OpenModel

User Guide

Neil Crout
School of Biosciences
University of Nottingham
UK

www.openmodel.info

2.4.2; 29 June 2016

Table of Contents

1. What is OpenModel

OpenModel is an integrated package designed to allow the specification, solution, visualisation, parameterisation and numerical analysis of certain classes of mathematical models. It is well suited to the solution of iterative dynamic models which can be specified as a combination of ordinary differential equations and simple assignments. However it has been applied to a range of different model types.

Its key features are:

- Modular structure
- Specification of model equations via a simple script
- Tabular, graphical and spatial outputs
- Model parameterisation (variety of methods: Marquardt, Metropolis-Hastings, Great Deluge)
- Uncertainty estimation
- Model Averaging
- Model Simplification or Reduction

2. About this user guide

This guide illustrates the features of OpenModel through a combination of step by step instructions and reference information. Probably the user guide is incomplete in that it does not explain all the features of OpenModel, we aim improve it as and when we discover defects or omissions.

The installation files include a number of models which illustrate various aspects of OpenModels application. You may find these more informative than the guide itself.

This guide is supplemented by an example model guide. This provides further step by step examples and overviews of model files supplied with the OpenModel installation.

3. Installation

OpenModel is downloaded as a windows msi file. Simply run this file and it will install to a folder of your choosing (by default it will install to **`c:\program files\University of Nottingham\OpenModel\`**).

The OpenModel installation includes the FreePascal compiler, FPC (www.freepascal.org/) which is used to compile the calculation scripts generated by OpenModel. Although FPC is available for a variety of different platform OpenModel is limited to i386/windows.

This procedure has been tested on Windows XP and Windows 7 systems.

The OpenModel installer will create:

- An Uninstall option in the OpenModel entry in the start menu.
- A default folder for OpenModel files (within your My Documents folder) called **`\OpenModel Files`**.
- Several example files within the **`\OpenModel Files`** folder.

Note: When you 'run' a model in OpenModel it will create files in a subfolder beneath the model file folder called **`OpenModelCache`**. If you do not have write access to this folder this will fail (e.g. on a network). To avoid this, simply create your model files in an appropriate location.

4. Your First Model: Step by Step Tutorial

4.1 Overview

OpenModel allows models to be specified using four different types of object.

- Modules

  Modules are the main building blocks of OpenModel models. They allow a related set of equations to be specified (assignments and differential equations) using a fairly standard mathematical syntax. Modules can call one another to accommodate the potentially complex structure of some models.

- Parameterisations

  Parameterisations enable sets of model parameter values to be integrated with information such as parameter constraints, error or prior distributions. A model can have several parameterisations defined. This enables the user to switch easily between different applications of the model.

- Data sheets

  Data sheets provide a means for external data to be used in a model (i.e. as a lookup table) or to be used for comparison to calculated values (i.e. to compare predictions to observations).

- Grids

  Grids provide a means to connect OpenModel models to external sources of spatial data, held as regular 2-dimensional grids.

The specification, evaluation and parameterisation of a simple model is described below in a step by step tutorial.

4.1 Creating a simple model

For the purposes of this tutorial we shall work with a very simple example model which describes, rather naively, the changes in the concentration of a contaminant in a lake system with time.

Lets imagine that we have

$$\frac{dL_1}{dt} = -k_1 L_1 + k_2 L_2$$

$$\frac{dL_2}{dt} = k_1 L_1 - k_2 L_2$$

where $L_1$and L_2 are the amount of contaminants in lakes 1 and 2 and k_1 and k_2 are rate coefficients (values 0.01 and 0.02 per unit time).

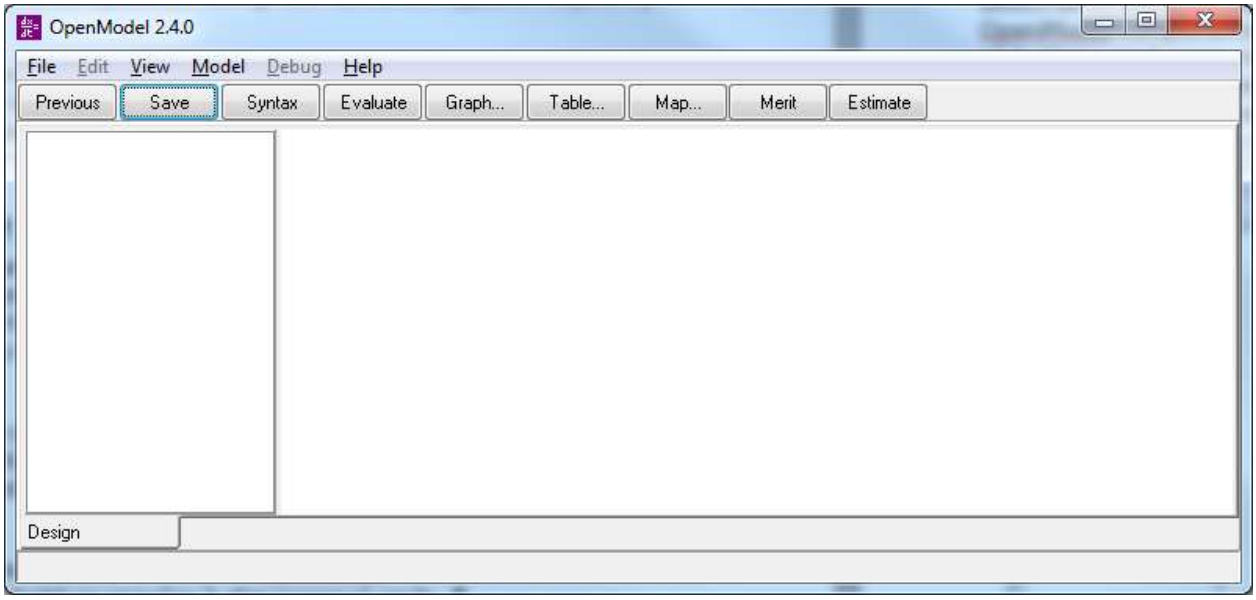
Lake 1 becomes contaminated at t=0 with 10kg of contaminant, lake 2 is initially uncontaminated. So we have initial conditions for the differential equations of L_1=10 and L_2=0.

We will now go through the steps required to specify this model using OpenModel.

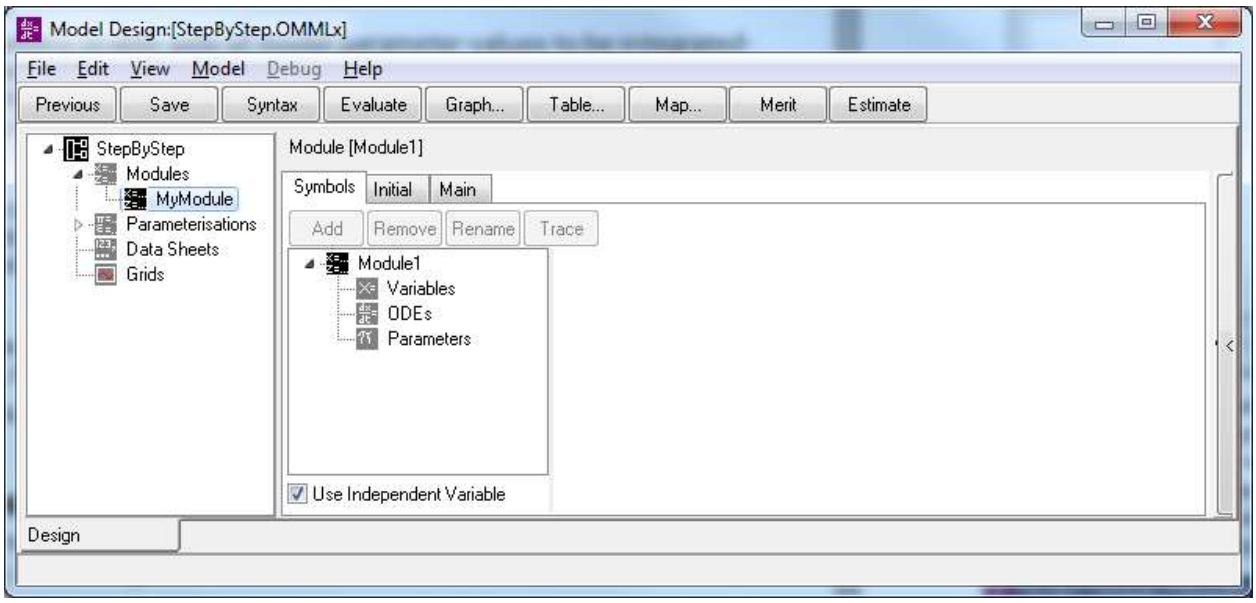
4.2 First steps: Creating a module

When OpenModel is started you will see a window such as the one shown below. On the left hand side is a tree view. This is the OpenModel navigation tree. It appears in a

variety of different guises throughout OpenModel and allows you to navigate and select different OpenModel objects. In the screenshot below the tree view is empty as no OpenModel components have been defined. Across the top of the window are a number of buttons and menu items whose use we will introduce as we go through the tutorial. The main workspace on the right hand side is where the selected OpenModel object is displayed (in this screenshot the model is empty so the workspace is empty).



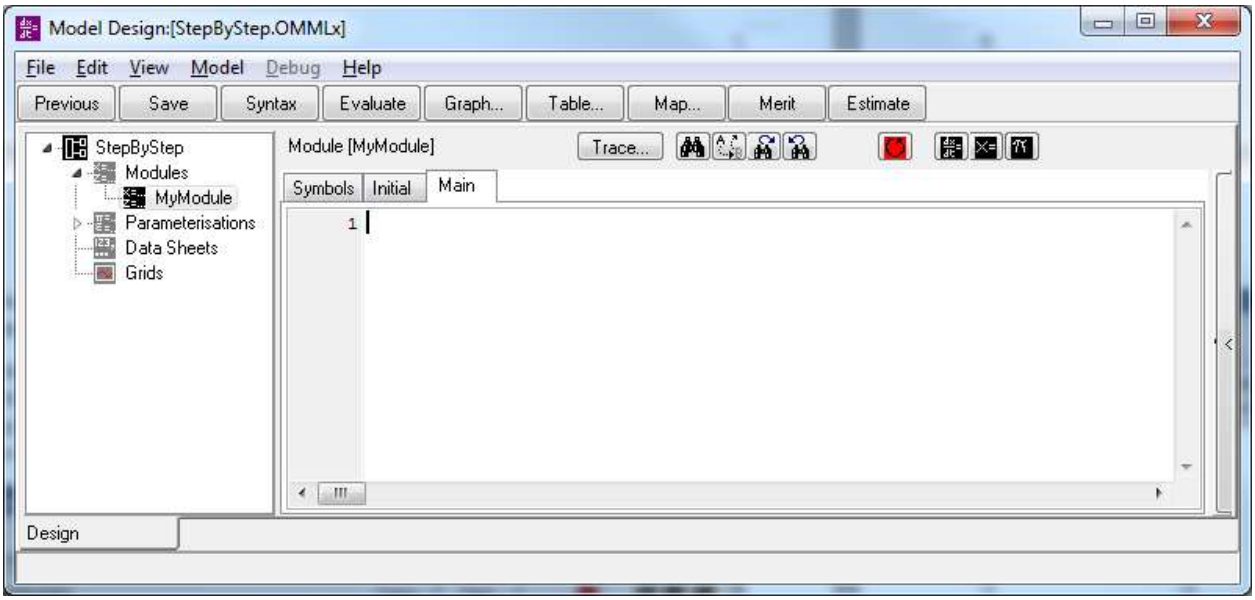
The first step is to create a new model within OpenModel. To do this select File|New from the menu, and an empty model will be created and displayed within OpenModel. In our example we have then selected File|SaveAs from the menu and given the file the name 'StepByStep'. The default file extension is '.OMMLx'. If you do this the display will be as shown below. The installation includes an example model StepByStep.OMMLx so you may wish to choose a different name to avoid overwriting the supplied file. The navigation tree on the left hand side of the window now shows the various OpenModel objects within the model.



In order to specify our model we need to add a module. You can add any number of inter-linked modules to an OpenModel model enabling complex structures to be created. However our example model can be specified quite simply using a single module.

To add a module select the module node in the navigation tree, right click and select Add Module from the pop up menu.

A module with a default name will be added to the model. It will appear in the navigation tree and when you select it in the tree it will be displayed as an initially empty module in the OpenModel workspace. In the example below we have first renamed it (by right clicking the module node in the navigation tree) to '**MyModule**'.



4.3 The module view

The module view is the area to the right of the OpenModel window. The module name is given across the top of the module view and the specification of the module can be viewed through the 3 tabbed pages.

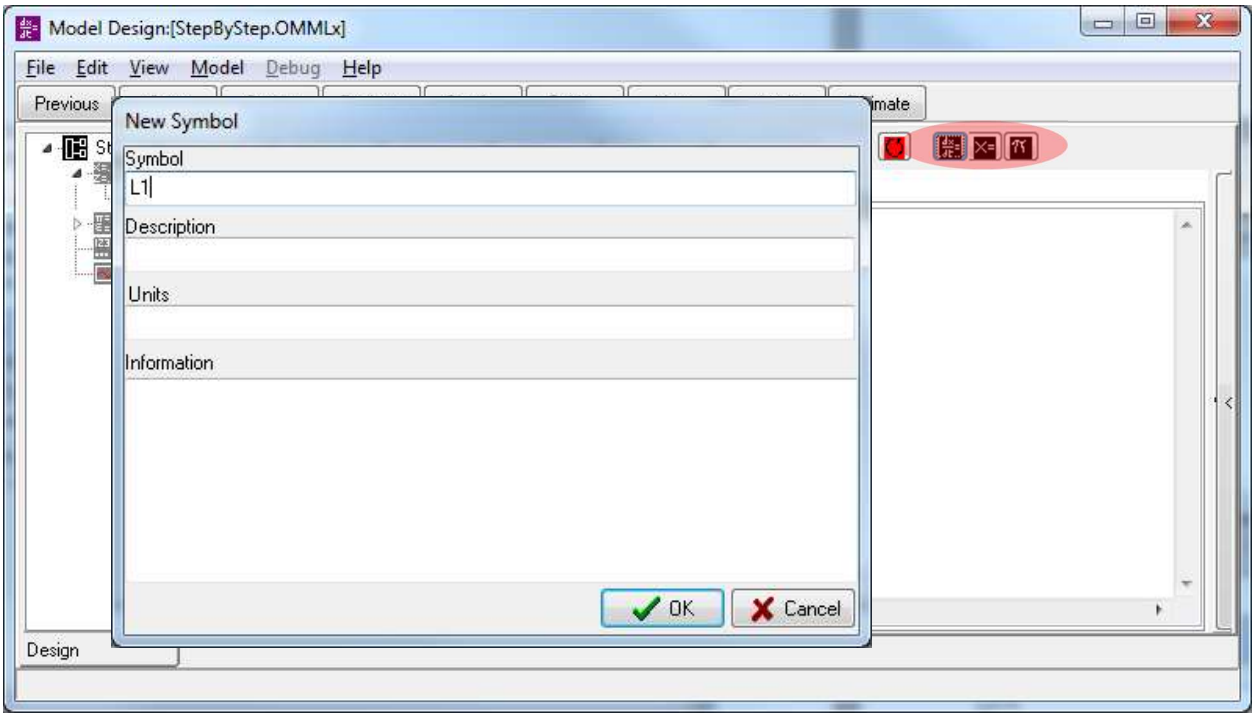
- Symbols. This defines the 'symbols' used within the module. These may be constants (i.e. parameters) or values calculated during the execution of the module (either directly assigned variables or ordinary differential equations).
- Initial. This is a sequence of model calculations which will be executed just once at the start of the module evaluation.
- Main. This is the models main sequence of calculations and statements which are executed for each iteration of the module. This page is the default view when a module is displayed.

Use of each of these displays is illustrated below with reference to our simple example.

4.3.1 Defining the symbols

The module will require 2 differential equations (**L1** and **L2**) and 2 parameters (**k1** and **k2**). You can add these in one of two ways. You can use the add symbols buttons on the top right of the module view. There are separate buttons for adding differential equations, variables (i.e. directly assigned variables) or parameters shown by the highlight in the screenshot below.

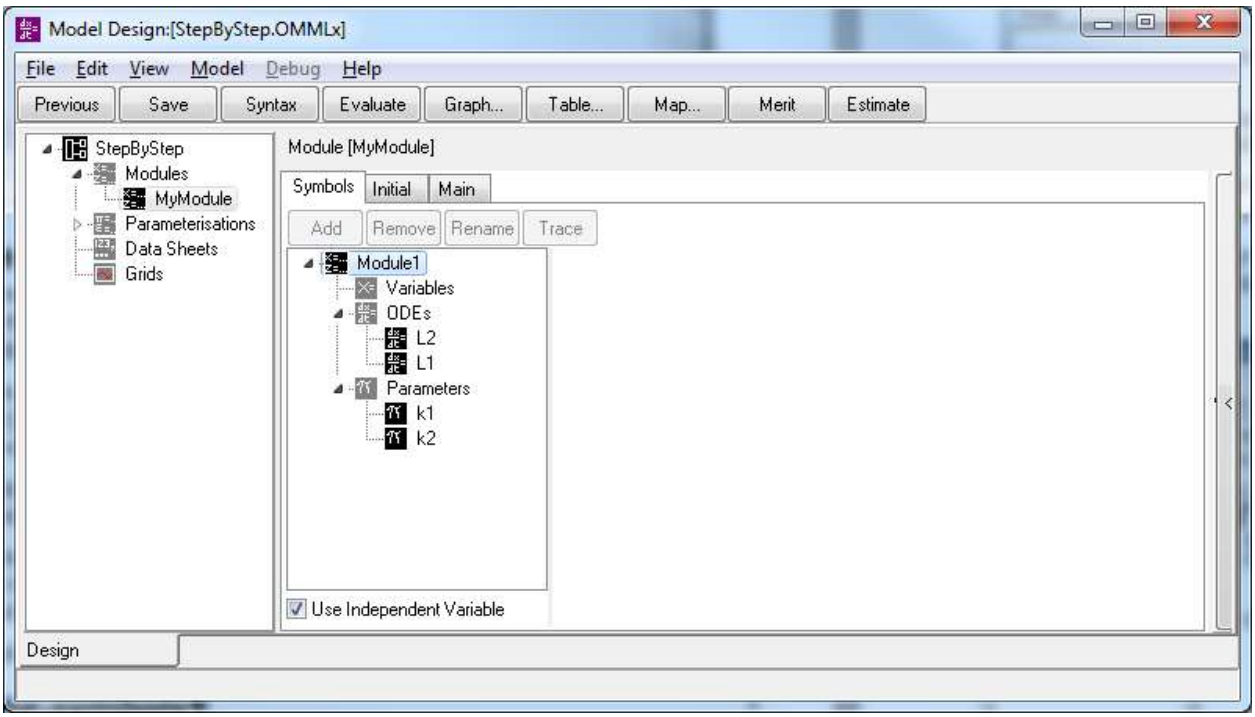
Selecting the differential equation button reveals the New Symbol dialog shown. In the presented example we have renamed the symbol **L1**.



The alternative method for adding variables is to switch to the symbols view of the module definition as shown below.

To add these simply select the appropriate node in the symbol tree and either right click to access the pop up menu or click the 'add' button on the tool bar.

The symbols will be given default names, which you can change via a pop up menu (right click on the symbol within the symbol treeview), or by using the rename button on the tool bar.

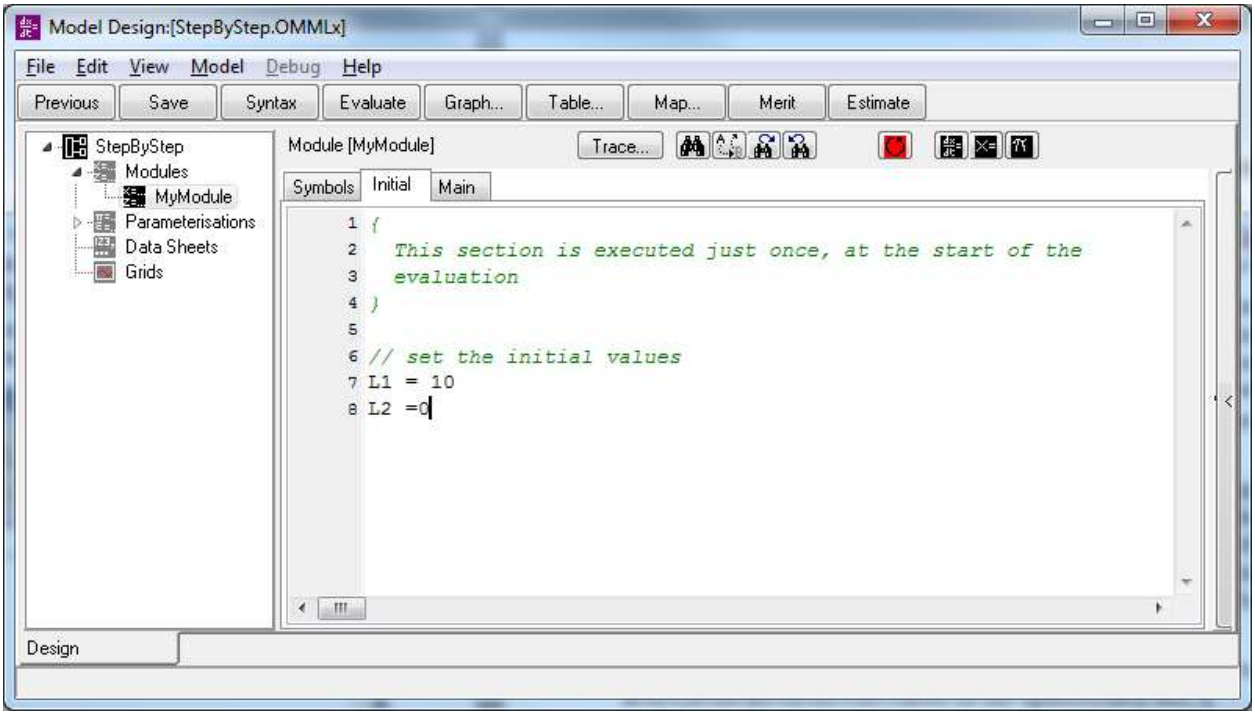


Once a symbol has been defined you can add a short description, units, or some extensive information text if you wish. This information isn't used by OpenModel for solving the model but can be useful to document the module.

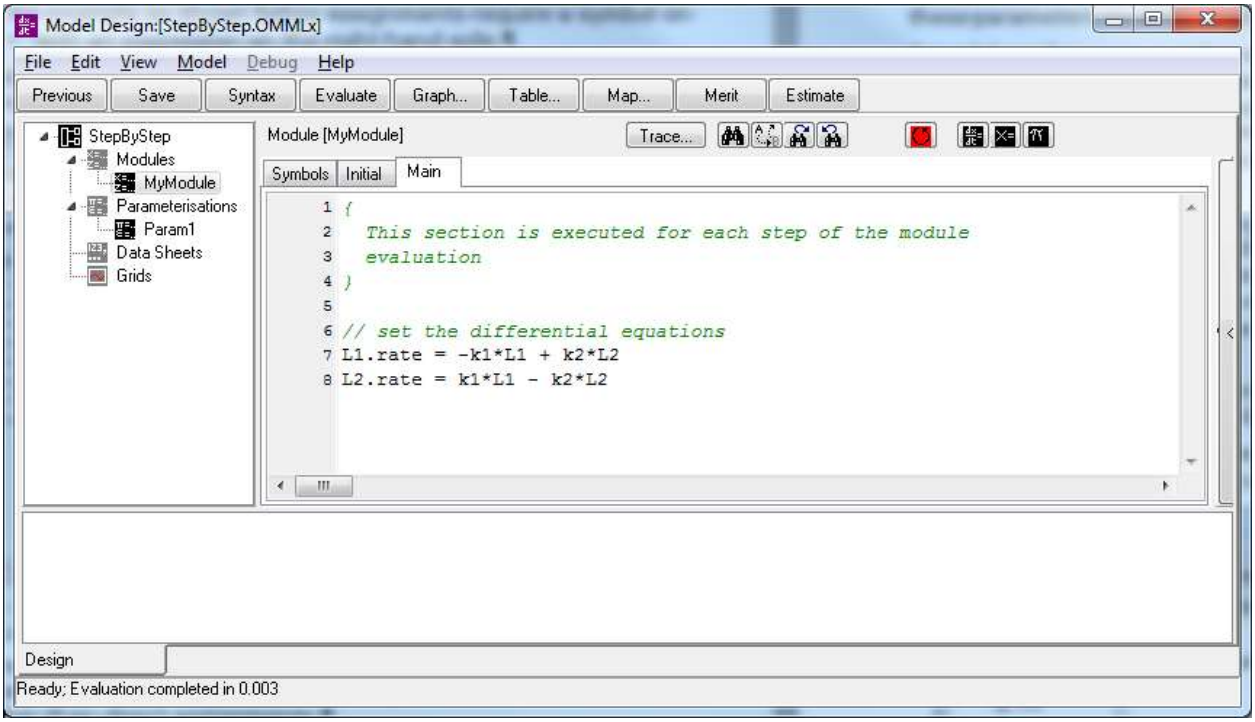
4.3.2 Entering the equations

In the case of differential equations initial values must be specified. This is done in the 'Initial' view. The equations are executed in the order they appear in the display. The syntax is fairly standard, for example as shown below assignments require a symbol on the left hand side of an '=' with an expression on the right hand side.

Comments are allowed using either C-style '//' at the start of the comment, or Pascal-style '{ }' blocks. Both are illustrated in the example shown.



The differential equations themselves are shown in the 'Main' view (below). Here the syntax is slightly more complex in order to specify that the equations refer to the rate of change of **`L1`** and **`L2`**, rather than direct assignments. This is accomplished with a '**`.rate`**' suffix to our symbols. **`L1.rate=`** sets the rate of change. **`L1=`** set the value of **`L1`**.



4.3.3 Defining the parameter values

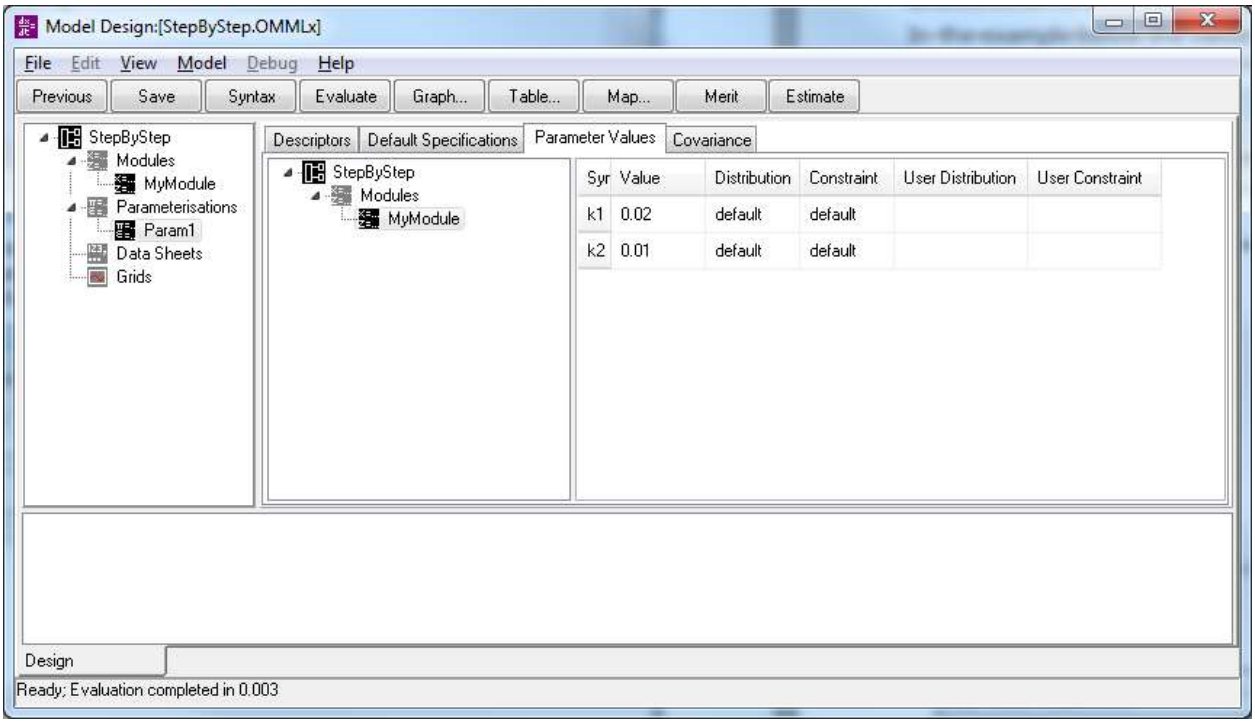
In the above equations the model has been specified using parameter symbols **k1** and **k2**. Before the model can be solved (or evaluated in OpenModel jargon) the values of these parameters must be specified[1]. This is done via the 'Parameterisation' display.

A model can have any number of parameterisations, enabling you to maintain different sets of parameter values and/or their distributions (for example, you might want to keep alternative parameter sets for different lakes). You can add, remove and copy parameterisations. By default a single empty parameterisation, **Param1**, will have been added when we created the model. Selecting this within the navigation tree will reveal the parameterisation view.

The parameterisation view is organised by module (using yet another navigation tree) and lists the parameters of each module. You can specify their value, and, optionally, information related to their distribution and constraints. Parameterisations allow default distributions and constraints to be set up for all parameters, as well as allowing for parameter specific distributions and/or constraints.

In order to evaluate a model any parameter values must be given a value. Distributions and constraints do not have to be specifically set.

In the example below the values of our parameters **k1** and **k2** have been set to their required values (0.01 and 0.02 respectively).



You are now ready to evaluate the model, although it would be a good idea to save the model first, if you haven't done so already, by clicking the Save button on the main tool bar or by selecting File|Save from the main menu.

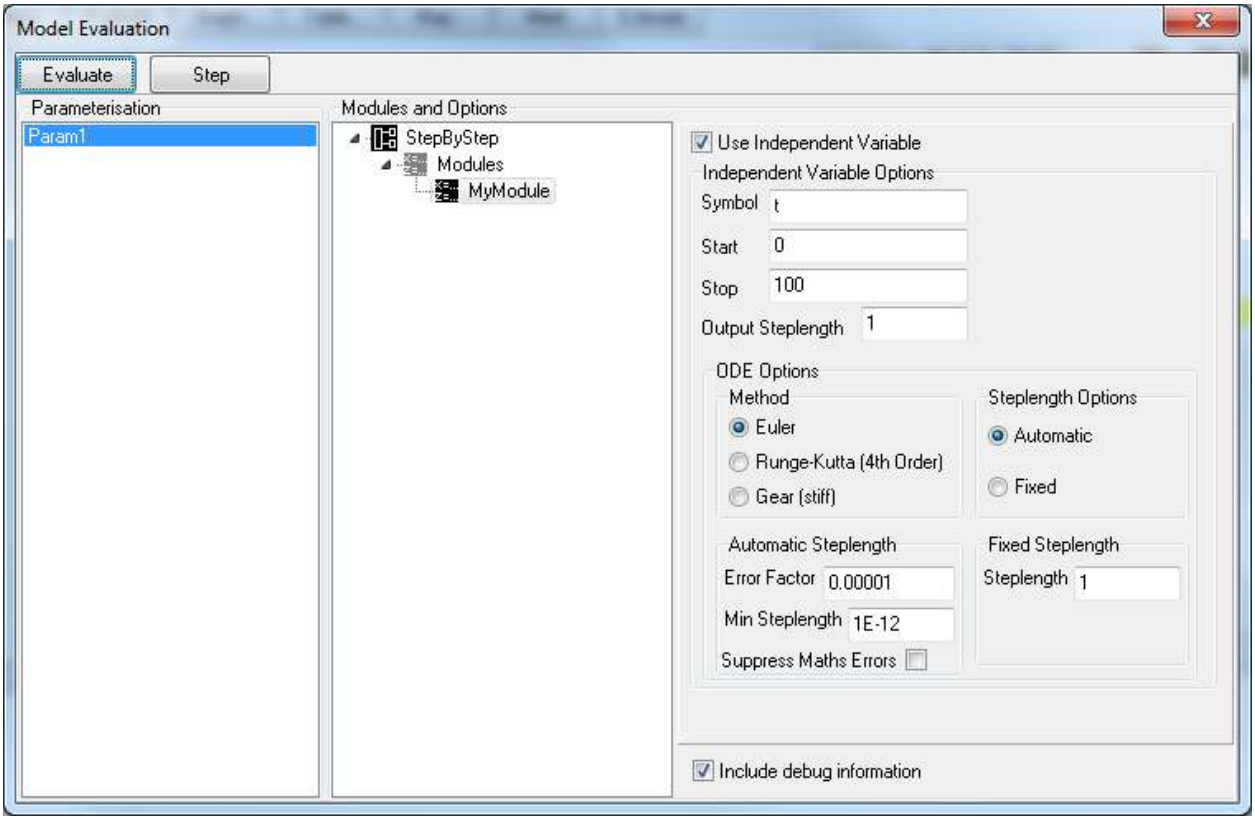
[1] The model could have been specified using the numerical values of the parameters, avoiding the need to set the value for each symbol. The advantage of using a parameter symbol is that the value can be set in a single location, and it can be used in conjunction with tools such as model fitting, monte carlo sampling etc.

4.4 Compiling and evaluating the model

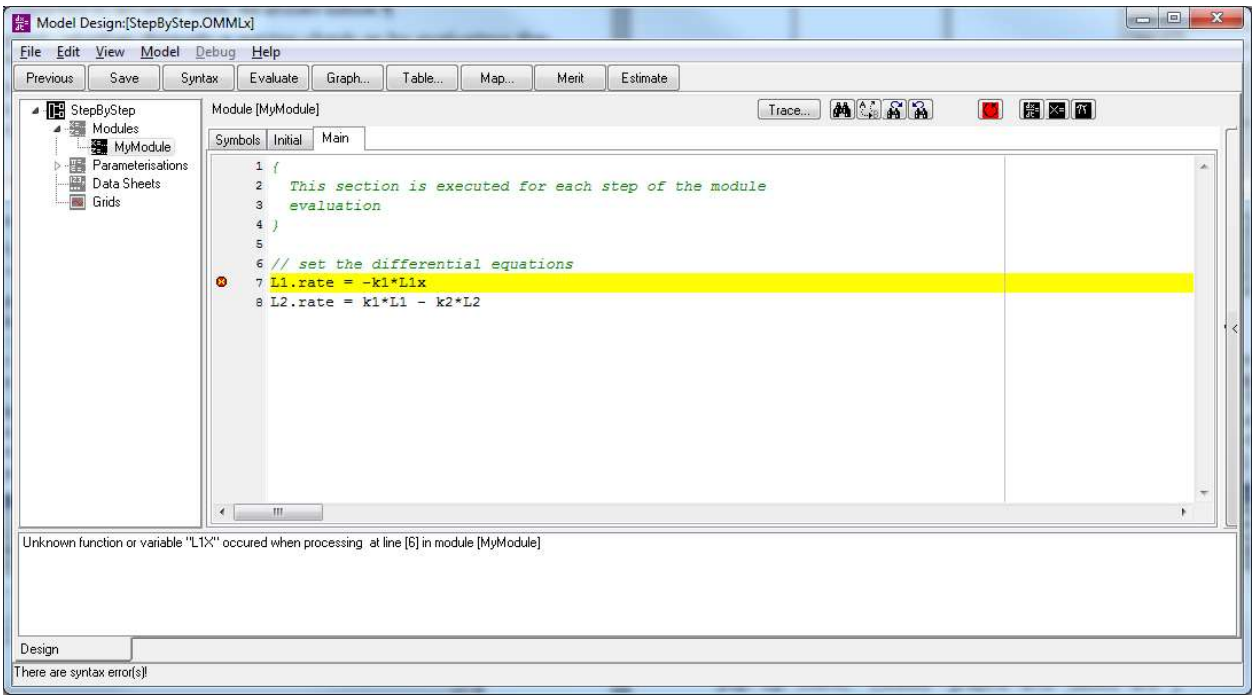
Before the model can be run, or evaluated, it has to be compiled. This is done automatically before an evaluation but if you want to check the syntax of the model at any stage, click Model|Syntax Check in the menu or click the 'Syntax' button on the toolbar. Any errors will be reported in an error view, as shown below.

To evaluate the model, that is to solve the model equations, over the defined range of the independent variable, select Model|Evaluate in the menu or click the 'Evaluate' button on the toolbar. This will display the model evaluation dialog box which displays a number of options:

- The Parameterisation list is used to specify which of the models parameterisations is to be used for the current evaluation (in this example there is only one parameterisation available).
- The Overlay Merit Function is used if you wish to display results with an overlay of user supplied data values (not discussed here)
- The Modules navigation tree in the centre of the dialog is used to specify which of the models modules is to be used as a starting point for the calculation. In a complex model it may be that modules call one another in a user defined sequence, however in this example there is only one module and therefore it is pre-selected as the start element. You don't need to change it.
- By default the model uses an independent variable, its symbol, range and number of output steps are defined. There are also options to select the method for solving any differential equations. The default values are all ideal for this example model so no changes are necessary.



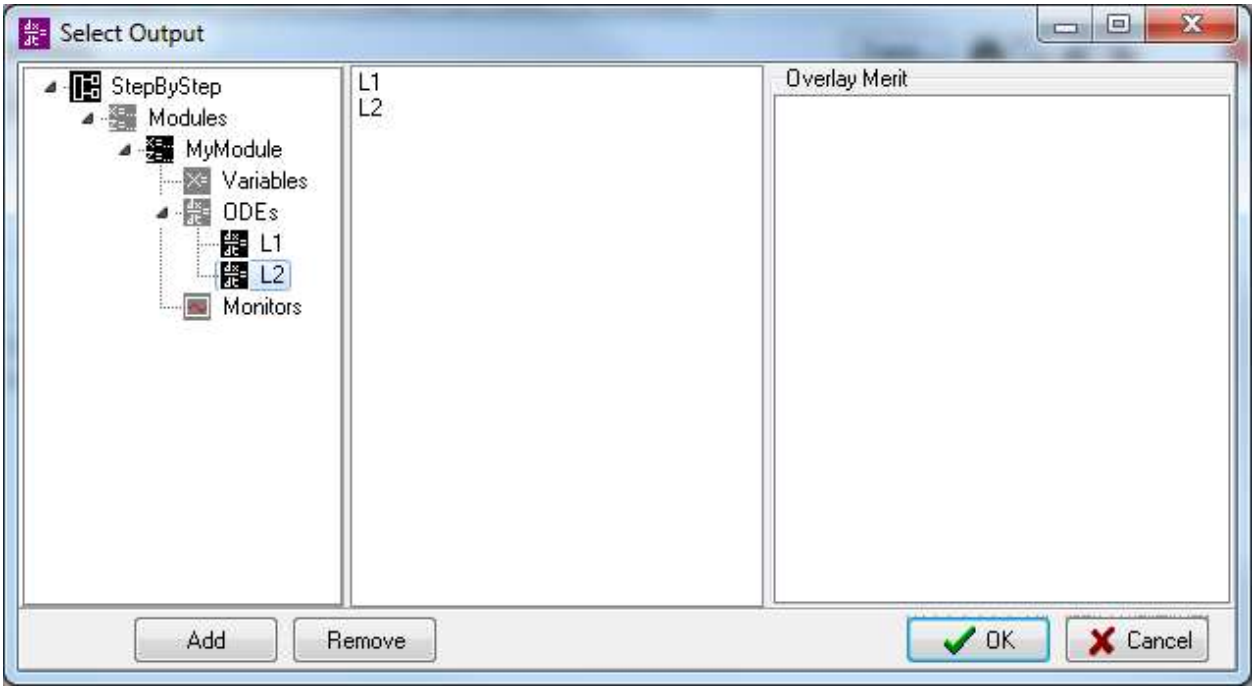
If the options are appropriately set the model can be evaluated by pressing the evaluate button. This will perform a syntax check of the model. If any errors are found they will be reported and the evaluation won't proceed as shown below.



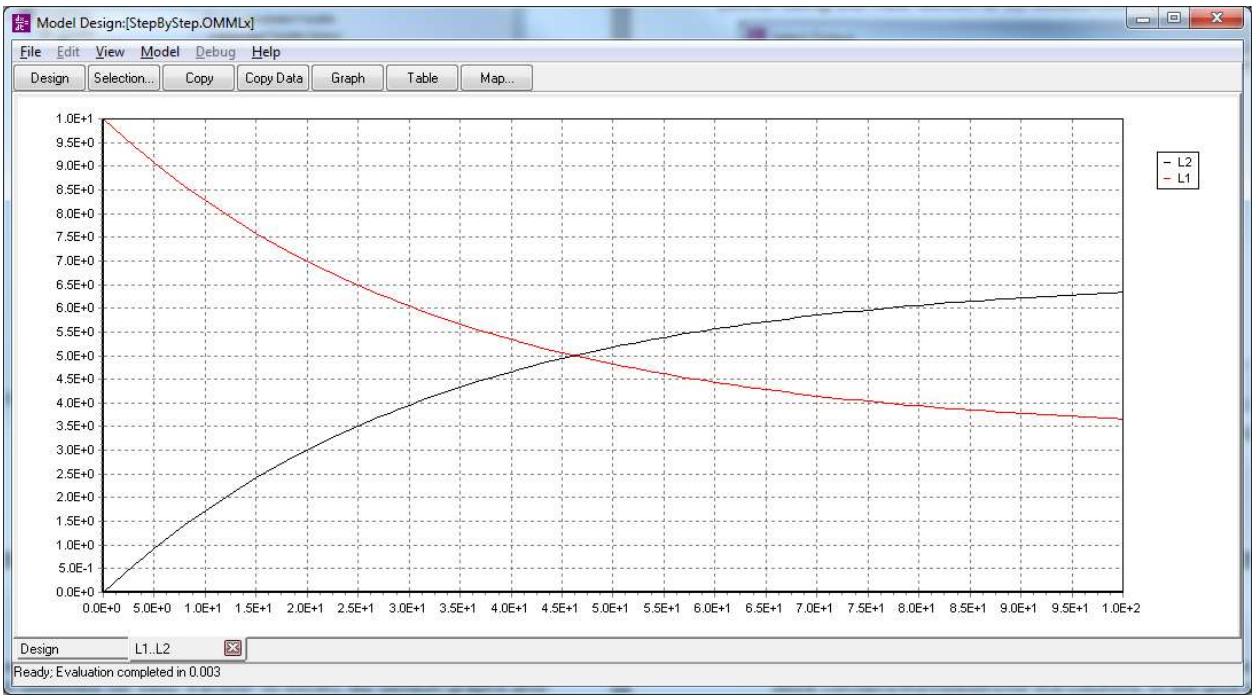
Once the syntax check is successfully completed the model will be evaluated and the results are available to you to interpret.

4.5 Viewing the results

Once the model has been run you will probably want to view the results. This can be done as either a graph or as a table. Select either View|Graph or View|Table from the menu or click the 'Graph' or 'Table' buttons on the toolbar. This will display the Select Output dialog. The left hand navigation tree gives a list of all the available calculated model symbols, select the ones you wish to display and add them to the right hand list (either using the 'Add' button or by double clicking on them).



When OK is clicked data for the symbols selected in the right hand list will be either graphed or tabled.



Both table and graph views have tools for changing the selected series, and copying the data to the clipboard (tab delimited for easy transfer to excel). By default graphs and tables will be automatically updated if the model is re-run. However they can be 'locked' against updates if required, right click the graph or table tab to toggle this option via the pop up menu. 'Locked' graphs and tables are indicated by a '[X]' symbol on their tab.

5. Using observational data

Often one needs to provide a data input to a model. Sometimes this is required as driving inputs to a model (i.e. the data are used directly in the model calculations). Another reason for connecting data is to compare the predictions of the model to observations, either visually or quantitatively using suitable statistical criteria.

In the case of using external data to drive the model the data can simply be connected to the model using a data sheet object (see 5.1). They can then be referenced symbolically in the model (see 5.2).

In order to compare the model with observations you must associate the observed data with the appropriate variables/odes in OpenModel. This is accomplished through the creation of merit function objects. This is also described in this section (see 5.3).

Merit objects are an important part of the procedure for the common task of adjusting the parameters of a model to obtain an improved fit of the model predictions to observed data. This sort of model fitting is described in section 6.

The use of data sheets and merit objects is outlined below, using as an example the supplied model Absalom.OMMLx. This implements a model of radiocaesium uptake by plants published by Absalom et al (2001).

5.1 Adding data to the model

Data are included in OpenModel using Data Sheet objects. You can add as many data sheet objects as you wish. They can be used to compare to predictions made by the model, or as inputs to the model itself (i.e. the values can be used within model calculations.

A data sheet comprises a column formatted set of data. The column headers are the 'symbol' used to reference that column of data within OpenModel. The first column of values is assumed to be the independent, or controlling, series within the data sheet. This will become clearer as we work through an example.

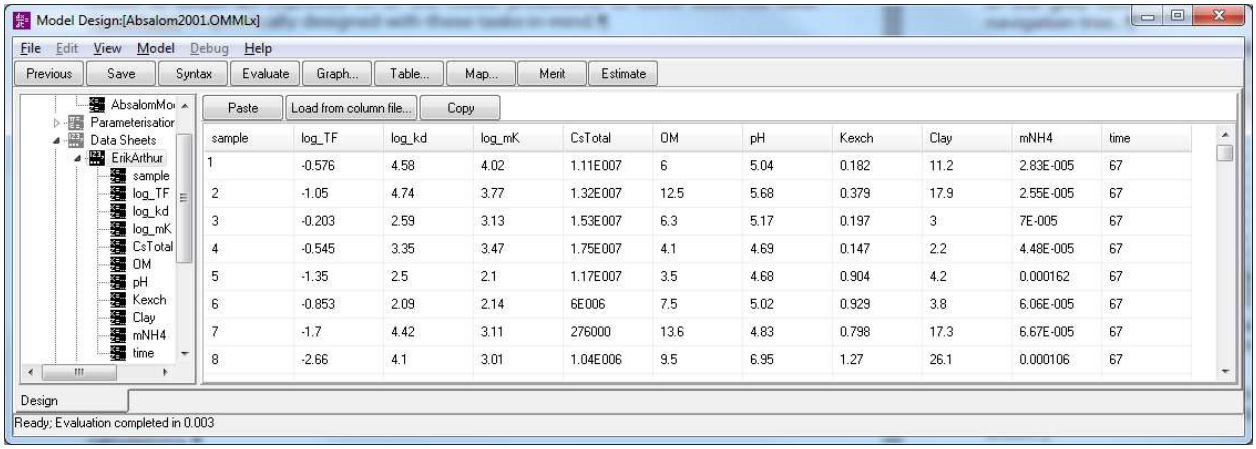
To add a data sheet to the model, right click on the data sheets node of the main navigation tree. The pop up menu will give you the option to add a data sheet. You can also rename it if you wish (this is recommended if you want to keep your model easy to understand). In the example below a data sheet called '**ErikArthur**' is shown, this is part of the model 'Absalom2001.OMMLx' which is included with the OpenModel installation.

To view the datasheet, click on its symbol in the navigation tool bar. The data sheet will initially be empty. There are 2 methods to add data to the data sheet:

- Paste a block of data from excel. OpenModel assumes that the first line of the block contains the headers for the columns. It will give an error if not. Empty cells in the data block will be treated as missing values. You can define missing values using a '*' character. In OpenModel such values are shown as large negative values (-1E308). Watch out for any odd characters excel may add (e.g. "" around strings). Spaces are not allowed in OpenModel symbols
- Use the 'Load from Column File' button on the data sheet view tool bar (see below). This will open a file open dialog enabling you to browse to the file of your choice. Sadly the supported file format is quite limited and inflexible. The data must be arranged in tab delimited columns, string column headers are required and will be used as the column symbols. Excel will readily create files in this format (use 'text tab delimited'). The requirements for the headers and data layout are the same as for pasting.

- Pasting from excel is the easiest method!

The screen shot below is the result of loading the file 'ErikArthur.txt' (included with the installation file as an example of the format). In the screen shot below these are shown in the grey column headers of the data sheet view and also as nodes within the navigation tree.



sample	log_TF	log_kd	log_mK	CsTotal	OM	pH	Kexch	Clay	mNH4	time
1	-0.576	4.58	4.02	1.11E007	6	5.04	0.182	11.2	2.83E-005	67
2	-1.05	4.74	3.77	1.32E007	12.5	5.68	0.379	17.9	2.55E-005	67
3	-0.203	2.59	3.13	1.53E007	6.3	5.17	0.197	3	7E-005	67
4	-0.545	3.35	3.47	1.75E007	4.1	4.69	0.147	2.2	4.48E-005	67
5	-1.35	2.5	2.1	1.17E007	3.5	4.68	0.904	4.2	0.000162	67
6	-0.853	2.09	2.14	6E006	7.5	5.02	0.929	3.8	6.06E-005	67
7	-1.7	4.42	3.11	276000	13.6	4.83	0.798	17.3	6.67E-005	67
8	-2.66	4.1	3.01	1.04E006	9.5	6.95	1.27	26.1	0.000106	67

Once a data sheet has been loaded it will be saved within the model definition, you will only need to repeat the loading process if you want to change the data or header columns.

The copy button in the data sheet view copies the entire datasheet onto the clipboard from where you can paste into another application (e.g. excel) or into another data sheet.

5.2 Referencing data within the model formulation

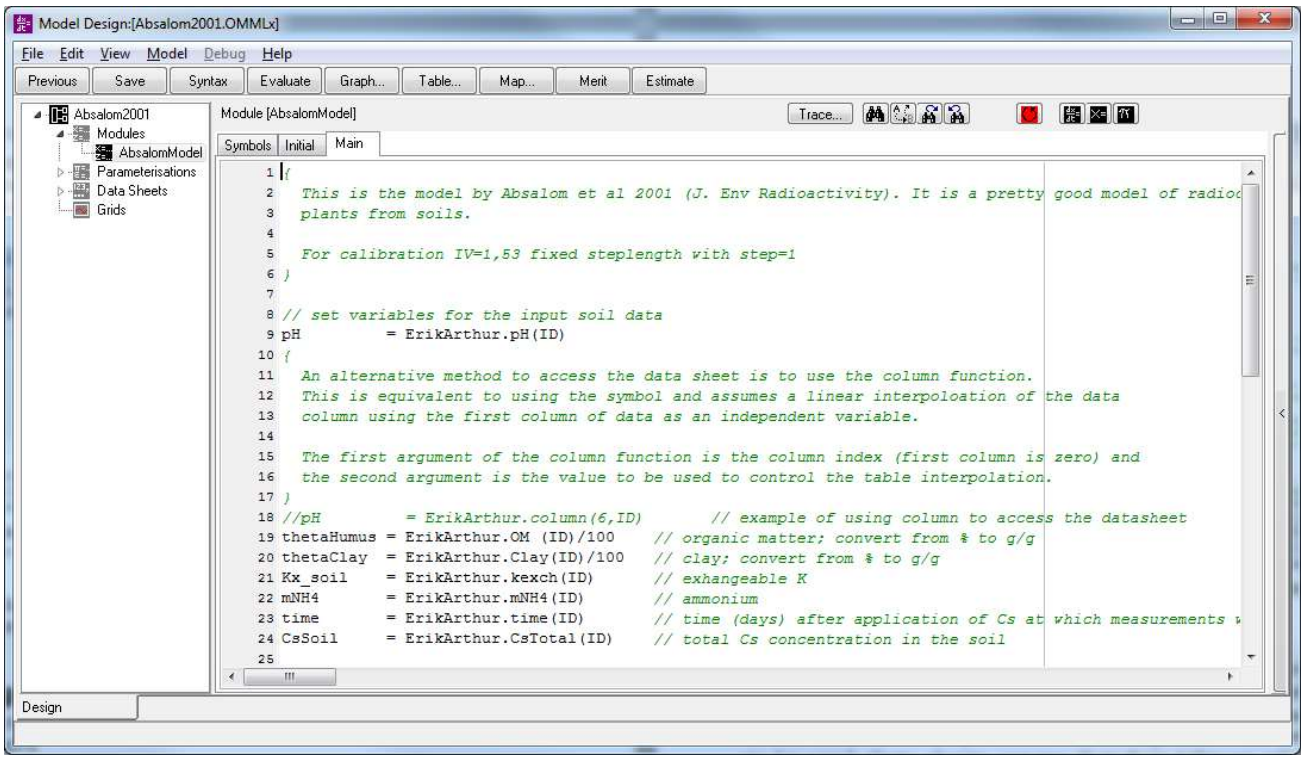
There are 3 ways in which data within a data table can be referenced (i.e. used) in the equations of a model script

- By symbol; i.e. the symbol used to denote the column of data in the data table is used to reference the data within an equation. In this case the value of the model independent variable is used to linearly interpolate the values in the table assuming that the first column in the data table is the independent variable values of the data.
- By column and row position within the data table. This simply treats the data table as a two dimensional grid and accesses the data accordingly.
- By column. This is equivalent to accessing the data by symbol, but using the column index instead of the symbol. As in the by symbol case the rows of the data table are linearly interpolated assuming that the first column in the data table is the independent variable values of the data.

5.2.1 Reference by Symbol

The model **`Absalom2001.OMMlx`** provides a good example of the use of data sheet values to drive a model calculation. The model iterates of over a sequence of soil samples, each denoted by a sample number with the symbol '**`ID`**' in the model (this is the independent variable of the model).

In the screen shot below model variables **`pH`**, **`thetaHumus`** etc, are set to values read from the data sheet '**`ErikArthur`**'.



From the above screen shot an example of the basic syntax assigning values from a data sheet to a variable is

```
pH = ErikArthur.pH(ID)
```

The variable **pH** is assigned a value based on the values of the column **pH** within the data sheet **ErikArthur** (the variable and the data column do not have to have the same symbol). The parentheses indicate which symbol within the module is to be used to decide which row of the pH values within the **pH** column of the datasheet **ErikArthur** is to be used. Data sheets assume that their leftmost column is the controlling, or independent, series of the data sheet.

For example, if **ID** is equal to 6 the procedure will search through the independent series of the data sheet (the left most column) for the value 6 and assign the corresponding value from the **pH** column to the variable. If there is no matching independent series value then the procedure will assign a value by linearly interpolating the appropriate values in the data sheet.

5.2.2 Reference by Column

This method for assigning values from data sheets linearly interpolates a data column (as for the reference by symbol method) selected by its index rather than by the column symbol. For example the 6th column (counting the first independent variable column as zero) is interpolated to an independent variable value of ID using

```
pH = ErikArthur.column(6,ID)
```

To re-iterate the column indices are zero based such that the left most column has an index of zero.

This form of data value assignment is also shown (commented out) in the example model Absalom2001.OMMLx.

This method is especially useful if you wish to read data table values into an array within the model script. This is illustrated in the example model **Crout2006.OMMLx** in which data are read in to arrays representing 23 different samples, each of which is used within the model.

5.2.3 Reference by Column and Row

This method assigns values from data sheets simply by row and column indices on the data sheet. For example direct access to the particular column (6) and row (3) of the data table

```
pH = ErikArthur.columnrow(6,3)
```

In this case row and column indices are zero based such that the left most column and top row of values (not including the header row) have indices of zero.

5.3 Defining the merit object

Specifying how data is compared to values predicted by the model is accomplished using a merit object.

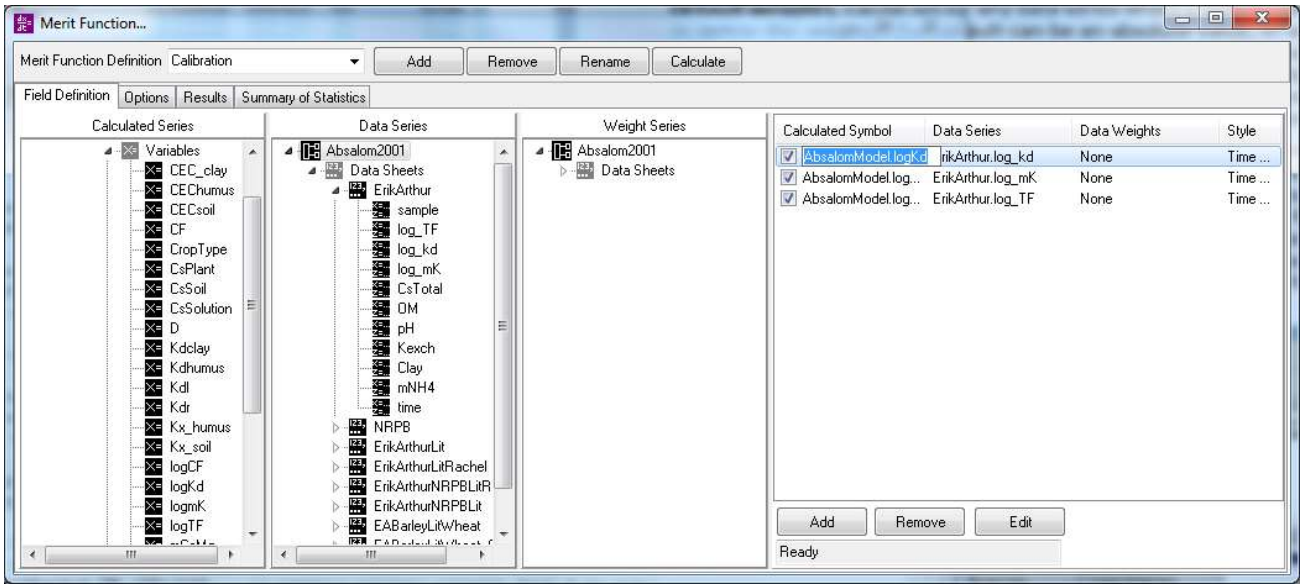
To control the merit objects within a model select Model|Merit Function from the menu (or click the Merit button on the main toolbar). This will display the merit function dialog box, shown below for the **`Absalom2001.OMMLx`** model included with the OpenModel installation files.

The top panel of the merit function dialog box shows which merit object is displayed, and allows the user to select a different object using the drop down list. Merit objects can be added, removed, and renamed using the buttons on the top panel.

A merit object comprises a list of 1 or more 'connections' between a calculated model series, and a data series. Optionally these connections can include additional data series to be used as a weight in the calculation of the merit function. In the example below the merit function comprises 3 such connections which are shown in the right hand panel. These specify that the model calculated values for **`AbsalomModel.logKd`** will be compared to **`ErikArthur.log_kd`** and that no weighting is required.

Each connection in the merit object can be enabled/disabled using the check box next to their entry (on by default). They can also be removed or edited as required.

To add a new connection select the required model symbol in the Calculated Series view; select the data series required in the Data Series view; if required select a further data series in the Weight Series view and then either double click or press the add button. The new connection will be added to the right hand view.



When the Calculate button is pressed the model will be run and the merit function statistics calculated. This will use the start module, parameterisation and options

specified on **Options** tab of the merit function dialog box. Once the calculation is complete (it requires a single iteration of the model) the results are shown on the **Results** tab and summarised on the **Summary of Statistics** tab.

Each of these is described below.

5.3.1 Options

Calculation Options

Merit Function: this can be set to a conventional residual sums of squares, or a sum of absolute deviations (SAD). The latter option 'works' computationally but should be very cautiously interpreted when used with any methods which assume a normally distributed error model (i.e. most of the fitting methods within OpenModel!).

Use Weights: this enables the use of a weighted RSS or SAD. Each contribution to the RSS is divided by the appropriate weight for that data point.

$$RSS = \sum \left(\frac{obs_i - model_i}{weight_i} \right)^2$$

$$SAD = \sum \left| \frac{obs_i - model_i}{weight_i} \right|$$

You can use weights and log transform at the same time (the weights are not logged), however your data is starting to become quite abstract in the opinion of these authors!

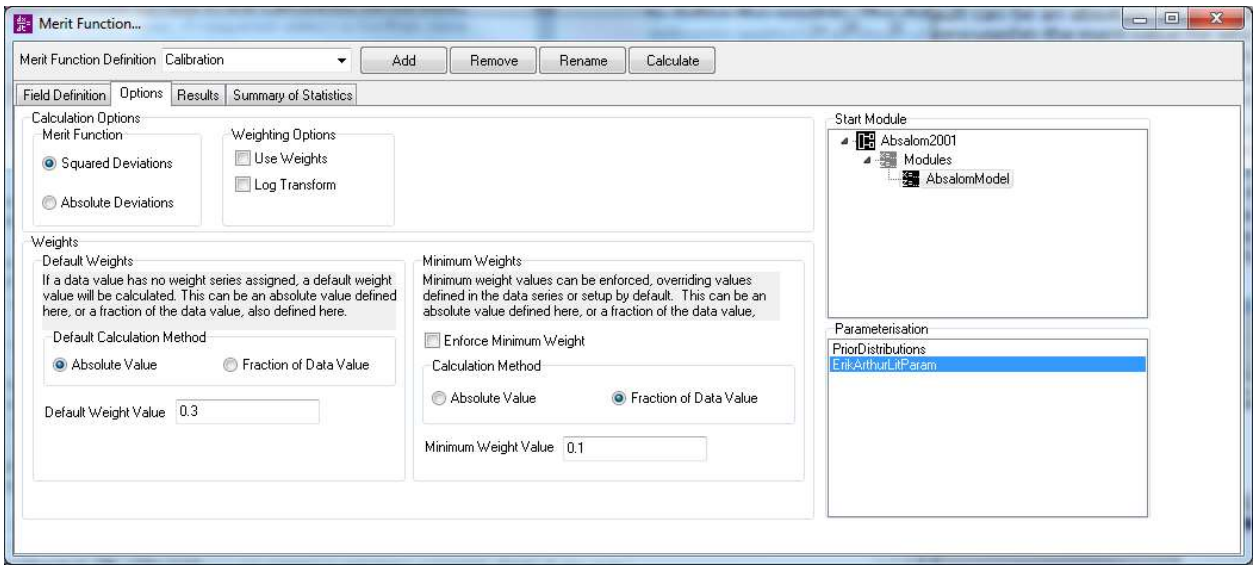
Start module and Parameterisation

These views simply enable the user to select the appropriate start module and parameterisation to be used for the calculation of the merit function.

Weights

Default weights: can be set for any data series where a specific data series is not used to define the weights. The default can be an absolute value, in which case the value defined is applied to all observations used in the merit value for which a weight series is not defined. Alternatively, the default weight can be set as a fractional value, in which case the weight for each observation (for which a weight series is not defined) is simply calculated as the product of the observation and the defined fractional value.

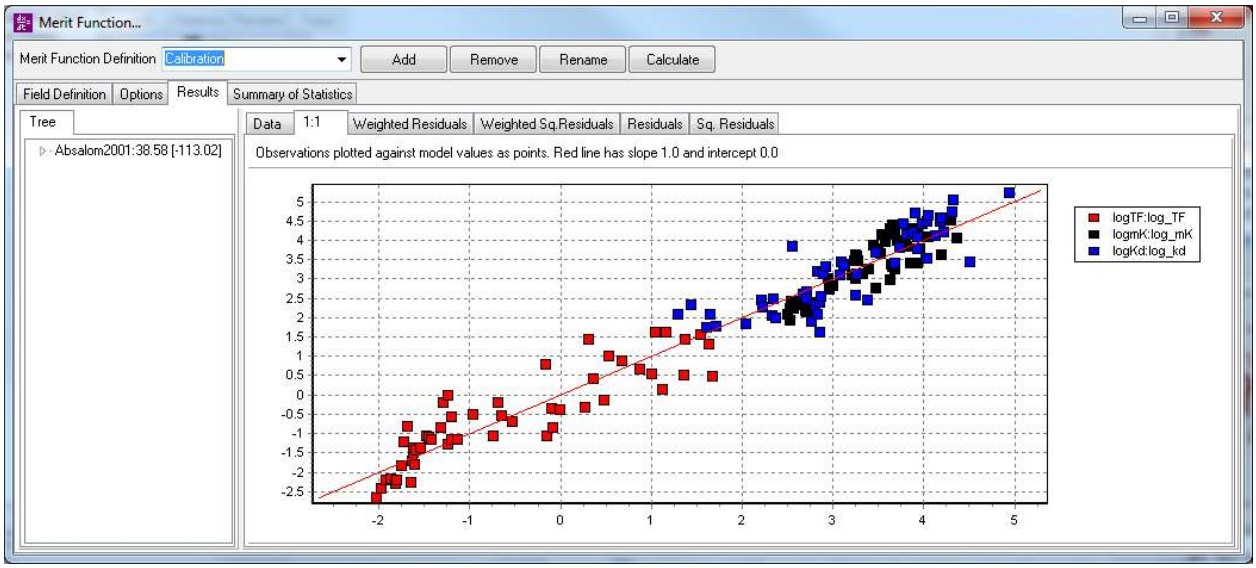
Minimum weights: anomalously low weight values can distort the merit function giving a disproportionate contribution to a very small number of observations. Defining minimum weights can be an effective remedy for this effect. For example a data series may be available with estimated standard errors which are used as the weight values. If these have been derived from a small number of replicate measurements it is quite common for a few of the SE values to be implausibly low. One may wish to set criteria such that the estimated SE value is used, subject to a minimum fractional or absolute value.



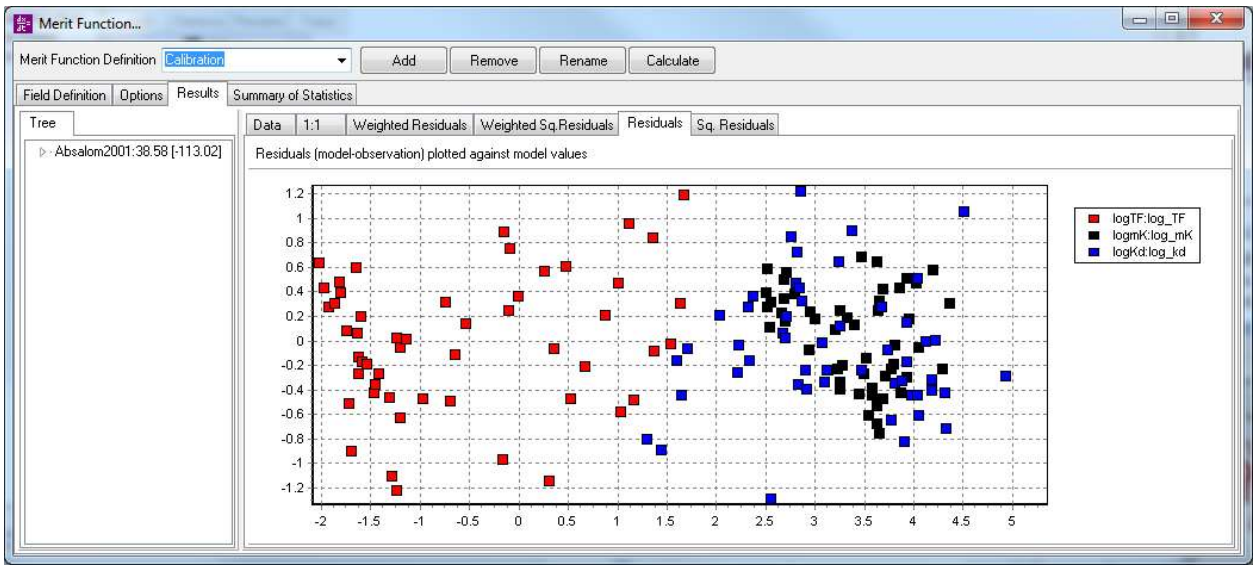
Results

The numerical results are shown in a tree giving a breakdown of the components of the residual sum of squares between series (and if the tree is expanded) by individual data point. The individual components of the residual sums of squares are also shown in a column format (suitable for pasting to excel) on the data view of the results tab.

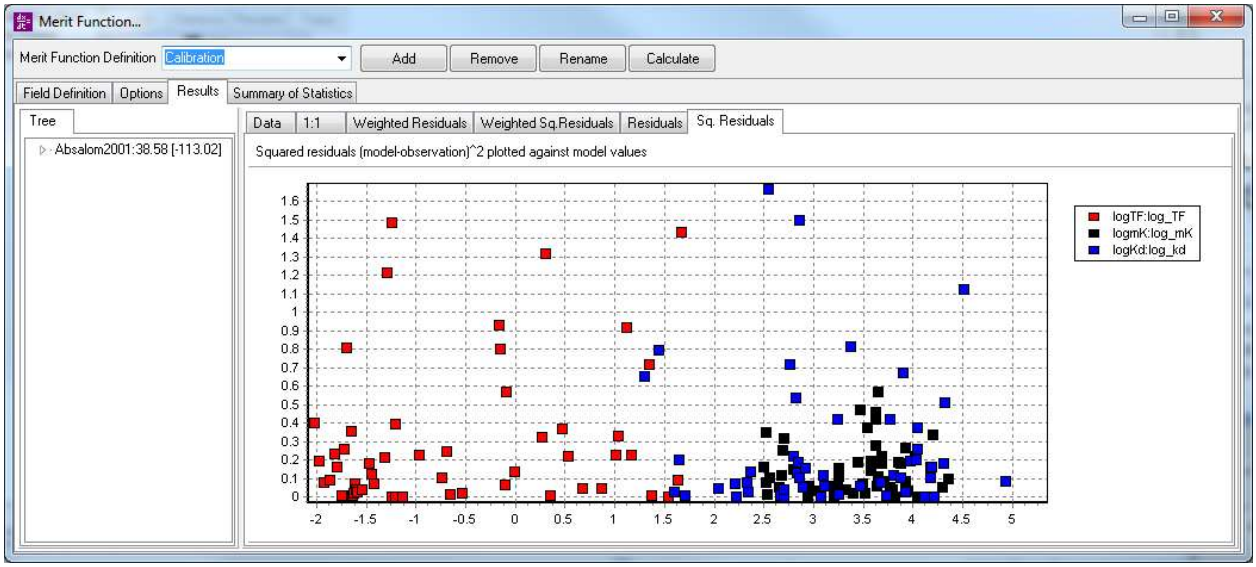
A number of graphical presentations are also available. As shown below a plot of the observations vs the predictions is shown, together with the ideal line of identity. As in this case the observations may comprise different series (i.e. comparisons for different model calculated variables/odes) and these are shown as different colours.



It can be useful to check the distribution of the residuals about the line of identity. This gives an indication of systematic biases in the model. This can be done using the plot of residuals (or weighted residuals if that is appropriate). Once again these are shown in different colours if multiple series are being used.



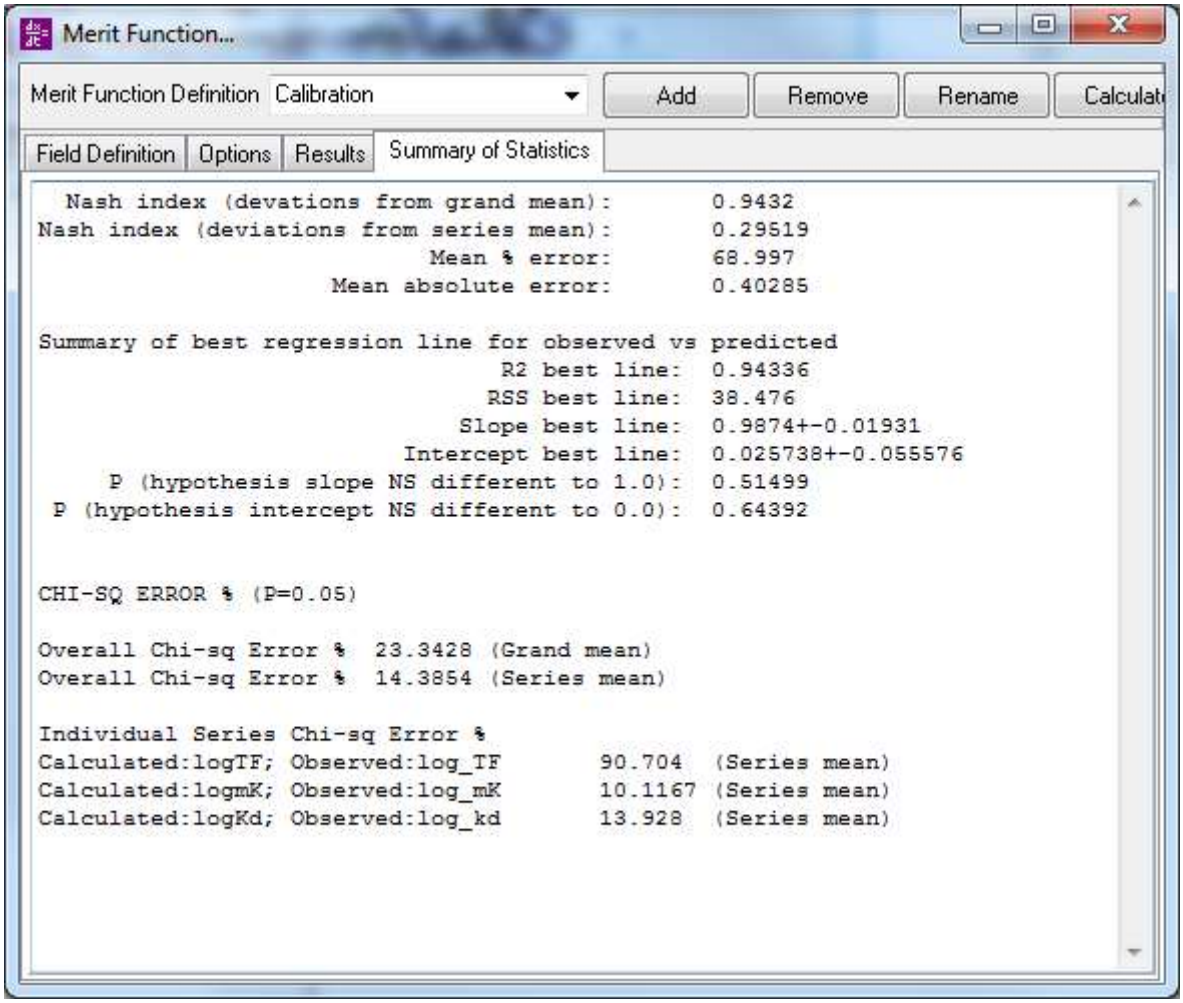
Combining series of different magnitudes into an overall measured of goodness of fit can be problematic (and give rise to difficulties obtaining an appropriate fit). Ideally the contributions of each observation to the overall merit value are approximately equivalent (i.e. you don't want to see the merit value dominated by one or two observations). So it can be useful to check that particular data points do not have an excessively high contribution to the RSS. This can be done visually using the plots of squared residuals (or weighted squared residuals if that is appropriate). The fractional contribution of each data point to the RSS is also shown in the data view.



Right clicking on any of the above merit result charts opens a pop up window with the optioms to set the veritical scale to logarithmic and to copy the chart to the clipboard.

Summary of Statistics

An example summary of statistics view is shown below. The values are presented in a simple text format which can be readily copied and pasted into excel (etc) as required.



The Nash Index is the proportion of the observed variation accounted for by the model. This is analogous to r^2 in linear regression although strictly r^2 only applies to the best fitting line, whereas the Nash Index applies to any model. Two versions of this statistic are presented. evaluated over the entire data set, Nash (grand mean) and Nash (series mean).

$$Nash(\text{grand mean}) = 1 - \frac{\sum (observed - model)^2 / weight^2}{\sum (observed - grand\ mean)^2 / weight^2}$$

Where grand mean is the mean of the combined observations across all series considered.

$$Nash(\text{series mean}) = 1 - \frac{\sum (observed - model)^2 / weight^2}{\sum (observed - series\ mean)^2 / weight^2}$$

Where series mean is the mean of the each individual series of observations.

If the merit function has a single series then Nash (grand mean) and Nash (series mean) should have the same value.

As Nash (series mean) is calculated with respect to the mean of the observed values for each series separately it is less likely to flatter the results, especially in cases when there are large differences in magnitudes between series. However if the series comprise just a few observations and have poorly defined means the nash (series means) values will not be robust. The choice of appropriate statistic depends on the circumstances.

The mean percentage and mean absolute error are simply given by

$$MPE = 100 \times \frac{\sum \left| observed - model \right| / observed}{N}$$

$$MAE = \frac{\sum \left| observed - model \right|}{N}$$

where N is the number of data point. Note, in the case of MPE and MAE no account is taken of the weighting of each data point.

Once again the use of these statistics depends on the circumstances. In the case presented the model-observations comparison uses log transformed values. The mean % error should be cautiously interpreted!

It can be useful to consider the best fitting line through the set of observation-calculated values. For an ideal model the best line would not be significantly different to the 1:1 line (i.e. slope=1 and intercept=0). The r^2 of the best line will always be at least as good as the Nash index (grand mean) and in practice will usually have a higher value. Of more interest than the r^2 of the best fitting line are the values of the intercept and slope, and in particular whether these are significantly different to 0.0 and 1.0 respectively. This is tested using a t-test and the resulting P values are presented.

In some application areas a 'chi-square error%' is used to help quantify model fit. The principle is to calculate the error in the observed values that would be required for the model to be a significant fit to the data at an appropriate significance level (i.e. P=0.05) using a chi-square test. At the risk of stating the obvious, this method does not calculate the error in the observations (how could it!). But addresses the question 'how big would the observation error have to be for the model fit to be significant at P<0.05?

To ease interpretation it is usually convenient to present this 'error' relative to the observed values, typically as a % of an appropriate observed mean value. The equation used is

$$\text{chi} - \text{sq error}\% = 100 \sqrt{\frac{1}{\chi^2} \sum \frac{(\text{observed} - \text{model})^2}{(\text{observed mean})^2}}$$

Where χ^2 is the critical value of χ^2 for P<0.05 with the appropriate degrees of freedom.

There are several choices of observed mean, should we use the grand mean over all series or the individual series means? Similarly the chi-sq % error can be summarised over the entire data set or for individual series. The appropriate choice will depend on the circumstances and OpenModel presents several variations.

In the case of a simple merit evaluation (i.e. without fitting) the degrees of freedom are taken to be the number of observed values minus the number of observed means used. If the merit value has been calculated through fitting then the degrees of freedom used in the fitting will be used (i.e. allowing for the number of adjustable parameters).

In the case shown the data have been log-transformed so have similar magnitudes and the overall chi-sq error by grand mean and series mean are similar. The chi-sq % error calculated for the series individually show an especially high value for logTF. Investigating this using the individual values (on the Results|Data tabs) reveals that this is due to the effect of one particularly low observation value which results in the chi-sq error being expressed as a high %. This would appear to be a distortion of the statistic which needs to be cautiously interpreted.

5.4 Layout of the Observation Data

By default OpenModel assumes that the observational data you are using is arranged as a time series and this is denoted by the style column in list on the right hand side of the

field definition tabbed view. The alternative layout is as a 'column'. These can be set by right clicking the entry in the style column.

Time Series (default)

If the observations are arranged as a time series OpenModel regards them as values which apply at the value of the independent given in the first column of the data sheet.

Column

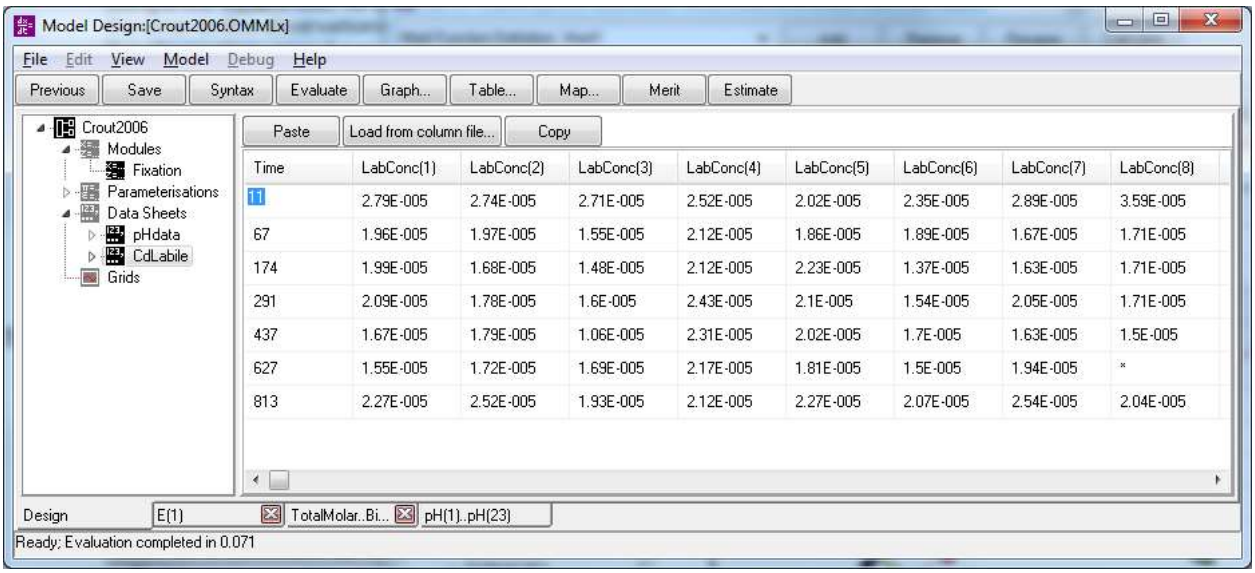
Column can only be used for comparing 1-D array variables with observations. In this case the merit function is calculated by comparing the array element values with the column of values on the data sheet. If the array is X(1..10) then X(1) is compared the first row of data, X(2) the second and so on. There is no attention paid to the independent variable, the comparison is made at the end of the model simulation, i.e. using the final values of the array variable. If the lengths of the data column or the array are not equal the comparison is made up to the length of the shorter component.

Column layout is useful for arranging comparison of a model to a complex arrangement of observations which would not be easy to set up as a set of time series values.

5.4.1 Using Observations Structured as Arrays

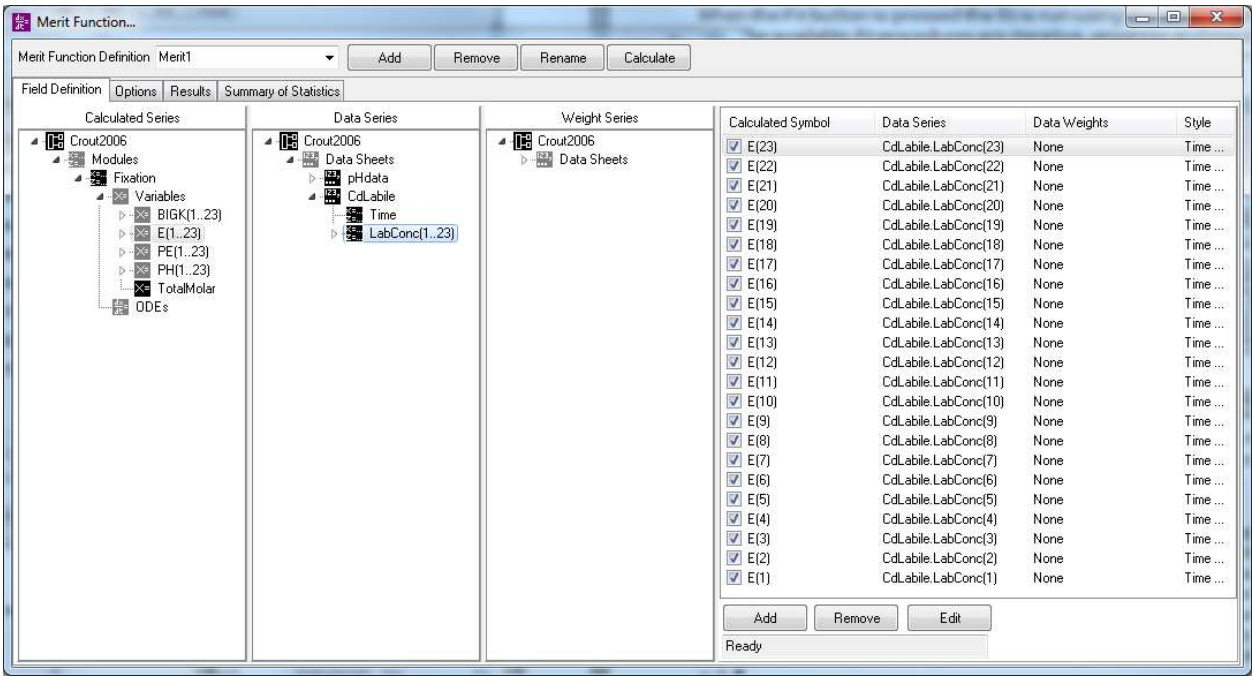
It can be useful to set up a merit object using observational data set up as an array. The example model **`Crout2006.OMMLx`** illustrates this facility. In this case the model script is being used to simulate the fixation of metals in 23 different soil types and the merit function is aggregating the deviation between model predictions and observation over the 23 soil types.

The data table is shown below; the important feature to note is that the observations are labelled with the same base symbol appended by indices, **`LabConc(1)`**, **`LabConc(2)`** and so on. The software interprets these as a 1 dimensional array within the data table (higher dimension arrays are not possible within data tables). The syntax is dependent on the use of parentheses, e.g. (1) etc.



This use of a data table array makes it easier to set up the merit object across the 23 soil types. In the model the objective is to link the array variable **`E(1..23)`** with the observations represented as **`LabConc(1..23)`** in the data table **`CdLabile`**. This is accomplished by selecting the 2 arrays and clicking the Add button (or simply double clicking once the field names are selected in the trees). The individual merit series (for each comparison) are then added to the right hand view in the merit function dialog box.

This can also be done expanding the two arrays and selecting/adding the series individually one by one, but this method is quite tedious!



6. Fitting the model

Fitting the model requires 3 steps the first two of which have already been described:

1. Add the required data to the model (using one or more datasheet objects) see section 5.1
2. Specify how these data are to be compared to the model predictions (using a merit object), see section 5.3
3. Specify which parameters are to be adjusted, and what methods to use for this (using an estimate parameters object). This procedure is described in this section.

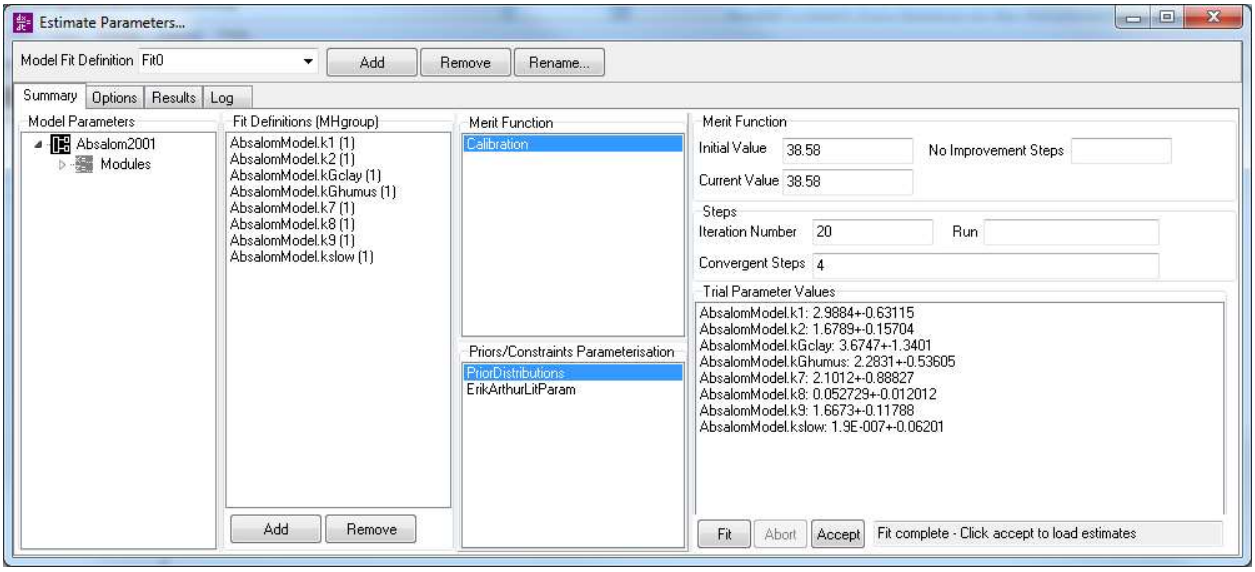
As an example we will continue to use the supplied model **`Absalom2001.OMMLx`**. This implements a model of radiocaesium uptake by plants published by Absalom et al (2001).

The adjustment of model parameters by fitting to a data set is accomplished using a 'fit' object. Select Model|Estimate Parameters to display the estimate parameters dialog box (shown below).

As with other OpenModel control objects you can select which object to display using the drop down list on the top panel of the dialog box, you can also add, remove and rename objects as required.

The left hand tree view shows the model parameters available, select the ones you wish to adjust, adding them to the **`Fit Definition`** list. You must also specify which of the model's merit functions is to be minimised by the fit by selecting an object in the merit function list (in the example below there is only 1).

When the Fit button is pressed the fit is run using the options specified on the Options tab. The available fit procedures are iterative, progress and results are summarised on the right hand side of the dialog. Final results are also shown on the Results tab, the format and interpretation depends on the fitting method used.

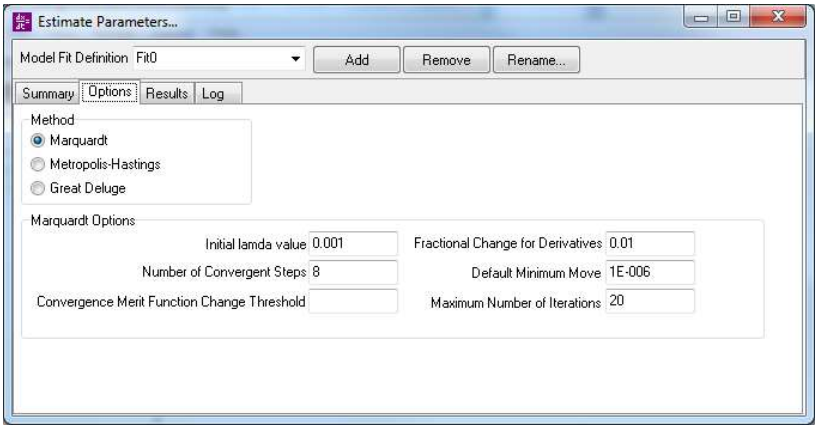


If you press the accept button a dialog will appear enabling you to insert the resulting parameter values into the parameterisation object of your choice.

6.1 Marquardt Method

OpenModel's default fitting method is Marquardt. This well known and elegant method is a deterministic search for the maximum likelihood parameter estimates (it minimises the

residual sums of squares). The method uses gradient information; the change in the residual sum of squares with respect to the parameters. In the OpenModel implementation this information is obtained numerically, by iteratively running the model with slightly adjusted parameter values to obtain the gradients. The number of models required per Marquardt step is therefore equal to the number of adjustable parameters+1.



Options

The Marquardt method combines a traditional steepest descent search with a direct estimation of the optimal parameter values based on the assumption that the shape of the residual sums of squares surface in parameter space is quadratic. The weighting between these two approaches is controlled by the **`lambda`** value, which varies automatically during the procedure. Large values represent steepest descent, small values quadratic estimates. The **`initial value of lambda`** can be set on the options page (0.001 is the value classically recommended).

A step is deemed to be convergent if the reduction in the residual sum of squares (as defined by the merit function object) is less than the **`Convergence Merit Function Change Threshold`**. If the **`number of..`** successive **`convergent steps`** equals the value specified (4 by default) then the procedure is deemed to have converged at the best parameter estimates and stops. If the residual sum of squares increases after a step (i.e. the method makes a bad move) then the step is not considered to be convergent (of course!).

As outlined above the gradient information required by the method is obtained by making small changes to the parameter values and re-running the model. The magnitude of the changes is controlled by the value of **`fractional change of derivatives`**. This should be as small as possible, but avoiding any effects of floating point accuracy. For most applications there will be no need to change the default value.

Common Problems

Marquardt is a deterministic method. For a given starting point in parameter space it will always convergence at the same answer (for a given set of the option values). This point may be a local rather than global minimum in the residual sum of squares surface. In general the method will perform better if 'good' starting values of the parameters can be provided. Normally the reliability of the parameter estimates obtained is tested by re-running the procedure from different starting points.

Occasionally a *curvature matrix error* will occur during the course of the fitting procedure (associated with a specific parameter in the model). The error indicates that the gradient of the residual sum of squares with respect to that parameter is zero. This causes an error in the procedure and it halts. It may be that the model (or fit definition) has been incorrectly specified (including a parameter which has no effect on the residual sum of squares will result in this error). Alternatively it can be that the gradients are very small,

and because of floating point limitations are recorded as zero, in these cases a solution can be to increase the **`fractional change of derivatives`**.

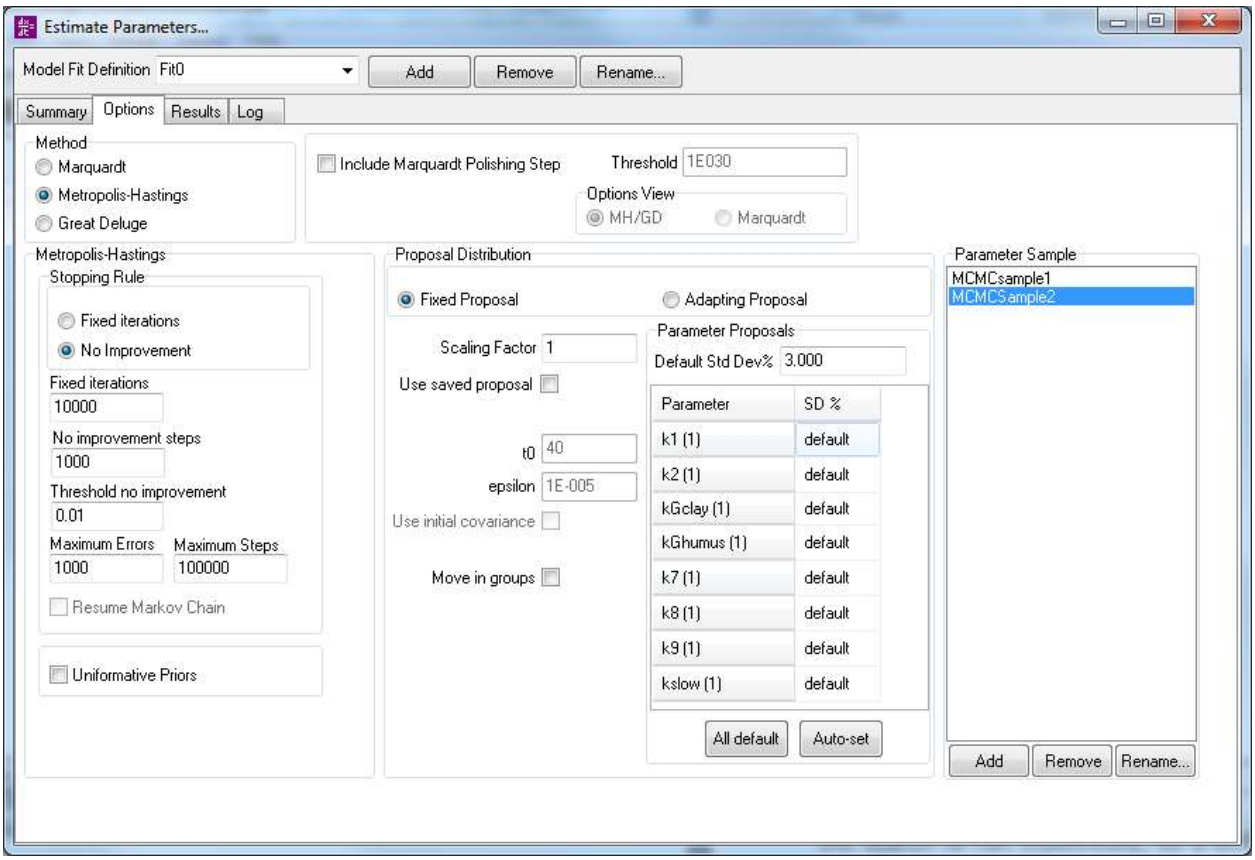
6.2 Metropolis-Hastings Parameter Search

OpenModel will allow the use of Metropolis-Hastings to search the parameter space. This can be used to perform 'Bayesian' model calibration using Markov Chain Monte Carlo (MCMC) or informally as an efficient stochastic parameter search procedure.

Select the Method as Metropolis-Hastings from the Options page of the Estimate Parameters dialog box. The example of **`Absalom2001.OMMLx`** is shown below together with descriptions of the various options available.

ParameterSample

Metropolis-Hastings results are written to a Parameter Sample object which can be separately saved, viewed and interpreted. In this example the results are to be saved to the parameter sample object **`MCMCSample`**. You can add, remove and rename these objects as required.



Stopping Rule

The length of the search can be controlled by setting the **`number of steps`**. If a numerical error occurs during the running of the chain (typically because a particular combination of parameter values is infeasible) this is recorded as an error, the run is ignored and the search will proceed. However this does run the risk of a bad set up resulting in an infinite loop so the user can specify the maximum acceptable number of errors (**`maximum errors`**). If the number of recorded errors exceeds this value the search will stop.

If **`fixed iterations`** is selected the search will run for the number of iterations specified under **`number of steps`**. If **`no improvement`** is selected the search will run for an indeterminate number of steps, stopping when there has been no improvement in the

residual sum of squares for the number of steps specified. This option is useful under some circumstances, especially if using a process such as model reduction where numerous fits may be undertaken. Of course there is the potential for the search to run indefinitely, so a limit to the number of steps taken can be applied (**maximum steps**).

A step is considered to show no improvement if the reduction in the merit function value is less than **threshold no improvement**.

Proposal Distribution

The proposal distribution controls the parameters 'walk' in the parameter space, i.e. how far they are likely to move in a given step. The proposal is not to be confused with the parameters prior distributions (set in the parameterisation object).

The proposal can be *fixed* or *adapting*. Under the fixed proposal the proposal distribution remains the same throughout the search of the parameter space. This makes the Metropolis-Hastings search a true markov process so the procedure can be considered a Markov Chain Monte Carlo (MCMC). This allows Bayesian model calibration and samples the posterior distribution of the model parameters. The adapting proposal adjusts the proposal as the method proceeds in order to improve the efficiency of the search. The method used is that described by Haario et al (2001) and this can greatly increase the efficiency of the search. The disadvantage is that the search is no longer truly markov (but see Haario et al (2001) for full discussion). If your purpose is simply to find the optimal set of parameter values you may not care whether the search procedure is truly markov. However you should be careful if you are trying to create a sample of parameter values from the posterior distribution and in this situation the fixed proposal should be used. However OpenModel allows you to undertake a trial search using an adapting proposal which can then be saved and fixed for an MCMC search.

Formally the proposal distribution used is a multi-variate normal distribution described by a covariance matrix which is estimated as the search proceeds. The diagonal elements are the proposal standard deviations for each parameter. In many case the covariances will not be available and the parameters treated independently.

Fixed Proposal

Under the fixed proposal option the proposal standard deviations for each parameter are set as either

(1) a % of the initial parameter value. This is set as a default value (**Default Std Dev%**) or specified individually for each parameter in the table in the centre of the dialog. Simply enter the required value, or double click the cell to toggle back to the 'default' setting.

There is a facility to **auto-set** the SD% for each parameter. This estimates the covariance of the parameters at the current set of values and sets the SD% accordingly (although they are treated as independent). This is useful to indicate the relative sensitivity of the model to changes in the parameter values.

(2) by checking the **Use saved proposal** box the last proposal covariance matrix calculated by the adapting proposal option can be used. This option is available if the adapting proposal option has been used previously. Under this option covariance between the parameter proposals is considered.

Adapting Proposal

Under the adapting proposal option an initial proposal distribution is progressively updated when the Metropolis-Hastings 'accepts' a new parameter set.

The initial proposal is defined using either

(1) the SD% values defined in the same fashion as used for the fixed proposal case.

(2) by calculating a covariance matrix at the initial parameter values (**Use initial covariance**).

The proposal updating starts after **t0** accepted steps. Increasing this value increases the size of the parameter sample used before the first update of the proposal distribution is made. This reduces the chance of small sample size causing a numerical failure. However it increases the number of steps before the benefits of adapting the proposal distribution start to be realised. In practice it is useful to increase t0 if the method fails immediately once the number of accepted steps exceeds t0. See Haario et al (2001) for more details.

Epsilon is a small value added to the diagonal elements of the proposal covariance to avoid a numerical error when the matrix is inverted. Once again, see Haario et al (2001) for full details.

Scaling Factor

Whether fixed or adapting the proposal distribution can be scaled using the **scaling factor** (by default equal to 1.0). This can be convenient as often one wishes to see the overall effect of a wider or narrower proposal.

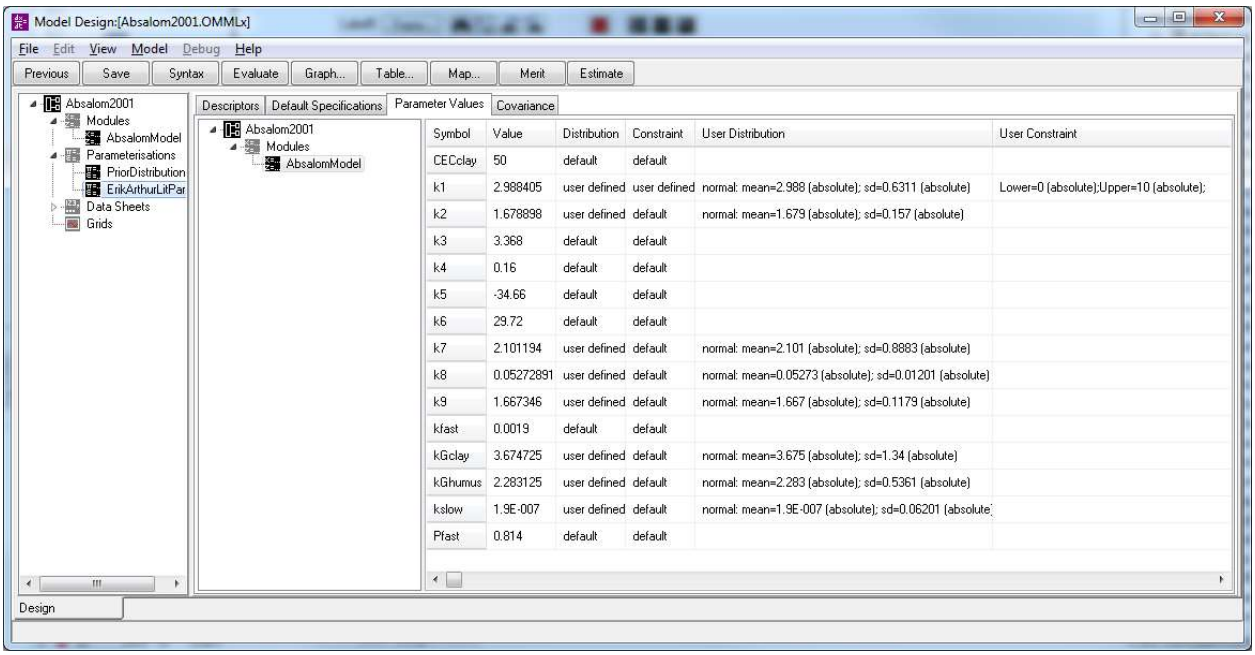
Typically the proposal distribution needs to be 'wide' enough to allow the search to move around quite freely, but not so high that the moves become almost random (as this becomes very inefficient). A key indicator of the search performance is the acceptance rate; typically 'people say' a target value of 20-25% acceptance is about right. The **scaling factor** can be adjusted to achieve this acceptance rate.

Move in Groups

Occasionally it is useful to make the chain move in stages, so that one group of parameters moves, and then another, rather than all parameters moving simultaneously. This option is enabled by checking the **move in groups**. Each step of the chain then only moves parameters in one group. So if there are 2 parameter groups and the chain will first step group 1 (all parameters in group 2 remaining at their current values), the next step will then move parameters in group 2 (group 1 remaining stationary). Groups are set by double clicking on each parameter in the Fit Definitions list on the Summary tab page.

Parameter Priors

A key step is to define the prior distributions for the parameters to be adjusted using Metropolis-Hastings. In this example the Parameterisation '**PriorDistributions**' is selected to define the prior distributions. This parameterisation was created by copying an existing model parameterisation (so the parameters all have the same values) but the values for the distributions were then adjusted to define appropriate prior distributions for each parameter. In this case the priors are 'informative' in that they define fairly specific ranges based on literature information (e.g. 'k2' in the example below). However the priors are quite broad and specified as uniform (so not that informative). Double clicking on the User Distribution field for a parameter gives an edit dialog box in which the field values can be changed as required. The distribution of a parameter can be set as 'default' (as for 'aa' in the example below), in many cases this is adequate and allows the system to be setup very quickly.



Symbol	Value	Distribution	Constraint	User Distribution	User Constraint
CECclay	50	default	default		
k1	2.988405	user defined	user defined	normal: mean=2.988 (absolute); sd=0.6311 (absolute)	Lower=0 (absolute);Upper=10 (absolute);
k2	1.678898	user defined	default	normal: mean=1.679 (absolute); sd=0.157 (absolute)	
k3	3.368	default	default		
k4	0.16	default	default		
k5	-34.66	default	default		
k6	29.72	default	default		
k7	2.101194	user defined	default	normal: mean=2.101 (absolute); sd=0.8883 (absolute)	
k8	0.05272891	user defined	default	normal: mean=0.05273 (absolute); sd=0.01201 (absolute)	
k9	1.667346	user defined	default	normal: mean=1.667 (absolute); sd=0.1179 (absolute)	
kfast	0.0019	default	default		
kGclay	3.674725	user defined	default	normal: mean=3.675 (absolute); sd=1.34 (absolute)	
kGhumus	2.283125	user defined	default	normal: mean=2.283 (absolute); sd=0.5361 (absolute)	
kslow	1.9E-007	user defined	default	normal: mean=1.9E-007 (absolute); sd=0.06201 (absolute)	
Pfast	0.814	default	default		

Marquardt Polishing Step

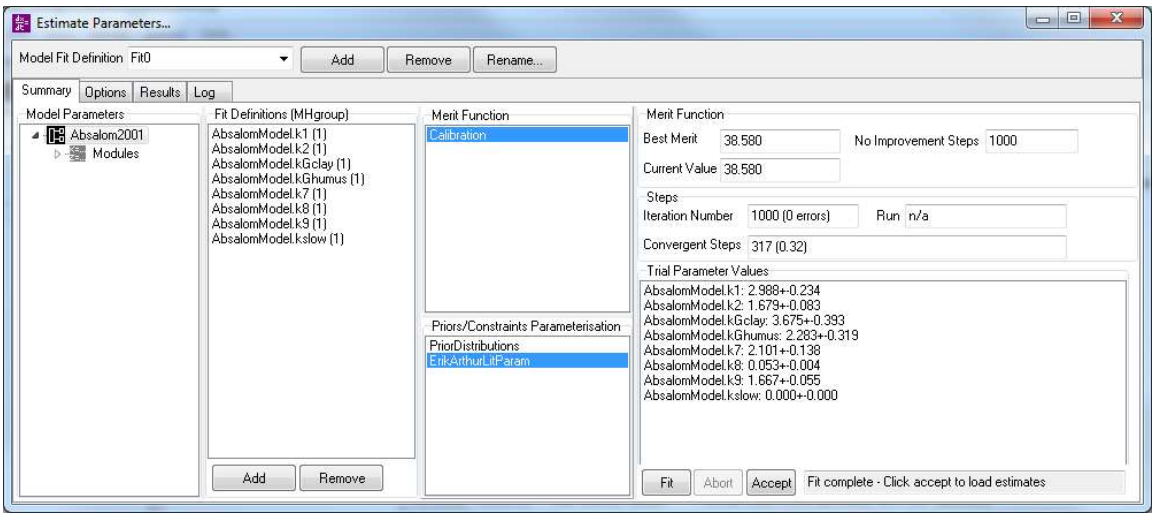
Typically the Metropolis-Hastings procedure will get close to the 'best' parameter sets but will not locate them exactly, at least not without an enormous computational effort. In MCMC applications this is not a problem, as the principal aim is to estimate the posterior distribution of the parameters rather than the maximum likelihood parameters estimates.

However it can be useful to take the final best estimates from Metropolis-Hastings and use them as starting values for Marquardt. The effect is to 'polish' the Metropolis-Hastings parameter estimates and efficiently move to the 'best' parameter estimates.

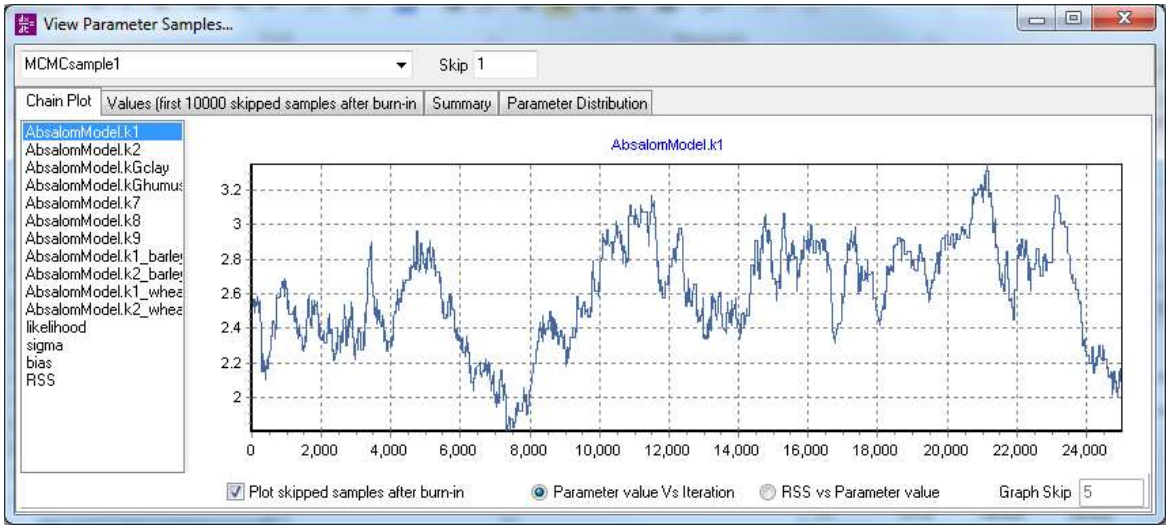
The **`Include Marquardt Polishing Step`** option provides this facility. At the end of the MCMC run Marquardt is undertaken. The options are defined as described above for the individual methods. The interface allows the user to toggle between the options for the 2 methods using the **`Options View`** radiobox.

Running the Search

Once the options are set and the prior distributions specified the metropolis-hastings search can be run. Press the 'Fit' button on the 'Summary' page in the same way as for Marquardt fitting. The current estimates of the parameter values will be shown as the search proceeds, as will the sigma estimate and the calculated residual sums of squares. Of course Metropolis-Hastings is stochastic so uphill moves can (and will) be made, so you will observe the RSS values go up as well as down. The number of steps currently taken is shown with the error count in parenthesis. The number of accepted samples is also continuously updated.



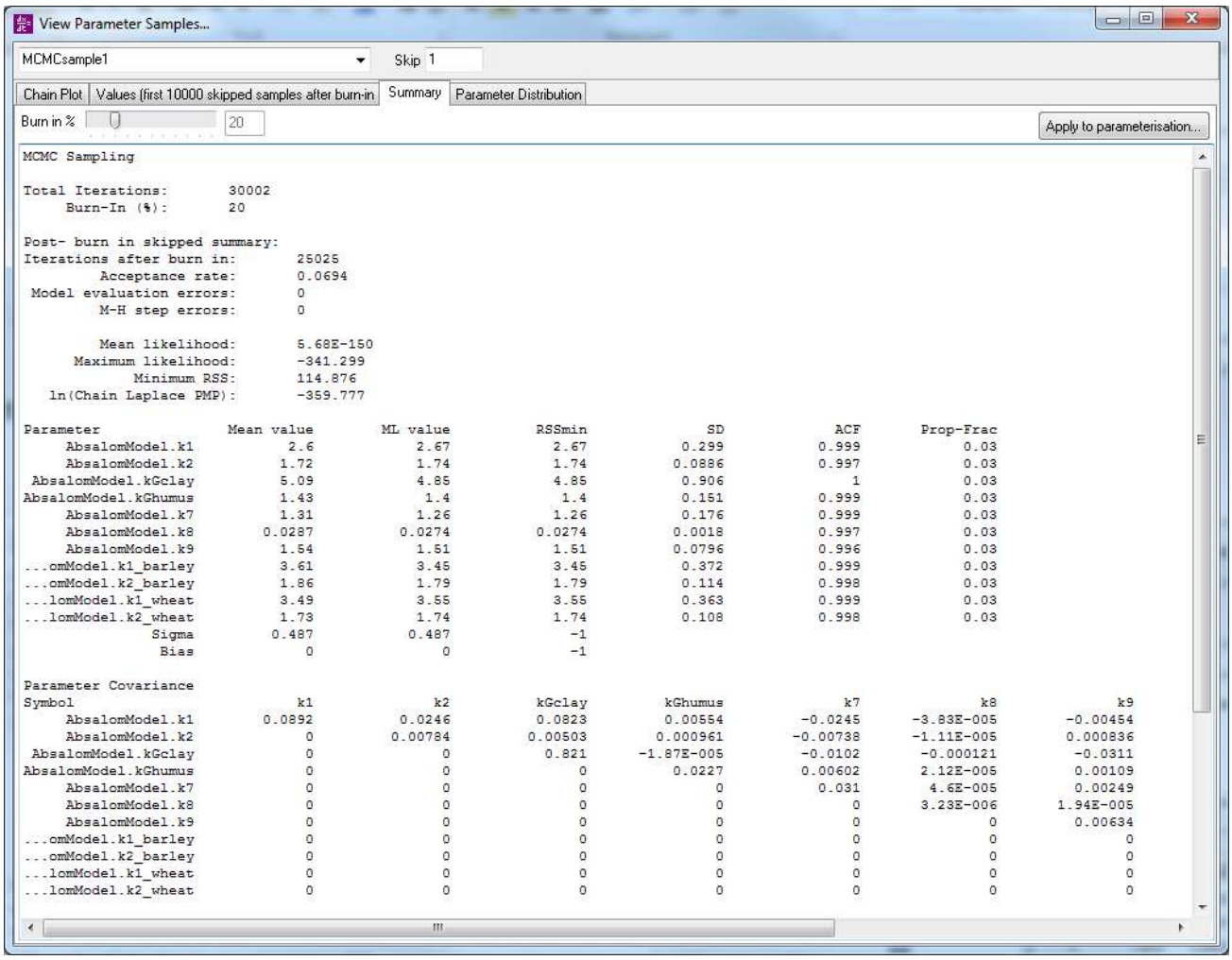
When the search is complete summary results will be displayed within the **`Estimate Parameters`** dialog. However more comprehensive information can be obtained by using the **`Display Parameter Sample...`** option from the view menu. This will allow you to view plots for each parameter (as in the case of k1 in the example below). Simple select the parameter required in the list on the left hand side of the dialog.



Correlation between 'chain' values is an inevitable consequence of the Metropolis-Hastings sampling of the parameters. Often it is useful to 'thin' the chain to reduce this auto-correlation, for example taking every 10^{th} or 100^{th} sample. In OpenModel this can be done by setting the skip interval on the tool bar of the **`View Parameter Samples`** dialog. The values and summary pages both refer to the skipped chain, after burn in.

The **`Values`** page (not illustrated) simply provides a tabulated set of the Metropolis-Hastings parameter samples and related values (likelihoods RSS etc). The data (skipped if appropriate) can be copied on the clipboard or exported to a file from the values tab page for external analysis as required.

The **`Summary`** page calculates means, covariances, and correlations of the parameters from the skipped, post- burn-in phase of the chain (which can be set using the slider at the top of the page). These values can be inserted into a parameterisation of your choice by pressing the **`Apply to parameterisation`** button (overwriting the previous values and distribution data – so be careful).



MCMC Sampling

Total Iterations:	30002
Burn-In (%):	20

Post- burn in skipped summary:

Iterations after burn in:	25025
Acceptance rate:	0.0694
Model evaluation errors:	0
M-H step errors:	0
Mean likelihood:	5.68E-150
Maximum likelihood:	-341.299
Minimum RSS:	114.876
ln(Chain Laplace PMP):	-359.777

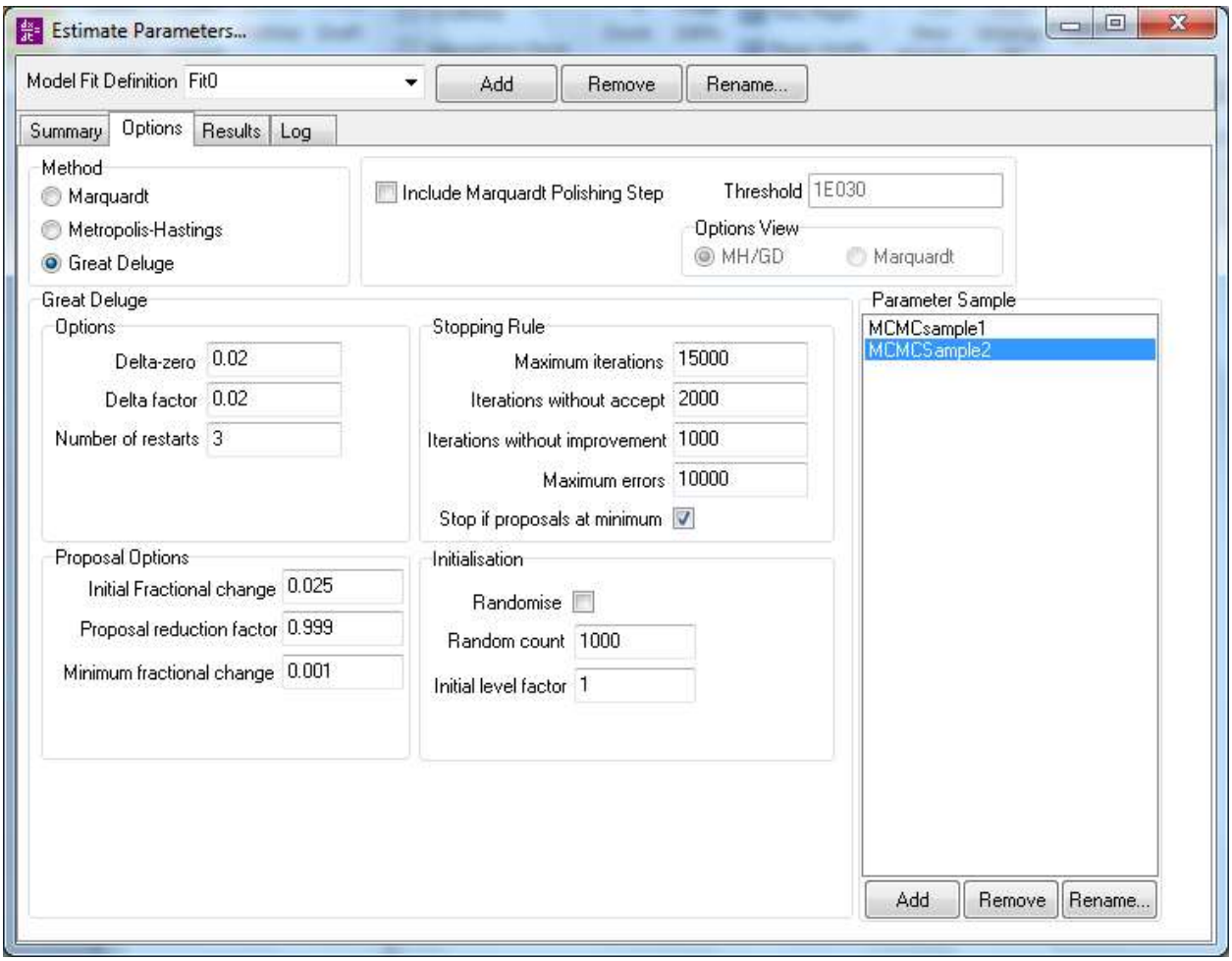
Parameter	Mean value	ML value	RSSmin	SD	ACF	Prop-Frac
AbsalomModel.k1	2.6	2.67	2.67	0.299	0.999	0.03
AbsalomModel.k2	1.72	1.74	1.74	0.0886	0.997	0.03
AbsalomModel.kGclay	5.09	4.85	4.85	0.906	1	0.03
AbsalomModel.kGhumus	1.43	1.4	1.4	0.151	0.999	0.03
AbsalomModel.k7	1.31	1.26	1.26	0.176	0.999	0.03
AbsalomModel.k8	0.0287	0.0274	0.0274	0.0018	0.997	0.03
AbsalomModel.k9	1.54	1.51	1.51	0.0796	0.996	0.03
...omModel.k1_barley	3.61	3.45	3.45	0.372	0.999	0.03
...omModel.k2_barley	1.86	1.79	1.79	0.114	0.998	0.03
...lomModel.k1_wheat	3.49	3.55	3.55	0.363	0.999	0.03
...lomModel.k2_wheat	1.73	1.74	1.74	0.108	0.998	0.03
Sigma	0.487	0.487	-1			
Bias	0	0	-1			

Parameter Covariance

Symbol	k1	k2	kGclay	kGhumus	k7	k8	k9
AbsalomModel.k1	0.0892	0.0246	0.0823	0.00554	-0.0245	-3.83E-005	-0.00454
AbsalomModel.k2	0	0.00784	0.00503	0.000961	-0.00738	-1.11E-005	0.000836
AbsalomModel.kGclay	0	0	0.821	-1.87E-005	-0.0102	-0.000121	-0.0311
AbsalomModel.kGhumus	0	0	0	0.0227	0.00602	2.12E-005	0.00109
AbsalomModel.k7	0	0	0	0	0.031	4.6E-005	0.00249
AbsalomModel.k8	0	0	0	0	0	3.23E-006	1.94E-005
AbsalomModel.k9	0	0	0	0	0	0	0.00634
...omModel.k1_barley	0	0	0	0	0	0	0
...omModel.k2_barley	0	0	0	0	0	0	0
...lomModel.k1_wheat	0	0	0	0	0	0	0
...lomModel.k2_wheat	0	0	0	0	0	0	0

6.3 Great Deluge Algorithm

The great deluge algorithm is a random walk optimisation method, similar in many respects to simulated annealing. However, according to its adherents, its setup is less problem specific making it easier to apply in practice. Burke et al (2004) give a description of the approach.



The procedure operates through an analogy between the merit function value and a notional water level which falls during the courses of the procedure. The initial water level is set at the initial merit function value multiplied by **`Initial Level Factor`** which can be set by the user (default value 1.0).

The range of the parameters is set using the constraints defined in the Prior Parameterisation.

By default the procedure starts with the parameter values set in the defined parameterisation. Alternatively, if the **`Randomise`** box is checked, a latin hypercube sample of **`Random Count`** evaluations is performed and the parameter set with the lowest merit function value is taken as the initial position.

A random move is made from the current location in the parameter space using a proposal distribution. The model's merit function is evaluated at the new parameter location. If the new value of the merit function is less than either the current water level or the current value of the merit function the move is 'accepted'. If not the move is rejected and a new random walk made from the same location.

After an accepted move the water level is reduced by an amount Δ using the relationships below

```
if WaterLevel>Merit then
  Δ = max(Delta0,Deltafactor*(Waterlevel-Merit))
else
  Δ = Delta0
```

where **`Delta0`** and **`Deltafactor`** are user defined values. The default value of **`Deltafactor`** is 0.02, so after an accepted move the water level is reduced by 2% of the difference between the water level and the merit value.

The proposal for each parameter is defined as a fractional value, set initially to **Initial Fractional Change** (default 0.025). On each rejected move this fractional width is reduced by the **Proposal Reduction Factor** (default 0.999) until it reaches the **Minimum fractional change**. So as the procedure advances the moves of the parameters are increasingly restricted.

The procedure stops when one of the following conditions, controlled through the values defined in the stopping rule box, apply:

- The water level falls below the lowest value of the merit function
- The maximum number of total iterations is exceeded
- The maximum number of iterations without an acceptance is exceeded
- The maximum number of iterations without an improvement in the best value of the merit function is exceeded
- The maximum number of model errors is exceeded.

In addition if the **Stop if proposals at minimum box** is checked the procedure stops if the proposals for all the parameters have been reduced to their minimum value; the case where no further significant movement of the parameters is possible.

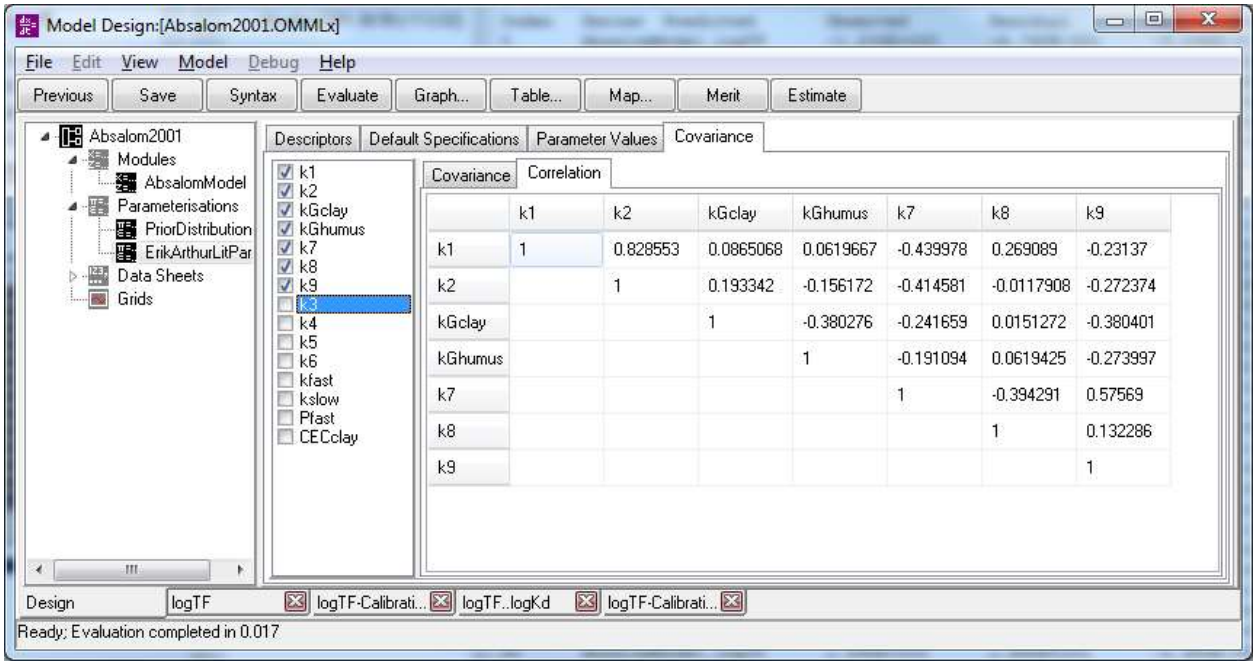
As with the metropolis-hastings method you can polish the result using Marquardt by checking the appropriate box.

7. Parameter Covariance

The parameter covariance and correlation view (Covariance tab of the parameterisation view) provides a means to inspect the covariance between any parameters in the model. These co-variances will have arisen if parameters have been simultaneously estimated via model fitting.

To use the view select the required parameters in the available parameters list, they will then be included in the covariance/correlation matrix display as shown below. Correlation and covariance values are available on separate tabs. Parameters without a covariance/correlation value are shown as -999.

In the example below only those parameters which have been fitted are selected for display.



8. Uncertainty/Confidence Interval Estimation

Uncertainty estimates on model predictions can be made using an 'uncertainty' object. Select **`Model|Uncertainty...`** to display the uncertainty dialog box (shown below for the example of **`Absalom2001.OMMLx`**).

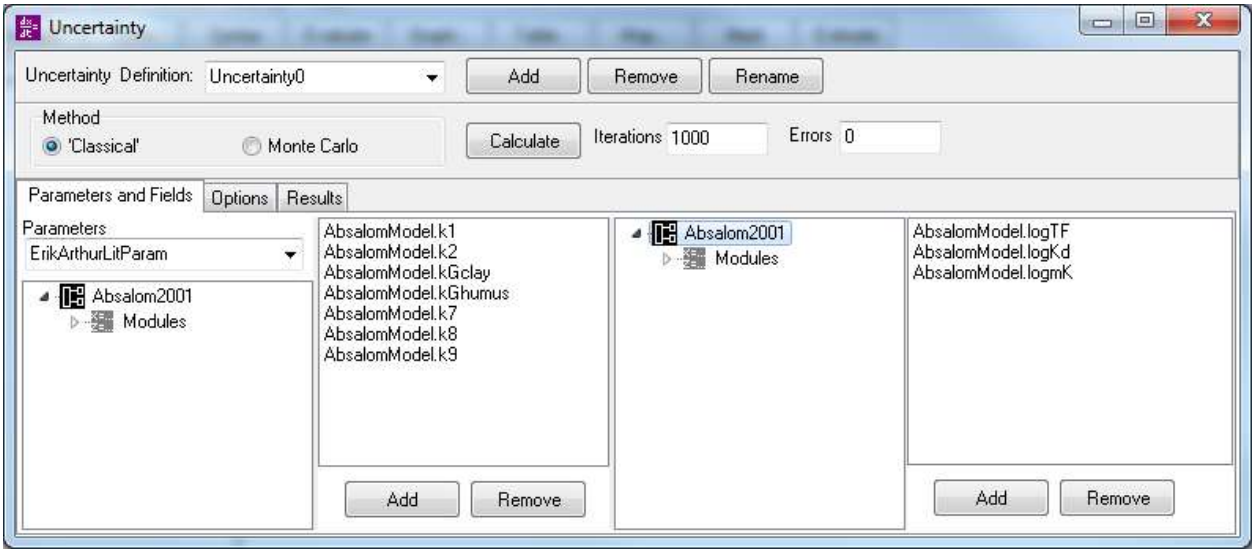
As with any other OpenModel control objects you can select which object to display using the drop down list on the top panel of the dialog box, you can also add, remove and rename objects as required.

The implemented the methods are:

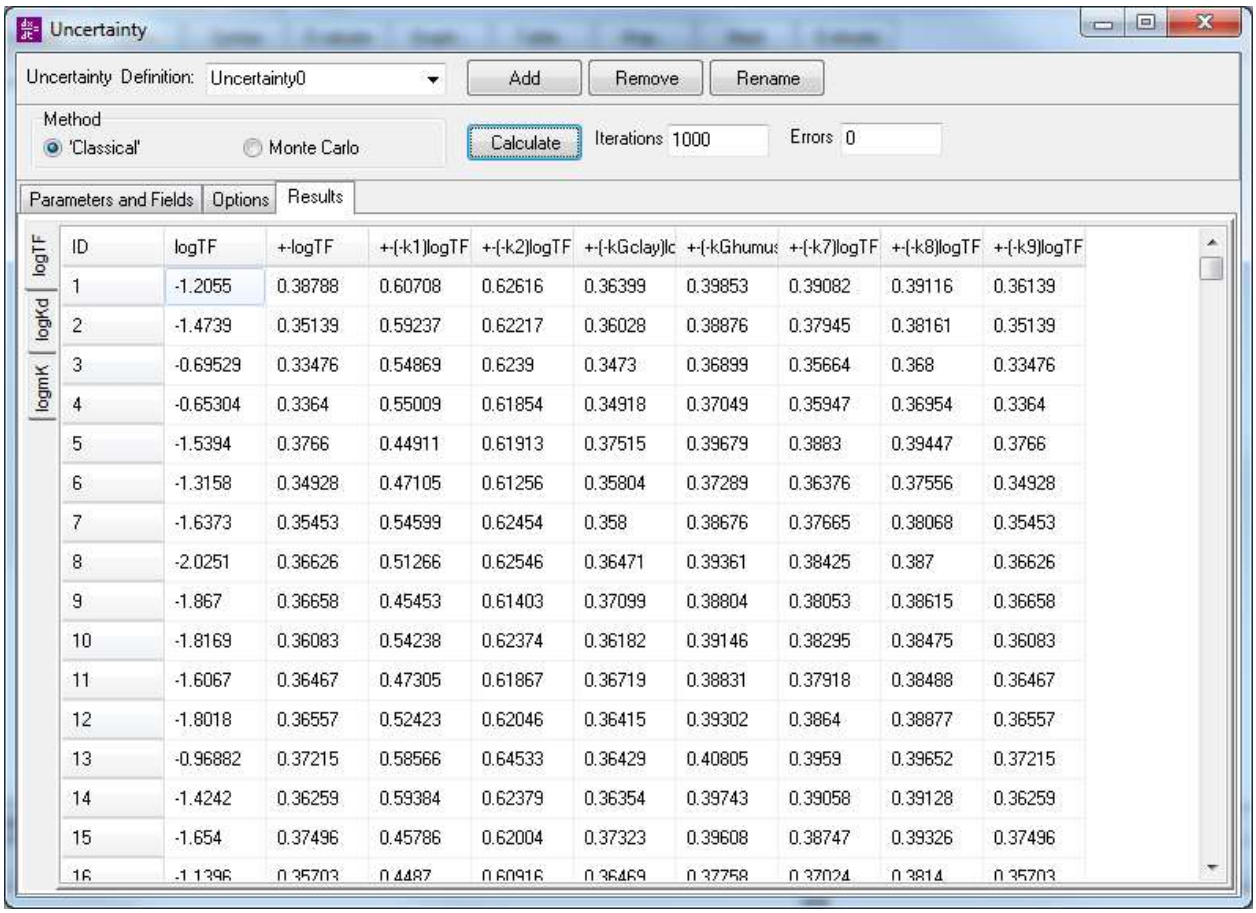
- a classical combination of the parameter standard errors (assuming normal distribution) of the selected parameterisation. This can optional use the full covariance matrix (if available) to allow for interaction between parameters.
- monte-carlo sampling of the selected parameters assuming that the parameters are independent.

Parameters included in the uncertainty estimation are selected from the left hand navigation tree and added the adjacent parameters list. Model variables whose uncertainty is required are selected on the right hand side of the dialog box.

When the Calculate button is pressed the uncertainty estimation is undertaken using the options selected on the Options tab.

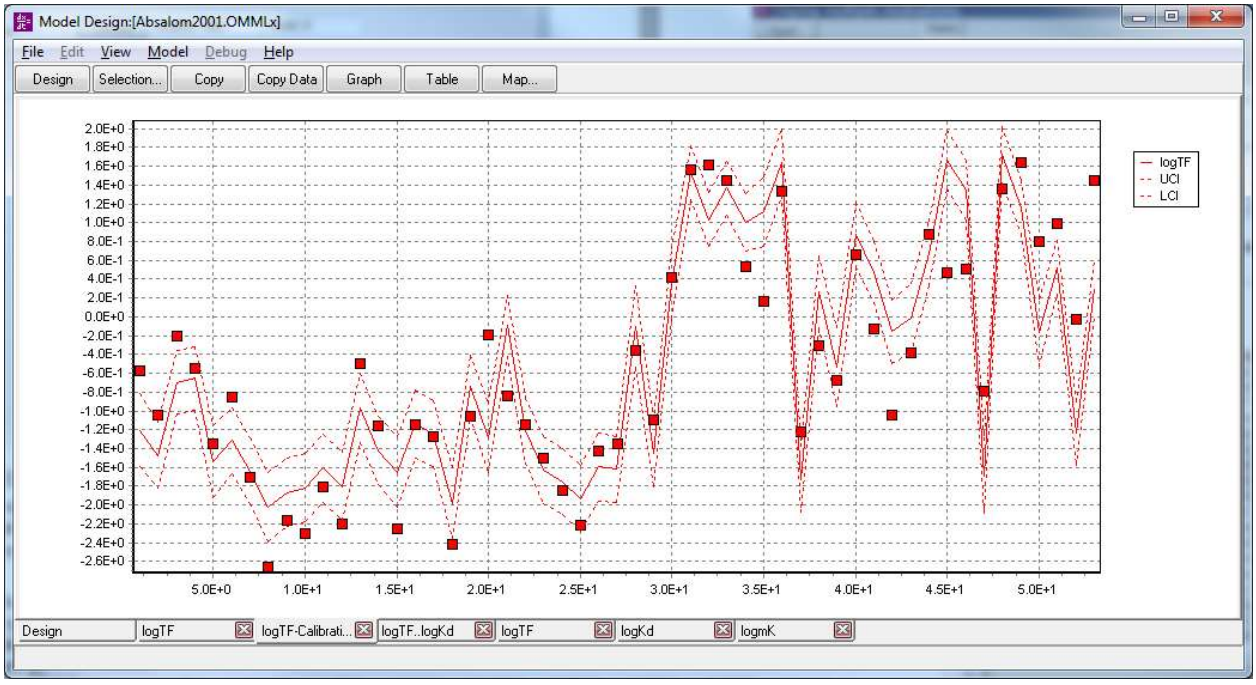


Results for a 'classical' analysis are displayed on the Results tab and typical output is shown below for the **`Absalom2001.OMMLx`** model. Each predicted series is shown on its own tab view. The predicted value of the variable is shown together with the uncertainty estimate (68% confidence interval). If co-variances have been included in the calculation the uncertainty estimate arising from the exclusion of each parameter from the calculation is also shown in additional columns (1 for each parameter).



ID	logTF	+-logTF	+-(-k1)logTF	+-(-k2)logTF	+-(-kGclay)lc	+-(-kGhumus	+-(-k7)logTF	+-(-k8)logTF	+-(-k9)logTF
1	-1.2055	0.38788	0.60708	0.62616	0.36399	0.39853	0.39082	0.39116	0.36139
2	-1.4739	0.35139	0.59237	0.62217	0.36028	0.38876	0.37945	0.38161	0.35139
3	-0.69529	0.33476	0.54869	0.6239	0.3473	0.36899	0.35664	0.368	0.33476
4	-0.65304	0.3364	0.55009	0.61854	0.34918	0.37049	0.35947	0.36954	0.3364
5	-1.5394	0.3766	0.44911	0.61913	0.37515	0.39679	0.3883	0.39447	0.3766
6	-1.3158	0.34928	0.47105	0.61256	0.35804	0.37289	0.36376	0.37556	0.34928
7	-1.6373	0.35453	0.54599	0.62454	0.358	0.38676	0.37665	0.38068	0.35453
8	-2.0251	0.36626	0.51266	0.62546	0.36471	0.39361	0.38425	0.387	0.36626
9	-1.867	0.36658	0.45453	0.61403	0.37099	0.38804	0.38053	0.38615	0.36658
10	-1.8169	0.36083	0.54238	0.62374	0.36182	0.39146	0.38295	0.38475	0.36083
11	-1.6067	0.36467	0.47305	0.61867	0.36719	0.38831	0.37918	0.38488	0.36467
12	-1.8018	0.36557	0.52423	0.62046	0.36415	0.39302	0.3864	0.38877	0.36557
13	-0.96882	0.37215	0.58566	0.64533	0.36429	0.40805	0.3959	0.39652	0.37215
14	-1.4242	0.36259	0.59384	0.62379	0.36354	0.39743	0.39058	0.39128	0.36259
15	-1.654	0.37496	0.45786	0.62004	0.37323	0.39608	0.38747	0.39326	0.37496
16	-1.1396	0.35703	0.4487	0.60916	0.36469	0.37758	0.37024	0.3814	0.35703

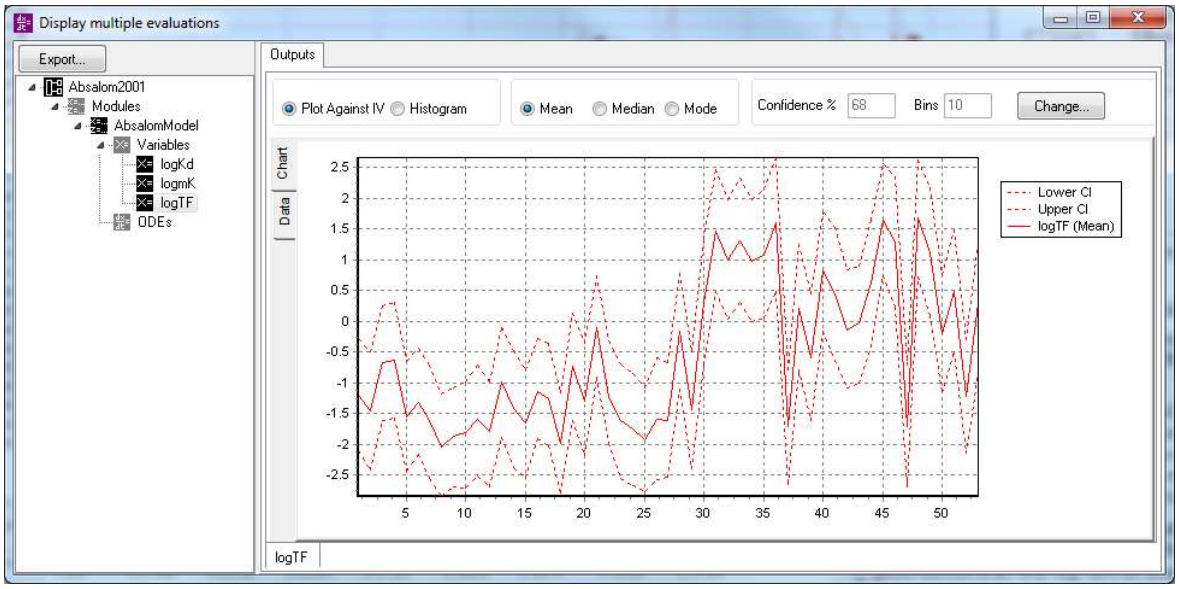
These results will also be shown in any tables where the variable is displayed. The overall confidence interval (i.e. the overall +-standard error shown in the table view above) will be shown in any graphs of model variables whose confidence interval has been calculated (see below).



Monte carlo analysis involves multiple running of the model with each individual model run being performed using parameter values sampled from the distributions defined in the selected parameterisation object. It is quite common for these samples to result in an infeasible parameter set which lead to a failed model run, therefore OpenModel will

ignore errors and try again. The number of such failed runs is shown in the error count box. If the number of errors is high (as judged by the user) it may be wise to check the setup of the model.

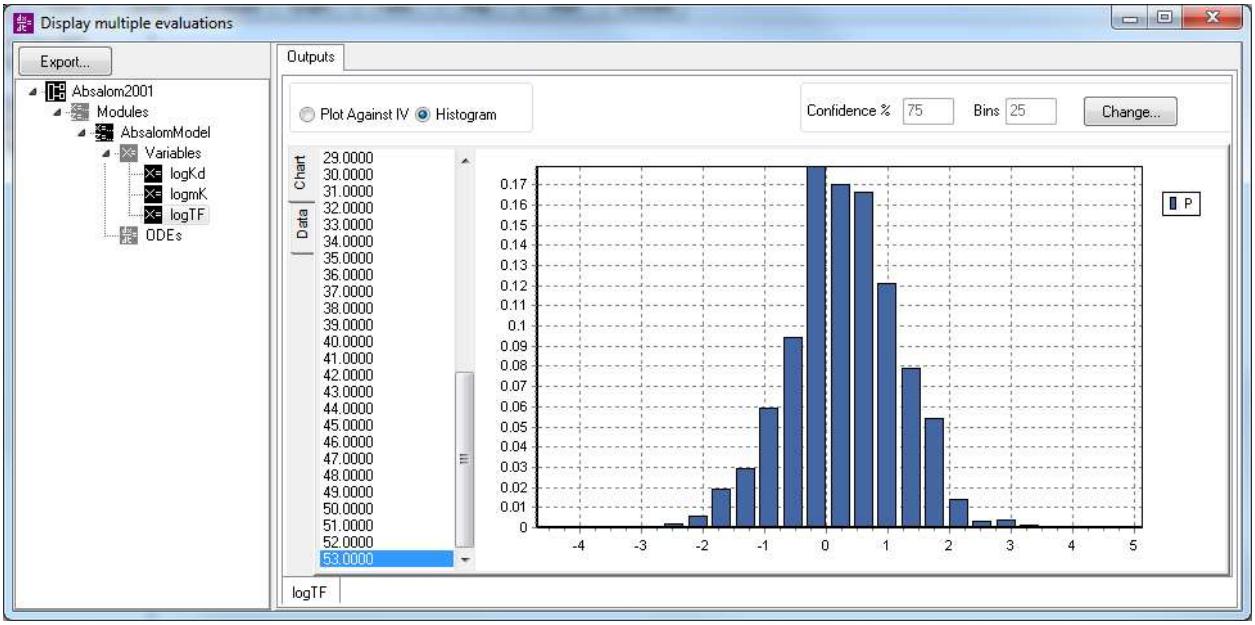
Each time the model is successfully run the data for the fields in the dialog boxes right hand list is saved within OpenModel. This data can be viewed by clicking on **`View|Display Multiple Evaluations`**. This displays the dialog box shown below using the model **`Absalom2001.OMMLx`** as an example.



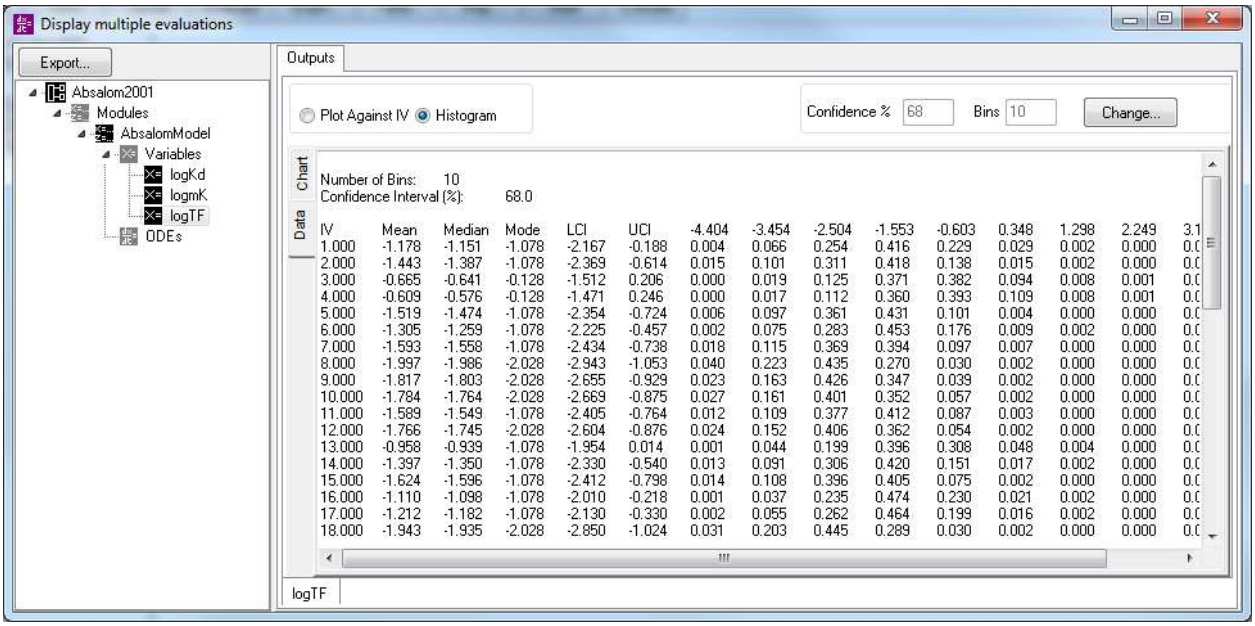
The left hand tree view shows the model fields for which multiple model run data are available (three fields in the example shown). Double click on the field of interest to display its data.

The data are 'binned' over the range of the values obtained. The number of bins can be adjusted by clicking on the **`change...`** button. This also allows the confidence interval range, which is used in some of the views, to be set. In the example below 10 bins have been used and the data are shown as the mean over the simulation with 68% confidence intervals. The median or modal values can also be shown.

The default view (above) is a plot of the model field against the models independent variable. A histogram plot of the field at specific values of the independent variable can also be shown (using the radio buttons at the top left of the view) as shown below.



The underlying data can also be shown by selecting the 'data' tab on the left hand side of the view. This gives the binned summaries as copy/paste-able text.



It is important to note that the monte carlo results created DO NOT take account of any parameter covariance.

9. Variable Replacement

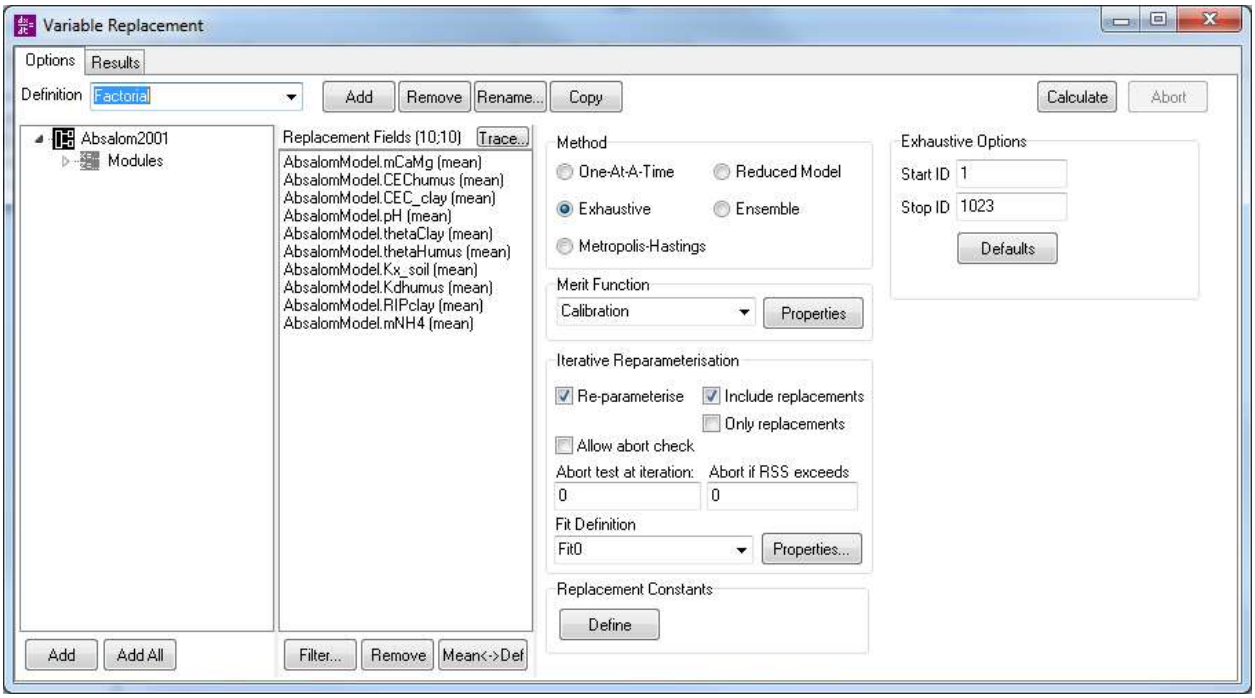
One of the main motivations for the development of OpenModel was to enable our work on variable replacement (or model reduction/simplification). The method involves replacing variables with constants (specifically the mean value of the variable in an unsimplified run of the model). The methods are described more fully in Cox et al (2006) and Crout et al (2009)

Variable replacements can be made using a 'replacement' object. Select **`Model|Variable Replacement...`** to display the dialog box shown below for the example of **`Absalom2001.OMMLx`**.

As with other OpenModel control objects you can select which object to display using the drop down list on the top panel of the dialog box, you can also add, remove and rename objects as required.

Specifying the replacements requires you select which variables you wish to include in the analysis (select in the left hand navigation tree and add to the selected list) and choose the search method.

A merit object must be specified to enable inter-comparisons of the models created. If required the models can be re-parameterised, in this case a fit object must be specified (this overrides the selected merit function.



When the **`calculate`** button is pressed the search procedure is executed and the results are shown on the **`Results`** tab. It should be noted that the procedure can be very iterative.

There are 5 modes although only 3 of these actually compare different forms of the reduced model.

- One At A Time. Variables are replaced individually. This allows a rapid screening for potentially interesting candidate replacement variables.
- Exhaustive. All the possible replacement permutations are considered (see Cox et al 2006 for details). This is very reliable way to search the replacement possibilities but can be computationally expensive.

- Metropolis-Hastings. This makes a random walk through the replacement space using defined likelihood options to control acceptance-rejection of individual steps.
- Reduced Model. This simply allows a particular combination of replaced variables to be run (out of a previously evaluated variable replacement definition).
- Ensemble. This allows predictions to be made over an ensemble of replaced models. The results can be viewed by selecting **`View|Display Multiple Evaluations...`**

The procedure compares the model performance of the different replacement combinations which can be iteratively re-parameterised if appropriate.

If the replacement does not involve re-parameterising each case is compared using the specified Merit Function. If the models are re-parameterised (i.e. **`Re-parameterise`** is checked) the selected Fit Definition is used to specify the fitting procedure (e.g. which parameters are to be fitted, by what methods and so on). In this case the merit function defined on the variable replacement form is not used.

Iterative Reparameterisation

Include replacements: if this is checked then the replacement constants used in the model reduction are included in the fitting procedure.

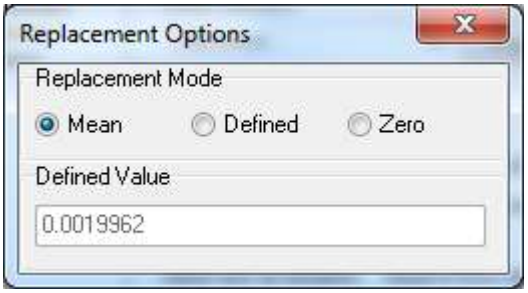
Only replacements: if this is checked only the replacement constants are included in the fitting procedure (the model parameters themselves are not adjusted)

Allow abort check: if this is selected the fitting procedure checks the merit function residual sums of squares at the fitting iteration given by **Abort test at iteration** and the fit stops if the value given by **Abort if RSS exceeds** is exceeded. Variable replacement with re-parameterisation can be very computationally intensive so this option provided a way to abandon fits which 'aren't going anywhere'.

Replacement Constants

Replacement constants have to be defined for each variable considered in the analysis. By default the replacement constant is taken as the mean value the variable attains over the evaluation of the full model (see Crout et al 2009 for full details). This is denoted by the text **`(mean)`** in the reduction fields list.

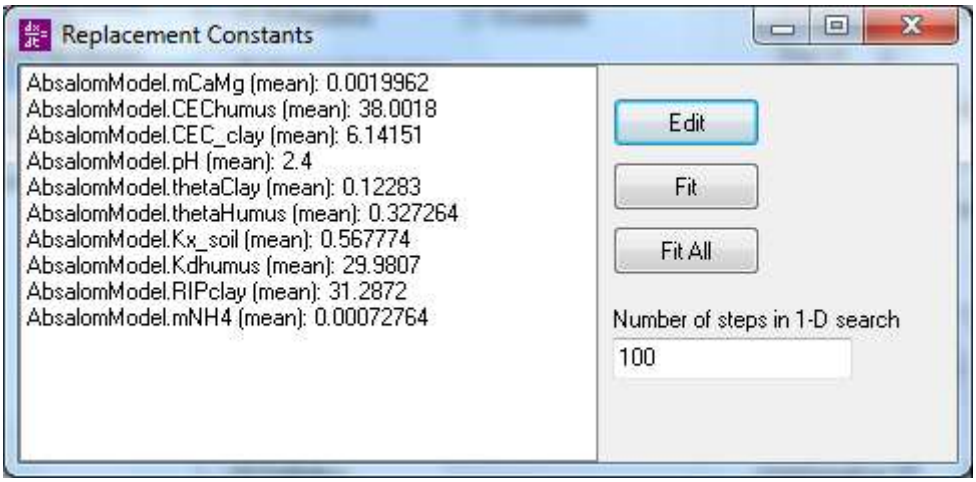
Alternative choices for the replacement constants are 'defined' and 'zero'. These can be set by double clicking on the variable in the reduction field list to display the dialog box shown below. If **`Defined`** is selected the **`Defined Value`** field can be edited. The use of **`Zero`** as the replacement constant is a useful way to remove parts of the model design during the analysis.



It is often convenient to quickly switch the replacement constant option between Mean and Defined and this can be accomplished for all the fields simultaneously using the **`Mean<>Def`** button below the reduction fields list.

The replacement constant options and defined values can also be edited through the dialog box shown below which is displayed when the **`Define`** button is pressed in the

`Replacement Constants` box. Double clicking on a variable in the list displays the replacement options (as for the main reduction fields list). In addition this dialog box allows the defined value to be fitted (**`Fit`** or **`Fit All`**) by finding the replacement constant which gives the minimum RSS. This fit is undertaken independently for each variable considered with the rest of the model in its un-reduced state. A simple line search is used over the range the variable takes in an evaluation of the full model.

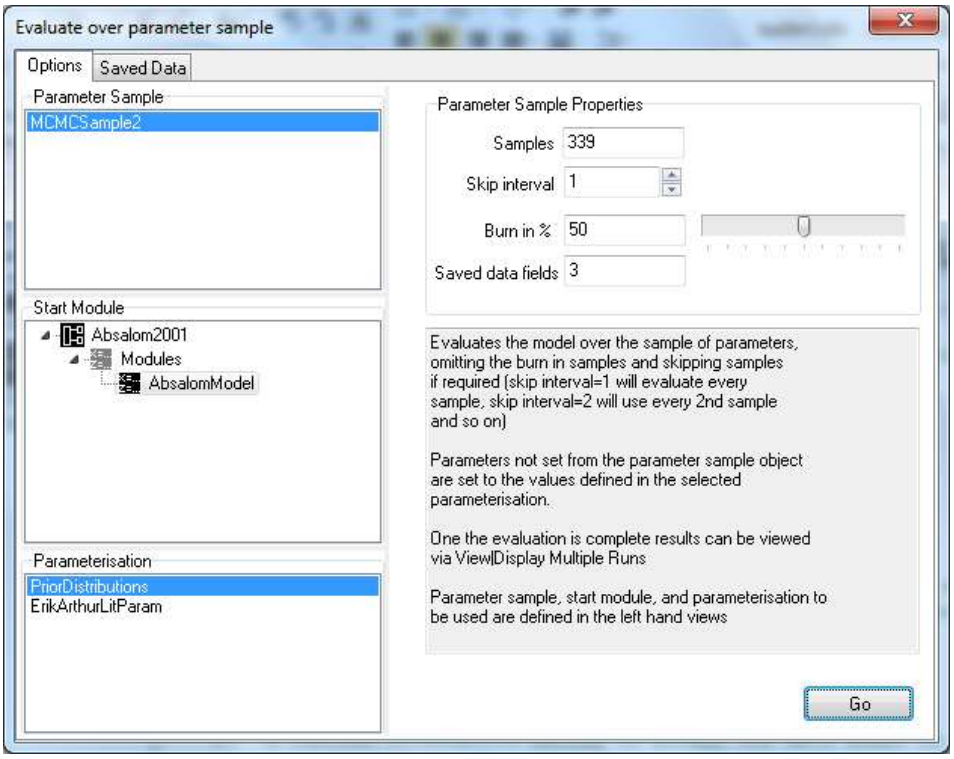


10.Evaluating over a Parameter Sample

The output of a metropolis-hastings search of the parameter space is collected within a parametersample object. This is a list of the accepted parameter values arising from running the chain. If the metropolis-hastings has been appropriately configured and if the chain is sufficiently long these parameter estimates can be regarded as representative samples of the posterior distribution (although thinning of the chain may be recommended as auto-correlation between adjacent steps in the chain is very likely). A particular practical benefit is that the parameters have been simultaneously sampled so the influence of covariance between parameters is included.

Once a set of parameters is available it can be used to make ensemble predictions with the model over all the parameter estimates within the parametersample object. In principle this is then a posterior distribution of the model prediction. Such a prediction does not have to be for the same setup as the original setup used for the model calibration, provided that the sample of parameter values remains appropriate for the purpose.

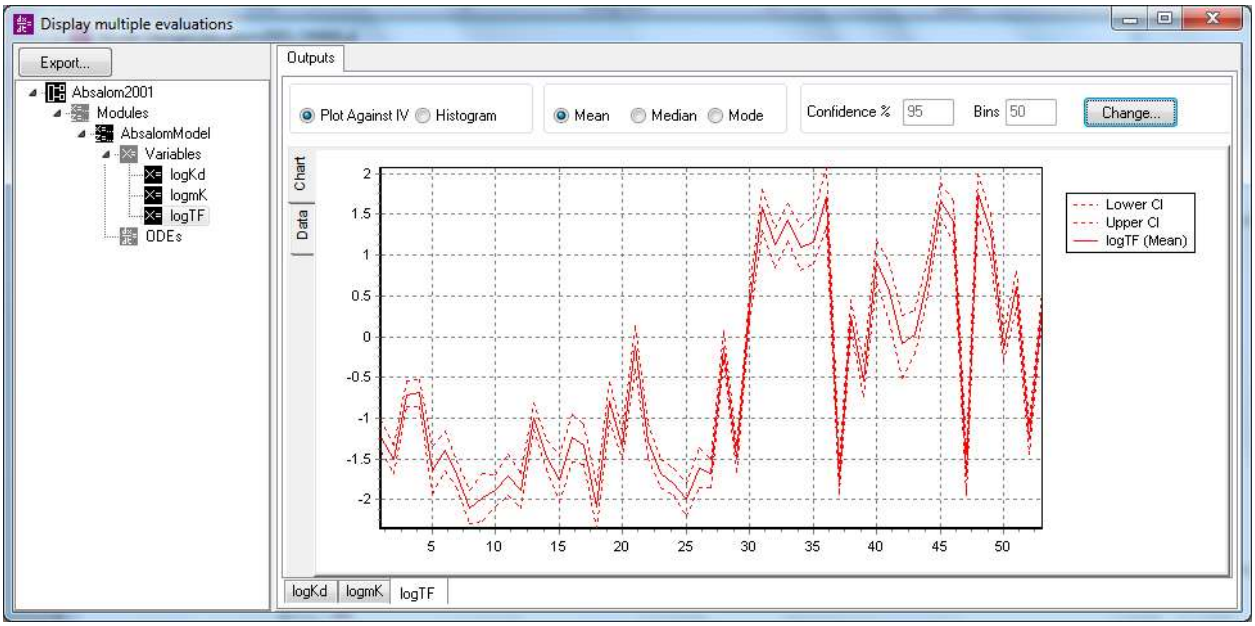
Select **`Model|Evaluate`** Sample from the menu to display the dialog shown below. This is a case where the parameter values in **`ParameterSample1`** are to be used in conjunction with the parameter values defined in the parameterisation **`Param1`**. The parametersamples properties are displayed and the user can adjust the **`Burn in %`** (this initial proportion of the parametersample is ignored in the evaluation) and the 'skip interval' (allowing the parameter samples to be thinned, either to reduce the amount of computation or to reduce the influence of correlation in the chain). In the example below a skip interval of 1 and a burn in of 50% are selected.



The multiple evaluation will be performed when the **`Go`** button is pressed. The main design window will display a progress report and indicate if any errors have occurred.

The results of multiple evaluations are stored (memory can be restricting if the model and/or the number of evaluations are large) and can be viewed by selecting **`View|Display Multiple Evaluations`** from the menu (exactly as for viewing the results of monte carlo uncertainty analysis).

An example of the display multiple evaluations form is shown below. You select which model variables you wish to display by selecting them in the left hand tree, simply double click a variable and it will be displayed as a tabbed page in the right hand side of the form. In the example below the variables **logCF**, **logKd** and **logmk** have been selected.



For each variable model results will have been stored for each of the model evaluations undertaken, which may be many 1000s. These are 'binned' and the contribution of each individual model evaluation is summed for each bin, such that the value for each bin is the proportion of the model evaluations that lie in each bin. In the example above the results have been shown plotted against the model independent variable. The central value is the mean **logCF** with upper and lower confidence intervals calculated using the binned values. It is possible to plot the median or modal value rather than the mean by selecting the appropriate radio-button from the tool bar. The width of the confidence interval and the number of bins can be changed by clicking the **Change** button, also on the tool bar. Increasing the number of bins will increase the 'resolution' of the output but at the expense of reducing the number of evaluations contributing to each bin, so the results can become noisier. In practice suitable settings depend on the particular application.

The graphical output can also be presented as a histogram for each value of the model independent variable (essentially looking at a slice across the output shown in the IV plot). In this view the left hand list view enables the required value of the model's independent variable to be selected.

You can use the tab controls on the left hand side of the output view to switch between the graphical outputs described above and a textual summary suitable for copying to the clipboard (via a right click) and hence to excel or other applications.

The entire output for each of the selected output tabs (i.e. the raw model results for each of the multiple evaluations) can be exported to a text file for further analysis using the export button. If a large number of evaluations have been performed this can produce a very large file!

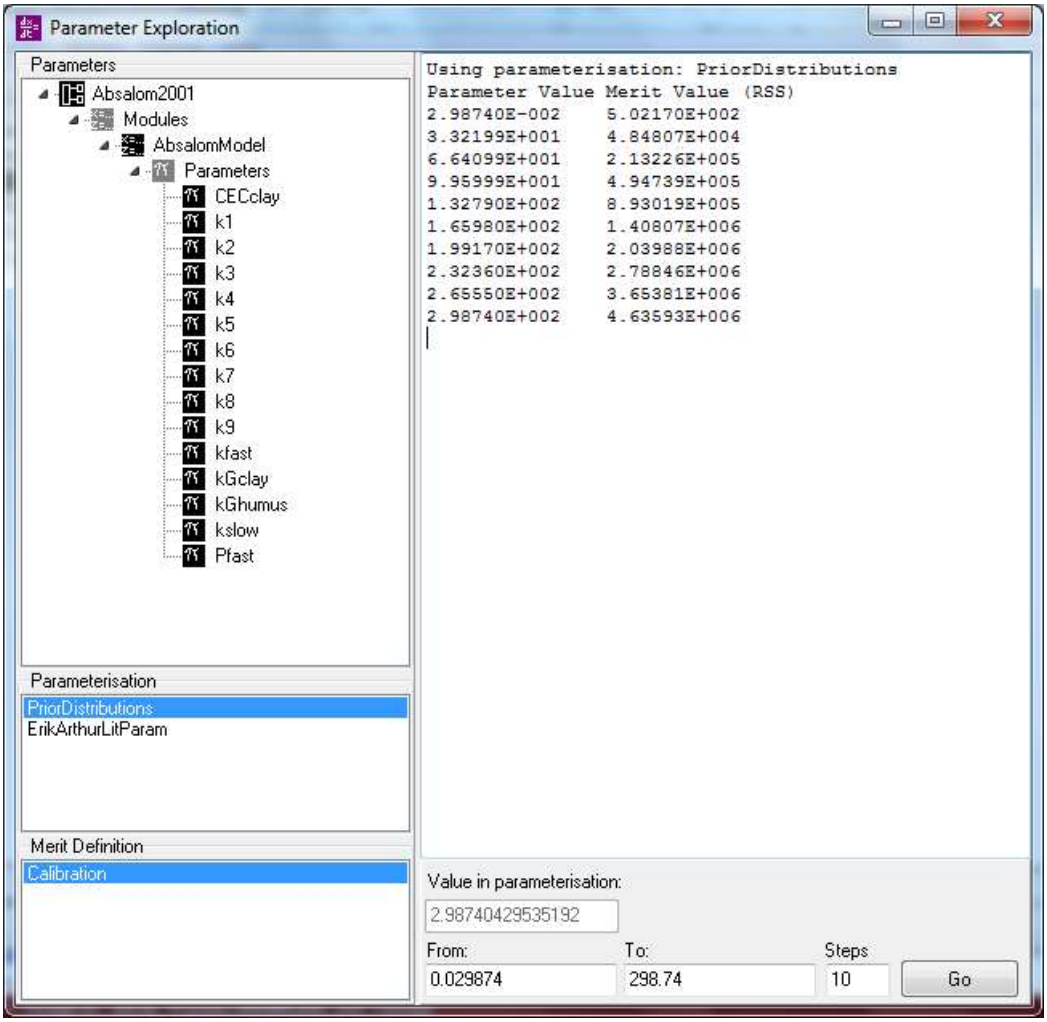
11.Parameter Exploration

The parameter exploration facility (**Model|Parameter Exploration**) provides a simple means to vary a defined parameter over a range of values to find the effect this has on a given merit function.

The dialog box lists the model's parameter values in a tree, select the required parameter (**CECclay** in the example below).By default the **From** and **To** fields are set to the allowed range of the parameter in the selected parameterisation.

The number of steps in the parameter values that will be used is set in the **Steps** edit box.

When the **Go** button is clicked the model is run using the selected parameterisation and setting the selected parameter to the appropriate value from the sequence. The result value of the merit function is presented together with the corresponding parameter value in the output view on the right hand side of the dialog box.



12.Analysing Model Structure

OpenModel has some simple tools (**`Trace Symbol`** and **`Categorise Symbols`**) for enabling you to work through the structure of a model. As model size increases this can be quite useful and reduces the amount of scrolling up and down through code required for model development.

- **Trace** allows you to view the sequence of relationships and symbols on which a given symbol depends, and in turn which it influences.
- **Categorise** lists symbols depending on their type and use within the model.

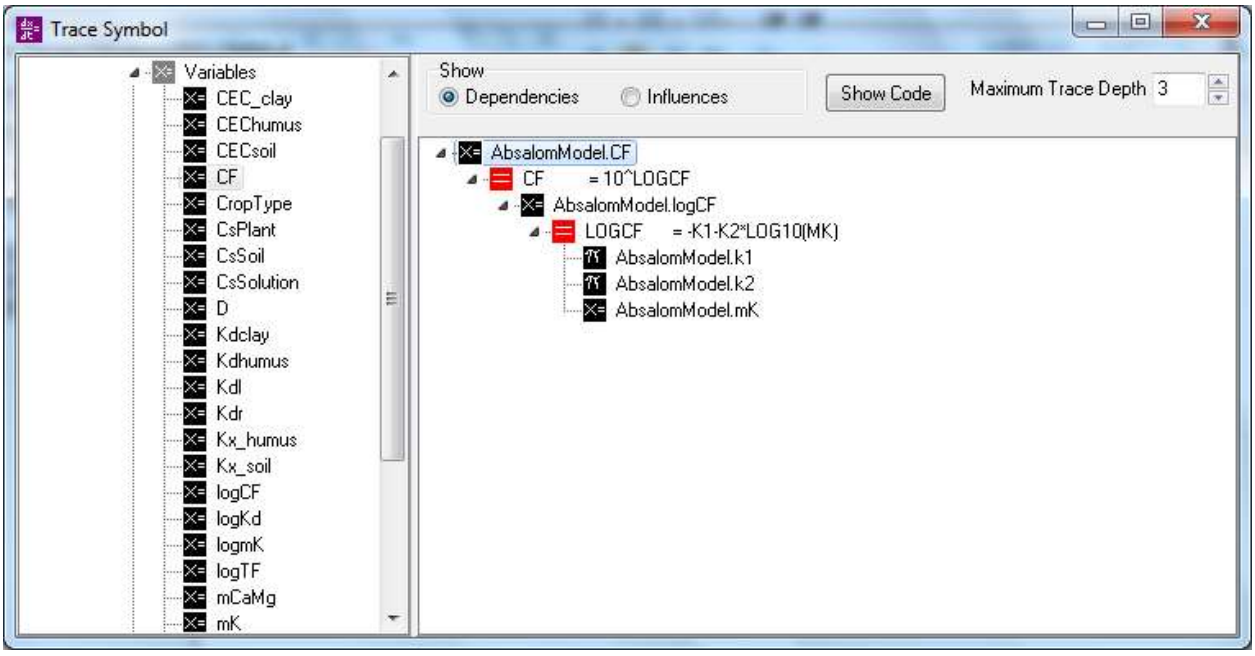
Both these tools require the model to have been compiled (and for some features to have been evaluated)

The trace dialog (**`View|Trace`**...) is shown below. Clicking a symbol in the left hand view displays its dependencies/influences and relationships to other model variables (you can switch between Dependencies or Influences).

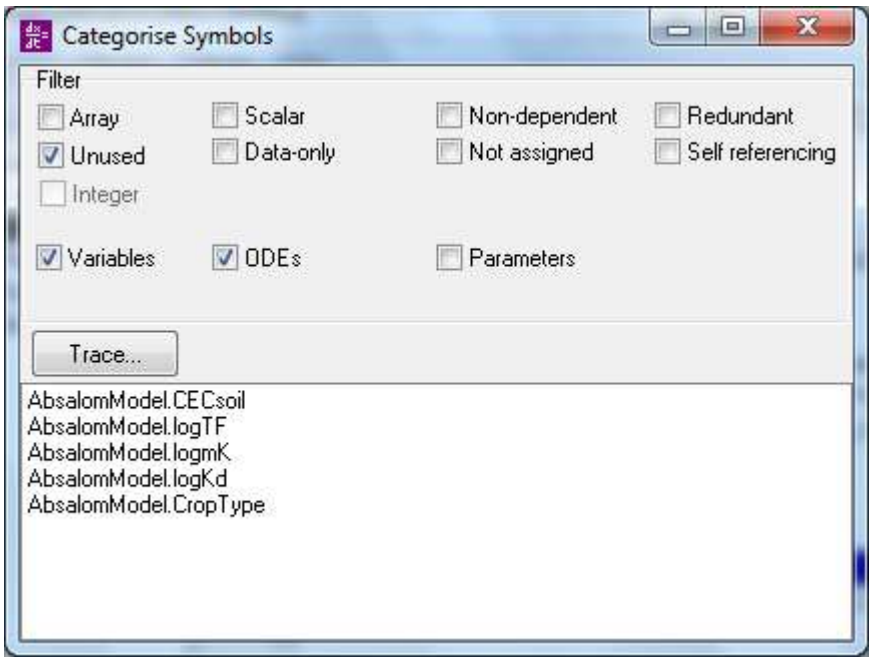
Double clicking on a symbol in the dependency/influence tree will display the trace for that symbol.

Double clicking on an assignment relationship (or clicking **`show code`**) in the trace tree will open the module view at that line in the model code.

In most models the symbols are ultimately all connected, so the trace tree can become very deep. As a result the trace procedure can become slow, especially if the symbol traced is used in many places in a large model. To control this, a **`maximum trace depth`** is defined. In the example below the **`maximum trace depth`** is set to 3 (i.e. variables up to 3 times removed are displayed in the trace tree).



An example of the **`Categorise Symbols`** dialog is shown below. This simply provides a list of model symbols subject to certain categories. Check the boxes for the different categories and the corresponding symbols are listed. These can be traced (as above) by either double clicking on the symbol in the list or by clicking the **`Trace`** button.



The categories available are:

- Array; this denotes that the symbol is an array
- Scalar; this denotes that the symbol is not an array
- Non-dependent; this denotes that the symbol does not depend on any other model symbol (e.g. x=1)
- Redundant; these are symbols which, usually for good reasons, are calculated from just one other variable (and perhaps fixed parameter values). For example the units of a variable may be transformed. They are redundant in that they are entirely dependent on another model symbol.
- Unused variables; these have values assigned to them but are not used for any other step or calculation in the model. Often the function of unused variables is as a model diagnostic or output. They may be relevant to the model application but not part of the interconnected model. Sometimes they are just unused, maybe some aspect of the model has been discarded but the symbols not removed, or perhaps something is incorrectly specified!
- Data-only; this denotes that the symbol depends only on values read from a data sheet
- Not assigned variables; although defined in the code these symbols never have a value assigned to them and therefore never play any part in the model calculations.
- Self referencing; this denotes that a model variable depends to some extent on itself (e.g. x <-- x + some calculation). In some models state variables might be represented as ordinary differential equations, sometimes as variables which accumulate by iterating over fixed time steps (i.e. self-referencing).

13.Using Spatial Data

OpenModel can read spatial or grid data from external data files and spatial models created using a raster approach based on 2-D arrays.

The example model ChildandSpatialDemo.OMMLx calculates the deviation from the monthly mean temperature for each location is a 0.5x0.5 degree latitude/longitude grid.

13.1 Reading Grid Data

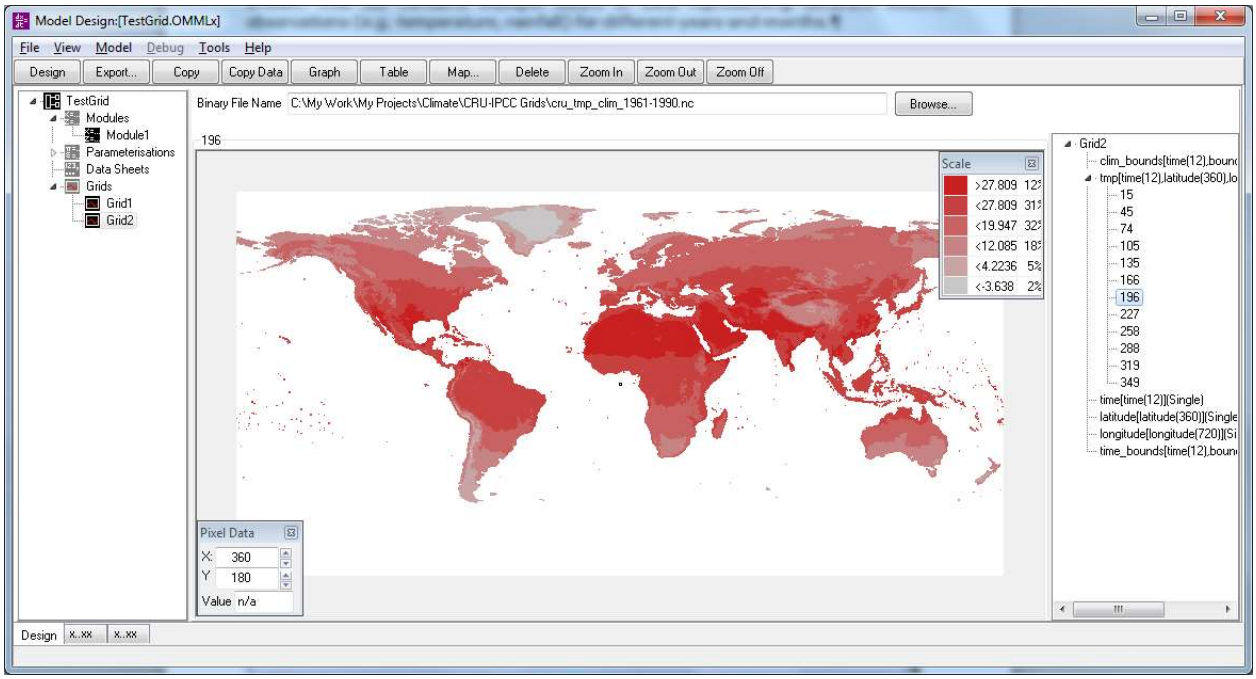
Two data file formats are available

- Native OpenModel Format (OMgrid). This is a simple binary format (specified below) which can be used to create multi-dimensional data sets. OpenModel can create files of this type from text files (see below).
- netCDF format. This is a widely used format for large multidimensional datasets, it seems to be especially popular in the earth systems science community. netCDF is fully documented by Unidata (www.unidata.ucar.edu/software/netcdf/) and there are numerous software applications available to view or manipulate netCDF data. netCDF files can be used to manage sophisticated multi-dimensional data and the file contents can be documented through properies held within the file.

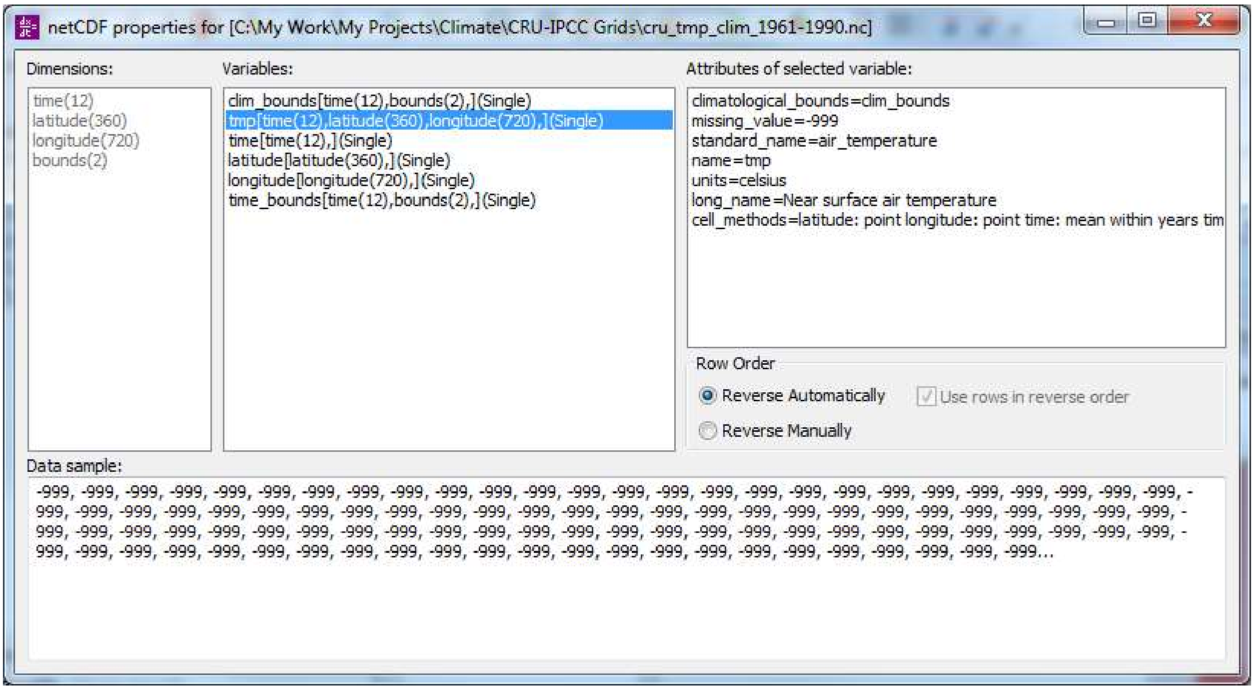
In the example model shown below (ChildandSpatialDemo.OMMLx) global monthly mean temperature data are held in a netCDF file which has been linked to the model using a grid object.

Grid data are added to a model by inserting a grid object into the model tree (right click the grid node in the model tree on the left hand side of the OpenModel design view). The filename is then entered by browsing to the required file. Files with the extension .OMgrid are assumed to be native OMgrid format and file with the extension .nc are assumed to be netCDF files. If you don't use these file extensions OpenModel will not correctly deduce the type of grid data you are using.

The structure of the data within the grid object is displayed in the right hand tree. 2-dimensional (and therefore map-able) data sets are shown as the lowest level nodes in the three. To display a map of the desired node, click on the associated object. In this case temperature ('tmp') at time value 196

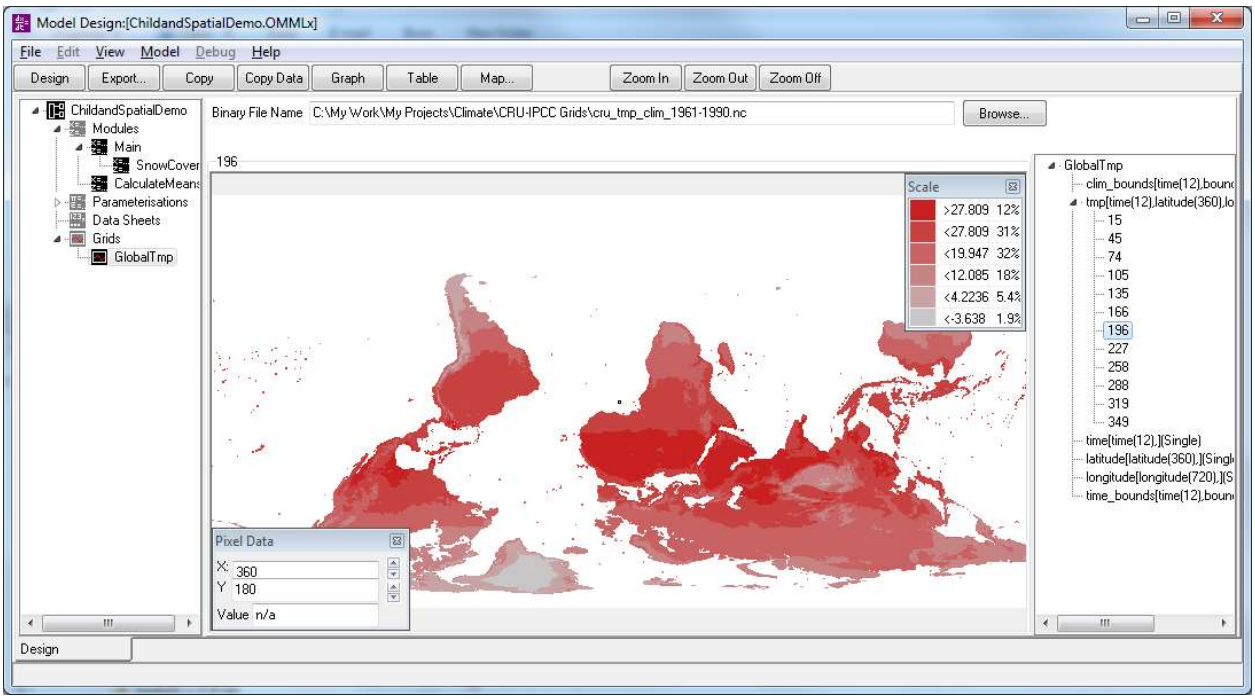


In the case of netCDF objects the self documenting properties and attributes can be displayed using a right click on the grid object in the left hand tree view. This will display a dialog box such as the one shown below. In this case the values associated with the latitude grid positions are shown.



In this example the variable **tmp** is selected to show its attributes and a sample of the values it contains.

There specific options to control row order. The map display in OpenModel takes the row and column positions of the top left grid cell to be 0,0. In a netCDF grid this may not be the case, it will depend on the values of the dimensions of the grid. In the example shown the vertical dimension is latitude, which ranges from -89.75 to +89.75 in steps of 0.5 (the famous half degree by half degree global representation). The first data value (0,0) in the data set occurs at the bottom left. By default OpenModel will infer the row order using the values of the vertical dimension and reverse the row order as it accesses the data. You can override this default behavoir and set the row order manually if you wish, after all it is only a convention that puts north at the top of the page!



13.2 OpenModel syntax for grid data

13.2.1 netCDF Objects

The code segment below illustrates the syntax for accessing data within a netCDF object. **`GlobalTmp`** is the netCDF object. It contains global weather data on a 0.5x0.5 degree grid (it is one of the IPCC baseline climate data sets) hence has 360 rows and 720 columns. The temperature data is held within a field named in the netCDF object as '**`tmp`**' which is defined with 3-dimensions. The first represents the month of the year and the second and third are the x and y position in the spatial grid. The syntax to access this data is **`Grid.tmp(month-index, aRow, aColumn)`** as shown in the example below.

Note that the column and row references are zero indexed and as described above OpenModel maps the grid cell 0,0 to the top left corner unless the row order is reversed.

In this case the data are being read into a 2-dimensional array **`Temp()`** for further calculation within the model.

```
for month=1,11
  for i=0,359
    for j=0,719
      if Temp(i,j)>GlobalTmp.tmp.missing
        Temp(i,j)=Temp(i,j) + GlobalTmp.tmp(month,i,j) // add them up
      endif
    endfor
  endfor
endfor
```

Typically netCDF files define a missing value property for each field within the file. The use of this is shown in the above code segment via the syntax **`GlobalTmp.tmp.missing`**

The data can also via access via the co-ordinate values using the function **`GlobalTmp.tmp(aX,aY, month,i,j)`** where **`aX`** and **`aY`** are in the same units as the grid co-ordinates. The example model **`ChildandSpatialDemoByCoords.OMMLx`** illustrates the use of cordinate access to the grid data.

```
for month=1,11
  for i=0,359
    y = -89.75 + 0.5*i
    for j=0,719
      if Temp(i,j)>GlobalTmp.tmp.missing
        x = -179.75 + 0.5*j
        Temp(i,j)=Temp(i,j) + GlobalTmp.tmp.ByCoord(x,y,month) // add them up
      endif
    endfor
  endfor
endfor
```

The authors have only occasionally used netCDF files with OpenModel and our implementation is probably quite rudimentary, and maybe ill-informed! We would be very happy to enhance the implementation if time and motivation are available.

13.2.2 OMgrid Objects

In the example below a 3-D OMgrid object **Weather** is used to provide meteorological values of **Tmax**, **Tmin**, **SunHrs** and **Rain** for a range of years and months. The data is accessed using the **GridByCoord** function. The first two arguments for the function (**east** and **north**) are the coordinates at which the values are required; the remaining arguments denote the location in the data grid. The first argument is the index of the different weather variables (0 for **Tmax** and so on), whereas **YearIndex** and **MonthIndex** represent the year and month for the required data.

```
  Tmax   = Weather.GridByCoord(east,north,0,YearIndex,MonthIndex)
  Tmin   = Weather.GridByCoord(east,north,1,YearIndex,MonthIndex)
  Sunhrs = Weather.GridByCoord(east,north,2,YearIndex,MonthIndex)
  Rain   = Weather.GridByCoord(east,north,3,YearIndex,MonthIndex)
```

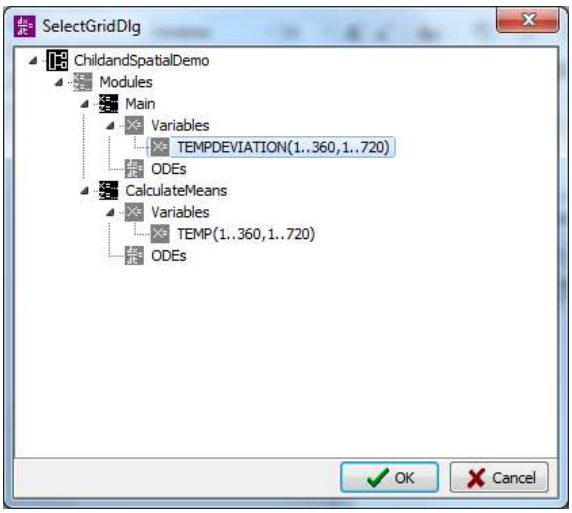
An alternative access method is to use the Grid function which accesses the data simply using the indices of the position of the values in the data grid.

```
  Tmax   = Weather.Grid(aCol,aRow,0,YearIndex,MonthIndex)
  Tmin   = Weather.Grid(aCol,aRow,1,YearIndex,MonthIndex)
  Sunhrs = Weather.Grid(aCol,aRow,2,YearIndex,MonthIndex)
  Rain   = Weather.Grid(aCol,aRow,3,YearIndex,MonthIndex)
```

Missing data with the data grid return the value **Weather.Missing** and the number of rows and columns can be accessed using **Weather.NCol** and **Weather.Nrow**

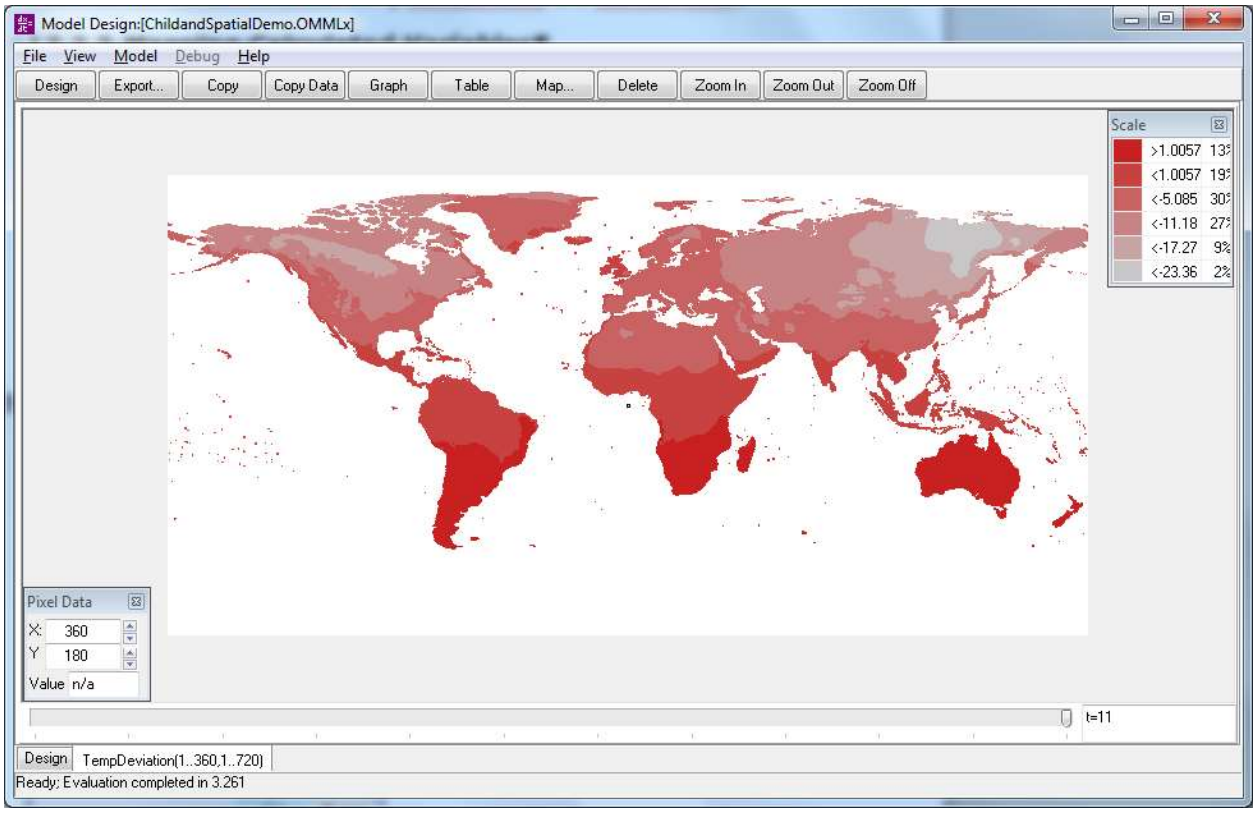
13.3 Mapping Calculated Variables

Any calculated 2-D array object can be displayed as a map by clicking the Map button on the main tool bar after the model has been evaluated. This will show a dialog box (as below) which displays any map-able (i.e. 2-D array) variables and ODEs within the model.



In this case the variable **`TempDeviations(1..360,1..720)`** has been displayed. The first dimension of the array (i.e. 1..360 in this case) is assumed for the purposes of mapping to be the rows. The second dimension (i.e. 1..720 in the example) is assumed to be the columns. The first element in the array, in the example **`TempDeviations(1,1)`**, is displayed in the top left corner.

As with scalar variables and ODEs in any model the calculated values are, by default, held in memory for each time step so they can be displayed (this can create enormous memory use for large grids or large number of time steps). The slider bar at the bottom of the map display allows you to map the different time steps as required. Initially the value of the array is shown for the last time point evaluated.



13.4 Creating OMgrid Objects

13.4.1 Importing Grid Data

OpenModel can import text files to create OMgrid files and link them to the model.

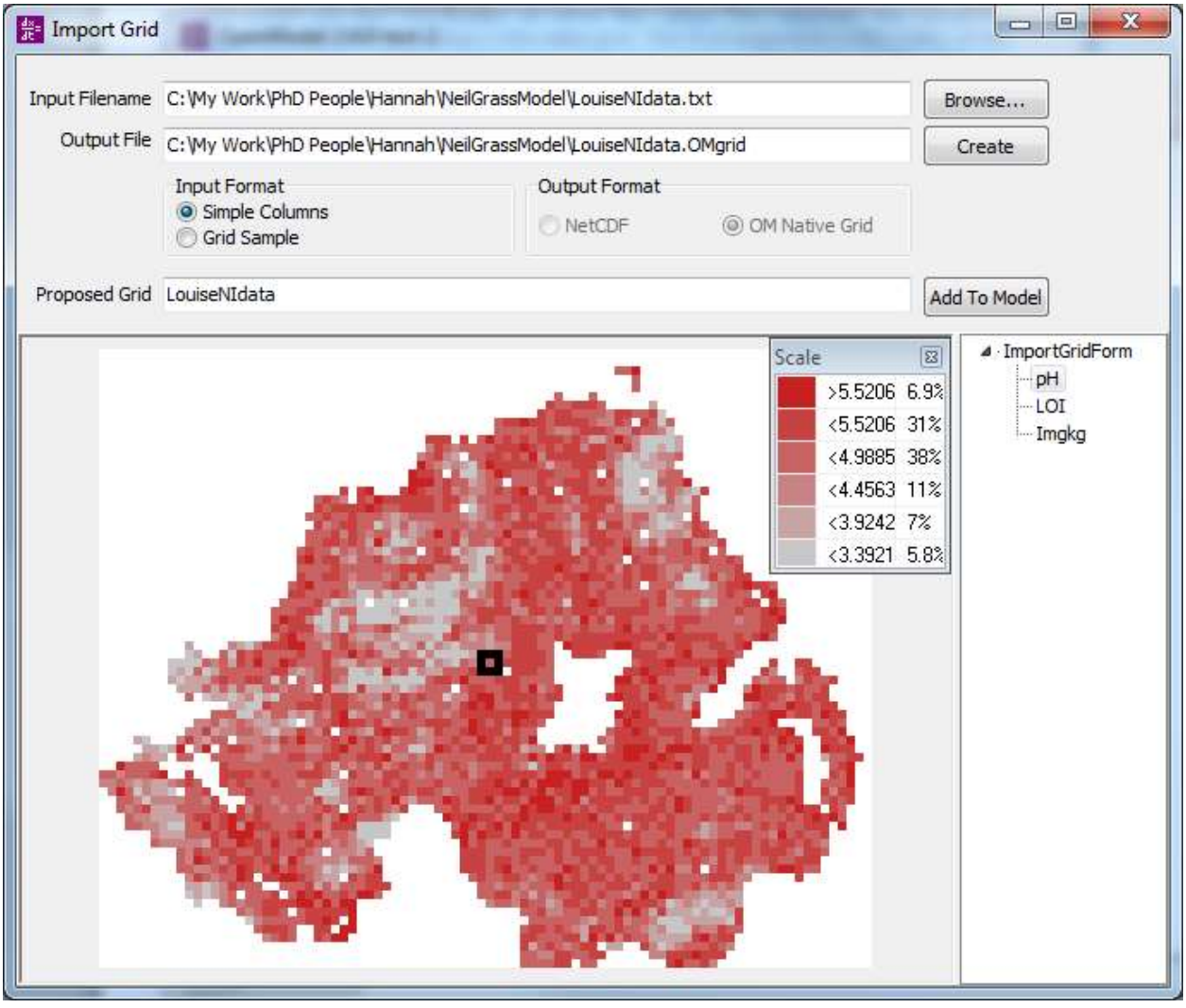
Selecting **`File|Import Grid`**… will display the dialog box shown below.

`Input Filename` specifies the text file to be imported. You may use the Browse button to locate this file (much easier). If you use the browse button the **`Output File`** field will be automatically filled with a proposed filename. This will have a file extension of .OMgrid. You can change the filename as appropriate but if you change the file extension OpenModel will not recognise it as an OMgrid object.

Two formats of input are allowed (a) Simple Columns (b) Grid Sample. These are described below).

The output file will be created when you click the Create button. If errors are encountered when processing the input file they will be reported with a line number for the position in the file where the error occurred (hopefully!). The most common error is an inconsistent number of data columns on each line arising from gaps in the data. Typically this is solved by defining a value as missing.

Once the file has been created its contents are displayed in the dialog as shown below. A proposed name for the grid object is indicated (you can change this). You can add it to the model structure by clicking **`Add To Model`**.



Simple Column Format

Data are provided with co-ordinates defined on a regular grid. The data value is stored at the grid location defined by the co-ordinates. If more than 1 data value exists in the file for a given grid location the final value read is used. Grid locations for which no data are provided are taken as 'missing'.

Files must be text tab delimited (readily created by Excel). They have several lines of required/optional fields to specify the data followed by the data columns.

The first few lines of a typical file are shown below

```
gridsize    50
missing     -999
data
east  north copper      zinc  cadmium     lead
1360  260   6.5   27    0.1   26
1360  310   24.3  35    0.4   46
1410  260   19.7  55    0.6   47
```

Compulsory header fields are:

- **Gridsize**. Specifies the height and width of each grid square in the units of the coordinates used.
- **Missing**. The value in the data set to denote missing data
- **Data**. This has no value, it simply denotes that the data is about to start

A header line for the columns of data themselves is compulsory. The first two columns are the x and y co-ordinates (a label must be provided for these 2 columns but it is not used). Any number of subsequent columns can be provided, each must have a header which is used to define the grid field in OpenModel. Field names must be alphanumeric (as with OpenModel variable names). The number of columns is determined by the number of column labels, confusion can arise if labels have spaces!

Optionally you can specifiy xLL and yLL, the co-ordinates of the lower left corner of the grid. If these are omitted the values will be set to the minimum X and Y values in the imported data.

Grid Sample Format

This format is useful if your data has arisen from samples collected across the area of interest but which are not defined on a regular grid. Data are provided with the coordinates at which they are collected. As the data is processed the appropriate grid square is calculated. If more than 1 data value is given for a grid square the imported values are averaged (arithmetic mean). Grid locations for which no data are provided are taken as 'missing'.

Files must be text tab delimited (readily created by Excel). They have several lines of required/optional fields to specify the data followed by the data columns.

The first few lines of a typical file are shown below

```
missing     -999
gridsize    2000
data
EASTING     NORTHING    pH    LOI   Imgkg
218177      358382      5.79  22.41 8.124724392
239518      347149      4.95  7.23  1.771359528
```

Compulsory header fields are:

- Gridsize. Specifies the height and width of each grid square in the units of the coordinates used.
- Missing. The value in the data set to denote missing data
- Data. This has no value, it simply denotes that the data is about to start.

A header line for the columns of data themselves is compulsory. The first two columns are the x and y co-ordinates (a label must be provided for these 2 columns but it is not used). Any number of subsequent columns can be provided, each must have a header which is used to define the grid field in OpenModel. Field names must be alphanumeric (as with OpenModel variable names). The number of columns is determined by the number of column labels, confusion can arise if labels have spaces!

Optionally you can specifiy xLL and yLL, the co-ordinates of the lower left corner of te grid. If these are omitted the values will be set to the minimum X and Y values in the imported data. This option is especially useful for this file format as the sample locations may not be on a regular grid and you may want to force the grid to a 'neat' xLL and yLL position.

Obviously this is a very rudimentary method for creating a spatial grid. Numeours specialist products exist for this type of analysis with 'proper' methods of averaging and interpolation. Outputs from these culd be readily imported into OpenModel using the column format described earlier.

13.4.2 OMgrid file specification

The OMgrid file format is straightforward and it is straightforward to create files directly if you have some programming skills.

OMgrid files comprise a series of values and strings which specify the layout of the data together with string specifiers to aid their interpretation when displayed. These specifiers are then followed by the data themselves. The specification values occur in the following order:

NFIELDS	Integer (4 byte signed)	Number of data fields
XLL	8 byte double precision real	horizontal co-ordinate of the lower left corner of the grid
YLL	8 byte double precision real	vertical co-ordinate of the lower left corner of the grid
DXY	8 byte double precision real	Size of the grid square sides in the units of the co-ordinate system
MISSING	8 byte double precision real	Code to denote missing values in the data (-999 is a popular choice)
LENGTHS()	NFIELD+2 array of integers	Lengths of each of the NFIELD+2 dimensions
NLABELS	Integer (4 byte signed)	Number of field labels (if zero then no field labels are given)
LABELS()	String Array (8 byte char) of length NLABELS Each string is stored as a 4 byte signed integer giving the number of characters, followed by that number of 8 byte ASCII characters	Used as a label for each

Data are stored as 4 byte (single precision) real values in a binary file. The data are represented using a number of dimensions (NFIELDS+2). As an example a grid with 3 type of weather data for different years and months of the tears would comprises a five dimensional grid of numbers, X[d1,d2,d3,d4,d5]. In this d4 and d5 represent the column and row position in the grid and d1, d2 and d3 the type, year and month specification. For example, the value located at X[0,1,3,100,50] would be the value of the first weather variable (Tmin), for the second year and the 4th month (Apr) for column 100 and row 50. Note that the arrays are zero based. The data are stored as a single block of binary values with the highest dimension varying fastest. The column and row position [0,0] is the top left of the grid.

The values of XLL, YLL and DXY are used if data on the grid is accessed by co-ordinate. If this is not required in a model, i.e. if data are only accessed by their column and row position then any value (e.g. 1.0) can be specified for these variables.

OMgrid files support a maximum of 3 data dimensions.

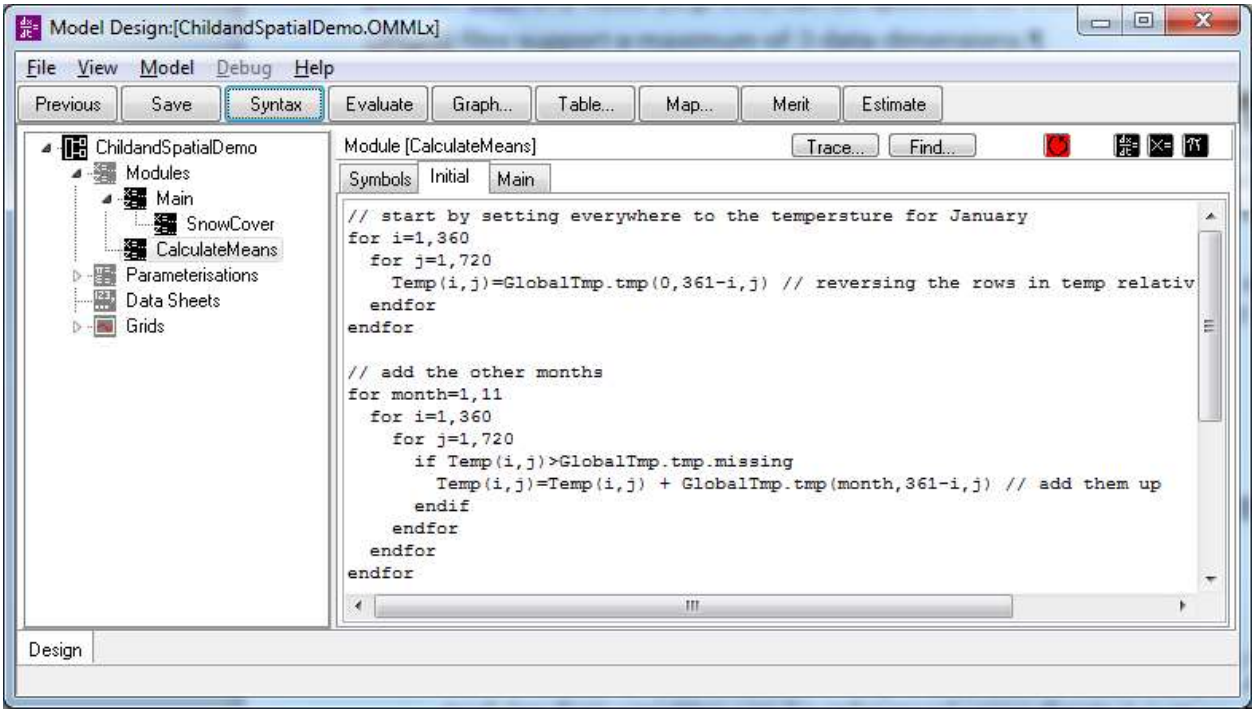
14.Using Sibling and Child Modules

If you are writing a more complex model it may be useful to create it as a series of inter-connected modules, passing control of the flow of calculations between them using **`CALL`** and **`EXIT`** statements. As explained in more detail below you can use sibling and child modules to help you manage more complex model scripts.

- Sibling Models are executed quite separately from one another although their variables may be referenced from another module using the syntax **`ModuleName.VariableName`**.
    - When a model is compiled for execution only those sibling modules which are either 'called' (via a **`call`** statement) or 'used' (via a **`use`** statement) are compiled. This is further discussed below (14.1.1)
- Child Modules are effectively embedded within their parent module, using them is more a matter of convenience, to help create clearer model scripts. As with sibling modules their variables can be referenced using the **`ModuleName.VariableName`** syntax.

Module names have global scope throughout the model. So if you have a module called **`BigSum`** you cannot also have a variable called **`BigSum`** in any module within the overall model.

Sibling and child modules are added by right clicking the parent module in the main left hand model tree. This displays a menu with options for adding child and sibling modules. In the example below (**`ChildAndSpatialDemo.OMMLx`**) the module **`Main`** has a sibling **`CalculateMeans`** and a child **`SnowCover`**



14.1.1 Sibling Modules: Call, Exit and Use Statements

If a module, **`MyModule`**, calls a sibling module, **`MySibling`** the sibling module will be completely executed before control passes back to the calling **`MyModule`**. i.e. The initialisation statements of **`MySibling`** will be executed after which the main statements

of the sibling module will be iterated over the independent variable of the sibling module.

The independent variable symbol, its start and stop values, and the solution method (euler, runge-kutta etc) (see page 63) are set independently in a sibling module.

The sibling module can return control to its calling module at any point through use of an **exit** statement. This returns control to the calling module without any further statements being executed.

When a model is compiled for execution the compilation procedure commences with the start module (obviously). All child modules are included, but sibling modules are only included if they are explicitly required. This applies if the sibling module is called (via a **call** statement) or 'used' (via a **use** statement).

As discussed above **call** transfers execution control between modules. A **use** statement does not transfer control, but forces the compilation of a module even if it is not explicitly called. This makes all the variables and parameters of the 'used' module available to those modules which are being executed. Why would you want to do this? Sometimes it is convenient to use a specific module to hold a group of variables or parameters. For example collecting all the output variables together in a separate module may make graphing them easier. Similarly, it may be convenient to group a particular set of related parameters. The **use** statement makes this possible without introducing the additional overhead of a **call** to the sibling module.

14.1.2 Child Modules: Call and Exit Statements

If a module, **MyModule**, has a child module, **MyChild** the initialisation statements of **MyChild** will be automatically executed after the execution of the **MyModule** initial statements. There is no need to 'call' the child module in this case.

However during the main iterative phase of the model evaluation the child module main statements are only executed when they are 'called' from another module, i.e.

```
call MyChild
```

will pass execution to the main statements of **MyChild** which will then be fully executed and control passed back once all the statements have been executed.

A called module can return control to its calling module at any point through use of an **exit** statement. This returns control to the calling module without any further statements being executed.

In the case of a child module the independent variable symbol, its start and stop values, and the solution method (euler, runge-kutta etc) (see page 63) are automatically set to be the same as the parent module.

14.2 Example Use of Sibling and Child Modules

The example model **ChildAndSpatialDemo.OMMLx**, although a little contrived, uses a Sibling and Child module to undertake several calculations based on air temperature using a global spatial grid. The model has three modules: **Main**, **SnowCover** (child) and **CalculateMeans** (sibling).

In its initial script the module **Main** calls the sibling **CalculateMeans**.

```
call CalculateMeans

for i=0,359 // mark any missing values
  for j=0,719
    if CalculateMeans.Temp(i,j)<=GlobalTmp.tmp.missing
      TempDeviation(i,j)=GlobalTmp.tmp.missing
    else
      TempDeviation(i,j)=0
    endif
  endfor
endfor
```

CalculateMeans performs a calculation of mean temperature for each location in the spatial grid, summing the monthly temperatures in the attached netCDF grid object and dividing them by 12. Having performed these calculations **CalculateMeans** returns to Main using an exit statement. This is included in the example for completeness, in practice control would be passed back at the end of the script as default behaviour.

The independent variable of **CalculateMeans** is not linked in any way to the independent variable of **Main**. What's more **CalculateMeans** is executed in its entirety when it is called (that is control passes through its initial to the iterative main script). As it happens this feature is not fully exploited in the example, **CalculateMeans** only has initial script statements.

```
// start by setting everywhere to the temperature for January
for i=0,359
  for j=0,719
    Temp(i,j)=GlobalTmp.tmp(0,359-i,j) // reversing the rows in temp relative to
GlobalTmp.tmp
  endfor
endfor

// add the other months
for month=1,11
  for i=0,359
    for j=0,719
      if Temp(i,j)>GlobalTmp.tmp.missing
        Temp(i,j)=Temp(i,j) + GlobalTmp.tmp(month,359-i,j) // add them up
      endif
    endfor
  endfor
endfor

// divide by 12 to get the means
for i=0,359
  for j=0,719
    if Temp(i,j)>GlobalTmp.tmp.missing
      Temp(i,j)=Temp(i,j)/12 // get the mean
    else
      Temp(i,j)=GlobalTmp.tmp.missing
    endif
  endfor
endfor

exit
```

When the child module **Snowcover** is called it is executed as if it were part of **Main**. Its initial statements are executed immediately after those of **Main** (as it happens **Snowcover** doesn't have any) and its main statements are executed when it is called from the main statements of **Main**.

15. Fixing Model Bugs

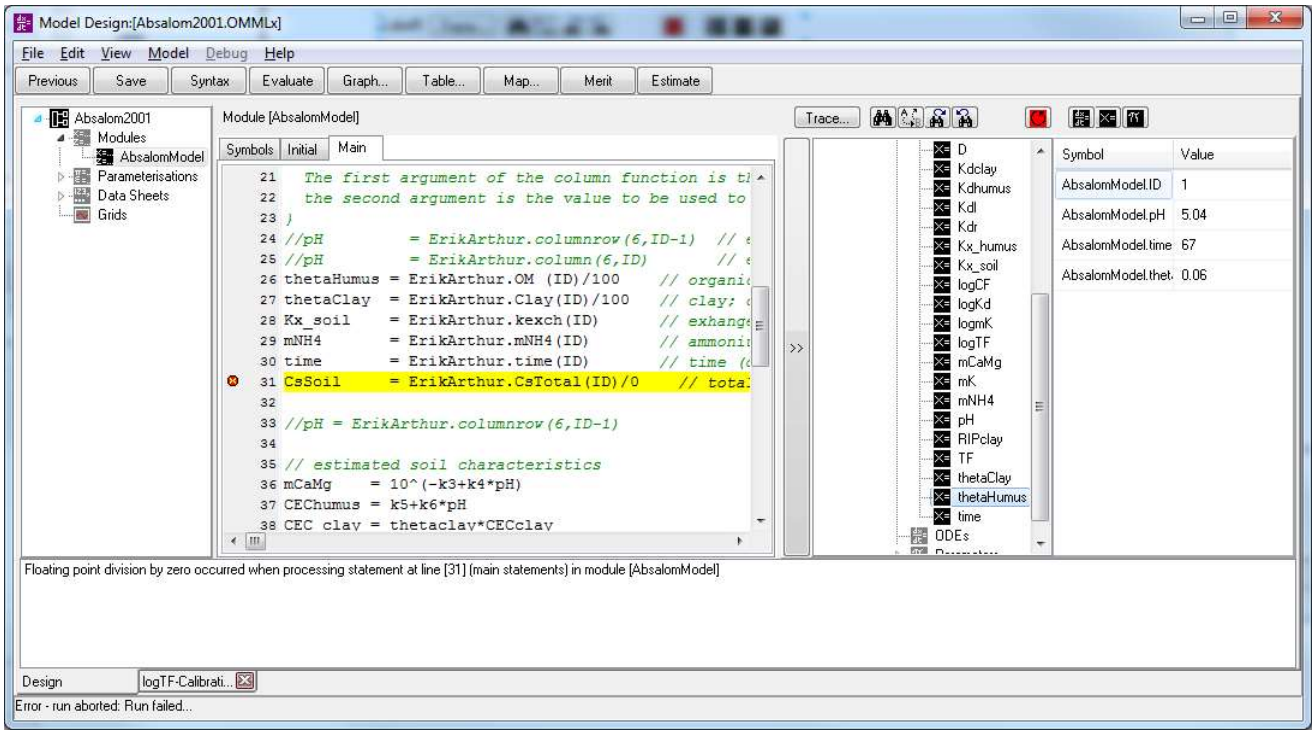
Regrettably you may make mistakes when writing a model script, with the result that the model may not behave as you expect. This may result from a calculation fault, i.e. the model execution stops and reports an error of some description. Or, it may be that the results are not plausible and you want to check the underlying calculations. OpenModel has some standard facilities for de-bugging models to help in this case.

15.1 Runtime Errors

If an error is encountered when the model runs, for example a division by zero occurs or the model attempts the calculate the log of a negative number the model evaluation will fail.

If you are using one of OpenModel's iterative procedures (for example monte carlo analysis, or model reduction) this will be noted as a model error and the procedure will try again. Overall execution will only stop if the (user defined) number of allowable errors for the procedure is exceeded. Exploring the model parameter's space, for example, may well result in infeasible parameter values being tried, and it would be very frustrating if the procedure simply gave up.

For one off model evaluation a runtime error will halt the execution of the model and the error will be reported. An inspection window will open on the right side of the script window. In the example below (**`Absalom2001.OMMLx`**) an error in the calculation of **`CsSoil`** has been manufactured by dividing by zero. The error (divide by zero) is reported in the message list as the bottom of the window and the value of several model variables have been inspected on the right. You can add variables to this list by double clicking their symbol in the model tree and remove a variable from the list by double clicking its row.



15.2 Debugging

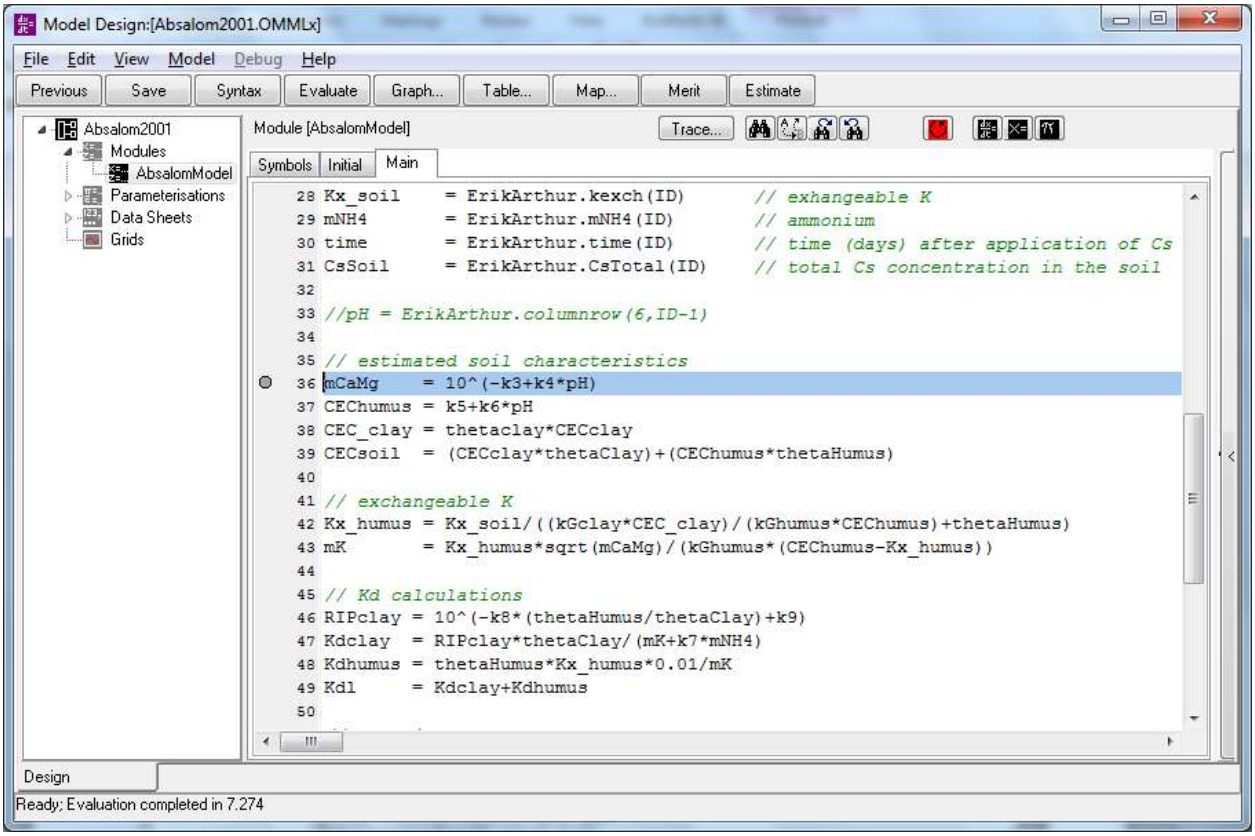
'Debugging' is when you run your model and inspect its behaviour, usually to fix a fault that you have inadvertently introduced. OpenModel includes some standard debugging facilities, these are:

Breakpoints: you can insert one or more break points into the model script. These cause the model execution to 'break' (i.e. pause) at the break point so you can inspect the value of model variables.

Watches: once model execution has been halted you can setup a 'watch' on any model variables you wish in order to monitor the progress of the model calculation.

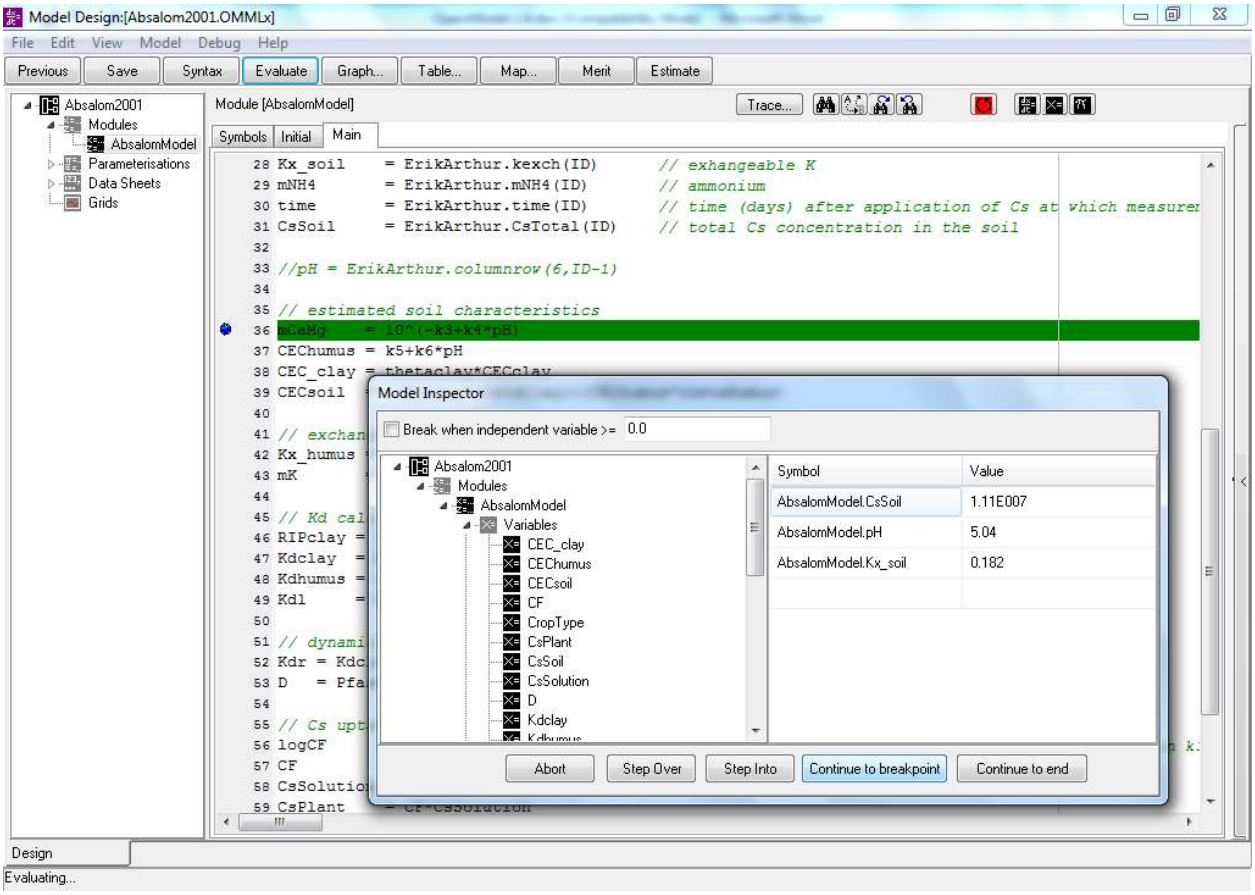
Step Over/Step Into: you can allow the calculation to proceed, stepping it line by line, monitoring any watches that are set up. If appropriate you can 'step into' or 'step over' any calls to other modules.

A breakpoint is inserted into the code by clicking to the left of line number of the required line (as in the example below).



When the model is evaluated execution stops at the moment before the line is executed, the line is highlighted and the Model Inspector dialog is displayed as shown below.

The current values of 'watched' variables are shown in the right hand view of the dialog box. You can add variables to this list by double clicking their symbol in the model tree shown on the left hand side of the dialog. In this example **`CsSoil`**, **`pH`**, and **`Kx_soil`** have been selected. To remove a variable from the list double click on its row.



The buttons on the bottom of the dialog box control the subsequent execution of the model.

Abort: this simply aborts the rest of the model run.

Step Over: this steps to the next line in the current module. If the current line is a call statement the call is executed and control passed back to the controlling module, stepping over the module call for the purposes of the debugging.

Step Into: this is equivalent to step over, except where the current line is a call statement the call is executed and the display steps into the called module.

Continue to Breakpoint: the model execution proceeds to the next breakpoint encountered. This may be a different position in the script, or the same position at a subsequent step in the calculation.

Continue to End: the model run proceeds, ignoring further break points until the normal end of the execution.

By default a breakpoint is triggered when it is first encountered. However it is quite common for a bug in a model script to be known to occur at a certain time point (independent variable) in the model execution. In this case it is more useful for the breakpoints to only occur after a certain time point (i.e. independent variable value) in the model run. To use this option check the '**`Break when independent variable>=`**' box at the top of the dialog box and enter the required value of the independent variable in the edit field.

16. OpenModel Syntax Reference

16.1 Initial and Main Statements

OpenModel modules have two sets of executable statements.

- Initial statements: these are executed once at the start of the model evaluation. The initialisation statements of any child modules are automatically executed after the execution of the parent modules initial statements. This can be used to setup any aspect of the module configuration but is especially useful for initialising the value of any ordinary differential equation symbols.
- Main Statements: these are executed iteratively to solve the model over the range of the models independent variable. The main statements of any child modules are only executed when they are 'called' from another module (see below: Call Statements).

16.2 Time Steps and Independent Variables

16.2.1 Operation of the Independent Variable

When a model is run the initial statements are processed with the independent variable set to its initial value. The main statements are then executed, also with the independent variable set to the initial value. The resulting values are recorded as the initial state of the model.

OpenModel then advances to the next time step, integrating any ODEs using the rates calculated at the initial independent variable. The main statements are then evaluated, setting any variables defined there. The resulting values are recorded as the state of the model at the end of the time step.

A potentially anomalous consequence of this method of operation is that at the start of the simulation variables can be set in both the initial and main statements and the recorded results are the net effect of the two sets of statements. For example, if a variable is defined in the initial statements as

```
counter = 0
```

and also defined in the main statements as

```
counter = counter +1
```

The recorded value for the initial step will be 1 (not zero). Its not a bug, its a feature!

16.2.2 Modules without Independent Variable

Modules can be defined as having no independent variable (simply uncheck the Use Independent Variable Checkbox in the Independent Variable definition part of the evaluation dialog box). When a model is evaluated in this case the initial statements are processed and then the main statements are executed. The resulting values are recorded and the model evaluation is complete.

16.2.3 Sibling and Child Modules: Independent Variable

The definition of the independent variable is handled differently in child and sibling modules. In the case of a top level sibling modules (i.e. modules with no parent) the independent variable symbol, its start and stop values, and the solution method (euler, runge-kutta etc) are set independently. In the case of a child module the independent variable symbol, its start and stop values, and the solution method (euler, runge-kutta etc) are automatically set to be the same as the parent module and cannot be set independently.

16.3 Arrays

Variable, ordinary differential equation and parameter symbols can all be defined as arrays or scalars. To define a symbol as an array simply name it using the array syntax, e.g.

```
x(1..10)
```

is an array with elements 1 to 10.

The indexing of array elements is quite flexible, so arrays can start at any positive integer value, for example **x(10..100)**.

The syntax is readily extended for multi-dimensional arrays, e.g. **x(1..10,1..2)**

Arrays can be referenced in statements specifically, e.g.

```
y = x(3)
```

assigns the value of **x(3)** to **y**.

Arrays can be also be referenced iteratively in loops using a loop iterator e.g.

```
y = x(i)
```

where i is a loop iterator controlled by a for loop (see Loops below)

16.4 Assignments

Symbols can be simply assigned using '=' notation, e.g.

```
x = 10
y = x*2 + z^2
```

16.5 Differential Equations

If **ode1** is a differential equation symbol it can be assigned values by 2 methods, firstly and specifically for differential equation symbols a statement of the form shown below can be used:

```
ode1.rate = x
```

This sets the rate of change of ode1 to the current value of **x**. This rate is then used when OpenModel iterates to solve the differential equations of the model.

Statements of the form below can also be used

```
ode1 = 10
```

This simply sets the value to 10 when the line is executed. This has the effect of overriding any differential rate of change that is defined for the symbol. This form is useful in the initial statements of the model for setting up the initial values of any differential equations. This form of assignment can also be used to control discrete, event type, of changes to differential equation symbols

16.6 Logic Statements

OpenModel has the standard if statement syntax to control model execution, for example

```
if y=1
      x = 10
      z = x*2 - 5
endif
```

In this piece of code the lines between the **if** and the **endif** statements are only executed if the variable y is equal to 1. The syntax can be extended to include an **else** statement, e.g.

```
if y=1
      x = 10
      z = x*2 - 5
else
      x = 12
      z = x*3 - 5
endif
```

In this case if **y** is equal to 1 then the first sub-block of code is executed, if this is not the case (i.e. all other cases) the second sub block of code is executed.

If statements can be nested any number of times, e.g.

```
if y=1
      if anothervariable >3
            x = 10
            z = x*2 – 5
      else
            x = 100
            z = 5
      endif
else
      x = 12
      z = x*3 - 5
endif
```

The **else** statement can be extended to an **elseif** statement to automatically create a cascade of ifs.

The available logical operators are tabulated vbelow.

equals	greater than	less than	greater than or equal	less than or equal	not equal
=	>	<	>=	<=	<>

16.6.1 Built In Procedures – Call Statements

Call statements can also be used to call built in procedures. These are compiled procedures which can be added to OpenModel by anyone interested in writing some Pascal code. For more details on writing built in procedure see section 17.2.

16.7 Loops

Sequences of statements can be performed iteratively using loops. OpenModel implements a single type of loop; the **for** loop. The iterates for a set number times using a control iterator (i,j,k are set as defaults in OpenModel). The end of the loop is denoted by an **endfor** statement. Any statements can be used inside a loop but they are especially usefully for working with arrays as in the example below.

```
for i=1,100
      x(i) = y(i)*z*a/h
      if x(i)>20
            breakloop
      endif
endfor
```

The **Breakloop** statement forces control to pass out of the loop and on to the statement after the appropriate **endfor** statement. This is a useful way to end a loop if certain conditions are met, e.g.

```
for i=1,100
      x(i) = y(i)*z*a/h
      if x(i)>20
            breakloop
      endif
endfor
```

16.8 ReadFile Statements

ReadFile statements provide a means to update parameter values using file values during a model evaluation. An example of when this might be useful is if site characteristics need to be changed at some point during the model evaluation.

The statement syntax is very simple, with the required file being given as an argument to the **ReadFile** statement. If no path is specified the file is assumed to be in the same folder as the model file.

```
ReadFile('filename.txt')
```

If the file cannot be found a runtime error is generated and execution of the model halts.

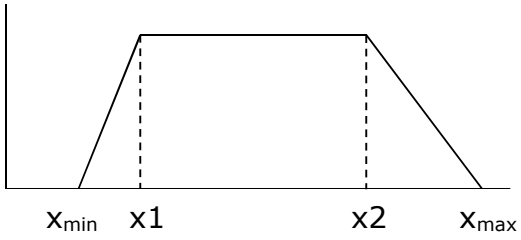
The files require a particular format. They must be text tab delimited files in the style illustrated below. The name of the parameter to be updated appears at the start of each line (including the module symbol) followed by the value to be set. As shown in the example below array parameters can be assigned, in which case the required values are given as a tab delimited list.

```
SoilParameters.Kq 0.3
SoilParameters.Ko 0.00007
SoilParameters.depth(1:2) 1 1.5
SoilParameters.SSAT(1:2) 44 44
SoilParameters.SDUL(1:2)  31.3 31.3
management.RUE 1.90 1.75
variety.ta(1:6)=11.68571 11.20714 9.857143 8.928571 8.614286 8.485712
```

17.Functions and Built-In Procedures

17.1 Functions

The following functions are defined in OpenModel

abs(x)	absolute value of x
log10(x)	log base 10 of x
ln(x)	natural log of x
logn(n,x);	log base n of x
exp(x)	exponential of x
sin(x)	sine of x
cos(x)	cosine of x
tan(x)	tangent of x
arcsin(x)	arc-sine of x
arccos(x)	arc-cosine of x
arctan(x)	arc-tangent of x
sinh(x)	sinh of x
cosh(x)	cosh of x
tanh(x)	tanh of x
arcsinh(x)	arc-sinh of x
arccosh(x)	arc-cosh of x
arctanh(x)	arc tanh of x
trunc(x)	truncates x to return the integer part; trunc(3.2)=3
ceil(x)	returns smallest integer value greater than x; ceil(3.2)=4
randn(mean,sd)	returns a normally distributed random number of mean and sd
rand	returns a uniformly distributed random number between 0 and 1
randbetween(x1,x2)	returns a random number between x1 and x2
ratelimit4(x,xmin,x1,x2,xmax)	Returns a 0 to 1 value based on this function 
min(x1,x2)	returns the min of x1 and x2
max(x1,x2)	returns the max of x1 and x2
frac(x)	returns the fractional part of x; frac(3.2)=0.2

mod(x1,x2)	mod(x1,x2+ = trunc(x1,xy)
round(x)	rounds x to the nearest integer
evansparslowF(x1,x2)	Specialist function; Evans and Parslow function used in light intensity calculations in some marine ecosystem models (Evans & Parslow 1985) $result = a - x2 * \ln\left(\frac{x2+a}{x1}\right)$ where $a = \sqrt{x1^2 + x2^2}$
sgn(x);	-1 if x<0; 0 if x=0; +1 if x>0

It is quite easy to add new functions to OpenModel although it does require some small changes to the OpenModel code.

17.2 Built-In Procedures

The following procedures are defined in the distributed version of OpenModel. If you are willing to write some pascal code you can add to these as described under **Adding Built-In Procedures**.

17.2.1 One-Dimensional Diffusion: diffusion1D

diffusion1D(C,Rate,D,dx,index1,index2,BC1mode,BC1, BC2mode,BC2,Gain)		Solves the diffusion equation in 1-dimension
	C	array representing the concentration of the material that is diffusing
	Rate	array (same size as C) of rates of change of C at each node (e.g. source terms). Elements set to zero if there are no source terms
	D	array (same size as C) representing the values f the diffusion coefficient at each node
	dx	Distance between each node (scalar)
	Index1	Index in the array C, Rate and D used as the first node in the diffusion solution. Allows a reduced part of the overall array to be under. Used as a zero based offset.
	Index2	Index in the array C, Rate and D used as the last node in the diffusion solution. Allows a reduced part of the overall array to be under. Used as a zero based offset.
	BC1mode	Specifies the type of boundary condition to be used. **`BC1mode=0`** then BC1 is the fixed concentration of C(min)

		BC1mode=1 then BC1 is the fixed flux incident on C(min) from outside the modelled region
	BC1	Value of the boundary condition for the i=C(min) node
	BC2mode	Specifies the type of boundary condition to be used. **BC1mode=0** then BC1 is the fixed concentration of C(min) BC1mode=1 then BC1 is the fixed flux incident on C(min) from outside the modelled region
	BC2	Value of the boundary condition for the i=C(max) node
	Gain	Overall gain of the system through the boundaries during the step; not including the effect of rate. Can use this to give the loss through an open boundary (if the other boundary is closed!)
Note: this procedure should only be used with euler fixed step integration as it is not consistent with the different step lengths used in the overall step of a procedure such as runge-kutta		

17.3 Adding Built-In Procedures

The built in procedure **diffusion1D** (see page 68) is an example of a built in procedure. Users can add further built in procedure as they require, although this does require some editing of pascal code in **MCfunctions.pas**! If you wish to add a build in procedure you are advised to first study the **diffusion1D** example.

Procedures in MCfunctions.pas that are to be used as built-in procedures are declared to OpenModel in LoadCallProcedures which can be found in MCfunctions.pas.

This simply returns a stringlist containing the specification of any procedures that you want OM to treat as a built-in procedure. OpenModel calls LoadCallProcedure at the start of the module compilation process and adds the defined built in procedures to its list of allowed calls.

The syntax used in the definition statement specifies the types of the arguments passed to the built in procedure. These can be array or float (corresponding to the double type in pascal). In the example below the additional argument_1 in parenthesis denotes that the length of the second array is to be the same length as for the first argument.

```
procedure LoadCallProcedures(sl:TStringList); // edit this procedure to add new call
procedures for OM
begin
  sl.clear;
  sl.add('DIFFUSION1D;array,array(argument_1),array(argument_1),float,float,float,
  float,float,float,float,float');
end;
```

The pascal code for the new procedure is also added to MCfunctions.pas and for the declaration of the procedure needs to be consistent with the specification given in LoadCallProcedures. Diffusion1D is provided as an example.

The declaration of Diffusion1D is shown below.

```
procedure Diffusion1D(var y:array of double; rate, d:array of double; dx, dindex1, dindex2,
BC1mode,BC1,BC2mode,BC2:double; var gain:double);
```

The use of diffusion1D is demonstrated in the model DiffusionTest.OMMLx which is included in the OpenModel installation folder. A code excerpt is shown below.

```
BC1 = 0
BC1mode = 1
BC2 = 1
BC2mode = 1

Call Diffusion1D(C,D,dx,BC1,BC1mode,BC2,BC2mode)  // do the diffusion
```

The statements **BC1=0** etc setp up the boundary conditions. The array **C** is the concentration profile and the array **D** is the spatial distribution of the diffusion coefficient. **C** can be of any (practical) length but OpenModel will insist that **D** is of the same dimension as **C** due to the definition of **Diffusion1D** in **LoadCallProcedures**.

If you let us have any useful (and working) built in procedures you develop and they will be added to the distributed version of OpenModel for others to use with appropriate credits.

Useful Global Definitions

OM defines the follow variables globally to enable built-in procedures to interface with OM.

- **OMglobal_deltaT**: current value of the time step being solved in OM (will vary if you are using automatic step lengths)
- **OMglobal_T**: current value of the independent variable (even if it doesn't have the symbol 'T' in the OM model!)

The example procedure **Diffusion1D** in **MCfunctions.pas** provides an example of the use of these variables.

18. References

Absalom JP, Young SD, Crout NMJ, Sanchez AL, Wright SM, Smolders E, Nisbet A, Gillett AG. (2001) Predicting the transfer of radiocaesium from organic soils to plants using soil characteristics. Journal of Environmental Radioactivity 52:31-43. More information at www.nottingham.ac.uk/environmental-modelling

Burke E, Bykov Y, Newall J, Petrovic, S (2004) A time-predefined local search approach to exam timetabling problems. IIE Transactions 36:509-528

Cox GM, Gibbons JM, Wood ATA, Craigon J, Ramsden SJ, Crout NMJ (2006). Towards the systematic simplification of mechanistic models. Ecological Modelling 198:240-246.

Crout NMJ, Tarsitano D, Wood AT (2009). Is my model too complex? Evaluating model formulation using model reduction. Environmental Modelling & Software, 24:1-7.

Crout NMJ, Tye AM, Zhang H, McGrath SP, Young SD (2006). Kinetics of metal fixation in soils: measurement and modelling by isotopic dilution. Environmental Toxicology and Chemistry 25:659-663.

Evans GT, Parslow JS (1985). A model of annual plankton cycles. Biol. Oceanogr. 3:327-347.

Haario H, Saksman E, Tamminen J (2001). An adaptive metropolis algoroithm. Bernouilli, 7:223-242