A User's Guide to the R library "ddesolve"
Version 1.00 – July 17, 2007

Jon Schnute, Alex Couture-Beil, and Rowan Haigh

1. Introduction

The R library **ddesolve** generates numerical solutions for systems of delay differential equations (DDEs) and ordinary differential equations (ODEs). The numerical routines come from Simon Wood's program `solve95` (http://www.maths.bath.ac.uk/~sw283/simon/dde.html, file: `solv95.zip`), originally written in C for the Microsoft Windows operating systems. With **ddesolve**, a user can write the gradient code for a system of DDEs or ODEs in the R language, rather than C. The code will then run on all platforms supported by R, and the results can be inspected using R's extensive graphics capabilities. Simon has very generously given us permission to publish **ddesolve** (including his embedded routines) under the GNU GENERAL PUBLIC LICENSE Version 2.

For more information about Simon and his work at the University of Bath (Bath, United Kingdom), see his home page http://www.maths.bath.ac.uk/~sw283/index.html. He has recently published a book about Generalized Additive Models (GAMs; Wood 2006) with two supporting R libraries **gamair** and **mgcv**, both available on the Comprehensive R Archive Network (CRAN, http://cran.r-project.org/). The first example in his book presents data from the Hubble Space Telescope and an analysis from Hubble's law that suggests the universe is about 13 billion years old. If this example piques your interest, install **gamair** from CRAN, and run the R code:

```
require(gamair)
data(hubble)
plot(hubble)
```

to see the distance x and velocity y relative to the earth for each of 24 galaxies. The relationship $y = \beta x$ determines Hubble's constant $\beta = 1/A$, where A is the age of the universe. See Simon's book for further details about this interesting estimation problem.

We originally noticed the `solv95` program in the context of a Python implementation (`PyDDE`, http://seis.bris.ac.uk/~bzzbjc/python/PyDDE/) by Ben Cairns at the School of Biological Sciences, University of Bristol (Bristol, UK). For more information about Ben, see his website at http://seis.bris.ac.uk/~bzzbjc/. We have designed **ddesolve** from `solv95` to perform similarly to the earlier R library **odesolve**, written by R. Woodrow Setzer with Fortran algorithms (notably `lsoda`) by Linda Petzold and Alan Hindmarsh at the Lawrence Livermore National Laboratory in Livermore, California.

The history of **ddesolve** illustrates the advantages of open source software. Jon (author JTS above) wanted to use R to solve delay differential equations. He knew about another implementation in the commercial package Matlab® (http://www.mathworks.com/), based on the function `dde23` (Shampine and Thomson 2000). Our programming wizard Alex (author ACB) looked at the code and thought it might be tricky to implement in R, partly because `dde23` was in Fortran and we knew more about the interface between R and C. Then Alex discovered `PyDDE` and `solv95`, and he obtained Simon's permission (and encouragement) to implement it in R. Based on his experience with another package **PBSmodelling** (Schnute et al. 2006), Alex quickly altered calls in the `solv95` C code to get values of the gradient from an R function. When Ben tried our initial R library, he liked it, but he wanted to obtain solutions at specified times, rather than the slightly irregular times generated by Simon's code. So Ben changed the C source code, using Simon's interpolation algorithms to get interpolated values at definite times. As a final touch, Alex implemented another feature of Simon's code called "switches", discussed below.

The demos included with **ddesolve** require **PBSmodelling** version 1.50 or later (http://cran.r-project.org/src/contrib/Descriptions/PBSmodelling.html). Although the numerical routines in **ddesolve** work without this extra package, **PBSmodelling** adds user interfaces that make it easier to experience and understand the operation of **ddesolve**.

2. Defining DDEs

To define a system of DDEs, a user must supply an R function that calculates the gradient of each variable in the system with respect to time. This gradient function must have one of the following two forms:

`gradfunc(t,y)` **or** `gradfunc(t,y,parms)`

where `t` = the current time of integration;
`y` = a vector of estimated state values at time `t`;
`parms` = an optional R object (such as a vector, list, or data frame) of additional input parameters for the DDE system.

The length n of the vector $\mathtt{y} = (y_1, \ldots, y_n)$ corresponds to the number of states in the system, where $n \geq 1$.

The function `gradfunc` must calculate the derivative dy_i / dt for each variable y_i ($i = 1, \ldots n$) and return the derivative values in one of the following two formats:

(1) a vector of n derivatives dy_i / dt, or

(2) a list in which the first element comprises a vector of n derivatives, and the second element comprises a numeric vector with additional values (of interest to the user) calculated within `gradfunc` at time `t`.

Consistent with the idea of *delay* differential equations, `gradfunc` can also depend on state values and their derivatives at times prior to the current time t. These must be accessed with calls to `pastvalue()` and `pastgradient()`. Both functions take a single argument, a time t_{lag} in the range $t_0 \leq t_{\text{lag}} < t$, where t_0 is the starting time of integration and t is the current time. The functions return a numeric vector of length *n*, where `pastvalue(tlag)` and `pastgradient(tlag)`have the components $y_i(t_{\text{lag}})$ and $dy_i(t_{\text{lag}})/dt$, respectively, for $i = 1, \ldots, n$. Usually, $t_{\text{lag}} = t - k$ is calculated as a fixed offset *k* back from the current time *t*; and a typical call might have the form `pastvalue(t-k)`. The calculation of `gradfunc` can involve numerous past values and gradients at various different time lags.

Wood (1999) also introduced the concept of *switches* that allow the DDE system to produce discontinuous changes in the state vector *y*. To implement *k* switches (i.e., *k* conditions in which the state vector can be discontinuous) with $k \geq 1$, a user needs to define two functions:

`switchfunc(t,y)` **or** `switchfunc(t,y,parms)`, which returns a numeric vector of length *k*; and

`mapfunc(t,y,sid)` **or** `mapfunc(t,y,sid,parms)`, which depends on a switch id number ($1 \leq$ `sid` $\leq k$) and returns a numeric vector of length *n*.

These functions specify, respectively, the circumstances that trigger a switch and the behaviour of the system when a switch occurs. Think of `switchfunc` defined by a vector of *k* functions $s_j(t, y)$ with $j = 1, \ldots, k$. Switch *j* takes place when s_j vanishes due to a change from positive to negative values (mathematically $s_j = 0$ and $\partial s_j / \partial t < 0$, where the symbol ∂ denotes partial differentiation). At a time *t* when switch *j* is triggered, `dde` automatically calls `mapfunc` with `sid` = *j*. Our "ice cream" demo in Section 4 below illustrates the process of writing code that includes switches.

3. Solving DDEs

Simon Wood's (1999) numerical routines produce the core functionality of **ddesolve**. The function

```
dde(y, times, func, parms=NULL, switchfunc=NULL, mapfunc=NULL,
     tol=1e-08, dt=0.1, hbsize=10000)
```

invokes the C routines used to numerically solve systems of DDEs, where

`y`	=	a vector of initial values for the states (this also determines *n*);
`times`	=	a numeric vector of explicit times at which the solution should be obtained;
`func`	=	a gradient function written to the specifications of `gradfunc` in Section 2;
`parms`	=	an optional vector of parameters to pass to `func`;
`switchfunc`	=	an optional function that determines conditions when the DDE system experiences switches, as describe in Section 2;
`mapfunc`	=	an optional function associated with `switchfunc` that describes how the DDE system changes (possibly discontinuously) at switch times;
`tol`	=	a scalar that sets the maximum error tolerated in the solution;
`dt`	=	the maximum initial time step used in constructing the numerical solution;
`hbsize`	=	history buffer size required for retaining lagged state variable values;

For consistency with the package **odesolve**, the argument name `func` corresponds to the system gradient function.

The return value of `dde` depends on the output format of the gradient, i.e., options (1) or (2) for the output of `gradfunc` in Section 2. With format (1), `dde` returns a data frame with $n+1$ columns and default column names `t`, `y1`, `y2`,..., `yn`. The first column represents the times at which the solution is reported (`times` , plus any additional times specified by `switchfunc`) and the remaining *n* columns contain the components of `y` estimated at these times. The default column names for these *n* columns are overridden by `names(y)` if the initial vector `y` has a `names` attribute. If `gradfunc` has format (2), then the data frame returned by `dde` is extend by an additional *m* columns, where *m* is the length of the vector of additional values reported by `gradfunc`. By default, these have column names `extra1`, `extra2`,..., `extram`. The names can be overridden by assigning a `names` attribute to the vector of additional information returned by `gradfunc`, i.e., the second component of the `list` output from `gradfunc`.

In summary, an application of **ddesolve** can include three user-defined functions:

- `myGrad(t,y)` or `myGrad(t,y,parms)`,
- `mySwitch(t,y)` or `mySwitch(t,y,parms)`,
- `myMap(t,y,sid)` or `myMap(t,y,sid,parms)`,

as well as a call to `dde`:

-
  ```
  dde(y, times, func=myGrad, parms, switchfunc=mySwitch,
     mapfunc=myMap, tol=1e-08, dt=0.1, hbsize=10000)
  ```

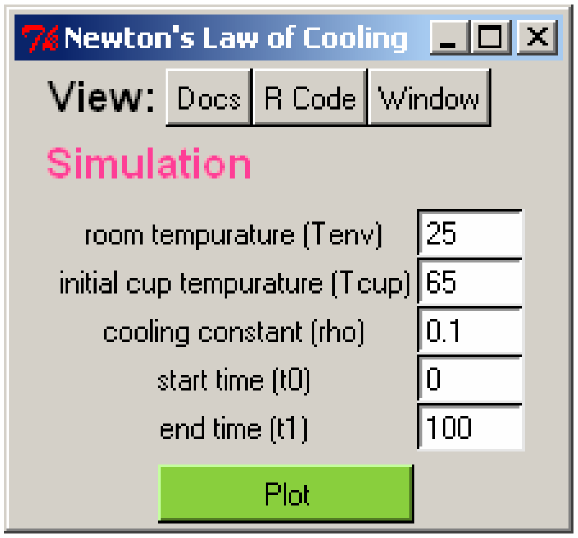
The gradient function must be defined, but the switch and map functions are optional (either both or neither). Similarly, the code may or may not involve an R object `parms` of parameters kept constant during the integration. The gradient function can call the predefined functions `pastvalue(tlag)` and `pastgradient(tlag)` to obtain lagged values of the state variables and their derivatives.

Remember that the gradient, switch, and map functions are called internally by `dde`; consequently, the argument list must precisely correspond to one of the prototypes listed at the start of the previous paragraph. Values of the arguments, `t`, `y`, `sid`, and `parms` will be set by `dde` when these functions are called. By default, `parms=NULL`, so that the user's functions should not depend on `parms` unless a value of `parms` is explicitly specified in the call to `dde`. Typically, `parms` might be a vector or list with named components.

4. Demos

The **ddesolve** library currently includes four demos that illustrate simple applications. As mentioned in Section 1, these require the R library **PBSmodelling** (version 1.50 or later) to create graphical user interfaces (GUIs) that aid model testing and exploration. Once **PBSmodelling** is installed and loaded with `require(PBSmodelling)`, call the function `runDemos()`, select **ddesolve**, and then choose one of the available demos. Alternatively, run R's native `demo()` function in lieu of `runDemos()`.

4.1. Cooling - Newton's Law of Cooling (ODE Example)



room tempurature (Tenv) 25
initial cup tempurature (Tcup) 65
cooling constant (rho) 0.1
start time (t0) 0
end time (t1) 100

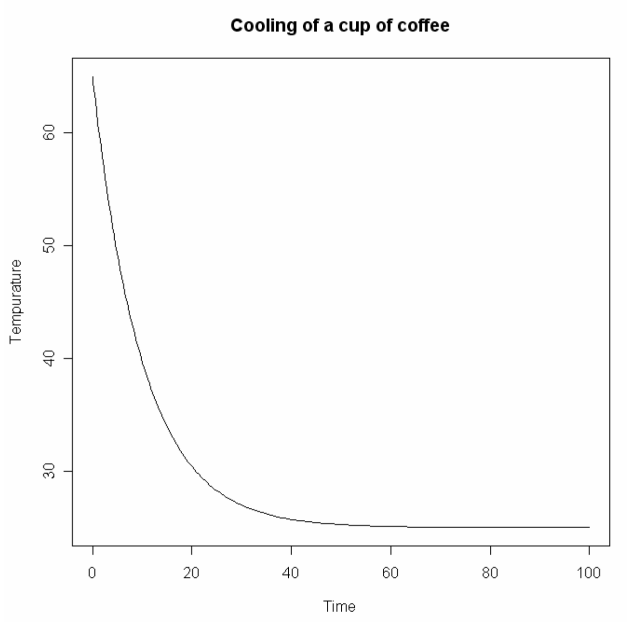


Figure 1. Newton's Law of Cooling demonstration.

This demo illustrates how to set up and solve a single ODE with **ddesolve**. For historical background, see http://en.wikipedia.org/wiki/Heat_conduction#Newton.27s_law_of_cooling. Imagine a hot cup of coffee that cools toward room temperature, where a constant ρ determines the rate of cooling. Newton's Law of Cooling suggests a simple differential equation to determine the coffee temperature $y(t)$ at time *t*:

$$\frac{dy}{dt} = -\rho\,(y - T_{\text{env}}),$$

where T_{env} is the ambient room temperature. If $y(0) = T_{\text{cup}}$ denotes the initial temperature of the coffee, then this equation has the analytical solution

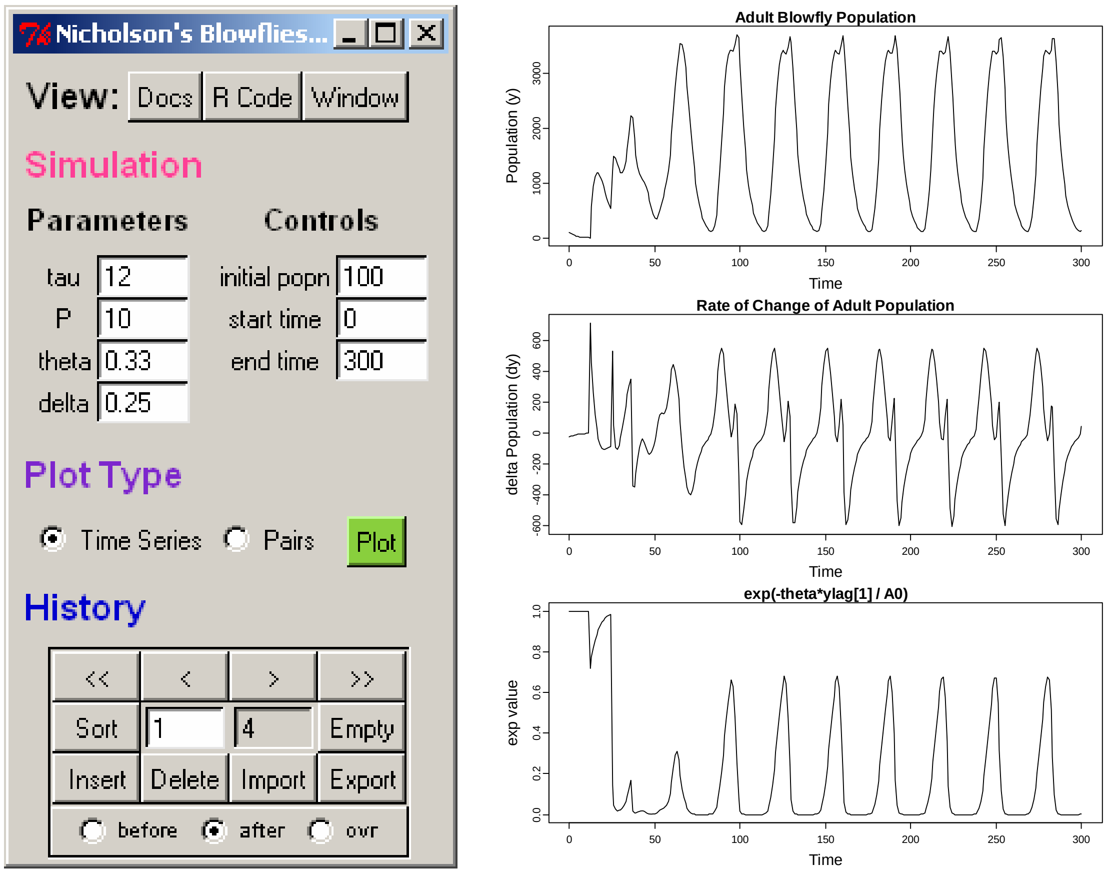
$$y(t) = T_{\text{env}} + \left(T_{\text{cup}} - T_{\text{env}}\right) e^{-\rho t},$$

where $y(t) = T_{\text{cup}}$ when $t = 0$ and $y(t) \to T_{\text{env}}$ as $t \to \infty$. The GUI in Figure 1 displays the code when you press the "R Code" button, as long as R-files (`*.r`) are associated with a suitable text editor on your system. Similarly, "Docs" displays documentation and "Window" displays the script used to produce the GUI. In this example, two key lines of the code are:

```
myGrad <- function(t, y) {return( -rho*(y[1]-Tenv)}
dde(y=Tcup, func=myGrad, times=seq(t0,t1,length=100), hbsize=0)
```

The parameters `rho`, `Tenv`, `Tcup`, `t0` (the start time), and `t1` (the end time) come from the GUI. This ordinary differential equation does not need a history buffer, so `hbsize=0`.

4.2 Blowflies – (DDE Example)



tau 12
P 10
theta 0.33
delta 0.25
initial popn 100
start time 0
end time 300

[✓ Time Series] Pairs

1 4

before [✓ after] ovr

Figure 2. Nicholson's blowflies model demonstration (included in Simon Wood's Solv95 User Manual as an example of solving a DDE).

As an example with a delay, Wood (1999) suggested a blowfly population model for adults $A(t)$ at time *t*:

$$\frac{dA}{dt} = \begin{cases} -\delta A(t), & t < t_0 + \tau; \\ PA(t-\tau)e^{-\theta A(t-\tau)/A_0} - \delta A(t), & t \geq t_0 + \tau; \end{cases},$$

$$A(t_0) = A_0 .$$

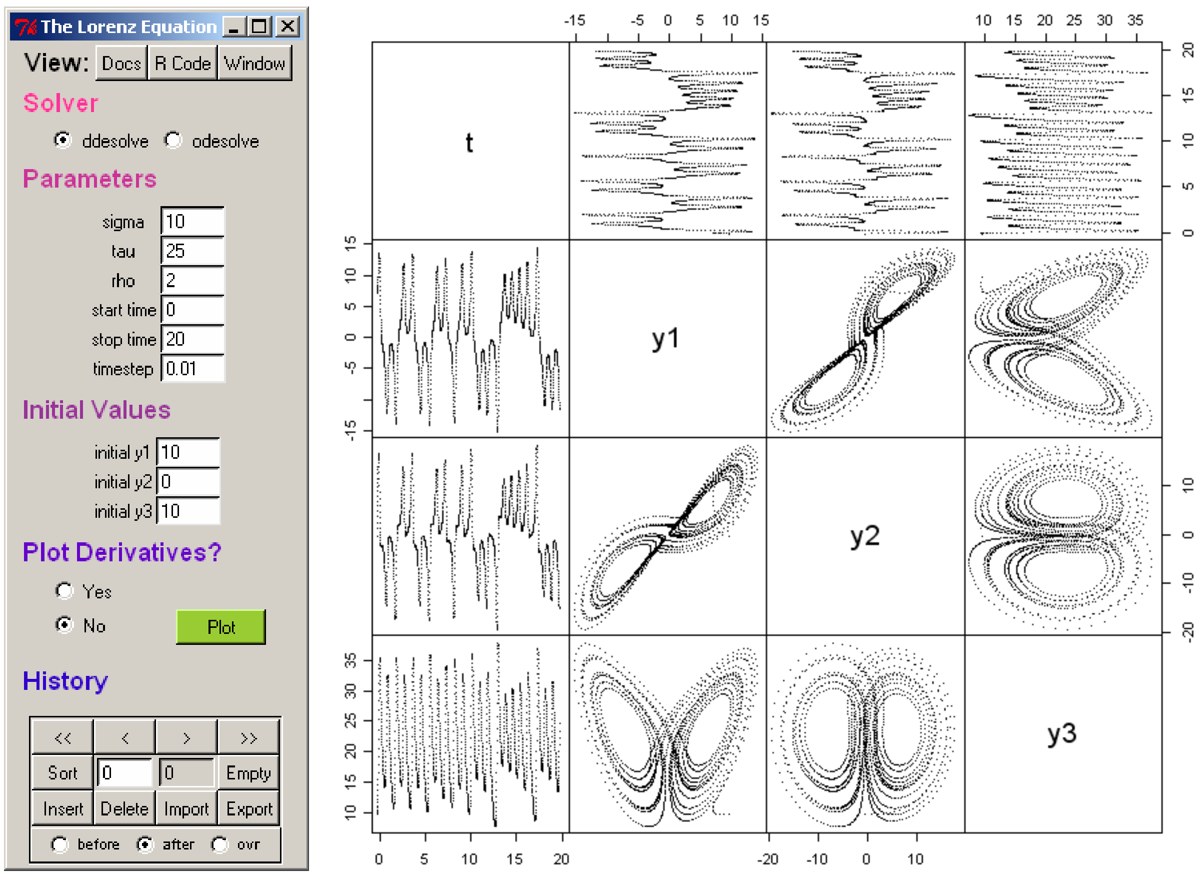
Here τ is the development time from egg to adult, *P* is the net production rate determined by adult fecundity and egg survival to adulthood, θ is a parameter determining how quickly fecundity declines with an increasing adult population, δ is the adult death rate, and t_0 is the initial time when $A(t)$ starts with the value A_0. We assume that $A(t) = 0$ for $t < t_0$. In our formulation, the differential equation also includes the parameter A_0, so that θ becomes dimensionless. Essentially, A_0 sets the scale for $A(t)$.

The GUI in Figure 2 allows the four parameters $(\tau, P, \theta, \delta)$ to be adjusted, along with the initial conditions (t_0, A_0) and the final time t_1. The graph at the left shows three panels: $A(t)$, $dA(t)/dt$, and $e^{-\theta A(t-\tau)/A_0}$. In this case, a key portion of the R code is:

```
myGrad <- function(t, y) {
  if (t-t0 >= tau) ylag <- pastvalue(t-tau)
  else ylag <- 0
  yexp <- exp(-theta*ylag[1]/A0)-delta*y[1]
  yp <- P*ylag[1]*yexp
  return( list(yp, c(dy=yp, exp=yexp)) ) }
```

where values of `tau`, `P`, `theta`, `delta`, `t0`, and `A0` come from the GUI.

4.3 Lorenz – (ODE Example)



Solver

[✓ ddesolve] odesolve

Parameters

sigma	10
tau	25
rho	2
start time	0
stop time	20
timestep	0.01

Initial Values

initial y1	10
initial y2	0
initial y3	10

Plot Derivatives?

Yes [✓ No]

History

0 0

before [✓ after] ovr

Figure 3. The Lorenz model demonstrates chaotic behaviour in the solution of thee linked differential equations.

The Lorenz model (http://planetmath.org/encyclopedia/LorenzEquation.html) consists of three ordinary differential equations for a three-dimensional state vector *y*:

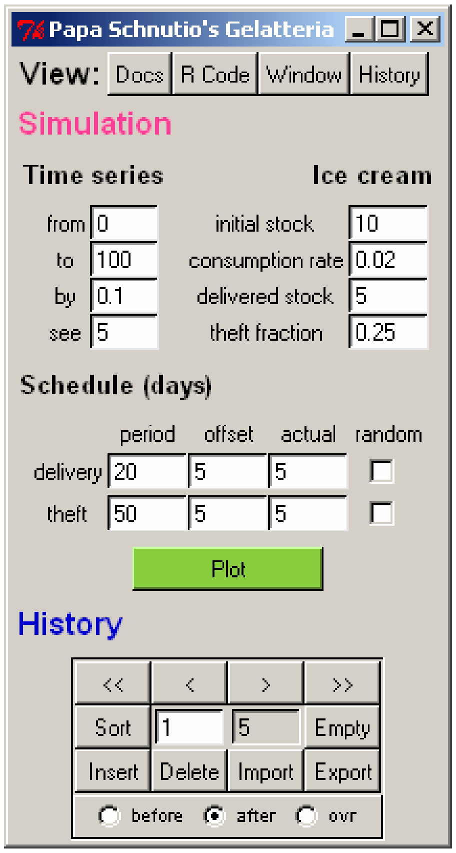
$$\frac{dy_1}{dt} = \sigma(y_2 - y_1),$$

$$\frac{dy_2}{dt} = y_1(\tau - y_3) - y_2,$$

$$\frac{dy_3}{dt} = y_1 y_2 - \rho y_3,$$

with three parameters (σ, τ, ρ). This demonstration includes a GUI for adjusting the parameters and initial conditions to see results from integrating the Lorentz model. It also allows the solution to be obtained with either **ddesolve** or **odesolve**. The choice of numerical solver should not affect the results of the plot, even though these two packages use different underlying algorithms for estimating the solution. Tests indicate that both solvers return comparable results, a result that gives us some confidence that **ddesolve** performs correctly.

4.4 Ice Cream Parlor – Raiders of the Lost Cone (Switches)


Papa Schnutio's Gelatteria
View: Docs R Code Window History
Plot
<< < > >>
Sort Empty
Insert Delete Import Export

Simulation

Time series		Ice cream	
from	0	initial stock	10
to	100	consumption rate	0.02
by	0.1	delivered stock	5
see	5	theft fraction	0.25

Schedule (days)

	period	offset	actual	random
delivery	20	5	5	☐
theft	50	5	5	☐

History

1 5

before [✓ after] ovr

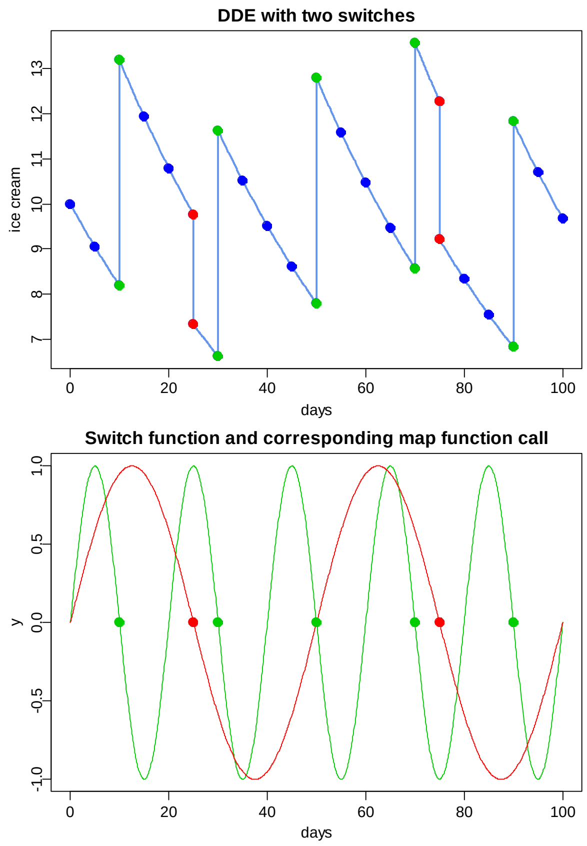


Figure 4. The ice cream parlor receives deliveries (switch 1, green) and experiences theft (switch 2, red).

To illustrate switches, Alex (author ACB) suggested an ice cream parlor that gets restocked periodically. But Jon (author JTS) wanted at least two switches, so he suggested that thieves might occasionally raid the parlor and steal some of the stock. Comments in Alex's code soon suggested a snappy title: *Raiders of the Lost Cone*. Rowan (author RH) and Alex quickly agreed that the parlor should have a flashing sign with the logo "Papa Schnutio's" that puts an Italian twist on Jon's Germanic last name. Jon hesitated, but he once lived in Italy for a year and couldn't resist the idea of Italian ice cream (*gelato*). So he agreed to an establishment named

Papa Schnutio's Gelatteria

Our model assumes an exponential depletion of the ice cream stock $y(t)$ with rate *r*:

$$\frac{dy}{dt} = -ry\,,$$

perhaps because a lower stock would offer fewer choices and thus discourage consumption. (Sometimes a *gelatteria* just doesn't have that perfect flavour you came for. It was there yesterday, but not today – a great disappointment!) We need two switch functions that we chose to be the sinusoids

$$s_i = \sin\left[2\pi\left(a_i + \frac{t}{p_i}\right)\right]$$

with offset parameters a_i and periods p_i for $i = 1, 2$. Switch 1 triggers restocking

$$y(t_+) = y(t) + Y$$

and switch 2 triggers theft events

$$y(t_+) = (1 - f)\,y(t)\,,$$

where $y(t_+)$ denotes the value of y after the switch, *Y* is the fixed amount of ice cream added to the stock, and *f* is the fraction of the stock removed by thieves.

In this example, key features of the code involve the sine wave used for switching

```
sinWave <- function(t,aa,pp) { sin( 2*pi*(aa + (t/pp)) ) }
```

the switch function

```
mySwitch <- function(t,y) {
  c( sinWave(t,a[1],p[1]), sinWave(t,a[2],p[2]) ) }
```

the map function

```
myMap <- function(t,y,swID) {
  if (swID==1) y <- y + Y else y <- (1-f)*y }
```

the gradient function

```
myGrad <- function(t,y) { -r*y }
```

and the call to the main routine

```
yout <- dde(y=y0, times=tt, func=myGrad,
  switchfunc=mySwitch, mapfunc=myMap)
```

where the initial stock `y0`, the desired output times `tt`, the consumption rate `r`, the amount `Y` brought by the supplier, the theft fraction `f`, the offset vector `a` (of length 2), and the period vector `p` (of length 2) correspond to values prescribed by the GUI.

4.5 Fish Population with a Fishery and a Reserve

Our motivation to produce **ddesolve** came primarily from biological models of fish populations that experience recruitment from larval production at an earlier time. (The blowflies example in Section 4.2 illustrates similar behaviour.) The package **PBSmodelling** includes a much more elaborate example (with complete documentation) in which a reserve from fishing protects a portion of the fish population. The model appears in two versions, with discrete and continuous time *t*. This example requires at least versions 1.50 and 1.00 of **PBSmodelling** and **ddeslove**, respectively. After these two packages have been installed, type the following commands in the R console:

```
require(PBSmodelling)
runExamples()
```

In the GUI to “Choose an Example”, press the radio button for the simulation “FishRes”. View the manual by pressing the “Docs” button, as in the examples discussed above in Sections 4.1-4.4.

5. The Algorithm

The R library **ddesolve** provides an interface to Simon Wood’s numerical routines found in `solv95`. The algorithm implemented in this package is the same as that in `solv95`, and is described by Wood (1999) in his user manual:

> “The method used for integration is an embedded RK2(3) scheme due to Fehlberg, and reported on page 170 of Hairer *et al.* (1987). Lagged variables (and gradients) are stored in a ring buffer 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.”

The original `solv95` software requires a user to write C code for a system of DDEs. This must be compiled and linked with `solv95`; then the resulting executable file gives a numerical solution. With **ddesolve**, a user codes the model in R, rather than C. A compiled version of the integration algorithm automatically comes with the library, which makes the numerical C routines compatible with R. Because the output appears as an object in R, a user can interpret the results using R's extensive capabilities for analysis and graphics.

The numerical routines have been preserved in the files `ddeq.c` and `ddeq.h`. The interface to `dde()` has been significantly altered and now appears in the file `ddesolve95.c`, which replaces `solv95.c`. The link between R and C is contained in `r_model.c`, adapted from a basic model template in the original `solv95` code bundle. This file now has many calls to the R application programming interface (API).

6. References

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

Butcher, J.C. 1987. The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods. John Wiley & Sons, Inc. 528 pp.

Hairer, E., Norsett, S.P., and Wanner, G. 1987. Solving Ordinary Differential Equations I. Springer-Verlag Berlin. 170 pp. RKF2(3)B.

Higman, D.J. 1993. Error control for initial value problems with discontinuities and delays. Applied Numerical Mathematics 12(4): 315-330.

Paul, C.A.H. 1992. Developing a delay differential equation solver. Applied Numerical Mathematics 9: 403-414.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. 1992. Numerical Recipes in C : The Art of Scientific Computing. Cambridge University Press.

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2006. PBS Modelling 1: user's guide. Canadian Technical Report of Fisheries and Aquatic Sciences 2674: viii + 112 pp. URL: http://cran.r-project.org/src/contrib/Descriptions/PBSmodelling.html

Shampine, L.F., and Thompson, S. 2000. Solving delay differential equations with dde23. Tutorial: http://www.radford.edu/~thompson/webddes/tutorial.html

Wood, S.N. 1999. Solv95: a numerical solver for systems of delay differential equations with switches. Saint Andrews, UK. 10 pp.
URL: http://www.maths.bath.ac.uk/~sw283/simon/dde.html, file: `solv95.zip`
(See `solv95-Manual.pdf` in the root library directory for **ddesolve**.)

Wood, S.N. 2006. Generalized Additive Models: An Introduction with R. Chapman & Hall/CRC. 416 pp.

Appendix A. R-manual for ddesolve

This appendix documents the objects (functions) available in **ddesolve**. Subsequent pages give indexed technical documentation for every object generated from `*.Rd` files written for the R documentation system. The package **PBSmodelling** includes a directory called `PBStools\` that contains useful batch files for building R packages, including the creation of the indexed manual included here.

Package ‘ddesolve’

July 17, 2007

Version 1.00

Date 2007-07-17

Title Solver for Delay Differential Equations

Author Alex Couture-Beil <alex@mofo.ca>, Jon T. Schnute <SchnuteJ@pac.dfo-mpo.gc.ca>, Rowan Haigh <HaighR@pac.dfo-mpo.gc.ca>

Maintainer Alex Couture-Beil <alex@mofo.ca>

Depends R (>= 2.4.1)

Description This package solves systems of delay differential equations. by interfacing numerical routines written by Simon N. Wood <s.wood _at_ bath.ac.uk>, with contributions by Benjamin J. Cairns <ben.cairns@bristol.ac.uk>. These numerical routines first appeared in Simon Wood's solv95 program.

R topics documented:

`dde` *Solve Delay Differential Equations*

Description

A solver for systems of delay differential equations based off numerical routines from Simon Wood's *solv95* program. This solver is also capable of solving systems of ordinary differential equations.

Please see the included demos for examples of how to use `dde`. To view available demos run `demo(package="ddesolve")`. The supplied demos require that the package **PBSmodelling** be installed.

Usage

```
dde(y, times, func, parms=NULL, switchfunc=NULL, mapfunc=NULL,
    tol=1e-08, dt=0.1, hbsize=10000)
```

Arguments

`y`	vector of initial values of the DDE system. The size of the supplied vector determines the number of variables in the system.
`times`	numeric vector of specific times to solve.
`func`	a user supplied function that computes the gradients in the DDE system at time `t`. The function must be defined using the arguments: `(t,y)` or `(t,y,parms)`, where `t` is the current time in the integration, `y` is a vector of the current estimated variables of the DDE system, and `parms` is any R object representing additional parameters (optional). The argument `func` must return one of the two following return types: 1) a vector containing the calculated gradients for each variable; or 2) a list with two elements - the first a vector of calculated gradients, the second a vector (possibly named) of values for a variable specified by the user at each point in the integration.
`parms`	any constant parameters to pass to `func`, `switchfunc`, and `mapfunc`.
`switchfunc`	an optional function that is used to manipulate state values at given times. The switch function takes the arguments `(t,y)` or `(t,y,parms)` and must return a numeric vector. The size of the vector determines the number of switches used by the model. As values of `switchfunc` pass through zero (from positive to negative), a corresponding call to `mapfunc` is made, which can then modify any state value.
`mapfunc`	if `switchfunc` is defined, then a map function must also be supplied with arguments `(t,y,switch_id)` or `t,y,switch_id,parms)`, where `t` is the time, `y` are the current state values, `switch_id` is the index of the triggered switch, and `parms` are additional constant parameters.
`tol`	maximum error tolerated at each time step (as a proportion of the state variable concerned)
`dt`	maximum initial time step
`hbsize`	history buffer size required for solving DDEs)

Details

The user supplied function `func` can access past values (lags) of `y` by calling the `pastvalue` function. Past gradients are accessible by the `pastgradient` function. These functions can only be called from `func` and can only be passed values of `t` greater or equal to the start time, but less than the current time of the integration point. For example, calling `pastvalue(t)` is not allowed, since these values are the current values which are passed in as `y`.

Value

A data frame with one column for `t`, a column for every variable in the system, and a column for every additional value that may (or may not) have been returned by `func` in the second element of the list.

If the initial `y` values parameter was named, then the solved values column will use the same names. Otherwise `y1`, `y2`, ... will be used.

If `func` returned a list, with a named vector as the second element, then those names will be used as the column names. If the vector was not named, then `extra1`, `extra2`, ... will be used.

See Also

`pastvalue`

Examples

```
##################################################
# This is just a single example of using dde.
# For more examples see demo(package="ddesolve")
# the demos require the package PBSmodelling
##################################################

#create a func to return dde gradient
require(ddesolve)
yprime <- function(t,y,parms) {
        if (t < parms$tau)
                lag <- parms$initial
        else
                lag <- pastvalue(t - parms$tau)
        y1 <- parms$a * y[1] - (y[1]^3/3) + parms$m * (lag[1] - y[1])
        y2 <- y[1] - y[2]
        return(c(y1,y2))
}

#define initial values and parameters
yinit <- c(1,1)
parms <- list(tau=3, a=2, m=-10, initial=yinit)

# solve the dde system
yout <- dde(y=yinit,times=seq(0,30,0.1),func=yprime,parms=parms)

# and display the results
plot(yout$t, yout$y1, type="l", col="red", xlab="t", ylab="y",
     ylim=c(min(yout$y1, yout$y2), max(yout$y1, yout$y2)))
lines(yout$t, yout$y2, col="blue")
legend("topleft", legend = c("y1", "y2"),lwd=2, lty = 1,
       xjust = 1, yjust = 1, col = c("red","blue"))
```

`pastvalue` *Retrieve Past Values (lags) During Gradient Calculation*

Description

These routines provides access to variable history at lagged times. The lagged time t must not be less than t_0, nor should it be greater than the current time of gradient calculation. The routine cannot be directly called by a user, and will only work during the integration process as triggered by the `dde` routine.

Usage

```
pastvalue(t)
pastgradient(t)
```

Arguments

`t`	access history at time `t`.

Value

vector of variable history at time `t`.

See Also

`dde`

Index