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** (**?**), 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).

The implementation includes stiff integration routines based on the ODEPACK Fortan codes (**?**). It also include Runge-Kutta solvers and the Euler method (**?**).

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.

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:

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

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

1. 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)
```

2. 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)
```

3. 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 as 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 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

1. 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)
```

2. 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 a matrix that contains the values of the state variables (columns) at the requested output times. The output is converted to a data frame and stored in '`out`'. Data frames have the advantage, that their columns can be accessed by name, rather than by number. For instance, `out$X` will take the outputted values of state variable `X`, and so on.

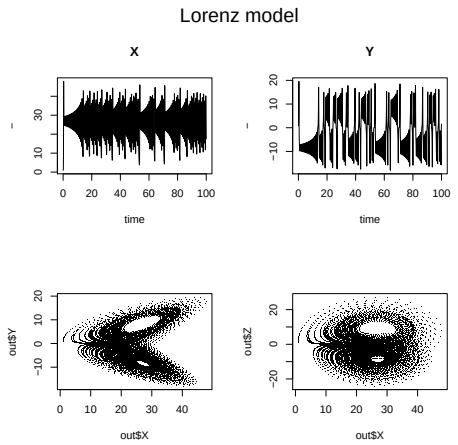

Figure 1: Solution of the ordinary differential equation - see text for R-code

```
> require(deSolve)
> out <- as.data.frame(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
```

3. Plotting results

Finally, the model output is plotted.

The figures are arranged in two rows and two columns (`mfrow`), and the size of the outer upper margin (the third margin) is increased (`oma`), such as to allow writing a figure heading (`mtext`). First the X concentration versus time is plotted, then the Y concentration versus time, and finally Y versus X and Z versus X.

```
> par(mfrow=c(2,2), oma=c(0,0,3,0))
> plot (times,out$X ,type="l",main="X", xlab="time", ylab="-")
> plot (times,out$Y ,type="l",main="Y", xlab="time", ylab="-")
> plot (out$X,out$Y, pch=".")
> plot (out$X,out$Z, pch=".")
> mtext(outer=TRUE,side=3,"Lorenz model",cex=1.5)
```

1.3. Solvers for initial value problems of ordinary differential equations

Package **deSolve** contains several IVP ordinary differential equation solvers. They can all 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.

Thus it should be possible to find, for one particular problem, the most efficient solver. 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 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(out <-rk4   (state,times,Lorenz,parameters)))

   user  system elapsed
   3.06    0.00    3.12

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

   user  system elapsed
   1.09    0.00    1.13

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

   user  system elapsed
   1.45    0.00    1.46

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

   user  system elapsed
   0.94    0.00    1.01

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

   user  system elapsed
   1.78    0.00    1.80

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

   user  system elapsed
   1.04    0.00    1.11
```

2. 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 many species,
- `ode.2D` integrates 2-dimensional problems,
- `ode.2D` integrates 2-dimensional problems.

As an example, consider the Aphid model described in **?**.

It is a model where aphids 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:

$$Flux_{diffusion} = -D\frac{\partial N}{\partial x}$$

with boundary conditions

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

and initial condition

$N_x = 0$ for $x \neq 30$

$N_x = 1$ 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.band(state,times,Aphid,parms=0,nspec=1)
+   ))

   user  system elapsed
   0.03    0.00    0.04
```

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 2.588637e-43 1.225264e-41 5.638675e-40 2.576711e-38
[3,]    2 2.502495e-37 9.091892e-36 3.159257e-34 1.078682e-32
[4,]    3 6.880643e-33 1.773667e-31 4.338516e-30 1.046752e-28
[5,]    4 1.045758e-29 2.167113e-28 4.193685e-27 7.961463e-26
[6,]    5 2.255197e-27 4.021606e-26 6.611523e-25 1.061826e-23

>   DENSITY   <- out[,2:(numboxes  +1)]
```

Finally, the output is plotted

```
>   filled.contour(x=times, y=Distance, DENSITY, color= topo.colors,
+                  xlab="time, days", ylab= "Distance on plant, m",
+                  main="Aphid density on a row of plants")
```

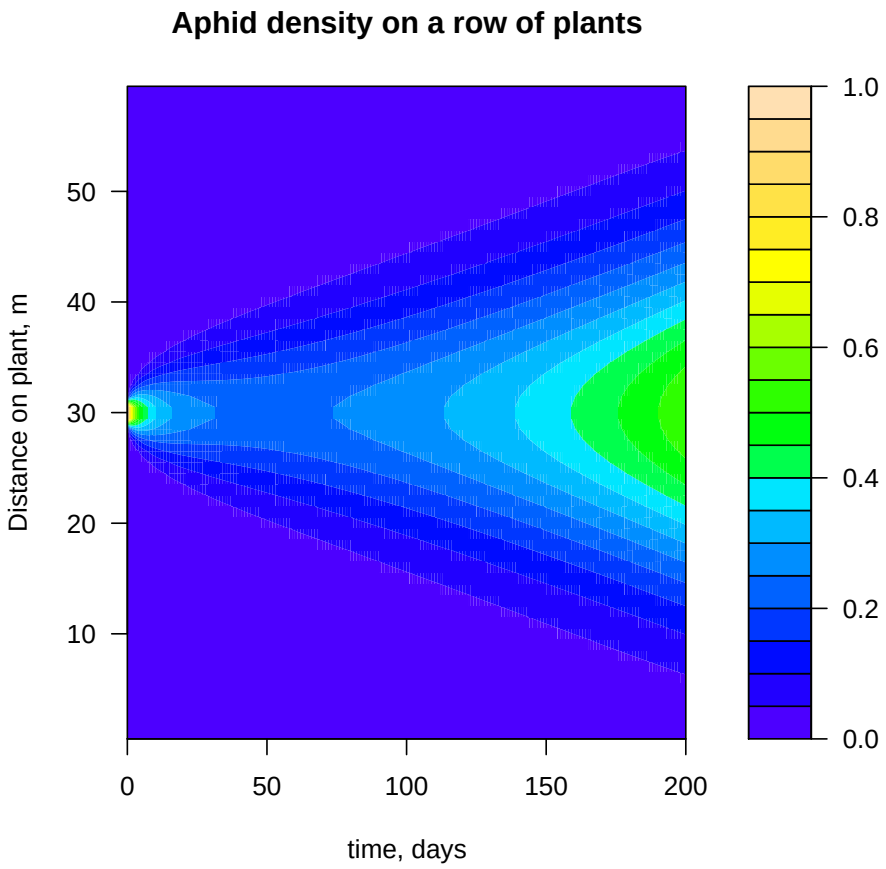

Figure 2: Solution of the 1-dimensional aphid model - see text for R -code

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

3. Differential algebraic equations

Function `daspk` from package **deSolve** solves (relatively simple) DAEs of index[1] 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))
+  }
> require(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)))

   user  system elapsed
   0.02    0.00    0.01

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

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

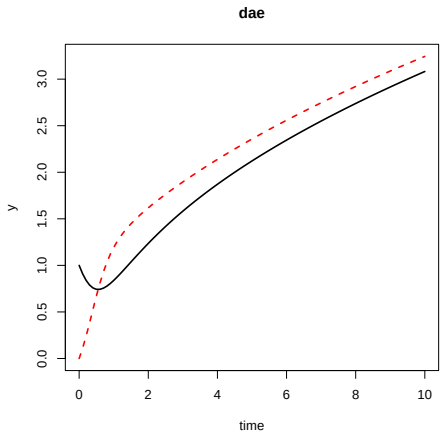

Figure 3: Solution of the differential algebraic equation model - see text for R-code

4. Using the integration options

The solvers from the ODEPACK integration routines can be optimised 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 subtantial time savingsL

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` to 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 a "non-stiff" (by setting `mf=10`), there is no need to specify the Jacobian.

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

5. A warning

The solvers from ODEPACK should be first choice for any problem. However, there are cases where they give very dubious results.

Consider the following Lotka-Volterra type of model:

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

and with the following (biologically not very realistic) [2] parameter values:

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

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

```
> xstart <- c(P = 0.5, K = 1)
> times  <- seq(0, 190, 0.1)
> out <- as.data.frame( ode(y = xstart,times = times,
+   func = lvmodel, parms = parms))
> tail(out)

      time   P   K
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 make sure that the model does not calculate with negative numbers, by taking the producer concentration to be at least = 0. [3] Thus,

[2] they are not realistic because consumers grow with 100 % efficiency

[3] note that this solution is what Karline Soetaert would do, Thomas Petzoldt does not like the idea, because of the mass balance violation it generates.

```
> lvmodel <- function(t, x, parms)  {
+   with(as.list(c(parms, x)), {
+     P <- max(0,P)
+     dP <- c*P  - d*K*P          # producer
+     dK <- e*P*K  - f*K          # consumer
+     res <- c(dP, dK)
+     list(res)
+   })
+ }
> out <- as.data.frame( ode(y = xstart,times = times,
+   func = lvmodel, parms = parms))
> tail(out)

      time             P            K
1896 189.5 -1.075252e-10 1.959310e-06
1897 189.6 -1.075252e-10 1.939814e-06
1898 189.7 -1.075252e-10 1.920513e-06
1899 189.8 -1.075252e-10 1.901403e-06
1900 189.9 -1.075252e-10 1.882484e-06
1901 190.0 -1.075252e-10 1.863753e-06
```

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 multicomponent 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
`daspk`	solves systems of 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
`lsodes`	integrates ODEs, using stiff method and assuming an arbitrary sparse Jacobian
`rk` or `rk4`	integrates ODEs, using Runge-Kutta methods
`euler`	integrates ODEs, using Euler's method

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