Solv95: a numerical solver for systems of delay differential equations with switches

Simon N. Wood,
Mathematical Institute, North Haugh, St. Andrews, Fife KY16 9SS, UK.
snw@st-and.ac.uk

Contents

1 Introduction

Solv95 is designed to allow numerical solution of systems of ordinary and delay differential equations, which may have state variable discontinuities (switches). To use it you need a C/C++ compiler/linker capable of producing 32 bit Windows executables with a graphical user interface. Suitable compiler/linkers are supplied commercially by Borland or Microsoft (these come with quite useful debugging facilities), or you can use GNU C/C++ for Windows, which is available without charge (but has slightly less user friendly debugging tools) . To solve a set of equations you must edit a template C file to specify the model. This is then compiled and linked to the numerical and graphical routines which will solve the model and display the results. solv95 was originally designed to solve population dynamic models.

Solv95's advantages are that the numerical analysis is particularly carefully done with respect to solution of delay differential equations, it is quite fast relative to other packages (using efficient adaptive

timestepping), its graphics use only the Windows API (built in to Windows) so you don't need extra toolboxes or add-ins to use it and it's open source. Switches are treated in a less rigorous manner (but I don't know of a package that does any better). Solv95's disadvantages are that it doesn't do anything except solve your equations, it does little to check that the user's code is error free, and some of the Windows programming is inelegant (it was my first attempt at Windows programming and was then ported from Windows 3.1 to OS/2 to Win32).

This manual is not designed to be comprehensive, but is designed to explain how to code up a model and how to compile and link it once you have done so. Under the assumption that most users will start with one of the supplied example models, I will first describe compiling and linking. I will then explain how models are specified in detail and provide examples of how to do so. Finally I will briefly discuss the algorithms used.

The package is provided on condition that the following be accepted: (i) The package comes with no warrenty and remains the property of the author. (ii) The package and compiled models may be re-distributed, provided that no charge is made for doing so, original authorship is fully acknowledged and these conditions are accepted by the recipient.

2 Compiling and Linking models

The package has been successfully used with Borland C/C++, Microsoft Visual C/C++ and GNU gcc ported to Windows. I'll describe how to use `gcc` first.

2.1 Using `gcc`

If you don't already have a port of `gcc` (the gnu compiler suite) for Windows then you need to get it. I used the very excellent `mingw32` (stands for Minimalist GNU Win32) downloaded from:
`ftp://ftp.xraylith.wisc.edu/pub/khan/gnu-win32/mingw32/egcs-1.1.1/`
But for more recent information checkout:
`http://www.xraylith.wisc.edu/~khan/software/gnu-win32/egcs.html`
An excellent tutorial on `gcc` for Windows is to be found at:
`http://www.geocities.com/Tokyo/Towers/6162/gcc.html`.
`mingw32` will give you a suite of command line tools, including a debugger `dbx` and the GNU compiler `gcc`.

Suppose that your model definition file is called `model.c`. To compile and link solv95:

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 solv95.c ddeq.c model.c` this will yield object files `solv95.o, ddeg.o` and `model.o`
3. Compile the windows resources that are needed:
   `windres -i solv95r.rc -o solv95r.o`.
4. Link the object files into an executable:
   `gcc -o solv95.exe ddeq.o solv95.o template.o solv95r.o -mwindows`

Now type `solv95` at the command prompt to run the program.

You may find it convenient to compile `solv95.c`, `ddeq.c` and `solv95r.rc` just once and then combine them into a library file with the command:
`ar -ru solv95.a ddeq.o solv95.o solv95r.o`
Once this has been done the line `gcc -c model.c` compiles the model, while:
`gcc -o solv95.exe model.o solv95.a -mwindows`
will link it (note that the `.a` file must come after the `.a` files in the argument list).

2.2 Borland or Microsoft

I only have access to rather old versions of these products, so these instructions will be quite vague. Assume that your model definition file is called `model.c`. Under both products you need to include `solv95.c`, `ddeq.c`, `solv95r.rc` and `model.c` in a 'project'. You then 'make' or 'build' the project to compile and link and should get an executable which can be run.

3 Defining models

To define a model you need to edit various functions in a standard template file. You may also need to use some standard functions. This section explains these. The following table summarises what you need to know about for each class of models that the package can deal with (SDDE stands for delay differential equations with switches):

Functions to define	ODE	DDE	SDDE
`initcons()`	•	•	•
`switchfunctions()`			•
`map()`			•
`grad()`	•	•	•
`storehistory()`		•	•
`initstate()`	•	•	•
`statescale()`	•	•	•
`initout()`	•	•	•
Functions to use			
`pastvalue()`		•	•
`pastgradient()`		•	•

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.

3.1 state variables and constants

The program aims to solve problems of the general form:

$$\begin{aligned}
\frac{\mathrm{d}s_0}{\mathrm{d}t} &= g_0(\mathbf{s}(t), \mathbf{s}(t-\tau_0), \mathbf{s}(t-\tau_1), \ldots, t) \\
\frac{\mathrm{d}s_1}{\mathrm{d}t} &= g_1(\mathbf{s}(t), \mathbf{s}(t-\tau_0), \mathbf{s}(t-\tau_1), \ldots, t) \\
\frac{\mathrm{d}s_2}{\mathrm{d}t} &= g_2(\mathbf{s}(t), \mathbf{s}(t-\tau_0), \mathbf{s}(t-\tau_1), \ldots, t) \\
\cdot & \quad \cdot \qquad\qquad \cdot \\
\cdot & \quad \cdot \qquad\qquad \cdot
\end{aligned}$$

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 $\mathbf{s}(t)$ is used to mean the vector of all state variables at time t, so $\mathbf{s}(t-\tau)$ is the vector of state variables time τ ago. To define a model the user must specificy 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 user defined constants (as may the initial values) - these constants are passed around in an array `c[]`. All arrays start at element `0`, not element `1`. Elements of `s[]` must not be modified anywhere, except in `map()` and `initstate()`. If you modify elements of `c[]` then the changes will be permanent until you next change them.

3.2 initcons(no_vars,no_cons)

The arguments of this function are both pointers to integers.

The purpose of this function is to specify the number of state variables and number of constants that the problem has. For example if your problem has one state variable and 2 constants, then the function should read:

```
{ *no_cons=2;
  *no_vars=1;
}
```

3.3 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.

3.4 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 `swichfunctions()` 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;}
```

3.5 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:

$$\frac{\mathrm{d}s_0}{\mathrm{d}t} = c_0 s_0 - c_1 s_0 s_1$$

then the corresponding line in the routine `grad()` would be:
`g[0]=c[0]*s[0]-c[1]*s[0]*s[1];`

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

$$\frac{\mathrm{d}s_0}{\mathrm{d}t} = \begin{cases} c_0 s_0(0) & t < c_1 \\ c_0 s_0(t - c_1) & t \geq c_1 \end{cases}$$

subject to the initial condition $s_0(0) = c_2$. 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<c[1]) g[0]=c[0]*c[2];
else g[0]=c[0]*pastvalue(0,t-c[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.

3.6 `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 stored[1]. The lagged values and gradients for each history variable are stored in a ringbuffer for access by `pastvalue()` or occasionally `pastgradient()` (see below).

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

$$\frac{\mathrm{d}s_0}{\mathrm{d}t} = \begin{cases} c_0 s_0(0) & t < c_1 \\ c_0 s_0(t - c_1) & t \geq c_1 \end{cases}$$

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:

[1] This consistency of integrator and interpolator is solv95's chief advantage over other DDE solving packages.

```
if (t<c[1]) g[0]=c[0]*c[2];
else g[0]=c[0]*pastvalue(0,t-c[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 variable).

3.7 initstate(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 3.45, you would use the following code:

```
s[0]=c[3];
s[1]=3.45;
```

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

3.8 statescale(double *scale)

Solv95 controls integration error by specifying that it should be less than some specified proportion 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 proportianal 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.

3.9 initout(usercontrol *out)

This routine is where you provide various initialization information for setting up the model and the display of the model's solutions. The routine is used to fill out the structure `out`: there is a large amount to specify, but it is straightforward to do so. The following sections cover the elements of `out`: any element marked by a * is optional.

3.9.1 Model control constants and defaults

- `out->nhv` the (integer) number of history variables in the model.
- `out->hbsize` the (integer) number of lagged values to store for each history variable, the larger this number, the larger the model time lags can be.
- `out->nlag` this is the (integer) number of distinct lags at which each particular history variable is to be accessed. For example if you want to access history variable 0 at lag $t - 12.0$, and history variable 1 at $t-3.0$, $t-20.0$ and $t-13.2$, then you would set `out->nlag=3`. All `out->nlag` does is to control the number of distinct place markers used in the history buffer for each state variable - the place markers are used to save time when searching for the right place in the history buffer. This feature is purely to allow efficient access of the history buffer - so you can always set `out->nlag=1`, it's just that this may be a bit inefficient.

- `out->nsw` the number of switches (which is also the number of switch functions).
- `out->dout` the output timestep - determines the approximate time between outputting results to windows and any specified file.
- `out->tol` the integration tolerance - that is the maximum error tolerated at each timestep (as a proportion of the state variable concerned).
- `out->dt` the default initial timestep this isn't too crucial, as the timestep will be rejected if it's too big. But note that the maximum timestep is set to $100 \times$ `out->dt` and the minimum to $1 \times 10^{-9} \times$ `out->dt` (I know, I know - you can change it in `dde()` in file ddeq.c.)
- `out->t0` the default integration start time.
- `out->t1` the default integration end time.

3.9.2 constants `c[]`

- `out->c[i]` the default value for the i^{th} constant `c[i]` i (starts at zero).

* `out->cname[i]` name for the i^{th} constant. e.g. to set the name of constant 0 to "fred" use the line `out->cname[0]="fred";`.

* `out->cinfo[i]` extra information about a constant that can be displayed at run time by clicking on the constant's name. e.g.

```
  out->cinfo[0]="This constant measures the rate of accumulation of \
frustration when writing documentation";
```

3.9.3 State variable labels and other information

* `out->label[i]` label for the i^{th} state variable. e.g. `out->label[1]="population of adults"`. It's only worth supplying these for state variables that you intend to output.

* `out->initialtitle` the title to be displayed when solv95 starts.

* `out->initialtext` explanatory text to be displayed when solv95 starts.

3.9.4 Windowed output

Solv95 allows the user to control how many windows output will be displayed in, as well as how many and which state variables will be displayed in each window.

* `out->xlabel` the label to be attached to the 'time' axis of each output plot.

- `out->no_windows` the (integer) number of windows for displaying output.
- `out->lines[i]` the number of state variables to be plotted in the i^{th} window.
- `out->index[i].win` the index of the window in which the i^{th} state variable is to be displayed - only supply this for those state variables that are to be displayed.
- `out->index[i].cur` the index of the curve within a particular window which will display state variable i. e.g. to display state variable 0 as curve 1 of window 2 use the lines:

```
out->index[0].win=2;
out->index[0].cur=1;
```

* `out->wname[i]` the name of the i^{th} window (displayed in the window header bar).

- `out->range[i].y0` the initial minimum y axis value for window i.
- `out->range[i].y1` the initial maximum y axis value for window i.

3.9.5 File output

A text output file can be produced, it's columns are time (or whatever variable is being integrated with respect to) followed by those state variables specified for output.

* `out->fileno` the integer number of state variables to output.
* `out->fout[i]` the index of the state variable to be output in column $i+1$ of the output file (i starts at zero, column zero of the output file is time).

3.10 `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:

$$\begin{aligned} \frac{\mathrm{d}s_0}{\mathrm{d}t} &= \frac{1}{2}c_0 s_0(t-\tau_0) + \frac{1}{2}c_0 s_0(t-\tau_1) - c_1 s_0(y) \\ \tau_0 &= 20 + 10\sin(2\pi(t-90)/365) \\ \tau_1 &= 2\tau_0 \end{aligned}$$

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.

3.11 `pastgradient(i,t,j)`

This is exactly the same as `pastvalue()` except that it returns the time derivative of the lagged variable - its use is somewhat dubious, since the order of approximation of the interpolated gradients is lower than the order of approximation of the integrator which may lead to infelicities in the numerical solution of models using this function.

3.12 Coding hints

- One use of switches, which is not obvious, is to make sure that the integrator "sees" short events - for example, if a population is at equilibrium it is possible for the integrator to be striding along taking such big steps that it steps right over a short pulse of immigrants - to get around this, you either need to modify the maximum timestep (see above) or put a switch in place somewhere during the pulse (which will force the integrator to stop at that point, thereby "noticing" the pulse).

- 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);
```

4 Examples

Solv95 comes with 3 examples model files: `ode_eg.c` is an ordinary differential equation model; `dde_eg.c` is a delay differential equation model; `sdde_eg.c` is a delay differential equation model with switches. The purpose of this section is simply to specify these 3 models mathematically so that, in conjunction with the 3 files, the reader can see how the different types of model are coded up in practice.

4.1 An ODE model

Consider the classic Lokta- Volterra predator- prey model:

$$\begin{aligned} \frac{\mathrm{d}P}{\mathrm{d}t} &= \alpha NP - \delta P \\ \frac{\mathrm{d}N}{\mathrm{d}t} &= \beta N - \gamma NP \end{aligned}$$

where Greek letters are used for model parameters, N is prey population and P is predator population (and typically $\gamma > \alpha$). The file `ode_eg.c` codes up this model defining: `s[0]` $\equiv N$, `s[1]` $\equiv P$, `c[0]` $\equiv \alpha$, `c[1]` $\equiv \delta$, `c[2]` $\equiv \beta$, `c[3]` $\equiv \gamma$, `c[4]` $\equiv N(0)$ and `c[5]` $\equiv P(0)$.

4.2 A DDE model

Consider Gurney and Nisbet's (1981) model of Nicholson's (1954) famous[2] blowflies:

$$\frac{\mathrm{d}A}{\mathrm{d}t} = \begin{cases} -\delta A(t) & t < \tau \\ PA(t-\tau)e^{-A(t-\tau)/A_0} - \delta A(t) & t \geq \tau \end{cases}$$

Where A is the population of adult flies in a laboratory population, τ is the development time from egg to adulthood, P is maximum adult fecundity rate multiplied by the survival rate from egg to adult, and A_0 is a parameter determining how quickly fecundity declines with adult population.

This model is coded in the file `dde_eg.c`.

4.3 A switched DDE model

This model (again vaguely ecological, but very simplistic for clarity) describes a resource replenished at regular intervals, grazed down by a population of consumers. It is assumed that consumers produce offspring in proportion to how much they eat, but that there is a lag between offspring production and those offspring starting to feed. The resources R are governed by:

$$\frac{\mathrm{d}R}{\mathrm{d}t} = -\alpha R(t)C(t)$$

except every T time units, when a quantity R_A is added to R. The consumer equation is:

$$\frac{\mathrm{d}C}{\mathrm{d}t} = \begin{cases} -\delta C(t) & t < \tau \\ \gamma C(t-\tau)R(t-\tau) - \delta C(t) & t \geq \tau \end{cases}$$

[2] If you're an ecologist!

where γ is the consumer fecundity per unit of resource (over α, strictly), τ is the development time and δ the *per capita* death rate.

The file `sdde_eg.c` codes up this model, with the assumption $C(0) = 1$ and $R(0) = 0$.

5 The algorithm

The method used for integration is an embedded RK2(3) scheme due to Fehlberg, and reported on page 170 of Hairer *at al.* (1987). Lagged variables (and gradients) are stored in a ringbuffer at each step of the integrator. Interpolation is required to estimate values of the lagged variables between storage times. For numerical probity it is essential that the interpolation of lagged variables is of a higher order of approximation than the integrator, otherwise the assumptions underlying the error estimate from the RK pair will not be met. The algorithm used in Solv95 uses cubic hermite interpolation (e.g. Burden and Faires 1987) to achieve this (which is the reason that gradients need to be stored along with lagged values). The consequences of not using consistent interpolation and integration schemes are vividly illustrated in Highman (1993). Paul (1992) was also influential in the design of the method used here, and the step size selection is straight out of Press *et al.* (1992) (method, not code!). The RK2(3) pair used is not actually optimal - it should be possible to derive an improved scheme - see Butcher (1987) for an explanation of how to go about it.

A deficiency of the algorithm is that switches are not tracked automatically - if a switch impacts on a lagged variable the resulting derivative discontinuities are only dealt with by adaptive timestepping through them (Note, however, that lagged variables and gradients are stored immediately before and immediately after the application of a switch - i.e. the interpolator is not applied blindly through switches!).

Burden, R.L. and J.D. Faires (1985) Numerical Analysis. Pridle Weber and Schmidt, Boston.

Butcher, J.C. (1987) The Numerical Analysis of Ordinary Differential Equations. John Wiley & sons, Chichester.

Hairer, E., S.P.Norsett & G.Wanner (1987) Solving Ordinary differential Equations I. Springer-Verlag Berlin. p170 RKF2(3)B

Highman, D.J. (1993) Appl. Numer. Math. 12:403-414

Paul, C.A.H (1992) Appl. Numer. Math. 9:403-414

Press *et al.* (1992) Numerical Recipes in C. CUP