

DDEfit: a program for fitting (partially specified) DDE's, ODE's and discrete time models to data

Simon N. Wood,
Mathematical Institute, North Haugh, St. Andrews, Fife KY16 9SS, UK.
snw@st-and.ac.uk
<http://www.ruwpa.st-and.ac.uk/simon.html>

Contents

1	Overview	2
2	Getting started	2
3	Computational requirements	3
3.1	gcc	4
3.2	Microsoft Visual C++	4
3.3	Other compilers	5
4	The Interface	5
4.1	The menu controls	5
4.2	The model and data series windows	7
4.3	The parameter window	7
4.4	The status window	7
4.5	The unknown function windows	7
5	Defining models	7
5.1	Global constants	7
5.2	Working with model unknowns	7
5.3	Continuous time models	8
5.3.1	State variables and constants	9
5.3.2	switchfunctions(sw,s,c,t)	9
5.3.3	map(s,c,t,swno)	9
5.3.4	grad(g,s,c,t)	10
5.3.5	storehistory(his,ghis,g,s,c,t)	10
5.3.6	initst(s,c,t)	11
5.3.7	statescale(double *scale)	11
5.4	Discrete time models	11
5.4.1	map(s,c,t,swno) for discrete time models	12
5.4.2	initst(s,c,t)	12
5.5	pastvalue(i,t,j)	12
5.6	initfit()	13
5.6.1	Constants	13
5.6.2	Fitting method and Error model	14
5.6.3	Unknown and known coefficients	14
5.6.4	Unknown functions	15
5.6.5	Data to fit	15
5.6.6	Fitting other statistics	16
5.6.7	Setting up output	16
5.7	inuf()	17
5.8	bootstrapping: inboot()	17
5.9	Miscellaneous modelling hints	18

6	Structure of input data files	19
7	Examples	19
7.1	cr.c: a simple discrete time consumer resource model	19
7.2	dexp.c: a simple continuous time model with a switch	20
7.3	znew.c: a discrete time model of Larch Bud Moth at 2 sites.	20
7.4	cop.c a continuous time models with unknown functions	21
8	Program structure	21

1 Overview

DDEfit is a package designed for fitting some classes of ecological model to data. The models can be specified as delay differential equations, differential equations or difference equations, and some partial differential equations can also be used. At present the population models dealt with are deterministic, so the data are modelled as noisy observations of a deterministic process, some components of which (parameters or functions) are unknown and must be estimated. Models may be partially specified. This means that some of the component functions that make up the specification of a model can be represented simply as general smooth non-parametric functions, rather than being written down parametrically.

Method details can be found in:

Wood, S.N. (2000) Partially Specified Ecological Models *Ecological Monographs*.

The user specifies their model, and chooses various characteristics of the fitting process by editing a template C source code file. This source code is then compiled and linked with the other source files that make up the project, to produce a model fitting program with a graphical user interface, which is designed to help fit the model to data. The program allows the user a fair degree of interactivity in the model fitting process, but also includes general automatic fitting methods.

Broadly the fitting process works like this. The user's biological model will usually be a fairly complicated mathematical construct, which can be solved numerically in order to predict populations or other variables at a series of times. Typically some parameters, initial values and/or functions that make up the model will be unknown to the modeller and must be estimated by fitting. That is, the model unknowns will be adjusted so as to achieve the best match between model and data, according to some criterion selected by the modeller (or forced upon them by what's implemented!). At a moderately abstract level, the user's model can be thought of as a function which will produce predictions corresponding to the user's data, *given particular values for the unknowns of the model*. Writing the data as a vector \mathbf{y} , the model predictions of the data as $\hat{\boldsymbol{\mu}}$ and the unknowns in a vector \mathbf{p} , then the model amounts to :

$$\hat{\boldsymbol{\mu}} = f(\mathbf{p})$$

where f is a vector valued function, defined implicitly by the users model. This general view of the model enables progress to be made on the problem of how to find the values for \mathbf{p} which will make $\hat{\boldsymbol{\mu}}$ best match \mathbf{y} . By considering Taylor series expansions of f about an estimate of \mathbf{p} , fairly general methods can be found for obtaining improved \mathbf{p} estimates.

2 Getting started

Once you are familiar with the package it is fairly straightforward to use (the most difficult task is usually model formulation), however there are a fair number of details to be dealt with if the package is to be used successfully, and to avoid frustration it is probably best to start by working through the example files provided with the package in association with this documentation.

The steps to be gone through to set up a model for fitting are as follows:

1. Write down the equations defining the model to be fitted, and specify any constraints to be applied to model unknowns.

2. Produce the text files containing the data to be fitted and the weights to give each datum.
3. Code the model (and specify any constraints) using the C template file provided.
4. Decide how the model and data are to be displayed and specify this by filling out appropriate entries in the C template file.
5. Choose a fitting method, and specify other details associated with fitting by filling out entries in the C template file.
6. Optionally, specify details of any bootstrapping analysis to be carried out in the C template file.
7. Compile the template file (and the other program files as needed) and link this to other compiled program files, to produce an executable model fitting program.
8. Once the model compiles and links without error, debug it (usually by using the simulation options provided).
9. Check through the model code trying to prove that it is incorrect, until you fail to do so, and can compile a final version of `DDEfit` to use for actual fitting.
10. Fit the model, usually iterating between automatic fitting and some hand tuning to get sensible starting values, or to escape silly local minima.

I'd suggest the following as the quickest route to productive use of the program.

1. Compile and link one of the simple examples provided, to check that you can get a model to compile and link effectively, and in order to find your way around the user interface of the compiled program.
2. Compile and link one or two more complicated examples, to get some familiarity with using the unknown function Windows etc.
3. Read through the documentation on model specification with reference to the examples already used.
4. Read through the documentation on setting up output and model fitting with reference to the examples already used.
5. Try adjusting various details of fitting and display in the example model template files, to check understanding of how these things are set up.
6. Try altering the models slightly, in order to check understanding of model specification.
7. Now move on to the problems that you are really interested in ...

3 Computational requirements

The package is usable on any machine running the Windows 9* or Windows NT operating system. You will need a C/C++ compiler/linker package which can produce multi-threaded Win32 GUI (graphical user interface) applications. All the windows features used are programmed directly using the Windows API, so you don't need any additional toolboxes and are not restricted to any particular compiler. There are currently several suitable compilers available free.

3.1 gcc

For information on gcc and Windows see:

<http://www.geocities.com/Tokyo/Towers/6162/gcc.html>

which includes links to suitable compilers, e.g.

<http://www.nanotech.wisc.edu/~khan/software/gnu-win32/gcc.html>

This compiler is excellent although the debugging tools are not quite as friendly as the offerings from commercial software houses.

Suppose that your model definition file is called `model.c`. To compile and link `DDEfit`:

1. Open an msdos prompt, and change to the directory containing all the `.c`, `.h` and `.rc` files.
2. Compile the source code for your model and the package with the line:

```
gcc -c ddefit95.c ddeq.c w95dde.c matrix.c qp.c spline.c gcv.c rangen.c stochmin.c  
map.c model.c
```

this will yield object files: `ddefit95.o`, `ddeq.o`, `w95dde.o`, `matrix.o`, `qp.o`, `spline.o`, `gcv.o`, `rangen.o`, `stochmin.o`, `map.o` and `model.o`
3. Compile the windows resources that are needed:

```
windres -i ddefit.rc -o ddefit.o.
```
4. Link the object files into an executable:

```
gcc -o ddefit.exe ddefit95.o ddeq.o w95dde.o matrix.o qp.o spline.o gcv.o rangen.o  
stochmin.o map.o model.o ddefit.o -mwindows
```

Now type `ddefit` at the command prompt to run the program.

Of course, it is more convenient to compile all the source files except the model template file just once and combine all that can be combined into a library. For example:

```
ar -ru ddefit.a ddefit95.o ddeq.o matrix.o qp.o spline.o gcv.o rangen.o stochmin.o map.o
```

produces a library called `ddefit.a` which can be linked to your compiled model file with the command:

```
gcc -o ddefit.exe model.o w95dde.o ddefit.o ddefit.a -mwindows
```

Note that because of the way the linker decides what to include from the `.a` file, `w95dde.o`, which contains the windows entry point (`WinMain()`), and `ddefit.o`, containing the compiled resources, should not be put in the library. For the same reason the `.a` file must come after the other three object files in the argument list to `gcc`.

3.2 Microsoft Visual C++

If you are going to use a commercial compiler, then I reluctantly recommend Microsoft Visual C++ (see the end of this section for one of the reasons for reluctance). I am currently using version 6.0. There is no point in using the command line version of this compiler, so I will assume that the visual studio is being used.

1. Start Microsoft Visual Studio.
2. Select `File>New>Projects>Win32 Application` from the main menu and subsequent dialogs. Give the project a name in the dialog box provided.
3. Find the “FileView” window.
4. Add `ddefit95.c`, `ddeq.c`, `w95dde.c`, `matrix.c`, `qp.c`, `spline.c`, `gcv.c`, `rangen.c`, `stochmin.c`, `map.c` and `model.c` to the “Source Files” folder. Right click on the folder to do get the option of adding files. (Ctrl and a mouse click allows you to pick a sequence of files in the File open dialog box).
5. Add `ddefit.rc` to the “Resource Files” folder.
6. Select `Project>Settings` from the main menu, and find the C/C++ tab in the resulting Project Settings window. Select `Code generation` in the Category option list. Then select `Debug Multi-threaded` or (if you anticipate making no coding or logical errors) `Multithreaded` from the Use run-time library list. Hit OK when done. (The compiler option being added here is `/MTd` or `/MT`.)

7. Select Build from the main menu to compile and link the project.
8. Note that if you delete one model file from the project and add an alternative one, you may need to select Rebuild All from the Build menu to get Visual Studio to notice!

Finally, some documentation from different parts of the MSDN online documentation

1. Window Classes in Win32
Extra Class and Window Bytes (cbClsExtra and cbWndExtra)

In Windows 95 and Windows NT, there is no reason to limit the number of extra bytes for a window or class to a small number such as 40. However, developers should choose reasonable values when determining their extra byte requirements.

2. RegisterClassEx

Windows 95: RegisterClassEx fails if the cbWndExtra or cbClsExtra member of the WNDCLASSEX structure contains more than 40 bytes.

3.3 Other compilers

Borland C provided a very nice development environment, but with every Borland compiler I've been able to try it on `sqrt(1e-311)` returns the value `1e-311` (with other floating point functions doing the same sort of thing). This can be quite inconvenient, and can cause occasional problems, as the code for DDEfit assumes graceful underflow to zero. However, a command line version of Borland C/C++ 5.5 is available free at:
<http://www.borland.com/bcppbuilder/freecompiler/>

4 The Interface

When you run DDEfit it will start by displaying a parent window containing child windows for model output with data, along with a child window for the unknown coefficients of the model (if there are any) and a child window for each unknown model function. Near the top of the parent window will be a menu from which the program can be controlled. On start up the initial values of the unknown coefficients and the initial unknown functions should be displayed, but no data or model runs will be present until you select Sim, Try or Fit. If you resize or move any of the child windows then the parent will remember this information and use it for window arrangement until you next select auto-arrangement.

The program is designed to allow the user to experiment with different parameters to get a reasonable set of starting parameters for automatic fitting. Fitting can be interrupted at any point for the user to alter parameters etc. and/or restart the fit. Hence a typical fitting session will be interactive, with the user helping the automatic routines, when they get stuck in awkward corners of parameter space.

4.1 The menu controls

The menus control simulation fitting and input/output. Table 1 lists the items and their function. The Sim menu is useful for exploring model behaviour, testing models and simulating data in order to test model fitting. It's always a good idea to see if you can obtain a decent fit to data simulated from a model before giving too much credence to the fit of the model to real data. Try and Fit are used for fitting to data. Try lets you play with unknown parameters and functions and see how this alters the fit, while Fit is used to start automatic fitting or bootstrapping. At present most of the control of fitting options has to be coded into `initfit()`. The File menu deals with reading in parameters from file and writing out various outputs and estimates to file. Stop lets you halt a fit, simulation or bootstrap in progress, while Arrange offers options for automatically arranging output Windows.

Menu Item	Sub-Item	Purpose
Sim		This menu controls simulation - that is model runs that are not designed to fit data (but may be used to simulate it).
	Run	Numerically solves the model once, using the parameters given in the parameter window and/or u.f.s displayed in the u.f. windows.
	Options	Set simulation options - output timestep, stop time and noise level for data simulation.
	Output	Output simulation to a file - This outputs either the state variables or noisy data simulated by adding normal random deviates to state variables at sample times.
Try		Runs model using parameters and u.f.s from parameter and u.f. windows and compares the result with data (which is also displayed) - fitting objective function values are given in the status window (without penalty terms).
Fit	Go	Perform a single fit of the model to user specified data, using the user specified method.
	Bootstrap	Run bootstrap replicates model fits, using the information provided by the user in <code>inboot()</code> .
File		Deal with file i/o.
	Read Parameters	Read unknown coefficients and parameters of u.f.s from a user specified file.
	Write Parameters	Write unknown coefficients and parameters of u.f.s to a user specified file.
	Output fit	Write fitted values and data to one file.
	Output r-squared	Output r^2 values to a file (overall and by series).
	Cov./cor. mnatrix	Write covariance/correlation matrix for all unknown parameters to a file (correlation below the leading diagonal, covariances on and above).
	Standard errors	Output parameter estimates and standard error estimates, for all parameters and for unknown functions as functions.
Stop		Stop the current simulation, trial, fit or bootstrapping (gracefully).
Arrange		Menu for automatic arrangement of child display windows within the main parent window. This stuff is horrible to write, because you don't know the number or size of windows in advance.
	Option 0	Arrangement with u.f.s along bottom of display.
	Option 1	Arrangement with u.f.s up the side of the display.

Table 1: The DDEfit menu.

4.2 The model and data series windows

These display state variables and data to be fitted in colour coded form, to help visually match model state variables and corresponding data. The y axis range can be set by clicking on the `Rescale` button at the bottom left of each such window.

4.3 The parameter window

This contains the current estimates of the unknown coefficients of the model. Values of parameters can be edited within this window whenever the program is not actually fitting or bootstrapping.

4.4 The status window

When model fitting or trying fits by hand, this window is present and displays information on the size of the objective function etc. The window is not present when simulating.

4.5 The unknown function windows

There will be one of these for each unknown function. They display $x - y$ plots of the u.f.s. The y axis of each can be rescaled by pressing the `R` button within the window. The axis label fonts rescale with window size, so if you can't read the axes, enlarge the window. The functions are plotted with at most 10 boxes superimposed on the line representing the function. These can be dragged using a mouse pointer, in order to adjust the form of the u.f. concerned. The x axis of the plot features a black line indicating the range of values at which the u.f. was evaluated in the last model run.

5 Defining models

This section deals with defining models for fitting. This is done by editing standard template files in `C`, which are then compiled and linked to the rest of the code that makes up `DDefit`. Use the model definition files supplied with the package as template files to modify. For maximum fitting efficiency discrete time models are dealt with differently from continuous time models, so this section starts with general information pertaining to both model classes, then deals specifically with continuous models, followed by discrete models, before finishing off with information common to both model classes: fit method selection, control of output and initialisation of model unknowns.

5.1 Global constants

You can specify global model constants at the top of the file. It is ok to set these from routines: `initst()`, `initfit()` and `inuf()`. You should be very careful about setting them elsewhere. Never set global constants from routine `grad()` - the way in which numerical differential equation solvers work means that `grad()` is frequently called with state variable values that are erroneous. `grad()` is not called in time order either! If you really must set global constants from within your model, then it is usually safe to do so from within `map()`. At the potential cost of some loss of numerical accuracy it is safe to set globals from routine `storehistory()` (the problem is that the algorithm for setting the timestep can not see the changes in the global variable, and hence ignores them when working out how big the integration step should be).

5.2 Working with model unknowns

Within your model code, unknown coefficients and unknown constants are accessed using the macros `uc()` and `uf()`. You set up the number and characteristics of unknown functions and constants in `initfit()` (see below). Initialisation of u.f.'s is done in `inuf()`, initialisation of u.c.'s is performed in `initfit()`

- The i^{th} unknown constant is accessed by `uc(i)`. i starts at 0. You can not write to unknown constants. Any function that you write which must access unknown constants must have the *fixed* constant vector `c` as an argument *even if your model has no fixed constants*.
- The i^{th} unknown function evaluated at x is accessed by `uf(i, x)`. i starts at 0. Again, unknown functions are read-only within your model code. The first derivative of the i^{th} unknown function w.r.t. its argument is obtained from `guf(i, x)`; the integral of this function from a to b is `iuf(i, a, b)`
- The program does not (yet) ensure that finite difference calculations will not involve constraint violations. Hence you get to ensure that your model copes gracefully with constraint violations.

u.f.'s and u.c.'s are displayed as follows:

- u.c.'s are displayed in labelled edit boxes in the *parameter window*. They may be edited and the model re-run by selecting `Try` or `Fit>Go` from the menu.
- Each u.f. has its own window. Before fitting, the u.f. can be altered by dragging the little boxes that the function connects. The plot can be rescaled by clicking on the `R` button. Font sizes scale with the window size. The horizontal plot axis shows the realized function range in black.

5.3 Continuous time models

To define a model you need to edit various functions in a standard template file. You will also need to use some standard functions. This section explains these for the case of continuous time models. The following table summarises what you need to know about for each class of continuous time models that the package can deal with (SDDE stands for delay differential equations with switches). You can define discrete time models using the information in this section, but for this case it is much more efficient to use the approach given in the next section:

Functions to define	ODE	DDE	SDDE
<code>initcons()</code>	•	•	•
<code>switchfunctions()</code>			•
<code>map()</code>			•
<code>grad()</code>	•	•	•
<code>storehistory()</code>		•	•
<code>initst()</code>	•	•	•
<code>statescale()</code>	•	•	•
<code>initfit()</code>	•	•	•
Functions to use			
<code>pastvalue()</code>		•	•
<code>pastgradient()</code>		•	•

Furthermore:

- Models which include unknown coefficients can access these using `uc()`.
- Models which include unknown functions will need to specify initial values for these in `inuf()`, while unknown functions can be accessed using `uf()`, `guf()` and `iuf()`.

Note that (i) functions that are not needed for a model class can be left empty, and (ii) in everything that follows (without loss of generality) I will assume that integration is with respect to time.

5.3.1 State variables and constants

The program aims to fit models of the general form:

$$\begin{aligned}\frac{ds_0}{dt} &= g_0(s(t), s(t-\tau_0), s(t-\tau_1), \dots, t) \\ \frac{ds_1}{dt} &= g_1(s(t), s(t-\tau_0), s(t-\tau_1), \dots, t) \\ \frac{ds_2}{dt} &= g_2(s(t), s(t-\tau_0), s(t-\tau_1), \dots, t) \\ &\vdots \\ &\vdots\end{aligned}$$

subject to initial conditions on the s_i 's, and with the possible addition of some discontinuities in the s_i 's. The s_i 's are the *state variables* of the problem and $s(t)$ is used to mean the vector of all state variables at time t , so $s(t-\tau)$ is the vector of state variables time τ ago. To define a model the user must specify the functions g_i which return the gradients of s_i w.r.t. time given the state variables at time t and at a series of lagged times. The user must also specify the starting values for all state variables.

Within the program the state variables are passed to and from functions as an array $s[]$. The functions defining the gradients of the state variables may depend on various known user defined constants (as may the initial values and any discontinuous changes in state) - these constants are passed around in an array $c[]$. The gradients, initial values and discontinuous changes can also depend on unknown constants accessed through `uc()` and unknown functions accessed through routines `uf()`, `iuf()` and `guf()`. All arrays start at element 0, not element 1. Elements of $s[]$ must not be modified anywhere, except in `map()` and `initstate()`.

5.3.2 `switchfunctions(sw, s, c, t)`

The arguments of this function are all pointers to double, except t , the time, which is a double. sw is an array in which to return switchfunction values; $s[]$ is the array of state variables; $c[]$ is the array of constants. Elements of $s[]$ must not be altered in this function.

Switches are discontinuous changes in state variables. Switchfunctions are functions that pass through zero, with a negative time derivative at the time that a switch is to occur. Switches have an index (starting at zero). Users define the switchfunctions in the `switchfunctions()` routine by setting the values of the switchfunctions in the array $sw[]$. For example, if the 0th switch is to occur every 33.2 time units starting at time 11.1 the following line could be used:

```
sw[0] = - sin(6.28318530718*(t-11.1)/33.2);
```

As another example, if switch 1 is to occur just once at time 55.67 then the following line is appropriate:

```
sw[1] = 55.67 - t;
```

Switchfunctions can depend on constants and state variables, although some care may be required to ensure a well defined model when switchfunctions are state variable dependent.

When the i^{th} switchfunction passes through zero, from positive to negative, then integration is suspended (in an orderly fashion) and the routine `map()` is called with i passed as argument `swno`. `map()` is where the user defines switches.

5.3.3 `map(s, c, t, swno)`

The arguments of this function are (pointer to double) $s[]$ the array of state variables; (pointer to double) $c[]$ the array of constants; (double) t the time; (integer) `swno`, the index of the switch whose switchfunction is passing through zero (with negative slope).

`map()` is where you must define the switches, i.e. the discontinuous changes in state variables $s[]$ triggered by a switchfunction descending through zero (see `switchfunctions()` above). For example, in the following code state variables 0 and 1 are doubled by switch 0 every time it is triggered, while state variable 0 is multiplied by $c[0]/t$ by switch 1 whenever it is triggered:

```
if (swno==0) {s[0]*=2;s[1]*=2;} else
if (swno==1) {s[0]*=c[0]/t;}
```

5.3.4 grad(g,s,c,t)

The arguments of this function are all pointers to double, except time t , which is a double; $s[]$ is the state variable array; $c[]$ is the array of model constants; $g[]$ is the array in which the user must return the time derivatives of the state variables. Elements of $s[]$ must not be altered in this function. It is very unlikely to be a good idea to change elements of $c[]$ in this function.

This is the routine in which the user specifies the gradients of the state variables at time t , given the the state variables at time t (supplied in $s[]$), the model constants, and (optionally) lagged values of the state variables (or lagged functions of state variables). For each state variable $s[i]$ you must supply a value in $g[i]$. For example if the equation for the gradient of state variable 0 is:

$$\frac{ds_0}{dt} = c_0 s_0 - \delta s_0 s_1$$

where δ is unknown coefficient 0, then the corresponding line in the routine `grad()` would be:
`g[0]=c[0]*s[0]-uc(0)*s[0]*s[1];`

Lagged variables are accessed using the routines `pastvalue()`. This is described later, but for completeness an example of its use is given here. Suppose that you want to code the equation:

$$\frac{ds_0}{dt} = \begin{cases} as_0(0) & t < b \\ as_0(t-b) & t \geq b \end{cases}$$

subject to the initial condition $s_0(0) = d$, where a , b and d are the first 3 unknown constants. Assuming that the 0^{th} history variable (see `storehistory()` and `pastvalue()`, below) contains the lagged values of $s[0]$ then the appropriate code would be:

```
if (t<uc(1)) g[0]=uc(0)*s[0];
else g[0]=uc(0)*pastvalue(0,t-uc(1),0);
```

Notes:

1. If the gradient changes discontinuously, then it is often good practice to specify a switch at the time of the discontinuity: this forces the integration to step to exactly the point of discontinuity, before continuing, which ensures that the continuity assumptions of the integrator are not violated (the specified switch should leave all state variables unchanged).
2. The specification of your model should never involve writing to global variables from within `grad()`: `grad()` is called multiple times per step of the integrator, and t may increase *or decrease* between calls. Similarly writing to static variables is very rarely appropriate as part of model specification.
3. $g[]$ contains meaningless values on entry to the function.

5.3.5 storehistory(his,ghis,g,s,c,t)

The arguments of this function are all pointers to double except for time t which is a double. $g[]$ and $s[]$ contain the current values of the gradients of the state variables (w.r.t. time) and the state variables; their elements must not be altered by this function. $c[]$ is the array of model constants. $his[]$ and $ghis[]$ are arrays in which you should store the value and time derivative, at time t , of any quantity that you want to use as a lagged variable within your model.

If your model equations depend on lagged quantities then these must be stored. `storehistory()` is the function that allows you to do this. The lagged variables (or “history variables”) are stored only at discrete times. Interpolation is necessary to estimate the values of lagged variables between these storage times. In order to ensure correct adaptive stepping for integration this interpolation must be performed to a higher order of accuracy than the integration. To achieve this requires that both the values *and time derivatives* of the state variables be stored¹.

¹This consistency of integrator and interpolator is quite important for ensuring reasonable accuracy of numerical derivatives of the fitted values w.r.t. parameter values.

The lagged values and gradients for each history variable are stored in a ringbuffer for access by `pastvalue()` (see below).

As an example, if your model is the simple DDE:

$$\frac{ds_0}{dt} = \begin{cases} as_0(0) & t < b \\ as_0(t-b) & t \geq b \end{cases}$$

where a is unknown coefficient 0 and b is unknown coefficient 1, then history variable 0 would be defined as s_0 . The appropriate piece of code within `storehistory()` would be:

```
his[0]=s[0];
ghis[0]=g[0];
```

and the within `grad()` the 0^{th} state variable would be used to specify the gradient of s_0 as follows:

```
if (t<uc(1)) g[0]=uc(0)*s[0];
else g[0]=uc(0)*pastvalue(0,t-uc(1),0);
```

History variables do not have to be state variables themselves (although it's usually easiest to set models up that way), for example if history variable 1 was to be $e^{s_0} + 2s_1$ then the appropriate code in `storehistory()` would be:

```
his[1]=exp(s[0])+2*s[1];
ghis[1]=g[0]*exp(s[0])+2*g[1];
```

If you must write to global variables, this routine is a reasonable place to do it, but be aware that you may compromise the accuracy of numerical model solution (since the adaptive stepping algorithm has no way of 'seeing' this change in a global variable).

5.3.6 `initst(s,c,t)`

The arguments of this routine are: (pointer to double) s , the state variable array; (pointer to double) c the constant array; (double) t the time at start of model integration.

This function is where the initial values of all state variables must be set by the user. For example to set state variable 0 to c_3 and state variable 1 to unknown coefficient 0, you would use the following code:

```
s[0]=c[3];
s[1]=uc(0);
```

It is quite safe to modify global variables and elements of $c[]$ within this function.

5.3.7 `statescale(double *scale)`

DDEfit controls integration error by specifying that its magnitude should be less than some specified proportion of the magnitude of each state variable. This can cause difficulties when some state variables are in the vicinity of zero. For example, if a state variable is at zero, but about to leave that state, then the integrator may attempt to estimate that variable with zero error - which requires a zero timestep! The solution adopted is to get the user to specify a small number to be added to the magnitude of each state variable when estimating proportional integration error. These numbers (one for each state variable) must be supplied by the user in array `scale[]`.

Any elements of `scale[]` that are unspecified are taken as zero.

5.4 Discrete time models

Discrete time models are much less fuss to specify! The relevant template file only requires that you fill out functions `map()`, `initst()` and (as with all models) `initfit()`. Unknowns are accessed using `uc()`, `uf()`, `guf()` and `iuf()` (as in the continuous time case).

Note that it is very important that discrete time model definition files include the line `#include map.h` and that `initfit()` somewhere includes the line: `d->discrete=1`.

5.4.1 `map(s,c,t,swno)` for discrete time models

The arguments `s` and `c` to this function are pointers to double while `t` is a double. `swno` is not used in the context of specifying discrete time models and is there for reasons of compatibility with the code for continuous models.

The purpose of this routine is to iterate a discrete time model of the general form:

$$\begin{aligned} s_0(t+1) &= f_0(s(t)) \\ s_1(t+1) &= f_1(s(t)) \\ s_2(t+1) &= f_2(s(t)) \\ &\vdots \\ &\vdots \end{aligned}$$

where $s(t)$ is the vector of state variables at time t .

On entry `s[]` contains $s(t)$. Within the routine the user must update `s[]` so that on exit it contains $s(t+1)$. For example suppose that the you want to code the model:

$$\begin{aligned} s_0(t+1) &= f_0(s_1(t)) \\ s_1(t+1) &= (s_0(t) + s_1(t))P \end{aligned}$$

where f_0 is an unknown function and P an unknown coefficient. Appropriate code would be:

```
y=s[0];          // y is a dummy variable used to store s_0(t)
s[0]=uf(0,s[1]);
s[1]=uc(0)*(y+s[1]);
```

5.4.2 `initst(s,c,t)`

`s` and `c` are pointers to double and `t` is a double.

This routine is used to specify initial conditions for the model. That is the values of the state variables `s[]` at the start of the model iteration. For example, if you have a three state variable model and wanted to set the initial values of the state variables to 1.0, the value of known constant zero and the value of unknown coefficient 1, you would need the code:

```
s[0]=1.0;
s[1]=c[0];
s[2]=uc(1);
```

5.5 `pastvalue(i,t,j)`

This function is called by the user to access history variables at lagged times. `i` is the index of the history variable (starting at zero); `t` is the (double) time at which the history variable is to be evaluated; `j` is the index of the history buffer location marker being used at this call. The function returns a double. Examples of the use of this function are given in the sections on `storehistory()` and `grad()` above, so here a slightly more complicated example is given.

Consider the contrived situation of a population that produces half its offspring in an environment promoting fast maturation and half in an environment promoting slow maturation, suppose also that the maturation time varies throughout the year. Suitable equations might be:

$$\begin{aligned} \frac{ds_0}{dt} &= \frac{1}{2}c_0s_0(t-\tau_0) + \frac{1}{2}c_0s_0(t-\tau_1) - c_1s_0(y) \\ \tau_0 &= 20 + 10 \sin(2\pi(t-90)/365) \\ \tau_1 &= 2\tau_0 \end{aligned}$$

Suitable code for this in `grad()` might be:

```

.
.
T0 = 20.0+10.0*sin(6.28318530718*(t-90.0)/365.0);
T1 = 2*T0;
if (t>T1)
g[0]=pastvalue(0,t-T1,0)*0.5*c[0]+pastvalue(0,t-T2,1)*0.5*c[0]-c[1]*s[0];
else
.
.

```

Note the important point that you can not request lagged variables from before the start of integration - instead you have to define your model in such a way that it does not require such values: this is always possible for a well defined model.

5.6 initfit()

This is a long routine, where you fill out a structure that will control various aspects of fitting. It's pretty straightforward, but also a bit tedious. I'll split the description into parts.

5.6.1 Constants

You start by filling out various model and fit specification constants.

- `d->discrete` must be set to 1 for a discrete time model or 0 for continuous time.
- `d->no_uc` is the number of unknown coefficients (ordinary model parameters) to be fitted.
- `d->no_uf` is the number of unknown functions in the model.
- `d->no_c` is the number of known constants, which are specified in this routine and passed to all model specification routines.
- `d->no_hv` is the number of history variables (lagged variables) - i.e. the number of variables stored in `storehistory()` and retrievable using `pastvalue()`.
- `d->no_s` is the number of state variables `s[]`
- `d->no_fit` is the number of state variables which will actually be fit to data.
- `d->no_sw` is the number of switch variables in the model.
- `d->hbssize` is how many lagged values to save for each history variable.
- `d->nlag` is the number of 'lag pointers' per history variable. This shouldn't be less than 1, and you can always get away with setting it to 1. If the same history variable is accessed at a number of different lags, then efficiency can be improved by giving each lag an index (the last argument of `pastvalue()`) `nlag` is the number of such lags.
- `d->tol` is the integration tolerance, that is the size of error to tolerate in the each state variable at each timestep as a proportion of the magnitude of that state variable.
- `d->t0` integration start time.
- `d->t1` integration stop time when simulating.
- `d->dt` the initial time step, used when integrating the model. The maximum timestep is also set using this. It's currently set to 100 times this value.
- `d->dout` the default output timestep when simulating.

5.6.2 Fitting method and Error model

There are 3 choices of fitting method. All use a quadratic model of the fitting objective for minimisation, and deal with linear constraints in a robust manner, but differ in how the quadratic model is constructed and used:

- 0 A Gauss-Newton method backed up by steepest descent. Gauss-Newton pretends that you have a linear model, and bases the quadratic approximation to the objective on this assumption, thereby neglecting part of the quadratic term that would exist in a full Taylor expansion of the objective. The routine in this case is quite robust, and the steepest descent back-up ensures that it persists until some sort of minimum is reached. The price is that it's the slowest method.
- 1 Quasi-Newton. This method builds up an approximation to the Hessian (second derivative) matrix in the Taylor expansion of the objective. It's quicker than Gauss Newton, and doesn't make any assumptions about linearity of the model. Again a relatively robust implementation is used.
- 2 Iterative least squares. This makes the same assumption as method 0: that the model can be approximated by a linear model. At each iteration this model is minimised subject to constraints by quadratic programming (implemented in a maximally stable manner). The approach is very fast, but omits the steepest descent backup used in option 0. In fact the only stepsize control is successive halving of the step if it fails to improve the objective.

Fitting method is selected by setting `d->fitmethod` to 0, 1 or 2 according to the above list. **Note** that if you are using automatic smoothing parameter selection then this option only determines which method is used to get initial parameter estimates, thereafter an iterative least squares algorithm is used.

`d->bsr_reps` controls bootstrap restarting. This is a method for avoiding local minima that are too shallow to have any statistical meaning. Each restart perturbs the fitting objective by replacing the original data with a non-parametric bootstrap resample from that data. Minimisation then proceeds with that objective. From the minimum of each bootstrap objective one tries to minimise the original objective again. The approach is neat because the bootstrap objectives are not different from the original objective statistically, but will have different local topography, i.e. the perturbation of the objective is what is reasonable given sampling error. Set `d->bsr_reps` to zero for no restarts, otherwise set it to the number of restarts required. This is only available for error model 0, and is not available with auto-selection of smoothing parameters.

The error model specifies the mean variance relationship assumed when fitting, it is specified by setting `d->error` to one of the following.

- 0 Variances independent of mean - e.g. Gaussian model.
- 1 Variance proportional to mean - e.g. Poisson.
- 2 Standard deviation proportional to mean - e.g. Gamma.

options 1 and 2 cause an iterative re-weighting scheme to be used - the same method that's used for maximum likelihood estimation with exponential family error models. Only option 0 works with unknown functions and autoselection of smoothing parameters (at present).

5.6.3 Unknown and known coefficients

You have to supply information about each unknown coefficient:

- `d->uco[i]` the initial value for the *i*th unknown constant.
- `d->cname[i]` the name of the *i*th unknown coefficient to be used when displaying output (optional, but recommended).
- `d->uctype[i]` specifies whether or not the *i*th unknown coefficient is constrained. Valid settings are UNBOUND, B_BELOW, B_ABOVE or the combination B_BELOW | B_ABOVE. There is also a setting FIXED which over-rides any other flags set and fixes a coefficient at the value that it is initialised to or reset to at run time.

- `d->uc1b[i]` lower bound on *i*th u.c. if `B_BELOW` set.
- `d->ucub[i]` upper bound on *i*th u.c. if `B_ABOVE` set.

In addition values of any known coefficients `d->c[i]`, must be supplied here.

5.6.4 Unknown functions

Several bits of information must be supplied for each unknown function:

- `d->ufdf[i]` - the maximum number of degrees of freedom allowed for the *i*th unknown function. This is the number of parameters actually used to specify it, but not the number of degrees of freedom it will eventually use (merely an upper bound on it!).
- `d->uft0[i]` - the lowest value of the argument of the *i*th unknown function.
- `d->uft1[i]` - the highest value of the argument of the *i*th unknown function.
- `d->uftype[i]` contains qualitative information about the *i*th u.f. The following flags are currently implemented:
 1. `B_BELOW` the function is bounded below by `d->uf1b[i]`.
 2. `B_ABOVE` the function is bounded above by `d->ufub[i]`.
 3. `INCREASING` the function is non-decreasing.
 4. `DECREASING` the function is non-increasing.
 5. `ORIGIN` the starting point of the function (`uf(i, d->uft0[i])`) is fixed at whatever you initialise it to in `inuf()`, or at run time.

Any logically consistent combination of these flags can be employed, e.g.

`d->uftype[i]=B_BELOW|INCREASING|B_ABOVE`, specifies that the *i*th u.f. is a monotonically increasing function, bounded above and below.

- `d->ufsp[i]` specifies the smoothing parameter for the *i*th u.f. If any smoothing parameters are set negative, then all smoothing parameters will be selected automatically.
- `d->ufspmax[i]` is the maximum smoothing parameter to use when searching for the s.p.'s. Set to negative value (or don't set) for no upper limit.

Unknown functions are initialized in `inuf()` (see below).

The methods used are quite capable of dealing with functions of more than one variable, but I have yet to implement these in this package.

5.6.5 Data to fit

`d->dfile` should be set to the name of the file of data to be fitted. This file consists of a column of sampling times, followed by columns of observed data at those times. * indicates a missing value. The weights for these values are supplied in `d->wfile`, this contains columns of weights corresponding to the columns of data in the data file, but no column of sample times.

`d->index[i]=j` tells the program that column *i* from the data file corresponds to state variable *j*, where column 0 of the data file is assumed to contain the sampling times. So, for example, `d->index[1]=0` tells the program that the first column of data is to be fitted by state variable 0, similarly `d->index[4]=23` states that `s[23]` will be fitted to the 4th column of data.

`d->rows` is used to specify the number of rows to be read. If it's left undefined or is less than 4 then all rows are read.

5.6.6 Fitting other statistics

It's straightforward to fit to various statistics of the data, rather than directly to the data. Although this idea seemed appealing, I didn't manage to find and statistics that were much use, *except* for one of Peter Turchin's suggestions: matching the ACF. ACF matching is often very good at getting the right frequency and shape for cyclic timeseries, where it otherwise very difficult to get started by trajectory matching.

Here is a list of available statistics (I've tried more, but have removed quite alot of them) All are applied separately to each series:

1. ACF This tries to match the ACF to lag 10, of each series. Mismatch of the ACF is measured by mean square deviation of data ACF from model ACF. For cyclic data, it's often good to start out using this with a pretty high weight, and gradually wind down the weight as you begin to find reasonable starting values for a trajectory match.
2. STDEV standard deviation.
3. ABSOLUTEDEV mean absolute deviation from the mean.
4. MEANGRAD the mean gradient of the series.
5. MEANFREQ mean frequency identified by a fourier method.
6. NO_STATS don't use extra statistics - usually best if you have decent starting values!

The statistics can be used singly or in combination by setting `d->statisticcs`. e.g. `d->statistics=ACF|STDEV` says that the ACF and standard deviation should be matched. The weight to give to matching the statistics is set in `d->wstats[]`. In the above example to set the weight for ACF to 1000.0 and the weight for the standard deviation to 10.0, use the lines `d->wstats[ACF]=1000.0;` and `d->wstats[STDEV]=10.0;`. The weights of statistics are ignored if those statistics are not being used!

5.6.7 Setting up output

In order to be able to keep an eye on the fit as it progresses, and to help find good starting values, graphical output is provided. You have to specify what output you want where.

- `d->no_windows` the number of windows to use for displaying the fit of state variables to data.
- `d->lines[i]` the number of state variables to output in the *i*th window.
- `d->windex[i].win` the window in which the *i*th state variable is to be displayed. You don't have to set this for every state variable. State variables that you ignore won't be plotted.
- `d->windex[i].cur` which number curve in window number `d->windex[i].win` corresponds to the *i*th state variable (if any). Note that if this state variable is being fitted to data, then its data will be plotted as well.
- `d->range[i].y0` the lower bound on the y axis of the *i*th window.
- `d->range[i].y1` the upper bound on the y axis of the *i*th window.
- `d->label[i]` the name of the *i*th state variable, to be used if it is to be plotted in an output window.
- `d->wname[i]` The name of the *i*th window.

5.7 inuf()

`inuf()` is the routine for initializing unknown functions, before fitting and for simulation. Note that nothing is done to ensure that the initialised functions meet any constraints on them, although you'll be warned if they don't. Note also that the u.f.s are represented by splines, and that although the function you use to initialise a u.f. may meet all the constraints specified in `d->uftype[]`, this may not always carry over to the spline that approximates it. Sorry.

The program will call `inuf()` with 2 arguments: `i` the index of the u.f. for which an initialization value is required and `t` the argument of the `i`th u.f. at which the initial value is required. You get to supply the initial values. For example, the following bit of code initialises u.f. 2 to be a quadratic:

```
if (i==2) return(0.2+0.1*t+0.03*t*t);
```

5.8 bootstrapping: inboot()

This routine takes a single argument, the bootstrap control structure. Leave the routine empty if you do not want to bootstrap. It's probably a good idea to leave bootstrapping alone until you are comfortable with model fitting using `DDEFit`.

Bootstrapping is the process of using your data as if it told you everything that you need to know about the population from which it came. There are three flavours:

1. Non-parametric bootstrapping - resample with replacement from the data itself, to simulate the process of going out and collecting the data again. All covariates stay with each resampled datum.
2. Parametric bootstrapping - use the data to estimate all unknown parameters of the population from which the data came: simulate data from this population.
3. Semi-parametric bootstrapping. The 'expected values' of the data are estimated from the fitted model, but to these are added residuals resampled with replacement from the population of residuals.

There are two reasons for wanting to do this:

1. To get confidence intervals for the unknown parameters and/or functions of the model.
2. To compare models for hypothesis testing.

To hypothesis test by model comparison, you usually want to test something like H_0 : The worse fitting model could have generated my data and the other model would still have appeared to do about as much better at fitting as it does with the actual data. vs. H_1 : the better fitting model really is the only one that could have generated the data. `DDEFit` allows such comparisons by a somewhat clunky mechanism: you can bootstrap parametrically or semi-parametrically using one model, saving the replicate data vectors in a file with the replicate measures of fit in another file. These replicates can be read into a later analysis to be fit by another model - the fit measures for the second model can then also be stored in another file.

Parametric bootstrapping uses the fitted values (from model fitting) and adds error to them to generate replicates. The errors are generated from a normal with the population mean and variance estimated from the residuals of the fitted model. As mentioned below, means and variances can be estimated separately for each fitted series. You can also bootstrap semi-parametrically. That is, add to the fitted values residuals resampled from the actual residuals.

In some circumstances different models may be fitted to different numbers of timeseries, this can be dealt with as well: data series that are not common to all models are treated as constraining data for the series that they do fit.

Here are the elements of the control structure:

- `bsc->parametric` - set to 1 for parametric bootstrap 0 for non-parametric, and 2 for semi-parametric.
- `bsc->lumped` - set to 1 if all stages are to be lumped to estimate error variances for parametric bootstrap, set to 0 to do things stage by stage.

- `bsc->iocontrol` 0 for no input or output of replicates. 1 to output replicates to a file. 2 to read replicates from a file.
- `bsc->fdname` - the name for the file to contain replicate data vectors.
- `bsc->fpname` - the name of the file to hold replicate parameter vectors.
- `bsc->SSname` - the name of the file to hold weighted sums of squares used as fit measure, for each replicate.
- `bsc->restarts[0/1/2]` is a 3 element array to contain the number of bootstrap restarts to perform *when bootstrapping*. Element 0 is number of b.s. restarts on first b.s. resample, element 1 is number of b.s. restarts on second b.s. resample and element 2 is number of restarts for all subsequent.
- `bsc->carry_p` should be set to zero if fitting to start from best fit parameters; to 1 if parameters are to be carried from on b.s. replicate to the next; to -1 to always use the initial parameter estimates.
- `bsc->reps` - how many bootstrap replicates to perform.
- `bsc->n_start_files` You can read start parameters from a file: this specifies the number of start files to use.
- `bsc->start_file[]`: array for start file names - you must initialise this, e.g. for a list of 3 files: `bsc->start_file=(char **)calloc(3,sizeof(char*))`; A * in front of the the first file-name indicates that this is a parameter file generated by a previous set of bootstrap runs, with a different set of parameters for each b.s. rep. e.g. `bsc->start_file[0]="*zd0.bsp"`; tells the program that `zd0.bsp` is a text file with one set of starting parameters on each line.

...that covers basic bootstrapping. If different models are to be fitted to different numbers of series then there are some extra options:

1. `bsc->nextra` the number of extra state variables, for insertion into the replicate data vectors. If this is zero then the following entries are not needed.
2. `bsc->extra[j][i]` the *i*th observation for the *j*th extra state variable. The memory allocation for, and filling out of, this array has to be performed by the user.
3. `bsc->exloc[j]` the fitted variable that the *j*th extra variable will correspond to. `bsc->exloc[j]` must be greater than `bsc->exloc[j-1]` for all relevant *j*.
4. `bsc->missing` the double value that will code for a missing value in `bsc->extra[][]`.
5. `bsc->ignore[i]` set to 1 if the *i*th fitted variable is to be ignored when calculating the fit measure for output to file, 0 otherwise.

5.9 Miscellaneous modelling hints

- Don't forget that numerical solution methods are stupid and don't know about magic numbers like zero. For example, even though the exact solution of a set of model equations may be intrinsically positive, this doesn't always mean that the numerical solution of the model will have this property: in such a circumstance you may have to make sure that your code deals with slightly negative values gracefully.
- Similarly, at present, the code does not check that the finite differencing used to estimate derivatives w.r.t. model parameters does not violate inequality constraints. Again in some circumstances this means that your model code will have to deal gracefully with apparently "impossible" values for parameters.
- If you find the way that lagged variables are treated awkward, then you might want to consider saving all state variables as history variables, and then rebuilding the lagged quantities that you are actually interested in from these lagged state variables when you need them (this approach is the one that allows easiest conversion of Solver code, for example).

- The methods are quite fussy about models being well formulated without big discontinuities in derivatives (unless these are dealt with using switches). It's worth being aware that it is quite easy to introduce discontinuities when specifying the start up conditions for a model with delays, and to be careful about the formulation of model start up process.
- You may want to write information to the screen from within your model - the easiest way to do this is to use a Windows "messagebox". Here is an example of a snippet of code that does just that:

```
char str[100];
.
.
sprintf(str,"Total host pop = %g",totpop);
MessageBox(HWND_DESKTOP,str,"Info!",MB_ICONEXCLAMATION|MB_OK);
```

6 Structure of input data files

The structure of these files is covered in `initfit()`, but here it is again. The data file should contain a column of sample times in ascending order plus a column for each series of observed data. The code for a missing observation is `*`. Here is an example file with two state variables:

```
1  3.4  *
2  4.5  0.5
3  5.6  0.6
4  *    0.4
5  *    0.3
6  1.2  0.8
7  1.5  0.7
8  3.1  0.4
```

(in general, of course sample times don't have to be equally spaced like this, unless you are dealing with a discrete time model). The weight file has a weight for each fitted series at each sample time (whether or not the datum is missing. For example the weight file for the above might be:

```
1  4
1  4
1  4
1  4
1  4
1  4
1  4
1  4
1  4
```

if we wanted to give the second series 4 times the weight of the first. (Note that, ideally, weights should be inversely proportional to sampling variance.)

7 Examples

There are a number of examples provided of discrete and continuous time models. This section gives mathematical statements of the coded models.

7.1 `cr.c`: a simple discrete time consumer resource model

This discrete time model describes a situation in which a consumers population growth rate is controlled by food supply f_t , while the food supply is determined by what the consumer has not eaten in the past 2 time periods. N_t

is the consumer population at time t .

$$\begin{aligned} N_{t+1} &= N_t \frac{af_t^m}{b + f_t^m} \\ f_{t+1} &= \alpha + \alpha S(N_t) + \alpha S(N_t)S(N_{t-1}) \\ S(N) &= c_0 + (1 - c_0)e^{-N^2/v} \end{aligned}$$

a, m, b, α, v and c_0 are unknown model coefficients along with $n_0 (= N_0$ and $N_{-1})$ and F_0 . Simulated data files are provided in `cr.dat` and `cr.w`.

7.2 dexp.c: a simple continuous time model with a switch

This is a simple and contrived example to illustrate the use of switches. There is a single state variable n . $n_0 (= n(0))$, b, d and T are all unknown coefficients.

$$\frac{dn}{dt} = \begin{cases} -dn(t) & t < T, t \neq 50 \\ bn(t - T) - dn(t) & t \geq T, t \neq 50 \end{cases}$$

$$n(50^+) = 0.5n(50^-)$$

where the slightly sloppy notations $n(50^+)$ and $n(50^-)$ mean n at the instant just after and just before time 50, respectively. Simulated data files are provided: `dexp.dat` and `dexp.w`.

7.3 znew.c: a discrete time model of Larch Bud Moth at 2 sites.

This is a model of the tri-trophic relationship between Larch Budmoth, its food and its parasitoids at 2 (nested) sites in Europe. Carefully hand simulated data are provided in `zwrong.dat` and `zwrong.w` (these are **not** the real data for this system). Let N_t, P_t and Q_t be host population, parasitoid population and food quality at time t . And let S_t and L_t be the practically measurable variables relating to P_t and Q_t , namely parasitism rate and pine needle length. The basic state equations are:

$$\begin{aligned} N_{t+1} &= N_t \frac{rQ_t}{\delta + Q_t} e^{-N_t/K} S_t \\ P_{t+1} &= \phi N_t (1 - S_t) \\ Q_{t+1} &= (1 - \alpha) \left(1 - \frac{N_t}{\gamma + N_t} \right) + \alpha Q_t \end{aligned}$$

where:

$$S_t = e^{\frac{aP_t}{1 + awP_t + ahN_t}}$$

In fact we can eliminate P_t as a state variable (and scale out ϕ at the same time) to obtain:

$$S_{t+1} = e^{\frac{aN_t(1-S_t)}{1 + awN_t(1-S_t) + ahN_{t+1}}}$$

Finally relating quality to needle length:

$$L_t = b + cQ_t$$

All parameters and initial conditions are unknown. Furthermore it is assumed that K is different at the 2 sites. Note that parasitism data relates only to one site, while needle length data relates only to the other. In this model some scaling of the data has been performed: LBM population density is raised to the power 0.1, while needle length has been divided by 20.

7.4 `cop.c` a continuous time models with unknown functions

`cop.c` contains 2 alternative models of a structured copepod population (select between them by setting `duff` to 0 or 1 near the top of the file). Let $R(t)$ be the recruitment to the first stage, $\mu_1(t)$ be the *per capita* death rate in the first 6 stages and $\mu_2(t)$ be the *per capita* death rate in the remaining stages. Let n_i and τ_i be the population and duration of stage i , respectively. The first (rather silly) model is:

$$\begin{aligned}\frac{dn_i}{dt} &= r_i(t) - m_i(t) - \mu_i(t)n_i(t) \quad \text{for } i = 1, \dots, 6 \\ \frac{dn_i}{dt} &= r_i(t) - m_i(t) - \mu_2(t)n_i(t) \quad \text{for } i = 7, \dots \\ \text{where } r_i(t) &= \begin{cases} R(t) & i = 1 \\ m_{i-1}(t) & \text{otherwise} \end{cases} \\ \text{and } m_i(t) &= n_i(t)/\tau_i\end{aligned}$$

The second model (a Gurney and Nisbet conveyor belt) is:

$$\frac{dn_i}{dt} = r_i(t) - m_i(t) - \delta_i(t)n_i(t)$$

where:

$$\delta_i(t) = \begin{cases} \mu_1(t) & i = 1, \dots, 6 \\ \mu_2(t) & i = 7, \dots \end{cases}$$

$$r_i(t) = m_{i-1}(t) \text{ and}$$

$$m_i(t) = \begin{cases} r_i(t - \tau_i)e^{-\int_{t-\tau_i}^t \delta_i(x)dx} & t > \tau_i \\ f_i(\tau_i - t)e^{-\int_0^t \delta_i(x)dx} & t \leq \tau_i \end{cases}$$

where $f_i(a)$ is the population per unit age interval in stage i at time 0, with argument a being the ‘age in stage’. For this model:

$$f_i(a) = \tilde{R}_i e^{-\delta_i(0)a}$$

where $\tilde{R}_1 = R(0)$ and $\tilde{R}_{i+1} = \tilde{R}_i e^{-\delta_i(0)\tau_i}$ for all $i > 1$.

The unknown functions of the model are each allowed at most 15 degrees of freedom in the coded specification and are all bounded below. Simulated data is provided in `cop.dat` and `cop.w`.

8 Program structure

These are very brief and incomplete notes on code structure. The code evolved over a fairly long period, was “research code” for a long time, and was eventually unscrambled and restructured into the current form. The program is multi-threaded. There is a main thread (code mostly in `w95dde.c`) which does all GUI stuff, handles all user i/o and creates threads for simulation and fitting as required. The simulation and fitting threads do not take input or give output directly to the user - it’s all funnelled through the GUI thread (apart from “stop” signals, which are a special case).

The best way to understand the details of the programs structure and operation is to print out the headers `ddefit95.h` and `w95dde.h`.

Generally all the data required for a simulation or fitting thread to run is passed to it via a pointer to a structure, when the thread is created. Part of this information is the handle of the applications main window (the parent window). When a thread has information to pass to the user, it does so by allocating memory for the information, and posting a message to the main window in the main thread (which it can do since it knows its handle), with a code indicating the type of data, and a pointer to the data. The main thread then deals with this information (itself, or by sending it on to one of the child windows). The main thread frees the memory used for data transfer after

dealing with the data. This structure avoids any possibility of inter-thread memory access conflict problems, so the code does not need to use events, semaphores or critical sections. It should also be painless to convert to a single threaded structure if required.

The GUI thread is programmed in C (rather than C++), but the following general principles have been used. As far as possible all the information required by a window is stored in its own window memory, initialized at start up. The window functions controlling windows avoid using global or static memory, which is what allows multiple copies of the same type of window to operate using the same Window function (the function for the parent window is an exception). Windows may access their children or their parent windows data (but don't go further than that). The only global data shared between threads is the variable used to indicate that the user has pressed the Stop button. The matrix allocation and freeing system uses some global structures to facilitate de-bugging. for this reason it is not used for allocating or freeing the memory associated with structures of type `matrix` within the main GUI thread.

If you look through the source code you will find a fair amount of code that isn't used. Some of this implements methods that are not yet incorporated into the restructured program or are but are not yet well tested enough to be documented: for example gradient matching, random effects process error, simulated annealing, robust smoothing parameter estimation, Quasi-Auto-Regressive fitting, post hoc process error estimation etc. Other unused routines are there because the relevant files are part of general purpose libraries, while others are just dead code, that I've missed.