Tutorials and worked examples for simulation, curve fitting, statistical analysis, and plotting.
http://www.simfit.org.uk

In the unlikely event that the model you want to plot, simulate or fit is not in the SIMFIT library of compiled models you will have to create a user–defined–model file. Actually, once the format for user–defined model equations is understood, it is very simple to create model files using any text editor. However, program **usermod** can be opened using the [A/Z] option on the SIMFIT main menu, and this is an easy way to create models and understand how to use them. It provides these options.

- Open a file from the selection of demonstration examples provided using the [Demo] button on the File-Open dialogue.
- View the model files supplied to examine the way these test examples formulate the various models and sets of models.
- Use the options provided to check the models or use them to plot, integrate, locate zeros, or perform constrained optimization.
- Invoke the facility to open a user-defined file or a default template in a text editor to create a model file, then save it to a temporary file and read it back into the main program to check, etc. until the syntax is correct before finally archiving the model.

From SIMFIT version 7.1.6 onwards there are two procedures that can be used to define models.

1. **Reverse Polish**
   This is the most versatile technique, but it is very verbose and may be found difficult to use by non-programmers.
2. **Standard mathematical expressions**
   This is far easier to understand by scientists, and is recommended when developing new models since it uses normal mathematical formulas but then SIMFIT automatically transforms them into reverse Polish at run time.

The most versatile manner is to use standard mathematical expressions for simple one-line equations, but to resort to a mixture of standard expressions linked by permitted SIMFIT reverse Polish stack operations for complicated models.

It should be pointed out that models in the compiled library are protected against critical events such as taking logs or square roots of negative numbers, or dividing by zero, but it is your responsibility when fitting your own user–defined models that the parameter values and independent variables are constrained to avoid such singularities.

Reverse Polish

Reverse Polish (i.e. last–in–first–out, or post–fix) is an excellent way to prepare user–defined models for a number of reasons.

1. It can be used to express any mathematical model unambiguously.
2. It is very similar to the way that computers perform calculations.
3. It is the way that the PostScript language and programmable calculators work.
4. There many model files distributed with SIMFIT to demonstrate the technique and documentation to explain it.

5. **usermod** can be used to view these then develop your own models, check them for correct syntax, then use them for plotting, calculating integrals, finding zeros of functions, fitting, or optimization.

As a simple example to introduce this technique, consider how to formulate damped simple harmonic motion, namely

$$f(x) = \delta \exp(-\gamma t) \cos(\alpha t - \beta)$$

with

$$\begin{aligned} \text{Frequency } \alpha &= p(1) \\ \text{Offset } \beta &= p(2) \\ \text{Decay constant } \gamma &= p(3) \\ \text{Amplitude } \delta &= p(4), \text{ and} \\ \text{Independent variable } t &= x. \end{aligned}$$

Here, for example, is the test file `usermod1.tf9` to simulate damped simple harmonic motion with these definitions

```
%
f(x) = p(4)*exp[-p(3)*x]*cos[p(1)*x - p(2)]
%
1 equation
1 variable
4 parameters
%
p(1)
x
multiply
p(2)
subtract
cosine
p(3)
x
multiply
negative
exponential
multiply
p(4)
multiply
f(1)
%
```

It consists of three distinct sections, separated by percentage signs as follows.

Section 1 Up to 24 lines containing any information with no formatting restrictions.

Section 2 The number of equations, variables (or else the phrase differential equation), and parameters.

Section 3 The reverse Polish commands.

In addition to browsing the library of models available using the [View] option from the main SimFIT menu, and examining the tutorial documents on the SimFIT website, such as `user_defined_models.html`, there are several additional sources of help.

- The readme files `w_readme.f4`, `w_readme.f5`, ... `w_readme.f10` also available from the [View] option on the main SimFIT menu.
- The SimFIT reference manual `w_manual.pdf` available from the [Manual] option on the main SimFIT menu.

- The help sections provided by program **usermod** which can be opened from the [A/Z] option on the main SimFIT menu.

It is recommended that the first few user-defined model files, i.e. `usermod1.tf1`, `usermod1.tf2`, and `usermod1.tf3` be examined carefully as they contain appended sections containing an analysis of the way the reverse Polish model definition scheme works.

Standard expressions

The usual mathematical notation can be used at any point in a model file provided that it is included in a `begin{expression}... end{expression}` structure like the following code for defining a cubic.

```
begin{expression}
f(1) = p(1) + p(2)x + p(3)x^2 + p(4)x^4
end{expression}
```

The essential rules for using expressions are now listed.

- Parameters must be indexed as `p(i)` where the index `i` is consistent with the header defining the number of parameters.
- The independent variables must be consistent with these rules.

  1 independent variable: x

  2 independent variables: x, y

  3 independent variables: x, y, z

  n independent variables: $y(1), y(2), \ldots, y(n)$
- The only other symbols that can be used are dummy variables (such as $A = x^2 + y^2$, etc. to avoid repetition as discussed below) and those listed for for use with SimFIT reverse Polish, e.g.
  `cos, sin, tan, exp, log, log10, pi, sqrt`, etc.
  with the constants and special functions described in `commands.txt` and `w_manual.pdf`.
- Successive functions must be indexed as `f(j)` where the index `j` is consistent with the header defining the number of equations.
- Make liberal use of the `*` sign for multiplication, and employ brackets `{.}, [.], (.)` to avoid ambiguities, for example:

  Use `x*log(x)` instead of `xlogx`

  Use `1/(2pi)` instead of `1/2pi`

  Use `exp(-kt)` instead of `e^-kt`
- It is possible to spread equations over several lines in an expression construct but blank lines should not be used anywhere in the model defining section of a user–defined model file.
- When the expression has been evaluated the result is added to the top of the stack so it can be left there, duplicated, stored, or used immediately to define a function value, which would pop the value off the stack and return the function value for use by the calling procedure.

It should be pointed out that when SimFIT reads a user-defined model file it creates a temporary copy called `f$parser.tmp` in your temporary file folder where equations defined in expression constructs are parsed and written out in reverse Polish. For this reason it is important to realize the need to examine this temporary file if the model does not compute properly, or crashes with error messages due to unrecognizable elements in expressions which have been written unchanged into the reverse Polish code.

Here is `usermod1_e.tf9` defining the previous model using a standard mathematical expression. Note the use of `_e` to indicate a model using expressions. It should be noted how the command `f(1)` is used to define the function.

```
%
f(x) = p(4)exp[-p(3)x]cos[p(1)x - p(2)]
%
1 equation
1 variable
4 parameters
%
begin{expression}
f(1) = p(4)exp(-p(3)x)cos(p(1)x - p(2))
end{expression}
%
```

Here is another example of the model file `d01faf_e.mod` that is much more succinct using the expression technique rather than reverse Polish

```
%
f(y) = {4y(1)y(3)^2[exp(2y(1)y(3))]}/{1 + y(2) + y(4)}^2
%
1 equation
4 variables
0 parameters
%
begin{expression}
f(1) = 4y(1)y(3)^2[exp(2y(1)y(3))]/[1.0 + y(2) + y(4)]^2
end{expression}
%
```

Here is an example with nine functions of nine variables `c05nbf_e.mod` which is much easier to understand than the reverse Polish version.

```
%
f(1)=(3-2x(1))x(1)-2x(2)+1, ..., f9=-x(8)+(3-2x(9))x(9)+1
%
9 equations
9 variables
0 parameters
%
begin{expression}
f(1) = (3 - 2y(1))y(1) + 1 - 2y(2)
f(2) = (3 - 2y(2))y(2) + 1 - y(1) - 2y(3)
f(3) = (3 - 2y(3))y(3) + 1 - y(2) - 2y(4)
f(4) = (3 - 2y(4))y(4) + 1 - y(3) - 2y(5)
f(5) = (3 - 2y(5))y(5) + 1 - y(4) - 2y(6)
f(6) = (3 - 2y(6))y(6) + 1 - y(5) - 2y(7)
f(7) = (3 - 2y(7))y(7) + 1 - y(6) - 2y(8)
f(8) = (3 - 2y(8))y(8) + 1 - y(7) - 2y(9)
f(9) = (3 - 2y(9))y(9) + 1 - y(8)
end{expression}
%
```

Using dummy variables in standard mathematical expressions

A common feature required when creating models with long or complicated expressions is to define dummy intermediate values to improve readability as in this example for a rational function

```
begin{expression}
alpha = p(1)x + p(2)x^2
beta = 1 + p(3)x + p(4)x^2
f(1) = alpha/beta
end{expression}
```

or, more usually, to save intermediate values for repeated re-use as in this example for `optimum_e.mod`.

```
%
Rosenbrock's 2-dimensional function for optimisation
f(1) = 100(y - x^2)^2 + (1 - x)^2
f(2) = d(f(1))/dx = -400x(y - x^2) - 2(1 - x)
f(3) = d(f(1))/dy = 200(y - x^2)
%
3 equations
2 variables
0 parameters
%
begin{expression}
A = y - x^2
B = 1 - x
f(1) = 100A^2 + B^2
f(2) = -400A - 2B
f(3) = 200A
end{expression}
```

Note the use of **A** and **B** dummy variables to avoid re-calculations. During model evaluation this method implements the SIMFIT `get(.)` and `put(.)` reverse Polish commands to archive and retrieve dummy variables.

This technique is especially useful when formulating Jacobians for systems of differential equations, as there are frequently common expressions between the family of differential equations and their Jacobians.