dde: A Package for Solving Delay Differential Equations

A J F Buckner

August 21, 2008

Contents

Chapter 1

Introduction

1.1 The dde package

dde is a package for solving ordinary (ODE) or delay (DDE) differential equations. It allows you to specify an ODE or DDE model in terms of a simple text-based language. You can use the programme to solve the system directly, or to generate C or Javascript code that can then be run independently of dde. It is based on the solv95 package written by Simon Wood: the numerical DDE solver is closely derived from Simon's, however the way that dde is used is completely different. Parts of this manual have been lifted from Simon's documentation for solv95.

dde is written in Objective-C and uses Apple's Foundation classes. It was written and compiled on an Apple Macintosh computer, but it may be possible to compile and use it under Gnustep on Linux. I haven't tried.

1.2 Obtaining dde

You can download both the source and binaries for dde from the web-site http://www.ashley.buckner.co.uk/Software/.

1.3 A Simple Model

dde expects models to be defined in a text file. You have to edit it using a text editor such as vi, emacs or even textedit. Don't use a word processor like Pages as it will stick lots of special formatting characters that will confuse dde no end. I use Smultron. dde is case-sensitive, so you *must* type `param` *not* `PARAM` or `Param`.

This is the DDE example model from the solv95 package written in dde. Note that the syntax is fairly close to such packages as ode and xpp. The

model itself can be written:

$$\begin{aligned} \frac{\mathrm{d}A}{\mathrm{d}t} &= PA_\tau \exp\left[-\left(A_\tau/A_0\right)\right] - \delta A \\ A(0) &= N_0 \\ A_\tau &= \begin{cases} A(t-\tau) & \text{if } t > \tau \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

This model becomes

```
option tstart = 0.0, tstop = 300, outstep = .1

param P = 10.0 label="Product of max. fecundity and juvenile survival."
param tau = 12.0 label="Development time."
param delta = 0.25 label="Per capita death rate"
param A0 = 300 label="Fecundity decay constant."
param N0 = 100.0 label="Initial population."

var A = N0 delay scale=0.0
aux Alag

Alag = A(t - tau)
A'= P*Alag*exp(-Alag/A0) - delta*A

print t, A, A(t - tau)
```

Here, `A` is the dependent variable of the model (the number of adult blowflies) whose initial value is `N0` ; `P = 10`, `tau = 12`, `delta = 0.25`, `A0 = 300` and `N0 = 100` are parameters; `Alag` is an auxiliary variable. The option statement requests output every 0.1 time units, with integration proceeding from 0 to 300 time units. The results are shown in figure 5.1

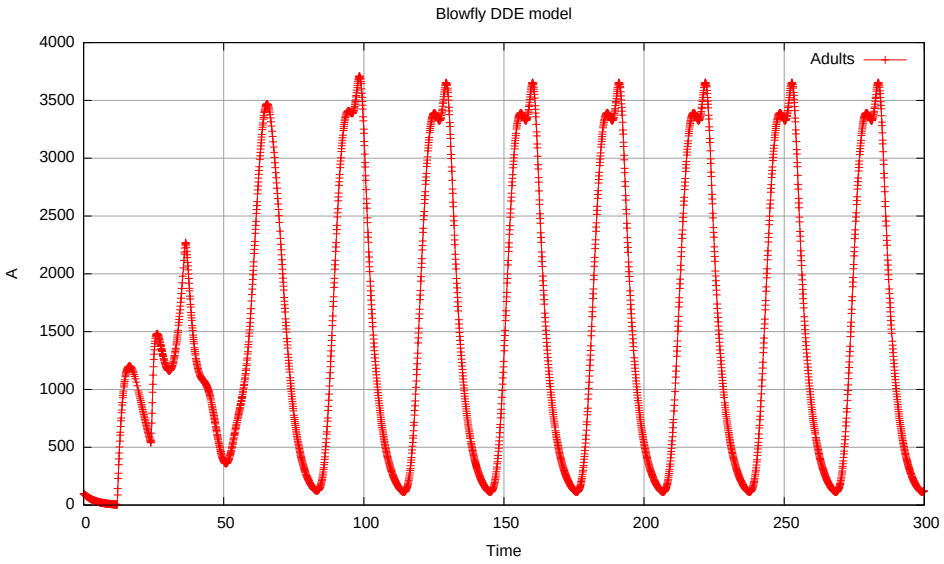

Figure 1.1: Numerical solution of the blowfly model

Chapter 2

Basic Usage

2.1 Defining models

A model in ddecan be defined as a system of equations of the form:

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

These define the dynamics of the model. In addition, dde allows you to define:

- State variables and their intial conditions...
- Parameters that control the particular behaviour of the model. These are constant across te lifetime of the model, but default values can be defined which can then be changed by the user on a run-by-run basis. These may also have labels attached to them;
- Auxiliary variables...
- Values to print out;
- States of the model where discontinuites in the state variables occur (called switches);
- Functions of a single variables defined as a number of (x,y) pairs: in keeping with system dynamics terminology, these are called table functions;

- Options to control the programme, such as the start and stop values of the independent value ('time'), how often to ouput results and the dde23 integrator tolerance.

2.2 Statements, Expressions and Names

A dde model is composed of a sequence of statements. A statement must end with a semicolon or a line-end. Some statements are evaluated once to initialise parameters and set initial values of state variables, others are evaluated every time the dde package needs to compute the avlues of the right-handsides of your model equations. Apart from splitting statements into initialisers and dynamic statements, dde does *not* sort equations: they are executed in the order you specify. This means that variables must be defined before they are used.

To tell dde that it is expecting a comment, start a line with either '//' or '#'. dde will ignore the rest of the line.

The syntax for expressions in dde follows the C programming language. The four arithmetic operators have their customary meanings, and the `^` operator can be used for exponentation: `x^y` returns `x` to the power of `y`. dde also includes the 'ternary' operator `choice?value1:value2`: if choice is true then return value1 else value2.

In addition, dde gives the following mathematical functions: `abs`, `sqrt` (square root), `int` (convert to integer), `float` (convert to floating point), `exp`, `log`, `ln`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `erf`, `xidz` and `zidz`. Most of these should be self-explanatory. The two functions `xidz` and `zidz` are useful for avoiding divide by zero errors: they both take two arguments x and y, if y is non-zero then the result is x/y; if y is zero then `xidz` returns x whereas `zidz` returns 0.

Names in dde follow the usual conventions: they must start with a letter or the '_' character. Subsequent characters must be letters, digits or '_'. All variables and parameters *must* be declared before they are used.

2.3 State variables

The syntax for defining a state variable is:

```
state x[=expr1] [label="Label"] [delay] [scale=expr2]
```

Text in square brackets ([]) is optional. The text `state` may be replaced by `var`. If you omit the initial value of a state variable in its declaration, you must specify it later on:

```
state x
...
x = 1000
```

`expr` is a valid `dde` expression. It is the initial value of the state variable. Variables may be given and (optional) text label to documents them. If a state variable is to be a delay variable use the 'delay' option. To evaluate the past value of a delayed variable, the syntax is:

```
x(tdelay[, initvalue[, lagmarker]])
```

The lagged time at which the value of should be returned is tdelay. If tdelay is less than the initial time value then the value `initvalue` is returned: if `initvalue` is not specified then the default initial value is 0. The use of the third argument `lagmarker` is fuly described in the solv95 manual.

Once you have declared a state variable, you need to define its rate of change with an assignemt of the '= form, e.g.

```
x' = (x0 - x)/timedelay;
```

2.4 Parameters

Parameters are values which are defined at the start of the solution of a model, and thereafter remain the same. The syntax for declaring a parameter is

```
parameter x[=expr] [label="Label"]
```

An alternative is to use `param` as an abbreviation for `parameter`. The only option is an explanatory label.

2.5 Auxiliaries

Auxiliary variables are variables that are evaluated afresh every time the right-handsides of the model equations are evaluated. They must be declared before they can be used.

```
auxiliary a[=expr]
```

`auxiliary` may be abbreviated `aux`. Auxiliaries may be redefined:

```
auxiliary a=1.0
...
a = 2*a
```

2.6 Switches

To set up a switch:

```
on expr do
    statement 1
    ...
    statement n
end
```

The value of expr is checked at each integration step. When it passes from positive to negative, the statements between `do` and `end` are executed in the order specified. The statements must all be assignments: you can change the values of state variables and parameters. It does not make sense to redefine auxiliaries as these will be re-computed at the next step anyway.

Switches are a feature borrowed directly from solv95. Sometimes, it is handy to be able to describe very fast processes that are effectively discontinuous. For example, the rate of change of the value of my bank balance is equal to my salary minus the rate at which I spend money. One way to represent this in a model would be

```
balance' = salary - expenditure
```

On a weekly scale, say, expenditure could be represented as a continuous process. However, my salary is paid in monthly. Another way to represent this would be to add my salary to my balance at the start of every month using a switch

```
on -sin(2*3.14159265358*t) do
    balance = balance  + salary
end
...
balance' = -expenditure
```

2.7 Table Functions

Sometimes it is handy to have a more flexible way of defining functions rather than just in terms of the standard mathematical functions that dde provides. It is nice to be able to 'draw' a function. dde allows to specify a function by telling it the values it takes a specified number of points. the syntax is either:

```
table tab=INTERPNAME(\{x1, y1\}, ..., \{xn, yn\})
```

where the x's and y's are expressions (including constant values) to define the points the curve must pass through or

```
table tab=regular INTERPNAME(y1, ...yn) [xmin=expr1] [xmax=expr2]
```

If all the x's are equally spaced. the expressions `expr1` and `expr2` optionally define the lowest and highest x's.

The values of INTERPNAME defines method used to interpolate values between x's. It can be

- **stepwise**: the function has a staircase-like shape
- **linear**: the function is piecewise linear between given x's
- **spline**: the function is a smooth function between giveb x's with continuous first and second derivatives.

Table functions are evaluated as a normal function:

```
table f = spline({0.00, 0.00}, {0.1, 0.305},  {0.2, 0.545},
        {0.3, 0.72}, {0.4, 0.835},  {0.5, 0.905},
        {0.6, 0.945},  {0.7, 0.97},  {0.8, 0.985},
        {0.9, 1.00}, {1, 1.00})
var x = 1.0

x'= -f(x)
```

The arguments to the table function definition need not be constants, although all table functions are defined when the model is initialised, so they may not contain auxiliaries.

```
table f = regular spline(a0, a1, 2*a3)
```

2.8 Output

Results from a model can be printed to the programme's standard output using the print statement:

```
print header="Header Text" footer="Footer Text" leader="Leader Text"
      trailer="Trailer Text" separator="Seperator" expr1, ..., exprn
```

The header is printed before any output, the trailer afterwards. Each line starts with an optional leader and ends with an optional trailaer. Output items are separated by separator.

As an example, this prints out an HTML table.

```
print header="Content-type: text/html\n\n<html><body>\n<table>"
          separator="</td><td>"
          leader="<tr><td>"
          trailer="</td></tr>"
          footer="</table>\n</body></html>"
          t, x
```

2.9 Options

The option statement allows you to change numerical and other parameters:

```
option opt = val, ..., opt = val
```

where values of opt may be

- tstop: the upper end of the integration interval
- tstep: the lower end of the integration interval
- epsilon: the (relative) integrator tolerance
- outstep: the average interval between output
- histsize: the size of the history buffer

`val` may be a numerical or string constant — if the latter then it must be enclosed in quotation marks. For example:

```
option tstart=1970, tstop=1986, outstep=0.5
```

Chapter 3

Advanced Usage

3.1 Arrays

It is often useful to be able to disaggregate variables: for example, . Another case is where you wish to solve a problem for several different values of the same parameter and plot all results together. dde allows variables to be *array*: this equivalent to putting a subscript on variables.

Arrays have dimensions. dde allows two sorts of dimensions, *ordered* dimensions where the order is important (e.g. layers of soil) with numerical subscript and *unordered* dimensions where order is unimportant (e.g. regions in country). You first have to declare the possible array dimensions:

```
dimension layer = 0:10
dimension region = {southeast, soutwest, northeast, northwest,
   westmids, eastmids, easteng, yorkshumb,
   london, scotland, wales}
```

Here, `layer` is an ordered dimension, `region` is unordered. You can use `dim` as an abbreviation for `dimension`.

Having defined dimensions, you can use them to refer to dimensions of variables:

```
state water[layer] = {200, 100, 100, 100, 100, 100, 100, 100, 100, 100.0}
...
state population[region] = 10000
...
auxiliary migration[southeast] = 0
...
state' = births - deaths + migration
...
water[0]' = -evaporation
water[1]' = diffusion[1]
...
```

If you omit any dimensions from an array variable, it is assumed that you want all possible subscripts...

Array variables may be multiple-dimensional:

```
state Nitrogen[region,layer] = 0
```

Here, all elements of `Nitrogen` are set to zero.

3.2 Control statements

`dde` only has one control statement, `if...then...else...end`. The syntax is:

```
if condition then
   statement1
   ...
   statementn
[else
   statement1
   ...
   statementn]
end
```

The `then` part is optional. Don't overdo use of `if...then` statements.

Related to the `if...then` statement is the `while...do` statement.

```
while condition do
   statement1
   ...
   statementn
end
```

It keeps executing the block statements as long as the condition is true.

The `for` loop structure can be used to execute a block of code a predetermined number of times. It has number of forms:

```
for indexvbl in dimension do
   statement1
   ...
   statementn
end

for indexvbl from expr1 to expr2 do
   statement1
   ...
   statementn
end
```

In each case `indexvbl` is a special sort of auxiliary variable that can only take on integer values. It has to be declared, thus:

```
index variable
```

The first form of the `for` loop is useful for unordered dimensions, the second for ordered dimensions.

3.3 User-defined Function

You can define your own functions, thus:

```
function name(arg1, ..., argn)
   statement1
   ...
   statementn
end
```

It is only worth defining functions if they are going to be used for more than one set of arguments. The result of the last statement is returned as the result of the function. For example:

```
function hyperb(x)
(x + sqrt(1 - x*x))/2
end
```

Chapter 4

Running dde

4.1 Invoking the programme

The usage for `dde` is:

```
dde -v -g -Dparam=value -E -f outputType -s sourceType
   file1.dde ... filen.dde
```

The `-v` flag tells `dde` to produce verbose output; the `-g` flag turns on debugging features.

The `-D` flag allows you to over-ride parameter definitions from the command line: for example, `-Dtau=7.5` sets the parameter `tau` (if it exists) to 7.5.

The `-f` flag NOT YET IMPLEMENTED instructs `dde` to convert the model into a file of type outputType, where outputType may be

- ddesolve: a C file for use with DDESolve or ddecgi.
- solv95: a C file for use with `solv95`.
- js: a JavaScript file that can be run with SpiderMonkey (REF)
- asp: a JavaScript file that can be run with Sun's ASP

The `-s` flag over-rides the input source type. NOT YET IMPLEMENTED.

The `-E` flag NOT YET IMPLEMENTED instructs `dde` to look for an environment variable called `QUERY_STRING`: if it can be found, `dde` attempts to parse it to look for parameter values which then over-ride any previous definitions (including via `-D` flags). This allows you to use `dde` in CGI applications.

All output is sent to the terminal, including output from `print` statements

4.2 Run files

NOT YET IMPLEMENTED.

4.3 When Things Go wrong...

To be done...

Chapter 5

Example Models

5.1 Blowfly DDE Model

```
option tstart = 0.0, tstop = 300, outstep = .1

param P = 10.0 label="Product of max. fecundity and juvenile survival."
param tau = 12.0 label="Development time."
param delta = 0.25 label="Per capita death rate"
param A0 = 300 label="Fecundity decay constant."
param N0 = 100.0 label="Initial population."

var A = N0 delay scale=0.0
aux Alag

Alag = A(t - tau)
A'= P*Alag*exp(-Alag/A0) - delta*A

print t, A, A(t - tau)
```

5.2 Simple Lotka-Voltera Predator-Prey ODE Model

```
param alpha = 0.005
param delta = 0.2
param beta = 1.0
param gamma = 0.02
param N0 = 100
param P0 = 100

state N = N0
state P = P0
```

```
N' = beta*N - gamma*N*P
P' = alpha*N*P - delta*P

print t, N, P

option tstart=0, tstop=300
```

5.3 Kaibab Plateau Model

```
param Area = 800000
param Food_max = 4.8e008
param Food_per_deer_normal = 1000

table Deer_fractional_increase_f = spline({0,-0.5}, {0.4,-0.15}, {0.8,0},
        {1.2,0.1}, {1.6,0.16}, {2,0.2})
table Deer_kills_p_predator_f = spline({0,0}, {0.005,5}, {0.01,10},
        {0.015,14}, {0.02,17.5}, {0.025,20})
table Food_consumed_p_deer_f = spline({0,0.5}, {0.1,450}, {0.2,810}, {0.3,1140},
        {0.4,1340}, {0.5,1540}, {0.6,1670}, {0.7,1790},
{0.8,1880}, {0.9,1960}, {1,2000}, {1.1,2000}, {1.2,2000})
table Food_regeneration_time_f = spline({0,35}, {0.25,15}, {0.5,5},
        {0.75,1.5}, {1,1})
table Fraction_harvest_p_yr_f = linear({1900,0}, {1905,0}, {1910,0.2}, {1915,0.2},
        {1920,0.2}, {1925,0.2}, {1930,0.2}, {1935,0.2},
        {1940,0.2}, {1945,0.2}, {1950,0.2})
table Predator_fractional_increase_f = spline({0,-0.2}, {5,0}, {10,0.08},
        {15,0.14}, {20,0.18})

state Deer = 4000
state Food = Area*590
state Predators = 160

aux Time = t

aux Deer_density = Deer/Area
aux Deer_kills_per_predator = Deer_kills_p_predator_f(Deer_density)
aux Deer_predation_rate = Predators*Deer_kills_per_predator

aux Food_per_deer_ratio = (Food/Deer)/(Food_per_deer_normal)
aux Deer_fractional_increase = Deer_fractional_increase_f(Food_per_deer_ratio)
aux Deer_net_increase = Deer*Deer_fractional_increase

aux Fraction_harvested_p_yr = Fraction_harvest_p_yr_f(Time)
```

```
aux Food_consumed_p_deer = Food_consumed_p_deer_f(Food_per_deer_ratio)
aux Food_consumption_rate = Deer*Food_consumed_p_deer

aux Food_regeneration_time = Food_regeneration_time_f(Food/Food_max)
aux Food_regeneration_rate = (Food_max - Food)/Food_regeneration_time

aux Predator_fractional_increase = Predator_fractional_increase_f(Deer_kills_per_pre
aux Predator_net_increase = Predators*Predator_fractional_increase
aux Predator_harvest = Predators*Fraction_harvested_p_yr

Deer'= Deer_net_increase - Deer_predation_rate
Food'= Food_regeneration_rate - Food_consumption_rate
Predators'= Predator_net_increase - Predator_harvest

print t, Deer, Food, Predators

option tstart=1900, tstop=1950
```

5.4 Rinaldi's Model of Love Dynamics

```
state L=0, P=0, Z=0

param alpha1 = 3
param alpha2 = 1
param alpha3 = 0.1

param beta1 = 1
param beta2 = 5
param beta3 = 10

param gamma = 1
param delta = 1
param A_L = 2
param A_P = -1

L'= -alpha1*L + beta1*(P*(1 - (P/gamma)^2) + A_P)
P'= -alpha2*P + beta2*(L + A_L/(1 + delta*Z))
Z'= -alpha3*Z + beta3*P

print t, L, P, Z

option tstart=0, tstop=20, outstep=0.1
```

5.5 Delay logistic model

```
dimension problem = {ex3a, ex3b, ex3c, ex3d}

param lambda[problem] = {1.5, 2.0, 2.5, 3.0}

state y[problem] = 0.0 delay

aux ylag[problem] = (t>1)?y[problem](t-1):t-1
y' = -lambda*ylag*(1 + y)

print t, y
```

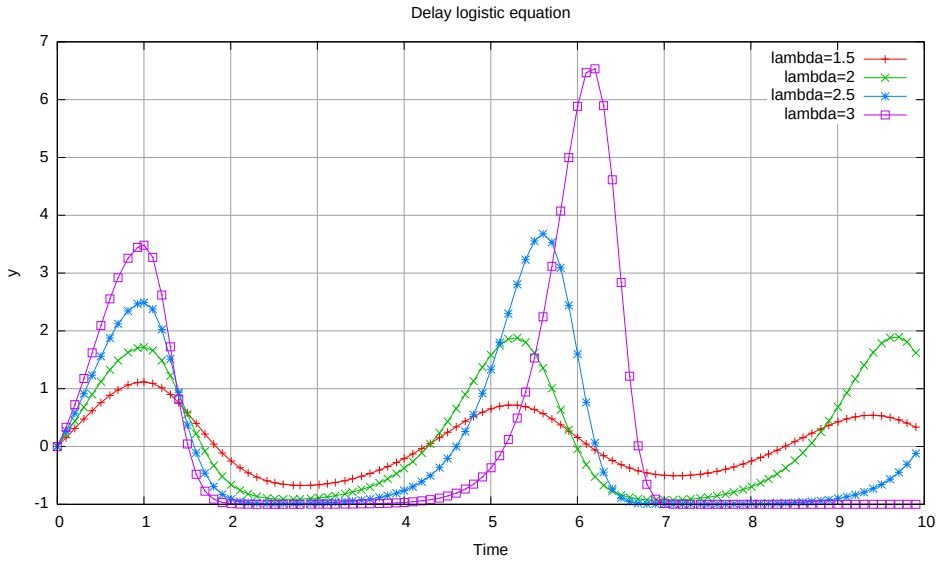

Figure 5.1: Numerical solution of the delay logistic model