real			array for $\mathrm{Re}[\chi(\mathrm{R})]$.
imag			array for $\mathrm{Im}[\chi(\mathrm{R})]$.
r			array of R data
group	$group		group name for output arrays
rmin	rmin	0	R_{min} FT parameter
rmax	rmax	0	R_{max} FT parameter
dr1	dr1		FT parameter
dr2	dr2		FT parameter
dr			sets both dr1 and dr2
rweight	rweight		R-weight FT parameter
rwindow	$rwindow		FT window function
altwindow			array for alternate FT window.

Output Program Variables: rmin, rmax, dr1, dr2, rweight, and rmax_out will be set on output.

In addition, several arrays will be created: $GROUP.rwin will contain the R-space window array, $W(R)$; $GROUP.q will contain the array of k values for the back-transform, $GROUP.chiq_mag will contain $|\tilde{\chi}(k)|$, $GROUP.chiq_re will contain $\mathrm{Re}[\tilde{\chi}(\mathrm{k})]$, and $GROUP.chiq_im will contain $\mathrm{Im}[\tilde{\chi}(\mathrm{k})]$.

Notes: See Appendix B.

rweight sets the R-weight for the Fourier Transform, but is not traditionally used.

Examples:

```
Iff> fftr(real=my.chir_re, imag = my.chir_im,
          rmin = 1.0, rmax =4.0, dr = 0.2)
```

See also: fftf(Section 9.14), window(Section 9.44), Appendix B.

9.16 get_path

Description: Convert Path Parameters from a FEFF path into regular program variables.

Input Program Variables: None.

Keywords/Values:

Keyword	Description
[1]path	Path index.
[2]group	group name for arrays and prefix for scalar names.
do_arrays	flag to create arrays

Output Program Variables: The scalar values of the Path Parameters `s02`, `e0`, `ei`, `delr`, `sigma2`, `third`, `fourth`, `degen`, and `reff`, will be written to variables `PREFIX_s02`, `PREFIX_e0`, `PREFIX_ei`, ... `PREFIX_reff` where `PREFIX` is the given path prefix.

If `do_arrays` is set, the arrays from the *`feffnnnn.dat`* file will be turned into IFEFFIT arrays, with names `PREFIX.k`, `PREFIX.amp`, `PREFIX.phase`, `PREFIX.caps`, `PREFIX.rep`, and `PREFIX.lambda`. These arrays are on a non-uniform and fairly sparse k-grid, which can be interpolated onto a uniform k-grid.

Notes: The default prefix for path NNN is `pathNNN`. That is, the default prefix for path 1 is `path001`.

Examples:

```
Iff> get_path(2)
Iff> show path002_reff, path002_s02
   path001_reff      =  3.6032000
   path001_s02       =  0.9300000
```

See also: `feffit`(Section 9.12), `ff2chi`(Section 9.13), and `path`(Section 9.25).

9.17 guess

Description: Define a fitting variable, and set it's initial value.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: Keywords for `guess` are interpreted as names of numeric scalars. The Values are interpreted as math expressions, evaluated upon definition to give the initial value of the variable.

Several Keyword/Value pairs can occur together. The parentheses are optional.

Examples:

```
Iff> guess (x = 1. , y = 2.00 )
Iff> guess  z =  y * sqrt(2)
```

See also: `set`(Section 9.39), `sync`(Section 9.42), `feffit`(Section 9.12), `minimize`(Section 9.23), `unguess`(Section 9.43).

9.18 history

Description: Open a file to record a "history" of IFEFFIT commands as they are executed.

Input Program Variables: None.

Keywords/Values:

Keyword	Variable	Default	Description
[1]file			name of history file.
off			flag to turn off history recording.

Output Program Variables: The variable `$historyfile` is updated to name the current (or most recently used) history file.

Notes: The formal syntax for setting variables is recorded. This reflects the fact that the commands recorded are those actually executed. Certain commands (such as the recording of macros and `history` itself) are not recorded.

The history file is *not* guaranteed to be readable until after IFEFFIT has been exited, or until the recording has been explicitly turned off with a `history(off)` command.

Examples:

```
Iff> history(ifeffit.his)
Iff> comment "Here's a comment"
Iff> a = 1.22
Iff> b = a * 8
Iff> history(off)
```

This will create a file *ifeffit.his*, and then fill it with the following lines:

```
# Here's a comment
def (a = 1.22)
def (b = a * 8)
```

See also: `comment`(Section 9.4), `load`(Section 9.20).

9.19 `linestyle`

Description: Manipulate the table of plotting linestyles. The linestyle table (like the color table) is used to set the default linestyles for the traces (x-y pairs) in a plot.

Input Program Variables: None.

Keywords/Values:

Keyword	Variable	Default	Description
show			display linestyle table
1		solid	first x-y trace
2		solid	second x-y trace
3		solid	third x-y trace
⋮			⋮

Output Program Variables: Valid values for the linestyles are: `solid`, `dashed`, `dotted`, `dot-dashed`, `pointsN` and `linespointsN` (both for N = 1,2,3,...).

Like for the color table, the linestyle table is affected by explicitly setting the linestyle of a trace with the `plot()` command.

Examples:

```
Iff> linestyle(1=solid, 2 =dashed)
```

See also: `color`(Section 9.3), `plot`(Section 9.27).

9.20 `load`

Description: Load a file of IFEFFIT commands and execute them. The file loaded can have any valid filename, and can itself `load` other files. This is especially useful in conjunction with the `history` command, and for writing macros.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None.

Notes: `load()`'s can be nested, so that `load`ed files can themselves contain `load` statements.

Examples:

```
Iff> load My_macros.iff
```

See also: `history`(Section 9.18), `macro`(Section 9.22), Chapter 10.

9.21 log

Description: Control the writing of commands and screen output to an external *log file*.

	Keyword	Variable	Default	Description
Keywords/Values:	file			log file name to open and write to
	close			close log file

Output Program Variables: None.

Notes: A log file cannot be named `close`.

If a file named `file.log` exists when executing `log(file=file.log)`, that file will be replace with the new log file.

Examples:

```
Iff> log(file = analysis.log)
Iff> log(close)
```

See also: `echo`(Section 9.8).

9.22 macro

Description: Define a macro – a sequence of IFEFFIT commands to be executed later by simply typing the macro name (and optional parameters). Macro definitions end with the line '`end macro`', and can call other macros.

Input Program Variables:

Keywords/Values: None.

Output Program Variables: None.

Notes: Macros may take positional parameters, which are interpreted as text strings. In the macro definition, these parameters are referenced by the special string variables $1, $2, . . . , $9.

Examples:

```
macro prex
  read_data( $1, type = xmu,  group = my)
  pre_edge(my.energy, my.xmu)
  show e0, edge_step
end macro
```

This macro will read in a $\mu(E)$ data file, subtract a line for the pre-edge, and plot the normalized, pre-edge subtracted $\mu(E)$ as a function of energy relative to E_0. It would be called like this:

```
Iff> pre my_data.xmu
```

Note that one parameter is used in this macro, and that it will continually overwrite the arrays in the '`my`' group.

See also: `macro`(Section 9.22), Chapter 10.

9.23 minimize

Description: Minimize an array in the least squares sense, by adjusting the values of the fitting variables. This gives a simple and flexible way to fit general data to a fairly simple models.

Input Program Variables: None.

Keywords/Values:

Keyword	Variable	Default	Description
[1]array			residual to be minimized
x			x-array associated with array
uncertainty			array of uncertainties in residual
xmin			low-x value for fit range
xmax			high-x value for fit range
toler		1.e-8	fitting tolerance
restraint			scalar fitting restraint

Output Program Variables: `chi_square`, `chi_reduced`. For each variable `XXX`, the variable `delta_XXX` will be given it's estimated uncertainty.

Notes: The array named by `x` is optional, and is necessary only if `xmin` or `xmax` are given. The array given by `uncertainty` is optional as well.

Currently, only 1 restraint scalar can be added.

Examples:

```
Iff> guess (a  = 1, b = 0)
Iff> my.resid = my.data - (a * my.x + b)
Iff> minimize(my.resid)
```

See also: `feffit`(Section 9.12), Chapter 8.

9.24 newplot

Description: Draw a new plot on the graphics device (usually the screen).

Input Program Variables: None.

Keywords/Values: Same as `plot`.

Output Program Variables: Same as `plot`.

Examples:

```
Iff> newplot(my.x, my.y)
```

See also: `plot`(Section 9.27), Chapter 5.

9.25 path

Description: Define a FEFF path and specify the path parameters. The paths defined in this way can be used in either ff2chi or feffit to create $\chi(k)$.

Paths are referred to by an integer *index*, which is a required keyword. Every path must have an index and a *feffnnnn.dat* file associated with it – all other path parameters are optional, taking "normal defaults" (zero for all parameters except S02 which defaults to one).

Input Program Variables: feff_file is used as the default *feffnnnn.dat* file.

Keywords/Values: All values are treated as text strings except for index which must be an integer. Except for label and feff, strings for path parameters are interpreted as math expressions, either giving a scalar or array.

Keyword	Variable	Default	Description
[1]index			path index, an integer used to make path lists
[2]feff			*feffnnnn.dat* file to use for this path
label			text string to describe path
s02		1	NS_0^2 – constant amplitude factor
e0			E_0 – energy shift
delr			ΔR – change in path distance
sigma2			σ^2 – mean-square-displacement
third			C_3 – third cumulant
fourth			C_4 – fourth cumulant
ei			E_i – shift in imaginary energy term.
k_array			array of k-values for k-dependent phase-shift and amplitudes
phase_array			array of k-dependent phase-shift
amp_array			array of k-dependent amplitude factor

Output Program Variables: None.

Notes: All the numerical Path Parameters can be generalized *expressions* of the fitting variables and other defined Program Variables. The path() command only defines the path, and may not even cause the *feffnnnn.dat* file to be read.

Examples:

```
Iff> path(index = 1, file = feff0001.dat, s02= 1,
          sigma2 = sig2)
Iff> path(2,  feff0002.dat, s02= 1,  sigma2 = sig2)
Iff>
Iff> path(3,  feff0003.dat, sigma2 = 'sig2 * sqrt(3)')
Iff> path(3,  delr   = reff * alpha)
```

See also: ff2chi(Section 9.13), feffit(Section 9.12), get_path() (Section 9.16), Chapter 7.

9.26 pause

Description: Write a message to the screen, and suspend the program until the user hits any key on the keyboard. This is often useful to put in macros or `load()`ed files that will perform multiple plotting tasks.

Input Program Variables: `&screen_echo`, which determines whether messages are sent to the screen, and `&pause_ignore`, which sets whether or not to ignore all `pause()` commands, will influence the operation of this command.

Keywords/Values: `pause()` takes one argument – a string that is printed to the screen to prompt the user to 'hit any key'. The default string is '`-- hit any key to continue -- `'.

Output Program Variables: None.

Notes: In order for the `pause()` command to actually be executed, `&screen_echo` must be 1 and `&pause_ignore` must be 0. Thus, setting `&pause_ignore` to 1 will suppress the command, which may be useful for batch processing or scripts.

Examples:

```
Iff> pause '>> hit any key to see chi(k) <<'
```

See also: `echo`(Section 9.8), `print`(Section 9.32).

9.27 plot

Description: The general plotting command, specifying x- and y-arrays to plot, and plot attributes. The plot created can be either to the graphics screen or to the current output graphics device (such as a postscript file).

The `plot` command takes a huge variety of arguments.

Input Program Variables: `$plot_device`, `$plot_file`, `$plot_xlabel`, `$plot_ylabel`, `$plot_title`, `$group`.

Keywords/Values:

Keyword	Variable	Default	Description
[1]x			x-array
[2]y			y-array
group	`$group`		group name
dy			array of error bar size for y
dx			array of error bar size for x
xmin			lower limit for x-range
xmax			upper limit for x-range
ymin			lower limit for y-range
ymax			lower limit for y-range
color		(table)	color for current trace
style		(table)	line style for current trace
width		2	line width for current trace
bg		white	background color
fg		black	foreground (labels,axis) color
grid		T	flag for showing grid
nogrid			flag for hiding grid
gridcolor		#CCBEE0	color of grid lines
xlabel	`$plot_xlabel`		string for x-axis label
ylabel	`$plot_ylabel`		string for y-axis label
title	`$plot_title`		string for plot title
key			text of key for legend
charfont		1	font for text strings
charsize		1.5	font size for all text strings
labelsize		1.5	font size for axis labels and titles
markersize		1.5	font size for point markers
textsize		1.5	font size for `text` labels
text			text string for general label
text_x			x-coordinate for text string
text_y			y-coordinate for text string
cleartext			flag to erase all ”text” labels
file	`$plot_file`		file name for non-screen outputs
device	`$plot_device`		name of plot device
new			flag for not overplotting
reset			flag to reset all plot attributes

Output Program Variables: `$plot_device`, `$plot_file`, `$plot_xlabel`, `$plot_ylabel`, `$plot_title`. If a plot attribute (color, style) for a particu-

lar trace is altered, it will be remembered until a 'reset' is issued.

Notes: Further details are in Chapter 5.

The default color and linestyle are dictated by internal tables.

Examples:

```
Iff> plot(my.x, my.y, color=green,xmin= 0,
         title = 'Y v. X')
```

See also: `color`(Section 9.3), `cursor`(Section 9.6), `newplot`(Section 9.24), `zoom`(Section 9.46), `plot_arrow()` (Section 9.28), `plot_marker()` (Section 9.29), `plot_text()` (Section 9.30), Chapter 5.

9.28 plot_arrow

Description: Add an arrow or line to the current plot.

Input Program Variables: None.

Keywords/Values:

Keyword	Description
[1]x1	x-coordinate of arrow tail
[2]y1	y-coordinate of arrow tail
[3]x2	x-coordinate of arrow head
[4]y2	y-coordinate of arrow head
clear	erase all arrows from plot
no_head	use no arrow head
fill	fill in arrow head
outline	use outline of arrow head
size	size of arrow head
angle	angle of arrow head
barb	size of arrow barb

Output Program Variables: None.

Examples:

```
Ifeffit> plot_arrow(x1=10, y1= 4, x2=25, y2=4,  barb=2)
```

See also: Section 5.5.

9.29 plot_marker

Description: Add a marker or symbol to the current plot. See Figure 2 for the available symbols.

Input Program Variables: None.

Keywords/Values:

Keyword	Description
[1]x	x-coordinate of marker
[2]y	y-coordinate of marker
[3]marker	integer of plot marker to use
clear	erase all markers from plot

Output Program Variables: None.

Examples:

```
Ifeffit> plot_marker(x=7000,y=4,marker=1)
```

See also: Section 5.5, Figure 2.

9.30 plot_text

Description: Add a text string to an arbitrary location on the current plot. This is equivalent to using `text_x`, `text_y`, and `text` arguments to `plot()`.

Input Program Variables: None.

Keywords/Values:

Keyword	Description
[1]x	x-coordinate of text
[2]y	y-coordinate of text
[3]text	text string to add to plot
size	font size for `text` labels
clear	erase all strings from plot

Output Program Variables: None.

Examples:

```
Ifeffit> plot_text(x=7025,y=0.3, text='400 K Data')
```

See also: `plot`(Section 9.27), Section 5.4.

9.31 pre_edge

Description: Calculate the pre-edge line through XAFS $\mu(E)$ data, the energy origin E_0, and the edge step.

Note that the spline command may call pre_edge for you if it appears that it has not already been called.

Input Program Variables: pre1, pre2, norm1, norm2.

Keywords/Values:

Keyword	Variable	Default	Description
[1]energy			energy array name
[2]xmu			xmu array name
group	$group		group name
e0	e0		E_0, the energy origin
edge_step	edge_step		Edge Step
pre1	pre1	-200.	pre-edge line lower limit
pre2	pre2	-50.	pre-edge line upper limit
norm1	norm1	100.	normalization line lower limit
norm2	norm2	300.	normalization line upper limit
pre_slope	pre_slope		slope of pre-edge line
pre_offset	pre_offset		offset of pre-edge line
find_e0		F	flag to force finding E_0

Output Program Variables: e0, edge_step, pre1, pre2, norm1, norm2, pre_slope, pre_offset, $group, and $GROUP.pre.

Notes: The edge step will be found unless specified. E_0 will be found unless specified and in the data range.

Examples:

```
Iff> read_data(my.xmu, group = my)
Iff> pre_edge(my.energy, my.xmu)
```

See also: bkg_cl() (Section 9.1), spline(Section 9.41).

9.32 `print`

Description: Write the *value* of a list of Program Variable or expressions to the screen. Because `print` uses list context, it does a poor job parsing complex expressions. Expressions that include spaces should be enclosed in parentheses. Alternatively, you can enclose strings in quotes to prevent them from evaluation. This gives a reasonably flexible way to format outputs.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None.

Examples:

```
Iff> print "7 * sqrt(99.11) = ",  (7 * sqrt(99.11))
  7 * sqrt(99.11) =     69.6878038
```

See also: `echo`(Section 9.8), `show`(Section 9.40).

9.33 `quit`

Description: Quit the program.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None.

Examples:

```
Iff> quit
```

See also: `exit`(Section 9.10).

9.34 read_data

Description: Read array data from ASCII column file.

Input Program Variables: $commentchar.

Keywords/Values:

Keyword	Variable	Default	Description
[1]file			Name of input file.
group			Default group name for arrays.
type			File Type to assume for array names.
label			Label line to use for array names.
npts			length of arrays to read.
narray			number of array to read.
commentchar	$commentchar	#	comment character for text lines.

All arrays read in will share a common group name. If not explicitly given, the group name will be determined from the file name. The arrays read in will be named according to conventions described in Chapter 3 and Chapter 4.

Output Program Variables: Arrays will be read in, and text strings will be read in. In addition, $group will hold the group name used, and $commentchar will hold the comment character used. $filetype will hold the file 'type', if appropriate. Most importantly, $column_label will hold what the colum label *should have been* to give the resulting array names. That is, it will contain a space-delimited list of array suffixes.

Comment strings at the top of the data file will also be saved in text strings with names $GROUP_title_01, $GROUP_title_02, ...

Examples:

```
Iff> read_data(file= My.dat, type=raw, group= my)
Iff> read_data( CuS04_002.dat, label = 'energy xmu i0')
```

See also: write_data() (Section 9.45), Chapter 3, Chapter 4.

9.35 rename

Description: Rename one or more Program Variables.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None.

Notes: Use of this command can be very detrimental to effective and rational use of definitions and complex fitting models. In short, the *name* of the variable is changed, but does not change its definition or the definitions of the variables that depend on it. Sometimes this is exactly what you want. Sometimes it is not.

Examples:

```
Iff> rename kmin kmin_save
Iff> rename my.x your.x

Iff> a = 1
Iff> b = a + 1
Iff> rename(a, c)
Iff> c = 5
Iff> print b
  6.000000
Iff> rename(b, d)
Iff> c = 10
Iff> print d
 11.000000
```

See also: `erase`(Section 9.9), `set`(Section 9.39), `show`(Section 9.40).

9.36 reset

Description: Reset all IFEFFIT Program Variables.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: All Program Variables are erased, and all program settings re-initialized.

Examples:

```
Iff> reset
```

See also: `exit`(Section 9.10).

9.37 restore

Description: Restore a save'ed IFEFFIT session.

Input Program Variables: None.

Keywords/Values: The only argument is the name of the save file to restore. The keyword file is optional.

Output Program Variables: All Program Variables read from the save file are updated.

Examples:

```
Iff> restore(my.sav)
```

See also: save(Section 9.38).

9.38 save

Description: Save all IFEFFIT program variables into a single file for later restoration.

Input Program Variables: All Program Variables.

Keywords/Values:

Keyword	Variable	Default	Description
[1]file		*ifeffit.sav*	Name of output save file.
npad		8	WordLength for PAD numbers.
no_strings		F	Flag to not save text strings.
no_arrays		F	Flag to not save arrays.
no_scalars		F	Flag to not save scalars.
no_sys		F	Flag to not save "system scalars".
with_strings		T	Flag to save text strings.
with_arrays		T	Flag to save arrays.
with_scalars		T	Flag to save scalars.
with_sys		T	Flag to save "system scalars".

Notes: A wordlength npad of 8 gives at least 12 significant digits. Higher precision can be achieved by setting npad as high as 12, which results in about 15 significant digits – roughly at the machine resolution of most implementations of double precision. True double precision cannot be guaranteed with this format, but 12 digits of *portable* data will mask many machine differences, and is probably good enough for most applications involving experimental data.

Output Program Variables: None.

Examples:

```
Iff> save(my.sav)
```

See also: restore(Section 9.37).

9.39 `set`

Description: Set a Program Variable. In contrast to `def`, `set` does not remember the definition (ie, the mathematical formula) of numerical Program Variables, but only the value at the time of creation.

Note that `def` is the default command, which means that `set` must be done explicitly.

Input Program Variables: None.

Keywords/Values: The keyword is taken as the name of the variable to be assigned, and the Value is taken as the mathematical expression to use for the definition.

Output Program Variables: Well, the Program Variable is set.

Examples:

```
Iff> set (b = a + 1,  c = 100 * sqrt(b) )
Iff> set my.chik = my.chi * my.k^kweight
```

Note that neither `b` nor `c` will change if `a` changes, and that `my.chik` will not change when `kweight` changes. Sometimes this kind of constance and predictability is exactly what you want. For those other times, you'll want `def`.

See also: `def`(Section 9.7), `sync`(Section 9.42), `print`(Section 9.32).

9.40 show

Description: Show information about IFEFFIT Program Variables, commands, macros, and `feffit` paths.

Input Program Variables: None.

Keywords/Values: The argument to `show` is usually interpreted as a list of the names of Program Variables to display. In addition to the 'normal' Program Variables, the names of user-defined macros can also be included. For scalars and strings, `show` will display the value of these Program Variables. For arrays, the number of points and maximum and minimum values are shown (the `print`(Section 9.32) command will print all the values of an array, if that's what you want).

`show` can also take a few "global" arguments to show several variables at once. All such global arguments begin with "@", which can be taken as a mnemonic for "all", and a few can take an additional argument. The table below lists the available "global" arguments.

Argument	Value	What is Shown
@scalars		all scalars.
@arrays		all arrays.
@strings		all text strings.
@variables		all fitting variables, with uncertainties.
@groups		all array "groups".
@group	group name	all arrays in selected group.
@paths		all paths for the current data set.
@path	path list	selected paths.
@commands		all commands, with brief description.
@macros		all user-defined macros.

See the examples below for syntax.

Output Program Variables: Outputs are written to the screen.

Examples:

```
Iff> show rmin, fit.chi
  rmin         =      1.30000000
  fit.chi      = 499 points [-0.5341341   : 0.5121385]
Iff> show @groups
  data
  fit
Iff> show @group=data
  data.k       = 499 points [ 0.500000E-01:  24.95000]
  data.chi     = 499 points [-0.1756163   : 0.1433899]
Iff> show @path=1
   PATH   1
     feff  = feffcu01.dat
     id    = Cu metal first neighbor
     reff  =    2.547800 , degen  =   12.000000
     s02   =    0.934530 , e0     =    0.558189
     dr    =    0.000771 , ss2    =    0.003483
```

```
3rd    =     0.000000 , 4th    =     0.000000
ei     =     0.000000 , phase  =     0.000000
```

See also: `set`(Section 9.39), `print`(Section 9.32), `echo`(Section 9.8).

9.41 spline

Description: Calculate the background spline $\mu_0(E)$ and EXAFS and $\chi(k)$ given arrays for $\mu(E)$. This command uses the AUTOBK algorithm, described in more detail in *XAFS Analysis with* IFEFFIT .

Input Program Variables: `e0`, `rbkg`, `toler`, `nknots`, `kmin_spl`, `kmax_spl`, `kweight_spl`, `dk1_spl`, `dk2_spl`, `$kwindow`, `edge_step`, `pre1`, `pre2`, `norm1`, `norm2`.

Keywords/Values:

Keyword	Variable	Default	Description
[1]energy			energy array name
[2]xmu			xmu array name
group	`$group`		group name
e0	`e0`		E_0, the energy origin
rbkg	`rbkg`	1.0	$R_{\rm bkg}$
toler	`toler`	1.d-3	Fitting tolerance
nknots	`nknots`		Number of knots in spline
kmin	`kmin_spl`		$k_{\rm min}$ for FFT in spline evaluation
kmax	`kmax_spl`		$k_{\rm max}$ for FFT in spline evaluation
kweight	`kweight_spl`	1	k-weight for FFT in spline
dk1	`dk1_spl`		δk_1 parameter for FFT in spline
dk2	`dk2_spl`		δk_2 parameter for FFT in spline
kwindow	`$kwindow`		name of FFT window type for spline
edge_step	`edge_step`		Edge Step
pre1	`pre1`	-200	pre-edge line lower limit
pre2	`pre2`	-50	pre-edge line upper limit
norm1	`norm1`	100	normalization line lower limit
norm2	`norm2`	300	normalization line upper limit
eefind		F	flag to force finding E_0
varye0		F	flag to allow E_0 to vary in spline fit
find_step		F	flag to force finding of Edge Step
fnorm		F	flag to normalize by $\mu_0(E)$.
do_pre		T	flag to force the finding of the pre-edge
do_spl		T	flag to force spline fit
interp		quad	method to use for data interpolation

Output Program Variables: `$group`, `e0`, `rbkg`, `nknots`, `kmin_spl`, `kmax_spl`, `kweight_spl`, `dk1_spl`, `dk2_spl`, `$kwindow`, `edge_step`, `pre1`, `pre2`, `norm1`, `norm2`.

Examples:

```
Iff>  spline(my.energy,my.xmu,rbkg=1.0,kmin=0)
```

See also: `pre_edge` (Section 9.31).

9.42 sync

Description: Synchronize numeric Program Variables (fitting variables, scalars and arrays) so that all dependencies are up-to-date and all values are consistent with one another. This command is implicitly run at the beginning of commands `ff2chi`, `feffit`, and `minimize`.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None. Well, all scalars and arrays are 're-arranged'.

Examples:

```
Iff> a  =  1,   b  =  3,   c  = (a + b)/2
Iff> b  =  5
Iff> show a, b, c
  a   =   1.00000
  b   =   5.00000
  c   =   2.00000
Iff> sync
Iff> show a, b, c
  a   =   1.00000
  b   =   5.00000
  c   =   3.00000
```

See also: `ff2chi`(Section 9.13), `feffit`(Section 9.12), `minimize`(Section 9.23), `set`(Section 9.39), `def`(Section 9.7), and section 3.3.

9.43 unguess

Description: Change all `guess()`ed program variables to `set()` variables with the current values.

Input Program Variables: None.

Keywords/Values: None.

Output Program Variables: None. Well, all fitting variables are changes to regular scalars.

Examples:

```
Iff> unguess()
```

See also: `set`(Section 9.39), `def`(Section 9.7), `guess`(Section 9.17), section 3.3.

9.44 window

Description: Generate an XAFS Fourier Transform window, without actually doing the Fourier transform.

Input Program Variables: kmin, kmax, dk1,dk2,kweight, $kwindow and rmax_out.

Keywords/Values:

Keyword	Variable	Default	Description
[1]real			array for $\mathrm{Re}[\chi(\mathrm{k})]$.
imag			array for $\mathrm{Im}[\chi(\mathrm{k})]$.
k			array of k data
group	$group		group name for output arrays
kmin	kmin	0	k_{min} FT parameter
kmax	kmax	0	k_{max} FT parameter
dk1	dk1		FT parameter
dk2	dk2		FT parameter
dk			sets both dk1 and dk2
kweight	kweight		k-weight FT parameter
kwindow	$kwindow		FT window function
altwindow			array for alternate FT window.

Output Program Variables: kmin, kmax, dk1, dk2, kweight, and rmax_out will be set on output.

The array $GROUP.win will contain the k-space window array, $W(k)$.

Notes: See Appendix B.

Examples:

```
Iff> window(real=my.chi, kmin = 1.0, kmax =16.0,
          dk=1.0, kweight=2., kwindow= 'hanning' )
```

See also: fftf(Section 9.14), fftr(Section 9.15), Appendix B.

9.45 write_data

Description: Write scalars, strings, and arrays to an ASCII data file.

Input Program Variables: `$group`, `$commentchar`.

Keywords/Values: The arguments for `write_data` are mostly interpreted as a list of Program Variables to write to the file. The list elements can be either strings or arrays, but (currently) not scalars. The text strings will be written first, followed by a line of minus signs, and then the arrays will be written in column format. Currently supported keywords are:

Keyword	Variable	Default	Description
[1]file			Name of output file.
group			Default group name for arrays.
npts			number of array points to write.
commentchar	`$commentchar`	#	comment character for text lines.

For writing groups of text strings, 'globs' are supported with the '*' character. That is, a '*' in the name of text string variables will be expanded so that all strings matching the pattern will be printed. Text lines will begin with a 'comment character'. By default, this is '#', but can be set to any two character sequence.

The maximum number of arrays that can be written to a single file is 16. Array *expressions* are not supported.

Output Program Variables: None.

Examples:

```
Iff> write_data(file=out.dat, $title*, my.x, my.y)
```

See also: `read_data` (Section 9.34).

9.46 zoom

Description: Zoom in on a region of the plot window. Using the cursor (typically a mouse), click on the lower-left and upper-right portion of the plot window you wish to enlarge.

Input Program Variables: None.

Keywords/Values: The following flags can be given to customize the behavior of `zoom()`:

Keyword	Description
show	print output string after selection
nobox	suppress drawing of 'active zoom box'

Output Program Variables: `cursor_x`, and `cursor_y` will contain the x and y positions of the cursor for the last point chosen.

Examples:

```
Iff>  zoom
```

See also: `cursor`(Section 9.6), `plot`(Section 9.27), Section 5.6.

10 Macros in IFEFFIT

As described so far in this Guide, IFEFFIT is definitely not suitable for the casual user. The syntax is a bit fussy and cryptic, and the whole notion of a command-based system is not generally considered 'user friendly'. Even for the experienced user, typing at a command prompt can get pretty tedious for repetitive tasks.

With that in mind, this chapter and the next might be the most important chapters in this Guide, because they are all about making IFEFFIT easier to use. With the macro capability described in this chapter, it is easy to write and customize files for "batch processing" of data. The next chapter will extend these ideas further, and move beyond the simple macros described here, and discusses writing full-blown application programs using IFEFFIT with real programming languages like fortran, C, Perl, Python, and Tcl.

Data analysis often involves repetitious processing of data, so IFEFFIT has a simple built-in macro capability that is easy-to-use and reasonably flexible. A macro is a named block of IFEFFIT lines that can be executed as a single unit by typing the name of the macro. An example:

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

With this definition, typing `make_ps` at the command line would execute the two `plot` commands, making a postscript named *`ifeffit.ps`* showing the current plot, and setting the plotting device back to the X-window. As you can probably tell, macros are defined with the command `macro` *macro_name*. All lines up to `end macro` (which must be on its own line) make up the text of the macro, and will be executed in order when the macro is invoked.

To make macros slightly more useful, you can use positional arguments to pass information into macros. The parameters are handled as text strings, and simply inserted in the macro text before being executed. In keeping with the IFEFFIT naming convention, and following many shell and batch processing facilities, the parameters are named \$1, \$2, ... \$9. So, to make the above macro a little more flexible, we use

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

Now, typing `make_ps my_plot.ps` at the command line will dump the current plot to a file named *`my_plot.ps`*, and you can make several different postscript files by changing the argument.

Unfortunately the file name in parameter \$1 is *required* by the macro, which might not be all that useful unless you always remembered it. To help with the problem of required arguments, you can specify default values for each argument when defining a macro, like this:

```
macro make_ps ifeffit.ps
  "Make Postscript file of current Plot"
  plot(device="/ps",file= $1)
end macro
```

This version of `make_ps` will use *`ifeffit.ps`* as the default value of the first argument. So typing `make_ps my.ps` will make a file called *`my.ps`*, and `make_ps` without any arguments will write *`ifeffit.ps`*.

Notice the extra line at the top of this macro:

```
  "Make Postscript file of current Plot"
```

This is the optional **macro description**, which acts as built-in documentation. If the first line of a macro is enclosed in single quotes, double quotes, or braces, it is used as the macro description. This line is not executed when the macro is run, but is only used to describe the macro to the outside world. Specifically, `show @macros` will show all macro names, default arguments, and description.

Here's an example macro to automate pre-edge subtraction:

```
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
```

Now, typing `do_pre_edge Data_1` would read in *`Data_1.xmu`*, calculate the normal pre-edge parameters like E_0, and the edge step, and write out the pre-edge subtracted $\mu(E)$ data (in `my.pre`) to *`Data_1.pre`*. And `do_pre_edge Data_2` would repeat these same steps on another file.

The variables $1 through $9 are special text strings that can only be used in macros. The contents of these strings are destroyed when the macro is exited. Within the macro, these strings are simply substituted in place. This is an admittedly feeble system – adding string manipulation functions and simple control structures would enhance the utility of macros, and is planned for future versions.

Macros can be nested. Parameters passed into underlying macros are always handled by position number, and the "argument stack" is automatically managed, which should be what you'd expect. That is, with

```
macro mac1
  mac2  $3  $2
  print $1
end macro
macro mac2
  print $1
  print $2
end macro
```

typing `mac1 A B C` will print "C", then "B", and then "A". As mentioned above, macro arguments are interpreted as strings and are substituted in place just before execution. To specify the argument values, then, it will often be helpful to separate them by commas, or to enclose them in double quotes ("...") or braces ({ ... }). Then

```
  mac1 ''As usual'', {1}, ''Here is the third argument''
```

will print

```
  Here is the third argument
  1
  As usual
```

The macro processing does essentially no error checking. And, again, the arguments $1 to $9 are simply inserted as text strings. So if an argument $1 to $9 is expected and is not explicitly provided and no default is given for it in the macro definition, then a null string will be used. If a null string is not a valid argument in some command, the command will still be executed, occasionally with non-sensical results or error messages. If you're working on a complex macro, it may be helpful to 'debug macros' by putting the line

```
show @args
```

Note that the following

```
print $1, $2, $3, $4
pause == my_macro:  are these parameters right? ==
```

may **not** be as helpful, since the arguments will already have been substituted in place of $1, ...$4.

Of course a key point of having macros is to re-use them. For that, you'll probably want to save your favorite macro definitions to a file and use the '`load`' command.

11 Scripting and Programming with IFEFFIT

The macro system in IFEFFIT described in the previous chapter only goes so far, and sometimes it's just not enough. There are several limitations to the 'IFEFFIT language' and to the macro mechanism that make complex data processing difficult. The lack of conditional (if-then-else) statements and loops (for or do) are the most notable missing features.

All of the programming capabilities missing from IFEFFIT and more are available in essentially every programming and scripting language. Modern scripting languages (which are roughly distinguished from programming languages by not having a compilation to machine code separate from execution but rather being run directly from the text of the code) are especially attractive for such applications, as they allow quick development and execution and a wide range of programming capabilities. Scripting languages are generally simpler to learn than full-blown programming languages, and more flexible and forgiving of errors to boot. Scripting languages have a proven track record as being useful for creating "wrappers" around low-lying libraries such as the IFEFFIT library.

There are several possible general-purpose scripting languages and a few "scientific visualization languages" that could, in principle, be used to extend IFEFFIT's capabilities. Tcl/Tk, Perl, Python, IDL, Java, VisualBasic, LabView, Matlab, and Mathematica all come to mind. At this time, IFEFFIT works well with Perl, Python, and Tcl as well as with C and Fortran. If you have a favorite language that is not on this list and would like to use IFEFFIT with it, please let me know.

The interface between IFEFFIT and the various programming and scripting languages are all quite similar. The interfaces allow you to send commands from the language just as you would type commands at the command-line program, and also provide ways to move Program Variables back and forth between the underlying engine and the calling program. This chapter describes using IFEFFIT from Fortran and C, as well as Perl, Python, and Tcl. Though the basic concepts are the same for all the languages, there are some slight differences in implementation so as to be able to best exploit the features of the different languages. Even if you're only planning on using one of the scripting languages, I recommend that you read the Fortran and C sections since it has the most complete description of the interface.

11.1 Which language to use?

If you're unfamiliar with the world of scripting languages and are interested in getting more out of IFEFFIT, you're probably wondering at this point which of the scripting language to use. Allow me to give some brief recommendations. These should be immediately be seen as the free advice of a highly opinionated person. Oddly, I believe they are in fairly close agreement with most others familiar with these languages.

If you already know C or Fortran, those are fine languages to use. If you don't know either C or Fortran it is difficult to recommend learning them just so you can write complex IFEFFIT scripts. Both languages are fairly intolerant of mistakes and have fewer features than the modern scripting languages. Once mastered, however, Fortran and C give very fast and efficient programs.

Whether or not you know C or Fortran, if you're interested in writing complex IFEFFIT scripts or programming in general, I recommend learning one of Perl, Python, or Tcl. These scripting languages are remarkably similar in that they all work well on every major platform, are free, and well-supported via the internet. They are all fairly easy to learn, and make it easy to write and debug simple scripts. They also hide the really ugly parts of C from you, and provide

'high-level' data structures which lets you do some fairly sophisticated things that would be more painful in C or Fortran. They all have their quirks, too, which can be both maddening and charming. Learning any of them will greatly improve your computer skills, as well as making complex IFEFFIT scripts possible.

Of the three, Tcl[**?**] is probably the oldest, simplest and least powerful language. That's not to say it's bad – it's power is certainly good enough for most things. It may even be the most popular of the three languages. Its syntax is very simple, and yet a lot of amazing things have been done in Tcl/Tk. These days Tcl is essentially synonymous with the truly wonderful and portable Tk GUI toolkit, and is often just referred to as Tcl/Tk. It is hard to over-emphasize the ability to write GUIs that work on Unix, Mac, and Win32 machines. Still, in my opinion Tcl/Tk is the least interesting scripting language.

Perl[**?**, **?**] is probably best known as a Unix system-administration and Web-scripting language. In some sense, Perl is Unix distilled into a single language, for both good and bad. It excels at string and text processing (parsing, pattern matching, and formatting of text output). If you're interested in learning web-scripting or Unix, learning Perl is a good choice. Perl attempts to be a 'natural language' and so provides several syntax options and some truly breathtaking constructs. All this can lead to programs that intermix punctuation-laden lines and near-English text. The Tk GUI toolkit works with Perl on Windows and Unix, but not on Macintoshes. Several IFEFFIT scripts have been written in Perl, and Bruce Ravel (who rewrote the ATOMS program in Perl), has some advanced Perl modules for using IFEFFIT.

If you don't already know Tcl or Perl, learning Python[**?**] may be your best bet. Though probably the least popular of the three languages, Python is growing in popularity and is especially good for scientific programming (complex math is supported!). Python has a very nice implementation of object-orientation and is generally hailed for its readability and "cleanliness". That's not to say that Python is without its own quirks, but it is almost certainly the most elegant and easiest to learn of the three scripting languages mentioned. As a bonus, the Tk GUI toolkit works with Python on Mac, Windows, and Unix. As an additional incentive, G.I.FEFFIT is written in python.

11.2 Controlling screen outputs: The echo buffer

An important consideration for either scripting or programming with IFEFFIT is how to handle the text messages that are written to the screen during an interactive session. These messages include unprompted warnings and error messages as well as information that you explicitly asked to be shown, say through a `show()` command. The variable `&screen_echo`, first mentioned in section 3.8, can be used to control whether this output is actually written to the screen (technically speaking, *standard output*, which may not even be visably available from your program) or saved to an *echo buffer* that you can access from your program.

11.3 The Fortran interface to IFEFFIT

IFEFFIT is written primarily in Fortran, so using it from within Fortran programs is quite easy. The basic use of IFEFFIT is to send command strings to an "IFEFFIT engine" which acts just like an interactive IFEFFIT session run at the command prompt. The underlying engine has its own set of Program Variables that are kept in its own memory space, separate from the calling program. The session "stays alive" until the calling program ends.

Though you could directly call any of the subroutines or functions in the IFEFFIT library, it is highly recommended that you **not** make such direct calls. Instead, you should use the functions

Table 9: Integer return values from `ifeffit()`, a function to execute IFEFFIT commands in an "IFEFFIT engine". The function is accessible from Fortran, C, C++, Perl, Python, and Tcl.

Return value	Meaning
0	normal, successful execution
-2	in the middle of a macro definition
-1	in the middle of an incomplete command line
1	normal exit
> 1	abnormal exit

provided in the application programming interface (API), as described here and encapsulated in the Fortran include file *ifeffit.inc* that can be found with the configuration files in the IFEFFIT distribution (typically in the */usr/local/share/ifeffit/config/* directory).

The IFEFFIT Fortran API defines eight external functions, all of which are integer functions. To use these functions, you can simply put a fortran `include` statement at the top of your program, so that instead of explicitly declaring the integer function `ifeffit`, you could say

```
      program use_if
      integer i
      include '/usr/local/share/ifeffit/config/ifeffit.inc'
c
      i = ifeffit(' ')
      i = ifeffit('read_data(cu.xmu, group=cu, type=xmu)')
      i = ifeffit('spline(cu.energy, cu.xmu, rbkg = 1.2)')
      i = ifeffit('plot(cu.k, cu.chi)')
      end
```

This shows a very simple IFEFFIT session converted into a Fortran program, using only the function `ifeffit()`. This function is the main interface to the underlying IFEFFIT engine.

As this example shows, it is recommended that you first call `ifeffit()` with an "initialization string", typically a blank line, but optionally setting system configuration variables.

How do you actually build an executable out of this program file? That, of course, depends on details of your system. Most of the settings needed are put in the file *Config.mak* in the same location as *ifeffit.inc*. This file contains Makefile instructions needed for linking your IFEFFIT application with the IFEFFIT library and all the other libraries needed to make an executable. An example Makefile (using the above code and the settings of Config.mak from a fairly normal linux system) is included in the *examples/scripting* section of the source distribution.

11.3.1 integer function `ifeffit()`

The `ifeffit()` function takes a string as an argument (up to 1024 characters), and returns an integer, which will have one of the following values given in Table 9

You should be somewhat careful about the characters you actually send to `ifeffit()`, especially with respect to non-printable characters and line-ending issues. Though it tries to remove non-printing characters, it may not be wise to simply open a file and send its contents to `ifeffit()` without checking that the file does not contain binary data.

11.3.2 integer function `iffputsca()`

The `iffputsca()` function takes two arguments: the first is a character string (up to 128 characters) that names an IFEFFIT scalar (following the naming rules outlined in chapter 3), and the second is a double precision value. The effect is to set the named scalar with the given value. If the scalar already exists, it will be overwritten. Note that this is equivalent to a `set()()` command, not a `def()()` command: for that, you should use the `ifeffit()` function.

`iffputsca()` always returns 0.

```
i = iffputsca('kmin', 3.d0)
x = sqrt(100.)
i = iffputsca('kmax', x)
i = ifeffit(' show kmin, kmax')
```

would show the values to be 3.0 and 10.0, respectively.

11.3.3 integer function `iffgetsca()`

The `iffgetsca()` function takes two arguments: the first is a character string (up to 128 characters) that names an existing IFEFFIT scalar (following the naming rules outlined in chapter 3), and the second is a double precision variable. The effect is to retrieve the value of the named IFEFFIT scalar and put it into the provided Fortran variable. If the scalar does not exist in the IFEFFIT session, the value will be set to 0.

`iffgetsca()` always returns 0.

```
i = ifeffit(' set var = sqrt(100.0)')
i = iffgetsca('var', x)
print*, ' x = ', x
```

would show the value 10.0.

11.3.4 integer function `iffputarr()`

The `iffputarr()` function takes three arguments: the first is a character string (up to 128 characters) that names an IFEFFIT array (following the naming rules outlined in chapter 3), the second is an integer giving the length of the array, and the third is a double precision array. The effect is to set the named IFEFFIT array with the provided array. If the array already exists, it will be overwritten.

`iffputarr()` always returns 0.

```
double precision x(200),y(200)
do i = 1, 200
  x(i) = i * 4.0
  y(i) = sin(x(i) / 100.)
end do
i = iffputarr('my.x', 100, x)
i = iffputarr('my.y', 100, y)
i = ifeffit(' show @arrays')
i = ifeffit(' plot my.x, my.y, color=red')
```

would show the arrays `my.x` and `my.y` to have 100 elements.

11.3.5 integer function `iffgetarr()`

The `iffgetarr()` function takes two arguments: the first is a character string (up to 128 characters) that names an existing IFEFFIT array (following the naming rules outlined in chapter 3), the second is a double precision array to store the output result. The effect is to retrieve the value of the named IFEFFIT array and store it into the provided Fortran array.

`iffgetarr()` will return the length of the output array. It is an error for the array to not contain enough elements to be filled.

```
double precision x(200)
i = ifeffit('my.x  = range(0,100,1)')
n = iffgetarr('my.x', x)
print*, 'x has ', n, ' elements:'
print* , x(1), x(2), ' ... ', x(n)
```

would show the array `x` to have 100 elements: 1, 2, . . . , 100.

11.3.6 integer function `iffputstr()`

The `iffputstr()` function takes two arguments: the first is a character string (up to 128 characters) that names an IFEFFIT string (following the naming rules outlined in chapter 3, but with the leading '$' optional), and the second is a character string for the value. The effect is to set the named string with the given value (only the first 128 characters will be used – any remaining characters will be ignored). If the string already exists, it will be overwritten.

`iffputstr()` always returns 0.

```
character*128  txt
txt = 'Here is a string'
i = iffputstr('text1', txt)
i = ifeffit(' show @strings')
```

would show the string `$text1` to be 'Here is a string'.

11.3.7 integer function `iffgetstr()`

The `iffgetstr()` function takes two arguments: the first is a character string (up to 128 characters) that names an existing IFEFFIT string (following the naming rules outlined in chapter 3, but with the leading '$' optional), and the second is a character string variable to hole the value. The effect is to retrieve the named string with the given value. The string variable provided should be large enough to hold the result (128 characters is a safe value, as the returned value will never be larger than that).

`iffgetstr()` will return the real, useful length of the string.

```
character*128  txt
i = ifeffit(' set $text1 = "string test 1"')
n = iffgetstr('text1', txt)
print*, ' txt = ', txt(1:n)
```

11.3.8 integer function `iffgetecho()`

The `iffgetecho()` function takes one argument: a character string variable to hold the value returned. The effect is to retrieve the next element in the "echo buffer". Any remaining lines in the echo buffer will be shifted down (or popped, as it is often called), and the value of `&echo_lines` will be decreased by one. The echo buffer will only be filled if the variable `&screen_echo` is set to 0, which indicates that you intend to handle all text that would be sent to the screen yourself. Since any command may write to the echo buffer, it is recommended that you check the value of `&echo_lines` and retrieve all "echo"ed lines after each command.

`iffgetecho()` will return the real, useful length of the echo string.

```
      character*128  txt(32)
      double precision xnbuff
      integer nbuff, j
      i = iffgetsca('&echo_lines', xnbuff)
      nbuff = min(32,int(xnbuff))
      do 10 j = 1, nbuff
         il = iffgetecho(txt(j))
         print*, ' echo line ', j , ' = ', txt(j)(1:il)
10    continue
```

11.4 The C interface to IFEFFIT

Accessing IFEFFIT from a C program is very easy. The basic concepts and many of the details of the IFEFFIT application interface (or API for the programmers out there) given here also apply to using IFEFFIT from within scripting languages, as described later in this chapter. C++, by the way, is similar enough to C that calling IFEFFIT from it should be straightforward once the C interface is described. If you've read the previous section, you'll find that the C interface is also very similar to the Fortran interface.

The basic use of the IFEFFIT C interface is to send command strings to an "IFEFFIT engine" which acts just like an interactive IFEFFIT session run at the command prompt. The underlying engine has its own set of Program Variables that are kept in its own memory space, separate from the calling program. The session "stays alive" until the calling program ends. Though you could directly call any of the subroutines or functions in the IFEFFIT library, it is highly recommended that you **not** make such direct calls. Instead, you should use the functions provided in the application programming interface (API), as described here and encapsulated in the C include file *`ifeffit.h`* that can be found with the configuration files in the IFEFFIT distribution (typically in the */usr/local/share/ifeffit/config/* directory).

The IFEFFIT C API defines eight external functions, all of which are integer functions. To use these functions, you can simply put an `include` directive at the top of your program:

```
#!include "ifeffit.h"
int main() {
  int i;
  i = ifeffit(" ");
  i = ifeffit("read_data(cu.xmu, group=cu, type=xmu)");
  i = ifeffit("spline(cu.energy, cu.xmu, rbkg = 1.2)");
  i = ifeffit("plot(cu.k, cu.chi)");
}
```

This shows a very simple IFEFFIT session converted into a C program, using only the function `ifeffit()`, the main interface to the underlying IFEFFIT engine.

As this example shows, it is recommended that you first call `ifeffit()` with an "initialization string", typically a blank line, but optionally setting system configuration variables.

How do you actually build an executable out of this program file? That, of course, depends on details of your system. Most of the settings needed are put in the file *Config.mak* in the same location as *ifeffit.h*. This file contains Makefile instructions needed for linking your IFEFFIT application with the IFEFFIT library and all the other libraries needed to make an executable. An example Makefile (using the above code and the settings of Config.mak from a fairly normal linux system) is included in the *examples/scripting* section of the source distribution.

11.4.1 function `ifeffit()`

The `ifeffit()` function takes 1 argument that is a character string up to 1024 characters long (including any newline characters and the like) and returns an integer. The string **up to the first newline character** is interpreted and run as a command by the IFEFFIT engine. After the command has been fully processed, an integer is returned, indicating a return status according to Table 9. For backwards compatibility, the function `iff_exec()` has identical behavior to `ifeffit()`.

11.4.2 function `iff_put_scalar()`

The `iff_put_scalar()` function takes two arguments: the first is a pointer to a character string (up to 128 characters) that names an IFEFFIT scalar (following the naming rules outlined in chapter 3), and the second is a pointer to a double. The effect is to set the named scalar with the given double precision value. If the scalar already exists in the IFEFFIT engine, it will be overwritten. Note that this is equivalent to a `set()()` command, not a `def()()` command: for that, you should use the `ifeffit()` function itself.

`iff_put_scalar()` always returns 0.

```
double x, *px;
int i;
x = 3.00;
i = iff_put_scalar("kmin", &x);
x = sqrt(100.0);
i = iff_put_scalar("kmax", &x);
i = ifeffit(" show kmin, kmax");
```

would show the values to be 3.0 and 10.0, respectively.

11.4.3 function `iff_get_scalar()` and `iff_scaval()`

The `iff_get_scalar()` function takes two arguments: the first is a pointer to a character string (up to 128 characters) that names an existing IFEFFIT scalar (following the naming rules outlined in chapter 3), and the second is a pointer to a double. The effect is to retrieve the value of the named IFEFFIT scalar and put it into the provided C pointer. If the scalar does not exist in the IFEFFIT session, the value will be set to 0.

`iff_get_scalar()` always returns 0.

```
double *x;
x = calloc(1,sizeof(double));
i = ifeffit(' set var = sqrt(100.0)');
i = iff_get_scalar('var', x);
printf(" x = %g \n", *x);
```

would show x to have the value 10.0.

A more convenient version of this function is also available: The `iff_scaval()` function takes one arguments: the name of an existing IFEFFIT scalar, and returns a pointer to its double value. The above code could thus be rewritten as

```
double *x;
i = ifeffit(' set var = sqrt(100.0)');
x = iff_scaval('var');
printf(" x = %g \n", *x);
```

would show x to have the value 10.0.

11.4.4 function `put_string()`

The `put_string()` function takes 2 arguments to set the value of a text string Program Variable. The first argument is the *name* of the variable in the IFEFFIT name space, and the second is the value for the variable. Both the name and the variable itself are text strings (up to 128 characters long). To comply with the IFEFFIT naming rules, the variable name needs to begin with a $.

11.4.5 function `get_string()`

The `get_string()` function takes 1 arguments that is the name of a text string Program Variable in the IFEFFIT name space, and returns its value, which will be a text string up to 128 characters long (make sure to allocate enough memory!). To comply with the IFEFFIT naming rules, the variable name must begin with a $.

11.4.6 function `put_array()`

The `put_array()` function takes 3 arguments to set the value of an array Program Variable. The first argument is the *name* of the variable in the IFEFFIT name space. The second is the number of points (type `int*`) in the array, and the third is the array itself (type `double*`). To comply with the IFEFFIT naming rules, the variable name needs to contain a dot '.'.

11.4.7 function `get_array()`

The `get_array()` function takes 2 arguments to retrieve an IFEFFIT array Program Variable into an array (some languages call these lists) in the calling language. The first argument is the name of the variable in the IFEFFIT name space, and the second is the array itself (type `double*`). To comply with the IFEFFIT naming rules, the variable name needs to contain a dot '.'. The value returned by `get_array()` will be the number of points in the array.

11.4.8 function `get_echo()`

11.5 The IFEFFIT Perl Module

Perl is an Open-Source scripting language, available for free at http://www.perl.com/. It runs on every significant operating system. For those with programming (especially C) or Unix experience, Perl is an easy language to learn, and is especially good for processing text files and controlling processes. For those without much programming experience, Perl is still fairly easy to learn. Many excellent books[**?**, **?**] are available, and the support available on the web is excellent. Perl's flexibility and text-processing capabilities makes it very useful for many projects, including web-scripting.

To use IFEFFIT from within a perl program, you need the IFEFFIT Perl Module, which is a pluggable extension to perl included in the IFEFFIT distribution. Using the IFEFFIT module is fairly easy. Somewhere near the top of your perl script you put '`use Ifeffit;`' to tell perl that you want to use the IFEFFIT module. This provides perl with a function called `ifeffit` that takes a text string as an argument, sends that to the IFEFFIT engine and returns after the command has executed. Subsequent calls to `ifeffit` continue in the same IFEFFIT session, so that variables (arrays, scalars, strings, etc.) in IFEFFIT's memory can be accessed by later IFEFFIT commands. A simple script might look like this:

```
#!/usr/bin/perl -w
use Ifeffit;
$plot_command = "plot(my.x,my.y,color=blue)";
ifeffit(" my.x  = indarr(600) / 300 ");
ifeffit(" my.y  = 4 * exp(-my.x/5.) * cos(4*my.x - 70)");
ifeffit($plot_command);
```

The `ifeffit` function returns an integer, which is normally 0 for 'success'. If the command appears to be an incomplete line (that is, the line is expected to be continued), `ifeffit` returns -1. If an 'exit' has been sent to `ifeffit`, it returns 1, and if a serious error occurs within IFEFFIT, a value greater than 1 is returned.

In addition to sending commands as text strings to the IFEFFIT function, you can also directly access the scalars, arrays, and text strings in IFEFFIT's store. There are six additional functions – one for each of "set" and "get" of the three data types: `get_scalar`, `get_string`, `get_array`, `put_scalar`, `put_string`, and `put_array`. To use these functions, you'll need to tell perl you want to use these functions (in keeping with perl's custom, the default behavior is to *not* provide loads of function names without explicitly asking for them) with

```
use Ifeffit ;
use Ifeffit qw(get_scalar get_string get_array);
use Ifeffit qw(put_scalar put_string put_array);
```

With these declarations, you can now put something like

```
ifeffit(" read_data(my.xmu, type = xmu)");
ifeffit(" spline(my.energy, my.xmu, rbkg = 1.0)");
$e0   = get_scalar("e0");
$rbkg = get_scalar("rbkg");
$perl_string = "Spline  Rbkg = $rbkg    E0 = $e0\n";
put_string("title1", $perl_string)
ifeffit(" write_data(file=my_out.chi,$title1, my.k, my.chi)");
```

in your perl script. After reading $\mu(E)$ data and doing a background subtraction, this script got the values of $R_{\rm bkg}$ and E_0, wrote those values into a perl text string and then passed that string directly back into IFEFFIT. Since IFEFFIT has essentially no string processing capabilities, this is a good way to write a decent title line for an output file. Finally, this script told IFEFFIT to write an output file for $\chi(k)$ using this newly-formed title line.

If the IFEFFIT perl module is installed on your system, more complete documentation is available by typing "`perldoc Ifeffit`" at a command-line prompt.

11.6 Using IFEFFIT from Python

Like Perl and Tcl, Python is an Open-Source scripting language that runs on every significant operating system and is available for free at http://www.python.org/. Python is easy to learn, and is growing in popularity, especially for scientific programming. It's supports the Tk toolkit on Unix, Windows, and Mac, and even has some 2d-plotting capabilities builtin.

To use IFEFFIT from within Python, you'll need to make the IFEFFIT extension to Python (consult the installation documentation). Once that's done, you can import the Ifeffit module to get the same sort of functions described above for Perl and Tcl. That is, there is an `ifeffit` function that takes a command string argument and returns an integer. There are also functions `get_scalar`, `get_string`, `get_array`, `put_scalar`, `put_string`, and `put_array` for copying each of the three data types back and forth between the underlying IFEFFIT session and the Python script itself.

```
#!/usr/bin/python
import Ifeffit
iff = Ifeffit.Ifeffit()
iff.ifeffit( "read_data(my.xmu, group=my, type = xmu)")
iff.ifeffit( "spline(my.energy, xmu = my.xmu, rbkg = 1.0)")
e0   = iff.get_scalar("e0")
rbkg = iff.get_scalar("rbkg")
str  = "Spline  Rbkg = %f8.2  E0 = %f9.2" % (rbkg, e0)
iff.put_string("title1", str)
iff.ifeffit( "write_data(file = my_out.chi, my.k, my.chi)")
```

11.7 Using IFEFFIT from Tcl

Tcl[**?**] is an Open-Source scripting language, available for free at http://www.scriptics.com/. Like Perl, it runs on every significant operating system, and is especially popular when used with it's exceptional Tk widget set for building cross-platform GUIs. The Tcl syntax is fairly simple, and many books and web-sites are devoted to it.

To get access to the IFEFFIT functionality from within Tcl, you'll need to make the IFEFFIT extension to Tcl for your system. Once that's done (see the installations instructions for details), you can "source" the Ifeffit.tcl file to get the same sort of functions described above for Perl. That is, there is an `ifeffit` function that takes a command string argument and returns an integer. There are also functions `get_scalar`, `get_string`, `get_array`, `put_scalar`, `put_string`, and `put_array` for copying each of the three data types back and forth between the underlying IFEFFIT session and the Tcl script itself.

A Tcl script using IFEFFIT might look like this:

```
source Ifeffit.tcl
```

```
ifeffit   "read_data(my.xmu, group=my, type = xmu)"
ifeffit   "spline(my.energy, xmu = my.xmu, rbkg = 1.0)  "
set e0         [ get_scalar "e0" ]
set rbkg       [ get_scalar "rbkg" ]
set tcl_string  "Spline  Rbkg = $rbkg    E0 = $e0"
put_string $title1  $tcl_string
ifeffit   "write_data(file = my_out.chi,  my.k, my.chi)"
```

A Glossary of Program Variables

This appendix describes and lists the Program Variables IFEFFIT expects you to use. It is a compilation of the Program Variables used as default input and output from the commands listed in chapter 9. It also describes common conventions used for the naming of arrays generated by the commands.

A.1 Scalar Naming Conventions

The tables below list the commonly used scalars and text strings Program Variables, giving a very brief description of the expected meaning for the variables and the commands that use this variable either as input or output. More details about the meaning and use of these variables can be found in the entry for each command.

General and System Variables

Program Variable	Description	Used in Commands
`$column_label`	data column label	`read_data()`
`$commentchar`	comment character	`read_data()`, `write_data()`
`&echo_lines`	number of lines to `echo()`	`echo()`, `load()`
`$filename`	file name for input	`read_data()`
`$group`	default group name	many
`$history_file`	name command history file	`history()`
`$&install_dir`	installation directory	`f1f2()`
`&pause_ignore`	ignore `pause()` commands	`pause()`
`&screen_echo`	screen echo setting	`echo()`, `log()`, `macro()`
`&sync_level`	synchronization level	`save()`, `set()`, `show()`
`$1` ... `$9`	macro arguments	at macro execution

Plotting

Program Variable	Description	Used in Commands
`cursor_x`	cursor x coordinate	`cursor()`, `zoom()`
`cursor_y`	cursor y coordinate	`cursor()`, `zoom()`
`$plot_device`	plotting device	`plot()`
`$plot_file`	filename for hardcopy plot	`plot()`
`$plot_title`	title for plot	`plot()`
`$plot_xlabel`	label for x-axis	`plot()`
`$plot_ylabel`	label for y-axis	`plot()`

FEFF Paths

Program Variable	Description	Used in Commands
`$feff_file`	name of FEFF file	`path()`
`path_index`	path index	`path()`
`reff`	R of a FEFF path	`get_path()`

Pre-Edge and Spline

Used in commands `pre_edge()`, and `spline()`

Program Variable	Description	Used in Commands
`e0`	E_0, where $k = 0$	
`edge_step`	$\Delta\mu(E_0)$, edge step	
`norm_c0`	normaliztion parameter	
`norm_c1`	normaliztion parameter	
`norm_c2`	normaliztion parameter	
`norm1`	normalization range	
`norm2`	normalization range	
`norm_order`	order of normalization polynomial	
`pre_offset`	offset of pre-edge line	
`pre_slope`	slope of pre-edge line	
`pre1`	pre-edge range	
`pre2`	pre-edge range	
`rbkg`	$R_{\rm bkg}$ for XAFS spline	`spline()`

(see also Fourier transform parameters for `spline()`)

Fourier Transform

Used in commands `chi_noise()`, `feffit()`, `fftf()`, and `fftr()`

Program Variable	Description	Used in Commands
`$altwindow`	alternate window array	
`dk1`	dk_1	
`dk2`	dk_2	
`dr1`	dR_1	
`dr2`	dR_2	
`kmax`	$k_{\rm max}$	
`kmin`	$k_{\rm min}$	
`kweight`	k-weight w	
`kwindow`	$k \rightarrow R$ window type	
`rmax`	$R_{\rm max}$	
`rmin`	$R_{\rm min}$	
`rwindow`	$R \rightarrow q$ window type	
`dk1_spl`	dk_1	`spline()`
`dk2_spl`	dk_2	`spline()`
`kmax_spl`	$k_{\rm max}$	`spline()`
`kmin_spl`	$k_{\rm min}$	`spline()`
`kweight_spl`	k-weight w	`spline()`

Fitting

Program Variable	Description	Used in Commands
`chi_square`	fit statistic	`feffit()`, `minimize()`
`chi_reduced`	fit statistic	`feffit()`, `minimize()`
`epsilon_k`	ϵ_k, uncertainty in $\chi(k)$	`chi_noise()`, `feffit()`
`epsilon_r`	ϵ_R, uncertainty in $\tilde{\chi}(R)$	`chi_noise()`, `feffit()`
`&fit_iteration`	number of fit interations	`feffit()`, `minimize()`
`$fit_space`	fitting space	`feffit()`
`n_idp`	N_{idp}	`feffit()`
`n_varys`	number of variables	`feffit()`, `minimize()`
`r_factor`	fit statistic	`feffit()`

A.2 Array Naming Conventions

Arrays created by commands will have predictable names, and depend only on the current group name, held in the string variable `$group`, or specified as a command argument. By convention, the prefix of the array name is the group name, and the suffix of the array name contains a meaningful identification for the data. The following table lists the commonly used suffixes, their meaning, and commands that use or create arrays with these suffixes.

Array suffix	Description	Used in Commands
`energy`	E	none
`xmu`	μ	none
`pre`	pre-edge subtracted μ	`pre_edge()`, `spline()`
`norm`	normalized `pre`	`pre_edge()`, `spline()`
`bkg`	background μ_0	`spline()`
`k`	k	`spline()`, `fftf()`, `feffit()`
`chi`	χ	`spline()`, `fftf()`, `feffit()`
`win`	W_k (k-window function)	`fftf()`, `feffit()`
`chir_mag`	$\lvert\chi(R)\rvert$	`fftf()`
`chir_re`	$\mathrm{Re}[\chi(\mathrm{R})]$	`fftf()`
`chir_im`	$\mathrm{Im}[\chi(\mathrm{R})]$	`fftf()`
`chir_phase`	$\mathrm{phase}[\chi(\mathrm{R})]$	`fftf()`

B Fourier Transforms in IFEFFIT

This appendix describes and lists the conventions used for Fourier transforms in IFEFFIT.

B.1 Fourier transform Conventions

Many of IFEFFIT's command use Fourier transforms (FT) to perform their tasks. In addition to `fftf()` and `fftr()`, which are principly designed to do Fourier transforms, the commands `chi_noise()`, `feffit()`, and `spline()` all do (or can do) Fourier transforms as part of their data processing. The form of the Fourier transform done by all these commands is the same, and is really an XAFS-specific Fourier transform that converts $\chi(k)$ into $\tilde{\chi}(R)$ in the forward direction and $\tilde{\chi}(R)$ into $\tilde{\chi}(k)$ in the reverse direction. The XAFS-specific FT done by these commands will be described in detail shortly. For now, an important point to emphasize is that all these commands share many arguments and grogram variables describing the Fourier transforms. These shared command arguments and program variables are the topic of this section.

The forward XAFS Fourier transform, done with `fftf()`, transforms $\chi(k)$ to $\tilde{\chi}(R)$. To do this, the $\chi(k)$ data is first multiplied by a k-weighting factor of the form k^w and a window function before the actual Fast Fourier transform is performed. The k-weighting factor w is used to "even out" the decaying $\chi(k)$ function and to emphasize different k-regions of the EXAFS in the resulting $\tilde{\chi}(R)$. Popular choics for w are 1, 2, and 3.

Formally, the XAFS Fourier transform can be written as

$$\tilde{\chi}(R) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dk e^{i2kR} k^w \tilde{\chi}(k) \Omega(k) \tag{4}$$

where $\Omega(k)$ is the window function, and w is the k-weighting factor. The window function can take a variety of functional forms, all of which rise from a small value (possible zero) at low-k, rise up to one, and then fall back towards zero at high-k. The window is intended to smooth out any ringing in the resulting FT amplitude while maintaining as much resolution as possible, and will be discussed in more detail in the next section.

A discrete form of the above formula is actually used so that the Fast Fourier Transform algorithm can be exploited. The key point here is that the data is sampled on a finite and *uniform* grid in k (or R for the back-transform). The k-space grid used throughout IFEFFIT is $\delta k = 0.05\,\text{Å}^{-1}$. The array sizes for $\chi(k)$ and $\tilde{\chi}(R)$ are $N_{\text{fft}} = 2048$, and the data is *zero-padded* out to high-k (or high-R). The zero-padding for $\chi(k)$ will smooth the data points in R-space, and the zero-padding of $\tilde{\chi}(R)$ will smooth the data in backtransformed-k-space. The grid in R-space is $\delta R = \pi / N_{\text{fft}}\,\delta k$, which is then ~ 0.0307 Å.

For the discrete Fourier transforms, we write $k_n = n\delta k$ and $R_m = m\,\delta R$, and have

$$\tilde{\chi}(R_m) = \frac{i\delta k}{\sqrt{\pi N_{\text{fft}}}} \sum_{n=1}^{N_{\text{fft}}} \chi(k_n)\,\Omega(k_n)\,k_n^w e^{2\pi i n m / N_{\text{fft}}} \tag{5}$$

for the forward transform and

$$\tilde{\chi}(k_n) = \frac{2i\delta R}{\sqrt{\pi N_{\text{fft}}}} \sum_{m=1}^{N_{\text{fft}}} \tilde{\chi}(R_m)\,\Omega(R_m)\,e^{-2\pi i n m / N_{\text{fft}}} \tag{6}$$

for the back transform. These normalizations preserve the symmetry properties of the Fourier Transforms with conjugate variables k and $2R$.

There are few slight complication with these formulas. The first arises from the fact that because the classic EXAFS equation has a term of the form e^{i2kR} or $\sin(2kR)$, it customary to use k and $2R$ as the Fourier conjugate variables while still desiring the R space function to be a function of R. This changes the normalization factors in front of the integral to those above.

The other minor complication is that the "measured" $\chi(k)$ derived from $\mu(E)$ and is a strictly real function while the Fourier transform inherently treats $\chi(k)$ as complex functions, signified by the˜above the χ). There is an ambiguity about how to construct the complex $\tilde{\chi}(k)$. In many formal treatments, the measured XAFS is written as the imaginary part of some function, so that constructing $\tilde{\chi}(k)$ as $(0, \chi_{\rm measured}(k))$ might seem a natural choice. For historical reasons, IFEFFIT uses the opposite convention, constructing $\tilde{\chi}(k)$ as $(\chi_{\rm measured}(k), 0)$. You can easily override this default however and do transforms assuming $\chi(k)$ is the imaginary part of $\tilde{\chi}(k)$. Normally, one does a forward transform with

```
Iff> fftf(real = data.chi)
```

which sets $\tilde{\chi}(k)$ as $(\chi_{\rm data}(k), 0)$. You can use

```
Iff> fftf(imag = data.chi)
```

to construct $\tilde{\chi}(k)$ as $(0, \chi_{\rm data}(k))$.

The Fourier transform requires that the $\chi(k)$ data begin at $k = 0$. More to the point, the `fftf()` command *assumes* that the supplied array for *chi* starts at $k = 0$ unless told otherwise. It is important to include the k-array with this keyword. If not given, the χ array will be assumed to have it's first point be $\chi(k = 0)$, and then to be input on an even k-grid with spacing 0.05 Å.

B.2 Fourier transform window functions

There are seven optional forms for the Fourier transform window $\Omega(k)$. There is quite a bit of literature on the different windows, and generally more opinion than justified reason for selecting one window function over others. I believe that all the window functions in IFEFFIT are appropriate and useful for EXAFS analysis. My recommendation is to pick one function and stick with it. If you're unsure about which one to pick, my favorites are the Hanning window (the default in IFEFFIT, largely for historical reasons) and the Kaiser-Bessel window. Again, I have not seen any objective rational for preferring any other windows, and the choice is really a matter of taste.

The available window functions are described below, first in the table giving a brief description, then with an equation for the window function, and finally with a representative plot. For simplicity, all are written as functions of k. The R-space windows are exactly analogous with `k` replaced by `r`.

`altwindow` names an alternative window **array** is named with this keyword, overriding the 'normal' window. The array specified must be created before invoking `feffit`.

Table 10: Table of Fourier Transform Window Functions. The first four windows list ramp up from 0 to 1 over a k-range defined by the `dk` parameter, stay at 1 for some k-range, and then drop back down to zero. The final three window functions apply a continuous function that may never go to zero over the entire k-range of the window. For each window type, the Key in the second column gives the value to use for the `kwindow` parameter of `fftf()`, or the `rwindow` parameter of `fftr()`.

Window Name	Key	Description
Hanning	`hanning`	ramps up and down as $\cos^2(k)$
Parzen	`parzen`	ramps up and down linear with k
Welch	`welch`	ramps up and down linear with k^2
Sine	`sine`	a Sine function over the full k-range
Gaussian	`gaussian`	a Gaussian function over the full k-range
Kaiser-Bessel	`kaiser`	a modified Bessel function over the full k-range

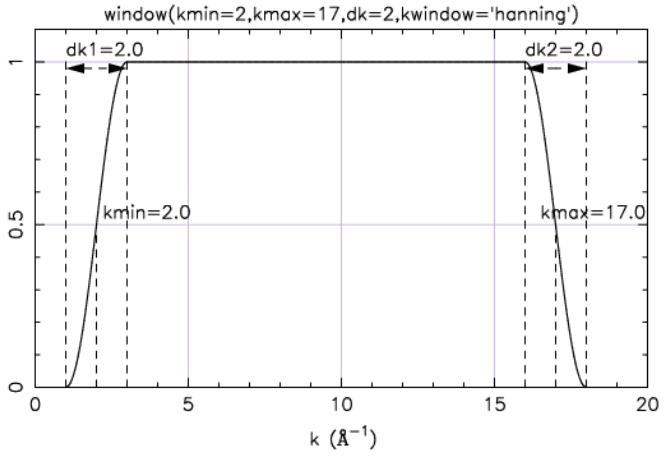

Figure 5: Anatomy of the Hanning Window.

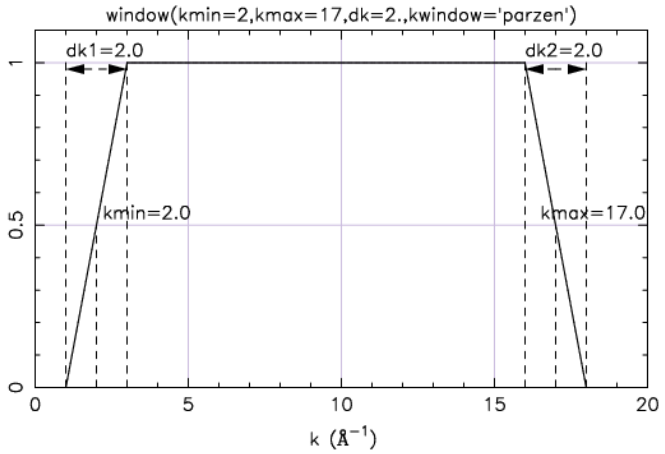

Figure 6: Anatomy of the Parzen Window.

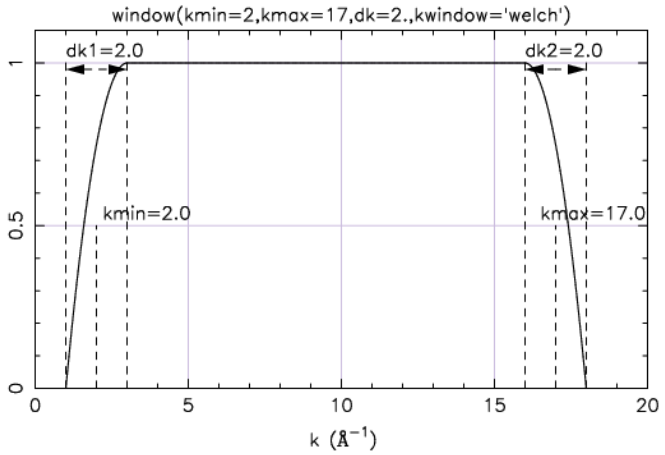

Figure 7: Anatomy of the Welch Window.

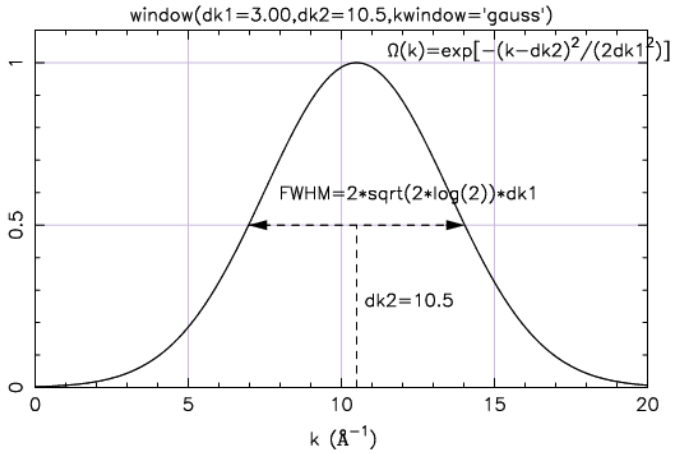

Figure 8: Anatomy of the Gauss Window.

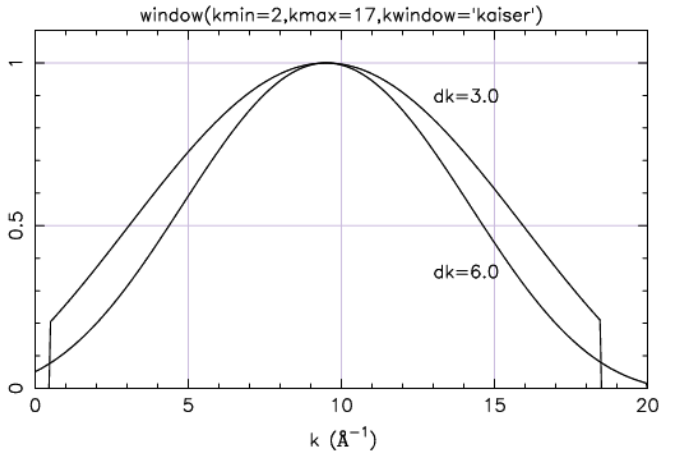

Figure 9: Anatomy of the Kaiser Window.

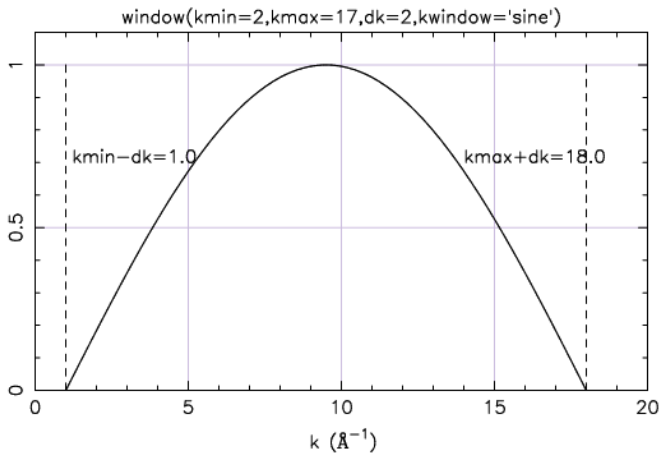

Figure 10: Anatomy of the Sine Window.