Package deSolve: Solving Initial Value Differential Equations in R

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)
Yerseke, The Netherlands

Thomas Petzoldt
Technische Universität Dresden
Germany

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

Abstract

R package **deSolve** (**??**) the successor of R package **odesolve** is a package to solve initial value problems (IVP) of:

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

The implementation includes stiff and nonstiff integration routines based on the **ODEPACK** FORTRAN codes (**?**). It also includes fixed and adaptive time-step explicit Runge-Kutta solvers and the Euler method (**?**), and the implicit Runge-Kutta method RADAU (**?**).

In this vignette we outline how to implement differential equations as R -functions. Another vignette ("compiledCode") (**?**), 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 (**?**).

Note that another package, **bvpSolve** provides methods to solve boundary value problems (**?**).

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.

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

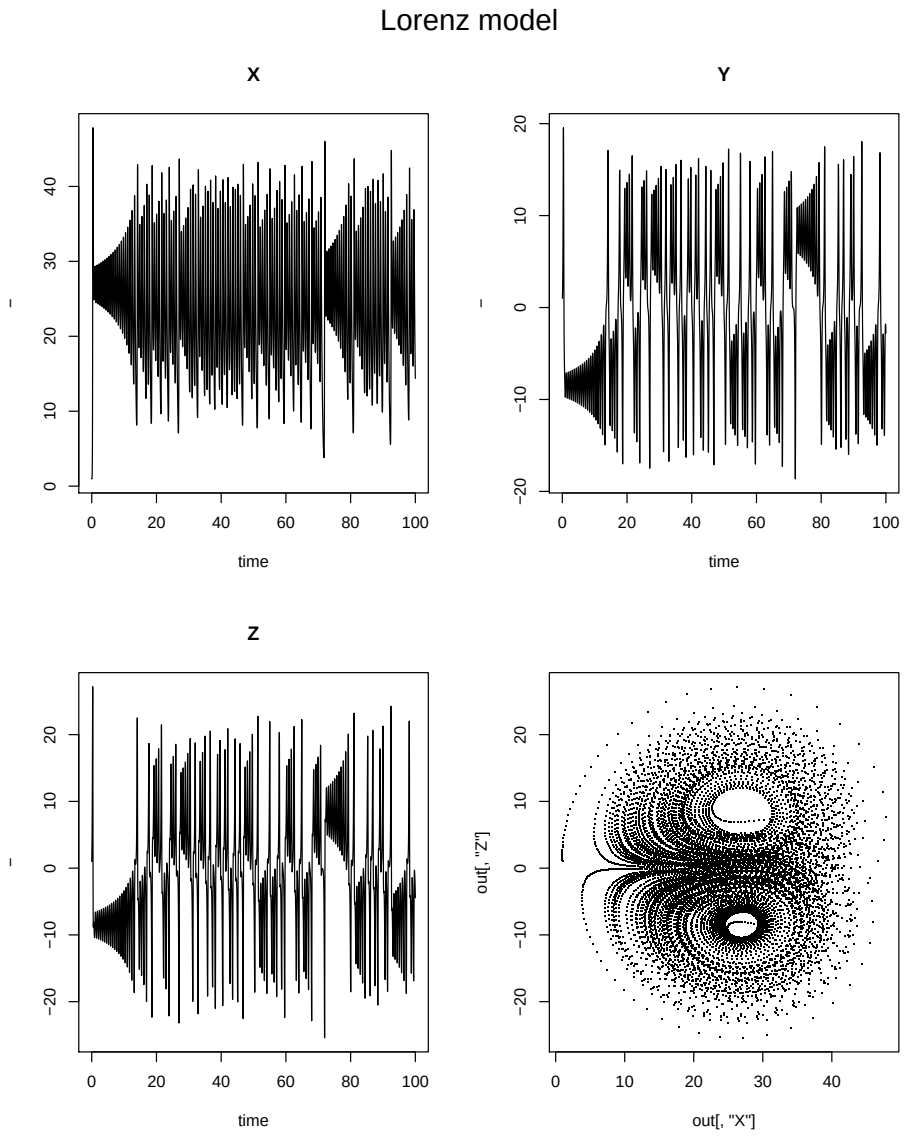

Figure 1: Solution of the ordinary differential equation - see text for R-code

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** (**?**), or from (**??**), the implicit Runge-Kutta method RADAU (**?**). The package contains also a de novo implementation of several Runge-Kutta methods (**???**).

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

For instance, the next statements will use integration method `radau` to solve the model, and set the tolerances to a higher value than the default. Both statements are the same:

```
> outb <- radau(state, times, Lorenz, parameters, atol = 1e-4, rtol = 1e-4)
> outc <- ode(state, times, Lorenz, parameters, method = "radau",
+              atol = 1e-4, rtol = 1e-4)
```

The default integration method, based on the FORTRAN code LSODA is one that switches automatically between stiff and non-stiff systems (**?**). This is a very robust method, but not necessarily the most efficient solver for one particular problem. See (**?**) 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 also gives a short overview of the available methods.

To show how to trigger the various 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
   1.64    0.01    1.67

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

   user  system elapsed
   0.59    0.00    0.59

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

   user  system elapsed
   0.81    0.00    0.81

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

   user  system elapsed
   0.55    0.00    0.56
```

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

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

   user  system elapsed
   0.83    0.00    0.85

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

   user  system elapsed
   0.58    0.00    0.60
```

2.1. Runge-Kutta methods and Euler

The explicit Runge-Kutta methods are de novo implementations in C, based on the Butcher tables (?). They comprise simple Runge-Kutta formulae (Euler's method `euler`, 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 ? (`rk..f`, `ode45`), ?? (`rk..dp.`), ? (`rk23bs`, `ode23`) and ? (`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 (?) 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
```

Fixed time-step methods

There are two explicit methods that do not adapt the time step: the `euler` method and the `rk4` method.

They are implemented in two ways:

- as a `rkMethod` of the **general** `rk` solver. In this case the time step used can be specified independently from the `times` argument, by setting argument `hini`. Function `ode` uses this general code.
- as **special** solver codes `euler` and `rk4`. These implementations are simplified and with less options to avoid overhead. The timestep used is determined by the time increment in the `times` argument.

For example, the next two statements both trigger the Euler method, the first using the "special" code with a time step = 1, as imposed by the `times` argument, the second using the generalized method with a time step set by `hini`. Unsurprisingly, the first solution method completely fails (the time step = 1 is much too large for this problem).

```
out  <- euler(y = state, times = 0:40, func = Lorenz, parms = parameters)
outb <- ode(y = state, times = 0:40, func = Lorenz, parms = parameters,
            method = "euler", hini = 0.01)
```

2.2. Model diagnostics and summaries

Function `diagnostics` prints several diagnostics of the simulation to the screen. For the Runge-Kutta and `lsode` routine called above 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: 12684
  3 The number of function evaluations for the problem so far: 16464
  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: 709

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

  1 The step size in t last used (successfully): 0.008487079
  2 The step size to be attempted on the next step: 0.008487079
  3 The current value of the independent variable which the solver has reached: 100.0047
  4 Tolerance scale factor > 1.0 computed when requesting too much accuracy: 0
```

There is also a `summary` method for `deSolve` objects. This is especially handy for multi-dimensional problems (see below)

```
> summary(out1)

                  X            Y            Z
Min.        0.96170   -17.910000   -24.030000
```

```
1st Qu.    17.15000    -7.255000    -7.018000
Median     23.24000    -1.677000    -1.376000
Mean       23.77000    -1.205000    -1.213000
3rd Qu.    30.20000     3.953000     3.421000
Max.       47.83000    19.560000    27.180000
N       10001.00000 10001.000000 10001.000000
sd          8.41055     7.875825     8.923277
```

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), using the method-of-lines approach. The PDEs are first written as ODEs, using finite differences. This can be efficiently done with functions from R-package **ReacTran** (**?**). However, here we will create the finite differences in R-code.

Several 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.3D` integrates 3-dimensional problems.

As an example, consider the Aphid model described in **?**. 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 **?** 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, names = "Aphid")
+ ))

   user  system elapsed
   0.04    0.00    0.03
```

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
```

The `summary` method gives the mean, min, max, ... of the entire 1-D variable:

```
> summary(out)

               Aphid
Min.    0.000000e+00
1st Qu. 1.705000e-03
Median  4.051000e-02
Mean    1.062000e-01
3rd Qu. 1.931000e-01
Max.    1.000000e+00
N       1.206000e+04
sd      1.303048e-01
```

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 this is a 1-D model, it is best solved with **deSolve** function `ode.1D`. A multi-species IVP example can be found in **?**. For 2-D and 3-D problems, we refer to the help-files of functions `ode.2D` and `ode.3D`.

The output of one-dimensional models can also be plotted using S3-method `plot.1D` and `matplot.1D`. In both cases, we can simply take a `subset` of the output, and add observations.

```
> data <- cbind(dist  = c(0,10, 20,  30,  40, 50, 60),
+               Aphid = c(0,0.1,0.25,0.5,0.25,0.1,0))

> par (mfrow = c(1,2))
> matplot.1D(out, grid = Distance, type = "l", mfrow = NULL,
+     subset = time %in% seq(0, 200, by = 10),
+    obs = data, obspar = list(pch = 18, cex = 2, col="red"))
> plot.1D(out, grid = Distance, type = "l", mfrow = NULL,
+     subset = time == 100,
+    obs = data, obspar = list(pch = 18, cex = 2, col="red"))
```

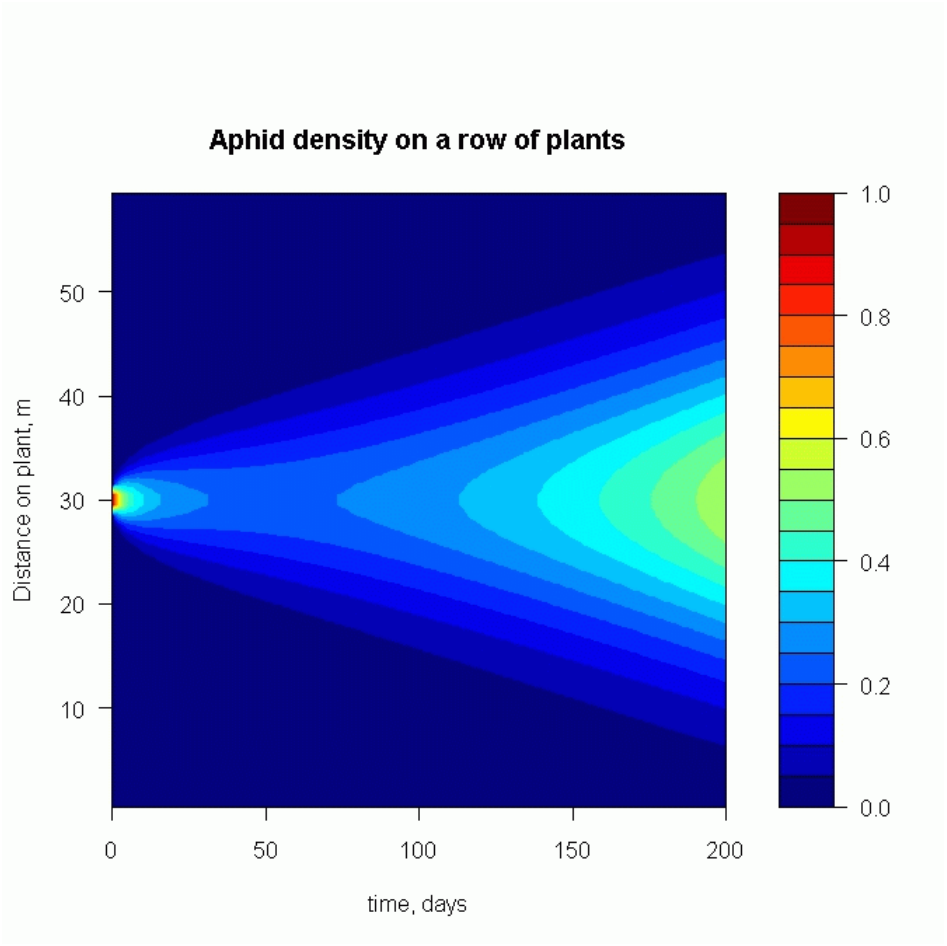

Figure 2: Solution of the 1-dimensional aphid model - see text for R -code

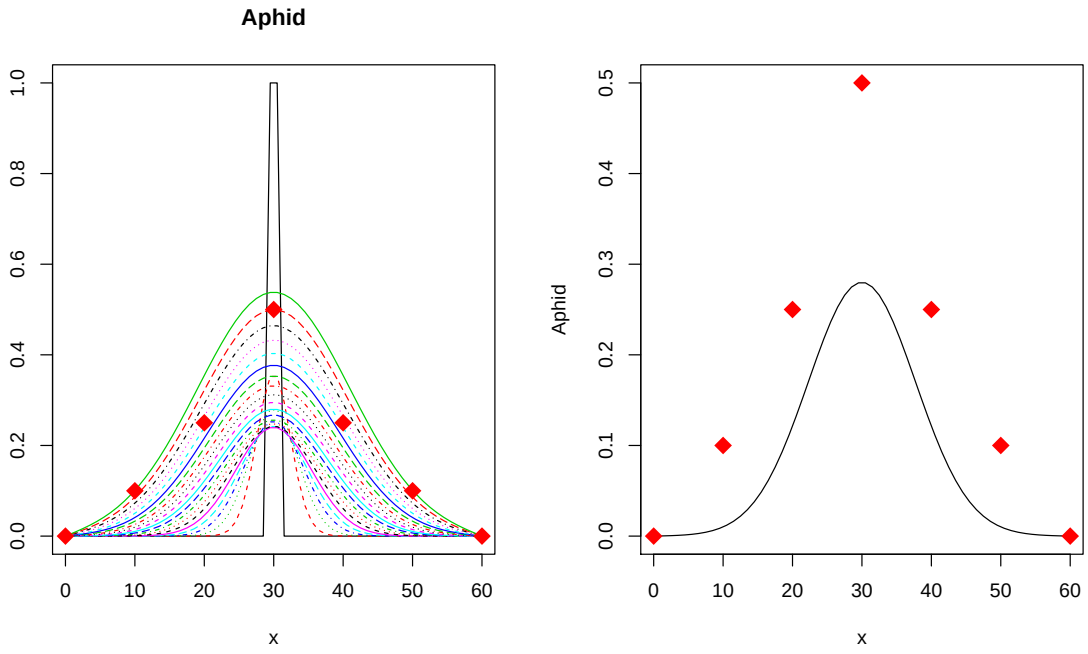

Figure 3: Solution of the Aphid model - plotted with matplot.1D, plot.1D - see text for R-code

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 (**?**),
- `daspk`, based on the backward differentiation code DASPK (**?**).

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
> 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

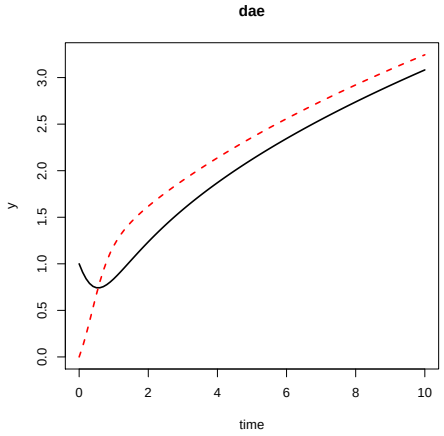

Figure 4: Solution of the differential algebraic equation model - see text for R-code

```
   user  system elapsed
      0       0       0
```

```
> 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 $Mdy/dt = f(t, y)$.

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)
```

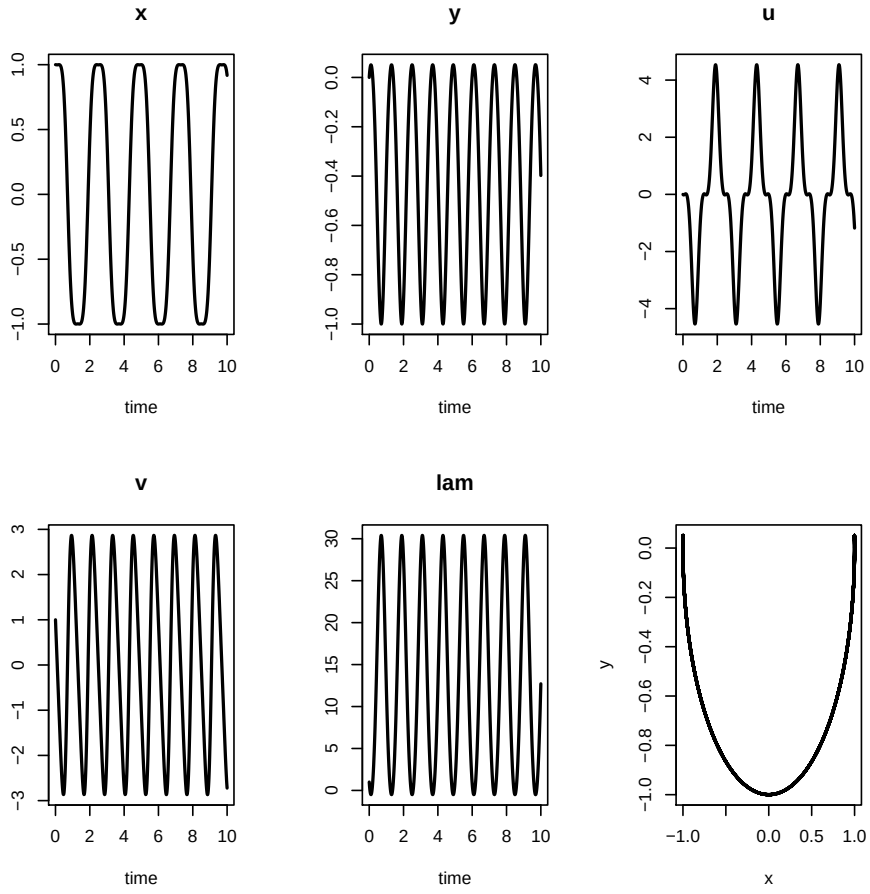

Figure 5: Solution of the pendulum problem, an index 3 differential algebraic equation using `radau` - see text for R-code

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 and roots

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.

Several integrators (`lsoda`, `lsodar`, `lsode`, `lsodes` and `radau`) can estimate the root of one or more functions. For the first 4 integration methods, the root finding algorithm is based on the algorithm in solver LSODAR, and implemented in FORTRAN. For `radau`, the root solving algorithm is written in C-code, and it works slightly different. Thus, some problems involving roots may be more efficient to solve with either `lsoda, lsode`, or `lsodes`, while other problems are more efficiently solved with `radau`.

If a root is found, then the integration will be terminated, unless an event function is defined.

A help file with information on roots and events can be opened by typing `?events` or `?roots`.

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 `ode`:

```
> 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:

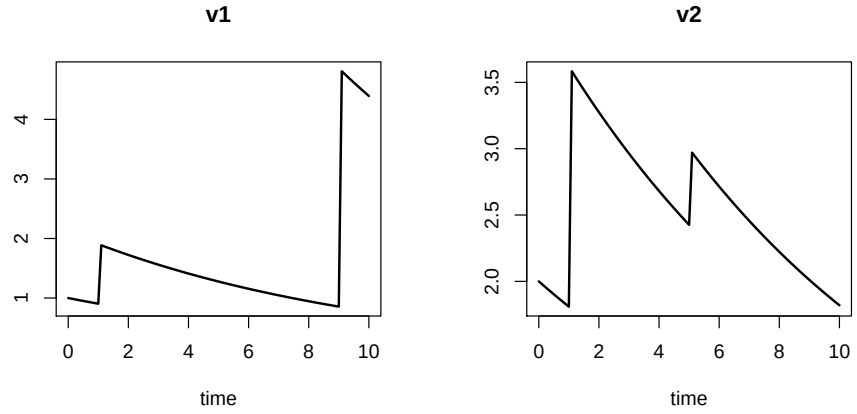

Figure 6: A simple model that contains events

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

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, here using `lsode`.

```
> 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")
```

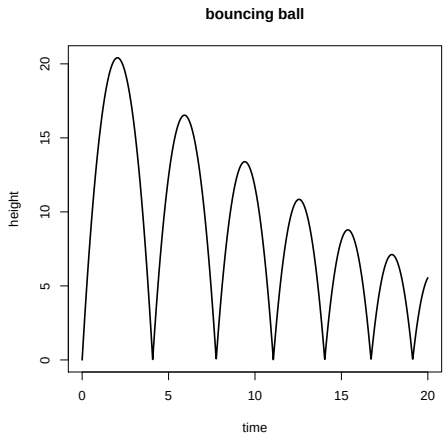

Figure 7: A model, with event triggered by a root function

7.3. Events and time steps

The use of events requires that all event times are contained in the output time steps, otherwise such events would be skipped. This sounds easy but sometimes problems can occur due to the limited accuracy of floating point arithmetics of the computer. To make things work as excpected, two requirements have to be fulfilled:

1. all event times have to be contained **exactly** in times, i.e. with the maximum possible accuracy of floating point arithmetics.
2. two time steps should not be too close together, otherwise numerical problems would occur during the integration.

Starting from version 1.10 of **deSolve** this is now checked (and if necessary also fixed) automatically by the solver functions. A warning is issued to inform the user about possible problems, especially that the output time steps were now adjusted and therefore different from the ones originally specified by the user. This means that all values of `eventtimes` are now contained but only the subset of times that have no exact or "rather close" neighbors in `eventtimes`.

Instead of relying on this automatism, matching times and eventtimes can also be managed by the user, either by appropriate rounding or by using function `cleanEventTimes` shown below.

Let's assume we have a vector of time steps `times` and another vector of event times `eventtimes`:

```
> times      <- seq(0, 1, 0.1)
> eventtimes <- c(0.7, 0.9)
```

If we now check whether the `eventtimes` are in `times`:

```
> eventtimes %in% times
```

```
[1] FALSE  TRUE
```

we get the surprising answer that this is only partly the case, because `seq` made small numerical errors. The easiest method to get rid of this is rounding:

```
> times2 <- round(times, 1)
> times - times2

 [1] 0.000000e+00 0.000000e+00 0.000000e+00 5.551115e-17 0.000000e+00
 [6] 0.000000e+00 1.110223e-16 1.110223e-16 0.000000e+00 0.000000e+00
[11] 0.000000e+00
```

The last line shows us that the error was always smaller than, say 10^{-15}, what is typical for ordinary double precision arithmetics. The accuracy of the machine can be determined with `.Machine$double.eps`.

To check if all `eventtimes` are now contained in the new times vector `times2`, we use:

```
> eventtimes %in% times2

[1] TRUE TRUE
```

or

```
> all(eventtimes %in% times2)

[1] TRUE
```

and see that everything is o.k. now.

In few cases, rounding may not work properly, for example if a pharmacokinetic model is simulated with a daily time step, but drug injection occurs at precisely fixed times within the day. Then one has to add all additional event times to the ordinary time stepping:

```
> times <- 1:10
> eventtimes <- c(1.3, 3.4, 4, 7.9, 8.5)
> newtimes <- sort(unique(c(times, eventtimes)))
```

If, however, an event and a time step are almost (but not exactly) the same, then it is more safe to use:

```
> times <- 1:10
> eventtimes <- c(1.3, 3.4, 4, 7.9999999999999999, 8.5)
> newtimes <- sort(c(eventtimes, cleanEventTimes(times, eventtimes)))
```

because `cleanEventTimes` removes not only the doubled 4 (like `unique`, but also the "almost doubled" 8, while keeping the exact event time. The tolerance of `cleanEventTimes` can be adjusted using an optional argument `eps`.

As said, this is normally done automatically by the differential equation solvers and in most cases appropriate rounding will be sufficient to get rid of the warnings.

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 (**?**).

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`.

```
> library(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)
```

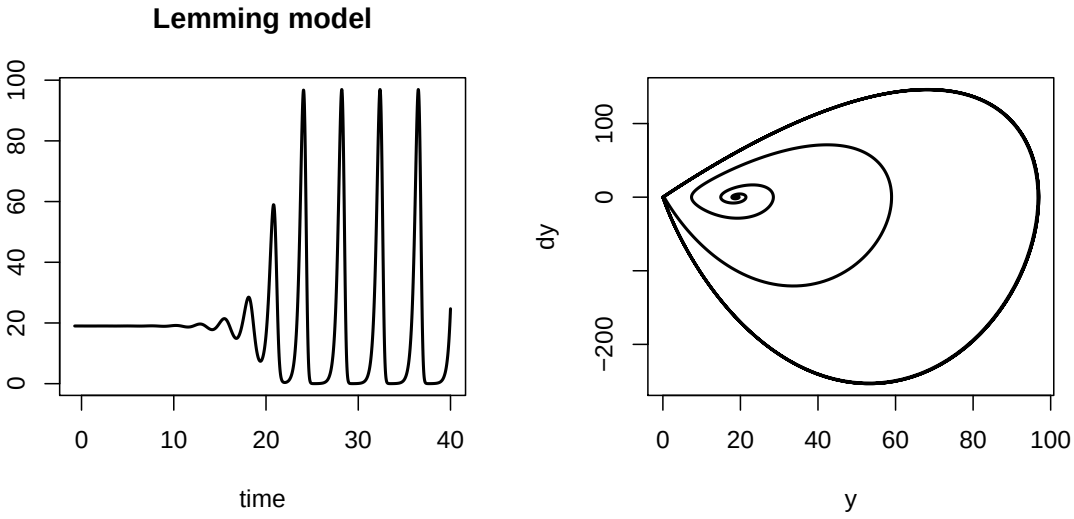

Figure 8: A delay differential equation model

9. Discrete time models, difference equations

There is one special-purpose solver, triggered with `method = "iteration"` which can be used in cases where the new values of the state variables are directly estimated by the user, and need not be found by numerical integration.

This is for instance useful when the model consists of difference equations, or for 1-D models when transport is implemented by an implicit or a semi-implicit method.

We give here an example of a discrete time model, represented by a difference equation: the Teasel model as from **?**, p287.

The dynamics of this plant is described by 6 stages and the transition from one stage to another is in a transition matrix:

We define the stages and the transition matrix first:

```
> Stages <- c("DS 1yr", "DS 2yr", "R small", "R medium", "R large", "F")
> NumStages   <- length(Stages)
> # Population matrix
> A <- matrix(nrow = NumStages, ncol = NumStages, byrow = TRUE, data = c(
+                     0,      0,      0,      0,      0,      322.38,
+                     0.966,  0,      0,      0,      0,      0     ,
+                     0.013,  0.01,   0.125,  0,      0,      3.448 ,
+                     0.007,  0,      0.125,  0.238,  0,      30.170,
+                     0.008,  0,      0.038,  0.245,  0.167,  0.862 ,
+                     0,      0,      0,      0.023,  0.75,   0      )  )
```

The difference function is defined as usual, but does not return the "rate of change" but rather the new relative stage densities are returned. Thus, each time step, the updated values are divided by the summed densities:

```
> Teasel <- function (t, y, p) {
+   yNew <-  A %*% y
+   list (yNew / sum(yNew))
+ }
```

The model is solved using method "iteration":

```
> out <- ode(func = Teasel, y = c(1, rep(0, 5) ), times = 0:50,
+            parms = 0, method = "iteration")
```

and plotted using R-function `matplot`:

```
> matplot(out[,1], out[,-1], main = "Teasel stage distribution", type = "l")
> legend("topright", legend = Stages, lty = 1:6, col = 1:6)
```

10. Plotting deSolve Objects

There are S3 `plot` and `image` methods for plotting 0-D (plot), and 1-D and 2-D model output (image) as generated with `ode`, `ode.1D`, `ode.2D`.

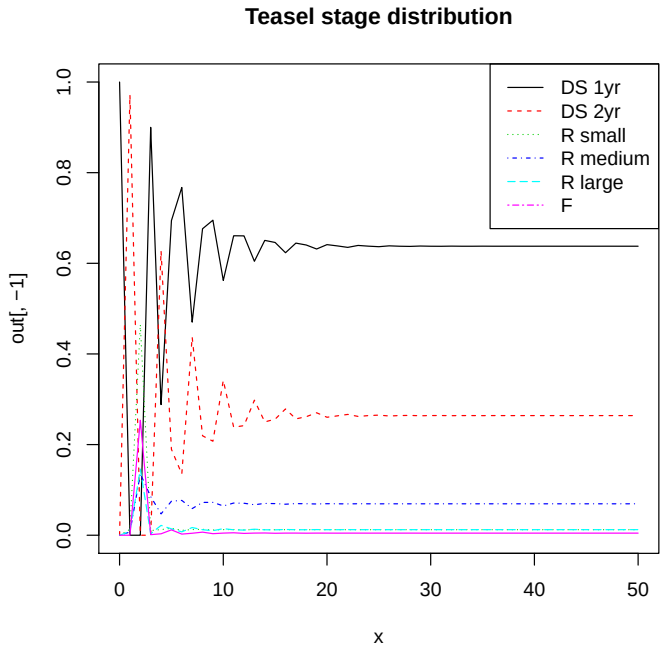

Figure 9: A difference model solved with method = "iteration"

How to use it and examples can be found by typing `?plot.deSolve`.

10.1. Plotting Multiple Scenario's

The `plot` method for `deSolve` objects can also be used to compare different scenarios, e.g from the same model but with different sets of parameters or initial values, with one single call to `plot`.

As an example we implement the simple combustion model, which can be found on http://www.scholarpedia.org/article/Stiff_systems:

$$y' = y^2 \cdot (1 - y)$$

The model is run with 4 different values of the initial conditions: $y = 0.01, 0.02, 0.03, 0.04$ and written to `deSolve` objects `out`, `out2`, `out3`, `out4`.

```
> library(deSolve)
> combustion <- function (t, y, parms)
+   list(y^2 * (1-y) )

> yini  <- 0.01
> times <- 0 : 200

> out  <- ode(times = times, y = yini,   parms = 0, func = combustion)
> out2 <- ode(times = times, y = yini*2, parms = 0, func = combustion)
```

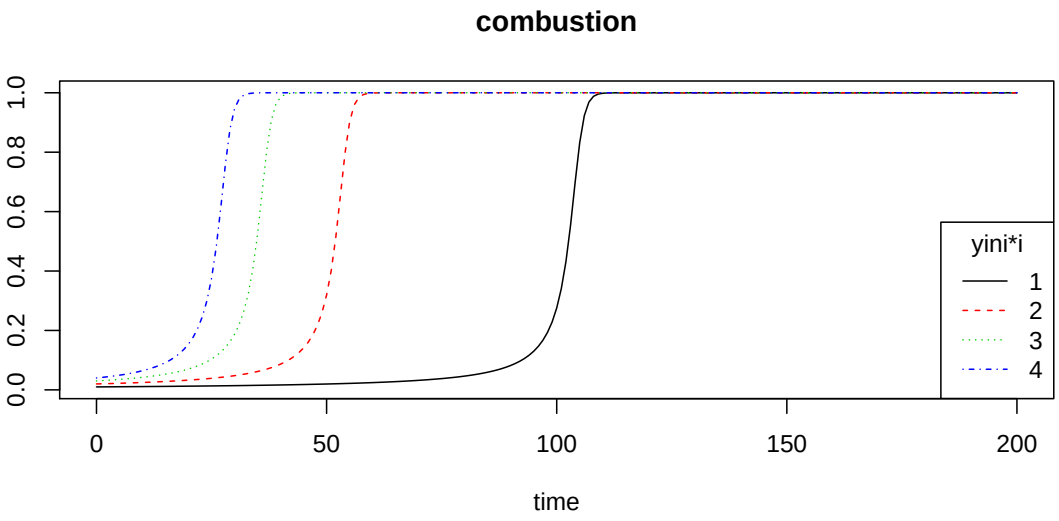

Figure 10: Plotting 4 outputs in one figure

```
> out3 <- ode(times = times, y = yini*3, parms = 0, func = combustion)
> out4 <- ode(times = times, y = yini*4, parms = 0, func = combustion)
```

The different scenarios are plotted at once, and a suitable legend is written.

```
> plot(out, out2, out3, out4, main = "combustion")
> legend("bottomright", lty = 1:4, col = 1:4, legend = 1:4, title = "yini*i")
```

10.2. Plotting Output with Observations

With the help of the optional argument `obs` it is possible to specify observed data that should be added to a `deSolve` plot.

We exemplify this using the `ccl4model` in package `deSolve`. (see `?ccl4model` for what this is about). This model example has been implemented in compiled code. An observed data set is also available, called `ccl4data`. It contains toxicant concentrations in a chamber where rats were dosed with CCl4.

```
> head(ccl4data)
```

```
   time initconc animal ChamberConc
1 0.083     1000      A    828.4376
2 0.167     1000      A    779.6795
3 0.333     1000      A    713.8045
4 0.500     1000      A    672.0502
5 0.667     1000      A    631.9522
6 0.833     1000      A    600.6975
```

We select the data from animal "A":

```
> obs <- subset (ccl4data, animal == "A", c(time, ChamberConc))
> names(obs) <- c("time", "CP")
> head(obs)

   time       CP
1 0.083 828.4376
2 0.167 779.6795
3 0.333 713.8045
4 0.500 672.0502
5 0.667 631.9522
6 0.833 600.6975
```

After assigning values to the parameters and providing initial conditions, the `ccl4model` can be run. We run the model three times, each time with a different value for the first parameter. Output is written to matrices `out` `out2`, and `out3`.

```
> parms <- c(0.182, 4.0, 4.0, 0.08, 0.04, 0.74, 0.05, 0.15, 0.32, 16.17,
+             281.48, 13.3, 16.17, 5.487, 153.8, 0.04321671,
+             0.40272550, 951.46, 0.02, 1.0,  3.80000000)
> yini <- c(AI = 21, AAM = 0, AT = 0, AF = 0, AL = 0, CLT = 0, AM = 0)
> out <- ccl4model(times = seq(0, 6, by = 0.05), y = yini, parms = parms)
> par2 <- parms
> par2[1] <- 0.1
> out2 <- ccl4model(times = seq(0, 6, by = 0.05), y = yini, parms = par2)
> par3 <- parms
> par3[1] <- 0.05
> out3 <- ccl4model(times = seq(0, 6, by = 0.05), y = yini, parms = par3)
```

We plot all these scenarios and the observed data at once:

```
> plot(out, out2, out3, which = c("AI", "MASS", "CP"),
+      col = c("black", "red", "green"), lwd = 2,
+      obs = obs, obspar = list(pch = 18, col = "blue", cex = 1.2))
> legend("topright", lty = c(1,2,3,NA), pch = c(NA, NA, NA, 18),
+   col = c("black", "red", "green", "blue"), lwd = 2,
+   legend = c("par1", "par2", "par3", "obs"))
```

If we do not select specific variables, then only the ones for which there are observed data are plotted. Assume we have measured the total mass at the end of day 6. We put this in a second data set:

```
> obs2 <- data.frame(time = 6, MASS = 12)
> obs2

  time MASS
1    6   12
```

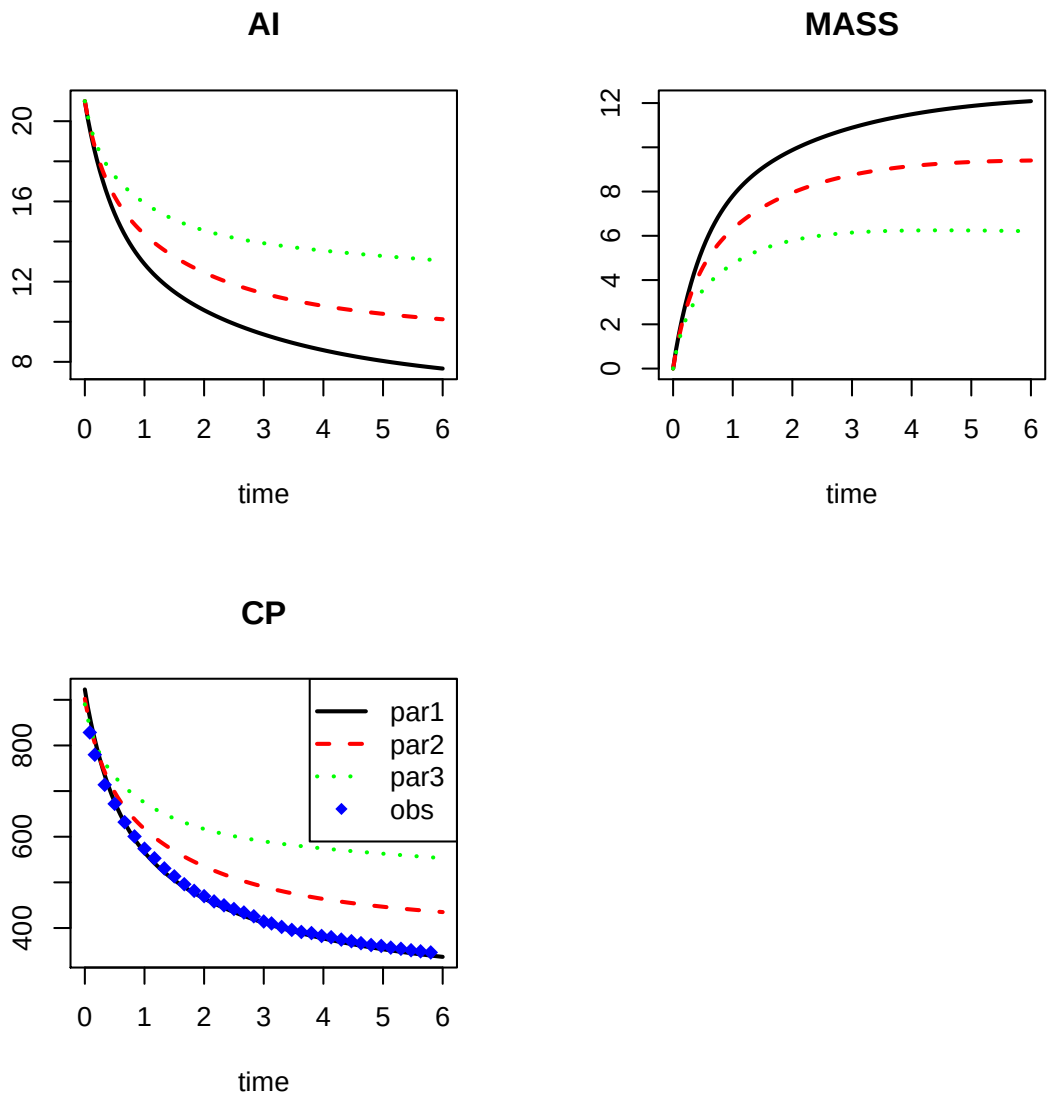

Figure 11: Plotting output and observations in one figure

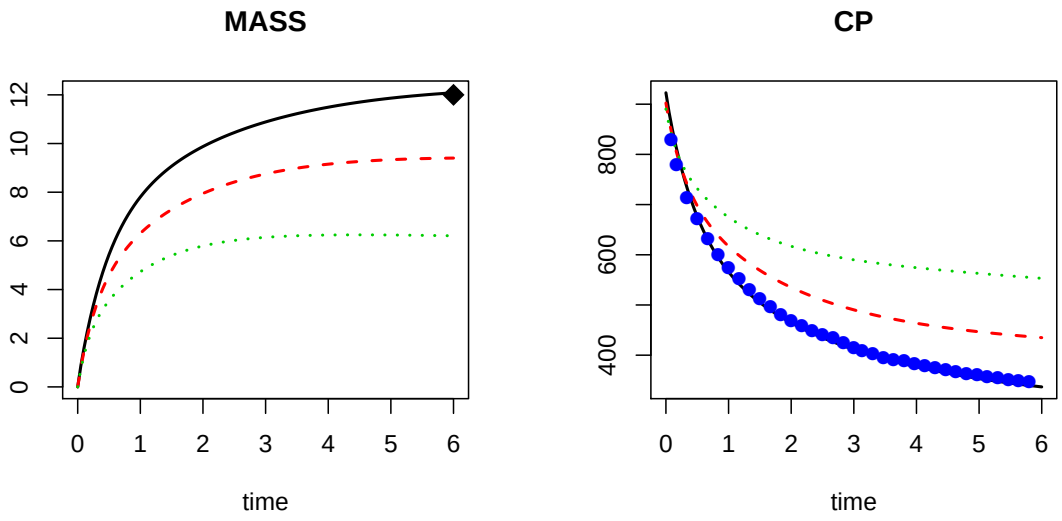

Figure 12: Plotting variables in common with observations

then we plot the data together with the three model runs as follows:

```
> plot(out, out2, out3, lwd = 2,
+      obs = list(obs, obs2),
+      obspar = list(pch = c(16, 18), col = c("blue", "black"),
+                    cex = c(1.2 , 2))
+      )
```

10.3. Plotting Summary Histograms

The `hist` function plots the histogram for each variable; all plot parameters can be set individually (here for `col`).

To generate the next plot, we overrule the default `mfrow` setting which would plot the figures in 3 rows and 3 columns (and hence plot one figure in isolation)

```
> hist(out, col = grey(seq(0, 1, by = 0.1)), mfrow = c(3, 4))
```

10.4. Plotting multi-dimensional output

The `image` function plots time versus x images for models solved with `ode.1D`, or generates x-y plots for models solved with `ode.2D`.

1-D model output

We exemplify its use by means of a Lotka-Volterra model, implemented in 1-D. The model describes a predator and its prey diffusing on a flat surface and in concentric circles. This is a 1-D model, solved in the cylindrical coordinate system.

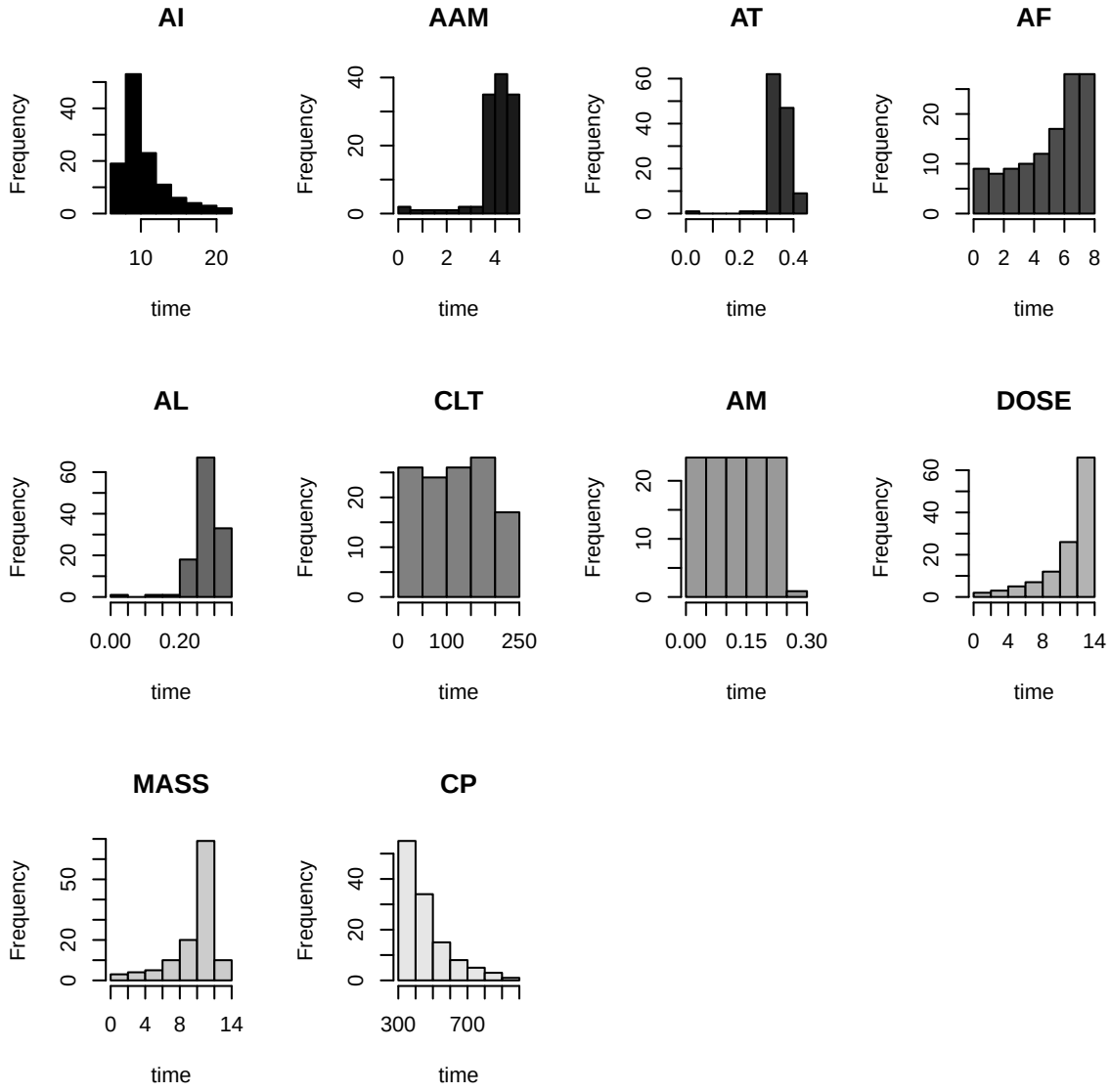

Figure 13: Plotting histograms of all output variables

Note that it is simpler to implement this model in R-package `ReacTran` (**?**).
We start by defining the derivative function

```
lvmod <- function (time, state, parms, N, rr, ri, dr, dri) {
  with (as.list(parms), {
    PREY <- state[1:N]
    PRED <- state[(N+1):(2*N)]

    ## Fluxes due to diffusion
    ## at internal and external boundaries: zero gradient
    FluxPrey <- -Da * diff(c(PREY[1], PREY, PREY[N]))/dri
    FluxPred <- -Da * diff(c(PRED[1], PRED, PRED[N]))/dri

    ## Biology: Lotka-Volterra model
    Ingestion     <- rIng  * PREY * PRED
    GrowthPrey    <- rGrow * PREY * (1-PREY/cap)
    MortPredator  <- rMort * PRED

    ## Rate of change = Flux gradient + Biology
    dPREY    <- -diff(ri * FluxPrey)/rr/dr   +
                GrowthPrey - Ingestion
    dPRED    <- -diff(ri * FluxPred)/rr/dr   +
                Ingestion * assEff - MortPredator

    return (list(c(dPREY, dPRED)))
  })
}
```

Then we define the parameters, which we put in a list

```
R  <- 20                        # total radius of surface, m
N  <- 100                       # 100 concentric circles
dr <- R/N                       # thickness of each layer
r  <- seq(dr/2,by = dr,len = N) # distance of center to mid-layer
ri <- seq(0,by = dr,len = N+1)  # distance to layer interface
dri <- dr                       # dispersion distances
parms <- c(Da     = 0.05,       # m2/d, dispersion coefficient
           rIng   = 0.2,        # /day, rate of ingestion
           rGrow  = 1.0,        # /day, growth rate of prey
           rMort  = 0.2 ,       # /day, mortality rate of pred
           assEff = 0.5,        # -, assimilation efficiency
           cap    = 10)         # density, carrying capacity
```

After defining initial conditions, the model is solved with routine `ode.1D`

```
state    <- rep(0, 2 * N)
state[1] <- state[N + 1] <- 10
```

```
times  <- seq(0, 200, by = 1)   # output wanted at these time intervals
print(system.time(
  out <- ode.1D(y = state, times = times, func = lvmod, parms = parms,
                nspec = 2, names = c("PREY", "PRED"),
                N = N, rr = r, ri = ri, dr = dr, dri = dri)
))
```

```
   user  system elapsed
   0.34    0.00    0.34
```

The `summary` method provides summaries for both 1-dimensional state variables:

```
summary(out)
```

```
                PREY          PRED
Min.        0.000000      0.000000
1st Qu.     1.997000      3.971000
Median      2.000000      4.000000
Mean        2.094000      3.333000
3rd Qu.     2.000000      4.000000
Max.       10.000000     10.000000
N       20100.000000 20100.000000
sd          1.648847      1.526742
```

while the S3-method `subset` can be used to extract only specific values of the variables:

```
p10 <- subset(out, select = "PREY", subset = time == 10)
head(p10, n = 5)
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
[1,] 6.304707 6.436374 6.687753 7.033897 7.436098 7.843497 8.198683
        [,8]     [,9]    [,10]    [,11]    [,12]    [,13]    [,14]
[1,] 8.44655 8.542749 8.457464 8.173474 7.682188 6.983525 6.09314
        [,15]    [,16]    [,17]    [,18]    [,19]     [,20]    [,21]
[1,] 5.054634 3.947258 2.876796 1.946159 1.22074 0.7120999 0.388549
         [,22]       [,23]       [,24]       [,25]       [,26]
[1,] 0.1996948 0.09733824 0.04526957 0.02018686 0.00866459
            [,27]        [,28]        [,29]        [,30]        [,31]
[1,] 0.003590384 0.001439632 0.0005595871 0.0002111643 7.745135e-05
             [,32]        [,33]        [,34]        [,35]        [,36]
[1,] 2.763954e-05 9.605149e-06 3.252992e-06 1.074404e-06 3.462836e-07
             [,37]        [,38]        [,39]        [,40]        [,41]
[1,] 1.089753e-07 3.350363e-08 1.006801e-08 2.958638e-09 8.506159e-10
             [,42]        [,43]        [,44]        [,45]        [,46]
[1,] 2.393626e-10 6.595368e-11 1.780133e-11 4.708254e-12 1.220725e-12
             [,47]        [,48]        [,49]        [,50]        [,51]
[1,] 3.103673e-13 7.740666e-14 1.894361e-14 4.550522e-15 1.07325e-15
```

```
            [,52]        [,53]        [,54]        [,55]        [,56]
[1,] 2.486015e-16 5.657001e-17 1.264912e-17 2.779928e-18 6.006319e-19
            [,57]        [,58]        [,59]        [,60]        [,61]
[1,] 1.276096e-19 2.666569e-20 5.481592e-21 1.108743e-21 2.207026e-22
            [,62]        [,63]        [,64]        [,65]        [,66]
[1,] 4.324292e-23 8.341186e-24 1.584226e-24 2.963127e-25 5.458728e-26
            [,67]        [,68]        [,69]        [,70]       [,71]
[1,] 9.906072e-27 1.771079e-27 3.120002e-28 5.416316e-29 9.26688e-30
            [,72]        [,73]        [,74]       [,75]        [,76]
[1,] 1.562748e-30 2.597843e-31 4.257409e-32 6.87897e-33 1.095928e-33
            [,77]        [,78]        [,79]        [,80]       [,81]
[1,] 1.721681e-34 2.667263e-35 4.075197e-36 6.140815e-37 9.12685e-38
            [,82]        [,83]       [,84]        [,85]        [,86]
[1,] 1.337994e-38 1.934836e-39 2.75999e-40 3.883822e-41 5.391532e-42
            [,87]        [,88]        [,89]        [,90]        [,91]
[1,] 7.383755e-43 9.976169e-44 1.329779e-44 1.748762e-45 2.268934e-46
            [,92]        [,93]        [,94]        [,95]        [,96]
[1,] 2.904397e-47 3.668048e-48 4.570454e-49 5.618603e-50 6.814592e-51
            [,97]        [,98]        [,99]       [,100]
[1,] 8.154381e-52 9.626882e-53 1.122896e-53 1.436052e-54
```

We first plot both 1-dimensional state variables at once; we specify that the figures are arranged in two rows, and 2 columns; when we call `image`, we overrule the default mfrow setting (`mfrow = NULL`). Next we plot "PREY" again, once with the default xlim and ylim, and next zooming in. Note that xlim and ylim are a list here. When we call `image` for the second time, we overrule the default `mfrow` setting by specifying (`mfrow = NULL`).

```
image(out, grid = r, mfrow = c(2, 2), method = "persp", border = NA,
      ticktype = "detailed", legend = TRUE)
image(out, grid = r, which = c("PREY", "PREY"), mfrow = NULL,
      xlim = list(NULL, c(0, 10)), ylim = list(NULL, c(0, 5)),
      add.contour = c(FALSE, TRUE))
```

2-D model output

When using `image` with a 2-D model, then the 2-D values at all output times will be plotted.

Sometimes we want only output at a specific time value. We then use S3-method `subset` to extract 2-D variables at suitable time-values and use R's `image`, `filled.contour` or `contour` method to depict them.

Consider the very simple 2-D model (100*100), containing just 1-st order consumption, at a rate `r_x2y2`, where `r_x2y2` depends on the position along the grid. First the derivative function is defined:

```
Simple2D <- function(t, Y, par) {
  y  <- matrix(nrow = nx, ncol = ny, data = Y)  # vector to 2-D matrix
  dY <- - r_x2y2 * y                            # consumption
```

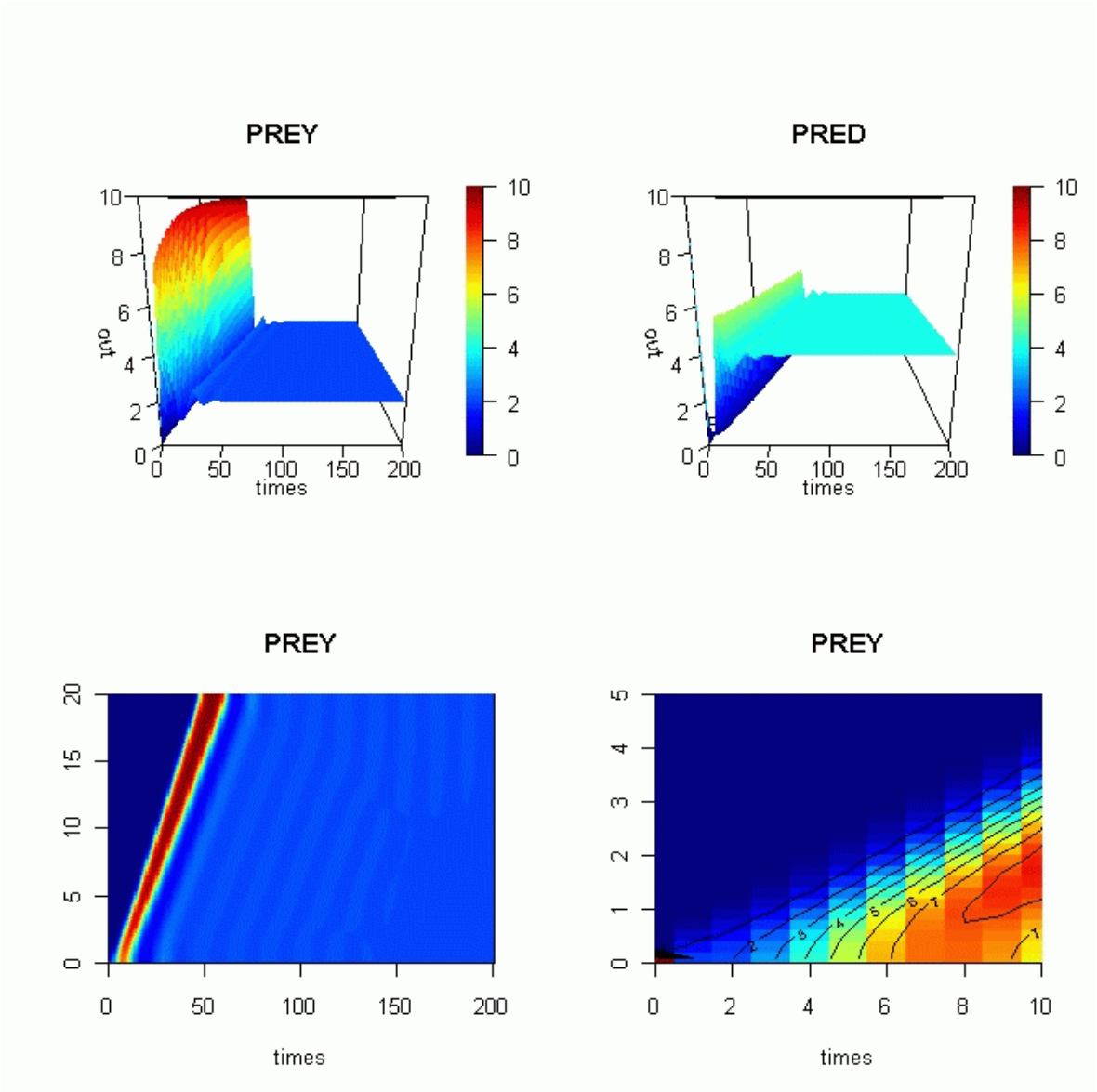

Figure 14: image plots

```
  return(list(dY))
}
```

Then the grid is created, and the consumption rate made a function of grid position (`outer`).

```
dy <- dx <- 1  # grid size
nx <- ny <- 100
x  <- seq (dx/2, by = dx, len = nx)
y  <- seq (dy/2, by = dy, len = ny)
# in each grid cell: consumption depending on position
r_x2y2 <- outer(x, y, FUN=function(x,y) ((x-50)^2 + (y-50)^2)*1e-4)
```

After defining the initial values, the model is solved using solver `ode.2D`. We use Runge-Kutta method `ode45`.

```
C <- matrix(nrow = nx, ncol = ny, 1)
ODE3 <- ode.2D(y = C, times = 1:100, func = Simple2D, parms = NULL,
               dimens = c(nx, ny), names = "C", method = "ode45")
```

We print a summary, and extract the 2-D variable at `time = 50`

```
summary(ODE3)
```

```
                  C
Min.    8.523000e-22
1st Qu. 4.332000e-06
Median  2.631000e-03
Mean    1.312000e-01
3rd Qu. 1.203000e-01
Max.    1.000000e+00
N       1.000000e+06
sd      2.489394e-01
```

```
t50 <-  matrix(nrow = nx, ncol = ny,
     data = subset(ODE3, select = "C", subset = (time == 50)))
```

We use function `contour` to plot both the consumption rate and the values of the state variables at `time = 50`.

```
par(mfrow = c(1, 2))
contour(x, y,  r_x2y2, main = "consumption")
contour(x, y, t50, main = "Y(t = 50)")
```

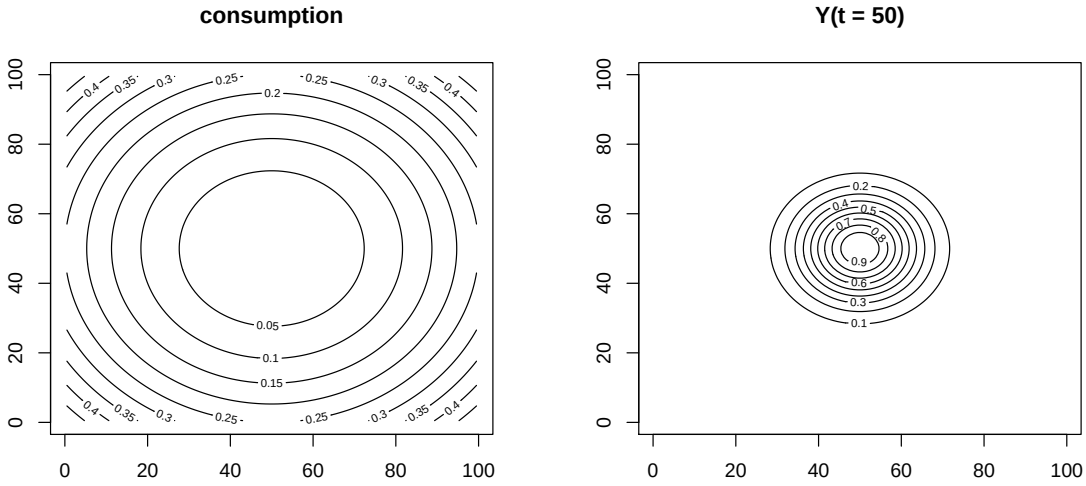

Figure 15: Contour plot of 2-D variables

11. Troubleshooting

11.1. 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:

```
PCmod <- 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 = 10, 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, 200, 0.1)
out <- ode(y = xstart, times = times,
  func = PCmod, parms = parms)
tail(out)
```

```
        time   P   C
[1996,] 199.5 NaN NaN
[1997,] 199.6 NaN NaN
[1998,] 199.7 NaN NaN
[1999,] 199.8 NaN NaN
[2000,] 199.9 NaN NaN
[2001,] 200.0 NaN NaN
```

We see that the simulation was stopped before reaching the final simulation time and both producers and consumer values may have negative values.

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 infinite or even NaNs (not a number). This is because the model equations are not intended to be used with negative numbers, as negative concentrations are not realistic.

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

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 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 = PCmod,
                          parms = parms, atol = 0)
matplot(out[,1], out[,2:3], type = "l",
        xlab = "time", ylab = "Producer, Consumer")
```

It is, of course, not possible to set both, `atol` and `rtol` simultaneously to zero. As we see from 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.

11.2. 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`.

11.3. 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 message, R1 = 53.4272, R2 = 2.44876e-15

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 = 53.4272

Warning messages:
1: In lsoda(y, times, func, parms, ...) :
  an excessive amount of work (> maxsteps ) was done,
  but integration was not successful - increase maxsteps
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 = 53.4272, the solver used a step size H = 2.44876e-15. 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, as 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 = 53.4272, maxstep = 5000 steps have been done.

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

Simply increasing maxsteps will not work and it makes more sense to first see if the output tells what happens:

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

You may, of course, consider to use another solver:

```
pars["rIng"] <- 100
out3 <- ode(func = LVmod, y = yini, parms = pars,
          times = times, method = "ode45", atol = 1e-14, rtol = 1e-14)
```

but don't forget to think about this too and, for example, increase simulation time to 1000 and try different values of `atol` and `rtol`. We leave this open as an exercise to the reader.

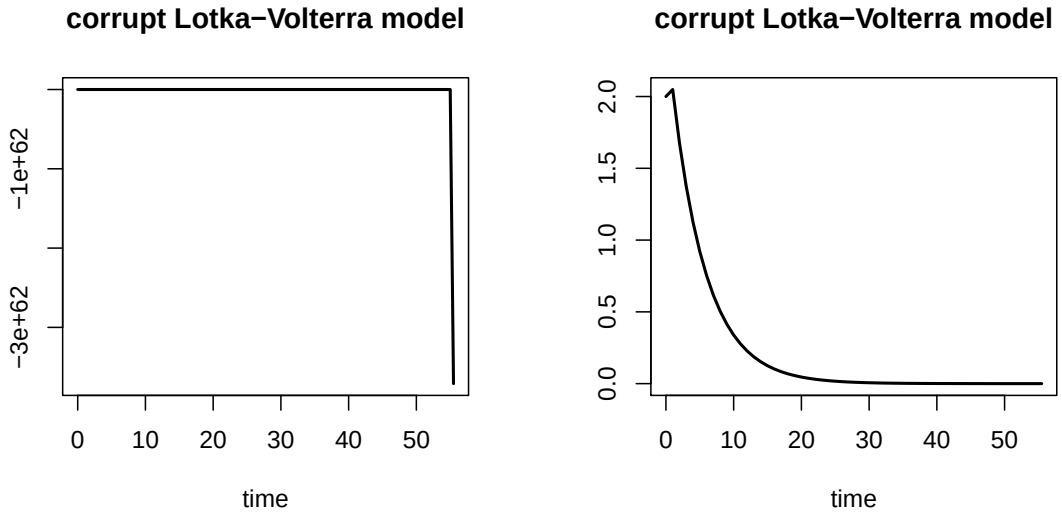

Figure 16: A model that cannot be solved correctly

Affiliation:

Karline Soetaert
Centre for Estuarine and Marine Ecology (CEME)
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: karline.soetaert@nioz.nl
URL: http://www.nioz.nl

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; includes a root solving procedure
`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. Note that, as from version 1.7, `lsodes` includes a root finding procedure, similar to `lsodar`
`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`)