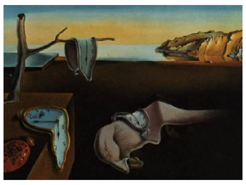

Journal of Statistical Software

February 2010, Volume 33, Issue 9. *http://www.jstatsoft.org/*

Solving Differential Equations in R: Package deSolve

Karline Soetaert
Netherlands Institute of Ecology

Thomas Petzoldt
Technische Universität Dresden

R. Woodrow Setzer
US Environmental Protection Agency

Abstract

In this paper we present the R package **deSolve** to solve initial value problems (IVP) written as ordinary differential equations (ODE), differential algebraic equations (DAE) of index 0 or 1 and partial differential equations (PDE), the latter solved using the method of lines approach. The differential equations can be represented in R code or as compiled code. In the latter case, R is used as a tool to trigger the integration and post-process the results, which facilitates model development and application, whilst the compiled code significantly increases simulation speed. The methods implemented are efficient, robust, and well documented public-domain Fortran routines. They include four integrators from the **ODEPACK** package (LSODE, LSODES, LSODA, LSODAR), DVODE and DASPK2.0. In addition, a suite of Runge-Kutta integrators and special-purpose solvers to efficiently integrate 1-, 2- and 3-dimensional partial differential equations are available. The routines solve both stiff and non-stiff systems, and include many options, e.g., to deal in an efficient way with the sparsity of the Jacobian matrix, or finding the root of equations. In this article, our objectives are threefold: (1) to demonstrate the potential of using R for dynamic modeling, (2) to highlight typical uses of the different methods implemented and (3) to compare the performance of models specified in R code and in compiled code for a number of test cases. These comparisons demonstrate that, if the use of loops is avoided, R code can efficiently integrate problems comprising several thousands of state variables. Nevertheless, the same problem may be solved from 2 to more than 50 times faster by using compiled code compared to an implementation using only R code. Still, amongst the benefits of R are a more flexible and interactive implementation, better readability of the code, and access to R's high-level procedures. **deSolve** is the successor of package **odesolve** which will be deprecated in the future; it is free software and distributed under the GNU General Public License, as part of the R software project.

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

1. Introduction

Many phenomena in science and engineering can be mathematically represented as initial value problems (IVP) of *ordinary differential equations* (ODE, Asher and Petzold 1998). ODEs describe how a certain quantity changes as a function of time or space, or some other variable (called the independent variable). They can be mathematically represented as:

$$y' = f(y, v, t)$$

where y are the differential variables, y' are the derivatives, v are other variables, and t is the independent variable. For the remainder, we will assume that the independent variable is "time". For this equation to have a solution, an extra condition is required. Here we deal only with models where some initial condition (at $t = t_0$) is specified:

$$y(t_0) = c$$

These are called *initial value problems* (IVP). The formalism above provides an *explicit* expression for y' as a function of y, x and t. A more general mathematical form is the *implicit* expression:

$$0 = G(y', y, v, t) \tag{1}$$

If, in addition to the ordinary differential equations, the differential variables obey some algebraic constraints at each time point:

$$0 = g(y, v, t) \tag{2}$$

then we obtain a set of *differential algebraic equations* (DAE). The two previous functions G (eq. 1) and g (eq. 2) can be combined to a new function F:

$$0 = F(y', y, v, t)$$

which is the formalism that we will use in this paper. Solving a DAE is more complex than solving an ODE. For instance, the initial conditions for a DAE must be chosen to be consistent. This is, the initial values of t, y and y', must obey:

$$0 = F(y'(t_0), y(t_0), v, t_0)$$

DAEs are commonly encountered in a number of scientific and engineering disciplines, e.g., in the modelling of electrical circuits or mechanical systems, in constrained variational problems, or in equilibrium chemistry (e.g., Brenan, Campbell, and Petzold 1996).

Most of the ODEs and DAEs are complicated enough to preclude finding an analytical solution, and therefore they are solved by *numerical* techniques, which calculate the solution only at a limited number of values of the independent variable (t).

A common theme in many of the numerical solvers, are their capabilities to solve *"stiff"* ODE or DAE problems. Formally, if the eigenvalue spectrum of the ODE system (i.e., of its Jacobian, see below) is large, the ODE system is said to be stiff (Hairer and Wanner 1980). As a less formal definition, an ODE system is called stiff if the problem changes on a wide variety of time scales, i.e., it contains both very rapidly and very slowly changing terms. Unless these stiff problems are solved with especially-designed methods, they require an excessive amount of computing time, as they need to use very small time steps to satisfy stability requirements (Press, Teukolsky, Vetterling, and Flannery 2007, p. 931).

Very often, stiff systems are most efficiently solved with implicit methods, which require the creation of a *Jacobian* matrix ($\frac{\partial f}{\partial y}$) and the solution of a system of equations involving this Jacobian. As we will see below, there is much to be gained by taking advantage of the sparsity of the Jacobian matrix. Except for the Runge-Kutta methods, all solvers implemented in **deSolve** are variable order, variable step methods, that use the backward differentiation formulas and Adams methods, two important families of multistep methods (Asher and Petzold 1998).

The remainder of the paper is organized as follows. In Section 2, the different solvers are briefly discussed and some implementation issues noted. Section 3 gives some example implementations of ODE, PDE and DAE systems in R (R Development Core Team 2009). In Section 4, we demonstrate how to implement the models in a compiled language. Numerical benchmarks of computational performance are conducted in Section 5. Finally, concluding remarks are given in Section 6.

The package is available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=deSolve`.

2. The integration routines

2.1. Implementation issues

The R package **deSolve** (Soetaert, Petzoldt, and Setzer 2009) is the successor of package **odesolve** (Setzer 2001), which will be deprecated in the future. Compared to **odesolve**, it includes a more complete set of integrators, a more extensive set of options to tune the integration routines, and provides more complete output. For instance, there was no provision to specify the structure of the Jacobian in **odesolve**'s routine `lsoda`, whereas this is now fully supported in **deSolve**. Moreover, as the applicability domain of the new package now includes DAEs and PDEs, the name **odesolve** was considered too narrow, warranting a new one (**deSolve**). Several integration methods in **deSolve** implement efficient, robust, and frequently-used open source routines. These routines are similar to one another and well documented, both in the source code, or by separate works (e.g., Brenan *et al.* 1996, for DASSL). The `lsoda` (Petzold 1983) implementation in **deSolve** is fully compatible with that in **odesolve** for systems coded fully in R. However, the calling sequence for systems using native language calls has changed between **odesolve** and **deSolve**. In the current version, the solvers take care of forcing function interpolation in compiled code. They also support events and time lags.

Functions `lsode`, `lsodes`, `lsoda`, and `lsodar` are R implementations of Fortran routines with the same name belonging to the **ODEPACK** collection (Hindmarsh 1983); functions `vode` and `zvode` implements the Fortran functions VODE and ZVODE (Brown, Byrne, and Hindmarsh 1989); function `daspk` implements the Fortran DASPK2.0 code (Brown, Hindmarsh, and Petzold 1994).

The collaboration between the three authors was greatly facilitated by the use of R-Forge (Theußl and Zeileis 2009, `http://R-Forge.R-project.org/`), the framework for R project developers, based on GForge (Copeland, Mas, McCullagh, Perdue, Smet, and Spisser 2006).

2.2. Integration options

When calling the integration routines, many options can be specified. We have tried, as far as possible, to retain the flexibility of the original codes; for most applications the defaults will do.

The sparsity structure of the *Jacobian* is crucial to efficiently solve (moderately) large systems of stiff equations. Sparse Jacobians can not only be generated much faster (fewer function calls), storing only the nonzero elements greatly reduces memory requirements. In addition, the resulting equations can be much more efficiently solved if the sparsity is taken into account.

Therefore, users have the option to specify whether the Jacobian has certain particular properties. By default it is considered to be a full matrix which is calculated by the solver, by numerical differencing (i.e., where the function gradient is estimated by successive perturbation of the state variables). To take advantage of a sparse Jacobian, the solver can be informed of any sparsity patterns. Thus, it is possible to specify that the Jacobian has a banded structure (`vode`, `daspk`, `lsode`, `lsoda`), or to use a more general sparse Jacobian (`lsodes`). In the latter case, `lsodes` can by itself determine the sparsity, but the user can also provide a matrix with row and column positions of its nonzero elements. In addition, `ode.1D, ode.2D` and `ode.3D` are specially designed to deal efficiently with the sparsity that arises in (PDE) models described in 1-, 2- and 3 (spatial) dimensions. With the exception of the Runge-Kutta solvers, all integration methods also provide the specification of an analytical Jacobian as an option, which may improve performance. Note, though, that a clear advantage of the finite difference approximation is that it is simpler.

Other important options are `rtol` and `atol`, relative and absolute *tolerances* that define error control. They are important because they not only affect the integration step taken, but also the numerical differencing used in the creation of the Jacobian.

2.3. A short description of the integrators

All `lsode`-type codes, `vode` and `daspk` use the variable-step, variable-order backward differentiation formula (BDF), suitable for solving stiff problems (order 1–5). The `lsode` family and `vode` also contain variable-step, variable-order Adams methods (order 1–12), which are well suited for nonstiff problems (Asher and Petzold 1998).

In detail:

- `ode`, `ode.1D, ode.2D` and `ode.3D` are wrappers around the integration routines described below. The latter three are especially designed to solve partial differential equations, where, in addition to the time derivative, the components also change in one, two or three (spatial) dimensions.

- `lsoda` automatically selects a stiff or nonstiff method. It may switch between the two methods during the simulation, in case the stiffness of the system changes. This is the default method used in `ode` and especially well suited for simple problems.

- `lsodar` is similar to `lsoda` but includes a method to find the root of a function.

- `lsode`, and `vode` also solve stiff and nonstiff problems. However, the user must decide whether a stiff or nonstiff method is most suited for a particular problem and select an appropriate solution method. `zvode` is a variant of `vode` that solves equations involving variables that are complex numbers. `lsode` is the default solver used in `ode.1D`
- `lsodes` exploits the sparsity in the Jacobian matrix by using linear algebra routines from the Yale sparse matrix package (Eisenstat, Gursky, Schultz, and Sherman 1982). It can determine the sparsity on its own or take it as input. Especially for large stiff problems with few interactions between state variables (leading to sparse Jacobians), dramatic savings in computing time can be achieved when using `lsodes`. It is the solver used in `ode.2D` and `ode.3D`
- `daspk` is the only integrator in the package that also solves differential algebraic equations of index zero and one. It can also solve ODEs.
- Finally, the package also includes solvers for several methods of the Runge-Kutta family (`rk`), with variable or fixed time steps. This includes the classical 4th order Runge-Kutta and the Euler method (`rk4`, `euler`).

  In addition, sets of coefficients (Butcher tableaus) for the most common Runge-Kutta-methods are availabe in function `rkMethod`, e.g., Heun's method, Bogacki-Shampine 2(3), Runge-Kutta-Fehlberg 4(5), Cash-Karp 4(5) or Dormand-Prince 4(5)7, and it is possible to provide user-specified tableaus of coefficients (for Details see Dormand and Prince 1981; Butcher 1987; Bogacki and Shampine 1989; Cash and Karp 1990; Press *et al.* 2007).

2.4. Output

All solvers return an array that contains, in its columns, the time values (1^{st} column) and the values of all state variables (subsequent columns) followed by the ordinary output variables (if any). This format is particularly suited for graphical routines of R (e.g., `matplot`). In addition, a `plot` method is included which, for models with not too many state variables, plots all output in one figure.

All Fortran codes have in common that they monitor essential properties of the integration, such as the number of Jacobian evaluations, the number of time steps performed, the number of integration error test failures, the stepsize to be attempted on the next step and so on. These performance indicators can be called upon by a method called `diagnostics`.

3. Examples: Model implementations in R

In this section we first implement a simple biological model, the Lotka-Volterra consumer-prey model, which is solved with the integration routine `ode` (which uses method `lsoda`). This model is then extended with a root function that stops the simulation at steady-state and which uses routine `lsodar`. An implementation in a 1-D and 2-D setting, demonstrates the capabilities of `ode.1D` and `ode.2D`. Finally, we end with a simple DAE model. Many more examples can be found in the **deSolve** package example files. Each model was run on a 2.5 GHz portable pc with an Intel Core 2 Duo T9300 processor and 3 GB of RAM. The CPU-times reported were estimated as the mean of 10 runs as follows:

```
R> print(system.time(for(i in 1:10)
+    out <- ode(func = LVmod, y = yini, parms = pars, times = times)
+  )/10)
```

3.1. A simple consumer-prey model

We start with a simple Lotka-Volterra type of model (Lotka 1925; Volterra 1926) describing consumer-prey interactions:

$$\frac{d\mathrm{P}}{dt} = r_G \cdot \mathrm{P} \cdot \left(1 - \frac{\mathrm{P}}{K}\right) - r_I \cdot \mathrm{P} \cdot \mathrm{C}$$
$$\frac{d\mathrm{C}}{dt} = k_{AE} \cdot r_I \cdot \mathrm{P} \cdot \mathrm{C} - r_M \cdot \mathrm{C}$$

Where P and C are prey and consumer concentrations, r_G is the growth rate of prey, K the carrying capacity, r_I the ingestion rate of the consumer, k_{AE} its assimilation efficiency and r_M the consumer's mortality rate. The implementation in R consists of three parts:

First the model function is defined. Here this function is called `LVmod0D`. It takes as input the current simulation time, the values of the state variables, and the model parameters. These three arguments have to be always present, and in this order; other arguments can be added after them. The solver will call this function at each time step during the integration process; at which `Time` contains the current simulation time; `State` the current value of the state variables and `Pars` the values of the parameters as passed to the solver.

Both `State` and `Pars` are a vector, with named elements; the `with` statement, allows using their names within the function.

The model returns a list, where the first element contains the derivatives, concatenated. Note that `Time` is not used here, but in many models, it is used, e.g., when there are external variables that depend on time.

```
R> LVmod0D <- function(Time, State, Pars) {
+    with(as.list(c(State, Pars)), {
+      IngestC <- rI * P * C
+      GrowthP <- rG * P * (1 - P/K)
+      MortC   <- rM * C
+
+      dP    <- GrowthP - IngestC
+      dC    <- IngestC * AE - MortC
+
+      return(list(c(dP, dC)))
+    })
+  }
```

Then the parameters are given a name and a value (`pars`), the state variables initialized (`yini`) and the time points at which we want output specified (`times`). Based on these inputs, the model is simulated. Here we use the default integration function `ode`, which is based on the `lsoda` method; its returned model output is written in a `matrix` called `out`. We print the

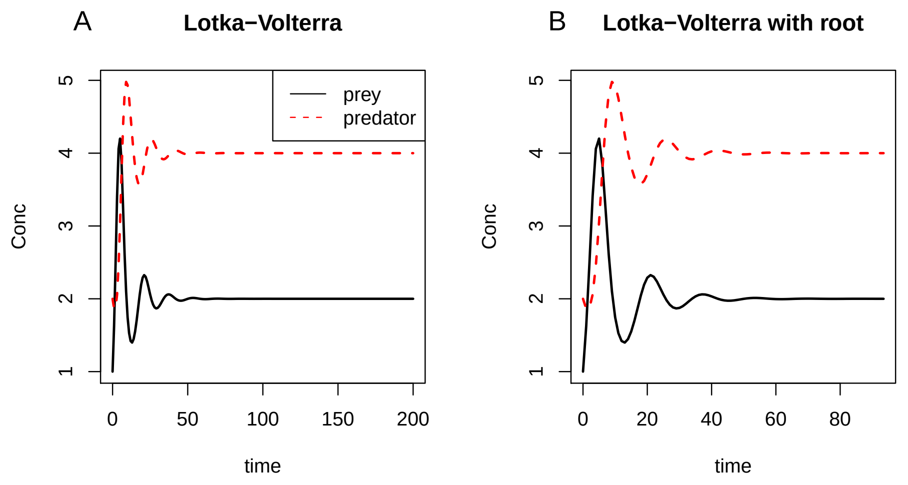

Figure 1: A. Results of the Lotka-Volterra model. B. The Lotka-Volterra model solved till steady-state.

first part of this matrix (`head(out)`). Matrix `out` has in its first column the time sequence, and in its next columns the prey and consumer concentrations.

```
R> pars <- c(rI = 0.2, rG = 1.0, rM = 0.2, AE = 0.5, K  = 10)
R> yini <- c(P = 1, C = 2)
R> times <- seq(0, 200, by = 1)
R> print(system.time(
+    out <- ode(func = LVmodOD, y = yini, parms = pars, times = times)))

   user  system elapsed
   0.04    0.00    0.04

R> head(out, n = 3)

     time        P        C
[1,]    0 1.000000 2.000000
[2,]    1 1.626853 1.863283
[3,]    2 2.488467 1.871156
```

Finally, the model output is plotted, using R function `matplot`.

```
R> matplot(out[,"time"], out[,2:3], type = "l", xlab = "time", ylab = "Conc",
+    main = "Lotka-Volterra", lwd = 2)
R> legend("topright", c("prey", "consumer"), col = 1:2, lty = 1:2)
```

The results (Figure 1A) clearly show that, after initial fluctuations, the consumer and prey concentrations reach a steady-state. It takes 0.04 (`lsoda, daspk`) and 0.02 (`lsode, vode, lsodes`) seconds to solve this model.

3.2. The consumer-prey model with stopping criterion

Supposing that we are interested in the initial phase only, we use the root finding functionality of function `lsodar` to halt the simulation after the state variables change less than some predefined amount (here 10^{-4}). Below, we define the root function (`rootfun`), which first estimates the rate of change and then calculates the difference between the sum of absolute values and a given tolerance (10^{-4}). If we run `lsodar`, the integration will stop if the sum of absolute values equals 10^{-4}; this is after 93.5 days. The results are depicted in Figure 1B; it takes 0.05 seconds to complete. This is slightly longer than `lsoda` takes to simulate the system for 200 days (0.04 seconds), since `LVmod0D` is called twice as often: once to evaluate the derivative and once to evaluate the root.

```
R> rootfun <- function(Time, State, Pars) {
+    dstate <- unlist(LVmod0D(Time, State, Pars))
+    sum(abs(dstate)) - 1e-4
+ }
R> out <- lsodar(func = LVmod0D, y = yini, parms = pars,
+    times = times, rootfun = rootfun)
R> tail(out, n = 2)
```

```
          time        P        C
[94,] 93.00000 1.999809 4.000187
[95,] 93.49872 1.999795 4.000148
```

3.3. Consumer and prey dispersing on a 1-D grid

Now the same consumer-prey dynamics is implemented on a 1-dimensional grid, and including dispersion with dispersion coefficient Da (Crank 1975). The extended formulations are:

$$\begin{aligned}\frac{\partial \mathrm{P}}{\partial t} &= \frac{\partial}{\partial x} Da \frac{\partial \mathrm{P}}{\partial x} + r_G \cdot \mathrm{P} \cdot \left(1 - \frac{\mathrm{P}}{K}\right) - r_I \cdot \mathrm{P} \cdot \mathrm{C} \\ \frac{\partial \mathrm{C}}{\partial t} &= \frac{\partial}{\partial x} Da \frac{\partial \mathrm{C}}{\partial x} + k_{AE} \cdot r_I \cdot \mathrm{P} \cdot \mathrm{C} - r_M \cdot \mathrm{C}\end{aligned}$$

These partial differential equations (PDE) are solved by discretizing in space first and integrating the resulting initial value ODE; this is a technique called the "method of lines" (Schiesser 1991). It is beyond the scope of this paper to derive how the spatial derivative is numerically approximated and implemented in R; interested readers may refer to Soetaert and Herman (2009) where this is discussed. Also, package **ReacTran** (Soetaert and Meysman 2009), implements numerical approximations of spatial derivatives. The function below implements the model. Essentially, we first estimate the dispersive fluxes on the box interfaces ($FluxP$, $FluxC$) as $Flux = -Da \cdot \frac{\partial C}{\partial x}$, after which the rate of change is estimated as the negative of the flux gradient ($\frac{\partial C}{\partial t} = -\frac{\partial Flux}{\partial x} + ...$). Estimating a gradient is best done with R function `diff`, which avoids the use of explicit loops, and is computationally very efficient. Note that, by imposing `P[1]` and `P[N]` at the upper and lower boundaries, we effectively impose a zero-gradient (or a zero-flux) boundary condition.

```
R> LVmod1D <- function (time, state, parms, N, Da, dx) {
+    with (as.list(parms), {
+      P <- state[1:N]
+      C <- state[-(1:N)]
+
+      ## Dispersive fluxes; zero-gradient boundaries
+      FluxP <- -Da * diff(c(P[1], P, P[N]))/dx
+      FluxC <- -Da * diff(c(C[1], C, C[N]))/dx
+
+      ## Biology: Lotka-Volterra dynamics
+      IngestC  <- rI * P * C
+      GrowthP  <- rG * P * (1- P/K)
+      MortC    <- rM * C
+
+      ## Rate of change = -Flux gradient + Biology
+      dP   <- -diff(FluxP)/dx + GrowthP - IngestC
+      dC   <- -diff(FluxC)/dx + IngestC * AE - MortC
+
+      return (list(c(dP, dC)))
+    })
+  }
```

The 20 meter model domain (`R`) is subdivided in 1000 boxes (`N`). After giving values to the box sizes (`dx`) and the dispersion coefficient (`Da`), and initializing the consumer and prey concentrations in each box (`yini`), the model can be solved for the requested time sequence (`times`). It is most efficient to do this with integration routine `ode.1D`, which is especially designed for solving this type of problems. Notwithstanding the large number of state variables (2000), it takes less than a second to run this model for 200 days.

```
R> R <- 20
R> N <- 1000
R> dx <- R/N
R> Da <- 0.05
R> yini <- rep(0, 2*N)
R> yini[500:501] <- yini[1500:1501] <- 10
R> times <-seq(0, 200, by = 1)
R> print(system.time(
+    out <- ode.1D(y = yini, times = times, func = LVmod1D,
+      parms = pars, nspec = 2, N = N, dx = dx, Da = Da)
+  ))

   user  system elapsed
   0.81    0.02    0.82
```

The matrix `out` has in its first column the time sequence, followed by 1000 columns with prey concentrations, one for each box, followed by 1000 columns with consumer concentrations. We plot only the prey concentrations (Figure 2).

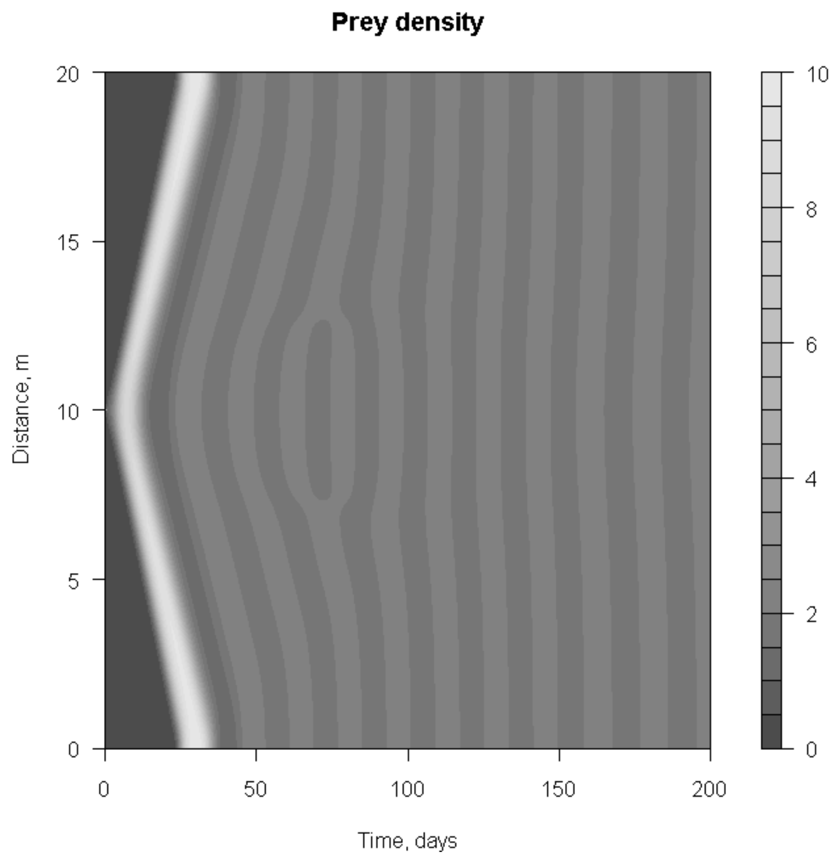

Figure 2: Results of the Lotka-Volterra model on a one-dimensional grid.

```
R> P   <- out[,2:(N + 1)]
R> filled.contour(x = times, z = P, y = seq(0, R, length=N),
+    color = gray.colors, xlab = "Time, days", ylab= "Distance, m",
+    main = "Prey density")
```

Function `ode.1D` was run using either `lsode`, `vode`, `lsoda` and `lsodes` as the integrator; it took 0.8 (`lsode`), 0.85 (`vode`), 1.2 (`lsoda`) and 0.65 (`lsodes`) seconds to finish the run.

3.4. Consumer and prey dispersing on a 2-D grid

Finally, we also implement the same consumer-prey dynamics on a 2-dimensional grid. The extended formulations now include dispersion in the x- and y-direction (dispersion coefficient Da):

$$\frac{\partial \mathrm{P}}{\partial t} = \frac{\partial}{\partial x} Da \frac{\partial \mathrm{P}}{\partial x} + \frac{\partial}{\partial y} Da \frac{\partial \mathrm{P}}{\partial y} + r_G \cdot \mathrm{P} \cdot \left(1 - \frac{\mathrm{P}}{K}\right) - r_I \cdot \mathrm{P} \cdot \mathrm{C}$$
$$\frac{\partial \mathrm{C}}{\partial t} = \frac{\partial}{\partial x} Da \frac{\partial \mathrm{C}}{\partial x} + \frac{\partial}{\partial y} Da \frac{\partial \mathrm{C}}{\partial y} + k_{AE} \cdot r_I \cdot \mathrm{P} \cdot \mathrm{C} - r_M \cdot \mathrm{C}$$

The function below implements this 2-D model. Note that, as for the 1-D case, the use of explicit looping is avoided: to estimate the gradient, we just subtract two matrices shifted with one row (x-direction) or one column (y-direction). The zero-fluxes at the boundaries are implemented by binding a row or column of 0-values (`zero`).

```
R> LVmod2D <- function (time, state, parms, N, Da, dx, dy) {
+    P <- matrix(nr = N, nc = N, state[1:NN])
+    C <- matrix(nr = N, nc = N, state[-(1:NN)])
+
+    with (as.list(parms), {
+      dP    <- rG * P *(1 - P/K) - rI * P *C
+      dC    <- rI * P * C * AE - rM * C
+
+      zero  <- numeric(N)
+
+      ## Fluxes in x-direction; zero fluxes near boundaries
+      FluxP <- rbind(zero, -Da * (P[-1,] - P[-N,])/dx, zero)
+      FluxC <- rbind(zero, -Da * (C[-1,] - C[-N,])/dx, zero)
+
+      dP    <- dP - (FluxP[-1,] - FluxP[-(N+1),])/dx
+      dC    <- dC - (FluxC[-1,] - FluxC[-(N+1),])/dx
+
+      ## Fluxes in y-direction
+      FluxP <- cbind(zero, -Da * (P[,-1] - P[,-N])/dy, zero)
+      FluxC <- cbind(zero, -Da * (C[,-1] - C[,-N])/dy, zero)
+
+      dP    <- dP - (FluxP[,-1] - FluxP[,-(N+1)])/dy
+      dC    <- dC - (FluxC[,-1] - FluxC[,-(N+1)])/dy
+
+      return(list(c(as.vector(dP), as.vector(dC))))
+    })
+  }
```

The 2-D model domain, extending 20 meters (`R`) in both x- and y-directions is subdivided into $50 \cdot 50$ boxes (`N`). This model can be solved efficiently only with integrator `ode.2D`. Here we need to specify the dimensionality of the model (`dimens`) and the length of the work array (`lrw`). It takes less than 3 seconds to solve this 5000 state-variable model for 200 days.

```
R> R  <- 20
R> N  <- 50
R> dx <- R/N
R> dy <- R/N
R> Da <- 0.05
R> NN <- N * N
R> yini     <- rep(0, 2 * N * N)
R> cc       <- c((NN/2):(NN/2 + 1) + N/2, (NN/2):(NN/2 + 1) - N/2)
R> yini[cc] <- yini[NN + cc] <- 10
```

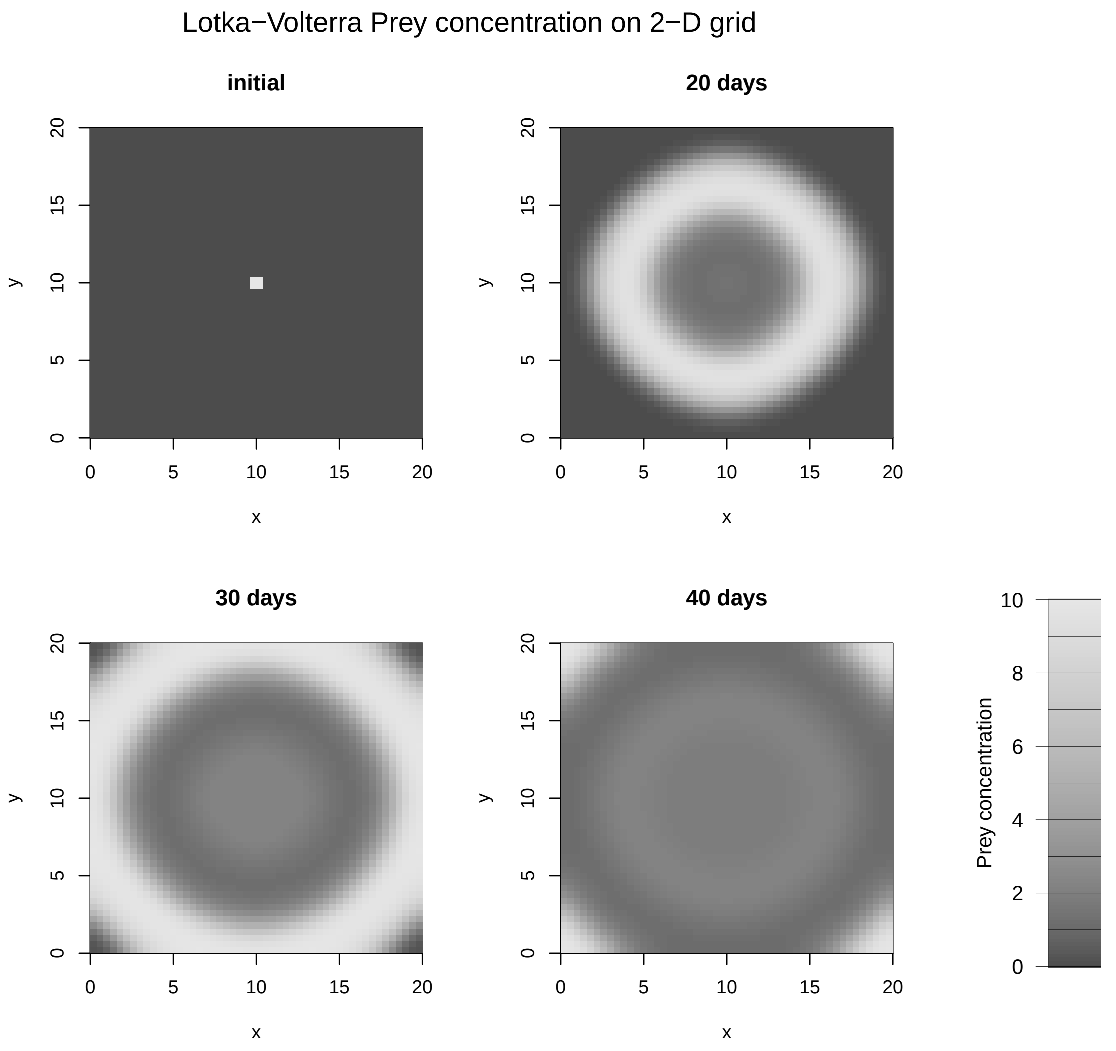

Figure 3: Results of the Lotka-Volterra model in a two-dimensional grid.

```
R> times  <- seq(0, 200, by = 1)
R> print(system.time(
+    out <- ode.2D(y = yini, times = times, func = LVmod2D,
+                  parms = pars, dimens = c(N, N), N = N,
+                  dx = dx, dy = dy, Da = Da, ynames = FALSE, lrw = 440000)
+  ))

   user  system elapsed 
   2.48    0.05    2.55 
```

Results are in Figure 3 for the initial condition, and after 20, 30 and 40 days.

Note: with the initial conditions used (nonzero concentration in the centre), this is not a particularly clever way of modeling dispersion on a 2-D surface. For this special case it is much more efficient to represent these dynamics in a 1-dimensional model, and using cylindrical coordinates; this model is included as an example model in the help file of `ode.1D`. The 2-D implementation here was added just for illustrative purposes.

3.5. A chemical example: DAE

In chemistry, stiffness frequently arises from the fact that some reactions occur much faster than others. One way to deal with this stiffness is to reformulate the ODE as a DAE (e.g., Hofmann, Meysman, Soetaert, and Middelburg 2008). Consider the following model: Three chemical species A, B, D are kept in a vessel; the following reversible reaction occurs:

$$\mathrm{D} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{A} + \mathrm{B}$$

In addition, D is produced at a constant rate, k_{prod}, while B is consumed at a 1st order rate, r. Implementing this model as an ODE system:

$$\begin{aligned}\frac{d[D]}{dt} &= k_{prod} - k_1 \cdot [D] + k_2 \cdot [A] \cdot [B]\\ \frac{d[A]}{dt} &= -k_2 \cdot [A] \cdot [B] + k_1 \cdot [D]\\ \frac{d[B]}{dt} &= -r \cdot [B] - k_2 \cdot [A] \cdot [B] + k_1 \cdot [D]\end{aligned}$$

The ODEs are now reformulated as a DAE. If the reversible reactions (involving k_1 and k_2) are much faster compared to the other rates (k_{prod}, r) then the three quantities D, A and B can be assumed to be in local equilibrium. Thus, at all times, the following relationship exists between the concentrations of A, B and D (here concentration of x is denoted with $[x]$):

$$K \cdot [D] = [A] \cdot [B]$$

where $K = k_2/k_1$ is the so-called equilibrium constant. The equilibrium description is complete by taking linear combinations of rates of changes, such that the fast reversible reactions vanish.

$$\begin{aligned}\frac{d[D]}{dt} + \frac{d[A]}{dt} &= k_{prod}\\ \frac{d[B]}{dt} - \frac{d[A]}{dt} &= -r \cdot [B]\end{aligned}$$

In this DAE, the fast equilibrium reactions (involving k_1 and k_2) have been removed.

DAEs are specified in implicit form (see Section 1):

$$0 = F(y', y, x, t)$$

In R we define a function that takes as input time (`t`), the values of the state variables (`y`) and their derivatives (`yprime`) and the parameter vector (`pars`), and that returns the results as a `list`; the first element of this list contains the implicit form of the differential equations (`res1, res2`) and of the algebraic equation (`eq`), concatenated. Additionally, other quantities can be returned as well (here `CONC`). Note that `y`, `yprime` and `pars` are vectors with named elements; their names are made available through the `with` statement.

```
R> Res_DAE <- function (t, y, yprime, pars, K) {
+    with (as.list(c(y, yprime, pars)), {
+
+      res1 <- -dD - dA + prod
+      res2 <- -dB + dA - r*B
+
+      eq   <- K*D - A*B
+
+      return(list(c(res1, res2, eq),
+                    CONC = A + B + D))
+    })
+  }
```

After defining the time sequence at which output is wanted (`times`), the parameter values (`pars`) and the initial concentrations of the state variables (`yini`) and their rates of changes (`dyini`), the model is solved with `daspk`. Note that in the example, the initial concentrations of the state variables (`yini`) are consistent (i.e., they obey ($K \cdot [D] = [A] \cdot [B]$), but the initial rates of changes (`dyini`) are not consistent. Thus, `daspk` will solve for them.

```
R> times <- seq(0, 100, by = 2)
R> pars  <- c(r = 1, prod = 0.1)
R> K     <- 1

R> yini  <- c(A = 2, B = 3, D = 2 * 3/K)
R> dyini <- c(dA = 0, dB = 0, dD = 0)

R> DAE <- daspk(y = yini, dy = dyini, times = times, res = Res_DAE,
+              parms = pars, atol = 1e-10, rtol = 1e-10, K = 1)

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

Note how we use the S3 `plot` method to plot all output variables at once on one figure (Figure 4). In the **deSolve** package the same model is also solved as an ODE (example 1 of `ode.1D`). The package also contains a DAE implementation of an electrical system, and all the models from Hofmann *et al.* (2008) in the `doc/examples` directory.

4. Model implementation in a compiled language

This final model is an example of writing the 0-dimensional consumer-prey model equations in a compiled language such as Fortran, C, or C++, and embedding it in R code. Implementations in compiled languages (cf. Tables 1 and 2) consist of:

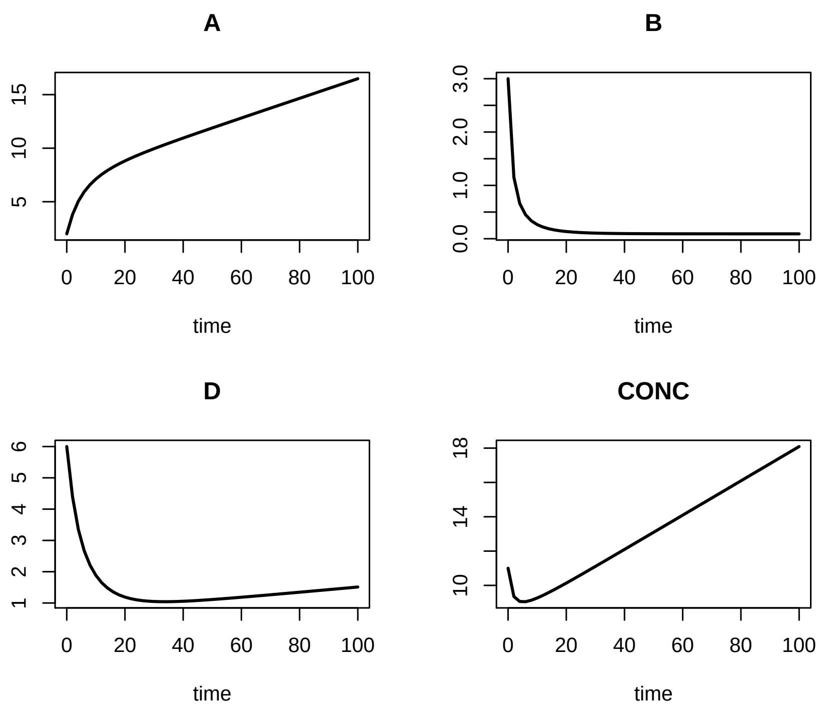

Figure 4: Results of the DAE chemical model. CONC = summed concentration of A, B and D.

- an initializer function, which sets the values of the parameters (`initparms()` in the example),
- if forcing functions are to be used, an initializer for the data for the forcing function (here absent),
- the model function which calculates the rate of change and output variables (`derivs()` in the example), and
- -optionally- a function that calculates an analytic Jacobian (here absent).

Each function has a standard calling sequence (see the package vignette *compiledCode* in the package **deSolve** for the details). The initializer subroutines serve just to link data from the R side of things with memory accessible to the native code, and will rarely be more complicated than the example shown here.

The bulk of the computation is carried out in the subroutine that defines the system of differential equations. In the initializer routine, parameters are passed to the native programs as one vector (containing 5 values). In Fortran, parameters are stored in a common block, in which the values are given a name (`rI`,..) in the model function to make it easier to understand the code, while it is a vector in the initializer routine. In the C code names can be assigned to these parameters as well as state variables and their derivatives via `#define` statements that make the code more readable.

```
c Initialiser for parameter common block
      subroutine initparms(odeparms)
      external odeparms
      double precision parms(5)
      common /myparms/parms
      call odeparms(5, parms)
      return
      end

c Rate of change and output variable
      subroutine derivs (neq, t, y, ydot, yout, IP)
      integer          neq, IP(*)
      double precision t, y(neq), ydot(neq), yout(*)
      double precision rI, rG, rM, AE, K
      common /myparms/rI, rG, rM, AE, K
      if(IP(1)<1) call rexit("nout should be at least 1")
      ydot(1) = rG*y(1)*(1-y(1)/K) - rI*y(1)*y(2)
      ydot(2) = rI*y(1)*y(2)*AE - rM*y(2)
      yout(1) = y(1) + y(2)
      return
      end
```

Table 1: Fortran implementation of the Lotka-Volterra model (`LVmod0D.f`).

The variable `yout(1)` holds the total concentration of consumer and prey at any given time. The model function must check whether enough memory is allocated for the output variable(s), to prevent a fatal error if the memory allocation is inadequate. The subroutines `rexit()` and `error()` are provided by R to gracefully exit with a message back to the R command prompt.

To run this model, the code must first be compiled. Given that the appropriate toolset is installed this can be done in R itself, using the system command: `system("R CMD SHLIB LVmod0D.f")` or `system("R CMD SHLIB LVmod0D.c")`. The compiler will create a file that can be linked dynamically into an R session, for example the dynamic link library (DLL) `LVmod0D.dll` on Windows respectively a shared library `LVmod0D.so` on other operating systems.

The dynamic library is loaded into the current R process using a call to `dyn.load()`.

```
R> dyn.load(paste("LVmod0D", .Platform$dynlib.ext, sep = ""))
```

After providing initial conditions of the state variables, the parameter vector, and the time sequence, the model is run by calling the integrator `ode`. The functions passed to the ODE solver are character strings (`"derivs", "initparms"`), giving the names of the compiled functions in the dynamically loaded shared library (`"LVmod0D"`). When finished, the DLL can be unloaded.

```
#include <R.h>

/* a trick to keep up with the parameters */
static double parms[5];
#define rI parms[0]
#define rG parms[1]
#define rM parms[2]
#define AE parms[3]
#define K  parms[4]

/* initializers */
void initparms(void (* odeparms)(int *, double *)) {
    int N=5;
    odeparms(&N, parms);
}

/* names for states and derivatives */
#define P y[0]
#define C y[1]
#define dP ydot[0]
#define dC ydot[1]

void derivs(int *neq, double *t, double *y,
            double *ydot, double *yout, int *ip){
    if (ip[0] < 1) error("nout should be at least 1");
    dP = rG*P*(1-P/K) - rI*P*C;
    dC = rI*P*C*AE - rM*C;
    yout[0] = P + C;
}
```

Table 2: C implementation of the Lotka-Volterra model (`LVmod0D.c`).

```
R> pars  <- c(rI = 0.2, rG = 1.0, rM = 0.2, AE = 0.5, K = 10)
R> yini  <- c(P = 1, C = 2)
R> times <- seq(0, 200, by = 1)
R> print(system.time(
+    out  <-  ode(func = "derivs", y = yini, parms = pars, times = times,
+      dllname = "LVmod0D", initfunc = "initparms", nout = 1,
+      outnames = c("total"))
+  ))

   user  system elapsed
   0.02    0.00    0.01

R> dyn.unload(paste("LVmod0D", .Platform$dynlib.ext, sep = ""))
```

5. Benchmarking

Model implementations written in compiled languages are expected to have one major advantage in comparison with implementations in pure R: they use less CPU time. In this section the performance of the two types of model implementation is illustrated by means of a set of test problems. In order to increase the computational demand in a systematic way, we use the consumer-prey model in different settings:

- The zero-dimensional case from Section 3.1.
- Several one-dimensional cases (Section 3.3), with varying number of grid cells (50, 100, 500, 1000, 2500, 5000). The latter is a 10000 state variable model.
- Two two-dimensional settings, on a $50 \cdot 50$ and $100 \cdot 100$ grid (Section 3.4).

All models were run for 200 days with a daily output interval, that is also the maximum time step. We tested two implementations in R and two Fortran codes:

- The R codes presented in Section 3, which include passing the names of state variables and parameters.
- A second implementation in R, where the names of parameters and state variables are not used (i.e., without the `with()`-function).
- A Fortran implementation, where the model was compiled as a DLL, loaded into R, and the integration routine was triggered from within R, same as in previous section.
- A second Fortran implementation, where the entire run was performed in Fortran.

The Fortran codes will not be given here but they are included in the R package (in the `doc/examples/dynload` subdirectory).

CPU (secs)	R code (1)	R code (2)	Fortran in R	All Fortran
0D	0.04	0.014	0.0006	
1D (50boxes)	0.16	0.13	0.008	0.008
1D (100boxes)	0.18	0.14	0.012	0.015
1D (500boxes)	0.38	0.33	0.096	0.1
1D (1000boxes)	0.58	0.54	0.21	0.21
1D (2500boxes)	1.4	1.35	0.64	0.58
1D (5000boxes)	3.0	2.9	1.6	1.35
2D (50·50boxes)	2.3	2.2	0.47	
2D (100·100)	16.5	16.4	4.1	

Table 3: CPU time (in seconds) needed to perform a run of the Lotka-Volterra model in 0-D, 1D and 2-D (rows) and for different implementations (columns): (1) R code as in Section 3; (2) R code without passing parameter and variable names; (3) model specified in a Fortran DLL, loaded into R and the integration triggered by R and (4) the entire application implemented in Fortran. All times reported are the mean of 10 consecutive runs.

The measure used to evaluate computational performance is the CPU time spent in these runs (Table 3). To obtain representative run times, we compare the average over 10 consecutive runs, and on the same machine. Times reported are seconds of computing time, on a 2.5 GHz portable pc with an Intel Core 2 Duo T9300 processor and 3 GB of RAM. Both `ode.1D` and `ode.2D` used integration routine `lsodes` for solving the model.

Except for the 0-dimensional model, there is little gain (10-25%) in computing time by not passing the parameter and state-variable names. In the simplest (0-D) model, the R version that does not pass names (R code 2) finishes in only 35% of CPU time compared to the full R implementation (R code 1).

The gain is much more pronounced when the model is implemented in Fortran rather than in R: here the Fortran implementation executes 2 to 20 (1-D, 2-D) to 66 times (0-D) faster (than R code 1). Finally, the difference between a Fortran model triggered by R, or a model completely implemented in Fortran is very small; part of the difference is due to the checking for illegal input values in the R integration routines.

6. Concluding remarks

The software R is rapidly gaining in popularity among scientists. With the launch of package **odesolve** (Setzer 2001), it became possible to use R as a tool to solve initial value problems of ordinary differential equations. The integration routines in this package opened up an entirely new field of application, although it took a while before this was acknowledged. More recent packages (**rootSolve, bvpSolve**) (Soetaert 2009; Soetaert, Cash, and Mazzia 2010) offer to solve boundary value problems of differential equations.

The paper in R News by Petzoldt (2003), demonstrated the suitability of R for running dynamic (ecological) simulations. More recently, a specially designed framework for ecological modelling in R, **simecol**, emerged (Petzoldt and Rinke 2007); packages for inverse modelling, (**FME**) and reactive transport modelling (**ReacTran**) (Soetaert and Petzoldt 2010; Soetaert and Meysman 2009) were created, while a framework for more general continuous dynamic modeling, **Rdynamic** (Setzer in prep.) is under construction. An increasing number of textbooks deal with the subject (Ellner and Guckenheimer 2006; Bolker 2008; Soetaert and Herman 2009; Stevens 2009).

In order to efficiently solve a variety of differential equation models, a flexible set of integration routines is required. It is with this goal in mind that the integration routines in **deSolve** were selected. Whereas the original integration routine in **odesolve** only efficiently solved relatively simple ODE systems, the suite of routines now also includes a solver for differential algebraic equations and methods to solve partial differential equations.

In this paper we have shown that thanks to these new functions, R can now more efficiently run 0-dimensional, 1- 2-, and even 3-dimensional models of small to moderate size. Apart from models implemented in pure R, it is possible to specify model functions in compiled code written in any higher-level language that can produce shared libraries (resp. DLLs). The integration routines then communicate directly with this code, without passing arguments to and from R, so R is used just to trigger the integration and post-process the results. As the entire simulation occurs in compiled code, there is *no loss in execution speed* compared to a model that is fully implemented in the higher level language. But even in this case, all the power of R as a pre- and post processing environment as well as its graphical and statistical facilities are immediately available – no need to import the model output from an external source.

In the examples, linking compiled code to the integrator, indeed made the model run faster with a factor 2 (for the 10000 state variable 1-D model) up to more than 50 times for the smallest (2 state variable) model application, than when implemented as an R function. There was only a small difference when running the model entirely in compiled code.

There are several reasons why compiled code is faster. First of all, R is an interpreted language, and therefore processes the program at runtime. Every line is interpreted multiple times at each time step. This makes interpreted code significantly slower than compiled code, which transform programs directly into machine code, before running. Note though that R is a vectorized language and, compared to some other interpreted languages, less performance is lost if R's high level functions, based on optimized machine code, are efficiently exploited (Ligges and Fox 2008). In our 1-D example, we used R function `diff` to take numerical differences, whilst in the 2-D model, entire matrices were subtracted. Because of this use of high-level functions, the simulation speed of these models, entirely specified in R, was quite impressive, approaching the implementation in Fortran. Performance of R code especially deteriorates when using loops. For instance, if the 2-D model is implemented by looping over all rows, then the simulation time increases tenfold; when looping over rows and columns, computation speed drops with 2 orders of magnitude! There are also trade-offs in using complex variable types of R, especially if R performs extensive copying or internal data conversion. For instance, the use of named variables and parameters introduced a computational overhead of around 70% in our simplest model example. However, the effect was relatively less significant, in the order of 10-20%, in more demanding models.

The use of code in a dynamically linked library also has its drawbacks. First of all, it is less flexible. Whereas it is simple to interact with models specified in R code, this is not at all the case for compiled code: before the model code can be executed, it has to be formally compiled, and the DLL loaded. Secondly, errors may be particularly hard to trace and may even cause R to terminate. The lack of easy access to R's high-level procedures is another drawback of using compiled code, where much more has to be hand-coded. Note though that, as from **deSolve** version 1.5, the interpolation of external signals (also called forcing functions) to the current timepoints is taken care of by the integration routines; the compiled-code equivalent of R function `approxfun`.

Putting these pros and cons together, the optimal approach is probably to use pure R for the initial model development (rapid prototyping). In case the model executes too slowly, or when a large number of simulations are performed, implementing the model in C, C++ or Fortran may be considered.

Finally, the creation and solution of a mathematical model is never a goal in itself. Models are used, amongst other things to challenge our understanding of a natural system, to make budgets or to quantify immeasurable processes or rates. When used in this way, the interaction with data is crucial, as is statistical treatment and graphical representation of the model outcome and the data. We hope that R's excellence in these fields, and the fact that it is entirely free, will give impetus to also using R as a modelling platform.

Acknowledgments

The authors would like to thank our many colleagues and other R enthousiasts who have tested the package. Two anonymous reviewers gave constructive comments on the paper. Also thanks to Jan de Leeuw and Achim Zeileis for bringing this to a happy conclusion. The United States Environmental Protection Agency through its Office of Research and Development collaborated in the research described here. It has been subjected to Agency review and approved for publication.

References

Asher UM, Petzold LR (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia.

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

Bolker B (2008). *Ecological Models and Data in R*. Princeton University Press, Princeton. URL `http://www.zoology.ufl.edu/bolker/emdbook/`.

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

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

Brown PN, Hindmarsh AC, Petzold LR (1994). "Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems." *SIAM Journal on Scientific and Statistical Computing*, **15**(6), 1467–1488. `doi:10.1137/0915088`.

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

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

Copeland T, Mas R, McCullagh K, Perdue T, Smet G, Spisser R (2006). *GForge Manual*. URL `http://GForge.org/docman/view.php/1/34/gforge_manual.pdf`.

Crank J (1975). *The Mathematics of Diffusion*. 2nd edition. Clarendon Press, Oxford.

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

Eisenstat SC, Gursky MC, Schultz MH, Sherman AH (1982). "Yale Sparse Matrix Package. i. The Symmetric Codes." *International Journal for Numerical Methods in Engineering*, **18**, 1145–1151.

Ellner SP, Guckenheimer J (2006). *Dynamic Models in Biology*. Princeton University Press, Princeton. URL http://www.cam.cornell.edu/~dmb/DMBsupplements.html.

Hairer E, Wanner G (1980). *Solving Ordinary Differential Equation: Stiff Systems Vol. 2*. Springer-Verlag, Heidelberg.

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

Hofmann AF, Meysman FJR, Soetaert K, Middelburg JJ (2008). "A Step-by-Step Procedure for pH Model Construction in Aquatic Systems." *Biogeosciences*, **5**(1), 227–251. URL http://www.biogeosciences.net/5/227/2008/.

Ligges U, Fox J (2008). "R Help Desk: How Can I Avoid This Loop or Make It Faster?" *R News*, **8**(1), 46–50. URL http://CRAN.R-project.org/doc/Rnews/.

Lotka AJ (1925). *Elements of Physical Biology*. Williams & Wilkins Co., Baltimore.

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

Petzoldt T (2003). "R as a Simulation Platform in Ecological Modelling." *R News*, **3**(3), 8–16. URL http://CRAN.R-project.org/doc/Rnews/.

Petzoldt T, Rinke K (2007). "**simecol**: An Object-Oriented Framework for Ecological Modeling in R." *Journal of Statistical Software*, **22**(9), 1–31. URL http://www.jstatsoft.org/v22/i09/.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007). *Numerical Recipes*. 3rd edition. Cambridge University Press.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Schiesser WE (1991). *The Numerical Method of Lines: Integration of Partial Differential Equations*. Academic Press, San Diego.

Setzer RW (2001). *The **odesolve** Package: Solvers for Ordinary Differential Equations*. R package version 0.1-1, URL http://CRAN.R-project.org/package=odeSolve.

Setzer RW (in prep.). ***RDynamic**, an R Package for Dynamic Modelling*. R package version 0.1-1, URL http://r-forge.r-project.org/projects/rdynamic/.

Soetaert K (2009). ***rootSolve**: Nonlinear Root Finding, Equilibrium and Steady-state Analysis of Ordinary Differential Equations*. R package version 1.6, URL http://CRAN.R-project.org/package=rootSolve.

Soetaert K, Cash JR, Mazzia F (2010). ***bvpSolve**: Solvers for Boundary Value Problems of Ordinary Differential Equations*. R package version 1.1, URL http://CRAN.R-project.org/package=bvpSolve.

Soetaert K, Herman PMJ (2009). *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform*. Springer-Verlag, New York.

Soetaert K, Meysman F (2009). ***ReacTran**: Reactive Transport Modelling in 1D, 2D and 3D*. R package version 1.1, URL http://CRAN.R-project.org/package=ReacTran.

Soetaert K, Petzoldt T (2010). "Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package **FME**." *Journal of Statistical Software*, **33**(3), 1–28. URL http://www.jstatsoft.org/v33/i03/.

Soetaert K, Petzoldt T, Setzer RW (2009). ***deSolve**: General Solvers for Initial Value Problems of Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), Differential Algebraic Equations (DAE), and Delay Differential Equations (DDE)*. R package version 1.7, URL http://CRAN.R-project.org/package=deSolve.

Stevens MHH (2009). *A Primer of Ecology with R*. Springer-Verlag, Berlin.

Theußl S, Zeileis A (2009). "Collaborative Software Development Using R-Forge." *The R Journal*, **1**(1), 9–14. URL http://journal.R-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf.

Volterra V (1926). "Fluctuations in the Abundance of a Species Considered Mathematically." *Nature*, **118**, 558–560.

A. Overview of the solver functions

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

Table 4: The differential equation solvers provided by package **deSolve**.

Affiliation:

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

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

R. Woodrow Setzer
National Center for Computational Toxicology
US Environmental Protection Agency
United States of America
URL: http://www.epa.gov/ncct/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 33, Issue 9 *Submitted:* 2008-07-17
February 2010 *Accepted:* 2010-02-12