European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 6, no. 1-2, 2011, pp. 51-65
ISSN 1790–8140

Solving ODEs, DAEs, DDEs and PDEs in R[1]

Karline Soetaert[2]
Centre for Estuarine and Marine Ecology,
Netherlands Institute of Ecology,
4401 NT Yerseke,
The Netherlands

Thomas Petzoldt
Institut für Hydrobiologie
Technische Universität Dresden
01062 Dresden,
Germany

Received 21 February, 2011; accepted in revised form 11 March, 2011

Abstract:

The open-source problem solving software R [1] has become one of the most widely used systems for statistical data analysis. As it is a powerful interpreted language, it is also very well suited for other disciplines in scientific computing. One of the fields where considerable progress has been made is the solution of differential equations.
In this paper we describe a set of recently developed tools, so-called R-packages, to efficiently solve and analyze initial value problems of differential equations in R. Most of the methods are based on well-tested open-source numerical codes, combining the robustness and efficiency of these codes with the flexibility of the R language.
We exemplify the use of these tools by several examples. We start by implementing a well-known test problem for nonstiff solvers, the Arenstorff orbit ordinary differential equations. Next we solve the pendulum problem, a DAE of index 3. A description of a bouncing ball shows how roots and events can be programmed in R. After that we describe how to implement delay differential equations, which we exemplify with a DDE that is subject to an impulse, triggered at specific times. We end with a rather stiff partial differential equation, a combustion problem modeled in 2-D.
The presented R packages provide additional facilities to efficiently plot the outcome, to compare different scenarios, to estimate summary statistics, or to display execution statistics that help in assessing the performance of a particular method.

Keywords: initial value problems, ordinary differential equations, partial differential equations, delay differential equations, differential algebraic equations, problem solving environment, R

Mathematics Subject Classification: 65L05, 65N06, 65Y15

1. Introduction

Differential equations are the mathematical formalism expressing conservation laws of e.g. energy, momentum, or mass, and are commonly used in many engineering and scientific disciplines. Real-life applications only rarely allow finding analytical solutions, but rather require numerical approaches.
Whereas many high-quality numerical codes exist in the scientific literature, these are often implemented in a language such as FORTRAN or C, and require the user to formulate the mathematical models in the same language. For scientists that are not trained in programming in these

[1] Published electronically May 15, 2011
[2] Corresponding author. E-mail: k.soetaert@nioo.knaw.nl

languages the need to use advanced numerical codes prevents them from using modeling as a routine tool in data analysis. Although several problem solving environments (PSE) exist to solve differential equations, e.g. MATLAB [2], Maple [3], or Mathematica [4] most are quite expensive and many scientists, for instance from the natural sciences, are not accustomed to using these programming environments. This is unfortunate, as mathematical models provide a very powerful way of understanding nature's complexity, unraveling important processes or for qualitative testing of alternative hypotheses. Thus, a lot of scientific work could benefit from applying mathematical modeling tools.

In recent years, the open-source software R [1] has emerged as the main platform for statistical computation and it is also often used to produce high-quality graphics. As its use in universities is growing, more and more students become acquainted with the language. Therefore, its extension with mathematical model solving capabilities opens opportunities to reach a wider audience that can potentially use scientific and engineering models. Although R is still predominantly applied for statistical analysis and graphical representation, it is now rapidly becoming more suitable for mathematical computing, e.g. by recent developments in the field of matrix algebra [5] or for solving complex differential equations [6]. Recently a number of books have applied R in the field of environmental modelling [7, 8].

In two previous papers [6, 9] we reported on how to use R for solving initial value problems of ODEs, DAEs and PDEs, and boundary value problems [9]. Our target audience for these papers consisted of R-users, which are often not acquainted with applying differential equations. Thus, we selected relatively simple problems, mainly from the biological sciences, while little was said about mathematical and implementation aspects, or about R.
Here we elaborate more on the technical aspects of our implementations, and we deliberately choose more challenging mathematical problems. In addition, we motivate as to why we use R for our scientific programming.

The paper is organized as follows: first we give a short introduction to the R-software. Then we present an overview over the differential equation algorithms implemented in a series of R-packages (section 3) and provide examples of the main classes of differential equations in sections 4 to 9. We end with some concluding remarks.

2. The R Software

R is both a programming language as well as a software environment providing a wide variety of statistical, computational and graphical functions, and interfaces to other interpreted or compiled languages. It originally started as an open-source "dialect" of the S-language, but since then has become the lingua franca of statistical computing.
There are many reasons to use R also as a PSE. First of all, R is open source, distributed under the GNU General Public License (http://www.gnu.org/licenses/). As the implemented functions can be easily accessed and if desired changed, users can build rapidly on the work of others, rather than having to re-implement software over and over again. Also, freely available source code is often checked more thoroughly than would be done otherwise. From a philosophical point of view it is natural that researchers, which are primarily funded by the public sector, share their software without charge.
There are also many obvious advantages of implementing scientific problems, fit them to data, plot the results, and analyze them statistically in the same environment. R is particularly strong in statistics and graphics, so that it is excellently suited for these post-processing tasks.
Similar as other interpreted languages (e.g. Matlab ©, [10]), R allows compact vector and matrix operations, provides efficient high-level commands and can therefore be utilized with relatively limited programming expertise. Because of that, it is extremely well suited for rapid prototyping, testing alternative formulations, or performing numerical experiments.
Last but not least, R is distributed in a unique way. Users can add algorithms and functions to the R base implementation by means of so-called R-*packages*. Developers can build upon existing packages hence need not copy the underlying codes. R-packages are shared with the rest of the R-community by posting them on the Comprehensive R Archive Network (http://CRAN.R-project.org/), where they are formally quality controlled (e.g. function documentation is mandatory). Once on CRAN, they can be downloaded and installed directly within the R software, making all the package functions readily available within R. The distributed repository network consisting of about 80 mirrors gives rapid access to any updates, while different package version numbers make these updates transparent. Moreover, an

archive directory at CRAN makes each package version traceable. The package system contrasts with other scientific computing languages, where scripts are dispersed over the internet, and that lack R's modularity.

3. Differential Equation Solvers in R

Rather than implementing new methods we have as much as possible started from well-established, freely available numerical codes. As these codes are mostly programmed in FORTRAN, while R is programmed in C, this requires writing a wrapper in the C language that takes care of the interface between R and the underlying numerical code. In short, the solver method is triggered from an R-function, in which the input is checked, and some initial preparation is done. This R-function then calls the wrapper, written in C, where memory is allocated, function pointers assigned to the correct addresses, and where the options specific to the underlying method are set, after which the actual integration method, programmed in FORTRAN, is called. During the integration, the solver requires several times the calculation of derivatives given the current values of dependent and independent variables. Typically these derivative functions are programmed in R, and interfacing the solver with R is also done within C-code. Finally, it is also in the C-driver that some advanced methods such as delays, events, updating of forcing functions, or root finding are implemented (see below).

The preparation done in the R-function and in the C-wrapper allows shielding the user from the implementation details of the underlying codes. Thus it is possible to have a relatively simple and uniform interface to all codes, while still the strengths and peculiarities of each solver are kept.
The way this is supported in R is as follows. R-functions comprise two different types of arguments: some arguments have a default value and need not be specified, unless one is not satisfied with this default. Other arguments are unspecified in the function definition and *must* be given a value upon calling the function.

For instance, the main R-function that solves ordinary differential equations is defined as:

```
ode (y, func, times, parms, method = "lsoda", …)
```

The mandatory arguments do not have a default value and thus they must to be given a value by the user. For function ode, they are the initial values of the dependent variables (y), the R-function defining the initial value problem (func), the parameter values (parms), and the times at which output is wanted. The integration "method" used by default is "lsoda"; hence this argument need not be specified if one is content with that.
The "…" (dots argument) allows to pass further arguments valid for the selected solver and need to be specified only when the user wants more control over the solution process. For instance for lsoda it is possible to change the default absolute and relative tolerances (default atol = rtol = $1e^{-6}$), the minimal, maximal and initial time step, the maximum order for Adams (12) and BDF (5) methods, the maximal number of steps between output intervals (5000) and so on. It is also possible to write a Jacobian function, or to specify the structure of the Jacobian.

The main packages that deal with differential equations, and implemented by us are in table 1.

Table 1. Main R-packages solving differential equations.

R-package	Functionality	Reference
deSolve	Initial value problems of differential equations	[6]
rootSolve	Steady state solution of differential equations	[11]
bvpSolve	Boundary value problems of differential equations	[12]
ReacTran	1-D, 2-D and 3-D reactive transport models over structured grids	[13]

R-package deSolve [6], provides functions to solve initial value problems (IVP) of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE) and delay differential equations (DDE). It implements both Adams and backward differentiation formulae (codes LSODA, LSODAR, LSODE, LSODES from ODEPACK [14, 15], VODE [16] and DASPK [17]), and the three-stage RADAU II-A implicit Runge-Kutta method [18]. In contrast, several explicit Runge-

Kutta methods were *de novo* implemented (in C-code), based on original publications [19-23] and using Butcher tables [24] and ideas from [25] for step-size control and interpolation.
Finally, deSolve also includes methods that are especially designed to solve ODEs resulting by numerical differencing 1-D, 2-D or 3-D PDE problems by the method-of-lines.

R-Package rootSolve [11] provides rootfinding algorithms, implementing amongst others the Newton-Raphson method [26]. Several functions solve for the steady-state solution of 1-, 2- and 3-dimensional problems. This is achieved using the same matrix algebra functions as in ODEPACK (the Yale sparse matrix package, [27]), or from Sparsekit [28], or based on LINPACK's banded matrix solvers [29].

R- Package bvpSolve [12] solves boundary-value problems of ODEs, either by shooting [25, 26], by the collocation methods COLSYS and COLNEW [25] or based on a mono-implicit Runge-Kutta formula using the code BVPTWPC [30].

Package ReacTran is a comprehensive collection of R functions for modeling reactive-transport processes in multi-phase 1-D, 2-D or 3-D model domains with simple geometries. It offers functions for the generation of structured grids, and discretizes the diffusive-advective-transport equations on these grids, based on the flux-conservative form of the equations [31]. In addition, it includes several upstream-biased advection schemes containing flux limiters that are based on total variation diminishing concepts [32], and whose implementation is based on the GOTM code [33].

The specific features of the solvers in package deSolve are in table 2. For most solvers we added root-finding capacity, and the possibility to simulate events and delays. Only the newly implemented explicit Runge-Kutta methods allow to solve a differential equation within a derivative function, or to call a solver within a solver ('Nesting'). As they have too many global variables (i.e. common blocks), none of the existing codes that we used allows the solvers to be run in parallel or in nested calls.

Table 2 Features of the solvers in R-package deSolve; √ denotes that the feature was present in the original code; √* denotes that this feature has been added by us.

Solver	y'= f(t,y)	My'=f(t,y)	F(y',t,y)=0	Roots	Events	Delays	Nesting
lsoda/lsodar	√			√	√*	√*	
lsode	√			√*	√*	√*	
lsodes	√			√*	√*	√*	
vode	√				√*	√*	
daspk	√*	√*	√		√*	√*	
radau	√	√		√*	√*	√*	
explicit R-K	√				√*		√*

In the next sections we document how to implement, solve and plot several types of differential equations, using functions from package deSolve.
In order to assess the performance, we report the required computational time. All timing runs were performed on an Intel® Core ™ 2 Duo CPU T 9300 with a clock frequency of 2.5 GHz.

4. A simple ODE

We start by solving a simple initial value problem, the Arenstorff orbit problem [34], which is a second-order standard test problem for nonstiff solvers. It describes the closed trajectory for three bodies moving in the same plane, two have mass μ and (1- μ), while the third body has negligible mass. The implementation in R is:

```
library(deSolve)
Arenstorff = function(t, y, mu) {
  D1 = ((y[1]  + mu)^2 + y[2]^2)^(3/2)
  D2 = ((y[1]-(1-mu))^2 + y[2]^2)^(3/2)
  dy1 = y[3]
  dy2 = y[4]
  dy3 = y[1] + 2*y[4] - (1-mu)*(y[1]+mu)/D1 - mu*(y[1] - (1-mu))/D2
```

```
   dy4 = y[2] - 2*y[3] - (1-mu)*y[2]/D1 - mu*y[2]/D2
   return(list( c(dy1, dy2, dy3, dy4) ))
}
mu   = 0.012277471
yini  = c(x = 0.994, y = 0, dx = 0, dy = -2.00158510637908252240537862224)
out  = ode(func = Arenstorff, y = yini, times = seq(from = 0, to = 18, by = 0.01), parms = mu )

plot (out[ ,c("x","y")],  type = "l", lwd = 2, col = "darkblue", main = "Arenstorff")
diagnostics(out)
```

After loading the package deSolve, the function defining the differential equation (Arenstorff) is implemented. It has time (t), the current state values (y) and the parameter (mu) as input arguments and it returns the derivative vector, in a list. A 'list' in R is a data structure that can contain many different types of other data.

The equations are solved with R's ordinary differential equation solver ode, which takes as input arguments the derivative function (func), the initial values of the state variables (y, a vector), a vector with the times for which output is wanted, and the parameter value (parms). The length of the initial value vector y determines the number of differential equations, and hence the number of derivatives that the solver expects to be returned from func. Note that the bizarre value of the initial condition is chosen so that at the end, the variables have returned to the initial condition.

The one but last statement plots variable "x" versus "y", using a line plot (type = "l") and with line width double the default (lwd = 2) (figure 1). The code ends with printing the solver diagnostics (results not shown). This amongst other things tells that lsoda (the default integration method used) selected the Adams method for solving this problem; 1957 steps were taken, requiring 3949 function evaluations, and the order of the method last used was 7.

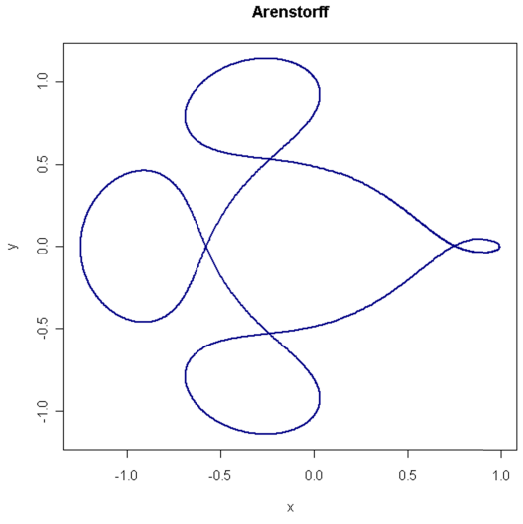

Figure 1. The Arenstorff ODE

5. DAEs

Many practical problems are more exactly described by a combination of differential and algebraic equations, so-called differential algebraic equations (DAE). Two solution methods for DAEs are in package deSolve. R-function radau is based on the standard implicit Runge-Kutta code written by Hairer and Wanner [18], and is capable of solving differential algebraic equations represented in linearly implicit form M y' = f(t, y) with index ≤ 3. Function daspk [17] solves general implicit DAEs of the form f(y', y, t) = 0 having index ≤ 1. As it is fairly simple to rewrite ODEs and linearly implicit DAEs in this form, the R-code is such that daspk can also solve these simpler equations, using the same formalism as the other solvers.

The R-definition of these solvers are:

```
radau (y, func, times, parms, nind = c(length(y), 0, 0), mass = NULL, …)
daspk (y, func = NULL, parms, dy = NULL, res = NULL, mass = NULL, …)
```

Function radau requires that the number of variables of index 1 to 3 is specified, as a three-valued vector, in argument nind; the default is to have all variables of index 1. In addition, for solving DAEs, radau requires that the mass matrix is provided. Problems to be solved by daspk can either be presented by the derivative function func and a mass matrix (or mass = NULL for ODEs), or via a residual function res. If used for solving DAEs, daspk also requires specification of the initial value of the derivatives (dy).

The pendulum problem in R

We show how to implement the index 3 pendulum equation, and solve it using radau; the equations for this problem can be found in [18] or [17]. It is assumed that the squared length of the pendulum (x^2+y^2) is 1.

```
library(deSolve)
Pendulum = function (t, y, p) {
  with (as.list(y), {
    dx = u
    dy = v
    du = -lambda * x
    dv = -lambda * y - 9.8
    res = x^2 + y^2 -1
   return (list(c(dx, dy, du, dv, res)))
  })
}
yini = c(x = 1, y = 0, u = 0, v = 1, lambda = 1)
M = diag(nrow = 5); M[5, 5] = 0

times = seq(from = 0, to = 10, by = 0.01)
out = radau (y = yini, func = Pendulum, parms = NULL, times = times, mass = M, nind = c(2, 2, 1))

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

The DAE function Pendulum calculates and returns the derivatives of the first 4 variables (dx, …, dv) and the residual of the algebraic equation (res). After defining a set of (consistent) initial conditions (yini), the mass matrix (M) is created. This consists of the unity (or diagonal) matrix with 5 rows and 5 columns, but where the element on position [5, 5] is 0 rather than 1. The first two equations are of index 1, followed by two of index 2, one of index 3; this is concatenated in a vector (c(2, 2, 1)) and passed to the solver via argument nind. The model is solved for 10 seconds, and output written at 0.01 second intervals (times). The first plot statement depicts at once all model variables against time, using the variable name as figure title. The last plot statement depicts "y" versus "x", showing the pendulums trajectory (figure 2).

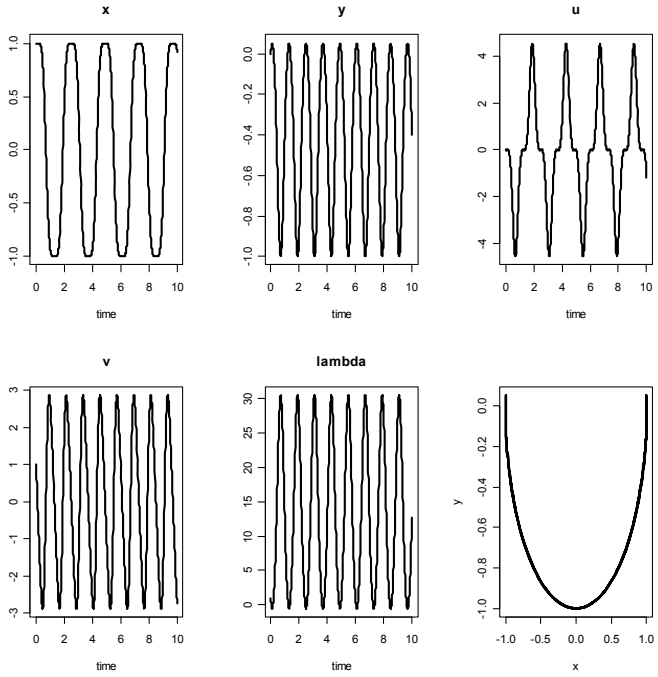

Figure 2. Output of the pendulum model, an index 3 DAE

6. Roots and Events

In many instances, the computation should be terminated or altered if a certain condition is met, i.e. at the root of a function. Determining this particular time and the event that ensues is an important part of the solution. Root detection in ODEs involves, in addition to the derivative function also a "root" function. If at the root the simulation has to continue but with altered state variables, then a third "event" function is required.
The problem can be stated as to solve a function y'=f(t, y, p) until a condition g(t*, y*, p) = 0 is met, after which either the simulation is stopped, or the states are altered according to y_n= e(t*, y*, p).

Root tracking has been implemented in the solvers in two ways. In the original RADAU code [18], the solver calls a subroutine each time it has performed a successful time step. In the R-implementation, this returns control to the C-wrapper. Here, it is checked whether the integration step includes an output time, and if so, RADAU's continuous output formula is invoked to obtain the state variable values at these time points. It is relatively straightforward to use this function also to check whether a sign change in a (set of) root function(s) has occurred, and if so, to locate the root, using Brent's method [25].
In contrast, root tracking was already present in one of the ODEPACK solvers, LSODAR, and coded in FORTRAN. For consistency, we implemented the same root-tracking function also for two other ODEPACK solvers, LSODE and LSODES.

During an "event" the state variables are instantaneously altered. In biological sciences for instance, this may occur because animals are transferred to new culture medium, or are released in the wild; in pharmacokinetic modeling, events may represent the injection of a drug in the blood stream. In many problem solving environments, these jumps in the states have to be taken care of by the user, e.g. the current integration is stopped, and the user changes the variables and reinitiates the integration. In R this has been automated, and implemented in C-code. When during a time step, an event occurs, the C-code changes the state variable values and the solver is informed of this fact, by setting the appropriate flag, such that it can adjust its time step to the new situation. This way the integration does not need to be halted.

Events may either be associated with a root (in which case it is not known in advance when they will occur), or the times at which an event occurs can be defined *a priori*. It is possible to define in a table (in R-terminology a data.frame) when an event occurs, which state variables it affects and how. It is also possible to effectuate the change in an event function (see next example).

The Bouncing Ball example in R

The most lucid example of a root function which triggers an event is a ball, falling under the force of gravity. When it hits the ground, it bounces back, at a velocity, reduced by a certain amount.

The differential equation reads: $y'' = -g$, which is rewritten as two first-order ODEs: $dy_1 = y_2$; $dy_2 = -g$, where y_1 represents the height of the ball, y_2 its velocity, g is the gravitational acceleration (9.8 ms^{-2}). An event is triggered when the ball hits the ground, i.e. its height (y_1) equals 0, thus the root function is $g(t, y, p) = y_1$. When the ball hits the ground, the event function is that it bounces (i.e. the ball's velocity changes sign) at a velocity that is reduced to 90%. Thus the event is specified as: $y_1 = 0$; $y_2 = -y_2 \cdot 0.9$

Implemented in R, and running two scenarios, this becomes:

```
library(deSolve)

Ballode = function(t, y, parms) {
   dy1 = y[2]
   dy2 = -9.8
  return (list(c(dy1, dy2)))
}

Root = function(t, y, parms)
    return (y[1])

Event = function(t, y, parms) {
    y[1] = 0
    y[2] = -0.9 * y[2]
    return(y)
}
yini1 = c(height = 0, v = 20)
yini2 = c(height = 0, v = 10)
times = seq(from = 0, to = 20, by = 0.1)

out = ode(times = times, y = yini1, func = Ballode, parms = NULL, rootfun = Root,
          events = list(func = Event, root = TRUE), method = "lsode")
out2 = ode(times = times, y = yini2, func = Ballode, parms = NULL, rootfun = Root,
          events = list(func = Event, root = TRUE), method = "lsode")
plot(out, out2, which = "height", main = "bouncing ball", ylab = "height", lwd = 2)
legend("topright", col = 1:2, lty = 1:2, legend = c("v = 20", "v = 10"), title = "initial value")
```

Function `Ballode` implements the differential equations which represent the ball's behavior between events. Function "`Root`" will return zero when the ball's height equals 0, at which time the event function "`Event`" will change the value of y_2. The integration is initiated with the ball's position at the ground. It is run twice, with different initial values of the upward velocity `v`, 20 (`yini1`) and 10 m s^{-1} (`yini2`) respectively. The integration is to proceed in the interval [0, 20] and produce output at 0.1 second intervals (`times`). Although the default solver selected by `ode` (`lsoda`) is perfectly capable of retrieving a root, here we use method "`lsode`" instead (purely for educational purposes). Finally the height of the ball is plotted for the two scenarios (`out1, out2`), and a `legend` added, thus producing figure 3.

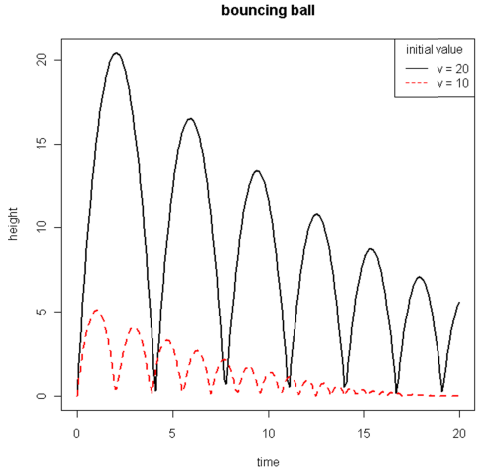

Figure 3. The bouncing ball example, an ODE including roots and events.

7. Delays

Delay differential equations (DDEs) are similar to ordinary differential equations, except that their evolution involves past values of the state variable. The solution of delay differential equations therefore requires knowledge of not only the current state, but also of the state and/or its derivative at a certain previous time.
The delay differential equation solvers were implemented in R as an extension for most available ODE and DAE initial value solvers, i.e. for all ODEPACK solvers, for daspk and radau (see table 2). It was implemented in C-code.

The R-function that solves delay differential equations is defined as:

```
dede (y, func, times, parms, method = "lsoda", control = NULL, ...)
```

where y, func, times, parms and method are the same as before, control can be a list with extra options relating to the delays and the ... is any option for the selected integration method.
Two utility functions allow retrieving past values of variables and derivatives: lagvalue and lagderiv respectively.

In order to extract the past information at random time points, the algorithm should be able to interpolate between two consecutive time-points, embracing the requested time value. Two interpolation methods are implemented to recover the past values and past derivates of state variables. In the first method, the values of the state variables and their derivates are stored at each time step, and cubic Hermite interpolation is used to recover the requested past values. In the second method, we make use of the dense output strategy in the underlying solvers. For the linear multistep methods (the ODEPACK solvers, daspk, vode) the necessary information comprises a history of the entire Nordsieck vector. For the solver radau, we keep a history of the parameters of the interpolating polynomial. In addition, both types of solvers also require the time-step used at each time point. Practical experience showed that Hermite interpolation is generally as precise as the dense output strategy. As the memory requirements of the dense output strategy is quite large compared to Hermite interpolation, the latter is the default. However, especially for the Adams method which can have order up to 12, the dense output can occasionally produce much more precise estimates.
For demanding models, the number of time-steps taken may be too large to be held in the buffer. The history buffer has a fixed size and past values are stored again from the beginning if the end is reached. The size of this buffer is one of the parameters that can be set in the control argument to the dede function.

A time dependent DDE with impulses in R

We now give an example of a DDE model that exhibits both discrete and continuous behavior over the time interval of interest. The discrete jumps in the states (events) occur at particular points in time. The example comes from [35] and describes delayed cellular neural networks with impulsive effects. As the delays for this problem at times vanish during the integration, this is a relatively difficult problem. The relatively complex equations can be found in [35] and are not repeated here.

```
library(deSolve)

Neural = function(t, y, p) {
  fun = function(x) return(0.5 * (abs(x+1) - abs(x-1)))

  tlag1 = t - (1 + cos(t))/2
  if (tlag1 > 0)  Lag1 = lagvalue(tlag1)  else  Lag1 = yini

  tlag2 = t - (1 + sin(t))/2
  if (tlag2 > 0)  Lag2 = lagvalue(tlag2)  else  Lag2 = yini

  dy1 = -6*y[1] + sin(2*t)*fun(y[1]) + cos(3*t)*fun(y[2]) + sin(3*t)*fun(Lag1[1]) + sin(t)*fun(Lag2[2]) + 4*sin(t)
  dy2 = -7*y[2] + cos(t)*fun(y[1])/3 + cos(2*t)*fun(y[2])/2 + cos(t)*fun(Lag1[1]) + cos(2*t)*fun(Lag2[2]) + 2*cos(t)

  return (list(c(dy1, dy2)))
}

eventfun = function(t, y, p)
  return (c(y[1] * 1.2, y[2] * 1.3))

yini = c(y1 = -0.5, y2 = 0.5)
out  = dede (func = Neural, y =yini, times = seq(from = 0, to = 40, by = 0.025), parms = 0,
             events = list(func = eventfun, times = seq(from = 2, to = 40, by = 2)) )
plot(out[,-1], type = "l", lwd = 2)
```

Note that a local function (fun) is defined within the derivative function (Neural). The past value at times tlag1 and tlag2 are requested via function lagvalue. This is only possible when these requested times are > 0, else the initial condition yini is used instead. During the event, the first and second value of y are increased with 20 and 30 % respectively. Note also how, in the call to the solver dede, the events are specified to occur at two-daily intervals (times). The last statement plots both variables versus one another (figure 4).

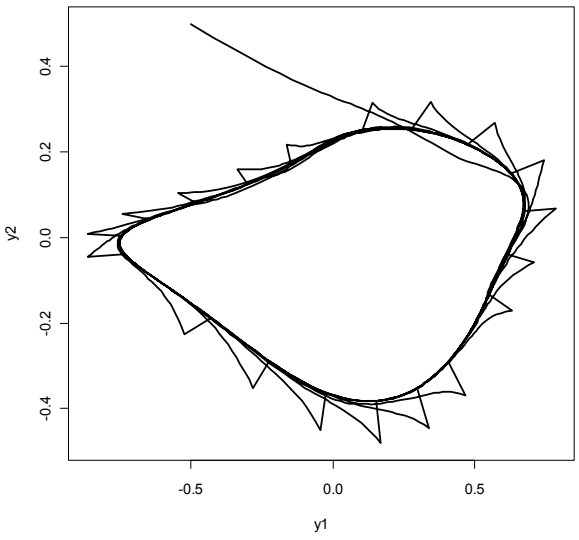

Figure 4 A time-dependent DDE with impulses

8. PDEs

An important class of differential equations arises by discretising partial differential equations, particularly parabolic equations that can be solved by the method of lines (MOL). This replaces all the spatial derivatives with finite differences, but leaves the time derivatives intact, after which an ordinary differential equation solver is used to solve the system of differential equations. When the system is stiff, much computational efficiency can be gained and memory requirements reduced by taking into account the special, sparse, structures that the Jacobian matrices have when discretising 1-D, 2-D or 3-D problems. In package deSolve, three special-purpose solvers implement the MOL for 1, 2 and 3-D problems: ode.1D, ode.2D, ode.3D. If the number of species that is described and the number of grid cells in both directions is known, then the structure of the system's sparsity can be easily derived. For instance, when the model is one-dimensional and stiff, and the variables are arranged grid-wise, then the Jacobian matrix will be banded, and this can be efficiently handled by radau, lsode, lsoda, vode and daspk. As it is more natural to arrange state variables according to species, in the C-implementation of function ode.1D, the state variables are re-ordered grid-wise before being presented to the solver.
If the model is multi-dimensional (2-D or 3-D), then the non-zero elements are located in a number of discrete bands parallel to the diagonal, and this structure can also easily be derived from the systems dimension. ODEs with such – arbitrarily sparse – systems are efficiently solved with function lsodes, which implements the Yale sparse matrix algebra package from [27].
Note that all these functions solve the linear system that arises by direct (LU) methods. The preconditioned Krylov methods, originally present in DASPK have not yet been made available in the R interface of R-function daspk.

Similar to the MOL implementation in R-package deSolve, it is possible to solve for the steady-state condition of 1-D, 2-D and 3-D problems, using root solvers from R-package rootSolve; they are in functions steady.1D, steady.2D, and steady.3D. By default these functions use the same matrix algebra functions, but they also include preconditioned solvers from FORTRAN package sparsekit [28].

The Combustion Problem in R

To illustrate how this works in R, we implement the relatively stiff combustion problem from [31]. Simplified, the reactive-transport equation describes the change in space and time of the reactant U, as a function of transport fluxes and reactions (reac), and where the flux is by diffusion, with diffusion coefficient K

$$\frac{\partial U}{\partial t} = -\nabla \cdot (-K \nabla U) + reac$$

As in many practical applications, one is not just interested in the value of the state variables, but also in the fluxes, it is natural to represent this parabolic equation as the gradient of fluxes across the grid interfaces:

$$\frac{\partial U}{\partial t} = -\nabla \cdot Flux + reac$$

where

$$Flux = -K \nabla U$$

In the combustion example implemented in R, state variables are represented by their values at certain grid points and spatial derivatives are approximated through differences in these values. Thus, the state variables are defined in the centre of grid cells, the derivatives and fluxes are defined on the cell interfaces. The behavior of the solution at the boundaries is prescribed as a known value (=1) for the downstream boundary, and a zero-flux boundary upstream. It is assumed that the diffusion coefficient K equals 1:

```
library(deSolve)
library(rootSolve)

N  = 100
dx = dy = 1/N
alfa  = 1; delta = 20; R = 5

Combustion = function(t, y, p) {
   U   = matrix(nrow = N, ncol = N, data = y)

   reac = R /alfa/delta * (1+alfa-U) * exp(delta*(1-1/U))

   Flux_X = - rbind(0, U[2:N, ] - U[1:(N-1), ], 1 - U[N,]) / dx
   Flux_Y = - cbind(0, U[, 2:N] - U[ ,1:(N-1)], 1 - U[,N]) / dy
   dU   = -  (Flux_X[2:(N+1), ] - Flux_X[1:N, ])/dx    -  (Flux_Y[ ,2:(N+1)] - Flux_Y[, 1:N])/dy   +   reac

   return ( list (dU) )
}

std = steady.2D(y = rep(1, N*N), parms = NULL, func = Combustion, nspec = 1,
             dimens = c(N, N), lrw = 1e6, positive = TRUE)

times = c(seq (from = 0, to = 0.24, by = 0.08),  seq(from = 0.3, to = 0.36, by = 0.02))
dyn   = ode.2D(y = rep(1, N*N), parms = NULL, func = Combustion, nspec = 1,
                  dimens = c(N, N), lrw = 1e6, times = times)

image(dyn, zlim = c(1, 2), mfrow = c(3, 3), legend = TRUE, ask = FALSE, main = paste("t =", times))
image(std, main="Steady-state", legend = TRUE, mfrow = NULL)
diagnostics(dyn)
```

The 2-dimensional domain (of length = 1) is subdivided into discrete computational grid cells, 100 in the X- and 100 in the Y-direction (N); the grid size in both directions is respectively dx, dy. Thus this model comprises N*N = 10000 coupled ODEs.
In the function implementing the derivative (Combustion), the variable values y are passed as a vector. Before calculating on them, this vector is recast into matrix form (U). R computes as easily on entire matrices, vectors as on single numbers. As it is much more efficient to perform vector or matrix calculations rather than using a loop, the reaction rate, reac, is calculated on the entire matrix U at once. The result of this calculation is a matrix, reac, which is of the same size as U.
Next the flux in X and Y direction is estimated (Flux_X, Flux_Y). For the internal cells, we estimate the gradient in X direction by subtracting two matrices, divided by dx. The first matrix contains all rows of U, except the first one, the other contains all rows except the last one. Here the notation 2:N creates a vector from 2 to N in integer steps. The notation U[2:N,] selects all columns (the second dimension is left blank), and all but the first row from matrix U. the notation U[,1:(N-1)] selects all except the last column. Padded to the internal gradients is the upstream boundary (at x = 0), which is a zero flux boundary, while at the downstream boundary, the value 1 is prescribed, and the gradient becomes 1-U[N,]. Binding a row or a column to a matrix is done by R-function rbind (for fluxes in X-direction) and cbind (Y-direction) respectively. The resulting flux matrices are of dimension (N+1, N) and (N, N+1) for the X- and Y-direction respectively. Finally, the derivative dU is calculated as the sum of the negative flux divergence in X- and Y-direction, and the reaction term, and returned as a list.

The model is solved in two ways. First, the steady-state condition is estimated, using function steady.2D from R-package rootSolve. We specify the number of species (nspec) and the dimensionality of the problem (dimens). We also need to give an estimate of the size of the work space (lrw). The Newton-Raphson method that is called by steady.2D requires an initial guess of the solution (y), but as it happens, for this problem this is not very critical; if we simply set the 10000 values equal to 1 (rep(1, N*N)), the solver finds the steady-state solution in 19 iterations. Note the argument positive = TRUE. Sometimes non-realistic solutions exist (e.g. with negative values), and this makes converging to a solution quite difficult. This option forces the solver to only find positive values (and in this particular case, the solver fails without this option).

Next the model is run dynamically, using solver ode.2D from R-package deSolve. From the start of the integration till t = 0.3 not much happens, so we initially request output at intervals = 0.08. After t = 0.3 however, the reactant ignites and changes are very fast, so from t = 0.3 to 0.36 we request output at intervals equal to 0.02. Thus the times vector consists of two combined sequences (times = c(seq (from = 0, to = 0.24, by = 0.08), seq(from = 0.3, to = 0.36, by = 0.02))).

The one but last two statements produce image plots for the dynamic simulation and the steady-state solution respectively (Figure 5). We request the figures to be arranged in 3 rows and 3 columns (mfrow = c(3, 3)), while the steady-state solution should be simply added to this arrangement (mfrow = NULL). Finally the diagnostics of the dynamic simulation is printed (not shown). From that we learn that lsodes has taken 448 steps, requiring 848 function evaluations, and 11 Jacobian evaluations and LU decompositions. The method last used was of order 3. It took about 5.7 seconds to solve the model dynamically, 1.4 seconds to estimate the steady-state condition for this 10000 state variable model.

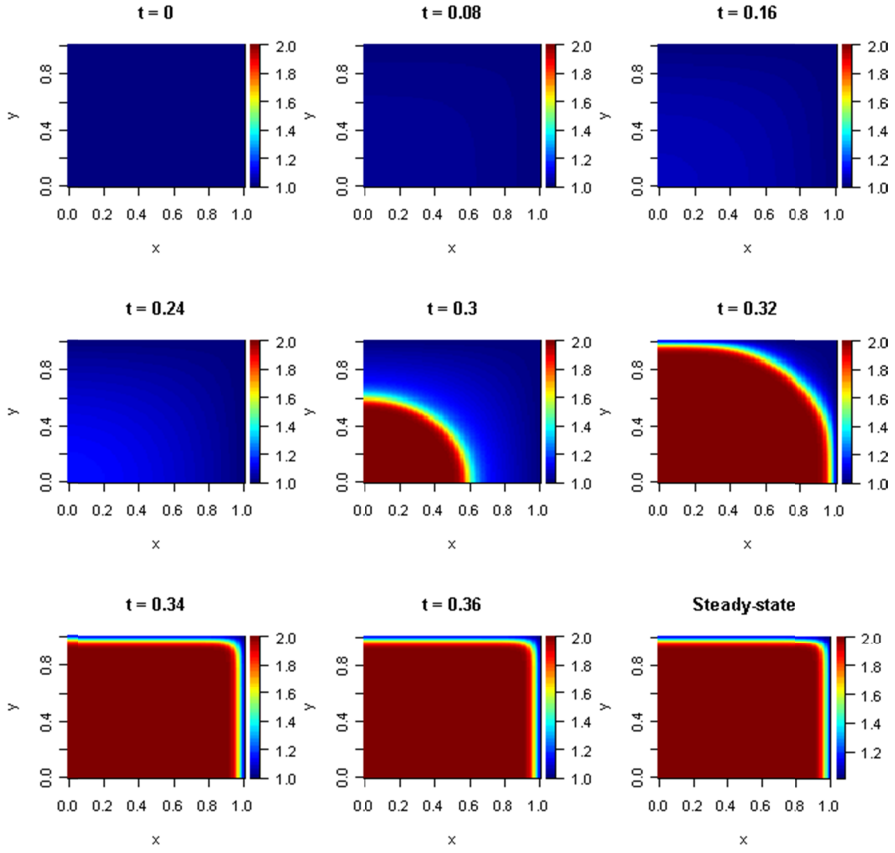

Figure 5 The combustion problem, dynamic and steady-state solution

9. Final Remarks

From the user's point of view good scientific software should be free, flexible, fast and efficient, and provide powerful analysis methods and good graphical capabilities. For some, user-friendliness is another requirement, while platform-independence is attractive to those scientists that want to share their results. While R is certainly free and flexible, runs on all main platforms, and is ideally suited for post-processing scientific results, it is not per se fast nor is it always considered to be very user-friendly. R is essentially a scripting language, which may be considered less user-friendly compared to graphical user interfaces. There do exist R-packages that provide drag-and-drop facility but they are not commonly used. Rather, at best most R-users work with an (ASCII) editor that provides R-sensitive syntax.

As it is an interpreted language, applications written in R code are not cheap in terms of CPU-time. Compared to compiled languages, interpreted code often increases CPU time with a factor in the order of 10 times and even more so if loops are used. In the PDE example (section 8) we tried to overcome this penalty by operating on entire matrices rather than using a loop. This way, the computational cost is just a few 10s of percentages (e.g. [6]). However, this approach cannot be applied for all problems, and in general the more statements appear in the derivative function, the slower R will be compared to compiled code. As we often use our models in "inverse mode", e.g. fit the models to data or in MCMC (Markov-Chain Monte Carlo) simulations [36], we often run a model in the order of $1e^5$ – $1e^6$ times, and then every (fraction of a) second gained is worthwhile. With this in mind, we added to the R-packages the option to program the derivative function (and – if desired – the Jacobian function) in a compiled language that produces a DLL (on Windows) or a shared object file (on UNIX like operating systems), such as FORTRAN or C. Although the setup for such models is still conveniently handled by R, and these models are solved using the same solvers, this now proceeds by calling compiled code directly from compiled code, and this reduces the computational cost to a few percent compared to a model where everything would have been programmed in a compiled language [6]. It would take too far to elaborate on that, but the interested user is referred to the technical manual [37].
Finally, the solvers implemented in R thus far mainly fall in the category of general problem solvers rather than being dedicated to solving a particular type of problem in the most efficient way. A lot of improvement can still be made, e.g. by including functions that also solve more specific problems such as Hamiltonian systems, or in which PDEs can also be solved using unstructured grids (FEM).

Acknowledgments

The success of the R-project is due to the hard work of the R Core Development Team, and of the increasing amount of enthusiasts that produce add-on functionality. We also thank our students and post-docs for testing the packages, and the package users all over the world for giving feedback and encouragement.
None of this would have been possible without the work of the mathematicians, computer scientists and others who share their numerical scientific codes. More specifically, we are most indebted to Alan Hindmarsh, Linda Petzold, Ernst Hairer, Uri Ascher, Bob Russell, Jeff Cash and Francesca Mazzia. Jeff is also thanked for the invitation to present this work at the ICNAAM conference 2010.

References

[1] R Development Core Team: R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2009), URL http://www.R-project.org/, ISBN 3-900051-07-0, 2010.
[2] L.F. Shampine and M.W. Reichelt, The MATLAB ODE Suite, *SIAM J. Sci. Comput.*, **18** 1-22 (1997).
[3] M.B. Monagan, K.O. Geddes., K.M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, P. DeMarco, *Maple Advanced Programming Guide (Maple 11)*. Maplesoft, 2007.
[4] S. Wolfram, et. al., *Mathematica Documentation*, http://reference.wolfram.com/
[5] D. Bates and M. Maechler: Matrix: A Matrix package for R, *R package version 0.999375-9*, 2008.
[6] K. Soetaert, T. Petzoldt and R.W. Setzer: Solving Differential Equations in R: Package deSolve, *Journal of Statistical Software* **33(9)** 1-25 (2010). http://www.jstatsoft.org/v33/i09.
[7] K. Soetaert and P. M. J. Herman: *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform,* Springer-Verlag, New York, 2009.
[8] M. H. H. Stevens: *A Primer of Ecology with R*, Springer-Verlag, Berlin, 2009.
[9] K. Soetaert, T. Petzoldt and R.W. Setzer: Solving Differential Equations in R. *The R Journal* **2(2)** 5-15 (2010).
[10] The Mathworks Inc., *MATLAB (R) release 2010a* (2010), URL http://www.mathworks.com/, MATLAB is a registed property of The Mathworks Inc.
[11] K. Soetaert: rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations http://CRAN.R-project.org/package=rootSolve, *R package version 1.6*, 2009.
[12] K. Soetaert, J. R. Cash and F. Mazzia: bvpSolve: Solvers for Boundary Value Problems of Ordinary Differential Equations. http://CRAN.R-project.org/package=bvpSolve, *R package version 1.1*, 2010.

[13] K. Soetaert and F. Meysman: ReacTran: Reactive Transport Modelling in 1D, 2D and 3D *http://CRAN.R-project.org/package=ReacTran, R package version 1.1*, 2010.
[14] A. C. Hindmarsh: ODEPACK, A Systematized Collection of ODE Solvers, *Scientific Computing, Vol.* (Editor: R. Stepleman, IMACS / North-Holland, Amsterdam), IMACS Transactions on Scientific Computation **1** (1983), 55–64
[15] L. R. Petzold: Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. *SIAM Journal on Scientific and Statistical Computing* **4** 136–148 (1983).
[16] P. N. Brown, G. D. Byrne and A. C. Hindmarsh: VODE, A Variable-Coefficient ODE Solver *SIAM Journal on Scientific and Statistical Computing* **10** 1038–1051 (1989).
[17] K. E. Brenan, S. L. Campbell and L. R. Petzold: *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM Classics in Applied Mathematics, 1996.
[18] E. Hairer, and G. Wanner: Solving Ordinary Differential Equations II: *Stiff and Differential-Algebraic Problems.* Second Revised Edition, Springer-Verlag, Heidelberg, 2010.
[19] E. Fehlberg: Klassische Runge-Kutta-Formeln fünfter and siebenter Ordnung mit Schrittweiten-Kontrolle, *Computing (Arch. Elektron. Rechnen)* **4** 93-106 (1967).
[20] J. R. Dormand and P. J. Prince: A family of embedded Runge-Kutta formulae. *J. Comput. Appl. Math.*, **6** 19-26 (1980).
[21] P. J. Prince and J. R. Dormand: High order embedded Runge-Kutta formulae. *J. Comput. Appl. Math.* 7 67-75 (1981).
[22] P. Bogacki and L.F. Shampine. A 3(2) pair of Runge–Kutta formulas. Applied Mathematics Letters 2 (4) 321–325 (1989)
[23] J. R. Cash and A. H. Karp: A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides. *ACM Transactions on Mathematical Software* **16** 201-222 (1990).
[24] J. C. Butcher: *The Numerical Analysis of Ordinary Differential Equations. Runge-Kutta and General Linear Methods*. Wiley, Chichester, 1987.
[25] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery: *Numerical Recipes*, 3rd edition, Cambridge University Press, 2007.
[26] U. M. Ascher, R. M. M. Mattheij and R. D. Russell: *Numerical solution of boundary value problems for ordinary differential equations*, Prentice Hall, Englewood Cliffs, N.J., 1988.
[27] S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A.H. Sherman: Yale Sparse Matrix Package. i. The Symmetric Codes *International Journal for Numerical Methods in Engineering* **18** 1145-1151, 1982.
[28] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations. VERSION 2* (1994).
[29] J. Dongarra, J. Bunch, C. Moler, and G. Stewart: *LINPACK Users Guide*, SIAM (1979).
[30] J. R. Cash, and F. Mazzia: A new mesh selection algorithm, based on conditioning, for two-point boundary value codes. *J. Comput. Appl. Math.* **184** 362–381 (2005).
[31] W. Hundsdorfer and J. Verwer: *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*. Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 2003.
[32] J. Pietrzak: The use of TVD limiters for forward-in-time upstream-biased advection schemes in ocean modeling *Monthly Weather Review* **126** 812–830 (1998).
[33] H. Burchard, K. Bolding and M. Villarreal: *GOTM, a general ocean turbulence model. Theory, applications and test cases*, tech Rep EUR 18745 EN. European Commission (1999).
[34] E. Hairer, S.P. Norsett and G. Wanner: *Solving Ordinary Differential Equations I: Nonstiff Problems.* Second Revised Edition, Springer-Verlag, Heidelberg, 2009.
[35] S.P. Corwin, S. Thompson and S.M. White: Solving ODEs and DDEs with Impulses. *Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)* **3(1-2)** 139-149, 2008.
[36] K. Soetaert and T. Petzoldt: Inverse modelling, sensitivity and Monte Carlo analysis in R using package FME. *Journal of Statistical Software* **33** 1–28 (2010), *http://www.jstatsoft.org/v33/i03/.*
[37] K. Soetaert, T. Petzoldt and R.W. Setzer: *R-package deSolve, Writing Code in Compiled Languages* (2009), *http://CRAN.R-project.org/package=deSolve*, package vignette.