Solving Differential Equations in R

by Karline Soetaert, Thomas Petzoldt and R. Woodrow Setzer[1]

Abstract Although R is still predominantly applied for statistical analysis and graphical representation, it is rapidly becoming more suitable for mathematical computing. One of the fields where considerable progress has been made recently is the solution of differential equations. Here we give a brief overview of differential equations that can now be solved by R.

Introduction

Differential equations describe exchanges of matter, energy, information or any other quantities, often as they vary in time and/or space. Their thorough analytical treatment forms the basis of fundamental theories in mathematics and physics, and they are increasingly applied in chemistry, life sciences and economics.

Differential equations are solved by integration, but unfortunately, for many practical applications in science and engineering, systems of differential equations cannot be integrated to give an analytical solution, but rather need to be solved numerically.

Many advanced numerical algorithms that solve differential equations are available as (open-source) computer codes, written in programming languages like FORTRAN or C and that are available from repositories like GAMS (`http://gams.nist.gov/`) or NETLIB (`www.netlib.org`).

Depending on the problem, mathematical formalisations may consist of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE), or delay differential equations (DDE). In addition, a distinction is made between initial value problems (IVP) and boundary value problems (BVP).

With the introduction of R-package `odesolve` (Setzer, 2001), it became possible to use R (R Development Core Team, 2009) for solving very simple initial value problems of systems of ordinary differential equations, using the `lsoda` algorithm of Hindmarsh (1983) and Petzold (1983). However, many real-life applications, including physical transport modeling, equilibrium chemistry or the modeling of electrical circuits, could not be solved with this package.

Since `odesolve`, much effort has been made to improve R's capabilities to handle differential equations, mostly by incorporating published and well tested numerical codes, such that now a much more complete repertoire of differential equations can be numerically solved.

More specifically, the following types of differential equations can now be handled with add-on packages in R:

- Initial value problems (IVP) of ordinary differential equations (ODE), using package **deSolve** (Soetaert et al., 2010b).
- Initial value differential algebraic equations (DAE), package **deSolve** .
- Initial value partial differential equations (PDE), packages **deSolve** and **ReacTran** (Soetaert and Meysman, 2010).
- Boundary value problems (BVP) of ordinary differential equations, using package **bvpSolve** (Soetaert et al., 2010a), or **ReacTran** and **rootSolve** (Soetaert, 2009).
- Initial value delay differential equations (DDE), using packages **deSolve** or **PBSddesolve** (Couture-Beil et al., 2010).
- Stochastic differential equations (SDE), using packages **sde** (Iacus, 2008) and **pomp** (King et al., 2008).

In this short overview, we demonstrate how to solve the first four types of differential equations in R. It is beyond the scope to give an exhaustive overview about the vast number of methods to solve these differential equations and their theory, so the reader is encouraged to consult one of the numerous textbooks (e.g., Ascher and Petzold, 1998; Press et al., 2007; Hairer et al., 2009; Hairer and Wanner, 2010; LeVeque, 2007, and many others).

In addition, a large number of analytical and numerical methods exists for the analysis of bifurcations and stability properties of deterministic systems, the efficient simulation of stochastic differential equations or the estimation of parameters. We do not deal with these methods here.

Types of differential equations

Ordinary differential equations

Ordinary differential equations describe the change of a *state variable* y as a function f of one *independent variable* t (e.g., time or space), of y itself, and, optionally, a set of other variables p, often called *parameters*:

$$y' = \frac{dy}{dt} = f(t, y, p)$$

[1]The views expressed in this paper are those of the authors and do not necessarily reflect the views or policies of the U.S. Environmental Protection Agency

In many cases, solving differential equations requires the introduction of extra conditions. In the following, we concentrate on the numerical treatment of two classes of problems, namely initial value problems and boundary value problems.

Initial value problems

If the extra conditions are specified at the initial value of the independent variable, the differential equations are called **initial value problems** (IVP).

There exist two main classes of algorithms to numerically solve such problems, so-called *Runge-Kutta* formulas and *linear multistep* formulas (Hairer et al., 2009; Hairer and Wanner, 2010). The latter contains two important families, the Adams family and the backward differentiation formulae (BDF).

Another important distinction is between *explicit* and *implicit* methods, where the latter methods can solve a particular class of equations (so-called "stiff" equations) where explicit methods have problems with stability and efficiency. Stiffness occurs for instance if a problem has components with different rates of variation according to the independent variable. Very often there will be a tradeoff between using explicit methods that require little work per integration step and implicit methods which are able to take larger integration steps, but need (much) more work for one step.

In R, initial value problems can be solved with functions from package **deSolve** (Soetaert et al., 2010b), which implements many solvers from ODEPACK (Hindmarsh, 1983), the code `vode` (Brown et al., 1989), the differential algebraic equation solver `daspk` (Brenan et al., 1996), all belonging to the linear multistep methods, and comprising Adams methods as well as backward differentiation formulae. The former methods are explicit, the latter implicit. In addition, this package contains a de-novo implementation of a rather general Runge-Kutta solver based on Dormand and Prince (1980); Prince and Dormand (1981); Bogacki and Shampine (1989); Cash and Karp (1990) and using ideas from Butcher (1987) and Press et al. (2007). Finally, the implicit Runge-Kutta method `radau` (Hairer et al., 2009) has been added recently.

Boundary value problems

If the extra conditions are specified at different values of the independent variable, the differential equations are called **boundary value problems** (BVP). A standard textbook on this subject is Ascher et al. (1995).

Package **bvpSolve** (Soetaert et al., 2010a) implements three methods to solve boundary value problems. The simplest solution method is the *single shooting method*, which combines initial value problem integration with a nonlinear root finding algorithm (Press et al., 2007). Two more stable solution methods implement a mono implicit Runge-Kutta (MIRK) code, based on the FORTRAN code `twpbvpC` (Cash and Mazzia, 2005), and the collocation method, based on the FORTRAN code `colnew` (Bader and Ascher, 1987). Some boundary value problems can also be solved with functions from packages **ReacTran** and **rootSolve** (see below).

Partial differential equations

In contrast to ODEs where there is only one independent variable, partial differential equations (PDE) contain partial derivatives with respect to more than one independent variable, for instance t (time) and x (a spatial dimension). To distinguish this type of equations from ODEs, the derivatives are represented with the ∂ symbol, e.g.

$$\frac{\partial y}{\partial t} = f(t, x, y, \frac{\partial y}{\partial x}, p)$$

Partial differential equations can be solved by subdividing one or more of the continuous independent variables in a number of grid cells, and replacing the derivatives by discrete, algebraic approximate equations, so-called finite differences (cf. LeVeque, 2007; Hundsdorfer and Verwer, 2003).

For time-varying cases, it is customary to discretise the spatial coordinate(s) only, while time is left in continuous form. This is called the method-of-lines, and in this way, one PDE is translated into a large number of coupled ordinary differential equations, that can be solved with the usual initial value problem solvers (cf. Hamdi et al., 2007). This applies to parabolic PDEs such as the heat equation, and to hyperbolic PDEs such as the wave equation.

For time-invariant problems, usually all independent variables are discretised, and the derivatives approximated by algebraic equations, which are solved by root-finding techniques. This technique applies to elliptic PDEs.

R-package **ReacTran** provides functions to generate finite differences on a structured grid. After that, the resulting time-varying cases can be solved with specially-designed functions from package **deSolve**, while time-invariant cases can be solved with root-solving methods from package **rootSolve** .

Differential algebraic equations

Differential-algebraic equations (DAE) contain a mixture of differential (f) and algebraic equations (g), the latter e.g. for maintaining mass-balance conditions:

$$y' = f(t, y, p)$$
$$0 = g(t, y, p)$$

Important for the solution of a DAE is its index. The index of a DAE is the number of differentiations

needed until a system consisting only of ODEs is obtained.

Function `daspk` (Brenan et al., 1996) from package **deSolve** solves (relatively simple) DAEs of index at most 1, while function `radau` (Hairer et al., 2009) solves DAEs of index up to 3.

Implementation details

The implemented solver functions are explained by means of the `ode`-function, used for the solution of initial value problems. The interfaces to the other solvers have an analogous definition:

```
ode(y, times, func, parms, method = c("lsoda",
    "lsode", "lsodes", "lsodar",
    "vode", "daspk", "euler", "rk4",
    "ode23", "ode45", "radau", "bdf",
    "bdf_d", "adams", "impAdams",
    "impAdams_d"), ...)
```

To use this, the system of differential equations can be defined as an R-function (`func`) that computes derivatives in the ODE system (the model definition) according to the independent variable (e.g. time `t`). `func` can also be a function in a dynamically loaded shared library (Soetaert et al., 2010c) and, in addition, some solvers support also the supply of an analytically derived function of partial derivatives (Jacobian matrix).

If `func` is an R-function, it must be defined as:

```
func <- function(t, y, parms, ...)
```

where `t` is the actual value of the independent variable (e.g. the current time point in the integration), `y` is the current estimate of the variables in the ODE system, `parms` is the parameter vector and `...` can be used to pass additional arguments to the function.

The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `t`, and whose next elements are optional global values that can be recorded at each point in `times`. The derivatives must be specified in the same order as the state variables `y`.

Depending on the algorithm specified in argument `method`, numerical simulation proceeds either exactly at the time steps specified in `times`, or using time steps that are independent from `times` and where the output is generated by interpolation. With the exception of method `euler` and several fixed-step Runge-Kutta methods all algorithms have automatic time stepping, which can be controlled by setting accuracy requirements (see below) or by using optional arguments like `hini` (initial time step), `hmin` (minimal time step) and `hmax` (maximum time step). Specific details, e.g. about the applied interpolation methods can be found in the manual pages and the original literature cited there.

Numerical accuracy

Numerical solution of a system of differential equations is an approximation and therefore prone to numerical errors, originating from several sources:

1. time step and accuracy order of the solver,
2. floating point arithmetics,
3. properties of the differential system and stability of the solution algorithm.

For methods with automatic stepsize selection, accuracy of the computation can be adjusted using the non-negative arguments `atol` (absolute tolerance) and `rtol` (relative tolerance), which control the local errors of the integration.

Like R itself, all solvers use double-precision floating-point arithmetics according to IEEE Standard 754 (2008), which means that it can represent numbers between approx. $\pm 2.25\ 10^{-308}$ to approx. $\pm 1.8\ 10^{308}$ and with 16 significant digits. It is therefore not advisable to set **rtol** below 10^{-16}, except setting it to zero with the intention to use absolute tolerance exclusively.

The solvers provided by the packages presented below have proven to be quite robust in most practical cases, however users should always be aware about the problems and limitations of numerical methods and carefully check results for plausibility. The section "Troubleshooting" in the package vignette (Soetaert et al., 2010d) should be consulted as a first source for solving typical problems.

Examples

An initial value ODE

Consider the famous van der Pol equation (van der Pol and van der Mark, 1927), that describes a non-conservative oscillator with non-linear damping and which was originally developed for electrical circuits employing vacuum tubes. The oscillation is described by means of a 2^{nd} order ODE:

$$z'' - \mu(1 - z^2)z' + z = 0$$

Such a system can be routinely rewritten as a system of two 1^{st} order ODEs, if we substitute z'' with y_1' and z' with y_2:

$$y_1' = y_2$$
$$y_2' = \mu \cdot (1 - {y_1}^2) \cdot y_2 - y_1$$

There is one parameter, μ, and two differential variables, y_1 and y_2 with initial values (at $t = 0$):

$$y_{1_{(t=0)}} = 2$$
$$y_{2_{(t=0)}} = 0$$

The van der Pol equation is often used as a test problem for ODE solvers, as, for large μ, its dynamics consists of parts where the solution changes very slowly, alternating with regions of very sharp changes. This "stiffness" makes the equation quite challenging to solve.

In R, this model is implemented as a function (`vdpol`) whose inputs are the current time (`t`), the values of the state variables (`y`), and the parameters (`mu`); the function returns a `list` with as first element the derivatives, concatenated.

```
vdpol <- function (t, y, mu) {
  list(c(
    y[2],
    mu * (1 - y[1]^2) * y[2] - y[1]
  ))
}
```

After defining the initial condition of the state variables (`yini`), the model is solved, and output written at selected time points (`times`), using **deSolve**'s integration function `ode`. The default routine `lsoda`, which is invoked by `ode` automatically switches between stiff and non-stiff methods, depending on the problem (Petzold, 1983).

We run the model for a typically stiff (`mu = 1000`) and nonstiff (`mu = 1`) situation:

```
library(deSolve)
yini <- c(y1 = 2, y2 = 0)
stiff <- ode(y = yini, func = vdpol,
    times = 0:3000, parms = 1000)

nonstiff <- ode(y = yini, func = vdpol,
    times = seq(0, 30, by = 0.01),
    parms = 1)
```

The model returns a matrix, of class `deSolve`, with in its first column the time values, followed by the values of the state variables:

```
head(stiff, n = 3)
```

```
     time       y1            y2
[1,]    0 2.000000  0.0000000000
[2,]    1 1.999333 -0.0006670373
[3,]    2 1.998666 -0.0006674088
```

Figures are generated using the S3 plot method for objects of class `deSolve`:

```
plot(stiff, type = "l", which = "y1",
  lwd = 2, ylab = "y",
  main = "IVP ODE, stiff")

plot(nonstiff, type = "l", which = "y1",
  lwd = 2, ylab = "y",
  main = "IVP ODE, nonstiff")
```

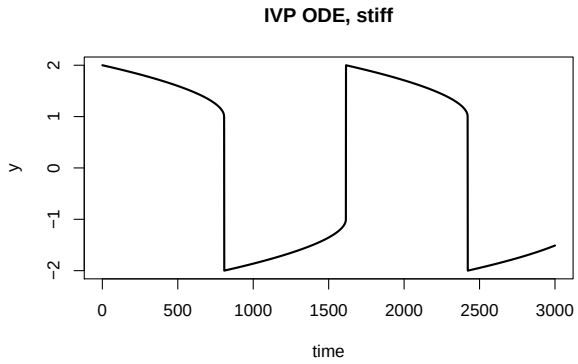

Figure 1: Solution of the van der Pol equation, an initial value ordinary differential equation, stiff case, $\mu = 1000$.

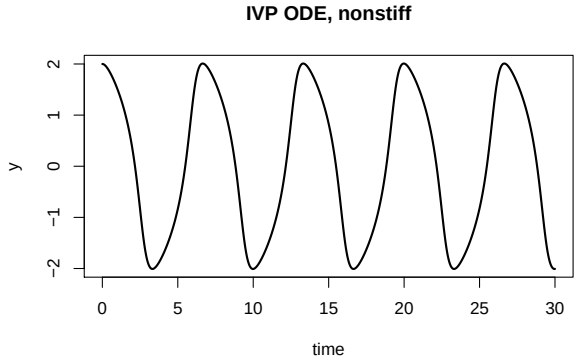

Figure 2: Solution of the van der Pol equation, an initial value ordinary differential equation, non-stiff case, $\mu = 1$.

solver	non-stiff	stiff
ode23	0.37	271.19
lsoda	0.26	0.23
adams	0.13	616.13
bdf	0.15	0.22
radau	0.53	0.72

Table 1: Comparison of solvers for a stiff and a non-stiff parametrisation of the van der Pol equation (time in seconds, mean values of ten simulations on an AMD AM2 X2 3000 CPU).

A comparison of timings for two explicit solvers, the Runge-Kutta method (`ode23`) and the `adams` method, with the implicit multistep solver (`bdf`, backward differentiation formula) shows a clear advantage for the latter in the stiff case (Figure 1). The default solver (`lsoda`) is not necessarily the fastest, but shows robust behavior due to automatic stiffness detection. It uses the explicit multistep Adams method for the non-stiff case and the BDF method for the stiff case. The accuracy is comparable for all

solvers with `atol` = `rtol` = 10^{-6}, the default.

A boundary value ODE

The webpage of Jeff Cash (Cash, 2009) contains many test cases, including their analytical solution (see below), that BVP solvers should be able to solve. We use equation no. 14 from this webpage as an example:

$$\xi y'' - y = -(\xi\pi^2 + 1)\cos(\pi x)$$

on the interval $[-1,1]$, and subject to the boundary conditions:

$$y_{(x=-1)} = 0$$
$$y_{(x=+1)} = 0$$

The second-order equation first is rewritten as two first-order equations:

$$y_1' = y_2$$
$$y_2' = 1/\xi \cdot (y_1 - (\xi\pi^2 + 1)\cos(\pi x))$$

It is implemented in R as:

```
Prob14 <- function(x, y, xi)  {
  list(c(
    y[2],
    1/xi * (y[1] - (xi*pi*pi+1) * cos(pi*x))
  ))
}
```

With decreasing values of ξ, this problem becomes increasingly difficult to solve. We use three values of ξ, and solve the problem with the shooting, the MIRK and the collocation method (Ascher et al., 1995).

Note how the initial conditions `yini` and the conditions at the end of the integration interval `yend` are specified, where `NA` denotes that the value is not known. The independent variable is called `x` here (rather than `times` in `ode`).

```
library(bvpSolve)
x <- seq(-1, 1, by = 0.01)
shoot <- bvpshoot(yini = c(0, NA),
    yend = c(0, NA), x = x, parms = 0.01,
    func = Prob14)

twp <- bvptwp(yini = c(0, NA), yend = c(0,
    NA), x = x, parms = 0.0025,
    func = Prob14)

coll <- bvpcol(yini = c(0, NA),
    yend = c(0, NA), x = x, parms = 1e-04,
    func = Prob14)
```

The numerical approximation generated by `bvptwp` is very close to the analytical solution, e.g. for $\xi = 0.0025$:

```
xi <- 0.0025
analytic <- cos(pi * x) + exp((x -
    1)/sqrt(xi)) + exp(-(x + 1)/sqrt(xi))
max(abs(analytic - twp[, 2]))
```

```
[1] 7.788209e-10
```

A similar low discrepancy ($4 \cdot 10^{-11}$) is noted for the $\xi = 0.0001$ as solved by `bvpcol`; the shooting method is considerably less precise ($1.4 \cdot 10^{-5}$), although the same tolerance (`atol` = 10^{-8}) was used for all runs.

The plot shows how the shape of the solution is affected by the parameter ξ, becoming more and more steep near the boundaries, and therefore more and more difficult to solve, as ξ gets smaller.

```
plot(shoot[, 1], shoot[, 2], type = "l", lwd = 2,
  ylim = c(-1, 1), col = "blue",
  xlab = "x", ylab = "y", main = "BVP ODE")
lines(twp[, 1], twp[, 2], col = "red", lwd = 2)
lines(coll[, 1], coll[, 2], col = "green", lwd = 2)
legend("topright", legend = c("0.01", "0.0025",
  "0.0001"), col = c("blue", "red", "green"),
  title = expression(xi), lwd = 2)
```

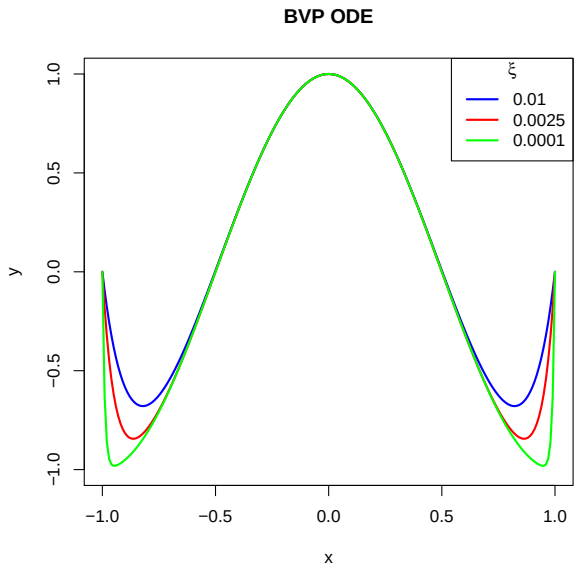

Figure 3: Solution of the BVP ODE problem, for different values of parameter ξ.

Differential algebraic equations

The so called "Rober problem" describes an autocatalytic reaction (Robertson, 1966) between three chemical species, y_1, y_2 and y_3. The problem can be formulated either as an ODE (Mazzia and Magherini, 2008), or as a DAE:

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$
$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 310^7 y_2^2$$
$$1 = y_1 + y_2 + y_3$$

where the first two equations are differential equations that specify the dynamics of chemical species y_1 and y_2, while the third algebraic equation ensures that the summed concentration of the three species remains 1.

The DAE has to be specified by the *residual function* instead of the rates of change (as in ODEs).

$$r_1 = -y_1' - 0.04y_1 + 10^4 y_2 y_3$$
$$r_2 = -y_2' + 0.04y_1 - 10^4 y_2 y_3 - 3\,10^7 y_2^2$$
$$r_3 = -1 + y_1 + y_2 + y_3$$

Implemented in R this becomes:

```
daefun<-function(t, y, dy, parms) {
    res1 <- - dy[1] - 0.04 * y[1] +
              1e4 * y[2] * y[3]
    res2 <- - dy[2] + 0.04 * y[1] -
              1e4 * y[2] * y[3] - 3e7 * y[2]^2
    res3 <- y[1] + y[2] + y[3] - 1
    list(c(res1, res2, res3),
         error = as.vector(y[1] + y[2] + y[3]) - 1)
}

yini  <- c(y1 = 1, y2 = 0, y3 = 0)
dyini <- c(-0.04, 0.04, 0)
times <- 10 ^ seq(-6,6,0.1)
```

The input arguments of function `daefun` are the current time (`t`), the values of the state variables and their derivatives (`y`, `dy`) and the parameters (`parms`). It returns the residuals, concatenated and an output variable, the error in the algebraic equation. The latter is added to check upon the accuracy of the results.

For DAEs solved with `daspk`, both the state variables and their derivatives need to be initialised (`y` and `dy`). Here we make sure that the initial conditions for `y` obey the algebraic constraint, while also the initial condition of the derivatives is consistent with the dynamics.

```
library(deSolve)
print(system.time(out <-daspk(y = yini,
  dy = dyini, times = times, res = daefun,
  parms = NULL)))

  user  system elapsed
  0.07    0.00    0.11
```

An S3 `plot` method can be used to plot all variables at once:

```
plot(out, ylab = "conc.", xlab = "time",
  type = "l", lwd = 2, log = "x")
mtext("IVP DAE", side = 3, outer = TRUE,
  line = -1)
```

There is a very fast initial change in concentrations, mainly due to the quick reaction between y_1 and y_2 and amongst y_2. After that, the slow reaction of y_1 with y_2 causes the system to change much more smoothly. This is typical for stiff problems.

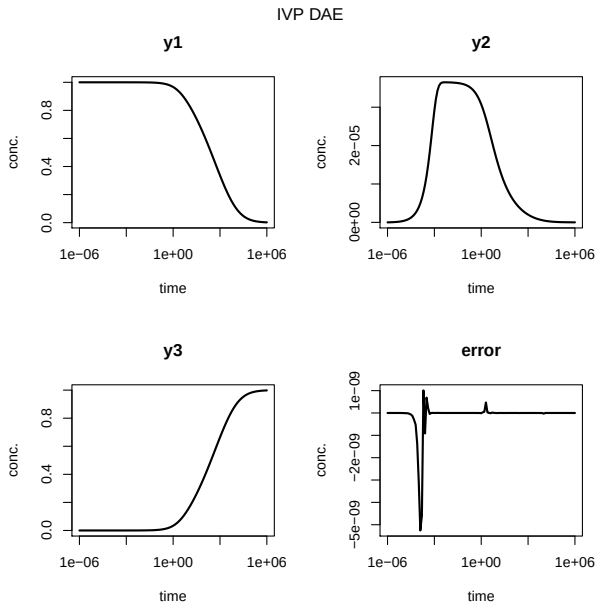

Figure 4: Solution of the DAE problem for the substances y_1, y_2, y_3; mass balance error: deviation of total sum from one.

Partial differential equations

In **partial differential equations** (PDE), the function has several independent variables (e.g. time and depth) and contains their partial derivatives.

Many partial differential equations can be solved by numerical approximation (finite differencing) after rewriting them as a set of ODEs (see Schiesser, 1991; LeVeque, 2007; Hundsdorfer and Verwer, 2003).

Functions `tran.1D`, `tran.2D`, and `tran.3D` from R package **ReacTran** (Soetaert and Meysman, 2010) implement finite difference approximations of the diffusive-advective transport equation which, for the 1-D case, is:

$$-\frac{1}{A_x} \cdot \left[\frac{\partial}{\partial x} A_x \left(-D \cdot \frac{\partial C}{\partial x} \right) - \frac{\partial}{\partial x} (A_x \cdot u \cdot C) \right]$$

Here D is the "diffusion coefficient", u is the "advection rate", and A_x is some property (e.g. surface area) that depends on the independent variable, x.

It should be noted that the accuracy of the finite difference approximations can not be specified in the **ReacTran** functions. It is up to the user to make sure that the solutions are sufficiently accurate, e.g. by including more grid points.

One dimensional PDE

Diffusion-reaction models are a fundamental class of models which describe how concentration of matter, energy, information, etc. evolves in space and time under the influence of diffusive transport and transformation (Soetaert and Herman, 2009).

As an example, consider the 1-D diffusion-reaction model in $[0,10]$:

$$\frac{\partial C}{\partial t} = \frac{\partial}{\partial x}\left(D \cdot \frac{\partial C}{\partial x}\right) - Q$$

with C the concentration, `t` the time, `x` the distance from the origin, `Q`, the consumption rate, and with boundary conditions (values at the model edges):

$$\frac{\partial C}{\partial x}_{x=0} = 0$$
$$C_{x=10} = C_{ext}$$

To solve this model in R, first the 1-D model `Grid` is defined; it divides 10 cm (`L`) into 1000 boxes (`N`).

```
library(ReacTran)
Grid <- setup.grid.1D(N = 1000, L = 10)
```

The model equation includes a transport term, approximated by **ReacTran** function `tran.1D` and a consumption term (`Q`). The downstream boundary condition, prescribed as a concentration (`C.down`) needs to be specified, the zero-gradient at the upstream boundary is the default:

```
pde1D <-function(t, C, parms)  {
  tran <- tran.1D(C = C, D = D,
                  C.down = Cext, dx = Grid)$dC
  list(tran - Q)  # return value: rate of change
}
```

The model parameters are:

```
D    <- 1    # diffusion constant
Q    <- 1    # uptake rate
Cext <- 20
```

In a first application, the model is solved to *steady-state*, which retrieves the condition where the concentrations are invariant:

$$0 = \frac{\partial}{\partial x}\left(D \cdot \frac{\partial C}{\partial x}\right) - Q$$

In R, steady-state conditions can be estimated using functions from package **rootSolve** which implement amongst others a Newton-Raphson algorithm (Press et al., 2007). For 1-dimensional models, `steady.1D` is most efficient. The initial "guess" of the steady-state solution (`y`) is unimportant; here we take simply `N` random numbers. Argument `nspec = 1` informs the solver that only one component is described.

Although a system of 1000 equations needs to be solved, this takes only a fraction of a second:

```
library(rootSolve)
print(system.time(
std   <- steady.1D(y = runif(Grid$N),
  func = pde1D, parms = NULL, nspec = 1)
))
```

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

The values of the state-variables (`y`) are plotted against the distance, in the middle of the grid cells (`Grid$x.mid`).

```
plot (Grid$x.mid, std$y, type = "l",
  lwd = 2, main = "steady-state PDE",
  xlab = "x", ylab = "C", col = "red")
```

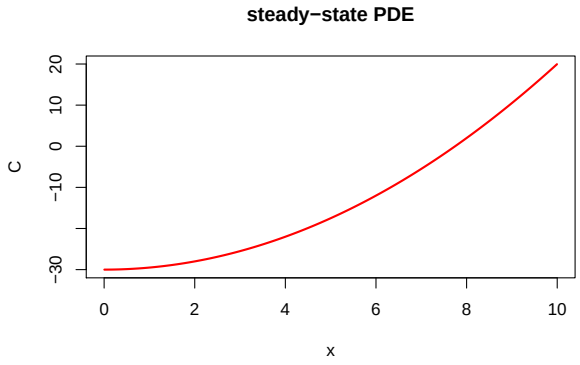

Figure 5: Steady-state solution of the 1-D diffusion-reaction model.

The analytical solution compares well with the numerical approximation:

```
analytical <- Q/2/D*(Grid$x.mid^2 - 10^2) + Cext
max(abs(analytical - std$y))
```

```
[1] 1.250003e-05
```

Next the model is run dynamically for 100 time units using **deSolve** function `ode.1D`, and starting with a uniform concentration:

```
require(deSolve)
times <- seq(0, 100, by = 1)
system.time(
  out <- ode.1D(y = rep(1, Grid$N),
    times = times, func = pde1D,
    parms = NULL, nspec = 1)
)
```

```
   user  system elapsed
   0.61    0.02    0.63
```

Here, `out` is a matrix, whose 1^{st} column contains the output times, and the next columns the values of the state variables in the different boxes; we print the first columns of the last three rows of this matrix:

```
tail(out[, 1:4], n = 3)
```

```
      time         1         2         3
[99,]   98 -27.55783 -27.55773 -27.55754
[100,]  99 -27.61735 -27.61725 -27.61706
[101,] 100 -27.67542 -27.67532 -27.67513
```

We plot the result using a blue-yellow-red color scheme, and using `deSolve`'s S3 method `image`. Figure 6 shows that, as time proceeds, gradients develop from the uniform distribution, until the system almost reaches steady-state at the end of the simulation.

```
image(out, xlab = "time, days",
      ylab = "Distance, cm",
      main = "PDE", add.contour = TRUE)
```

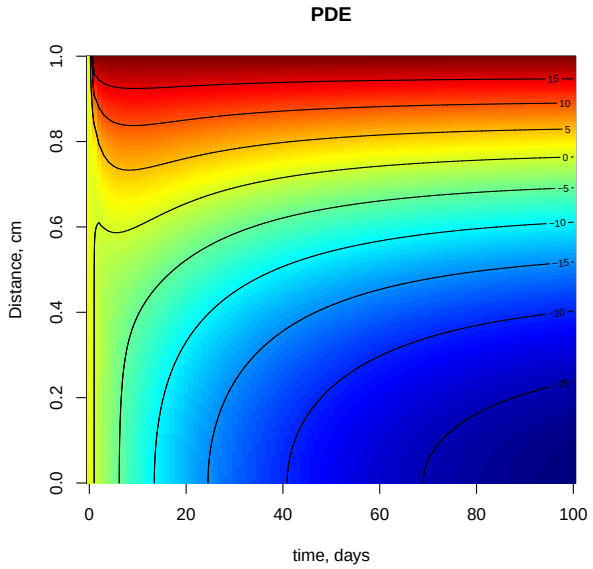

Figure 6: Dynamic solution of the 1-D diffusion-reaction model.

It should be noted that the steady-state model is effectively a boundary value problem, while the transient model is a prototype of a "parabolic" partial differential equation (LeVeque, 2007).

Whereas R can also solve the other two main classes of PDEs, i.e. of the "hyperbolic" and "elliptic" type, it is well beyond the scope of this paper to elaborate on that.

Discussion

Although R is still predominantly applied for statistical analysis and graphical representation, it is more and more suitable for mathematical computing, e.g. in the field of matrix algebra (Bates and Maechler, 2008). Thanks to the differential equation solvers, R is also emerging as a powerful environment for dynamic simulations (Petzoldt, 2003; Soetaert and Herman, 2009; Stevens, 2009).

The new package **deSolve** has retained all the funtionalities of its predecessor **odesolve** (Setzer, 2001), such as the potential to define models both in R code, or in compiled languages. However, compared to **odesolve**, it includes a more complete set of integrators, and a more extensive set of options to tune the integration routines, it provides more complete output, and has extended the applicability domain to include also DDEs, DAEs and PDEs.

Thanks to the DAE solvers `daspk` (Brenan et al., 1996) and `radau` (Hairer and Wanner, 2010) it is now also possible to model electronic circuits or equilibrium chemical systems. These problems are often of index ≤ 1. In many mechanical systems, physical constraints lead to DAEs of index up to 3, and these more complex problems can be solved with `radau`.

The inclusion of BVP and PDE solvers have opened up the application area to the field of reactive transport modelling (Soetaert and Meysman, 2010), such that R can now be used to describe quantities that change not only in time, but also along one or more spatial axes. We use it to model how ecosystems change along rivers, or in sediments, but it could equally serve to model the growth of a tumor in human brains, or the dispersion of toxicants in human tissues.

The open source matrix language R has great potential for dynamic modelling, and the tools currently available are suitable for solving a wide variety of practical and scientific problems. The performance is sufficient even for larger systems, especially when models can be formulated using matrix algebra or are implemented in compiled languages like C or Fortran (Soetaert et al., 2010b). Indeed, there is emerging interest in performing statistical analysis on differential equations, e.g. in package **nlmeODE** (Tornøe et al., 2004) for fitting non-linear mixed-effects models using differential equations, package **FME** (Soetaert and Petzoldt, 2010) for sensitivity analysis, parameter estimation and Markov chain Monte-Carlo analysis or package **ccems** for combinatorially complex equilibrium model selection (Radivoyevitch, 2008).

However, there is ample room for extensions and improvements. For instance, the PDE solvers are quite memory intensive, and could benefit from the implementation of sparse matrix solvers that are more efficient in this respect[2]. In addition, the methods implemented in **ReacTran** handle equations defined on very simple shapes only. Extending the PDE approach to finite elements (Strang and Fix, 1973) would open up the application domain of R to any irregular geometry. Other spatial discretisation schemes could be added, e.g. for use in fluid dynamics.

Our models are often applied to derive unknown parameters by fitting them against data; this relies on the availability of apt parameter fitting algorithms.

Discussion of these items is highly welcomed, in the new special interest group about dynamic mod-

[2] for instance, the "preconditioned Krylov" part of the `daspk` method is not yet supported

[3] `https://stat.ethz.ch/mailman/listinfo/r-sig-dynamic-models`

els[3] in R.

Bibliography

U. Ascher, R. Mattheij, and R. Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. Philadelphia, PA, 1995.

U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, 1998.

G. Bader and U. Ascher. A new basis implementation for a mixed order boundary value ODE solver. *SIAM J. Scient. Stat. Comput.*, 8:483–500, 1987.

D. Bates and M. Maechler. **Matrix***: A Matrix Package for R*, 2008. R package version 0.999375-9.

P. Bogacki and L. Shampine. A 3(2) pair of Runge-Kutta formulas. *Appl. Math. Lett.*, 2:1–9, 1989.

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.

P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. **VODE**, a variable-coefficient ode solver. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051, 1989.

J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations, Runge-Kutta and General Linear Methods*. Wiley, Chichester, New York, 1987.

J. R. Cash. *35 Test Problems for Two Way Point Boundary Value Problems*, 2009. URL http://www.ma.ic.ac.uk/~jcash/BVP_software/PROBLEMS.PDF.

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.

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.

A. Couture-Beil, J. T. Schnute, and R. Haigh. **PBSddesolve***: Solver for Delay Differential Equations*, 2010. R package version 1.08.11.

J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *J. Comput. Appl. Math.*, 6:19–26, 1980.

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

E. Hairer, S. P. Noørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems. Second Revised Edition*. Springer-Verlag, Heidelberg, 2009.

S. Hamdi, W. E. Schiesser, and G. W. Griffiths. Method of lines. *Scholarpedia*, 2(7):2859, 2007.

A. C. Hindmarsh. **ODEPACK**, a systematized collection of ODE solvers. In R. Stepleman, editor, *Scientific Computing, Vol. 1 of IMACS Transactions on Scientific Computation*, pages 55–64. IMACS / North-Holland, Amsterdam, 1983.

W. Hundsdorfer and J. Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations. Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2003.

S. M. Iacus. **sde***: Simulation and Inference for Stochastic Differential Equations*, 2008. R package version 2.0.3.

IEEE Standard 754. Ieee standard for floating-point arithmetic, Aug 2008.

A. A. King, E. L. Ionides, and C. M. Breto. **pomp***: Statistical Inference for Partially Observed Markov Processes*, 2008. R package version 0.21-3.

R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations, Steady State and Time Dependent Problems*. SIAM, 2007.

F. Mazzia and C. Magherini. *Test Set for Initial Value Problem Solvers, release 2.4*. Department of Mathematics, University of Bari, Italy, 2008. URL http://pitagora.dm.uniba.it/~testset. Report 4/2008.

L. R. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput.*, 4: 136–148, 1983.

T. Petzoldt. R as a simulation platform in ecological modelling. *R News*, 3(3):8–16, 2003.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.

P. J. Prince and J. R. Dormand. High order embedded Runge-Kutta formulae. *J. Comput. Appl. Math.*, 7:67–75, 1981.

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.

T. Radivoyevitch. Equilibrium model selection: dTTP induced R1 dimerization. *BMC Systems Biology*, 2:15, 2008.

H. H. Robertson. The solution of a set of reaction rate equations. In J. Walsh, editor, *Numerical Analysis: An Introduction*, pages 178–182. Academic Press, London, 1966.

W. E. Schiesser. *The Numerical Method of Lines: Integration of Partial Differential Equations*. Academic Press, San Diego, 1991.

R. W. Setzer. *The* **odesolve** *Package: Solvers for Ordinary Differential Equations*, 2001. R package version 0.1-1.

K. Soetaert. **rootSolve***: Nonlinear Root Finding, Equilibrium and Steady-State Analysis of Ordinary Differential Equations*, 2009. R package version 1.6.

K. Soetaert and P. M. J. Herman. *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform*. Springer, 2009. ISBN 978-1-4020-8623-6.

K. Soetaert and F. Meysman. **ReacTran***: Reactive Transport Modelling in 1D, 2D and 3D*, 2010. R package version 1.2.

K. Soetaert and T. Petzoldt. Inverse modelling, sensitivity and Monte Carlo analysis in R using package **FME**. *Journal of Statistical Software*, 33(3):1–28, 2010. URL http://www.jstatsoft.org/v33/i03/.

K. Soetaert, J. R. Cash, and F. Mazzia. **bvpSolve***: Solvers for Boundary Value Problems of Ordinary Differential Equations*, 2010a. R package version 1.2.

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

K. Soetaert, T. Petzoldt, and R. W. Setzer. *R Package* **deSolve***: Writing Code in Compiled Languages*, 2010c. **deSolve** vignette - R package version 1.8.

K. Soetaert, T. Petzoldt, and R. W. Setzer. *R Package* **deSolve***: Solving Initial Value Differential Equations*, 2010d. **deSolve** vignette - R package version 1.8.

M. H. H. Stevens. *A Primer of Ecology with R*. Use R Series. Springer, 2009. ISBN: 978-0-387-89881-0.

G. Strang and G. Fix. *An Analysis of The Finite Element Method*. Prentice Hall, 1973.

C. W. Tornøe, H. Agersø, E. N. Jonsson, H. Madsen, and H. A. Nielsen. Non-linear mixed-effects pharmacokinetic/pharmacodynamic modelling in nlme using differential equations. *Computer Methods and Programs in Biomedicine*, 76:31–40, 2004.

B. van der Pol and J. van der Mark. Frequency demultiplication. *Nature*, 120:363–364, 1927.

Karline Soetaert
Netherlands Institute of Ecology
K.Soetaert@nioo.knaw.nl

Thomas Petzoldt
Technische Universität Dresden
Thomas.Petzoldt@tu-dresden.de

R. Woodrow Setzer
US Environmental Protection Agency
Setzer.Woodrow@epamail.epa.gov

Table 2: Summary of the main functions that solve differential equations.

Function	Package	Description
ode	deSolve	IVP of ODEs, full, banded or arbitrary sparse Jacobian
ode.1D	deSolve	IVP of ODEs resulting from 1-D reaction-transport problems
ode.2D	deSolve	IVP of ODEs resulting from 2-D reaction-transport problems
ode.3D	deSolve	IVP of ODEs resulting from 3-D reaction-transport problems
daspk	deSolve	IVP of DAEs of index ≤ 1, full or banded Jacobian
radau	deSolve	IVP of DAEs of index ≤ 3, full or banded Jacobian
dde	PBSddesolve	IVP of delay differential equations, based on Runge-Kutta formulae
dede	deSolve	IVP of delay differential equations, based on Adams and BDF formulae
bvpshoot	bvpSolve	BVP of ODEs; the shooting method
bvptwp	bvpSolve	BVP of ODEs; mono-implicit Runge-Kutta formula
bvpcol	bvpSolve	BVP of ODEs; collocation formula
steady	rootSolve	steady-state of ODEs; full, banded or arbitrary sparse Jacobian
steady.1D	rootSolve	steady-state of ODEs resulting from 1-D reaction-transport problems
steady.2D	rootSolve	steady-state of ODEs resulting from 2-D reaction-transport problems
steady.3D	rootSolve	steady-state of ODEs resulting from 3-D reaction-transport problems
tran.1D	ReacTran	numerical approximation of 1-D advective-diffusive transport problems
tran.2D	ReacTran	numerical approximation of 2-D advective-diffusive transport problems
tran.3D	ReacTran	numerical approximation of 3-D advective-diffusive transport problems

Table 3: Summary of the auxilliary functions that solve differential equations.

Function	Package	Description
lsoda	deSolve	IVP ODEs, full or banded Jacobian, automatic choice for stiff or non-stiff method
lsodar	deSolve	same as lsoda, but includes a root-solving procedure.
lsode, vode	deSolve	IVP ODEs, full or banded Jacobian, user specifies if stiff or non-stiff
lsodes	deSolve	IVP ODEs, arbitrary sparse Jacobian, stiff method
rk4, rk, euler	deSolve	IVP ODEs, using Runge-Kutta and Euler methods
zvode	deSolve	IVP ODEs, same as vode, but for complex variables
runsteady	rootSolve	steady-state ODEs by dynamically running, full or banded Jacobian
stode	rootSolve	steady-state ODEs by Newton-Raphson method, full or banded Jacobian
stodes	rootSolve	steady-state ODEs by Newton-Raphson method, arbitrary sparse Jacobian