Package deSolve: Solving Initial Value Differential Equations in R

Karline Soetaert
Centre for
Estuarine and Marine Ecology
Netherlands Institute of Ecology
The Netherlands

Thomas Petzoldt
Technische Universität
Dresden
Germany

R. Woodrow Setzer
National Center for
Computational Toxicology
US Environmental Protection Agency

Abstract

R package **deSolve** (Soetaert, Petzoldt, and Setzer 2010a,b) the successor of R package **odesolve** is a package to solve initial value problems (IVP) of:

- ordinary differential equations (ODE),
- differential algebraic equations (DAE) and
- partial differential equations (PDE).
- delay differential equations (DeDE).

The implementation includes stiff integration routines based on the **ODEPACK** FORTRAN codes (Hindmarsh 1983). It also includes fixed and adaptive time-step explicit Runge-Kutta solvers and the Euler method (Press, Teukolsky, Vetterling, and Flannery 1992), and the implicit Runge-Kutta method RADAU (Hairer and Wanner 2010).

In this vignette we outline how to implement differential equations as R -functions. Another vignette ("compiledCode") (Soetaert, Petzoldt, and Setzer 2008), deals with differential equations implemented in lower-level languages such as FORTRAN, C, or C++, which are compiled into a dynamically linked library (DLL) and loaded into R (R Development Core Team 2008).

Keywords: differential equations, ordinary differential equations, differential algebraic equations, partial differential equations, initial value problems, R.

1. A simple ODE: chaos in the atmosphere

The Lorenz equations (Lorenz, 1963) were the first chaotic dynamic system to be described. They consist of three differential equations that were assumed to represent idealized behavior of the earth's atmosphere. We use this model to demonstrate how to implement and solve differential equations in R. The Lorenz model describes the dynamics of three state variables, X, Y and Z. The model equations are:

$$\frac{dX}{dt} = a \cdot X + Y \cdot Z$$
$$\frac{dY}{dt} = b \cdot (Y - Z)$$
$$\frac{dZ}{dt} = -X \cdot Y + c \cdot Y - Z$$

with the initial conditions:

$$X(0) = Y(0) = Z(0) = 1$$

Where a, b and c are three parameters, with values of -8/3, -10 and 28 respectively.

Implementation of an IVP ODE in R can be separated in two parts: the model specification and the model application. Model specification consists of:

- Defining model parameters and their values,
- Defining model state variables and their initial conditions,
- Implementing the model equations that calculate the rate of change (e.g. dX/dt) of the state variables.

The model application consists of:

- Specification of the time at which model output is wanted,
- Integration of the model equations (uses R-functions from **deSolve**),
- Plotting of model results.

Below, we discuss the R-code for the Lorenz model.

1.1. Model specification

Model parameters

There are three model parameters: a, b, and c that are defined first. Parameters are stored as a vector with assigned names and values:

```
> parameters <- c(a = -8/3,
+                 b = -10,
+                 c =  28)
```

State variables

The three state variables are also created as a vector, and their initial values given:

```
> state <- c(X = 1,
+            Y = 1,
+            Z = 1)
```

Model equations

The model equations are specified in a function (`Lorenz`) that calculates the rate of change of the state variables. Input to the function is the model time (`t`, not used here, but required by the calling routine), and the values of the state variables (`state`) and the parameters, in that order. This function will be called by the R routine that solves the differential equations (here we use `ode`, see below).

The code is most readable if we can address the parameters and state variables by their names. As both parameters and state variables are 'vectors', they are converted into a list. The statement `with(as.list(c(state,parameters)), ...)` then makes available the names of this list.

The main part of the model calculates the rate of change of the state variables. At the end of the function, these rates of change are returned, packed as a list. Note that it is necessary to return the rate of change in the same ordering as the specification of the state variables (this is very important). In this case, as state variables are specified X first, then Y and Z, the rates of changes are returned as dX, dY, dZ.

```
> Lorenz<-function(t, state, parameters) {
+   with(as.list(c(state, parameters)),{
+     # rate of change
+     dX <- a*X + Y*Z
+     dY <- b * (Y-Z)
+     dZ <- -X*Y + c*Y - Z
+
+     # return the rate of change
+     list(c(dX, dY, dZ))
+   })   # end with(as.list ...
+ }
```

1.2. Model application

Time specification

We run the model for 100 days, and give output at 0.01 daily intervals. R's function `seq()` creates the time sequence:

```
> times     <-seq(0,100,by=0.01)
```

Model integration

The model is solved using **deSolve** function `ode`, which is the default integration routine. Function `ode` takes as input, a.o. the state variable vector (`y`), the times at which output is required (`times`), the model function that returns the rate of change (`func`) and the parameter vector (`parms`).

Function `ode` returns an object of class `deSolve` with a matrix that contains the values of the state variables (columns) at the requested output times.

```
> require(deSolve)
> out <- ode(y = state, times = times, func = Lorenz, parms = parameters)
> head(out)
```

```
     time         X        Y        Z
[1,] 0.00 1.0000000 1.000000 1.000000
[2,] 0.01 0.9848912 1.012567 1.259918
[3,] 0.02 0.9731148 1.048823 1.523999
[4,] 0.03 0.9651593 1.107207 1.798314
[5,] 0.04 0.9617377 1.186866 2.088545
[6,] 0.05 0.9638068 1.287555 2.400161
```

Plotting results

Finally, the model output is plotted. We use the plot method designed for objects of class `deSolve`, which will neatly arrange the figures in two rows and two columns; before plotting, the size of the outer upper margin (the third margin) is increased (`oma`), such as to allow writing a figure heading (`mtext`). First all model variables are plotted versus `time`, and finally `Z` versus `X`:

```
> par(oma = c(0, 0, 3, 0))
> plot(out, type = "l", xlab = "time", ylab = "-")
> plot(out[, "X"], out[, "Z"], pch = ".")
> mtext(outer = TRUE, side = 3, "Lorenz model", cex = 1.5)
```

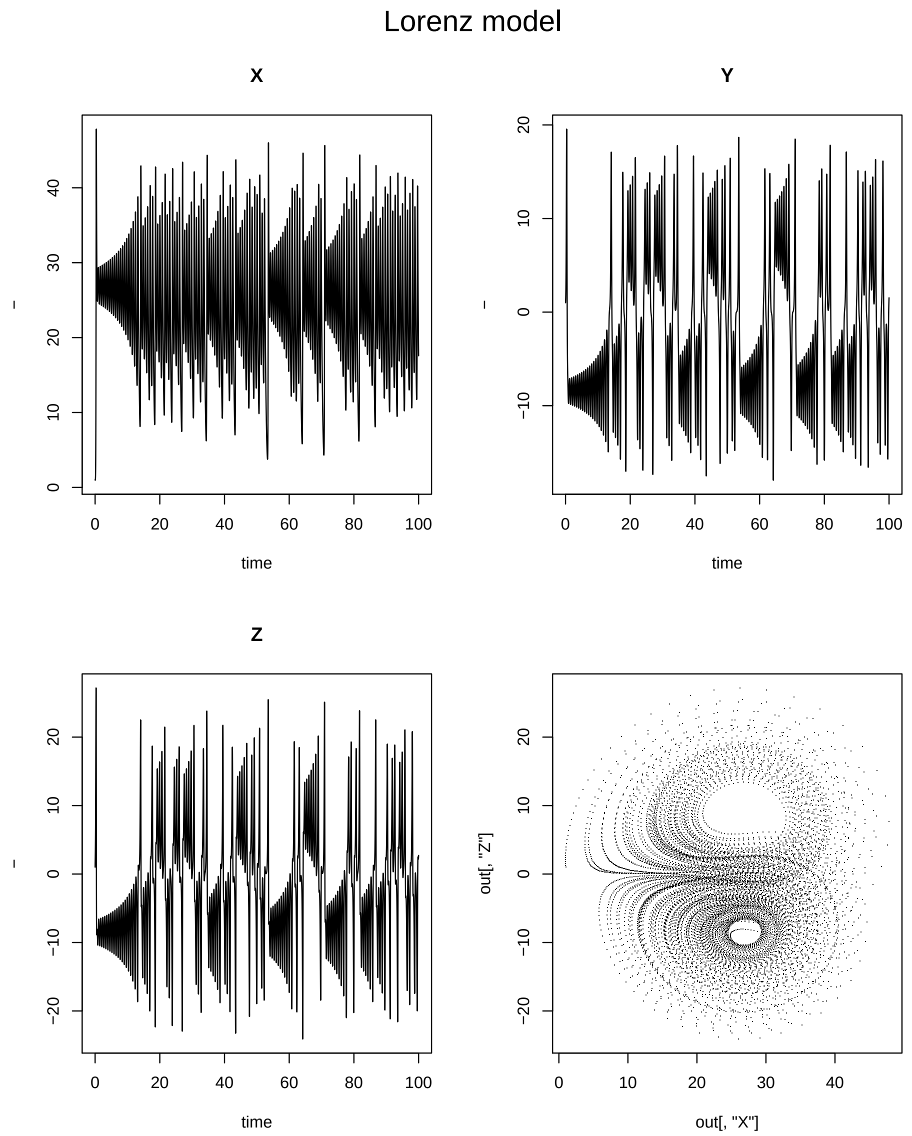

Figure 1: Solution of the ordinary differential equation - see text for R-code

2. Solvers for initial value problems of ordinary differential equations

Package **deSolve** contains several IVP ordinary differential equation solvers, that belong to the most important classes of solvers. Most functions are based on original (FORTRAN) implementations, e.g. the Backward Differentiation Formulae and Adams methods from **ODEPACK** (Hindmarsh 1983), or from (Brown, Byrne, and Hindmarsh 1989; Petzold 1983), the implicit Runge-Kutta method RADAU (Hairer and Wanner 2010). The package contains also a de novo implementation of several explicit Runge-Kutta methods (Butcher 1987; Press *et al.* 1992; Hairer, Norsett, and Wanner 2009).

All methods[1] can be triggered from function `ode` (by setting the argument `method`), or can be run as stand-alone functions. Moreover, for each integration routine, several options are available to optimise performance.

The default integration method, based on the FORTRAN code LSODA is one that switches automatically between stiff and non-stiff systems (Petzold 1983). Thus it should be possible to find, for one particular problem, the most efficient solver. See (Soetaert *et al.* 2010a) for more information about when to use which solver in **deSolve**. For most cases, the default solver, `ode` and using the default settings will do. Table 1 gives a short overview of the available methods.

We solve the model with several integration routines, each time printing the time it took (in seconds) to find the solution:

```
> print(system.time(out1 <- rk4   (state, times, Lorenz, parameters)))

   user  system elapsed
   4.65    0.00    4.67

> print(system.time(out2 <- lsode (state, times, Lorenz, parameters)))

   user  system elapsed
   1.66    0.00    1.66

> print(system.time(out  <- lsoda (state, times, Lorenz, parameters)))

   user  system elapsed
   2.22    0.00    2.23

> print(system.time(out  <- lsodes(state, times, Lorenz, parameters)))

   user  system elapsed
   1.53    0.00    1.50

> print(system.time(out  <- daspk (state, times, Lorenz, parameters)))

   user  system elapsed
   2.43    0.00    2.44
```

[1] except `zvode`, the solver used for systems containing complex numbers.

```
> print(system.time(out  <- vode  (state, times, Lorenz, parameters)))

   user  system elapsed 
   1.58    0.00    1.57 
```

2.1. Runge-Kutta methods

The explicit Runge-Kutta methods are de novo implementations in C, based on the Butcher tables (Butcher 1987). They comprise simple Runge-Kutta formulae (Heun's method `rk2`, the classical 4th order Runge-Kutta, `rk4`) and several Runge-Kutta pairs of order 3(2) to order 8(7). The embedded, explicit methods are according to Fehlberg (1967) (`rk..f`, `ode45`), Dormand and Prince (1980, 1981) (`rk..dp.`), Bogacki and Shampine (1989) (`rk23bs`, `ode23`) and Cash and Karp (1990) (`rk45ck`), where `ode23` and `ode45` are aliases for the popular methods `rk23bs` resp. `rk45dp7`.

With the following statement all implemented methods are shown:

```
> rkMethod()

 [1] "euler"   "rk2"     "rk4"     "rk23"    "rk23bs"  "rk34f"  
 [7] "rk45f"   "rk45ck"  "rk45e"   "rk45dp6" "rk45dp7" "rk78dp" 
[13] "rk78f"   "irk3r"   "irk5r"   "irk4hh"  "irk6kb"  "irk4l"  
[19] "irk6l"   "ode23"   "ode45"  
```

This list also contains implicit Runge-Kutta's (`irk..`), but they are not yet optimally coded. The only well-implemented implicit Runge-Kutta is the `radau` method (Hairer and Wanner 2010) that will be discussed in the section dealing with differential algebraic equations.

The properties of a Runge-Kutta method can be displayed as follows:

```
> rkMethod("rk23")

$ID
[1] "rk23"

$varstep
[1] TRUE

$FSAL
[1] FALSE

$A
     [,1] [,2] [,3]
[1,]  0.0    0    0
[2,]  0.5    0    0
[3,] -1.0    2    0

$b1
```

```
[1] 0 1 0

$b2
[1] 0.1666667 0.6666667 0.1666667

$c
[1] 0.0 0.5 2.0

$stage
[1] 3

$Qerr
[1] 2

attr(,"class")
[1] "list"     "rkMethod"
```

Here `varstep` informs whether the method uses a variable time-step; `FSAL` whether the first same as last strategy is used, while `stage` and `Qerr` give the number of function evaluations needed for one step, and the order of the local truncation error. `A,b1,b2,c` are the coefficients of the Butcher table. Two formulae (`rk45dp7, rk45ck`) support dense output.

It is also possible to modify the parameters of a method (be very careful with this) or define and use a new Runge-Kutta method:

```
> func <- function(t, x, parms) {
+   with(as.list(c(parms, x)),{
+     dP  <- a * P      - b * C * P
+     dC  <- b * P * C  - c * C
+     res <- c(dP, dC)
+     list(res)
+   })
+ }
> rKnew <- rkMethod(ID = "midpoint",
+   varstep = FALSE,
+   A       = c(0, 1/2),
+   b1      = c(0, 1),
+   c       = c(0, 1/2),
+   stage   = 2,
+   Qerr    = 1
+ )
> out <- ode(y = c(P = 2, C = 1), times = 0:100, func,
+       parms = c(a = 0.1, b = 0.1, c = 0.1), method = rKnew)
> head(out)
```

```
     time        P        C
[1,]    0 2.000000 1.000000
```

```
[2,]    1 1.990000 1.105000
[3,]    2 1.958387 1.218598
[4,]    3 1.904734 1.338250
[5,]    4 1.830060 1.460298
[6,]    5 1.736925 1.580136
```

2.2. Model diagnostics

Function `diagnostics` prints several diagnostics of the simulation to the screen. For the Runge-Kutta and `lsode` routine they are:

```
> diagnostics(out1)

--------------------
rk return code
--------------------

  return code (idid) =  0
  Integration was successful.

--------------------
INTEGER values
--------------------

  1 The return code : 0
  2 The number of steps taken for the problem so far: 10000
  3 The number of function evaluations for the problem so far: 40001
 18 The order (or maximum order) of the method: 4

> diagnostics(out2)

--------------------
lsode return code
--------------------

  return code (idid) =  2
  Integration was successful.

--------------------
INTEGER values
--------------------

  1 The return code : 2
  2 The number of steps taken for the problem so far: 12755
  3 The number of function evaluations for the problem so far: 16577
  5 The method order last used (successfully): 5
```

```
  6 The order of the method to be attempted on the next step: 5
  7 If return flag =-4,-5: the largest component in error vector 0
  8 The length of the real work array actually required: 58
  9 The length of the integer work array actually required: 23
 14 The number of Jacobian evaluations and LU decompositions so far: 716

--------------------
RSTATE values
--------------------

  1 The step size in t last used (successfully): 0.01
  2 The step size to be attempted on the next step: 0.01
  3 The current value of the independent variable which the solver has reached: 100.0052
  4 Tolerance scale factor > 1.0 computed when requesting too much accuracy: 0
```

3. Partial differential equations

As package **deSolve** includes integrators that deal efficiently with arbitrarily sparse and banded Jacobians, it is especially well suited to solve initial value problems resulting from 1, 2 or 3-dimensional partial differential equations (PDE). These are first written as ODEs using the method-of-lines approach.

Three special-purpose solvers are included in **deSolve**:

- `ode.band` integrates 1-dimensional problems comprizing one species,
- `ode.1D` integrates 1-dimensional problems comprizing one or many species,
- `ode.2D` integrates 2-dimensional problems,
- `ode.2D` integrates 2-dimensional problems.

As an example, consider the Aphid model described in Soetaert and Herman (2009). It is a model where aphids (a pest insect) slowly diffuse and grow on a row of plants. The model equations are:

$$\frac{\partial N}{\partial t} = -\frac{\partial Flux}{\partial x} + g \cdot N$$

and where the diffusive flux is given by:

$$Flux = -D\frac{\partial N}{\partial x}$$

with boundary conditions

$$N_{x=0} = N_{x=60} = 0$$

and initial condition

$$N_x = 0 \text{ for } x \neq 30$$
$$N_x = 1 \text{ for } x = 30$$

In the method of lines approach, the spatial domain is subdivided in a number of boxes and the equation is discretized as:

$$\frac{dN_i}{dt} = -\frac{Flux_{i,i+1} - Flux_{i-1,i}}{\Delta x_i} + g \cdot N_i$$

with the flux on the interface equal to:

$$Flux_{i-1,i} = -D_{i-1,i} \cdot \frac{N_i - N_{i-1}}{\Delta x_{i-1,i}}$$

Note that the values of state variables (here densities) are defined in the centre of boxes (i), whereas the fluxes are defined on the box interfaces. We refer to Soetaert and Herman (2009) for more information about this model and its numerical approximation.

Here is its implementation in R. First the model equations are defined:

```
> Aphid <- function(t, APHIDS, parameters) {
+   deltax     <- c (0.5, rep(1, numboxes - 1), 0.5)
+   Flux       <- -D * diff(c(0, APHIDS, 0)) / deltax
+   dAPHIDS    <- -diff(Flux) / delx + APHIDS * r
+
+   # the return value
+   list(dAPHIDS )
+ } # end
```

Then the model parameters and spatial grid are defined

```
> D         <- 0.3    # m2/day  diffusion rate
> r         <- 0.01   # /day    net growth rate
> delx      <- 1      # m       thickness of boxes
> numboxes  <- 60
> # distance of boxes on plant, m, 1 m intervals
> Distance  <- seq(from = 0.5, by = delx, length.out = numboxes)
```

Aphids are initially only present in two central boxes:

```
> # Initial conditions:  # ind/m2
> APHIDS        <- rep(0, times = numboxes)
> APHIDS[30:31] <- 1
> state         <- c(APHIDS = APHIDS)      # initialise state variables
```

The model is run for 200 days, producing output every day; the time elapsed in seconds to solve this 60 state-variable model is estimated (`system.time`):

```
> times <-seq(0, 200, by = 1)
> print(system.time(
+   out <- ode.1D(state, times, Aphid, parms = 0, nspec = 1)
+ ))

   user  system elapsed 
   0.16    0.00    0.15 
```

Matrix `out` consist of times (1st column) followed by the densities (next columns).

```
> head(out[,1:5])

     time      APHIDS1      APHIDS2      APHIDS3      APHIDS4
[1,]    0 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[2,]    1 1.667194e-55 9.555028e-52 2.555091e-48 4.943131e-45
[3,]    2 3.630860e-41 4.865105e-39 5.394287e-37 5.053775e-35
[4,]    3 2.051210e-34 9.207997e-33 3.722714e-31 1.390691e-29
[5,]    4 1.307456e-30 3.718598e-29 9.635350e-28 2.360716e-26
[6,]    5 6.839152e-28 1.465288e-26 2.860056e-25 5.334391e-24
```

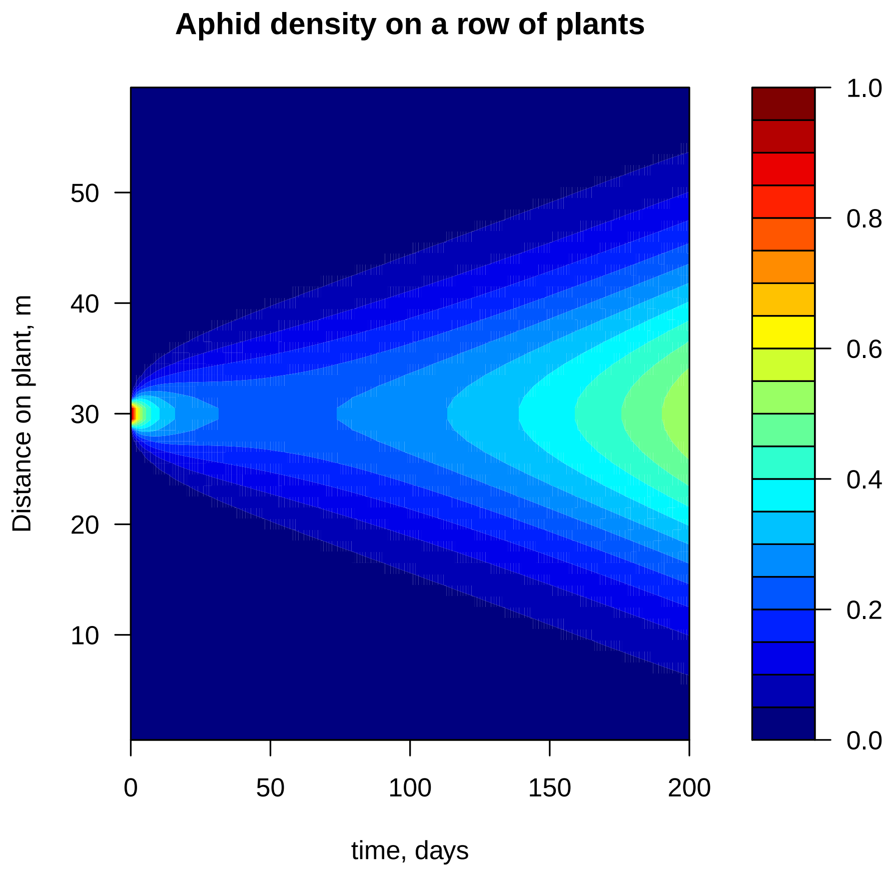

Figure 2: Solution of the 1-dimensional aphid model - see text for R -code

Finally, the output is plotted. It is simplest to do this with **deSolve**'s S3-method `image`

```
> image(out, method = "filled.contour", grid = Distance,
+       xlab = "time, days", ylab = "Distance on plant, m",
+       main = "Aphid density on a row of plants")
```

As the 1-D model describes only one species, it is best solved with **deSolve** function `ode.1D`. A multi-species IVP example can be found in Soetaert and Herman (2009). For 2-D problems, we refer to the help-files of function `ode.2D`.

4. Differential algebraic equations

Package **deSolve** contains two functions that solve initial value problems of differential algebraic equations. They are:

- `radau` which implements the implicit Runge-Kutta RADAU5 (Hairer and Wanner 2010),
- `daspk`, based on the backward differentiation code DASPK (Brenan, Campbell, and Petzold 1996).

Function `radau` needs the input in the form $My' = f(t, y, y')$ where M is the mass matrix. Function `daspk` also supports this input, but can also solve problems written in the form $F(t, y, y') = 0$.

`radau` solves problems up to index 3; `daspk` solves problems of index ≤ 1.

4.1. DAEs of index maximal 1

Function `daspk` from package **deSolve** solves (relatively simple) DAEs of index[2] maximal 1.

The DAE has to be specified by the *residual function* instead of the rates of change (as in ODE). Consider the following simple DAE:

$$\begin{aligned} \frac{dy_1}{dt} &= -y_1 + y_2 \\ y_1 \cdot y_2 &= t \end{aligned}$$

where the first equation is a differential, the second an algebraic equation. To solve it, it is first rewritten as residual functions:

$$\begin{aligned} 0 &= \frac{dy_1}{dt} + y_1 - y_2 \\ 0 &= y_1 \cdot y_2 - t \end{aligned}$$

In R we write:

```
> daefun <- function(t, y, dy, parameters) {
+   res1 <- dy[1] + y[1] - y[2]
+   res2 <- y[2] * y[1] - t
+
+   list(c(res1, res2))
+ }
> library(deSolve)
> yini  <- c(1, 0)
> dyini <- c(1, 0)
> times <- seq(0, 10, 0.1)
> ## solver
> print(system.time(out <- daspk(y = yini, dy = dyini,
+                            times = times, res = daefun, parms = 0)))
```

[2] note that many – apparently simple – DAEs are higher-index DAEs

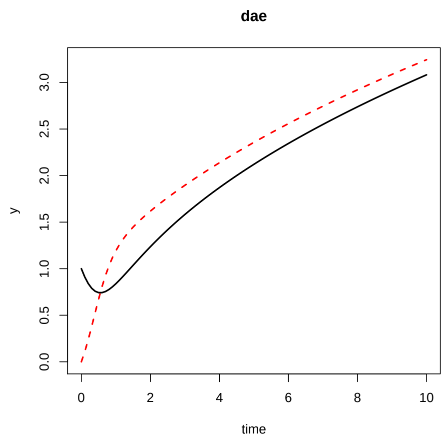

Figure 3: Solution of the differential algebraic equation model - see text for R-code

```
   user  system elapsed 
   0.02    0.00    0.02 
```

```
> matplot(out[,1], out[,2:3], type = "l", lwd = 2,
+         main = "dae", xlab = "time", ylab = "y")
```

4.2. DAEs of index up to three

Function `radau` from package **deSolve** can solve DAEs of index up to three provided that they can be written in the form $M dy/dt = f(t, y)$.

Consider the well-known pendulum equation:

$$\begin{aligned}
x' &= u \\
y' &= v \\
u' &= -\lambda x \\
v' &= -\lambda y - 9.8 \\
0 &= x^2 + y^2 - 1
\end{aligned}$$

where the dependent variables are x, y, u, v and λ.

Implemented in R to be used with function `radau` this becomes:

```
> pendulum <- function (t, Y, parms) {
+   with (as.list(Y),
+     list(c(u,
+            v,
+           -lam * x,
+           -lam * y - 9.8,
+            x^2 + y^2 -1
```

```
+       ))
+   )
+ }
```

A consistent set of initial conditions are:

```
> yini <- c(x = 1, y = 0, u = 0, v = 1, lam = 1)
```

and the mass matrix M:

```
> M <- diag(nrow = 5)
> M[5, 5] <- 0
> M

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    0
```

Function radau requires that the index of each equation is specified; there are 2 equations of index 1, two of index 2, one of index 3:

```
> index <- c(2, 2, 1)
> times <- seq(from = 0, to = 10, by = 0.01)
> out  <- radau (y = yini, func = pendulum, parms = NULL,
+                 times = times, mass = M, nind = index)

> plot(out, type = "l", lwd = 2)
> plot(out[, c("x", "y")], type = "l", lwd = 2)
```

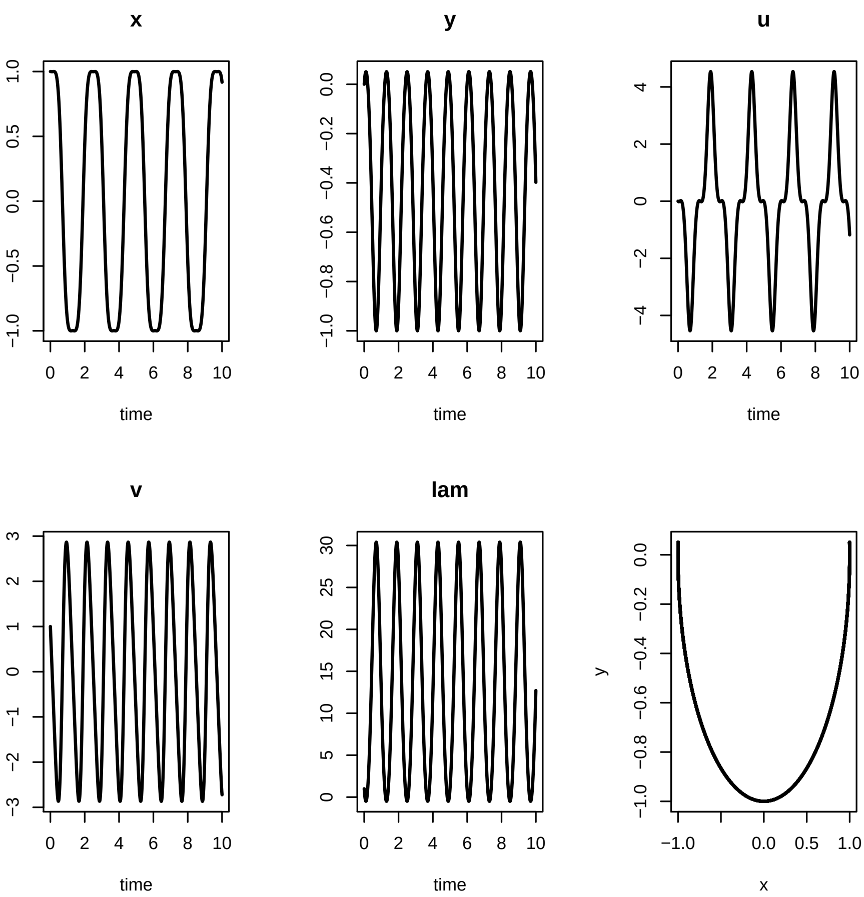

Figure 4: Solution of the pendulum problem, an index 3 differential algebraic equation using `radau` - see text for R-code

5. Integrating systems containing complex numbers, function zvode

Function `zvode` solves ODEs that are composed of complex variables. We use `zvode` to solve the following system of 2 ODEs:

$$\frac{dz}{dt} = i \cdot z$$
$$\frac{dw}{dt} = -i \cdot w \cdot w \cdot z$$

where

$$w(0) = 1/2.1$$
$$z(0) = 1$$

on the interval $t = [0, 2\pi]$

```
> ZODE2 <- function(Time, State, Pars) {
+   with(as.list(State), {
+        df <- 1i * f
+        dg <- -1i * g * g * f
+        return(list(c(df, dg)))
+   })
+ }
> yini  <- c(f = 1+0i, g = 1/2.1+0i)
> times <- seq(0, 2 * pi, length = 100)
> out   <- zvode(func = ZODE2, y = yini, parms = NULL, times = times,
+   atol = 1e-10, rtol = 1e-10)
```

The analytical solution is:

$$f(t) = \exp(1i \cdot t)$$

and

$$g(t) = 1/(f(t) + 1.1)$$

The numerical solution, as produced by `zvode` matches the analytical solution:

```
> analytical <- cbind(f = exp(1i*times), g = 1/(exp(1i*times)+1.1))
> tail(cbind(out[,2], analytical[,1]))

                      [,1]                  [,2]
[95,]  0.9500711-0.3120334i 0.9500711-0.3120334i
[96,]  0.9679487-0.2511480i 0.9679487-0.2511480i
[97,]  0.9819287-0.1892512i 0.9819287-0.1892512i
[98,]  0.9919548-0.1265925i 0.9919548-0.1265925i
[99,]  0.9979867-0.0634239i 0.9979867-0.0634239i
[100,] 1.0000000+0.0000000i 1.0000000-0.0000000i
```

6. Making good use of the integration options

The solvers from **ODEPACK** can be fine-tuned if it is known whether the problem is stiff or non-stiff, or if the structure of the Jacobian is sparse. We repeat the example from `lsode` to show how we can make good use of these options.

The model describes the time evolution of 5 state variables:

```
> f1 <- function  (t, y, parms) {
+   ydot <- vector(len = 5)
+
+   ydot[1] <-  0.1*y[1] -0.2*y[2]
+   ydot[2] <- -0.3*y[1] +0.1*y[2] -0.2*y[3]
+   ydot[3] <-           -0.3*y[2] +0.1*y[3] -0.2*y[4]
+   ydot[4] <-                     -0.3*y[3] +0.1*y[4] -0.2*y[5]
+   ydot[5] <-                               -0.3*y[4] +0.1*y[5]
+
+   return(list(ydot))
+ }
```

and the initial conditions and output times are:

```
> yini  <- 1:5
> times <- 1:20
```

The default solution, using `lsode` assumes that the model is stiff, and the integrator generates the Jacobian, which is assummed to be *full*:

```
> out   <- lsode(yini, times, f1, parms = 0, jactype = "fullint")
```

It is possible for the user to provide the Jacobian. Especially for large problems this can result in substantial time savings. In a first case, the Jacobian is written as a full matrix:

```
> fulljac <- function  (t, y, parms) {
+   jac <- matrix(nrow = 5, ncol = 5, byrow = TRUE,
+                data = c(0.1, -0.2,  0  ,  0  ,  0  ,
+                        -0.3,  0.1, -0.2,  0  ,  0  ,
+                         0  , -0.3,  0.1, -0.2,  0  ,
+                         0  ,  0  , -0.3,  0.1, -0.2,
+                         0  ,  0  ,  0  , -0.3,  0.1))
+   return(jac)
+ }
```

and the model solved as:

```
> out2 <- lsode(yini, times, f1, parms = 0, jactype = "fullusr",
+               jacfunc = fulljac)
```

The Jacobian matrix is banded, with one nonzero band above (up) and one below(down) the diagonal. First we let `lsode` estimate the banded Jacobian internally (`jactype = "bandint"`):

```
> out3 <- lsode(yini, times, f1, parms = 0, jactype = "bandint",
+               bandup = 1, banddown = 1)
```

It is also possible to provide the nonzero bands of the Jacobian in a function:

```
> bandjac <- function  (t, y, parms) {
+   jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
+                 data = c( 0   , -0.2, -0.2, -0.2, -0.2,
+                           0.1,  0.1,  0.1,  0.1,  0.1,
+                          -0.3, -0.3, -0.3, -0.3,  0))
+   return(jac)
+ }
```

in which case the model is solved as:

```
> out4 <- lsode(yini, times, f1, parms = 0, jactype = "bandusr",
+               jacfunc = bandjac, bandup = 1, banddown = 1)
```

Finally, if the model is specified as "non-stiff" (by setting `mf=10`), there is no need to specify the Jacobian:

```
> out5  <- lsode(yini, times, f1, parms = 0, mf = 10)
```

7. Events

As from version 1.6, `events` are supported. Events occur when the values of state variables are instantaneously changed. They can be specified as a `data.frame`, or in a function. Events can also be triggered by a root function.

7.1. Event specified in a data.frame

In this example, two state variables with constant decay are modeled:

```
> eventmod <- function(t, var, parms) {
+   list(dvar = -0.1*var)
+ }
> yini <- c(v1 = 1, v2 = 2)
> times <- seq(0, 10, by = 0.1)
```

At time 1 and 9 a value is added to variable `v1`, at time 1 state variable `v2` is multiplied with 2, while at time 5 the value of `v2` is replaced with 3. These events are specified in a `data.frame`, eventdat:

```
> eventdat <- data.frame(var = c("v1", "v2", "v2", "v1"), time = c(1, 1, 5, 9),
+   value = c(1, 2, 3, 4), method = c("add", "mult", "rep", "add"))
> eventdat

  var time value method
1  v1    1     1    add
2  v2    1     2   mult
3  v2    5     3    rep
4  v1    9     4    add
```

The model is solved with `vode`:

```
> out <- ode(func = eventmod, y = yini, times = times, parms = NULL,
+   events = list(data = eventdat))

> plot(out, type = "l", lwd = 2)
```

7.2. Event triggered by a root function

This model describes the position (y1) and velocity (y2) of a bouncing ball:

```
> ballode<- function(t, y, parms) {
+   dy1 <- y[2]
+   dy2 <- -9.8
+   list(c(dy1, dy2))
+ }
```

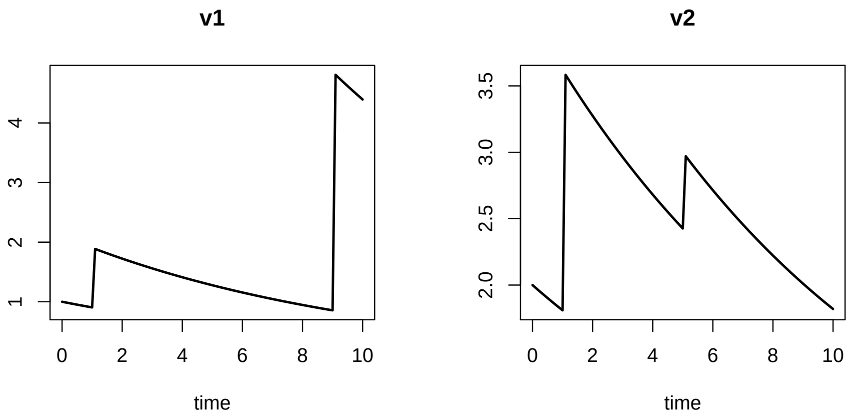

Figure 5: A simple model that contains events

An event is triggered when the ball hits the ground (height = 0) Then velocity (`y2`) is reversed and reduced by 10 percent. The root function, `y[1] = 0`, triggers the event:

```
> root <- function(t, y, parms) y[1]
```

The event function imposes the bouncing of the ball

```
> event <- function(t, y, parms) {
+  y[1]<- 0
+  y[2]<- -0.9 * y[2]
+  return(y)
+ }
```

After specifying the initial values and times, the model is solved. Both integrators `lsodar` or `lsode` can estimate a root.

```
> yini  <- c(height = 0, v = 20)
> times <- seq(from = 0, to = 20, by = 0.01)
> out   <- lsode(times = times, y = yini, func = ballode, parms = NULL,
+   events = list(func = event, root = TRUE), rootfun = root)

> plot(out, which = "height", type = "l",lwd = 2,
+   main = "bouncing ball", ylab = "height")
```

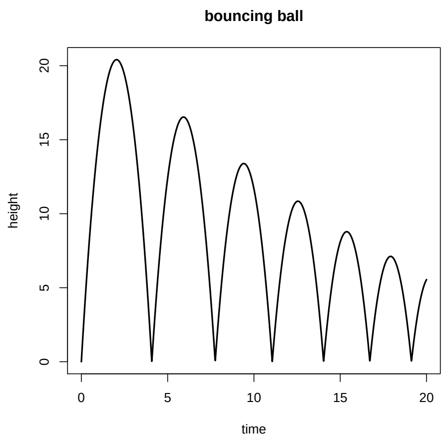

Figure 6: A model, with event triggered by a root function

8. Delay differential equations

As from deSolve version 1.7, time lags are supported, and a new general solver for delay differential equations, `dede` has been added. We implement the lemming model, example 6 from Shampine and Thompson, 2000 solving delay differential equations with `dde23`.

Function `lagvalue` calculates the value of the state variable at t-0.74. As long a these lag values are not known, the value 19 is assigned to the state variable. Note that the simulation starts at time = - 0.74.

```
> require(deSolve)
> #-----------------------------
> # the derivative function
> #-----------------------------
> derivs <- function(t, y, parms) {
+   if (t < 0)
+     lag <- 19
+   else
+     lag <- lagvalue(t - 0.74)
+
+   dy <- r * y* (1 - lag/m)
+   list(dy, dy = dy)
+ }
> #-----------------------------
> # parameters
> #-----------------------------
>
> r <- 3.5; m <- 19
> #-----------------------------
> # initial values and times
> #-----------------------------
>
> yinit <- c(y = 19.001)
> times <- seq(-0.74, 40, by = 0.01)
> #-----------------------------
> # solve the model
> #-----------------------------
>
> yout <- dede(y = yinit, times = times, func = derivs,
+              parms = NULL, atol = 1e-10)

> plot(yout, which = 1, type = "l", lwd = 2, main = "Lemming model", mfrow = c(1,2))
> plot(yout[,2], yout[,3], xlab = "y", ylab = "dy", type = "l", lwd = 2)
```

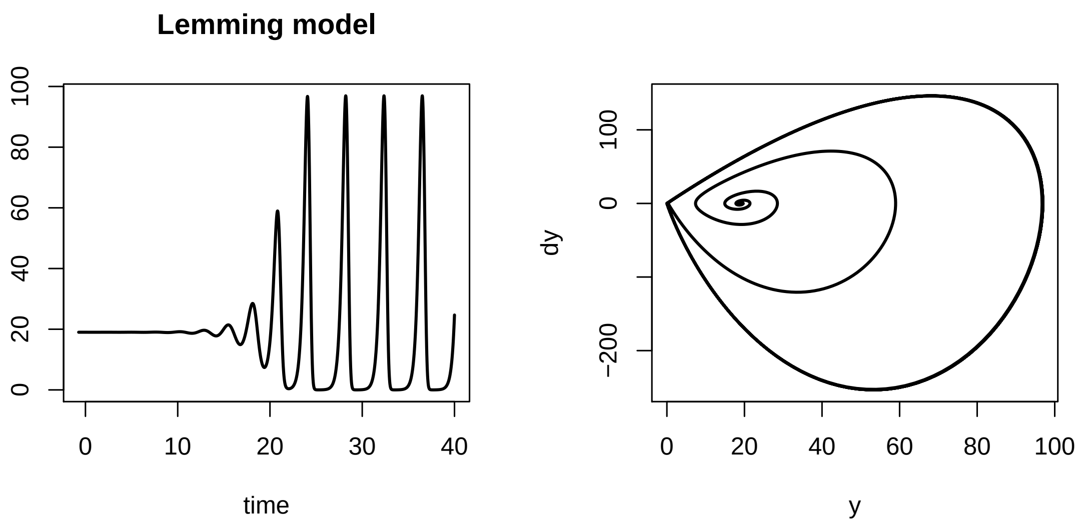

Figure 7: A delay differential equation model

9. Troubleshooting

9.1. Avoiding numerical errors

The solvers from **ODEPACK** should be first choice for any problem and the defaults of the control parameters are reasonable for many practical problems. However, there are cases where they may give dubious results. Consider the following Lotka-Volterra type of model:

```
> SPCmod <- function(t, x, parms)  {
+   with(as.list(c(parms, x)), {
+     dP <- c*P   - d*C*P      # producer
+     dC <- e*P*C - f*C        # consumer
+     res <- c(dP, dC)
+     list(res)
+   })
+ }
```

and with the following (biologically not very realistic)[3] parameter values:

```
> parms  <- c(c = 5, d = 0.1, e = 0.1, f = 0.1)
```

After specification of initial conditions and output times, the model is solved - using `lsoda`:

```
> xstart <- c(P = 0.5, C = 1)
> times  <- seq(0, 190, 0.1)
> out <- ode(y = xstart, times = times,
+   func = SPCmod, parms = parms)
> tail(out)
```

```
         time   P   C
[1896,] 189.5 NaN NaN
[1897,] 189.6 NaN NaN
[1898,] 189.7 NaN NaN
[1899,] 189.8 NaN NaN
[1900,] 189.9 NaN NaN
[1901,] 190.0 NaN NaN
```

At the end of the simulation, both producers and consumer values are Not-A-Numbers!

What has happened? Being an implicit method, `lsoda` generates very small negative values for producers, from day 40 on; these negative values, small at first grow in magnitude until they become NaNs. This is because the model equations are not intended to be used with negative numbers, as negative concentrations are not realistic.

A quick-and-dirty solution is to reduce the maximum time step to a considerably small value (e.g. `hmax = 0.02` which, of course, reduces computational efficiency. However, a much better

[3]they are not realistic because producers grow unlimited with a high rate and consumers with 100 % efficiency

solution is to think about the reason of the failure, i.e in our case the **absolute** accuracy because the states can reach very small absolute values. Therefore, it helps here to reduce `atol` to a very small number or even to zero:

```
> out <- ode(y = xstart,times = times, func = SPCmod,
+                          parms = parms, atol = 0)
> matplot(out[,1], out[,2:3], type = "l")
```

It is, of course, not possible to set both, `atol` and `rtol` simultaneously to zero. As we see at this example, it is always a good idea to test simulation results for plausibility. This can be done by theoretical considerations or by comparing the outcome of different ODE solvers and parametrizations.

9.2. Checking model specification

If a model outcome is obviously unrealistic or one of the **deSolve** functions complains about numerical problems it is even more likely that the "numerical problem" is in fact a result of an unrealistic model or a programming error. In such cases, playing with solver parameters will not help. Here are some common mistakes we observed in our models and the codes of our students:

- The function with the model definition must return a list with the derivatives of all state variables in correct order (and optionally some global values). Check if the number and order of your states is identical in the initial states `y` passed to the solver, in the assignments within your model equations and in the returned values. Check also whether the return value is the last statement of your model definition.
- The order of function arguments in the model definition is `t, y, parms, ...`. This order is strictly fixed, so that the **deSolve** solvers can pass their data, but naming is flexible and can be adapted to your needs, e.g. `time, init, params`. Note also that all three arguments must be given, even if `t` is not used in your model.
- Mixing of variable names: if you use the `with()`-construction explained above, you must ensure to avoid naming conflicts between parameters (`parms`) and state variables (`y`).

The solvers included in package **deSolve** are thorougly tested, however they come with no warranty and the user is solely responsible for their correct application. If you encounter unexpected behavior, first check your model and read the documentation. If this doesn't help, feel free to ask a question to an appropriate mailing list, e.g. `r-help@r-project.org` or, more specific, `r-sig-dynamic-models@r-project.org`.

9.3. Making sense of deSolve's error messages

As many of **deSolve**'s functions are wrappers around existing FORTRAN codes, the warning and error messages are derived from these codes. Whereas these codes are highly robust, well tested, and efficient, they are not always as user-friendly as we would like. Especially some of the warnings/error messages may appear to be difficult to understand.

Consider the first example on the `ode` function:

```
> LVmod <- function(Time, State, Pars) {
+   with(as.list(c(State, Pars)), {
+     Ingestion    <- rIng  * Prey*Predator
+     GrowthPrey   <- rGrow * Prey*(1-Prey/K)
+     MortPredator <- rMort * Predator
+
+     dPrey        <- GrowthPrey - Ingestion
+     dPredator    <- Ingestion*assEff -MortPredator
+
+     return(list(c(dPrey, dPredator)))
+   })
+ }
> pars    <- c(rIng   = 0.2,    # /day, rate of ingestion
+              rGrow  = 1.0,    # /day, growth rate of prey
+              rMort  = 0.2 ,   # /day, mortality rate of predator
+              assEff = 0.5,    # -, assimilation efficiency
+              K      = 10)     # mmol/m3, carrying capacity
> yini    <- c(Prey = 1, Predator = 2)
> times   <- seq(0, 200, by = 1)
> out     <- ode(func = LVmod, y = yini,
+               parms = pars, times = times)
```

This model is easily solved by the default integration method, `lsoda`.

Now we change one of the parameters to an unrealistic value: `rIng` is set to 100. This means that the predator ingests 100 times its own body-weight per day if there are plenty of prey. Needless to say that this is very unhealthy, if not lethal.

Also, `lsoda` cannot solve the model anymore. Thus, if we try:

```
> pars["rIng"] <- 100
> out2 <- ode(func = LVmod, y = yini,
+            parms = pars, times = times)
```

A lot of seemingly incomprehensible messages will be written to the screen. We repeat the latter part of them:

```
DLSODA-  Warning..Internal T (=R1) and H (=R2) are
      such that in the machine, T + H = T on the next step
     (H = step size). Solver will continue anyway.
      In above,  R1 =  0.8562448350331D+02   R2 =  0.3273781304624D-17
DLSODA-  Above warning has been issued I1 times.
     It will not be issued again for this problem.
      In above message,  I1 =        10
DLSODA-  At current T (=R1), MXSTEP (=I1) steps
      taken on this call before reaching TOUT
      In above message,  I1 =      5000
      In above message,  R1 =  0.8562448350331D+02
Warning messages:
```

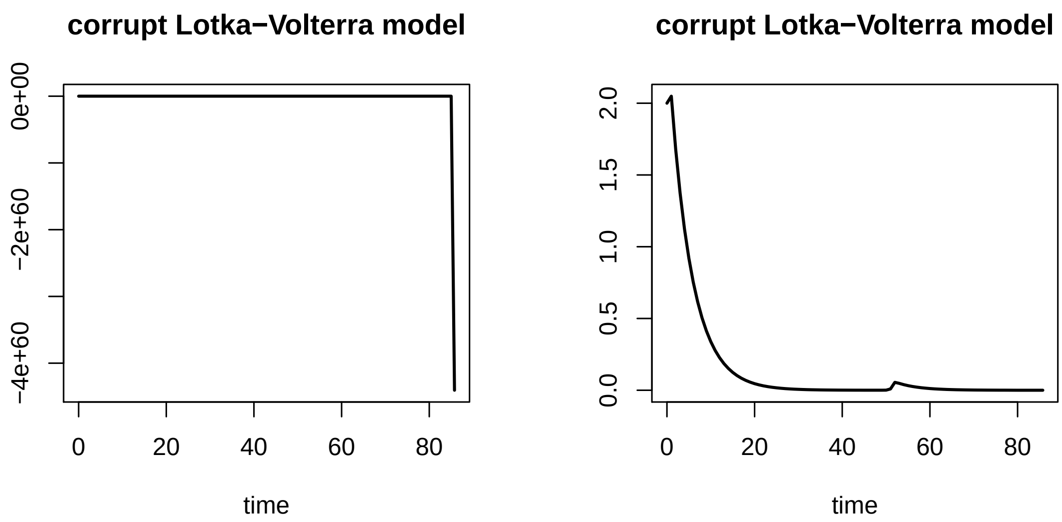

Figure 8: A model that cannot be solved correctly

```
1: In lsoda(y, times, func, parms, ...) :
  an excessive amount of work (> maxsteps ) was done, but integration was not successful -
2: In lsoda(y, times, func, parms, ...) :
  Returning early. Results are accurate, as far as they go
```

The first sentence tells us that at T=0.8562448350331e+02, the solver used a step size H= 0.3273781304624e-17. This step size is so small that it cannot tell the difference between T and T+H. Nevertheless, the solver tried again.

The second sentence tells that, at this warning has been occurring 10 times, it will not be outputted again.

As expected, this error did not go away, so soon the maximal number of steps (5000) has been exceeded. This is indeed what the next message is about:

The third sentence tells that at T=0.8562448350331e+02, maxstep =5000 steps have been done.

The one before last message tells why the solver returned prematurely, and suggests a solution.

Note: simply increasing maxsteps will not work. It makes more sense to first see if the output tells what happens:

```
> plot(out2, type = "l", lwd = 2, main = "corrupt Lotka-Volterra model")
```

References

Bogacki P, Shampine LF (1989). "A 3(2) Pair of Runge-Kutta Formulas." *Applied Mathematics Letters*, **2**, 1–9.

Brenan KE, Campbell SL, Petzold LR (1996). *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM Classics in Applied Mathematics.

Brown PN, Byrne GD, Hindmarsh AC (1989). "**VODE**, A Variable-Coefficient ODE Solver." *SIAM Journal on Scientific and Statistical Computing*, **10**, 1038–1051.

Butcher JC (1987). *The Numerical Analysis of Ordinary Differential Equations, Runge-Kutta and General Linear Methods*, volume 2. John Wiley & Sons, Chichester and New York.

Cash JR, Karp AH (1990). "A Variable Order Runge-Kutta Method for Initial Value Problems With Rapidly Varying Right-Hand Sides." *ACM Transactions on Mathematical Software*, **16**, 201–222.

Dormand JR, Prince PJ (1980). "A family of embedded Runge-Kutta formulae." *Journal of Computational and Applied Mathematics*, **6**, 19–26.

Dormand JR, Prince PJ (1981). "High Order Embedded Runge-Kutta Formulae." *Journal of Computational and Applied Mathematics*, **7**, 67–75.

Fehlberg E (1967). "Klassische Runge-Kutta-Formeln fuenfter and siebenter Ordnung mit Schrittweiten-Kontrolle." *Computing (Arch. Elektron. Rechnen)*, **4**, 93–106.

Hairer E, Norsett SP, Wanner G (2009). *Solving Ordinary Differential Equations I: Nonstiff Problems. Second Revised Edition*. Springer-Verlag, Heidelberg.

Hairer E, Wanner G (2010). *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems. Second Revised Edition*. Springer-Verlag, Heidelberg.

Hindmarsh AC (1983). "**ODEPACK**, a Systematized Collection of ODE Solvers." In R Stepleman (ed.), "Scientific Computing, Vol. 1 of IMACS Transactions on Scientific Computation," pp. 55–64. IMACS / North-Holland, Amsterdam.

Petzold LR (1983). "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations." *SIAM Journal on Scientific and Statistical Computing*, **4**, 136–148.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992). *Numerical Recipes in FORTRAN. The Art of Scientific Computing*. Cambridge University Press, 2nd edition.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

Soetaert K, Herman PMJ (2009). *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform*. Springer. ISBN 978-1-4020-8623-6.

Soetaert K, Petzoldt T, Setzer R (2010a). "Solving Differential Equations in R: Package **deSolve**." *Journal of Statistical Software*, **33**(9), 1–25. ISSN 1548-7660. URL http://www.jstatsoft.org/v33/i09.

Soetaert K, Petzoldt T, Setzer RW (2008). *R package* ***deSolve****: Writing Code in Compiled Languages*. **deSolve** vignette - R package version 1.8.

Soetaert K, Petzoldt T, Setzer RW (2010b). *deSolve: General solvers for initial value problems of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE) and delay differential equations (DDE)*. R package version 1.8.

Affiliation:

Karline Soetaert
Centre for Estuarine and Marine Ecology (CEME)
Netherlands Institute of Ecology (NIOO)
4401 NT Yerseke, Netherlands
E-mail: k.soetaert@nioo.knaw.nl
URL: http://www.nioo.knaw.nl/ppages/ksoetaert

Thomas Petzoldt
Institut für Hydrobiologie
Technische Universität Dresden
01062 Dresden, Germany
E-mail: thomas.petzoldt@tu-dresden.de
URL: http://tu-dresden.de/Members/thomas.petzoldt/

R. Woodrow Setzer
National Center for Computational Toxicology
US Environmental Protection Agency
URL: http://www.epa.gov/comptox

Table 1: Summary of the functions that solve differential equations

Function	Description
`ode`	integrates systems of ordinary differential equations, assumes a full, banded or arbitrary sparse Jacobian
`ode.1D`	integrates systems of ODEs resulting from 1-dimensional reaction-transport problems
`ode.2D`	integrates systems of ODEs resulting from 2-dimensional reaction-transport problems
`ode.3D`	integrates systems of ODEs resulting from 3-dimensional reaction-transport problems
`ode.band`	integrates systems of ODEs resulting from unicomponent 1-dimensional reaction-transport problems
`dede`	integrates systems of delay differential equations
`daspk`	solves systems of differential algebraic equations, assumes a full or banded Jacobian
`radau`	solves systems of ordinary or differential algebraic equations, assumes a full or banded Jacobian
`lsoda`	integrates ODEs, automatically chooses method for stiff or non-stiff problems, assumes a full or banded Jacobian
`lsodar`	same as `lsoda`, but includes a root-solving procedure
`lsode` or `vode`	integrates ODEs, user must specify if stiff or non-stiff assumes a full or banded Jacobian; Note that, as from version 1.7, `lsode` includes a root finding procedure, similar to `lsodar`.
`lsodes`	integrates ODEs, using stiff method and assuming an arbitrary sparse Jacobian
`rk`	integrates ODEs, using Runge-Kutta methods (includes Runge-Kutta 4 and Euler as special cases)
`rk4`	integrates ODEs, using the classical Runge-Kutta 4th order method (special code with less options than `rk`)
`euler`	integrates ODEs, using Euler's method (special code with less options than `rk`)
`zvode`	integrates ODEs composed of complex numbers, full, banded, stiff or nonstiff

Table 2: Meaning of the integer return parameters in the different integration routines. If `out` is the output matrix, then this vector can be retrieved by function `attributes(out)$istate`; its contents is displayed by function `diagnostics(out)`. Note that the number of function evaluations, is without the extra evaluations needed to generate the output for the ordinary variables.

Nr	Description
1	the return flag; the conditions under which the last call to the solver returned. For `lsoda, lsodar, lsode, lsodes, vode, rk, rk4, euler` these are: 2: the solver was successful, -1: excess work done, -2: excess accuracy requested, -3: illegal input detected, -4: repeated error test failures, -5: repeated convergence failures, -6: error weight became zero
2	the number of steps taken for the problem so far
3	the number of function evaluations for the problem so far
4	the number of Jacobian evaluations so far
5	the method order last used (successfully)
6	the order of the method to be attempted on the next step
7	If return flag = -4,-5: the largest component in the error vector
8	the length of the real work array actually required. (FORTRAN code)
9	the length of the integer work array actually required. (FORTRAN code)
10	the number of matrix LU decompositions so far
11	the number of nonlinear (Newton) iterations so far
12	the number of convergence failures of the solver so far
13	the number of error test failures of the integrator so far
14	the number of Jacobian evaluations and LU decompositions so far
15	the method indicator for the last succesful step, 1 = adams (nonstiff), 2 = bdf (stiff)
17	the number of nonzero elements in the sparse Jacobian
18	the current method indicator to be attempted on the next step, 1 = adams (nonstiff), 2 = bdf (stiff)
19	the number of convergence failures of the linear iteration so far

Table 3: Meaning of the double precision return parameters in the different integration routines. If `out` is the output matrix, then this vector can be retrieved by function `attributes(out)$rstate`; its contents is displayed by function `diagnostics(out)`

Nr	Description
1	the step size in t last used (successfully)
2	the step size to be attempted on the next step
3	the current value of the independent variable which the solver has actually reached
4	a tolerance scale factor, greater than 1.0, computed when a request for too much accuracy was detected
5	the value of t at the time of the last method switch, if any (only `lsoda, lsodar`)