R Package **deSolve**, writing code in compiled language

K. Soetaert
Netherlands Institute of Ecology
The Netherlands
R. Woodrow Setzer
National Center for Computational Toxicology
US Environmental Protection Agency

May 25, 2008

1 Introduction

deSolve, the successor of **R** package **odesolve** 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.
- 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 Rmodels and DLLmodels respectively.

Whereas **R models** are easy to implement, they allow simple interactive development, produce highly readible code and access to **R** s 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 **R** s high-level commands, the time gain will be modest. This was demonstrated in Soetaert08, 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 soetaert08, a very simple, 0-D, lotka-volterrra 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.

2 A simple ODE example

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

$\frac{dy1}{dt} = -k1 \cdot y1 + k2 \cdot y2 \cdot y3$
$\frac{dy2}{dt} = k1 \cdot y1 - k2 \cdot y2 \cdot y3 - k3 \cdot y2 \cdot y2$
$\frac{dy3}{dt} = k3 \cdot y2 \cdot y2$

where y1, y2 and y3 are state variables, and k1, k2 and k3 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 = func(t, y, parms,...)
```

where t is the current time point in the integration, y is the current estimate of the variables in the ODE system, and parms is a vector or list containing the parameter values. ... (optional) are any other arguments passed to the function. The return value of *func* should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose second element contains output variables that are required at each point in time.

The **R** implementation of the simple ODE is given below:

```
model<-function(t,Y,parameters)
{

 with (as.list(parameters),{

   dy1 = -k1*Y[1] + k2*Y[2]*Y[3]
   dy3 = k3*Y[2]*Y[2]
   dy2 = -dy1 - dy3

   list(c(dy1,dy2,dy3))
                              })
}
```

The jacobian associated to the above example is:

```
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:

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

2.2 ODE model implementation in C

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**:

```
/* file mymod.c */
#include <R.h>
static double parms[3];
#define k1 parms[0]
#define k2 parms[1]
#define k3 parms[2]

/* initializer  */
void initmod(void (* odeparms)(int *, double *))
{
    int N=3;
    odeparms(&N, parms);
}

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

    yout[0] = y[0]+y[1]+y[2];
}

/* The jacobian matrix */
void jac(int *neq, double *t, double *y, int *ml, int *mu,
           double *pd, int *nrowpd, double *yout, int *ip)
{
  pd[0]               = -k1;
  pd[1]               = k1;
  pd[2]               = 0.0;
  pd[(*nrowpd)]       = k2*y[2];
  pd[(*nrowpd) + 1]   = -k2*y[2] - 2*k3*y[1];
  pd[(*nrowpd) + 2]   = 2*k3*y[1];
  pd[(*nrowpd)*2]     = k2*y[1];
```

```
  pd[2*(*nrowpd) + 1] = -k2 * y[1];
  pd[2*(*nrowpd) + 2] = 0.0;
}
/* END file mymod.c */
```

The implementation in **C** consists of three parts.

- 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.

- Function *derivs* then calculates the values of the derivatives. The derivative function is defined as:

  ```
  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 room is allocated for the output variables ("if (ip[0] <1)"), else an error is passed to R and the integration is stopped.

- In **C**, the call to the function that generates the jacobian is as:

  ```
  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, this is equal to **neq*. In case the jacobian is banded, **nrowpd* will be equal to **mu+*ml+1*. [2]

[1] readers familiar with the source code of the ODEPACK solvers may be surprised to find the two vectors *yout* and *nout* 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 (and as we think the omission of these vectors in the ODEPACK solvers is a design flaw), we have altered the ODEPACK FORTRAN codes to consistently pass these vectors.

[2] readers familiar with the implementation of the FORTRAN code DVODE may notice that this is not the format in which DVODE requires the specification of the Jacobian; this code needs an extra *mu* empty rows. As we have taken the philosophy to make the model specification independent of the integrator that will be used, the facility to use *vode* with a Jacobian specified in compiled code has been toggled off. Use the related code *lsode* instead.

2.3 ODE model implementation in FORTRAN

Models may also be defined in **FORTRAN**.

```
c file mymod.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/k1, k2, k3

          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 mymod.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 aboove, the code in mymod.f and mymod.c must first be compiled.

This can simply be done in R itself, using the *system* command:

```
system("R CMD SHLIB mymod.f")
```

for the **FORTRAN** code or

```
system("R CMD SHLIB mymod.c")
```

for the **C** code

This will create file mymod.dll After loading the DLL, the model can be run:

```
dyn.load("mymod.dll")

parms <- c(k1 = 0.04, k2 = 1e4, k3=3e7)
Y     <- c(y1=1.0,y2=0.0,y3=0.0)
times <- c(0,0.4*10^(0:11) )

out <- ode(Y,times,func="derivs",parms=parms,
           jacfunc="jac",dllname="mymod",
           initfunc="initmod",nout=1,outnames="Sum" )
```

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 *jacfun* and *initfunc* (if present) should also be specified in a compiled language. It is not allowed to mix R-functions and compiled functions

Note also that, when invoking the integrator, we have to specify the number of ordinary output variables, *nout*. This is because the integration routine has to allocate memory to pass these output variables back to **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 **R** may freeze and need to be terminated! Therefore it is advised that one checks in the code wheter nout has been specified correctly. In the **FORTRAN** example above, the statement *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 *nout* to a too large value.

Finally, in order to label the output matrix, the name of the ordinary output variable has to be passed explicitly (*outnames*). The names of the state variables are known through their initial condition (*y*)

3 deSolve integrators that support DLL models

Not all integration routines included in **deSolve** can solve **DLL models**. To date those that can are:

- all solvers of the *lsode* familiy: *lsoda*, *lsode*, *lsodar*, *lsodes*
- *vode*
- *daspk*

For some of these solvers the interface is slightly different (e.g. *daspk*), while in others (*lsodar*, *lsodes*) different functions can be defined. How this is implemented in a compiled language is discussed next.

3.1 DAE models, integrator *daspk*

daspk is the only integrator in the package that solves DAE models. DAEs are specified in implicit form:

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

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 compiled language are also different. For code written in **C**, the calling sequence for *res* must be:

```
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 array that contains the current value of the state variables, *ydot* points to an array that will contain the derivatives, *delta* points to an array that will contain the calculated residuals. *cj* points to a scalar, which is normally proportional to the inverse of the stepsize, 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
```

3.2 the root function from integrator *lsodar*

lsodar is an extended version of integrator *lsoda* that includes a root finding function. This function is spedified via argument *rootfunc*.

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:

```
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:

```
      subroutine myroot(neq, t, y, ng, gout)
      integer :: neq, ng
      double precision :: t, y(neq), gout(ng)

      gout(1) = y(1) - 1.e-4
      gout(2) = y(3) - 1e-2

      return
      end
```

3.3 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:

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

4 final remark

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.