The essential BYOM Manual

Tjalling Jager*

April 3, 2015

About this document

The Build Your Own Model (BYOM) files respresents a flexible platform to develop, simulate and fit models expressed as ordinary differential equations (ODEs). Make sure you are able to work with Matlab at a basic level. If your mastery of ODEs is rusty, you might want to read the refresher and make some exercises (see http://www.debtox.info/dynmodtox.php). Regularly check the web page http://www.debtox.info/byom.php to see if there are updates to the BYOM files or its packages.

The BYOM platform will take most of the difficult things away from you, so you can focus on modelling issues instead of programming issues. The script files in the examples directory demonstrate the use of BYOM, and can form the basis for your own experimentation. Do not modify the files in the examples directory too much; make a copy of these files into a new directory when you move to a new topic.

This manual starts with the essential things you need to know to get started (Section 1). The remainder of this document contains the more detailed background information, and guidance for how to bend BYOM to your will! If you are missing a certain functionality in BYOM, please let me know, and I will consider adding it in a future update.

I would like to thank the participants of the TKTD summerschool (http://www.debtox.info/dynmodtox.php) for using and commenting on BYOM, which has been a major driver to improve its functionality.

*DEBtox Research, De Bilt, The Netherlands. Email: tjalling@debtox.info, http://www.debtox.info/

Contents

1 Essentials to get started

1.1 Installation

To install BYOM, unpack the ZIP file to a location of your choice. Do *not* rename any of the directories. The first files to study are in the `examples` directory. It is best not to modify these files, so you can always return to a working model. To create your own model, copy the files in `examples` to a new directory, somewhere as sub- or sub-sub-directory of the `BYOM` directory (do *not* rename the `BYOM` directory, put all new directories *below* this one, and don't start any of the directory names with '`BYOM`'). Make sure that the new directory includes a copy of the functions `derivatives.m` (which holds your model equations), `call_deri.m` (calls the derivatives functions with initial state values and includes the events function), and `pathdefine.m` (makes sure the directory `../BYOM/engine` can be found by Matlab). Modify `derivatives.m` to represent your own models. In several cases, (slight) modifications of `call_deri.m` will be needed. I suggest naming your script files starting with `byom_` to clearly distinguish them from the functions. Do *not* include spaces in script or directory names (Matlab does not like that). The engine directory contains the files for fitting the model to data; in general, there should be no need to modify these files.

1.2 Example scripts

I included several scripts to demonstrate the use of BYOM. The example file `byom_bioconc_extra.m` script describes a lot of options in commented-out sections, whereas a cleaner version is provided in `byom_bioconc_start.m`. The use of the simulator is demonstrated in the file `byom_bioconc_sim.m` (but check out the dedicated package `SIMbyom` for that).

The supplied example is the case of simple first-order kinetics for the internal concentration C_i in an organism:

$$\frac{d}{dt}C_i(t) = k_e(P_{iw}C_w - C_i) \quad \text{with } C_i(0) = 0 \text{ mg/kg}_{\text{wwt}} \tag{1}$$

The other symbols in this equation are for the elimination rate (k_e), the bioconcentration factor (P_{iw}) and the external concentration (C_w). The water concentration is not constant. There is first-order disappearance (with rate constant k_d) until the external concentration reaches a lower limit C_t:

$$\frac{d}{dt}C_w(t) = \begin{cases} -k_d C_w & \text{if } C_w > C_t \\ 0 & \text{otherwise, with } C_w(0) = C_{w0} \text{ mg/L} \end{cases} \tag{2}$$

This is a rather weird scenario, but is intended to demonstrate switching in the model, and how this can be dealt with in the ODE solver. In general, ODE solvers do not like hard switches, because one or more of the derivatives does not exist in a certain point. The function for C_w is discontinuous because for $C_w > C_t$ it is some negative value, but at C_t it immediately becomes zero. In response, most solvers will decrease the step size, leading to slow calculations (which are still imprecise). In general, this does not cause huge problems, so initially, don't worry about switches.

The best solution to switches is to *tell* the solver that there is a switch. In Matlab this works by defining an `events` function. This option is not well explained in the Matlab `doc` files, so I prepared an example in the BYOM files. Study `call_deri.m` to see how the events function can be used to catch the discontinuity (in this function locate the variable `eventson` to turn events capturing on or off; by default it is set

to off). An added benefit of the events function is that it allows you to calculate exactly when the switch takes place (returned in the script with the variable `tt`).

In most cases, you will want to start with simulations before you fit the model to some data. A quick switch in the script allows you do to do just that. Locate the variable `fit` below the parameter definition in the script file. Set to zero, this switch bypasses the fitting to produce a simulation using the provided parameter values only.

Note that the BYOM code always keeps a log file (`results.out`), where it collects the final output to screen. This is handy to look back to the results of an earlier run (Matlab has only a limited history in the command window).

To get started: run the script `byom_bioconc_start.m`, but first set `fit = 0;` (around Line 100) to run a simulation. Interpret the plots made in relation to the equations given above. Note that the symbols relate to the model lines with the same colour. The parameter values used are defined around Line 91-94. Walk thorough the script file to get an idea of what does what. Open the file `derivatives.m` that contains the model equations, and do the same. Next, set `fit = 1;` and run the script again to fit the parameters (the ones with a 1 as the second element of their definition). Try to make some confidence intervals by un-commenting parts of the code at the bottom of the script.

1.3 Matlab tricks used

cell array for the data set To hold all data, I use a single variable `DATA`. Having a single variable is nice, so it can be made 'global' (because it is needed in several of the functions), and we have the flexibility to have more than one data set without a lot of re-programming. The use of a 'cell array' allows multiple data matrices with different sizes; the number of the data set is given between curly braces.

Data sets are linked to state variables. This implies that every state should have a data set (a zero is filled in automatically when no data are entered in the script). It is possible to have more than one data set per state variable (starting with version 2.0, see comment in `byom_bioconc_extra.m`). If you have data for one state variable, but different measures, some more trickery might be needed. For example, it you have body size as weight and length, you can make two states for the body size, which allows to insert two data sets. The order of the data sets is the same as the order of the states in the initial state matrix `X0mat`, and in `derivatives`.

Parameter structure All parameter values are part of a 'structure'. In this simple case, you can view a structure as a vector of parameter values where you address the individual elements by a *name* rather than a *position*. This reduces the number of errors when re-programming existing files. After running the example script, type `par` in the command window to see what this structure looks like.

Using functions The scripts `byom_...` call another script called `calc_and_plot`, which calls several functions like `call_deri` (which again calls a function). Functions are very much like scripts (they are also text files with the extension `.m`), with one big difference: they run in their own protected memory. Variables that you define in a script are also available in the command window; but they are *not* available in a function. Thus, you have to tell the function which values it must use, and which it should return to the script/workspace calling it. A function thus has an input and an output (check out the first line in the function `call_deri`).

Another way to pass information into a function is to use 'global' parameters; defining a parameter as global means that you can use it anywhere (as long as you define it as global in each file; see global definition of `X0mat` in the script as well as `start_calc` in the engine). Be careful with the use of globals, and preferably only use them for things that remain constant during the running of the code.

Use the debugger When there is an error in a function, you cannot ask for variables in the command window (as the function runs into its own part of the memory). For this reason, Matlab includes a 'debugger'. Turn on the debugger by selecting 'debug' from the menu and selecting 'stop if errors/warnings'. E.g., select 'always stop if error'. Now, if there is an error message in a function, Matlab will stop there, and you can try to figure out what went wrong. Make sure to quit the debugger before you re-run the script (e.g., type `dbquit` in the command window). If something goes wrong without producing a hard error, you can force the debugger to stop by typing `error` into your function.[1]

1.4 Tips to work with BYOM

The engine The engine directory contains all the files needed to fit a model to data. In general, it will not be necessary to modify these files, but feel free to see how they operate. The engine also contains functions to calculate asymptotic standard errors (ASE's) and profile likelihoods. The calculation of ASE's is not very robust (and I probably also did not programme them in the most efficient manner); the profiles are much better, but also more time consuming. The end of each `byom_` script is dedicated to the calculation of confidence intervals.

Running scenario's The variable `X0mat` holds the initial states of the system (in a column) for every scenario (the rows). The first row contains the 'name' of the scenario. You can choose any number for that 'name'; it will be passed on to your `derivatives` function as parameter `c` (as you might want to use it for a constant exposure concentration). Note that in `derivatives`, variables `c` and `t` come in as single values, and not as vectors.

Making plots Plotting the results is done in the script file `calc_and_plot` in the `engine` directory. Here, plotting is done in a clever way, which should not require modification when you change your model. However, feel free to replace this section with your own plotting codes (see e.g., how calculation and plotting is done without `calc_and_plot` in the package `DEBkiss1`).

Simulating first, fitting later In most cases, it is a good strategy to start with simulations to test whether the model works as expected, get the model parameters in the neighbourhood of the data, and only then fit the model to the data. In the `byom_bioconc_...` scripts, there is a switch for simulations, just below the parameter definition (use `fit=0` to simulate). For more advanced simulation and plotting of dynamic systems, take a look at the `SIMbyom` package.

The data set Data are entered in matrix form, with time steps in the first column, and the 'scenarios' in the first row. Scenarios can hold different treatments, e.g., different exposure concentrations, and are the same as used in matrix `X0mat`.

[1] It is possible to set breakpoints, but these are removed again when using a 'clear all'.

The scenario/concentration value in the first row is handed over to `derivatives` as the variable `c` (note that scenarios are run one at a time, so `c` is always a single number, not a vector). The first entry in the matrix `(1,1)` is used to specify which likelihood function you want to use. Note that the -1 (for the multinomial likelihood) is used for survival data (or other quantal responses) only.

Using part of the data If you do not want to fit certain scenarios (e.g., certain concentrations), you can temporarily remove those scenarios from your data sets. For example, if you have scenarios 1, 2 and 3 in your data set, you can remove scenario 2 from data set 1 by adding in your script: `DATA{1}(:,3)=[];`. This removes an entire column from the data set. Note that the first column is for the observation time, so removing column 3 removes scenario 2. Alternatively, you can remove columns from `X0mat` in the same fashion.

Extended parameter structure Instead of a single value for each parameter, the fitting scripts specify a 5-vector. The first element is the parameter's starting value. The second is a switch to fit the parameter (1) or to fix it to the starting value (0). The next two elements specify the minimum-maximum range for the parameter, to avoid unrealistic values (or numerical problems). Make sure that each parameter has a starting value that is within the allowed range. The last element indicates whether the parameter will be fitted on normal scale (1) or on ^{10}log-scale (0). If this element is omitted from the parameter vector, a 1 will be used as default. Fitting on log-scale is more efficient when the parameter range to search is very wide (e.g., an elimination rate that may range from 10^{-3} to 10).

Log files Note that the BYOM code always keeps a log file (`results.out`), where it collects the final output to screen. This is handy to look back to the results of an earlier run (Matlab has only a limited history in the command window). Apart from the standard log file, the profile likelihood also makes a log file if it finds a better optimum (`profiles_newopt.out`). This means you can break off a run (which can be very time consuming) and still access the parameters for the better optimum.

Fitting explicit equations You can use the same handy BYOM platform also for fitting equations that are not specified by ODEs but by explicit functions (e.g., a polynomial relationship). Advantage is that you can than use profile likelihoods, fixing or fitting several parameters, use priors, etc. In the function `call_deri`, you can specify that you want to use an explicit function (or set of functions) by setting `useode=0`. The model than goes into `simplefun` (and `derivatives` is not used at all).

1.5 Modify the deluxe code to suit your needs

To get started with BYOM, I suggest to start with the `..._start` script. To modify this script to suit your modelling needs, proceed along these lines:

1. Decide on the state variables in your model. How many are there? Write down the equations on a piece of paper (and check them) before starting to code.
2. In the script file, make sure that there is a data set specified for each state variable. If you have no data for one of the states, simply specify `DATA{1}=0` (this is automatically done if you forget it). A matrix with weights may be added, but this is not required.

3. Modify `X0mat` to the scenarios that you want to calculate. The numbers in the first row are the 'names' of the scenarios you want to run (you could use them to specify the concentration of a chemical in each treatment). These are the same numbers as used in the first row of your data sets. You can specify more or less scenarios that you have in the data set, no problem. After the row with the names follow rows with the initial values for each of the states (generally, the value at `t=0`).
4. The section 'initial values for model parameters' in the script specifies the model parameters. Modify and add parameters as much as you like. Give them a name that is logical to you. For each parameter, you need to specify an initial value, whether you want to fit it or not, and a minimum and maximum allowed value.
5. Ignore all the 'optional' parts for now.
6. Specify a time vector for the model calculations (modify the `t=linspace(0,50,100)`). This only affects the plotted model curves, and not the fitting. If you want to fit only part of the data set, modify the data set itself, or `X0mat`.
7. Specify whether you want to fit or only simulate (`fit`), and whether you would like to see model curves for all scenarios or only for the ones where there are actually data points (`sho`).
8. Specify what you would like to see on the axes and legends of the plots (a general text will be inserted if you forget this).
9. At the bottom of the script, confidence intervals can be made, but leave this commented for now (profiles are recommended for intervals).
10. Next, turn to the file `derivatives`. This file consists of three sections: unpacking and renaming the state vector `X` into more useful variables, unpacking and renaming the parameter structure `par` into more useful variables, and the model equations themselves. Enter the ODEs directly in there. At the end, the various derivatives are combined into a derivatives vector `dX`. Note: if you specified three data sets in your script, you need to specify three derivatives. The parameters that you specified in the script are used here too, so make sure they match. Also note that the 'names' for each treatment enter in this function as the variable `c`.
11. Finally turn to the file `call_deri`, which calls the `derivatives` function. There is not much to change here, mainly the handling of events. If you are unsure how to specify an events function, just turn it off: `eventson = 0`. If you want to fit the initial value of a state, you need to define a parameter for that, and modify it so it is included in the initial state vector `X0`. A commented section clarifies how that works.
12. Run the file and enjoy! If you want to explore more options, check out the `..._extra` script.

2 BYOM in depth: the data set

2.1 Entering data

Suppose we have data for state variable 1, for two treatments (that we will call `1` and `2`), at three time points. The data are entered in array form as follows:

```
DATA{1} = [1    1     2
           0  0.2     0
           1  0.6   2.2
           2  1.1   4.6];
```

The data are for a continuous variable, and the observations will not be transformed (the `1` in the first position). If there are missing data (places in the matrix where there are no observations), a `NaN` (not a number) can be inserted.

Here, I used a `1` and `2` to specify the two treatments. I can use any number that I want there, as long as they correspond to the numbers in the first row of `XOmat`. For example:

```
XOmat = [1 2
         0 0]; % initial values for state variable 1
```

This signifies that I would like to simulate or fit both treatments, and that the state variable starts at zero in both treatments. I could also perform a different analysis, e.g.:

```
XOmat = [2 3
         0 0]; % initial values for state variable 1
```

Treatment `1` is now ignored (not used in fitting and not shown in plotting), but treatment `3` is now included. This treatment has no data, so it is only simulated, and I need to ensure that `derivatives` knows what to do with this treatment. Furthermore, the switch after the parameters can be used to modify the plotting behaviour: `sho=1` will plot treatment `3`, but `sho=0` will ignore it, as there are no data for this treatment.

Note that it is not necessary to start the time vector at $t = 0$. If you want to start at a different time point, e.g. $t = 50$, use:

```
glo.Tinit = 50;
```

The values in `XOmat` now represent the initial values at $t = 50$. Plotting will also now start from this value, irrespective of how you define the time vector for plotting t.

The numbers that specify the treatment will enter the `derivatives` as the variable `c`. In `derivatives`, we have to specify what to do with `c`. For example, we can switch as follows:

```
switch c
     case 1
     % put code here to specify what happens when c==1
     case 2
     % put code here to specify what happens when c==2
end
```

Or take a certain value from a vector:

```
F = [12 28 36]; % vector of f values for each treatment
f = F(c); % for parameter f, take the c-th element from vector F
```

Or use the variable `c` itself in the ODEs:

```
dX = c * X; % differential equation for X
```

This last option was what I had in mind, and why it is called `c`: for applications in ecotoxicology, we can take the exposure concentrations as the 'indicators' for the treatments. This is also how it was used in the example scripts `byom_bioconc...`.

2.2 Using weight factors

For each data set, we can add a matrix with weight factors of the same size as the observations. It does not matter if we enter the weights matrix as:

```
W{1} = [10    10
         9     8
         6     7];
```

or as:

```
W{1} = [1   1     2
        0  10    10
        1   9     8
        2   6     7];
```

In the latter case, the first row and first column are simply removed (in the engine script `prelim_checks`), so make sure that the order of the treatments is the same in `W` as in the corresponding `DATA`.

Generally, you can use the weights matrix to give certain observations more weight than others. For example, if the observations are means, the weights can specify the number of replicates underlying each mean. Statistically, it is usually better to enter all replicates as the data set. If you want to plot only the means, set the switch after the parameter definition: `repls=0`.

For survival data, the weights matrix has a different use (for survival data, you always have to enter the number of animals alive at each observation point, so additional weighing makes little sense). As in the DEBtoxM packages, the weights matrix can be used to specify animals that were removed alive during the test (e.g., for body-residue analysis) or animals that escaped from their container. Provide the number of missing/removed animals at the time point where they were last seen alive (and use zeros elsewhere). This way, all of the information that is available is included into the fit. Note that for plotting, the survival probability is used, also for the data points, so when there are missing data, the data points are corrected to allow for a straightforward comparison to the model prediction.

2.3 Combining data sets

Suppose that I have two data sets for the same state variable, at a different treatment (`1` and `2`):

```
a = [1   1
     1 0.2
     3 0.6
     5 1.1];

b = [1   2
     0   0
     2 2.2
     4 4.6];
```

Each data set has three observations, with continuous observations that will not be transformed (the `1` in the first position). However, the timing of the observations (first column in the data set) differs between the two sets. There are two simple ways in which we can now include the dataset in BYOM. The first is to invoke a small utility in the engine, which is called `mat_combine`:

```
DATA{1} = mat_combine(0,a,b);
```

If we now want to see what has happened to the data set for state 1, we can ask Matlab:

```
DATA{1}

ans =

            1            1            2
            0          NaN            0
            1          0.2          NaN
            2          NaN          2.2
            3          0.6          NaN
            4          NaN          4.6
            5          1.1          NaN
```

The two datasets are combined into a single one, with a single time vector. The missing observations are filled with a `NaN` (not a number) and will be ignored in the fitting and plotting. This utility can also be used to estimate missing data (using a cubic spline):

```
DATA{1} = mat_combine(1,a,b);
```

Which leads to:

```
DATA{1}

ans =

            1            1            2
            0     0.039583            0
            1          0.2       1.0755
            2      0.38819          2.2
            3          0.6       3.3745
            4      0.83681          4.6
            5          1.1       5.8734
```

Alternatively, we can define two separate data sets as follows:

```
DATA{1,1} = a;
DATA{2,1} = b;
```

This option will produce the same fits as combining the data sets with `NaN`s. However, using `mat_combine`, both data sets will be plotted in the same figure window whereas defining two data sets will produce two plot windows (one for each data set).

2.4 Weighing data for each data set differently

In some cases, it may be necessary to weigh complete data sets differently. For example, I have two state variables, and I want to make sure that the model fits state 1 as closely as possible, at the expense of the fit to state 2. Below the parameter definition, set:

```
glo.wts = [10 1]; % extra weight factor for each data set
```

This gives the data for state 1 twice the weight as that of state 2. If we included two data sets for one state, we need to specify the weights as:

```
glo.wts = [10;1]; % extra weight factor for each data set
```

In that case, the first data set for state 1 gets 10 times the weight of the second data set for state 1. If you use this option, make sure that the matrix entered in `glo.wts` has exactly the same size as the data set (try: `size(DATA)` at the Matlab prompt).

2.5 Providing residual variance for each data set

By default, the residual variance (that specifies the probability distribution for the distance between model and data) is derived from the data themselves, automatically (see the technical document on http://www.debtox.info/book.php on treating the s.d. as a 'nuisance parameter'). This generally works very well. However, for very small data sets this might produce unwanted results, especially when the model can go *exactly* through the data point(s). In those cases, it makes sense to specify the variance for each data set. For an example with two state variables:

```
glo.var = [0.12 0.41]; % supply residual variance for two states
```

If you apply transformations, this is the variance *after* transformation. As with the `glo.wts` in the previous section, the `glo.var` must have the same size as the data set. For survival data, the entry in `glo.var` is meaningless, and is ignored.

2.6 Zero-variate data

The way of specifying data sets in BYOM assumes that there is an independent variable (generally 'time') against which the observations on a state variable can be plotted. Some types of data, however, do not fall neatly into this category. Suppose I have an energy-budget model that specifies growth and reproduction of an animal. I have data on length versus time (which can go into `DATA{1}`), but for reproduction I only have a maximum reproduction rate of 120 eggs/day (something that I cannot

put a time point on, and cannot go nicely into `DATA{2}`). The model parameters that I want to fit *do* specify the maximum reproduction rate, but to include this 'zero-variate' information into the fit, we need to enter them differently.

After the parameters, I can provide this data point as follows:

```
zvd.Rm = [120 5]; % zero-variate data point with normal s.d.
```

Note that I have to specify a standard deviation to judge the residual (the difference between the model prediction and the data point). As there is only a single data point, there is no information in the data set itself on the residual standard deviation. I assume that a normal distribution is appropriate.

Somewhere, we need to tell the fitting routine what model prediction to compare this zero-variate data point to. This needs to be done in `call_deri`. For example, if the maximum reproduction rate is the product of parameter `a` and `b`:

```
zvd.Rm(3) = par.a(1) * par.b(1); % predicted max. repro rate
```

If you have a separate `Rm` for different scenarios, you need to specify a new zero-variate data point for each one. For example in the script file, specify:

```
zvd.Rm1 = [120 5]; % zero-variate data point with normal s.d.
zvd.Rm2 = [172 5]; % zero-variate data point with normal s.d.
```

And in `call_deri`:

```
switch c
    case 1
        zvd.Rm1(3) = ... ; % predicted max. repro rate
    case 2
        zvd.Rm2(3) = ... ; % predicted max. repro rate
end
```

2.7 Splining a data set

Sometimes, a data set is used as a forcing function for the model, and not as something to fit on. A good example is the use of measured (or estimated) exposure concentrations in a pulse-exposure experiment. The function `derivatives` needs to have a concentration at each possible value for time t, but we have measurements only at a few times. The solution is to interpolate using a cubic spline.

We don't use the global array `DATA` in this case, as we are going to work with this data set as a forcing and not as observations on a state variable. For example, we have an exposure scenario with two pulses for the water concentration C_W:

```
Cw = [0      1
      0    108
      1    102
      1.01   0
      2.99   0
      3    112
      4    107
      4.01   0
      6      0];
```

Note that we still need to have the scenario number in the first row, as we might have different forcing functions for each treatment. We can prepare this data set for splining with the following utility:

```
make_pp(Cw,1); % prepare the spline for interpolation
```

Now, the information of the spline through the data points is made available in two global parameters: `pp_coll` and `pp_scen`. The `1` at the end makes sure that also a plot is produced of the pulse scenario, with the spline through the points (which can help to see if the spline is realistic). In `derivatives`, we can use this information again. First, in the top of the file, make the new globals available by adding:

```
global pp_coll pp_scen % for splined external concentrations
```

At the place where the exposure information is needed (here in the form of the variable `Cexp`), we can use the spline as follows:

```
[pp_exist,pp_loc] = ismember(c,pp_scen); % where is c in the global?
c = pp_coll{pp_loc}(t); % use the pp form already provided
```

A more elaborate example is provided in the `GUTS2` package, in the dedicated sub-folder `timevar_diazinon`.[2]

[2]In older versions of Matlab, the code needs to be different to make it work. The sub-folder for diazinon in the GUTS2 package has some code to use both the old and the new method.

3 BYOM in depth: the model parameters

3.1 Defining parameters

Model parameters are specified in the structure `par`. To define a parameter `a` with a starting value of 3.2, to be fitted, with a realistic range of $0 - 100$, and fitted on normal scale:

```
par.a = [3.2 1 0 100 1];
```

Make sure that the starting value lies with the allowable range. For each parameter, you can specify whether you want to fit it or keep it fixed to the starting value. This is especially handy when the model has a large number of parameters: keep some parameters fixed, and fit the rest. Copy the results into the starting value position, and fit more parameters, etc. The last element is optional; a 1 will be filled in if you forget it, which is done to ensure compatibility with older script files.

In `derivatives`, the same parameter names are available. They can be used directly in equations, such as:

```
dX = par.a(1) * X; % differential equation for X
```

or first translated into an easier format to increase readibility:

```
a = par.a(1); % extract parameter a from the structure
dX = a * X; % differential equation for X
```

Note that the `(1)` is needed after the `par.a` as otherwise you will get the entire vector of five values (the fit 1/0, the min-max range, and the log-scale 1/0).

The same parameter structure `par` is available in `call_deri` (and in the events section of that function). If possible, avoid having calculations in `call_deri`, but in some cases that is unavoidable (e.g., when a parameter is used to specify a starting value or a zero-variate model prediction).

3.2 Global parameters

Some parameters are needed to provide information to `derivatives` but may never be fitted (e.g., a reference temperature, or a parameter to switch something). In those cases, it makes more sense to specify them as a global in the structure `glo`. For example, if we have a reference temperature of 283 Kelvin:

```
glo.Tref = 283; % reference temperature as global
```

In `derivatives`, this value can then be used, e.g.:

```
Tref = glo.Tref; % reference temperature as global
```

The same global structure `glo` is available in `call_deri` (and in the events section of that function).

Several globals are already in use by BYOM, so avoid them: `glo.wts`, `glo.var`, `glo.Tinit`, `glo.sel` (in the GUTS packages), `glo.spln` (for fitting splining nodes), `glo.opt` (for optimisation options), and `glo.plt` (for plotting options).

3.3 Adding a prior distribution

Every model parameter can have a prior probability distribution. In a Bayesian framework, this is the degree of belief in a parameter's value before looking at the data. In a frequentist framework, we can still work with priors but we probably need to call them 'prior likelihood functions', and they would need to follow from an analysis of other data sets (which is not strictly needed for Bayesians). In either case, a prior can be defined in your BYOM script. For example, to use a triangular distribution for parameter a, with a mode of 10, and a range of $6 - 18$:

```
pri.a   = [2 6 18 10]; % triangular, min, max and center
```

The `2` indicates the choice for a triangular distribution. The possible distributions are specified in the engine function `calc_prior`, and also include beta, normal and lognormal at the moment (but it is easy to include more distributions here). Note that I use the Statistics toolbox for most of these distributions. If no prior is defined for a parameter, a uniform distribution is used, with the ranges as specified in the regular definition of the parameter in the structure `par`.

The likelihood function that is maximised combines the fit to the data with the prior information. In this example, parameter a cannot take values outside the prior range, and values near the mode will be preferred (unless the data have a strong preference for values closer to the edges of the range). The prior is very similar to a zero-variate data point that we discussed earlier. Only, the distribution for the zero-variate data point is tested against a model output instead of a model parameter.

3.4 Fitting splining nodes

This option is used to reconstruct a forcing of the model. Best example is when we have growth of an organism under unknown food conditions, we can use the growth data to reconstruct the food availability over time that the organism experienced. The relative food availability f, is given by a spline function, defined by several 'nodes'. We can arbitrarily set the time vector for these nodes, but their value of f should be a model parameter, to be optimised on the data. An example of 4 nodes, at regular intervals, with an initial estimate for f, is given in a matrix:

```
SPL = [ 0 0.7
        5 0.7
       10 0.7
       15 0.7];
```

To translate this matrix into a part of the parameter structure `par`, we can use the `packunpack` function in the engine as follows:

```
glo.spln{1} = SPL(:,1); % put time vector in global
glo.spln{2} = 'node'; % name for the parameters that represent nodes
glo.spln{3} = [0 1.05]; % min/max range for f
par = packunpack(3,par,SPL); % pack matrix SPL into par structure
```

This creates the paramaters `par.node1`, `par.node2`, `par.node3` and `par.node4`, with a minimum-maximum range, that will be fitted to the data. In `derivatives`, we can extract the matrix again from the parameter structure:

```
splnmat = packunpack(4,par,0); % extract nodes from par structure
f = interp1(splnmat(:,1),splnmat(:,2),t,'pchip'); % estimate f at t
```

Disadvantage of this way to implement this is that it is slow: `derivatives` is called many times for one iteration of the optimisation routine, and the packing and unpacking of the structure is relatively slow. It is also unnecessary as the nodes do not change with the value for t that is tried in the ODE solver. It would suffice to unpack the parameter structure only once per iteration of the optimisation routine. This can be done by unpacking in `call_deri`, and using a global to convey that information to `derivatives`.

4 BYOM in depth: confidence intervals

This manual is not the place to go into the details of confidence intervals, likelihood functions, and Bayesian statistics. Some more information is presented in the refresher that can be downloaded from http://www.debtox.info/downloads/coursemat/refresher.pdf, and the technical document from http://www.debtox.info/downloads/book/debtox_tech.pdf.

4.1 Asymptotic standard errors

Asymptotic standard errors and correlations between parameters can be calculated by calling the function `calc_ase` in your script, after the call to `calc_and_plot`, as follows:

```
calc_ase(par_out)
```

Basis for the calculation is a numerical derivation of the second derivative of the likelihood function to the parameters (the Fisher information matrix, see https://en.wikipedia.org/wiki/Fisher_information).

I am not very happy with this calculation, it uses only the local information around the best estimate and it is not very robust (the outcome often depends on how far away from the best estimate we take the other points to calculate the derviatives), so I would suggest to use profile likelihoods instead.

4.2 Profile likelihood

Profiling is a great way to produce relevant confidence intervals on parameter estimates. Also, it might find a better estimate (when the optimisation routine ended in a local minimum). The parameters for the better estimate will be presented on screen and in the log file `profiles_newopt.out`. The profile is plotted as it is calculated, so you can see what's going on. The shape of the profile can also be very informing about the type of likelihood 'landscape' that this parameter lives in. The broken line indicates the cut-off criterion for 95% probability from the χ^2 distribution, which signifies the confidence interval.

Profiles have to be made for each parameter separately by adding a call to `calc_proflik` line after `calc_and_plot`. For example, for the parameter a use:

```
calc_proflik(par_out,'a')
```

The scripts `byom_bioconc_...` also contains a piece of code to automatically create profiles for every parameter that is fitted to the data.

Note that profiling for complex models can be very slow, so only do this after you are happy with the fit.

4.3 Bayes and the slice sampler

A Bayesian way to construct credible intervals is to take a sample from the joint posterior distribution of the parameters. If you did not define prior distributions, uniform ones will be assumed (specified by the min-max ranges in your parameter structure `par`). A discussion of the Bayesian philosophy is outside of the scope of this manual. What you get, however, is a sample from parameter space, that is used to construct credible intervals for the parameters, and to show the correlation structure of the parameter estimates. The sample can subsequently be used to make intervals on model predictions. The code in the script is:

```
calc_slice(par_out,[500,1,0],mfilename);
calc_conf
calc_ellipse(mfilename_base,par_out,'ke','Piw');
```

The function `calc_slice` creates the sample (this requires the Statistics toolbox of Matlab). First argument is the number of samples; 500 samples is not very much, but illustrates how the method works (more samples obviously require more calculation time). Second argument is the 'thinning'; how many samples are kept and how many are thrown away (`1` means that all samples are kept). The last argument is how many samples are thrown away before collecting them. If you start from the best fit for the parameters, a burn-in should not be needed, so keep this at `0`.

The script `calc_conf` calculates and plots credible intervals on the model curves as dotted lines. The function `calc_ellipse` plots an 'error ellipse' for any two parameters that you like (in the example ke and Piw). Just run the example in the script `byom_bioconc_deluxe` and try to interpret the output on the screen and in the plots.

The `GUTS2` package makes nice use of these options to create credible intervals on the LC50 (which is a model output of GUTS).