simecol-Howto: Tips, Tricks and Building Blocks

Thomas Petzoldt
Technische Universität Dresden

Abstract

This document is intended as a lose collection of sections that describe different aspects of modelling and model implementation in R unsing the **simecol** package.

The howto-collection supplements the original publication of the package (Petzoldt and Rinke 2007) from which an updated version is also part of this package. Please refer to the original publication when citing this work.

Keywords: R, **simecol**, ecological modeling, object-oriented programming (OOP), compiled code, debugging.

1. Introduction

2. Building simecol objects

The intention behind **simecol** is the construction of "'all-in-one"' model objects. That is, everything that defines one particular model, equations and data are stored together in one `simObj` (spoken: sim-Object), and only some general algorithms (e.g. diferential equation solvers or interpolation routines) remain external, preferably as package functions (e.g. function `lsoda` in package **deSolve** or as functions in the user workspace.

This strategy has three main advantages:

1. You can have several independent versions of one model in the computer memory at the same time. These instances may have different settings, parameters and data or even use different formula, but they don't interfere with each other. Moreover, if all data and functions are *encapsulated* in their `simObj`ects, identifiers can be re-used and it is, for example, not neccessary to keep track over a large number of variable names or to invent new identifiers for parameter sets of different scenarios.

2. You can give `simObj`ects away, either in binary form or as source code objects. Everything essential to run such a model is included, not only the formula but also defaults for parameter and data. You, or your users need only R, some packages and your model object. It is also possible to start model objects directly from the internet or, on the other side, to distribute model collections as R packages.

3. All `simObj`ects can be handled, simulated and modified with the same generic functions, e.g. `sim`, `plot` or `parms`. Your users can start playing with your models without the need to understand all the internals.

While it is, of course, preferable to have all parts of a model encapsulated in one object, it is not mandatory to have the complete working model object before using simecol.

simecol models (`simObjects`) can be built up step by step, starting with mixed applications composed by rudimentary simObjects and ordinary user space functions and then encapsulating all the parts until the final all-in-one object is ready.

2.1. An Example

We start with the example, given in the simecol-introduction (Petzoldt and Rinke 2007), an implementation of the UPCA model of Blasius *et al.* (1999), but we write it in the usual **deSolve** style, e.g. without using simecol:

```
R> f <- function(x, y, k) {
+      x * y/(1 + k * x)
+  }
R> func <- function(time, y, parms) {
+      with(as.list(c(parms, y)), {
+          du <- a * u - alpha1 * f(u, v, k1)
+          dv <- -b * v + alpha1 * f(u, v, k1) + -alpha2 *
+              f(v, w, k2)
+          dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
+          list(c(du, dv, dw))
+      })
+  }
R> times <- seq(0, 100, 0.1)
R> parms <- c(a = 1, b = 1, c = 10, alpha1 = 0.2, alpha2 = 1,
+      k1 = 0.05, k2 = 0, wstar = 0.006)
R> y <- c(u = 10, v = 5, w = 0.1)
```

The model is defined by 5 variables in the R user space, namely f, func, times, parms and init. The implementation is similar to the examples of package **deSolve** and we can solve it exactly in the same manner:

```
R> library(deSolve)
R> out <- lsoda(y, times, func, parms)
R> matplot(out[, 1], out[, -1], type = "l")
```

2.2. Transistion to simecol

If we compare this example with the simecol structure, we may see that they are kind of similar. This obvious coincidence is quite natural, because the notation of both, **deSolve** and **simecol**, is based on the state-space notation of control theory[1].

Due to this, it needs only small restructuring and renaming to form a `simObj`:

[1] see http://en.wikipedia.org/wiki/State_space_(controls), version of 2008-11-01

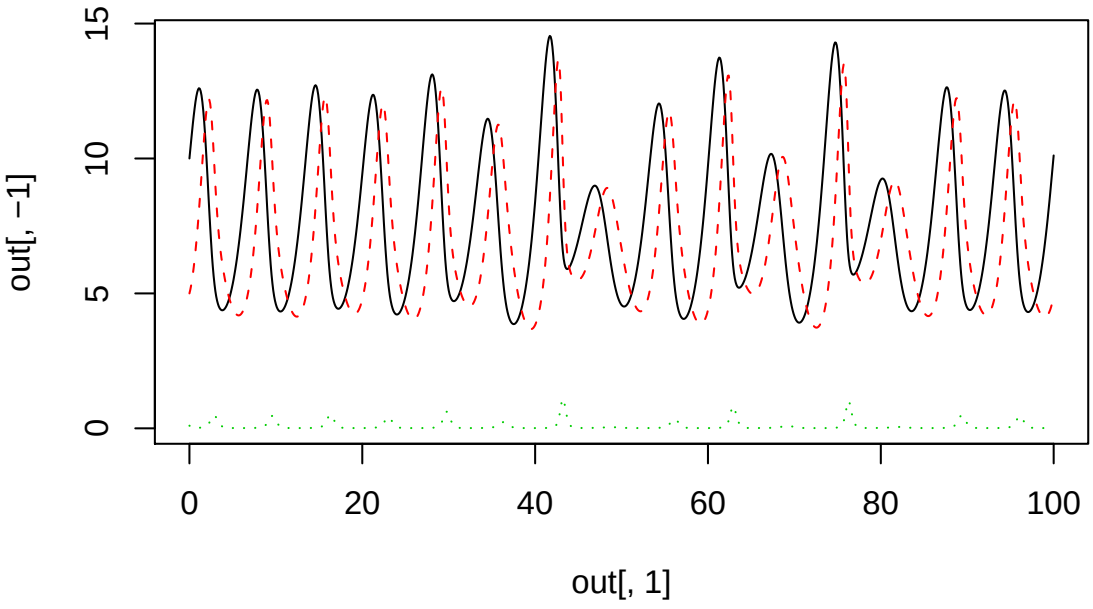

Figure 1: Output of UPCA model, solved mit `lsoda` from package **deSolve**.

```
R> library("simecol")
R> f <- function(x, y, k){x*y / (1+k*x)}  # Holling II
R> upca <- new("odeModel",
+    main = function(time, y, parms) {
+      with(as.list(c(parms, y)), {
+        du <-  a * u           - alpha1 * f(u, v, k1)
+        dv <- -b * v           + alpha1 * f(u, v, k1) +
+                               - alpha2 * f(v, w, k2)
+        dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
+        list(c(du, dv, dw))
+      })
+    },
+    times  = seq(0, 100, 0.1),
+    parms  = c(a=1, b=1, c=10, alpha1=0.2, alpha2=1, k1=0.05, k2=0, wstar=0.006),
+    init   = c(u=10, v=5, w=0.1),
+    solver = "lsoda"
+  )
```

You may notice, that the assignment operators "`<-`" changed to declarative "`=`" for the slot definitions, that some of the names (y, func) were changed to the pre-defined slot names of simecol and that all the slot definitions are now comma separated arguments of the `new` function that creates the `upca` object. The solver method `lsoda` is also given as a character

string pointing to the original `lsoda` function in package **deSolve**.

The new object can now be simulated with the `sim` function of **simecol**, that returns the object with all original slots and an additional slot `out` holding the output values. A generic `plot` function is available for basic plotting of the outputs:

```
R> upca <- sim(upca)
R> plot(upca)
```

and it is also possible to extract the results with the accessor function `out`, and to use an arbitrary, user-defined plot function:

```
R> plotupca <- function(obj, ...) {
+      o <- out(obj)
+      matplot(o[, 1], o[, -1], type = "l", ...)
+      legend("topleft", legend = c("u", "v", "w"), lty = 1:3,
+          , bg = "white", col = 1:3)
+  }
R> plotupca(upca)
```

O.K., that's it, but note that function `f` is not yet part of the simecol object, what we call here a "mixed implementation". This function `f` is rather simple here, and it would be also possible to call functions of arbitrary complexity from `main`.

2.3. Creating scenarios

After defining one simecol object (that we can call a parent object or a prototype), we can now create derived objects, simply by copying (cloning) and modification. As an example, we create two scenarios with different parameter sets:

```
R> sc1 <- sc2 <- upca
R> parms(sc1)["wstar"] <- 0
R> parms(sc2)["wstar"] <- 0.1
R> sc1 <- sim(sc1)
R> sc2 <- sim(sc2)
R> par(mfrow = c(1, 2))
R> plotupca(sc1, ylim = c(0, 250))
R> plotupca(sc2, ylim = c(0, 250))
```

If we simulate and plot these scenarios, we see an exponentially growing u in both cases, and cycles resp. an equilibrium state for v and w for the scenarios respectively (figure 2).

If we change now the functional response function f from Holling II to Lotka-Volterra:

```
R> f <- function(x, y, k) {
+      x * y
+  }
```

both model scenarios, sc1 and sc2 are affected by this new definition:

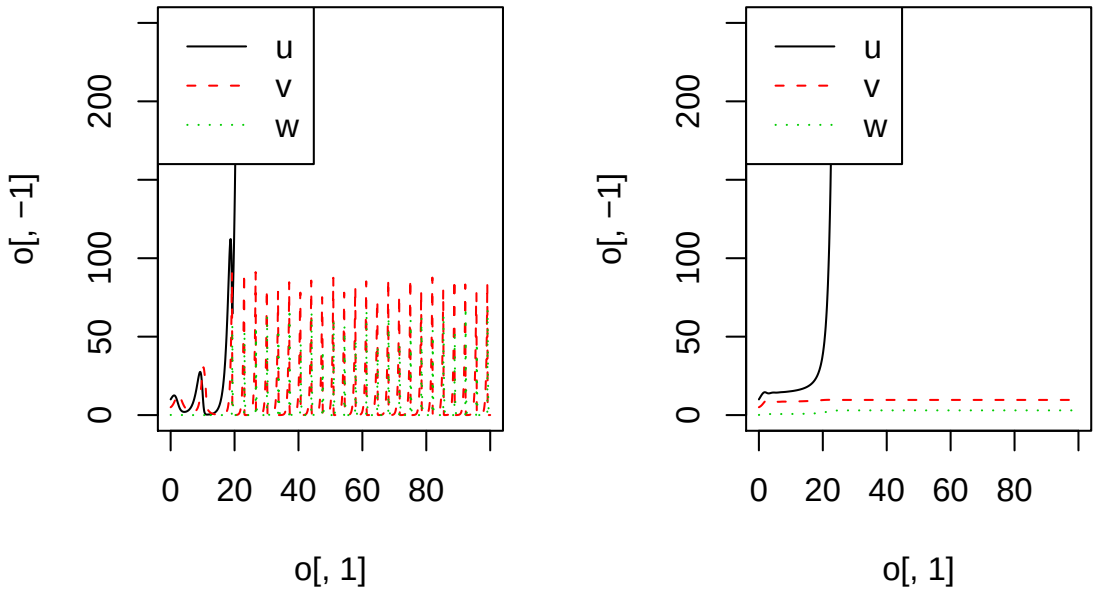

Figure 2: Two scenarios of the UPCA model (left: wstar=0, right: wstar=0.1; functional response f is Holling II).

```
R> sc1 <- sim(sc1)
R> sc2 <- sim(sc2)
R> par(mfrow = c(1, 2))
R> plotupca(sc1, ylim = c(0, 15))
R> plotupca(sc2, ylim = c(0, 15))
```

with a stable limit cycle for u and v in scenario 1 and an equilibrium for all state variables in scenario 2 (figure 3). You may also note that the new function `f` has exactly the same parameters as above, including the, in the second case obsolete, parameter `k`.

In the examples above, function `f` is an ordinary function in the user workspace, but it is also possible to implement such functions (or sub-models) directly as part of the model object. As one possibility, one might consider to define local functions within `main`, but that would have the disadvantage that such functions are not easily accessible from outside.

To enable this, simecol has an optional slot "equations", that can hold a list of submodels. This equations-slot can be defined either during object creation, or functions may be added afterwards. In the following, we derive to new clones with the default parameter settings from the original upca-object, and then add the Holling II functional resonse to scenario 1 and the Lotka-Volterra functional response to scenario 2 (figure 4):

```
R> sc1 <- sc2 <- upca
R> equations(sc1)$f <- function(x, y, k) {
```

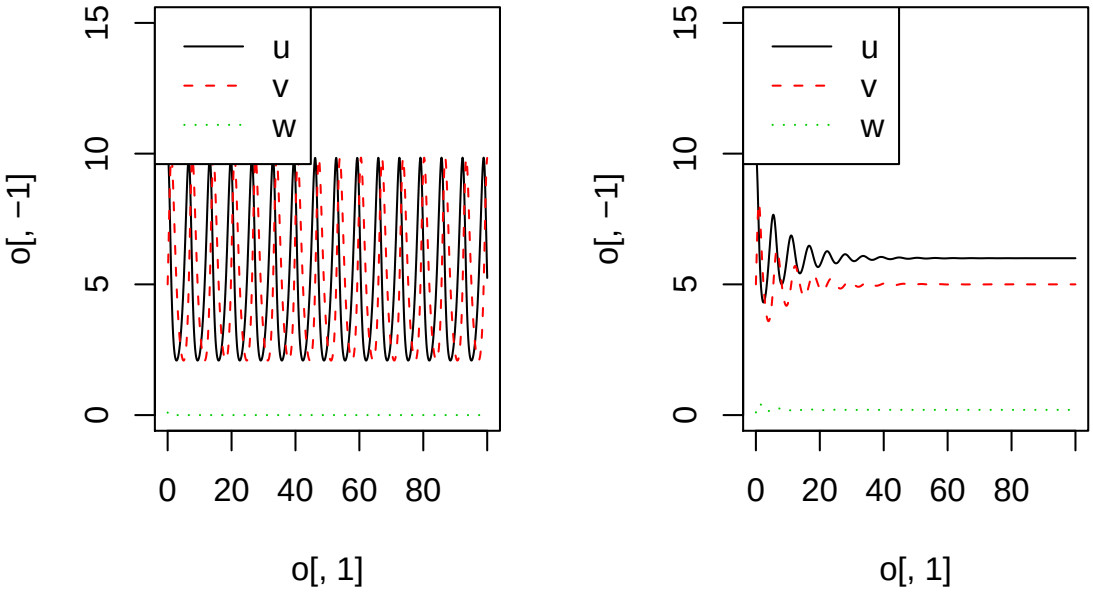

Figure 3: Two scenarios of the UPCA model (left: wstar=0, right: wstar=0.1; functional response f is Holling II).

```
+       x * y/(1 + k * x)
+   }
R> equations(sc2)$f <- function(x, y, k) {
+       x * y
+   }
R> sc1 <- sim(sc1)
R> sc2 <- sim(sc2)
R> par(mfrow = c(1, 2))
R> plotupca(sc1, ylim = c(0, 15))
R> plotupca(sc2, ylim = c(0, 15))
```

This method allows to compare models with different structure and it is also possible to define model objects with different versions of submodels built-in, that can be alternatively enabled:

```
R> upca <- new("odeModel", main = function(time, y, parms) {
+       with(as.list(c(parms, y)), {
+           du <- a * u - alpha1 * f(u, v, k1)
+           dv <- -b * v + alpha1 * f(u, v, k1) + -alpha2 *
+               f(v, w, k2)
+           dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
+           list(c(du, dv, dw))
```

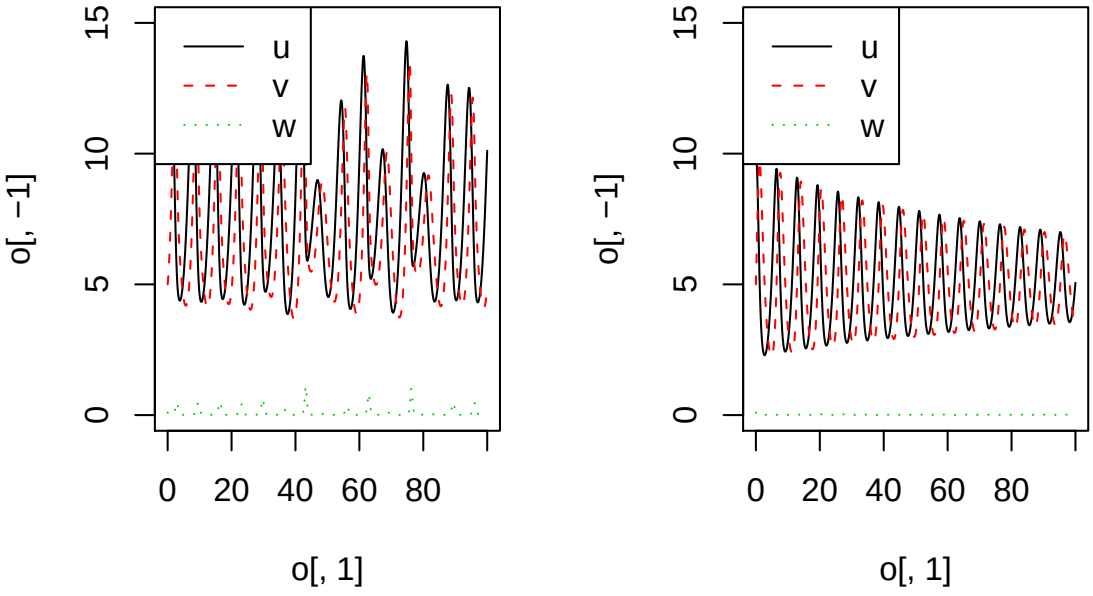

Figure 4: Two scenarios of the UPCA model (left: functional response f is Holling II, right f is Lotka-Volterra).

```
+      })
+  }, equations = list(f1 = function(x, y, k) {
+      x * y
+  }, f2 = function(x, y, k) {
+      x * y/(1 + k * x)
+  }), times = seq(0, 100, 0.1), parms = c(a = 1, b = 1, c = 10,
+      alpha1 = 0.2, alpha2 = 1, k1 = 0.05, k2 = 0, wstar = 0.006),
+      init = c(u = 10, v = 5, w = 0.1), solver = "lsoda")
R> equations(upca)$f <- equations(upca)$f1
```

2.4. Debugging

As stated before, joint encapsulation of all functions and data in simObjects has many advantages, but there is also one disadvantage, namely debugging. Debugging of S4 objects may be cumbersome, especially when slot-functions (e.g. main, equations, initfunc) come into play. These difficulties are not much important for well-functioning ready-made model objects, but they may appear as an additional burden during the development of simecol objects, in particular if these models are technically not that simple as our example.

Fortunately, there is an easy workaround, that is, implementing the technically challenging parts in the user-workspace first using the above mentioned mixed style and after everything

works satisfactorily, integrating these parts to the object. In the example below, we implement the main model as workspace function `fmain`[2] with the same interface (parameters and return values) as above, that is then called by the `main`-function of the simObj:

```
R> f <- function(x, y, k) {
+      x * y/(1 + k * x)
+  }
R> fmain <- function(time, y, parms) {
+      with(as.list(c(parms, y)), {
+          du <- a * u - alpha1 * f(u, v, k1)
+          dv <- -b * v + alpha1 * f(u, v, k1) + -alpha2 *
+              f(v, w, k2)
+          dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
+          list(c(du, dv, dw))
+      })
+  }
R> upca <- new("odeModel", main = function(time, y, parms) fmain(time,
+      y, parms), times = seq(0, 100, 0.1), parms = c(a = 1,
+      b = 1, c = 10, alpha1 = 0.2, alpha2 = 1, k1 = 0.05,
+      k2 = 0, wstar = 0.006), init = c(u = 10, v = 5, w = 0.1),
+      solver = "lsoda")
```

This function `fmain` as well as any other submodels like `f` can now be debugged with the usual R tools, e.g. `debug`:

```
R> debug(fmain)
R> upca <- sim(upca)
```

and debugging can be stopped by `undebug(fmain)`. If everything works, you can add the body of `fmain` to `upca` manually, and it is even possible to do this in the formalized `simecol` way of object modification:

```
R> main(upca)        <- fmain   # assign workspace function to main slot
R> equations(upca)$f <- f       # assign workspace function to equations
R> rm(fmain, f)   # optional, for saving memory and avoiding confusion
R> str(upca)      # show the object

Formal class 'odeModel' [package "simecol"] with 10 slots
  ..@ parms    : Named num [1:8] 1e+00 1e+00 1e+01 2e-01 1e+00 5e-02 0e+00 6e-03
  .. ..- attr(*, "names")= chr [1:8] "a" "b" "c" "alpha1" ...
  ..@ init     : Named num [1:3] 10 5 0.1
  .. ..- attr(*, "names")= chr [1:3] "u" "v" "w"
  ..@ observer : NULL
  ..@ main     :function (time, y, parms)
```

[2]Note that this function must never be named "func", for some rather esotheric internal reasons which we shall not discuss further here.

```
..@ equations:List of 1
.. ..$ f:function (x, y, k)
..@ times    : num [1:1001] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ...
..@ inputs   : NULL
..@ solver   : chr "lsoda"
..@ out      : NULL
..@ initfunc : NULL
```

3. Different ways to store simObjects

One of the main advantages of **simecol** is, that model objects can be made persistent and that it is easy to distribute and share simObjects over the internet.

The most obvious and simple form is, of course, to use the original source code of the objects, i.e. the function call to the `new` with all the slots which creates the S4-object (see section 2.2), but there are also other possibilities.

simecol objects can be saved in binary form as S4 object binaries with the `save` method of R, which stores the whole object with all its equations, initial values, parameters etc. and also the simulation outputs if the model was simulated before saving.

```
R> save(upca, file="upca.Rdata")  # persistent storage of the model object
R> load("upca.Rdata")             # load the model
```

Another possibility is conversion of the S4 object to a list representation:

```
R> l.upca <- as.list(upca)
```

This method allows to get an alternative text representation of the `simObj`, that can be manipulated by code parsing programs or dumped to the hard disk:

```
R> dput(l.upca, file = "upca_list.R")
```

and this is completely reversible via:

```
R> l.upca <- dget("upca_list.R")
R> upca <- as.simObj(l.upca)
```

Sometimes it may be useful to store simObjects in an un-initialized form, in particular if they are to be distributed in packages.

Let's demonstrate this again with a simple Lotka-Volterra model. In the first step, we define a function, that returns a `simecol` object:

```
R> genLV <- function() {
+      new("odeModel", main = function(time, init, parms) {
+          x <- init
```

```
+          p <- parms
+          dx1 <- p["k1"] * x[1] - p["k2"] * x[1] * x[2]
+          dx2 <- -p["k3"] * x[2] + p["k2"] * x[1] * x[2]
+          list(c(dx1, dx2))
+      }, parms = c(k1 = 0.2, k2 = 0.2, k3 = 0.2), times = c(from = 0,
+          to = 100, by = 0.5), init = c(prey = 0.5, predator = 1),
+          solver = "lsoda")
+  }
```

Now, the function contains the instruction, how R can create a new instance of such a model. The `simecol` object is not created yet, but a call to the creator function can bring it into live:

```
R> lv1 <- genLV()
R> plot(sim(lv1))
```

This style is used in package **simecolModels**[3], a collection of (mostly) published ecological models.

4. Implementing models in compiled languages

Compilation of model code can speed up simulations considerably and there are two ways to gain from compiled code. It is possible to to use functions written in C/C++ or Fortran in the ordinary way described in the "Writing R Extensions" manual (R Development Core Team 2006). This can speed up computations but still needs communication overhead because the control is given back to R in every simulation step.

In addition to this it is also possible to enable direct communication between integration routines and the model code if both are available in compiled code and if direct call of a compiled model is supported by the integrator. All integrators of the `lsoda`-family of solvers support this and additional solvers may support this in the future, see the **deSolve** documentation for details.

Now, let's inspect an example. We firstly provide our model as described in the `deSolve` documentation, here again the Lotka-Volterra-model:

```
/* file: clotka.c */
#include <R.h>

static double parms[3];

#define k1 parms[0]
#define k2 parms[1]
#define k3 parms[2]

/* It is possible to define global variables here */
static double aGlobalVar = 99.99;   // for testing only
```

[3] **simecolModels** can be downloaded from the R-Forge server, http://simecol.r-forge.r-project.org/.

```
/* initializer: same name as the dll (without extension) */
void clotka(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
    Rprintf("model parameters succesfully initialized\n");
}

/* Derivatives */
void dlotka(int *neq, double *t, double *y, double *ydot, double *yout, int *ip) {
    // sanity checks
    if (ip[0] < 2) error("nout should be at least 2");
    // derivatives
    ydot[0] = k1 * y[0]        - k2 * y[0] * y[1];
    ydot[1] = k2 * y[0] * y[1] - k3 * y[1];

    // additional outputs, here for demo purposes only
    yout[0] = aGlobalVar;
    yout[1] = ydot[0];
}
```

The 'define' macros are a typical C-trick to get readable names for the parameters. This method is simple and efficient, but there are, of course, other possibilities, for example using dynamic variables.

..... continue here with compilation

Compilation requires an installed C compiler, e.g. an appropriate version of `gcc` on Linux or Macintosh, resp. the R-Tools collection http://www.murdoch-sutherland.com/Rtools/ provided by Duncan Murdoch for Windows.

```
R> system("R CMD SHLIB clotka.c")
```

..... and the R part:

```
R> modeldll <- dyn.load("clotka.dll")
R> clotka <- new("odeModel", main = function(time, init, parms) {
+      list(lib = "clotka", func = "dlotka", jacfunc = NULL,
+          nout = 2)
+  }, parms = c(k1 = 0.2, k2 = 0.2, k3 = 0.2), times = c(from = 0,
+      to = 100, by = 0.5), init = c(prey = 0.5, predator = 1),
+      solver = function(init, times, funclist, parms, ...) {
+          f <- funclist()
+          as.data.frame(lsoda(init, times, func = f$func,
+              parms = parms, dllname = f$lib, jacfunc = f$jacfunc,
+              nout = f$nout, ...))
+      })
R> clotka <- sim(clotka)
```

```
R> plot(clotka)
R> times(clotka)["to"] <- 1000
R> plot(sim(clotka))
R> plot(sim(clotka, atol = 1))
R> dyn.unload(as.character(modeldll[2]))
```

... describe the list structure and the solver

References

Blasius B, Huppert A, Stone L (1999). "Complex Dynamics and Phase Synchronization in Spatially Extended Ecological Systems." *Nature*, **399**, 354–359.

Petzoldt T, Rinke K (2007). "**simecol**: An Object-Oriented Framework for Ecological Modeling in R." *Journal of Statistical Software*, **22**(9), 1–31. ISSN 1548-7660. URL http://www.jstatsoft.org/v22/i09.

R Development Core Team (2006). *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9, URL http://www.R-project.org.

Affiliation:

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/