R Package deSolve, Writing Code in Compiled Languages

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)
Yerseke
The Netherlands

Thomas Petzoldt
Technische Universität Dresden
Germany

R. Woodrow Setzer
National Center for Computational Toxicology
US Environmental Protection Agency

Abstract

This document describes how to use the deSolve package (Soetaert, Petzoldt, and Setzer 2010a) to solve models that are written in FORTRAN or C.

Keywords: differential equation solvers, compiled code, performance, FORTRAN, C.

1. Introduction

deSolve (Soetaert et al. 2010a; Soetaert, Petzoldt, and Setzer 2010b), the successor of R package odesolve (Setzer 2001) is a package to solve ordinary differential equations (ODE), differential algebraic equations (DAE) and partial differential equations (PDE). One of the prominent features of deSolve is that it allows specifying the differential equations either as:

- pure R code (R Development Core Team 2008),
- functions defined in lower-level languages such as FORTRAN, C, or C++, which are compiled into a dynamically linked library (DLL) and loaded into R.

In what follows, these implementations will be referred to as R models and DLL models respectively. Whereas R models are easy to implement, they allow simple interactive development, produce highly readable code and access to Rs high-level procedures, DLL models have the benefit of increased simulation speed. Depending on the problem, there may be a gain of up to several orders of magnitude computing time when using compiled code.

Here are some rules of thumb when it is worthwhile or not to switch to DLL models:

- As long as one makes use only of Rs high-level commands, the time gain will be modest. This was demonstrated in Soetaert et al. (2010a), where a formulation of two interacting populations dispersing on a 1-dimensional or a 2-dimensional grid led to a time gain of a factor two only when using DLL models.

- Generally, the more statements in the model, the higher will be the gain of using compiled code. Thus, in the same paper (Soetaert et al. 2010a), a very simple, 0-D, Lotka-Volterra type of model describing only 2 state variables was solved 50 times faster when using compiled code.
As even R models are quite performant, the time gain induced by compiled code will often not be discernible when the model is only solved once (who can grasp the difference between a run taking 0.001 or 0.05 seconds to finish). However, if the model is to be applied multiple times, e.g. because the model is to be fitted to data, or its sensitivity is to be tested, then it may be worthwhile to implement the model in a compiled language.

Starting from deSolve version 1.4, it is now also possible to use forcing functions in compiled code. These forcing functions are automatically updated by the integrators. See last chapter.

2. A simple ODE example

Assume the following simple ODE (which is from the LSODA source code):

\[
\begin{align*}
\frac{dy_1}{dt} &= -k_1 \cdot y_1 + k_2 \cdot y_2 \cdot y_3 \\
\frac{dy_2}{dt} &= k_1 \cdot y_1 - k_2 \cdot y_2 \cdot y_3 - k_3 \cdot y_2 \cdot y_2 \\
\frac{dy_3}{dt} &= k_3 \cdot y_2 \cdot y_2
\end{align*}
\]

where \( y_1, y_2 \) and \( y_3 \) are state variables, and \( k_1, k_2 \) and \( k_3 \) are parameters.

We first implement and run this model in pure R, then show how to do this in C and in FORTRAN.

2.1. ODE model implementation in R

An ODE model implemented in pure R should be defined as:

\[
yprime = \text{func}(t, y, \text{parms}, \ldots)
\]

where \( t \) is the current time point in the integration, \( y \) is the current estimate of the variables in the ODE system, and \( \text{parms} \) is a vector or list containing the parameter values. The optional dots argument (\( \ldots \)) can be used to pass any other arguments to the function. The return value of \( \text{func} \) should be a list, whose first element is a vector containing the derivatives of \( y \) with respect to time, and whose next elements contain output variables that are required at each point in time.

The R implementation of the simple ODE is given below:

\[
\begin{verbatim}
R> model <- function(t, Y, parameters) {
+   with (as.list(parameters),{
+       dy3 = k3*Y[2]*Y[2]
+       dy2 = -dy1 - dy3
+   )
+ }
\end{verbatim}
\]
The Jacobian \( \left( \frac{\partial y}{\partial y'} \right) \) associated to the above example is:

```r
R> jac <- function (t, Y, parameters) {
+   with (as.list(parameters),{
+     PD[1,1] <- -k1
+     PD[1,2] <- k2*Y[3]
+     PD[1,3] <- k2*Y[2]
+     PD[2,1] <- k1
+     PD[2,3] <- -PD[1,3]
+     PD[3,2] <- k3*Y[2]
+     PD[2,2] <- -PD[1,2] - PD[3,2]
+     return(PD)
+   })
+ }
```

This model can then be run as follows:

```r
R> parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
R> Y <- c(1.0, 0.0, 0.0)
R> times <- c(0, 0.4*10^(0:11))
R> PD <- matrix(nrow = 3, ncol = 3, data = 0)
R> out <- ode(Y, times, model, parms = parms, jacfunc = jac)
```

### 2.2. ODE model implementation in C

In order to create compiled models (.DLL = dynamic link libraries on Windows or .so = shared objects on other systems) you must have a recent version of the GNU compiler suite installed, which is quite standard for Linux. Windows users find all the required tools on [http://www.murdoch-sutherland.com/Rtools/](http://www.murdoch-sutherland.com/Rtools/). Getting DLLs produced by other compilers to communicate with R is much more complicated and therefore not recommended. More details can be found on [http://cran.r-project.org/doc/manuals/R-admin.html](http://cran.r-project.org/doc/manuals/R-admin.html).

The call to the derivative and Jacobian function is more complex for compiled code compared to R-code, because it has to comply with the interface needed by the integrator source codes. Below is an implementation of this model in C:

```c
/* file mymod.c */
#include <R.h>
static double parms[3];
```
The implementation in C consists of three parts:

1. After defining the parameters in global C-variables, through the use of \#define statements, a function called initmod initialises the parameter values, passed from the R-code.

   This function has as its sole argument a pointer to C-function odeparms that fills a double array with double precision values, to copy the parameter values into the global variable.
2. Function `derivs` then calculates the values of the derivatives. The derivative function is defined as:

```c
void derivs (int *neq, double *t, double *y, double *ydot,
            double *yout, int *ip)
```

where `*neq` is the number of equations, `*t` is the value of the independent variable, `*y` points to a double precision array of length `*neq` that contains the current value of the state variables, and `*ydot` points to an array that will contain the calculated derivatives.

`*yout` points to a double precision vector whose first `nout` values are other output variables (different from the state variables `y`), and the next values are double precision values as passed by parameter `rpar` when calling the integrator. The key to the elements of `*yout` is set in `*ip`

`*ip` points to an integer vector whose length is at least 3; the first element (`ip[0]`) contains the number of output values (which should be equal or larger than `nout`), its second element contains the length of `*yout`, and the third element contains the length of `*ip`; next are integer values, as passed by parameter `ipar` when calling the integrator.\(^1\)

Note that, in function `derivs`, we start by checking whether enough memory is allocated for the output variables (if `ip[0] < 1`), else an error is passed to R and the integration is stopped.

3. In C, the call to the function that generates the Jacobian is as:

```c
void jac(int *neq, double *t, double *y, int *ml,
         int *mu, double *pd, int *nrowpd, double *yout, int *ip)
```

where `*ml` and `*mu` are the number of non-zero bands below and above the diagonal of the Jacobian respectively. These integers are only relevant if the option of a banded Jacobian is selected. `*nrow` contains the number of rows of the Jacobian. Only for full Jacobian matrices, is this equal to `*neq`. In case the Jacobian is banded, the size of `*nrowpd` depends on the integrator. If the method is one of `lsode`, `lsoda`, `vode`, then `*nrowpd` will be equal to `*mu + 2 * *ml + 1`, where the last `*ml` rows should be filled with 0s.

For `radau`, `*nrowpd` will be equal to `*mu + *ml + 1`

See example “odeband” in the directory `doc/examples/dynload`, and chapter 4.6.

### 2.3. ODE model implementation in FORTRAN

Models may also be defined in FORTRAN.

---

\(^1\)Readers familiar with the source code of the ODEPACK solvers may be surprised to find the double precision vector `yout` and the integer vector `ip` at the end. Indeed none of the ODEPACK functions allow this, although it is standard in the `vode` and `daspk` codes. To make all integrators compatible, we have altered the ODEPACK FORTRAN codes to consistently pass these vectors.
R Package *deSolve*, Writing Code in Compiled Languages

```fortran
! file mymodf.f
!
! subroutine initmod(odeparms)
!  external odeparms
!  double precision parms(3)
!  common /myparms/parms
!
!     call odeparms(3, parms)
!     return
! end
!
! subroutine derivs (neq, t, y, ydot, yout, ip)
!  double precision t, y, ydot, k1, k2, k3
!  integer neq, ip(*)
!  dimension y(3), ydot(3), yout(*)
!  common /myparms/k1,k2,k3
!
!     if(ip(1) < 1) call rexit("nout should be at least 1")
!
!     ydot(1) = -k1*y(1) + k2*y(2)*y(3)
!     ydot(3) = k3*y(2)*y(2)
!     ydot(2) = -ydot(1) - ydot(3)
!
!     yout(1) = y(1) + y(2) + y(3)
!     return
! end
!
! subroutine jac (neq, t, y, ml, mu, pd, nrowpd, yout, ip)
!  integer neq, ml, mu, nrowpd, ip
!  double precision y(*), pd(nrowpd,*), yout(*), t, k1, k2, k3
!  common /myparms/(*), pd(nrowpd,*)
!
!     pd(1,1) = -k1
!     pd(2,1) = k1
!     pd(3,1) = 0.0
!     pd(1,2) = k2*y(3)
!     pd(2,2) = -k2*y(3) - 2*k3*y(2)
!     pd(3,2) = 2*k3*y(2)
!     pd(1,3) = k2*y(2)
!     pd(2,3) = -k2*y(2)
!     pd(3,3) = 0.0
!
!     return
! end
!
! c end of file mymodf.f
```

In **FORTRAN**, parameters may be stored in a common block (here called *myparms*). During the initialisation, this common block is defined to consist of a 3-valued vector (unnamed), but in the subroutines *derivs* and *jac*, the parameters are given a name (*k1*, ...).
2.4. Running ODE models implemented in compiled code

To run the models described above, the code in mymod.f and mymod.c must first be compiled. This can simply be done in R itself, using the system command:

\[ R> \text{system("R CMD SHLIB mymod.f")} \]

for the FORTRAN code or

\[ R> \text{system("R CMD SHLIB mymod.c")} \]

for the C code.

This will create file mymod.dll on windows, or mymod.so on other platforms.

We load the DLL, in windows as:

\[ \text{dyn.load("mymod.dll")} \]

and in unix:

\[ \text{dyn.load("mymod.so")} \]

or, using a general statement:

\[ \text{dyn.load(paste("mymod", .Platform$dynlib.ext, sep = ",")}) \]

The model can now be run as follows:

\[
\begin{align*}
\text{parms} & \leftarrow \text{c(k1 = 0.04, k2 = 1e4, k3=3e7)} \\
\text{Y} & \leftarrow \text{c(y1 = 1.0, y2 = 0.0, y3 = 0.0)} \\
\text{times} & \leftarrow \text{c(0, 0.4*10^{(0:11) } )} \\
\text{out} & \leftarrow \text{ode(Y, times, func = "derivs", parms = parms,} \\
& \quad \text{jacfunc = "jac", dllname = "mymod",} \\
& \quad \text{initfunc = "initmod", nout = 1, outnames = "Sum")}
\end{align*}
\]

The integration routine (here ode) recognizes that the model is specified as a DLL due to the fact that arguments func and jacfunc are not regular R-functions but character strings. Thus, the integrator will check whether the function is loaded in the DLL with name mymod.

Note that mymod, as specified by dllname gives the name of the shared library without extension. This DLL should contain all the compiled function or subroutine definitions referred to in func, jacfunc and initfunc.

Also, if func is specified in compiled code, then jacfunc and initfunc (if present) should also be specified in a compiled language. It is not allowed to mix R-functions and compiled functions.

\[ ^2 \text{This requires a correctly installed GNU compiler, see above.} \]
Note also that, when invoking the integrator, we have to specify the number of ordinary output variables, \texttt{nout}. This is because the integration routine has to allocate memory to pass these output variables back to \texttt{R}. There is no way to check for the number of output variables in a DLL automatically. If in the calling of the integration routine the number of output variables is too low, then \texttt{R} may freeze and need to be terminated! Therefore it is advised that one checks in the code whether \texttt{nout} has been specified correctly. In the \texttt{FORTRAN} example above, the statement \texttt{if (ip(1) < 1) call rexit("nout should be at least 1") does this. Note that it is not an error (just a waste of memory) to set \texttt{nout} to a too large value.

Finally, in order to label the output matrix, the names of the ordinary output variables have to be passed explicitly (\texttt{outnames}). This is not necessary for the state variables, as their names are known through their initial condition (\texttt{y}).

3. Alternative way of passing parameters and data in compiled code

All of the solvers in \texttt{deSolve} take an argument \texttt{parms} which may be an arbitrary \texttt{R} object. In models defined in \texttt{R} code, this argument is passed unprocessed to the various functions that make up the model. It is possible, as well, to pass such \texttt{R}-objects to models defined in native code.

The problem is that data passed to, say, \texttt{ode} in the argument \texttt{parms} is not visible by default to the routines that define the model. This is handled by a user-written initialization function, for example \texttt{initmod} in the \texttt{C} and \texttt{FORTRAN} examples from sections 2.2 and 2.3. However, these set only the \texttt{values} of the parameters.

\texttt{R}-objects have many attributes that may also be of interest. To have access to these, we need to do more work, and this mode of passing parameters and data is much more complex than what we saw in previous chapters.

In \texttt{C}, the initialization routine is declared:

\begin{verbatim}
void initmod(void (* odeparms)(int *, double *));
\end{verbatim}

That is, \texttt{initmod} has a single argument, a pointer to a function that has as arguments a pointer to an \texttt{int} and a pointer to a \texttt{double}. In \texttt{FORTRAN}, the initialization routine has a single argument, a subroutine declared to be external. The name of the initialization function is passed as an argument to the \texttt{deSolve} solver functions.

In \texttt{C}, two approaches are available for making the values passed in \texttt{parms} visible to the model routines, while only the simpler approach is available in \texttt{FORTRAN}. The simpler requires that \texttt{parms} be a numeric vector. In \texttt{C}, the function passed from \texttt{deSolve} to the initialization function (called \texttt{odeparms} in the example) copies the values from the parameter vector to a static array declared globally in the file where the model is defined. In \texttt{FORTRAN}, the values are copied into a \texttt{COMMON} block.

It is possible to pass more complicated structures to \texttt{C} functions. Here is an example, an initializer called \texttt{deltamethrin} from a model describing the pharmacokinetics of that pesticide:

\begin{verbatim}
#include <R.h>
#include <Rinternals.h>
#include <R_ext/Rdynload.h>
\end{verbatim}
#include "deltamethrin.h"

/* initializer */
void deltamethrin(void(* odeparms)(int *, double *))
{
    int Nparms;
    DL_FUNC get_deSolve_gparms;
    SEXP gparms;
    get_deSolve_gparms = R_GetCCallable("deSolve","get_deSolve_gparms");
    gparms = get_deSolve_gparms();
    Nparms = LENGTH(gparms);
    if (Nparms != N_PARMS) {
        PROBLEM "Confusion over the length of parms"
        ERROR;
    } else {
        _RDy_deltamethrin_parms = REAL(gparms);
    }
}

In deltamethrin.h, the variable _RDy_deltamethrin_parms and macro N_PARMS are declared:

#define N_PARMS 63
static double *_RDy_deltamethrin_parms;

The critical element of this method is the function R_GetCCallable which returns a function (called get_deSolve_gparms in this implementation) that returns the parms argument as a SEXP data type. In this example, parms was just a real vector, but in principle, this method can handle arbitrarily complex objects. For more detail on handling R objects in native code, see R Development Core Team (2008).

4. deSolve integrators that support DLL models

In the most recent version of deSolve all integration routines can solve DLL models. They are:

- all solvers of the lsode family: lsoda, lsode, lsodar, lsodes,
- vode, zvode,
- daspk,
- radau,
- the Runge-Kutta integration routines (including the Euler method).
For some of these solvers the interface is slightly different (e.g. zvode, daspk), while in others (lsodar, lsodes) different functions can be defined. How this is implemented in a compiled language is discussed next.

4.1. Complex numbers, function zvode

zvode solves ODEs that are composed of complex variables. The program below uses zvode to solve the following system of 2 ODEs:

\[
\begin{align*}
\frac{dz}{dt} &= i \cdot z \\
\frac{dw}{dt} &= -i \cdot w \cdot w \cdot z
\end{align*}
\]

where

\[
\begin{align*}
w(0) &= 1/2.1 + 0i \\
z(0) &= 1i
\end{align*}
\]

on the interval \( t = [0, 2 \pi] \)

The example is implemented in FORTRAN\(^3\), FEX implements the function func:

```fortran
SUBROUTINE FEX (NEQ, T, Y, YDOT, RPAR, IPAR)
INTEGER NEQ, IPAR(*)
DOUBLE COMPLEX Y(NEQ), YDOT(NEQ), RPAR(*), CMP
DOUBLE PRECISION T
character(len=100) msg

C the imaginary unit i
CMP = DCMPLX(0.0D0,1.0D0)

YDOT(1) = CMP*Y(1)
YDOT(2) = -CMP*Y(2)*Y(2)*Y(1)

RETURN
END
```

JEX implements the function jacfunc

```fortran
SUBROUTINE JEX (NEQ, T, Y, ML, MU, PD, NRPD, RPAR, IPAR)
INTEGER NEQ, ML, MU, NRPD, IPAR(*)
DOUBLE COMPLEX Y(NEQ), PD(NRPD,NEQ), RPAR(*), CMP
DOUBLE PRECISION T

C the imaginary unit i
```

---

\(^3\)this can be found in file "zvodedll.f", in the dynload subdirectory of the package
Karline Soetaert, Thomas Petzoldt, R. Woodrow Setzer

\[ \text{CMP} = \text{DCMPLX}(0.0, 1.0) \]

\[ \text{PD}(2,3) = -2.0 \times \text{CMP} \times Y(1) \times Y(2) \]
\[ \text{PD}(2,1) = -\text{CMP} \times Y(2) \times Y(2) \]
\[ \text{PD}(1,1) = \text{CMP} \]

RETURN
END

Assuming this code has been compiled and is in a DLL called "zvodedll.dll", this model is solved in R as follows:

\[
\text{dyn.load("zvodedll.dll")}
\]
\[
\text{outF} \leftarrow \text{zvode(func = "fex", jacfunc = "jex", y = yini, parms = NULL, times = times, atol = 1e-10, rtol = 1e-10, dllname = "zvodedll", initfunc = NULL)}
\]

Note that in R names of FORTRAN DLL functions (e.g. for func and jacfunc) have to be given in lowercase letters, even if they are defined upper case in FORTRAN.

Also, there is no initialiser function here (initfunc = NULL).

4.2. DAE models, integrator daspk
daspk is one of the integrators in the package that solve DAE models. In order to be used with DASPK, DAEs are specified in implicit form:

\[ 0 = F(t, y, y', p) \]
i.e. the DAE function (passed via argument res) specifies the “residuals” rather than the derivatives (as for ODEs).

Consequently the DAE function specification in a compiled language is also different. For code written in C, the calling sequence for res must be:

\[
\text{void myres(double *t, double *y, double *ydot, double *cj, double *delta, int *ires, double *yout, int *ip)}
\]

where *t is the value of the independent variable, *y points to a double precision vector that contains the current value of the state variables, *ydot points to an array that will contain the derivatives, *delta points to a vector that will contain the calculated residuals. *cj points to a scalar, which is normally proportional to the inverse of the stepsizes, while *ires points to an integer (not used). *yout points to any other output variables (different from the state variables y), followed by the double precision values as passed via argument rpar; finally *ip is an integer vector containing at least 3 elements, its first value (*ip[0]) equals the number of output variables, calculated in the function (and which should be equal to nout), its second element equals the total length of *yout, its third element equals the total length of *ip, and finally come the integer values as passed via argument ipar.

For code written in FORTRAN, the calling sequence for res must be as in the following example:
subroutine myresf(t, y, ydot, cj, delta, ires, out, ip)
  integer :: ires, ip(*)
  integer, parameter :: neq = 3
  double precision :: t, y(neq), ydot(neq), delta(neq), out(*)
  double precision :: K, ka, r, prod, ra, rb
  common /myparms/K, ka, r, prod

  if(ip(1) < 1) call rexit("nout should be at least 1")
  ra = ka * y(3)
  rb = ka/K * y(1) * y(2)

  !! residuals of rates of changes
  delta(3) = -ydot(3) - ra + rb + prod
  delta(1) = -ydot(1) + ra - rb
  delta(2) = -ydot(2) + ra - rb - r*y(2)
  out(1) = y(1) + y(2) + y(3)
  return
end

Similarly as for the ODE model discussed above, the parameters are kept in a common block
which is initialised by an initialiser subroutine:

subroutine initpar(daspkparms)
  external daspkparms
  integer, parameter :: N = 4
  double precision parms(N)
  common /myparms/parms
  call daspkparms(N, parms)
  return
end

See the ODE example for how to initialise parameter values in C.
Similarly, the function that specifies the Jacobian in a DAE differs from the Jacobian when
the model is an ODE. The DAE Jacobian is set with argument jacres rather than jacfunc
when an ODE.

For code written in FORTRAN, the jacres must be as:

subroutine resjacfor (t, y, dy, pd, cj, out, ipar)
  integer, parameter :: neq = 3
  integer :: ipar(*)
  double precision :: K, ka, r, prod
  double precision :: pd(neq,neq), y(neq), dy(neq), out(*)
  common /myparms/K, ka, r, prod
!res1 = -dD - ka*D + ka/K*A*B + prod
   PD(1,1) = ka/K*y(2)
   PD(1,2) = ka/K*y(1)
   PD(1,3) = -ka -cj

!res2 = -dA + ka*D - ka/K*A*B
   PD(2,1) = -ka/K*y(2) -cj
   PD(2,2) = -ka/K*y(2)
   PD(2,3) = ka

!res3 = -dB + ka*D - ka/K*A*B - r*B
   PD(3,1) = -ka/K*y(2)
   PD(3,2) = -ka/K*y(2) -r -cj
   PD(3,3) = ka

return
end

4.3. DAE models, integrator radau

Function radau solves DAEs in linearly implicit form, i.e. in the form $M \dot{y} = f(t, y, p)$. The derivative function $f$ is specified in the same way as for an ODE, i.e.

```c
void derivs (int *neq, double *t, double *y, double *ydot, double *yout, int *ip)
```

and

```fortran
subroutine derivs (neq, t, y, ydot, out, IP)
```

for C and FORTRAN code respectively.

To show how it should be used, we implement the caraxis problem as in (Mazzia and Magherini 2008). The implementation of this index 3 DAE, comprising 8 differential, and 2 algebraic equations in R is the last example of the radau help page. We first repeat the R implementation:

```r
R> caraxisfun <- function(t, y, parms) {
+   with(as.list(c(y, parms)), {
+     yb <- r * sin(w * t)
+     xb <- sqrt(L * L - yb * yb)
+     Ll <- sqrt(xl^2 + yl^2)
+     Lr <- sqrt((xr - xb)^2 + (yr - yb)^2)
+     +
+     dxl <- ul; dyl <- vl; dxr <- ur; dyr <- vr
+     +
+     dul <- (L0-Ll) * xl/Ll + 2 * lam2 * (xl-xr) + lam1*xb
+     dv1 <- (L0-Ll) * yl/Ll + 2 * lam2 * (yl-yr) + lam1*yb - k * g
+     +
+     dur <- (L0-Lr) * (xr-xb)/Lr - 2 * lam2 * (xl-xr)
+   })
+ }
```
The implementation in FORTRAN consists of an initialiser function and a derivative function.

```fortran
! Initialiser for parameter common block

subroutine initcaraxis(daeparms)
```

---

The implementation in FORTRAN consists of an initialiser function and a derivative function.
Figure 1: Solution of the caraxis model - see text for R-code

external daeparms
integer, parameter :: N = 8
double precision parms(N)
common /myparms/parms

call daeparms(N, parms)
return
end

c rate of change

c subroutine caraxis(neq, t, y, ydot, out, ip)
implicit none
integer neq, IP(*)
double precision t, y(neq), ydot(neq), out(*)
double precision eps, M, k, L, L0, r, w, g
common /myparms/ eps, M, k, L, L0, r, w, g
double precision xl, yl, xr, yr, ul, vl, ur, vr, lam1, lam2
double precision yb, xb, Ll, Lr, dxl, dyl, dxr, dyr
double precision dul, dv1, dur, dvr, c1, c2
c expand state variables
  xl = y(1)
yl = y(2)
xr = y(3)
yr = y(4)
ul = y(5)
v1 = y(6)
ur = y(7)
vr = y(8)
lam1 = y(9)
lam2 = y(10)

yb = r * sin(w * t)
xb = sqrt(L * L - yb * yb)
Ll = sqrt(xl**2 + yl**2)
Lr = sqrt((xr - xb)**2 + (yr - yb)**2)

dxl = ul
dyl = vl
dx = ur
dyr = vr

dul = (L0-Ll) * xl/Ll + 2 * lam2 * (xl-xr) + lam1*xb
dvl = (L0-Ll) * yl/Ll + 2 * lam2 * (yl-yr) + lam1*yb - k*g
dur = (L0-Lr) * (xr-xb)/Lr - 2 * lam2 * (xl-xr)
dvr = (L0-Lr) * (yr-yb)/Lr - 2 * lam2 * (yl-yr) - k*g

c1 = xb * xl + yb * yl
c2 = (x1 - xr)**2 + (y1 - yr)**2 - L * L

c function values in ydot
ydot(1) = dxl
ydot(2) = dyl
ydot(3) = dxr
ydot(4) = dyr
ydot(5) = dul
ydot(6) = dvl
ydot(7) = dur
ydot(8) = dvr
ydot(9) = c1
ydot(10) = c2

return
end

Assuming that the code is in file “radaudae.f”, this model is compiled, loaded and solved in R as:

system("R CMD SHLIB radaudae.f")
dyn.load(paste("radaudae", .Platform$dynlib.ext, sep = "))
outDLL <- radau(y = yini, mass = Mass, times = times, func = "caraxis",
initfunc = "initcaraxis", parms = parameter,
dllname = "radaudae", nind = index)
4.4. The root function from integrators lsodar and lsode

lsodar is an extended version of integrator lsoda that includes a root finding function. This function is specified via argument rootfunc. In deSolve version 1.7, lsode has also been extended with root finding capabilities.

Here is how to program such a function in a lower-level language. For code written in C, the calling sequence for rootfunc must be:

```c
void myroot(int *neq, double *t, double *y, int *ng, double *gout, double *out, int *ip )
```

where *neq and *ng are the number of state variables and root functions respectively, *t is the value of the independent variable, y points to a double precision array that contains the current value of the state variables, and gout points to an array that will contain the values of the constraint function whose root is sought. *out and *ip are a double precision and integer vector respectively, as described in the ODE example above.

For code written in FORTRAN, the calling sequence for rootfunc must be as in following example:

```fortran
subroutine myroot(neq, t, y, ng, gout, out, ip)
    integer :: neq, ng, ip(*)
    double precision :: t, y(neq), gout(ng), out(*)
    gout(1) = y(1) - 1.e-4
    gout(2) = y(3) - 1e-2
    return
end
```

4.5. jacvec, the Jacobian vector for integrator lsodes

Finally, in integration function lsodes, not the Jacobian matrix is specified, but a vector, one for each column of the Jacobian. This function is specified via argument jacvec.

In FORTRAN, the calling sequence for jacvec is:

```fortran
SUBROUTINE JAC (NEQ, T, Y, J, IAN, JAN, PDJ, OUT, IP)
    DOUBLE PRECISION T, Y(*), IAN(*), JAN(*), PDJ(*), OUT(*)
    INTEGER NEQ, J, IP(*)
```

4.6. Banded jacobians in compiled code

In the call of the jacobian function, the number of bands below and above the diagonal (ml, mu) and the number of rows of the Jacobian matrix, nrowPD is specified, e.g. for FORTRAN code:
SUBROUTINE JAC (neq, T, Y, ml, mu, PD, nrowPD, RPAR, IPAR)

The jacobian matrix to be returned should have dimension nrowPD, neq.
In case the Jacobian is banded, the size of nrowPD depends on the integrator. If the method is one of lsode, lsoda, vode, or related, then nrowPD will be equal to mu + 2 * ml + 1, where the last ml rows should be filled with 0s.
For radau, nrowpd will be equal to mu + ml + 1

Thus, it is important to write the FORTRAN or C-code in such a way that it can be used with both types of integrators - else it is likely that R will freeze if the wrong integrator is used.

We implement in FORTRAN, the example of the lsode help file. The R-code reads:

```r
R> ## the model, 5 state variables
R> f1 <- function (t, y, parms) {
+   ydot <- vector(len = 5)
+   +   return(list(ydot))
+ }
R> ## the Jacobian, written in banded form
R> bandjac <- function (t, y, parms) {
+   jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
+                  data = c( 0 , -0.2, -0.2, -0.2, -0.2,
+                           0.1, 0.1, 0.1, 0.1, 0.1,
+                           -0.3, -0.3, -0.3, -0.3, 0))
+   +   return(jac)
+ }
R> ## initial conditions and output times
R> yini <- 1:5
R> times <- 1:20
R> ## stiff method, user-generated banded Jacobian
R> out <- lsode(yini, times, f1, parms = 0, jactype = "bandusr",
+               jacfunc = bandjac, bandup = 1, banddown = 1)
```

In FORTRAN, the code might look like this:

```fortran
c Rate of change
subroutine derivsband (neq, t, y, ydot, out, IP)
  integer neq, IP(*)
  DOUBLE PRECISION T, Y(5), YDOT(5), out(*)

  ydot(1) = 0.1*y(1) -0.2*y(2)
```
\begin{verbatim}
ydot(2) = -0.3*y(1) + 0.1*y(2) - 0.2*y(3)
ydot(3) = -0.3*y(2) + 0.1*y(3) - 0.2*y(4)
ydot(4) = -0.3*y(3) + 0.1*y(4) - 0.2*y(5)
ydot(5) = -0.3*y(4) + 0.1*y(5)
RETURN
END

c The banded jacobian
subroutine jacband (neq, t, y, ml, mu, pd, nrowpd, RP, IP)
INTEGER neq, ml, mu, nrowpd, ip(*)
DOUBLE PRECISION T, Y(5), PD(nrowpd,5), rp(*)
PD(:, :) = 0.D0
PD(1, 1) = 0.D0
PD(1, 2) = -.02D0
PD(1, 3) = -.02D0
PD(1, 4) = -.02D0
PD(1, 5) = -.02D0
PD(2, :) = 0.1D0
PD(3, 1) = -0.3D0
PD(3, 2) = -0.3D0
PD(3, 3) = -0.3D0
PD(3, 4) = -0.3D0
PD(3, 5) = 0.D0
RETURN
END

Assuming that this code is in file "odeband.f", we compile from within R and load the shared library (assuming the working directory holds the source file) with:

\texttt{system("R CMD SHLIB odeband.f")}
\texttt{dyn.load(paste("odeband", .Platform$dynlib.ext, sep = ""))}

To solve this problem, we write in R

\texttt{out2 <- lsode(yini, times, "derivsband", parms = 0, jactype = "bandusr",
                    jacfunc = "jacband", bandup = 1, banddown = 1, dllname = "odeband")}
\texttt{out2 <- radau(yini, times, "derivsband", parms = 0, jactype = "bandusr",
                    jacfunc = "jacband", bandup = 1, banddown = 1, dllname = "odeband")}

This will work both for the \texttt{lsode} family as for \texttt{radau}. In the first case, when entering subroutine \texttt{jacband}, \texttt{nrowpd} will have the value 5, in the second case, it will be equal to 4.
5. Testing functions written in compiled code

Two utilities have been included to test the function implementation in compiled code:

- **DLLfunc** to test the implementation of the derivative function as used in ODEs. This function returns the derivative $\frac{dy}{dt}$ and the output variables.
- **DLLres** to test the implementation of the residual function as used in DAEs. This function returns the residual function $\frac{dy}{dt} - f(y,t)$ and the output variables.

These functions serve no other purpose than to test whether the compiled code returns what it should.

5.1. DLLfunc

We test whether the ccl4 model, which is part of deSolve package, returns the proper rates of changes. (Note: see example(ccl4model) for a more comprehensive implementation)

```
R> ## Parameter values and initial conditions
R> Parms <- c(0.182, 4.0, 4.0, 0.08, 0.04, 0.74, 0.05, 0.15, 0.32,
+              16.17, 281.48, 13.3, 16.17, 5.487, 153.8, 0.04321671,
+              0.4027255, 1000, 0.02, 1.0, 3.8)
R> yini <- c( AI=21, AAM=0, AT=0, AF=0, AL=0, CLT=0, AM=0 )
R> ## the rate of change
R> DLLfunc(y = yini, dllname = "deSolve", func = "derivsccl4",
+         initfunc = "initccl4", parms = Parms, times = 1,
+         nout = 3, outnames = c("DOSE", "MASS", "CP") )
```

```
$dy
   AI    AAM    AT    AF    AL
-20.0582048 6.2842256 9.4263383 0.9819102 2.9457307

$var
  DOSE  MASS  CP
1.758626 0.000000 922.727067
```

5.2. DLLres

The deSolve package contains a FORTRAN implementation of the chemical model described above (section 4.1), where the production rate is included as a forcing function (see next section).

Here we use DLLres to test it:

```
R> pars <- c(K = 1, ka = 1e6, r = 1)
R> ## Initial conc; D is in equilibrium with A,B
```
6. Using forcing functions

Forcing functions in DLLs are implemented in a similar way as parameters. This means:

- They are initialised by means of an initialiser function. Its name should be passed to the solver via argument `initforc`.

  Similar as the parameter initialiser function, the function denoted by `initforc` has as its sole argument a pointer to the vector that contains the forcing function values in the compiled code. In case of C code, this will be a global vector; in case of FORTRAN, this will be a vector in a common block.

  The solver puts a pointer to this vector and updates the forcing functions in this memory area at each time step. Hence, within the compiled code, forcing functions can be assessed as if they are parameters (although, in contrast to the latter, their values will generally change). No need to update the values for the current time step; this has been done before entering the `derivs` function.

- The forcing function data series are passed to the integrator, via argument `forcings`; if there is only one forcing function data set, then a 2-columned matrix (time, value) will do; else the data should be passed as a list, containing (time, value) matrices with the individual forcing function data sets. Note that the data sets in this list should be in the same ordering as the declaration of the forcings in the compiled code.

A number of options allow to finetune certain settings. They are in a list called `fcontrol` which can be supplied as argument when calling the solvers. The options are similar to the arguments from R function `approx` however the default settings are often different.

The following options can be specified:
• **method** specifies the interpolation method to be used. Choices are "linear" or "constant", the default is "linear", which means linear interpolation (same as `approx`)

• **rule**, an integer describing how interpolation is to take place outside the interval \([\text{min}(\text{times}), \text{max}(\text{times})]\). If `rule` is 1 then an error will be triggered and the calculation will stop if extrapolation is necessary. If it is 2, the default, the value at the closest data extreme is used, a warning will be printed if `verbose` is TRUE.

Note that the default differs from the `approx` default.

• **f**, for method="constant" is a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If \(y_0\) and \(y_1\) are the values to the left and right of the point then the value is \(y_0*(1-f)+y_1*f\) so that \(f=0\) is right-continuous and \(f=1\) is left-continuous. The default is to have \(f=0\). For some data sets it may be more realistic to set \(f=0.5\).

• **ties**, the handling of tied times values. Either a function with a single vector argument returning a single number result or the string "ordered".

Note that the default is "ordered", hence the existence of ties will NOT be investigated; in practice this means that, if ties exist, the first value will be used; if the dataset is not ordered, then nonsense will be produced.

Alternative values for `ties` are `mean`, `min` etc... which will average, or take the minimal value if multiple values exist at one time level.

The default settings of `fcontrol` are:

```r
fcontrol=list(method="linear", rule=2, f=0, ties="ordered")
```

Note that only ONE specification is allowed, even if there is more than one forcing function data set. (may/should change in the future).

### 6.1. A simple FORTRAN example

We implement the example from chapter 3 of the book ([Soetaert and Herman 2009](#)) in FORTRAN.

This model describes the oxygen consumption of a (marine) sediment in response to deposition of organic matter (the forcing function). One state variable, the organic matter content in the sediment is modeled; it changes as a function of the deposition \(\text{Flux}\) (forcing) and organic matter decay (first-order decay rate \(k\)).

\[
\frac{dC}{dt} = \text{Flux}_t - k \cdot C
\]

with initial condition \(C(t=0) = C_0\); the latter is estimated as the mean of the flux divided by the decay rate.

The FORTRAN code looks like this:

```fortran
c Initialiser for parameter common block
subroutine scocpar(odeparms)
```
Here the subroutine *scocpar* is business as usual; it initialises the parameter common block (there is only one parameter). Subroutine *odeforcs* does the same for the forcing function, which is also positioned in a common block, called *myforcs*. This common block is made available in the derivative subroutine (here called *scocder*), where the forcing function is
named *depo*.

At each time step, the integrator updates the value of this forcing function to the correct time point. In this way, the forcing functions can be used as if they are (time-varying) parameters. All that’s left to do is to pass the forcing function data set and the name of the forcing function initialiser routine. This is how to do it in R.

First the data are inputted:

```R
R> Flux <- matrix(ncol=2, byrow=TRUE, data=c(
  + 1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186, 
  + 93, 0.140, 103, 0.255, 113, 0.231, 123, 0.309, 133, 1.127, 143, 1.923, 
  + 153, 1.091, 163, 1.001, 173, 1.691, 183, 1.404, 194, 2.266, 204, 0.767, 
  + 214, 0.893, 224, 0.737, 234, 0.772, 244, 0.726, 254, 0.624, 264, 0.439, 
  + 274, 0.168, 284, 0.280, 294, 0.202, 304, 0.193, 315, 0.086, 325, 0.286, 
  + 335, 1.889, 345, 0.896, 355, 0.355, 365, 0.681, 365, 1.135))
R> head(Flux)

[,1] [,2]
[1,]  1 0.654
[2,] 11 0.167
[3,] 21 0.060
[4,] 41 0.070
[5,] 73 0.277
[6,] 83 0.186
```

and the parameter given a value (there is only one)

```R
R> parms <- 0.01
```

The initial condition \( Y_{ini} \) is estimated as the annual mean of the Flux and divided by the decay rate (parameter).

```R
R> meanDepo <- mean(approx(Flux[,1], Flux[,2], xout=seq(1,365,by=1))$y)
R> Yini <- c(y=meanDepo/parms)
```

After defining the output times, the model is run, using integration routine *ode*.

The name of the derivate function "*scocder*", of the dll "*deSolve*"\(^4\) and of the initialiser function "*scocpar*" are passed, as in previous examples. In addition, the forcing function data set is also passed (forcings=Flux) as is the name of the forcing initialisation function (initforc="*scocforc*").

```R
R> times <- 1:365
R> out <- ode(y=Yini, times, func = "scocder", 
+   parms = parms, dllname = "deSolve", 
+   initforc="scocforc", forcings=Flux,

\(^4\)this example is made part of the deSolve package, hence the name of the dll is "deSolve"


Now, the way the forcing functions are interpolated are changed: Rather than linear interpolation, constant (block, step) interpolation is used.

```R
fcontrol <- list(method="constant")
out2 <- ode(y=Yini, times, func = "scocder",
+         parms = parms, dllname = "deSolve",
+         initforc="scocforc", forcings=Flux, fcontrol=fcontrol,
+         initfunc = "scocpar", nout = 2,
+         outnames = c("Mineralisation","Depo"))
```

Finally, the results are plotted:

```R
par (mfrow=c(1,2))
plot(out, which = "Depo", col="red",
+     xlab="days", ylab="mmol C/m²/ d", main="method='linear'")
lines(out[,"time"], out[,"Mineralisation"], lwd=2, col="blue")
legend("topleft",lwd=1:2,col=c("red","blue"), c("Flux","Mineralisation"))
plot(out2, which = "Depo", col="red",
+     xlab="days", ylab="mmol C/m²/ d", main="method='constant'")
lines(out2[,"time"], out2[,"Mineralisation"], lwd=2, col="blue")
```

### 6.2. An example in C

Consider the following R-code which implements a resource-producer-consumer Lotka-Volterra type of model in R (it is a modified version of the example of function `ode`):

```R
SPCmod <- function(t, x, parms, input) {
  with(as.list(c(parms, x)), {
    import <- input(t)
    dS <- import - b*S*P + g*C  # substrate
    dP <- c*S*P - d*C*P         # producer
    dC <- e*P*C - f*C          # consumer
    res <- c(dS, dP, dC)
    list(res, signal = import)
  })
}
R Package `deSolve`, Writing Code in Compiled Languages

Figure 2: Solution of the SCOC model, implemented in compiled code, and including a forcing function – see text for R-code

```r
+  })
+ }

R> ## The parameters
R> parms <- c(b = 0.1, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)
R> ## vector of timesteps
R> times <- seq(0, 100, by=0.1)
R> ## external signal with several rectangle impulses
R> signal <- as.data.frame(list(times = times,
       + import = rep(0, length(times)))))
R> signal$import <- ifelse((trunc(signal$times) %% 2 == 0), 0, 1)
R> sigimp <- approxfun(signal$times, signal$import, rule = 2)
R> ## Start values for steady state
R> xstart <- c(S = 1, P = 1, C = 1)
R> ## Solve model
R> print (system.time(
+ out <- ode(y = xstart,times = times,
+       + func = SPCmod, parms, input = sigimp)
+ ))

user  system elapsed
0.165  0.003  0.170

All output is printed at once:

R> plot(out)

The C-code, in file `Forcing_lv.c`, can be found in the packages `/doc/examples/dynload` subdirectory. It can be compiled, from within R by

```r
this can be opened by typing browseURL(paste(system.file(package = "deSolve"),
"/doc/examples/dynload", sep = ""))
```
Figure 3: Solution of the Lotka-Volterra resource (S)-producer (P) - consumer (C) model with time-variable input (signal) - see text for R-code
After defining the parameter and forcing vectors, and giving them comprehensible names, the
parameter and forcing initialiser functions are defined (\texttt{parmsc} and \texttt{forcc} respectively). Next
is the derivative function, \texttt{derivsc}.

\begin{verbatim}
#include <R.h>
static double parms[6];
static double forc[1];
/* A trick to keep up with the parameters and forcings */
#define b parms[0]
#define c parms[1]
#define d parms[2]
#define e parms[3]
#define f parms[4]
#define g parms[5]
#define import forc[0]
/* initializers: */
void odec(void (* odeparms)(int *, double *))
{
  int N=6;
  odeparms(&N, parms);
}
void forcc(void (* odeforcs)(int *, double *))
{
  int N=1;
  odeforcs(&N, forc);
}
/* derivative function */
void derivsc(int *neq, double *t, double *y, double *ydot,
              double *yout, int*ip)
{
  if (ip[0]<2) error("nout should be at least 2");
  ydot[0] = import - b*y[0]*y[1] + g*y[2];
  ydot[1] = c*y[0]*y[1] - d*y[2]*y[1];
  yout[0] = y[0] + y[1] + y[2];
  yout[1] = import;
}
\end{verbatim}
After defining the forcing function time series, which is to be interpolated by the integration routine, and loading the DLL, the model is run:

```r
Sigimp <- approx(signal$times, signal$import, xout=ftime, rule = 2)$y
forcings <- cbind(ftime, Sigimp)
dyn.load("Forcing_lv.dll")
out <- ode(y=xstart, times, func = "derivsc",
           parms = parms, dllname = "Forcing_lv", initforc = "forcc",
           forcings=forcings, initfunc = "parmsc", nout = 2,
           outnames = c("Sum","signal"), method = rkMethod("rk34f"))
dyn.unload("Forcing_lv.dll")
```

This code executes about 30 times faster than the R-code.

With a longer simulation time, the difference becomes more pronounced, e.g. with times till 800 days, the DLL code executes 200 times faster\(^6\).

7. Implementing events in compiled code

An event occurs when the value of a state variable is suddenly changed, e.g. a certain amount is added, or part is removed. The integration routines cannot deal easily with such state variable changes. Typically these events occur only at specific times. In deSolve, events can be imposed by means of an input file that specifies at which time a certain state variable is altered, or via an event function.

Both types of events combine with compiled code.

Take the previous example, the Lotka-Volterra SPC model. Suppose that every 10 days, half of the consumer is removed.

We first implement these events as a `data.frame`

```r
R> eventdata <- data.frame(var=rep("C",10), time=seq(10,100,10), value=rep(0.5,10),
                           + method=rep("multiply",10))
R> eventdata

   var time value method
 1   C   10  0.5  multiply
 2   C   20  0.5       multiply
 3   C   30  0.5       multiply
 4   C   40  0.5       multiply
 5   C   50  0.5       multiply
 6   C   60  0.5       multiply
 7   C   70  0.5       multiply
 8   C   80  0.5       multiply
 9   C   90  0.5       multiply
10  C  100  0.5       multiply
```

---

\(^6\)this is due to the sequential update of the forcing functions by the solvers, compared to the bisectioning approach used by approxfun

---
This model is solved, and plotted as:

```r
dyn.load("Forcing_lv.dll")
out2 <- ode(y = y, times, func = "derivsc",
              parms = parms, dllname = "Forcing_lv", initforc="forcc",
              forcings = forcings, initfunc = "parmsc", nout = 2,
              outnames = c("Sum", "signal"), events=list(data=eventdata))
dyn.unload("Forcing_lv.dll")
plot(out2, which = c("S","P","C"), type = "l")
```

The event can also be implemented in C as:

```c
void event(int *n, double *t, double *y) {
    y[2] = y[2] * 0.5;
}
```

Here n is the length of the state variable vector y, and is then solved as:

```r
dyn.load("Forcing_lv.dll")
out3 <- ode(y = y, times, func = "derivsc",
              parms = parms, dllname = "Forcing_lv", initforc="forcc",
              forcings = forcings, initfunc = "parmsc", nout = 2,
              outnames = c("Sum", "signal"),
              events = list(func="event",time=seq(10,90,10)))
dyn.unload("Forcing_lv.dll")
```

8. Delay differential equations

It is now also very simple to implement delay differential equations in compiled code and solve them with `dede`. In order to do so, you need to get access to the R-functions `lagvalue` and `lagderiv` that will give you the past value of the state variable or its derivative respectively.

8.1. Delays implemented in Fortran

If you use Fortran, then the easiest way is to link your code with a file called `dedeUtils.c` that you will find in the packages subdirectory `inst/doc/dynload-dede`. This file contains Fortran-callable interfaces to the delay-differential utility functions from package `deSolve`, and that are written in C. Its content is:

```c
void F77_SUB(lagvalue)(double *T, int *nr, int *N, double *ytau) {
    static void(*fun)(double, int*, int, double*) = NULL;
    if (fun == NULL)
        fun = (void(*)(double, int*, int, double*))R_GetCallable("deSolve", "lagvalue");
    fun(*T, nr, *N, ytau);
    return;
}
```
Figure 4: Solution of the Lotka-Volterra resource (S) – producer (P) – consumer (C) model with time-variable input (signal) and with half of the consumer removed every 10 days - see text for R-code
void F77_SUB(lagderiv)(double *T, int *nr, int *N, double *ytau) {
    static void(*fun)(double, int*, int, double*) = NULL;
    if (fun == NULL)
        fun = (void(*)(double, int*, int, double*))R_GetCCallable("deSolve", "lagderiv");
    fun(*T, nr, *N, ytau);
    return;
}

Here \( T \) is the time at which the value needs to be retrieved, \( nr \) is an integer that defines the number of the state variable or its derivative whose delay we want, \( N \) is the total number of state variables and \( ytau \) will have the result.

We start with an example, a Lotka-Volterra system with delay, that we will implement in Fortran (you will find this example in the package directory \texttt{inst/doc/dynload-de}\texttt{de}, in file \texttt{dede\_lvF.f}).

The R-code would be:

\begin{verbatim}
R> derivs <- function(t, y, parms) {
+    with(as.list(c(y, parms)), {
+        if (t < tau)
+            ytau <- c(1, 1)
+        else
+            ytau <- lagvalue(t - tau, c(1, 2))
+        dN <- f * N - g * N * P
+        dP <- e * g * ytau[1] * ytau[2] - m * P
+        list(c(dN, dP), tau=ytau[1], tau=ytau[2])
+    })
+}
R> yinit <- c(N = 1, P = 1)
R> times <- seq(0, 500)
R> parms <- c(f = 0.1, g = 0.2, e = 0.1, m = 0.1, tau = .2)
R> yout <- dede(y = yinit, times = times, func = derivs, parms = parms)
R> head(yout)
\end{verbatim}

\begin{verbatim}
  time  N   P  tau  tau
[1,]  0 1.000000 1.000000 1.000000 1.000000
[2,]  1 0.911919 0.922822 0.927752 0.937889
[3,]  2 0.844143 0.850251 0.856293 0.864392
[4,]  3 0.792459 0.782398 0.760565 0.731514
[5,]  4 0.753739 0.660748 0.730536 0.672085
[6,]  5 0.725689 0.660748 0.730536 0.672085
\end{verbatim}

In Fortran the code looks like this:

\begin{verbatim}
! file dede\_lfF.f
! Initializer for parameter common block
\end{verbatim}
subroutine initmod(odeparms)
  external odeparms
  double precision parms(5)
  common /myparms/parms

  call odeparms(5, parms)
  return
end

! Derivatives and one output variable
subroutine derivs(neq, t, y, ydot, yout, ip)
  integer neq, ip(*)
  double precision t, y(neq), ydot(neq), yout(*)

  double precision N, P, ytau(2), tlag
  integer nr(2)

  double precision f, g, e, m, tau
  common /myparms/f, g, e, m, tau

  if (ip(1) < 2) call rexit("nout should be at least 2")

  N = y(1)
  P = y(2)
  nr(1) = 0
  nr(2) = 1
  ytau(1) = 1.0
  ytau(2) = 1.0

  tlag = t - tau
  if (tlag .GT. 0.0) call lagvalue(tlag, nr, 2, ytau)

  ydot(1) = f * N - g * N * P
  ydot(2) = e * g * ytau(1) * ytau(2) - m * P

  yout(1) = ytau(1)
  yout(2) = ytau(2)

  return
end

During compilation, we need to also compile the file dedeUtils.c. Assuming that the above Fortran code is in file dede_lvF.f, which is found in the working directory that also contains file dedeUtils.c, the problem is compiled and run as:

  system("R CMD SHLIB dede_lvF.f dedeUtils.c")
  dyn.load(paste("dede_lvF", .Platform$dynlib.ext, sep=""))
yout3 <- dede(yinit, times = times, func = "derivs", parms = parms, dllname = "dede_lxF", initfunc = "initmod", nout = 2)

8.2. Delays implemented in C

We now give the same example in C-code (you will find this in directory inst/doc/dynload-dede/dede_lxF.c).

```c
#include <R.h>
#include <Rinternals.h>
#include <Rdefines.h>
#include <R_ext/Rdynload.h>

static double parms[5];
#define f parms[0]
#define g parms[1]
#define e parms[2]
#define m parms[3]
#define tau parms[4]

/* Interface to dede utility functions in package deSolve */
void lagvalue(double T, int *nr, int N, double *ytau) {
  static void(*fun)(double, int*, int, double*) = NULL;
  if (fun == NULL)
    fun = (void(*)(double, int*, int, double*))R_GetCCallable("deSolve", "lagvalue");
  return fun(T, nr, N, ytau);
}

void lagderiv(double T, int *nr, int N, double *ytau) {
  static void(*fun)(double, int*, int, double*) = NULL;
  if (fun == NULL)
    fun = (void(*)(double, int*, int, double*))R_GetCCallable("deSolve", "lagderiv");
  return fun(T, nr, N, ytau);
}

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

/* Derivatives */
void derivs (int *neq, double *t, double *y, double *ydot, double *yout, int *ip) {
  if (ip[0] < 2) error("nout should be at least 1");
  double N = y[0];
}
9. Difference equations in compiled code

There is one special-purpose solver, triggered with method = "iteration" which can be used in cases where the new values of the state variables are estimated by the user, and need not be found by integration.

This is for instance useful when the model consists of difference equations, or for 1-D models when transport is implemented by an implicit or a semi-implicit method.

An example of a discrete time model, represented by a difference equation is given in the help file of solver ode. It consists of the host-parasitoid model described as from Soetaert and Herman (2009, p283).

We first give the R-code, and how it is solved:

```r
Parasite <- function (t, y, ks) {
  P <- y[1]
  H <- y[2]
  T <- *t - tau;
  if (*t > tau) {
    lagvalue(T, nr, Nout, ytau);
  }
  ydot[0] = f * N - g * N * P;
  ydot[1] = e * g * ytau[0] * ytau[1] - m * P;
  yout[0] = ytau[0];
  yout[1] = ytau[1];
}
```

Assuming this code is in a file called dede_lv.c, which is in the working directory, this file is then compiled and run as:

```sh
system("R CMD SHLIB dede_lv.c")
dyn.load(paste("dede_lv", .Platform$dynlib.ext, sep=""))
yout2 <- dede(yinit, times = times, func = "derivs", pars = parms, dllname = "dede_lv", initfunc = "initmod", nout = 2)
dyn.unload(paste("dede_lv", .Platform$dynlib.ext, sep=""))
```
\[ f \leftarrow A \times P / (ks + H) \]
\[ P_{\text{new}} \leftarrow H \times (1 - \exp(-f)) \]
\[ H_{\text{new}} \leftarrow H \times \exp(rH \times (1 - H) - f) \]

\text{list (c(Pnew, Hnew))}

\}

\rH \leftarrow 2.82 \quad \# \text{ rate of increase}
A \leftarrow 100 \quad \# \text{ attack rate}
ks \leftarrow 15. \quad \# \text{ half-saturation density}

out \leftarrow \text{ode (func = Parasite, y = c(P = 0.5, H = 0.5), times = 0:50,}
\quad \text{parms = ks, method = "iteration"})

Note that the function returns the updated value of the state variables rather than the rate of change (derivative). The method “iteration” does not perform any integration.

The implementation in FORTRAN consists of an initialisation function to pass the parameter values (\text{initparms}) and the “update” function that returns the new values of the state variables (\text{parasite}):
require(deSolve)
rH <- 2.82 # rate of increase
A <- 100    # attack rate
ks <- 15.   # half-saturation density

parms <- c(rH = rH, A = A, ks = ks)
out <- ode(func = "parasite", y = c(P = 0.5, H = 0.5), times = 0:50,
    initfunc = "initparms", dllname = "difference", parms = parms,
    method = "iteration")

10. Final remark

Detailed information about communication between C, FORTRAN and R can be found in R Development Core Team (2009).
Notwithstanding the speed gain when using compiled code, one should not carelessly decide to always resort to this type of modelling.
Because the code needs to be formally compiled and linked to R much of the elegance when using pure R models is lost. Moreover, mistakes are easily made and paid harder in compiled code: often a programming error will terminate R. In addition, these errors may not be simple to trace.
References


Affiliation:

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: karline.soetaert@nioz.nl
URL: http://www.nioz.nl

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
URL: http://www.epa.gov/comptox