simecol-Howto: Tips, Tricks and Building Blocks

Thomas Petzoldt
Technische Universität Dresden

Abstract

This document is intended as a loose collection of sections that describe different aspects of modelling and model implementation in R with the **simecol** package. It supplements the original publication of Petzoldt and Rinke (2007) from which an updated version, **simecol-introduction**, is also part of this package. Please refer to the JSS publication when citing this work.

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

1. 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. differential equation solvers or interpolation routines) remain external, preferably as package functions (e.g. function `lsoda` in package **deSolve** (Soetaert, Petzoldt, and Setzer 2009) 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 necessary 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 starting to use **simecol**.

simecol models (in the following called `simObjects`) can be built up step by step, starting with mixed applications composed by rudimentary `simObjects` and ordinary user space functions. When everything works, you should encapsulate all the main parts of your model in the `simObject` to get a clean object that does not interfere with others.

1.1. An Example

We start with the example, given in the simecol-introduction (Petzoldt and Rinke 2007), an implementation of the UPCA model of Blasius, Huppert, and Stone (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 help page 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")
```

1.2. Transition 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`:

```
R> library("simecol")
R> f <- function(x, y, k){x*y / (1+k*x)}  # Holling II
```

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

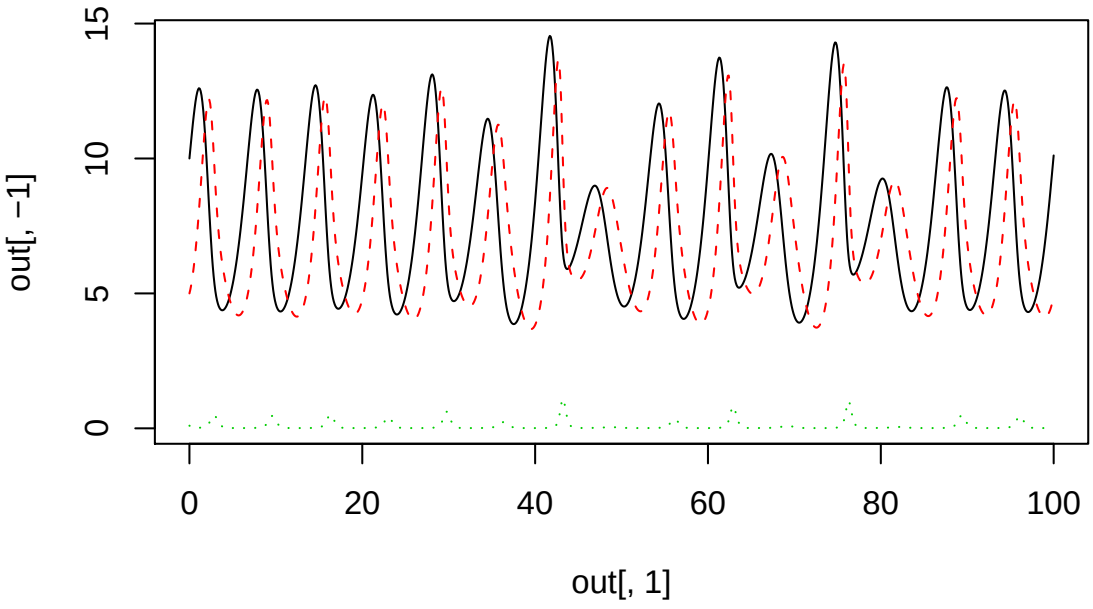

Figure 1: Output of UPCA model, solved with `lsoda` from package **deSolve**.

```
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 a declarative equal sign "`=`" 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 very easily with the `sim` function of **simecol**, that returns the object with all original slots and one additional slot `out` holding the output values. A generic `plot` function is also available for basic plotting of the outputs:

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

It is now also possible to extract the results from `upca` with a so called accessor function `out`, and to use arbitrary, user-defined plot functions:

```
R> plotupca <- function(obj, ...) {
+      o <- out(obj)
+      matplot(o[, 1], o[, -1], type = "l", ...)
+      legend("topright", 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, that's why we call here a "mixed implementation". This function `f` is rather simple here, but it would be also possible to call functions of arbitrary complexity from `main`.

1.3. Creating scenarios

After defining one `simecol` object (that we can call a parent object or a **prototype**), we may 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:

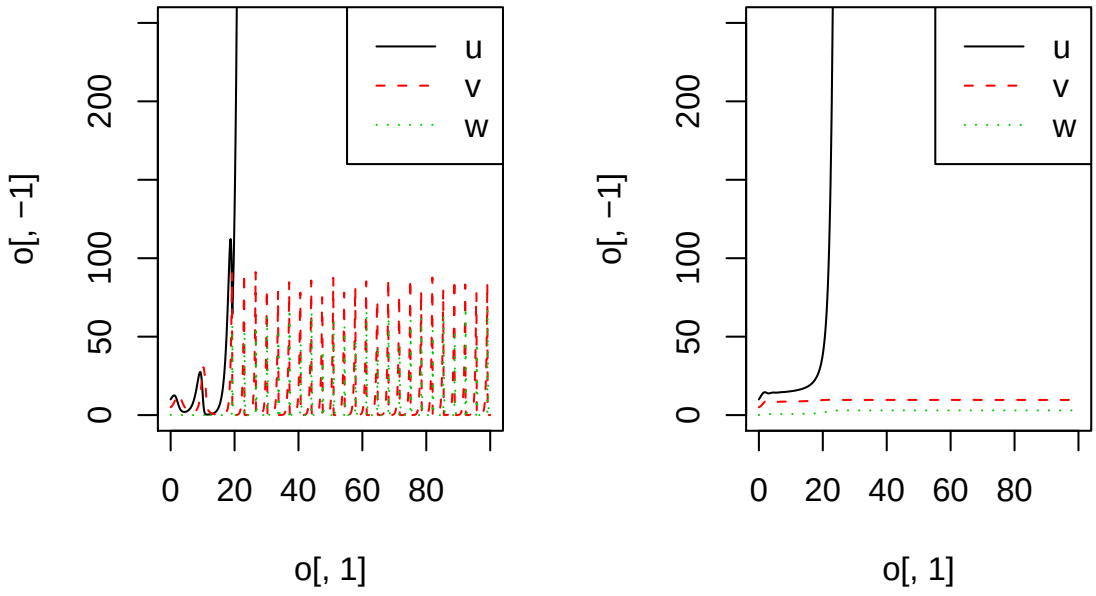

Figure 2: Two scenarios of the UPCA model (left: wstar=0, right: wstar=0.1; functional response f is Holling II).

```
R> sc1 <- sim(sc1)
R> sc2 <- sim(sc2)
R> par(mfrow = c(1, 2))
R> plotupca(sc1, ylim = c(0, 20))
R> plotupca(sc2, ylim = c(0, 20))
```

Now, we get a stable 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` was 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 allow the latter, **simecol** has an optional slot "equations", that can hold a list of submodels. Such an equations-slot can be defined either during object creation, or functions may be added afterwards. In the following, we derive two new clones with default parameter settings from the original `upca`-object, and then assign one version (the Holling II functional response) to scenario 1 and the other version (simple multiplicative Lotka-Volterra functional response) to scenario 2 (figure 4):

```
R> sc1 <- sc2 <- upca
```

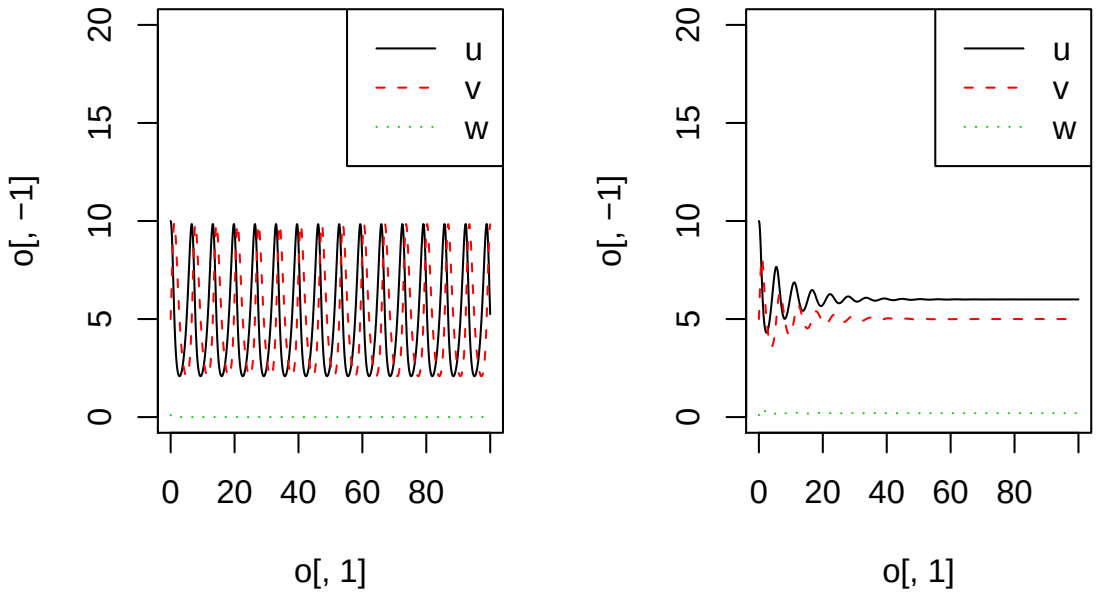

Figure 3: Two scenarios of the UPCA model (left: wstar=0, right: wstar=0.1; functional response f is Holling II).

```
R> equations(sc1)$f <- function(x, y, k) {
+      x * y/(1 + k * x)
+  }
R> equations(sc2)$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, 20))
R> plotupca(sc2, ylim = c(0, 20))
```

This method allows to compare models with different structure in the same way like scenarios with different parameter values. In addition, 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)
```

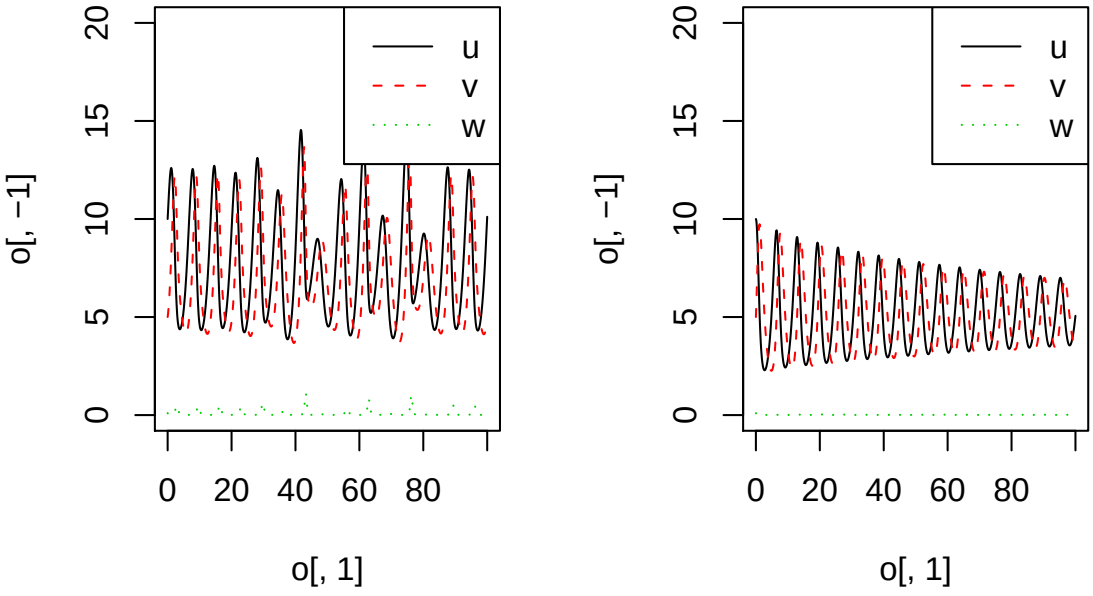

Figure 4: Two scenarios of the UPCA model (left: functional response `f` is Holling II, right functional response is Lotka-Volterra).

```
+          dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
+          list(c(du, dv, dw))
+      })
+  }, 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
```

1.4. Debugging

As stated before, all-in-one encapsulation of all functions and data in `simObjects` has many advantages, but there is also one disadvantage, namely debugging. Debugging of S4 objects is sometimes cumbersome, especially if 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 appear as an additional burden during model building, in particular if these models are technically not that simple as in our example.

Fortunately, there are easy workarounds. One of them is implementing the technically challenging parts in the user-workspace first using the above mentioned mixed style. Then, after developing and debugging the model and if everything works satisfactory, integrating the parts into the object is straightforward, given that you keep the general structure in mind. In the example below, we implement the main model as a 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)
```

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" ...
```

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

```
..@ init     : Named num [1:3] 10 5 0.1
.. ..- attr(*, "names")= chr [1:3] "u" "v" "w"
..@ observer : NULL
..@ main     :function (time, y, parms)
..@ 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
```

Now, you can delete `f` and `fmain` and you have a clean workspace with only the necessary objects.

2. 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 `simObj`ects 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 `new` with all the slots which creates the S4-object (see section 1.2), but there are also other possibilities.

simecol objects can be saved in machine readable 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
```

Conversion of the S4 object to a list representation is another possibility, that yields a representation that is readable by humans **and** by R:

```
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.

3. Methods to work with S4 objects

```
R> showMethods("sim")

Function: sim (package simecol)
obj="gridModel"
obj="odeModel"
obj="simObj"

R> getMethod("sim", "odeModel")

Method Definition:

function (obj, initialize = TRUE, ...)
{
    if (initialize & !is.null(obj@initfunc))
        obj <- initialize(obj)
```

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

```
    times <- fromtoby(obj@times)
    func <- obj@main
    inputs <- obj@inputs
    equations <- obj@equations
    environment(func) <- environment()
    equations <- addtoenv(equations)
    out <- do.call(obj@solver, list(obj@init, times, func, obj@parms,
        ...))
    obj@out <- out
    invisible(obj)
}
<environment: namespace:simecol>

Signatures:
        obj
target  "odeModel"
defined "odeModel"
```

4. Implementing models in compiled languages

Compilation of model code can speed up simulations considerably and there are several ways to call compiled code from R. So, it is possible 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** vignette "Writing Code in Compiled Language", 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
```

```
/* 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 check for the 2 'additional outputs'
  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];

  // the 2 additional outputs, here for demo purposes only
  yout[0] = aGlobalVar;
  yout[1] = ydot[0];
}
```

Using `#define` macros are a typical C-trick to get readable names for the parameters. This method is simple and efficient, and there are, of course, more elaborate possibilities. One alternative is using dynamic variables, another is doing call-backs to proglangR.

The C code can now be compiled into a so-called shared library (on Linux) or a DLL on Windows, that can be linked to R.

Compilation requires an installed C compiler (**gcc**) and some other tools that are quite standard on Linux, and which are also available for the Macintosh or, form of the **R-Tools** collection[4] provided by Duncan Murdoch for Windows.

If the tools are installed, compilation can be done directly from R with:

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

The result, a shared library or DLL can now be linked to the current R session with `dyn.load`, that we show here for Windows, and what is quite similar for Linux (see R Development Core Team 2006, for details). Note that you set the working directory of R to the path where the DLL resides or use the full path in the call to `dyn.load`.

```
R> modeldll <- dyn.load("clotka.dll")
```

You can now call the derivatives `dlotka` of the model in the `main` function of a **simecol**-object, but we don't want to go this way here. What we do is going the more efficient way, i.e. we tell the solver `lsoda` where it finds the model in the DLL.

The trick consists of two parts:

[4] http://www.murdoch-sutherland.com/Rtools/

1. we write an almost empty `main` function that returns all the information that the ODE solver needs in form of a list,
2. instead of putting a character reference to an existing solver function into the `solver` slot (e.g. `"lsoda"`) we write a user-defined interface to the solver and assign it to the solver-slot as shown in the example.

Now, we can simulate our model as usual, but avoid interpretation and communication overhead of R during the integration.

```
clotka <- new("odeModel",
  ## note that 'main' does not contain any equations directly
  ## but returns information where these can be found
  ## 'nout' is the number of 'additional outputs'
  main = function(time, init, parms) {
     # a list with: dllname, func, [jacfunc], nout
     list(lib     = "clotka",
          func    = "dlotka",
          jacfunc = NULL,
          nout    = 2)
  },
  ## parms, times, init are provided as usual, enabling
  ## scenario control like for 'ordinary' simecol models
  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),
  ## special solver function that evaluates funclist
  ## and passes its contents directly to the lsoda
  ## in the 'compiled function' mode
  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, ...)
    )
  }
)

clotka <- sim(clotka)

## the two graphics on top are the states
## the other are additional variables returned by the C code
## (for demonstration purposes here)
plot(clotka)

## Another simulation with more time steps
times(clotka)["to"] <- 1000
plot(sim(clotka))
```

```
## another simulation with intentionally reduced accuracy
## for testing
plot(sim(clotka, atol=1))

dyn.unload(as.character(modeldll[2]))
```

You should note a considerable speed-up and you may ask if this is still a **simecol** object, because the main parts are now in C and you may also ask, why one should still write models in R if C or FORTRAN are so much faster.

The answer is, that speed of computation is not the only factor. What counts is a good compromise between execution speed and programming effort. Programming in scripting languages like R is much more convenient than in compiled languages like C or FORTRAN and programming in compiled languages does only pay its effort required if models are quite large or if a large number of model runs is required. And even in such cases, a mixed approach R **and** C can be efficient, because it is only necessary to implement the core functionality of the model in C and most of data manipulation and scenario control can be done in R.

simecol follows exactly this philosophy. Implementing everything in R is highly productive if speed is of minor importance, and but you **can** use C etc. whenever necessary, and even in that case you still have the scenario management and data manipulation features of **simecol**.

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.

Soetaert K, Petzoldt T, Setzer RW (2009). ***deSolve**: General Solvers for Ordinary Differential Equations (ODE) and for Differential Algebraic Equations (DAE)*. R package version 1.2-3.

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/