Package ccSolve: solving numerical problems in compiled code.

Karline Soetaert
Royal Netherlands Institute of Sea Research
Yerseke, The Netherlands

Abstract

Package **ccSolve** (Soetaert 2014) generates compiled code to solve numerical problems (differential equations, root solving problems, optimization and least squares problems) in R. It works with solvers from the R-packages **deSolve** (Soetaert, Petzoldt, and Setzer 2010b), **bvpSolve** (Soetaert, Cash, and Mazzia 2010a), **rootSolve** (Soetaert 2009), **deTestSet** (Soetaert, Cash, and Mazzia 2014a), and provides extensions to the functions `optim`, `optimize`, and `uniroot` from R's **stats** package (R Development Core Team 2014) and to function `nls.lm` from the R-package **minpack.lm** (Elzhov, Mullen, Spiess, and Bolker 2013).

Problems specified in compiled code may speed up the solution with a factor up to 50 times over problems specified in R-code. However, typically the speed gain is just a factor two to an order of magnitude, while in certain cases the speed gain may be even negligible.

Before deciding to solve a problem via compiled code, one needs to take into account that the problem also needs to be compiled and this often takes a few seconds. So, the functions from **ccSolve** may not provide a good alternative to R-code for one-time use. However, as the shared object or DLL can be saved and loaded, the compilation needs to be done only once so that **ccSolve** may prove worthwhile for analyses that need to be repeated multiple times.

Keywords: differential equations, root solving, minimization, R .

1. Introduction

The package **ccSolve** (Soetaert 2014), provides an interface to problems written in compiled code for solvers from the R-packages **deSolve** (Soetaert *et al.* 2010b), **bvpSolve** (Soetaert *et al.* 2010a), **rootSolve** (Soetaert 2009), **deTestSet** (Soetaert *et al.* 2014a), **minpack.lm** (Elzhov *et al.* 2013) and some numerical solvers from the base R-package **stats**(R Development Core Team 2014). It is meant to speed up the solution of initial value (IVP) and boundary value problems (BVP) for ordinary differential equations (ODE), differential agebraic equations (DAE), partial differential equations (PDE), of functions that solve for the root of nonlinear equations, and of least-squares and optimization problems.

The idea is to formulate the problem as text strings that are either valid Fortran, F95 or C code, but without specifying the headers and declaration section of the code. The **ccSolve** functions are then used to complete these codes, by adding the required declarations and parts of the codes that perform technical manipulations. The functions also compile the code, and load the DLL or shared object. This resulting object can then be used as argument in the associated solver.

The package comprizes:

- function `compile.ode` to create compiled code for solving initial value ordinary differential equation problems, or for linearly implicit differential algebraic equations, that can be solved with functions from the R-packages **deSolve** (Soetaert *et al.* 2010b) and **deTestSet** (Soetaert *et al.* 2014a).
- funcion `compile.steady` to be used with the steady-state solvers of the package **rootSolve** (Soetaert 2009).
- function `compile.bvp` to create compiled code for boundary value problems, to be used with functions from the R-package **bvpSolve** (Soetaert *et al.* 2010a).
- function `compile.dae` to generate compiled code for differential algebraic initial value problems written in implicit form, to be used with solver `daspk` from **deSolve** or `mebdfi` from **deTestSet**.
- function `compile.multiroot` to compile root finding problems, used with solvers from the R-package **rootSolve**.
- function `compile.nls` and `compile.optim` to generate compiled code for solving non-linear least squares and optimization problems, to be used with extended versions of solvers from the **stats** (R Development Core Team 2014) package.
- function `compile.optimize` and `compile.uniroot` to compile one dimensional optimization and root finding problems, to be used with extended versions of th `optimize` and `uniroot` function from **stats**.

The solver packages (**deSolve** , **bvpSolve** , **rootSolve** , and **deTestSet**) already included the facility to write problems in compiled code, as described in the **deSolve** vignette ('compiledCode') (Soetaert, Petzoldt, and Setzer 2014b). Their solvers have been extended to also accept compiled code generated by **ccSolve** [1].

ccSolve also contains functions `ccoptim`, `ccoptimize`, `ccuniroot` and `ccnls` that extend the `optim`, `optimize`, `uniroot` and `nls` functions to also support problems written in compiled code. To make these extensions, the original formulations from the **stats** package (R Development Core Team 2014), and from the R-packge **minpack.lm** (Elzhov *et al.* 2013) were altered.

Writing compiled code and linking this to R is a rather technical endeavour: it requires problem codes to be written in separate files, according to strict rules, that are then compiled, linked and loaded. In contrast, the new package **ccSolve** allows to define the problem as text strings, in R, and it takes care of the technical aspects. To do so, it relies strongly on the R-package **inline** (Sklyar, Murdoch, Smith, Eddelbuettel, Francois, and Soetaert 2015) that was therefore extended (e.g. to make it more Fortran-Friendly).

[1]you may need to update your version

Depending on the problem, compiled functions may be up to 50 times faster than R-functions, but in some cases the speed gain will be as small as a few percent only. One also needs to consider that the compilation itself will easily take a few seconds.

2. Overview

The R-functions that make compiled code for differential equation, and root-solving problems are called `compile.ode`, `compile.bvp`, `compile.dae`, `compile.steady` and `compile.multiroot`. As an example, the arguments of `compile.ode` and `compile.bvp` are:

```
 args(compile.ode)

function (func, jacfunc = NULL, rootfunc = NULL, eventfunc = NULL,
    parms = NULL, y = NULL, forcings = NULL, outnames = NULL,
    declaration = character(), includes = character(), language = "F95",
    ...)
NULL
```

```
 args(compile.bvp)

function (func, jacfunc = NULL, bound = NULL, jacbound = NULL,
    parms = NULL, yini = NULL, forcings = NULL, outnames = NULL,
    declaration = character(), includes = character(), language = "F95",
    ...)
NULL
```

The functions that make compiled versions of problems to be used with extensions of certain numerical solvers from R-base are similar, i.e. for solving optimization problems:

```
 args(compile.optim)

function (func, jacfunc = NULL, data = NULL, par = NULL, declaration = character(),
    includes = character(), language = "F95", ...)
NULL
```

Here `func`, `jacfunc`, `rootfunc`, `eventfunc`, `bound`, `jacbound` are character vectors that contain the body of the code representing the respective functions. These texts can be written in Fortran, F95 or C, and the compiling functions will expand them, by adding the function or subroutine definition, and the variable declarations, but also (if applicable) code parts that perform initialisation or finalisation.

The arguments `parms`, `forcings` and `data` allow to use the names of the model parameters, forcing functions and data sets, in the codes. The compiling functions will then either create a common block or module (F95) or declare global variables (C) and include code parts to allow the solvers to put the values of the parameters or the data into these structures at the start of the model application (parameters, data) or at each time point (forcings).

Arguments `y` or `yini` (for `compile.bvp`) specify the names of state variables. The compiling functions then add the declarations for the state variables and their derivatives to the code; for a state variable named "state", its derivative is defined as "dstate". Also, code parts are added that map the state variable vector to these names at the start of the derivative function, while at the end of the derivative function, the derivatives are written to the output vector. In a similar way, the functions `compile.optim`, `compile.optimize` and `compile.root` allow to declare the names of the unknowns to be solved for, by adding the argument `par` during compilation.

Argument `outnames` allows to define the names of output variables which are then declared in the code, and their values stored by the solver at each time point.

Arguments `declaration` add extra declarations to the code, which will be pasted after the functions or subroutines general declarations, but before the actual code, while `includes` will be added before the functions.

To date, the `proglang` to choose from is either Fortran, F95 or C.

3. Rootsolving problems

Root solvers try to find the values of `x` for which `f(x)` equals 0.

Different solvers are invoked whether `x` and `f(x)` are one value or a vector.

3.1. One dimensional root finding

Base R contains a function, called `uniroot` that solves for a single root `x, f(x)` within an interval; a simple extension from R-package **rootSolve** , `uniroot.all` finds several single roots within this interval.

Compiled code that specifies such single root-finding problems is generated with function `compile.uniroot`, whose arguments are:

```
 args(compile.uniroot)
```

```
function (func, declaration = character(), includes = character(),
    language = "F95", ...)
NULL
```

The **ccSolve** function `ccuniroot` extends the original C-code from the **stats** package to accept problems specified in compiled code ([2])

As an example, the following problem:

$$f(x) = \frac{1}{cos(1 + x^2)} = 0 \qquad (1)$$

is solved. The implementation and solution in R is straightforward; (the problem is solved 100 times to be sure that the system time is measurable):

[2] I have used the uniroot function from version R2.12, as it is simpler than later functions

```
rootR <- function (x) 1/cos(1+x^2)
print(system.time(
  for (i in 1:100) AA <- uniroot.all(rootR, c(-10, 10))
 ))

  user  system elapsed
  0.51    0.00    0.52
```

In the compiled code version, the problem is written as a string, the function value is put in a double precision number `f`. For Fortran, F95, both `f` and `x` are a single number, all variables are vectors in C.

It is not too difficult to define this problem in F95 and in C and solve it:

```
croot.f95 <- compile.uniroot("f = 1.d0 / cos(1.d0+x*x)")
croot.C   <- compile.uniroot("f[0] = 1.0/cos(1.0+x[0]*x[0]);", language = "C")
print(system.time(
  for (i in 1:100) A2 <- ccuniroot.all(croot.f95, c(-10, 10))
 ))

  user  system elapsed
  0.10    0.00    0.09

print(system.time(
  for (i in 1:100) A3 <- ccuniroot.all(croot.C, c(-10, 10))
 ))

  user  system elapsed
  0.10    0.00    0.09

max(abs(AA-A2))

[1] 0
```

Note the use of double precision numbers in the F95 definition (1.d0); this is necessary to prevent the compiler to use single precision arithmetic.

If we print the F95 or C code, the wrapper written by `compile.uniroot` is clear:

```
code(croot.f95)

 1:
 2:  SUBROUTINE func ( x, f, rpar, ipar )
 3: IMPLICIT none
 4: DOUBLE PRECISION x
 5: DOUBLE PRECISION f
 6: DOUBLE PRECISION rpar(*)
 7: INTEGER ipar(*)
```

```


   f = 1.d0 / cos(1.d0+x*x)

RETURN
END

```

```
code(croot.C)
```

```
#include <R.h>


void func ( double * x, double * f, double * rpar, int * ipar ) {


   f[0] = 1.0/cos(1.0+x[0]*x[0]);

}
```

The returned compiled object can be queried using `ccfunc`:

```
ccfunc(croot.f95, x = 1)
```

```
[1] -2.402998
```

3.2. multiple roots

The R-function that creates the compiled code for multidimensional root-solving problems (where `x` and `f(x)` are a vector, of equal length) is called `compile.multiroot`. It extends the root solving functions of the R-package **rootSolve**: `multiroot` and `multiroot.1D`.

```
args(compile.multiroot)
```

```
function (func, jacfunc = NULL, parms = NULL, x = NULL, declaration = character(),
    includes = character(), language = "F95", ...)
NULL
```

A simple two-equation model

We start by implementing a simple two-equation model that we will solve with function `multiroot` from R-package `rootSolve`.

We look at the arguments of the function `multiroot` first:

```
args(multiroot)
```

```
function (f, start, maxiter = 100, rtol = 1e-06, atol = 1e-08,
    ctol = 1e-08, useFortran = TRUE, positive = FALSE, jacfunc = NULL,
    jactype = "fullint", verbose = FALSE, bandup = 1, banddown = 1,
    parms = NULL, ...)
NULL
```

The function specifying the problem is passed via argument `f`, while the initial guess of the x-values are in `start`. In addition, it is possible to pass a function that returns the jacobian and/or to specify its structure.

The R-code to solve the first problem is:

```
 fun.R <- function(x){
    c(x[1] - 4*x[1]^2 - x[1]*x[2],
      2*x[2] - x[2]^2 + 3*x[1]*x[2] )
  }
 sol <- multiroot(f = fun.R, start = c(1, 1))
 sol

$root
[1] 2.500000e-01 8.961967e-12

$f.root
[1] -1.927643e-08  2.464541e-11

$iter
[1] 7

$estim.precis
[1] 9.650537e-09
```

In the compiled code version, the user must specify the values of `f` based on the inputted values `x`; it is solved using the same function as the R-problem:

```
 fun.f95 <- "
   f(1) = x(1) - 4.d0*x(1)**2. - x(1) *x(2)
   f(2) = 2.d0*x(2) - x(2)**2 + 3.d0*x(1)*x(2)
  "
 cfun.f95 <- compile.multiroot(fun.f95)
 multiroot(f = cfun.f95, start = c(1, 1))

$root
[1] 2.500000e-01 8.961967e-12

$f.root
[1] -1.927643e-08  2.464541e-11
```

```
$iter
[1] 7

$estim.precis
[1] 9.650537e-09
```

[3]

The jacobian can also be specified as a string, and added during compilation. When we solve this problem, the solver needs to be notified that the jacobian is explicitly provided by the user (the default is to estimate it numerically):

```
jacfun.f95 <- "
   df (1, 1) = 1.d0 - 8.d0*x(1) - x(2)
   df (1, 2) = -x(1)
   df (2, 1) = 3.d0*x(2)
   df (2, 2) = 2.d0 - 2.d0*x(2) + 3.d0*x(1)
 "
cfunjac.f95 <- compile.multiroot(func = fun.f95, jacfunc = jacfun.f95)
multiroot(f = cfunjac.f95, start = c(1, 1), jactype = "fullusr")
```

```
$root
[1] 2.500000e-01 8.962379e-12

$f.root
[1] -1.927376e-08  2.464654e-11

$iter
[1] 7

$estim.precis
[1] 9.649203e-09
```

The extended code is:

```
code(cfunjac.f95)
```

```
  1:
  2:  SUBROUTINE func ( n, t, x, f, rpar, ipar )
  3: IMPLICIT none
  4: INTEGER n
  5: DOUBLE PRECISION t
  6: DOUBLE PRECISION x(*)
  7: DOUBLE PRECISION f(*)
  8: DOUBLE PRECISION rpar(*)
```

[3]The compiled function is only twice as fast as the original R-function, so it does not really make sense to do the effort here.

```
INTEGER ipar(*)



 f(1) = x(1) - 4.d0*x(1)**2. - x(1) *x(2)
 f(2) = 2.d0*x(2) - x(2)**2 + 3.d0*x(1)*x(2)


RETURN
END

 SUBROUTINE jacfunc ( n, t, x, ml, mu, df, nrowpd, rpar, ipar )
IMPLICIT none
INTEGER n
DOUBLE PRECISION t
DOUBLE PRECISION x(*)
INTEGER ml
INTEGER mu
DOUBLE PRECISION df(nrowpd,*)
INTEGER nrowpd
DOUBLE PRECISION rpar(*)
INTEGER ipar(*)


 integer ix, jx
 do ix = 1, n
  do jx = 1, n
   df(ix,jx) = 0.d0
  enddo
 enddo

  df (1, 1) = 1.d0 - 8.d0*x(1) - x(2)
  df (1, 2) = -x(1)
  df (2, 1) = 3.d0*x(2)
  df (2, 2) = 2.d0 - 2.d0*x(2) + 3.d0*x(1)


RETURN
END

```

Note the function arguments, `n, t, x, f, rpar, ipar` and the jacobian arguments, `n, t, x, ml, mu, df, nrowpd, rpar, ipar`, which are enforced by the underlying solver codes. Many of these arguments will in general not be used (t, rpar, ipar) [4]. The user must only specify the values of `f` or `df`$=\partial fx/\partial x$ based on the inputted values of `x`. At the start of the

[4]This means that none of these names can be used e.g. as a parameter or an internal variable name.

subroutine, the values of `df` are put to 0.

six equations, using a parameter vector

We now solve for the root of 6 equations, using the names of parameters. Parameters are passed to the rootsolving function via the `parms` argument. If we use the vector also as argument during compilation, then their names will be declared and values passed. [5]:

```
sixeq.f95 <- "
  f(1) = x(1) + x(2)/x(6) + x(3) + a*x(4) - b
  f(2) = a*x(3) + c*x(4) + x(5) + x(6) - d
  f(3) = x(1) + b*x(2) + exp(x(4)) + x(5) + x(6) + e
  f(4) = a*x(3) + x(3)*x(5) - x(2)*x(3) - ff*(x(5)**2)
  f(5) = g*(x(3)**2) - x(4)*x(6)
  f(6) = h*x(1)*x(6) - x(2)*x(5)
 "
parms <- c(a = 2, b = 2, c = 3, d = 4, e = 8, ff = 0.1, g = 8, h = 50)

csixeq <- compile.multiroot(sixeq.f95, parms = parms)      # parms passed durng compilati
multiroot(f = csixeq, start = rep(1, 6), parms = parms)    # parms passed when solving
```

```
$root
[1]  0.2945830 -6.6086178  0.3549273 -0.2963191  7.5801232 -3.4010206

$f.root
[1]  4.955591e-12  1.110223e-13  8.141932e-12 -1.308287e-12
[5]  1.459322e-11  5.432170e-10

$iter
[1] 69

$estim.precis
[1] 9.538785e-11
```

When a parameter vector is passed upon compilation, function `compile.multiroot` will create a common block (Fortran, F95) or a global vector (C), assigning the parameter *names* in the compiled code. In addition, an initialiser function is added to the code, that will put the parameter values, as passed to the solver, in this common block or global variables. A printout of the F95 code shows what this looks like for the current problem; the vignette ”compiledCode” (Soetaert *et al.* 2014b) from the **deSolve** package gives more information to how this works.

```
code(csixeq)
```

```
  1: 
  2:  SUBROUTINE initpar(deparms)
```

[5]Note that we cannot use `f` as a parameter name here, as this is also the name of the function value vector. We called the offending parameter `ff` instead

```
EXTERNAL deparms
DOUBLE PRECISION parms(8)
COMMON / xcbpar / parms
CALL deparms(8, parms)

END

 SUBROUTINE func ( n, t, x, f, rpar, ipar )
IMPLICIT none
INTEGER n
DOUBLE PRECISION t
DOUBLE PRECISION x(*)
DOUBLE PRECISION f(*)
DOUBLE PRECISION rpar(*)
INTEGER ipar(*)


  double precision  a, b, c, d, e, ff, g, h
  common /  xcbpar  /  a, b, c, d, e, ff, g, h



 f(1) = x(1) + x(2)/x(6) + x(3) + a*x(4) - b
 f(2) = a*x(3) + c*x(4) + x(5) + x(6) - d
 f(3) = x(1) + b*x(2) + exp(x(4)) + x(5) + x(6) + e
 f(4) = a*x(3) + x(3)*x(5) - x(2)*x(3) - ff*(x(5)**2)
 f(5) = g*(x(3)**2) - x(4)*x(6)
 f(6) = h*x(1)*x(6) - x(2)*x(5)


RETURN
END

```

variable number of equations

We end this section with a large problem that solves the so-called Rosenbrock equation. We first implement it in R-code:

```
rosenbrock.R <- function(x) {
   f[i.uneven] <- 1 - x[i.uneven]
   f[i.even]   <- 10 *(x[i.even] - x[i.uneven]^2)
   f
 }
n <- 100000
i.uneven <- seq(1, n-1, by = 2)
```

```
i.even <- i.uneven + 1
f <- vector(length = n)
```

Solving this with 100000 equations using a default (full) jacobian takes almost forever, as this problem has a jacobian of size 100000^2, so we will not do this.

The problem is however solved very fast when we specify the special structure of the jacobian which has nonzero values only on the diagonal and immediately below the diagonal (the latter due to the dependence of `f(i)` on `x(i-1)`).

We therefore solve the problem using function `multiroot.1D` from the R-package **rootSolve**, which assumes a banded Jacobian.

Although we will define a problem consisting of $1e^5$ equations, as the R-code uses vectorised calculations, this is very fast:

```
print(system.time(
 AR <- multiroot.1D(f = rosenbrock.R, start = runif(n), nspec = 1))
 )
```

```
   user  system elapsed
   0.12    0.02    0.14
```

The implementation in Fortran 95 consists of two loops; note that "**" denotes the power in Fortran.

```
rosenbrock.f95 <- "
   integer i
   do i  = 1, n-1, 2
    f(i) = 1 - x(i)
   enddo
   do i  = 2, n, 2
    f(i) = 10 *(x(i) - x(i-1)**2)
   enddo
  "
cRosenbrock.f95 <- compile.multiroot(rosenbrock.f95)
```

In C, it is similar:

```
rosenbrock.C <- "
   int i;
   for(i = 0; i < *n-1; i = i+2)
    f[i] = 1 - x[i];
   for(i = 1; i < *n; i = i+2)
    f[i] = 10 *(x[i] - x[i-1]*x[i-1]);
  "
cRosenbrock.C <- compile.multiroot(rosenbrock.C, language = "C")
```

The value of `n` will be known when the model is called.

```
print(system.time(
   A <- multiroot.1D(f = cRosenbrock.f95, start = runif(100000),  nspec = 1))
 )
```

```
   user  system elapsed 
   0.03    0.00    0.03 
```

The solution of this set of equations is 1 for all variables:.

```
range(A$root)
```

```
[1] 1 1
```

3.3. Steady-states of differential equations

Steady-state solvers find the roots or steady-state solutions of differential equations. In R, the functions `stode` and `stodes` from the R-package **rootSolve** do this. In order to generate compiled versions, the function `compile.steady` should be used. It will be discussed in a later section.

4. Least squares

These are methods to find the best-fitting curve through a data set, by adjusting parameter values in a way that it minimises the squared residuals of the model to the data.

R's standard non-linear least-squares method is called `nls`, but as this is implemented in R-code, it was not simple, nor deemed appropriate, to make it suitable for solving compiled code problems. However, an R interface to the Levenberg-Marquardt nonlinear least-squares algorithm found in MINPACK has been provided in the R-package **minpack.lm** (Elzhov *et al.* 2013), so this was used as the basis for extension to compiled code instead. Apart from the Fortran versus R implementation, the method in the latter package by default scales the parameters, which may make it slightly more efficient than R's `nls` method.

The first example of `nls` is used to show the implementation of least squares problems in compiled code.

```
DNase1 <- subset(DNase, Run == 1)
head (DNase1)
```

```
  Run       conc density
1   1 0.04882812   0.017
2   1 0.04882812   0.018
3   1 0.19531250   0.121
4   1 0.19531250   0.124
5   1 0.39062500   0.206
6   1 0.39062500   0.215
```

```
print(system.time(
 for (i in 1:100)
   fm <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                  data = DNase1,
                  start = list(xmid = 0, scal = 1))
 ))
```

```
   user  system elapsed 
   0.49    0.00    0.48 
```

```
summary(fm)
```

```
Formula: density ~ 1/(1 + exp((xmid - log(conc))/scal))

Parameters:
     Estimate Std. Error t value Pr(>|t|)
xmid -0.02883    0.30785  -0.094    0.927
scal  0.45640    0.27143   1.681    0.115

Residual standard error: 0.3158 on 14 degrees of freedom

Number of iterations to convergence: 14 
Achieved convergence tolerance: 1.631e-06
```

Here the parameters are called `xmid` and `scal`, while the data sets, as present in `data.frame` `DNase1`, are called `conc` and `density`.

Compiled code cannot work with the formula notation, so the residuals are specified instead, and put in vector `f`.

To make the compiled code readable, the problem can be compiled such that the parameters to be solved for and the names of the data are known in the model functions.

During compilation, only the names are important; the actual values will be passed during the application. This is done by calling an added subroutine `initdat`, at the start of the application. Here, memory will be allocated for the data, and a solver function (`nlsdat`) is called to copy the data. At the end of the application, the same subroutine is called to free the allocated memory. All of that is taken care of by the **ccSolve** functions.

F95 codes can work with number, vectors or matrices similar to R itself, so this gives the most simple codes:

```
fm.f95 = "f = density - 1.d0/(1.d0 + dexp((xmid - dlog(conc))/scal))"
cfm.f95 <- compile.nls(func = fm.f95, par = c(xmid = 0, scal = 1),
   data = DNase1[,-1])
```

The single string has been expanded to produce a rather complicated program by the `compile.nls` function:

```
code(cfm.f95)
```

```

module modnlsdata
 implicit none
 integer, parameter :: nvar = 2
 double precision, dimension (:), allocatable ::conc, density
end module modnlsdata

subroutine initdat(nlsdat, m)
 use modnlsdata
 external nlsdat
 integer m

 if (allocated(conc)) deallocate(conc)
 if (allocated(density)) deallocate(density)
 if (m <= 0) return
 allocate(conc( m))
 allocate(density( m))
 call nlsdat(1,conc )
 call nlsdat(2,density )

return
end

 SUBROUTINE func ( n, ndat, x, f, rpar, ipar )
USE modnlsdata
IMPLICIT none
INTEGER n
INTEGER ndat
DOUBLE PRECISION x(n)
DOUBLE PRECISION f(ndat)
DOUBLE PRECISION rpar(*)
INTEGER ipar(*)


   double precision  xmid, scal

 xmid = x(1)
 scal = x(2)
 f = density - 1.d0/(1.d0 + dexp((xmid - dlog(conc))/scal))

RETURN
END

```

For C the residuals should be calculated using a loop:

```
cfm.C <- compile.nls(func = '
   int i;
   for (i = 0; i < *ndat; i++)
     f[i] = density[i] - 1.0/(1.0 + exp((xmid - log(conc[i]))/scal));',
   parms = c(xmid = 0, scal = 1),
   data = DNase1[,-1], language = "C")
```

The compiled code problem is faster:

```
 print(system.time(
  for (i in 1:100)
   fm2 <- ccnls(fn = cfm.f95, data = DNase1[,-1],
     par = c(xmid = 0, scal = 1))
  ))

   user  system elapsed
   0.04    0.00    0.05

 summary(fm2)

Parameters:
     Estimate Std. Error t value Pr(>|t|)
xmid -0.02885    0.30786  -0.094    0.927
scal  0.45643    0.27144   1.682    0.115

Residual standard error: 0.3158 on 14 degrees of freedom
Number of iterations to termination: 11
Reason for termination: Relative error in the sum of squares is at most `ftol'.
```

5. optimization problems

In optimization problems, one tries to find values of x for which $f(x)$ reaches either a minimum or a maximum. In contrast to the root solving methods, here the lengths of x and $f(x)$ need not be the same.

The `optim` and `optimize` functions from the R-base **stats** package (R Development Core Team 2014) have been extended to support optimizing problems written in compiled code. The new solver functions are called `ccoptim` and `ccoptimize` and take the same arguments as the `optimize` functions. The compiling functions are called `compile.optim` and `compile.optimize`.

5.1. Data fitting problem

The optimization functions are often used for fitting a model to data. To implement the previous example for use with `optim`, the sum of squared residuals is minimised. The functions to minimize, in R and F95 are:

```
Opt.R <- function(p) {
   with (DNase1[,-1],
    sum((density - 1/(1 + exp((p[1] - log(conc))/p[2])))^2)
   )
 }
print(system.time(for (i in 1:100)
 A <- optim (fn = Opt.R, par = c(xmid = 0, scal = 1),
   method = "CG")))
```

```
  user  system elapsed
  0.62    0.02    0.64
```

```
Opt.f95 = "f = sum((density - 1.d0/(1.d0 + dexp((xmid - dlog(conc))/scal)))**2)"
cOpt.f95 <- compile.optim(func = Opt.f95, par = c(xmid = 0, scal = 1),
   data = DNase1[,-1])
code(cOpt.f95)
```

```
 1:
 2: module modnlsdata
 3:  implicit none
 4:  integer, parameter :: nvar = 2
 5:  integer :: ndata
 6:  double precision, dimension (:), allocatable ::conc, density
 7: end module modnlsdata
 8:
 9: subroutine initdat(nlsdat, m)
10:  use modnlsdata
11:  external nlsdat
12:  integer m
13:
14:  if (m >= 0) ndata = m
15:  if (allocated(conc)) deallocate(conc)
16:  if (allocated(density)) deallocate(density)
17:  if (m <= 0) return
18:  allocate(conc( m))
19:  allocate(density( m))
20:  call nlsdat(1,conc )
21:  call nlsdat(2,density )
22:
23: return
24: end
25:
26:  SUBROUTINE func ( n, x, f, rpar, ipar )
27: USE modnlsdata
28: IMPLICIT none
29: INTEGER n
30: DOUBLE PRECISION x(n)
```

```
DOUBLE PRECISION f
DOUBLE PRECISION rpar(*)
INTEGER ipar(*)


   double precision  xmid, scal

 xmid = x(1)
 scal = x(2)
 f = sum((density - 1.d0/(1.d0 + dexp((xmid - dlog(conc))/scal)))**2)

RETURN
END

```

```
print(system.time(for (i in 1:100)
 AA <- ccoptim (fn = cOpt.f95, par = c(xmid = 0, scal = 1),
   method = "CG", data = DNase1[,-1])))
```

```
   user  system elapsed
   0.07    0.00    0.06
```

```
AA
```

```
$par
       xmid        scal
-0.02882867  0.45639965

$value
[1] 1.396339

$counts
function gradient
      42       17

$convergence
[1] 0

$message
NULL
```

5.2. The Brown problem

There also exist minimzation problem for which the time gain upon using compiled code is not large. An example is the Brown problem, one of the many problems present as a demo in the **optimx** package (Nash and Varadhan 2011):

```
brown.R <- function(p) {
  sum((p[odd]^2)^(p[even]^2 + 1) + (p[even]^2)^(p[odd]^2 + 1))
 }
npar <- 100
p0 <- rnorm(npar, sd = 2)
n <- npar
odd <- seq(1, n, by = 2)
even <- seq(2, n, by = 2)
print(system.time(
   ans.opt <- optim(par = p0, fn = brown.R, method = "BFGS")))
```

```
   user  system elapsed 
   0.45    0.00    0.46 
```

The Fortran 95 version is:

```
brown.f95 <- "integer i
   f = 0.d0
   do i = 1, n-1, 2
      f = f + (x(i)**2)**(x(i+1)**2 + 1.d0) + (x(i+1)**2)**(x(i)**2 + 1.d0)
   enddo
 "
ccbrown <- compile.optim(brown.f95)
print(system.time(
   ans.cc <- ccoptim(par = p0, fn = ccbrown, method = "BFGS")))
```

```
   user  system elapsed 
   0.22    0.00    0.22 
```

6. initial value problems of differential equations

Here we give some typical uses of the function `compile.ode` that creates the compiled code for initial value problems of ordinary differential equations, and of differential algebraic equations written in linear implicit form.

Its argument are:

```
args(compile.ode)
```

```
function (func, jacfunc = NULL, rootfunc = NULL, eventfunc = NULL, 
    parms = NULL, y = NULL, forcings = NULL, outnames = NULL, 
    declaration = character(), includes = character(), language = "F95", 
    ...) 
NULL
```

6.1. Simple ODE initial value problem

The famous Lorenz equations model chaos in the earth's atmosphere. The implementation in R and compiled code is treated in the last chapter.

6.2. A discrete time model

In a difference equation, one specifies the new value of y rather than the derivative.

We implement the host-parasitoid model as in (Soetaert and Herman 2009); its implementation in R is:

```
parms <- c(rH = 2.82, A = 100, ks = 1)
parasite.R <- function (t, y, parms) {
   with (as.list(parms), {
    P <- y[1]
    H <- y[2]
    f <- A * P / (ks +H)
    Pnew <- H* (1-exp(-f))
    Hnew <- H * exp(rH*(1.-H) - f)
    list (c(Pnew, Hnew))
   })
 }
```

In Fortran 95, and using parameter and state variable names:

```
declaration <- "        double precision ff"
parasite.f90 <- "
         ff = A * P / (ks + H)
         dP = H * (1.d0 - exp(-ff))
         dH = H * exp (rH * (1.d0 - H) - ff)
 "
parms <- c(rH = 2.82, A = 100, ks = 15)
yini <- c(P = 0.5, H = 0.5)
cParasite <- compile.ode(func = parasite.f90, parms = parms,
   y = yini, declaration = declaration, language = "Fortran")

system.time(out <- ode (func = parasite.R, y = yini, parms = parms, times = 0:1000,
     method = "iteration"))

  user  system elapsed
  0.05    0.00    0.05

system.time(outc <- ode (func = cParasite, y = yini, parms = parms, times = 0:1000,
      method = "iteration"))

  user  system elapsed
     0       0       0
```

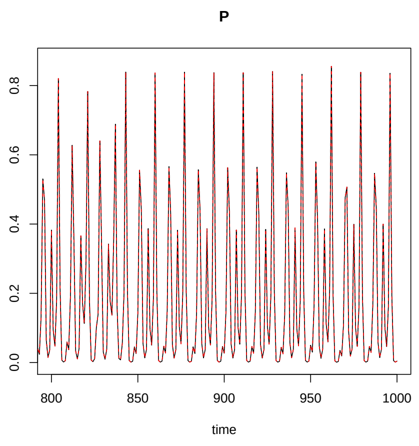

Figure 1: Solution of the iteration problem

```
plot(out, outc, xlim = c(800, 1000), which = "P")
```

6.3. A DAE written in linearly-implicit form

We implement the car axis problem, formulated in (Soetaert *et al.* 2014a), and which was solved in R in (Soetaert, Cash, and F 2012). It is an index 3 DAE which can be written as M*y = f(t,y,p).

Function caraxis.f95 implements the right-hand side, without the heading. The declarations are in a separate string

```
declaration <- "double precision :: Ll, Lr, xb, yb"
caraxis.f95 <- "
     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      + 2.0 * lam2 * (xl-xr) + lam1*xb
     dvl  = (L0-Ll) * yl/Ll      + 2.0 * lam2 * (yl-yr) + lam1*yb - k * g

     dur  = (L0-Lr) * (xr-xb)/Lr - 2.0 * lam2 * (xl-xr)
     dvr  = (L0-Lr) * (yr-yb)/Lr - 2.0 * lam2 * (yl-yr) - k * g

     dlam1 = xb * xl + yb * yl
```

```
      dlam2 = (xl - xr)**2 + (yl - yr)**2. - L * L
  "
```

The 8 parameters and the initial conditions are passed to the `compile.ode` function

```
eps <- 0.01; M <- 10; k <- M * eps^2/2;
L <- 1; L0 <- 0.5; r <- 0.1; w <- 10; g <- 1
parameter <- c(eps = eps, M = M, k = k, L = L, L0 = L0,
                r = r, w = w, g = g)
yini <- c(xl = 0, yl = L0, xr = L, yr = L0,
           ul = -L0/L, vl = 0,
           ur = -L0/L, vr = 0,
           lam1 = 0, lam2 = 0)
ccaraxis <- compile.ode(caraxis.f95, parms = parameter, y = yini,
   declaration = declaration)
```

The first 4 variables are of index 1; the next 4 of index 2, and the last 2 variables are of index 3:

```
index <- c(4, 4, 2)
```

After specifying the mass matrix, and the output times, the model is solved three times with different parameter values.

```
Mass      <- diag(nrow = 10, 1)
Mass[5,5] <- Mass[6,6] <- Mass[7,7] <- Mass[8,8] <- M * eps * eps/2
Mass[9,9] <- Mass[10,10] <- 0
Mass

      [,1] [,2] [,3] [,4]  [,5]  [,6]  [,7]  [,8] [,9] [,10]
 [1,]    1    0    0    0 0e+00 0e+00 0e+00 0e+00    0     0
 [2,]    0    1    0    0 0e+00 0e+00 0e+00 0e+00    0     0
 [3,]    0    0    1    0 0e+00 0e+00 0e+00 0e+00    0     0
 [4,]    0    0    0    1 0e+00 0e+00 0e+00 0e+00    0     0
 [5,]    0    0    0    0 5e-04 0e+00 0e+00 0e+00    0     0
 [6,]    0    0    0    0 0e+00 5e-04 0e+00 0e+00    0     0
 [7,]    0    0    0    0 0e+00 0e+00 5e-04 0e+00    0     0
 [8,]    0    0    0    0 0e+00 0e+00 0e+00 5e-04    0     0
 [9,]    0    0    0    0 0e+00 0e+00 0e+00 0e+00    0     0
[10,]    0    0    0    0 0e+00 0e+00 0e+00 0e+00    0     0
```

```
times <- seq(0, 3, by = 0.01)
outDLL <- daspk(y = yini, mass = Mass, times = times, func = ccaraxis,
                 parms = parameter, nind = index)
p2 <- parameter; p2["r"] <- 0.2
outDLL2 <- daspk(y = yini, mass = Mass, times = times, func = ccaraxis,
                 parms = p2, nind = index)
p2["r"] <- 0.05
outDLL3 <- daspk(y = yini, mass = Mass, times = times, func = ccaraxis,
                 parms = p2, nind = index)
```

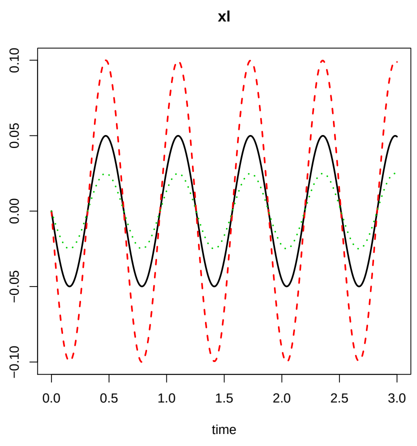

Figure 2: Solution of the linearly-implicit DAE problem

```
plot(outDLL, outDLL2, outDLL3, which = 1, type = "l", lwd = 2)
```

6.4. Steady-state of differential equations

Finding the steady-state of a set of differential equations is somewhat inbetween root solving and differential equation solving. This is because the problems are defined as differential equations, yet they are solved as root solving problems.

To complete the differential equation section, we implement a simple sediment biogeochemical model, which is an example from the **rootSolve** function `stode`.

In addition to the 9 parameters (argument `parms`) that we pass during compilation, we also povide the names of the state variables (`y`) and one output variable (`outnames`).

As we are now dealing with differential equations, we compile the code with `compile.ode`. This function is treated in detail in next section.

We separate the declarations in the code from the body of the code. This is necessary as function `compile.ode` adds lines of code to the program.

```
declaration <- "  double precision :: Min, oxicmin, anoxicmin
 "
cBiogeo.f95 <- "
   Min       = r*OM
   oxicmin   = Min*(O2/(O2+ks))
   anoxicmin = Min*(1-O2/(O2+ks))* SO4/(SO4+ks2)

   dOM  = Flux - oxicmin - anoxicmin
   dO2  = -oxicmin      -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2)
   dSO4 = -0.5*anoxicmin  +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4)
   dHS  = 0.5*anoxicmin   -rox*HS*(O2/(O2+ks)) + D*(BHS-HS)
   SumS = SO4 + HS
  "
```

Note that the state variables (OM, O2, SO4, HS) are called by their name rather than by their position in the state variable vector. In the code the derivatives (called dOM, dO2, dSO4, dHS) are given a value.

The parameter values are:

```
pars <- c(D = 1, Flux = 100, r = 0.1, rox = 1,
           ks = 1, ks2 = 1, BO2 = 100, BSO4 = 10000, BHS = 0)

y <- c(OM = 1, O2 = 1, SO4 = 1, HS = 1)
cBiogeo <- compile.ode(func = cBiogeo.f95, parms = pars, y = y,
   outnames = "SumS", declaration = declaration)
```

When compiling this problem, we passed the parameter vector (`parms`), the name of the output variable (argument `outnames`), and the names of the state variables, via the initial condition vector (argument `y`). Consequently, parameter names, state variable names and ordinary variable names are known in the subroutine. In addition, at each time step, the state variables get their current value, while the derivatives, specified by the user are put in the derivative vector `f` at the end of the subroutine. The derivatives of the state variables are declared as "dOM, dO2, ...". The entire model code is:

```
code(cBiogeo)
```

```

 SUBROUTINE initpar(deparms)
EXTERNAL deparms
DOUBLE PRECISION parms(9)
COMMON / xcbpar / parms
CALL deparms(9, parms)

END

 SUBROUTINE func ( n, t, y, f, rpar, ipar )
IMPLICIT none
INTEGER n
DOUBLE PRECISION t
DOUBLE PRECISION y(*)
DOUBLE PRECISION f(*)
DOUBLE PRECISION rpar(*)
INTEGER ipar(*)


  double precision  D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
  common /  xcbpar  /  D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
   double precision :: Min, oxicmin, anoxicmin

   double precision  OM, O2, SO4, HS
   double precision  dOM, dO2, dSO4, dHS

  double precision SumS


  if (ipar(1) <  1 ) call rexit('nout should be >=  1  ')

 OM = y(1)
 O2 = y(2)
 SO4 = y(3)
 HS = y(4)

  Min       = r*OM
  oxicmin   = Min*(O2/(O2+ks))
  anoxicmin = Min*(1-O2/(O2+ks))* SO4/(SO4+ks2)

  dOM  = Flux - oxicmin - anoxicmin
  dO2  = -oxicmin      -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2)
  dSO4 = -0.5*anoxicmin  +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4)
  dHS  = 0.5*anoxicmin   -rox*HS*(O2/(O2+ks)) + D*(BHS-HS)

  SumS = SO4 + HS

 f(1) = dOM
 f(2) = dO2
 f(3) = dSO4
 f(4) = dHS
 rpar(1) = SumS


RETURN
END

```

The problem is solved by direct iteration; as there may be a -biologically unrealistic- negative solution, positivity is enforced via argument `pos`. When triggering the solver, the parameters, initial conditions and names of the output variables are passed, consistent with the ones used to compile the model.

```
ST <- stode (y = y, func = cBiogeo, parms = pars,
   pos = TRUE, outnames = "SumS", nout = 1)
ST
```

```
$y
[1] 1000.012783    6.825178 9996.587411    3.412589
```

```
$SumS
[1] 10000

attr(,"precis")
[1] 2.549712e+03 5.753884e+01 2.039705e+01 8.527476e+00 2.168616e+00
[6] 1.515096e-01 7.266703e-04 1.664189e-08
attr(,"steady")
[1] TRUE
```

```
 pars["Flux"] <- 200
 ST2 <- stode (y = y, func = cBiogeo, parms = pars,
    pos = TRUE, outnames = "SumS", nout = 1)
 ST2
```

```
$y
[1] 2000.1344467    0.4950409 9949.7524796   50.2475204

$SumS
[1] 10000

attr(,"precis")
[1] 2.574712e+03 4.957463e+01 1.319487e+00 1.732569e-02 2.913095e-06
[6] 1.811884e-13
attr(,"steady")
[1] TRUE
```

The compiled model can also be used to run in dynamic mode:

```
 out <- ode(y = y, func = cBiogeo, times = 0:50, parms = pars,
    outnames = "sumS", nout = 1)
 tail(out, n = 2)
```

```
      time       OM        O2      SO4       HS  sumS
[50,]   49 1985.240 0.5028346 9950.576 49.42405 10000
[51,]   50 1986.657 0.5020824 9950.498 49.50240 10000
```

7. boundary value problems

The R-package **bvpSolve** numerically solves boundary value problems (BVP) of ordinary differential equations (ODE), and of differential algebraic equations. It has two solvers that can be used with problems written in compiled code:

- `bvptwp`, a mono-implicit Runge-Kutta (MIRK) method
- `bvpcol`, a collocation method.

ccSolve function `compile.bvp` makes compiled code from text strings that define the body of the derivative function defining the boundary value problems (`func`) and (optionally) the jacobian function (`jacfunc`), the boundary function (`bound`) and the jacobian of the boundary function (`jacbound`).

Whereas the implementation of BVP problems have much in common with those of IVP in R, one notable exception is that the independent variable is called `x` (denoting space) in BVPs whereas it is `t` (for time) in IVPs.

In both type of problems, the state variables are in a vector called `y`, the function value in a vector `f`, and the jacobian in a vector or matrix called `df`.

Its arguments are:

```
 args(compile.bvp)

function (func, jacfunc = NULL, bound = NULL, jacbound = NULL,
    parms = NULL, yini = NULL, forcings = NULL, outnames = NULL,
    declaration = character(), includes = character(), language = "F95",
    ...)
NULL
```

Here, parms and forcings, if passed will define parameters and forcings, to be used in the code and will set their values upon solving the problem, either at the start (parms) or for each x-value (forcings). This will be done by the solver. By specifying `outnames`, output variables will be defined that can be given a value in the code (by the user).

7.1. The swirling flow III problem

The 'Swirling Flow III' BVP is a test problem in (Ascher, Mattheij, and Russell 1995). The two equations, of second and fourth order are:

$$\begin{aligned} g'' &= (gf' - fg')/eps \\ f'''' &= (-ff''' - gg')/eps \end{aligned}$$

with boundary conditions:

$$g(0) = -1, f(0) = 0, f'(0) = 0, g(1) = 1, f(1) = 0, f'(1) = 0$$

Rewritten as a first-order system, defining $y_1 = g$, $y_3 = f$

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= (y_1 * y_4 - y_3 * y_2)/eps \\ y_3' &= y_4 \\ y_4' &= y_5 \\ y_5' &= y_6 \\ y_6' &= (-y_3 y_6 - y_1 y_2)/eps \end{aligned}$$

The boundary equations are simple, so there is no need to specifiy them as a function (see next chapter for that).

We start by declaring the problem-specific parts, the x-domain and the boundary conditions:

```
require(bvpSolve)
x       <- seq(0, 1, 0.01)
yini <- c(-1, NA, 0, 0, NA, NA)
yend <- c(1 , NA, 0, 0, NA, NA)
```

The parameter `eps` in this BVP defines the stiffness of the problem. We use this example to show the two ways of passing parameters in BVPs

passing parameters via rpar

The first way, not recommended, is to pass parameters via the vector `rpar`.

```
fswirl <- "
   eps = rpar(1)
   f(1) = Y(2)
   f(2) = (Y(1)*Y(4) - Y(3)*Y(2))/eps
   f(3) = Y(4)
          f(4) = Y(5)
          f(5) = Y(6)
   f(6) = (-Y(3)*Y(6) - Y(1)*Y(2))/eps
 "
cswirl <- compile.bvp(func = fswirl, declaration = "double precision:: eps")
print(system.time(sol <- bvptwp(x = x, func = cswirl,
                   yini = yini, yend = yend, eps = 0.01, parms = 0.01)))
```

```
   user  system elapsed
      0       0       0
```

We can also specify the problem in higher-order form, but then it can only be solved with `bvpcol`:

```
fswirl2 <- "
   eps = rpar(1)
   f(1) = (Y(1)*Y(4) - Y(3)*Y(2))/eps
   f(2) = (-Y(3)*Y(6) - Y(1)*Y(2))/eps
 "
cswirl2 <- compile.bvp(func = fswirl2, declaration = "double precision:: eps")
print(system.time(sola <- bvpcol(x = x, func = cswirl2, order = c(2, 4),
                   yini = yini, yend = yend, eps = 0.01, parms = 0.01)))
```

```
   user  system elapsed
      0       0       0
```

passing parameters via the parms argument

It is more convenient to define parameters via the `parms` argument during compilation, as in this case, the parameter is named and either declared, in a common block(Fortran, F95) or as a global variable (C):

```
fswirl3 <- "
   f(1) = Y(2)
   f(2) = (Y(1)*Y(4) - Y(3)*Y(2))/eps
   f(3) = Y(4)
          f(4) = Y(5)
          f(5) = Y(6)
   f(6) = (-Y(3)*Y(6) - Y(1)*Y(2))/eps
 "
cswirl3 <- compile.bvp(fswirl3, parms = c(eps = 0.1))
print(system.time(solb <- bvptwp(x = x, func = cswirl3,
                   yini = yini, yend = yend, eps = 0.01, parms = 0.01)))
```

```
  user  system elapsed 
     0       0       0 
```

We can use this to solve the problem for successively smaller values of `eps`:

```
print(system.time(sol2 <- bvptwp(x = x, func = cswirl,
                   xguess = sol[,1], yguess = t(sol[,-1]),
                   yini = yini, yend = yend, eps=0.001, epsini = 0.01,
                   parms=0.001)))
```

```
  user  system elapsed 
  0.02    0.00    0.02 
```

```
print(system.time(sol3 <- bvptwp(x = x, func = cswirl,
                   xguess = sol2[,1], yguess = t(sol2[,-1]),
                   yini = yini, yend = yend, eps = 0.0001,
                   epsini = 0.001, parms = 0.0001)))
```

```
  user  system elapsed 
  0.05    0.00    0.05 
```

```
print(system.time(sol3b <- bvptwp(x = x, func = cswirl3,
                   xguess = sol2[,1], yguess = t(sol2[,-1]),
                   yini = yini, yend = yend, eps = 0.0001,
                   epsini = 0.001, parms = 0.0001)))
```

```
  user  system elapsed 
  0.03    0.00    0.03 
```

When we use conditioning the problem can be solved for even smaller values of `eps`:

```
print(system.time(sol4 <- bvptwp(atol = 1e-5, x = x, func = cswirl, cond = TRUE,
                   xguess = sol3[,1], yguess = t(sol3[,-1]),
                   yini = yini, yend = yend, eps = 5e-5 , parms=5e-5)))
```

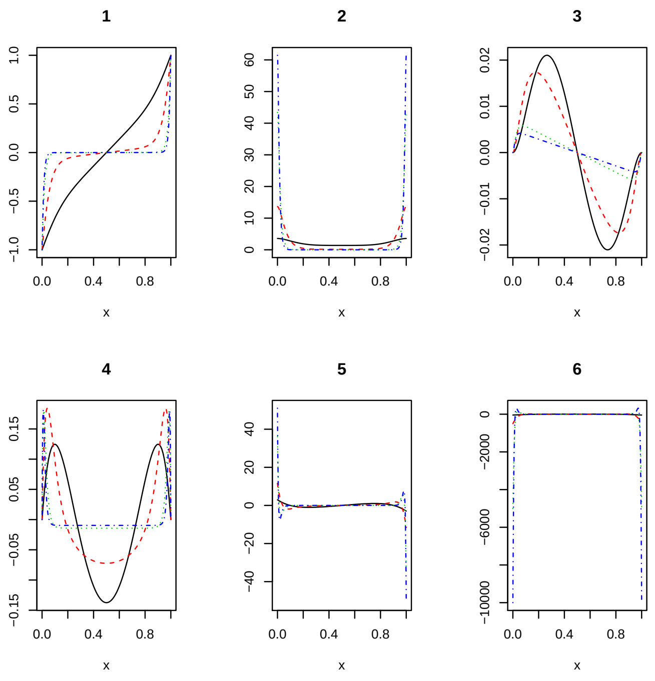

Figure 3: Solution of the BVP swirl problem for different values of eps

```
   user  system elapsed 
    0.1     0.0     0.1 
```

```
plot(sol, sol2, sol3, sol4)
```

7.2. Specifying all functions in compiled code BVPs

We implement the measels problem as from (Ascher *et al.* 1995) and (Soetaert *et al.* 2012). It models the spread of measels in three equations and for one year; it is a boundary value problem as the condition at the end of the year has to be equal to the starting conditions.

Its implementation in R is:

```
require(bvpSolve)
measel.R <- function(t, y, pars)  {
   bet <- 1575*(1+cos(2*pi*t))
   dy1 <- mu-bet*y[1]*y[3]
   dy2 <- bet*y[1]*y[3]-y[2]/lam
   dy3 <- y[2]/lam-y[3]/vv
   dy4 <- 0
   dy5 <- 0
   dy6 <-0
   list(c(dy1, dy2, dy3, dy4, dy5, dy6))
 }
dmeasel.R <- function(t, y, pars) {
   df <- matrix (data = 0, nrow = 6, ncol = 6)
   bet <- 1575*(1+cos(2*pi*t))
   df[1,1] <-  -bet*y[3]
   df[1,3] <-  -bet*y[1]

   df[2,1] <-  bet*y[3]
   df[2,2] <-  -1/lam
   df[2,3] <-  bet*y[1]

   df[3,2] <- 1/lam
   df[3,3] <- -1/vv

   return(df)
 }
bound.R <- function(i, y, pars) {
   if ( i == 1 | i == 4) return(y[1] - y[4])
   if ( i == 2 | i == 5) return(y[2] - y[5])
   if ( i == 3 | i == 6) return(y[3] - y[6])
 }
dbound.R <- function(i, y, pars,vv) {
   if ( i == 1 | i == 4) return(c(1, 0, 0, -1 ,0, 0))
   if ( i == 2 | i == 5) return(c(0, 1, 0, 0, -1, 0))
   if ( i == 3 | i == 6) return(c(0, 0, 1, 0, 0, -1))
 }
```

which specifies the derivative function, the jacobian, the boundary function and the jacobian of the boundary respectively. To solve it, good initial conditions are needed:

```
mu  <- 0.02
lam <- 0.0279
vv  <- 0.1
x <- seq (0, 1, by = 0.01)
yguess <- matrix(ncol = length(x), nrow = 6, data = 1)
rownames(yguess) <- paste("y", 1:6, sep = "")
print(system.time(
   solR <- bvptwp(func = measel.R, jacfunc = dmeasel.R,
     bound = bound.R, jacbound = dbound.R,
     xguess = x, yguess = yguess,
     x=x, leftbc = 3, ncomp = 6,
     nmax = 100000, atol = 1e-4)
 ))

  user  system elapsed
  1.10    0.06    1.16
```

The compiled code implementation is:

```
measel.f95 <- "
    bet = 1575d0*(1.+cos(2*pi*x))
    f(1) = mu - bet*y(1)*y(3)
    f(2) = bet*y(1)*y(3) - y(2)/lam
    f(3) = y(2)/lam-y(3)/vv
    f(4) = 0.d0
    f(5) = 0.d0
    f(6) = 0.d0
 "
dmeasel.f95 <- "
   bet = 1575d0*(1+cos(2*pi*x))
   df(1,1) =  -bet*y(3)
   df(1,3) =  -bet*y(1)

   df(2,1) =  bet*y(3)
   df(2,2) =  -1.d0/lam
   df(2,3) =  bet*y(1)

   df(3,2) = 1.d0/lam
   df(3,3) = -1.d0/vv
 "
bound.f95 <- "
   if ( i == 1 .OR. i == 4) g = (y(1) - y(4))
   if ( i == 2 .OR. i == 5) g = (y(2) - y(5))
   if ( i == 3 .OR. i == 6) g = (y(3) - y(6))
 "
dbound.f95 <- "
   if ( i == 1 .OR. i == 4) THEN
     dg(1) = 1.
     dg(4) = -1.
   else if ( i == 2 .OR. i == 5) then
     dg(2) = 1.
     dg(5) = -1.
   else
     dg(3) = 1.
     dg(6) = -1.
   end if
 "
parms <- c(vv = 0.1, mu = 0.02, lam = 0.0279)
cMeasel <- compile.bvp(func = measel.f95, jacfunc = dmeasel.f95,
   bound = bound.f95, jacbound = dbound.f95, parms  = parms,
   declaration = "double precision, parameter :: pi = 3.141592653589793116d0\n double pre
x <- seq (0, 1, by = 0.01)
yguess <- matrix(ncol = length(x), nrow = 6, data = 1)
rownames(yguess) <- paste("y", 1:6, sep = "")
```

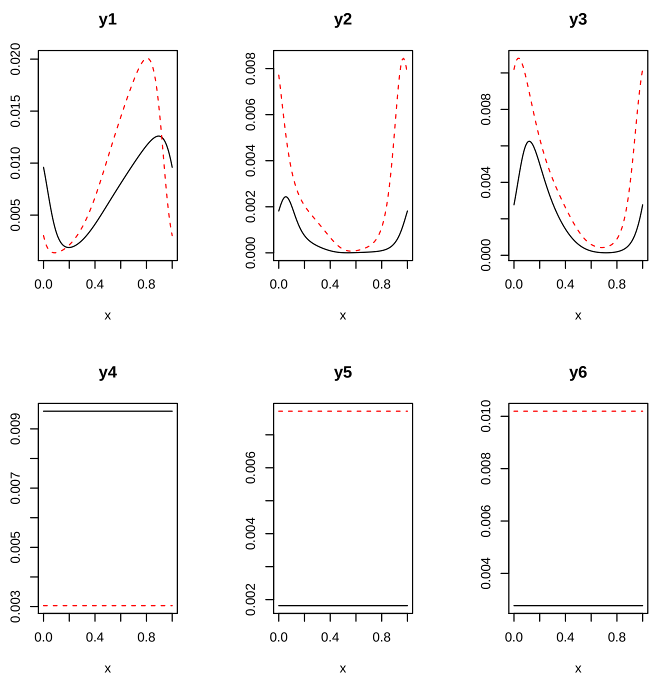

Figure 4: Two solutions of the measel BVP problem

```
print(system.time(
   sol1 <- bvptwp(func = cMeasel,
     xguess = x, yguess = yguess,
     x = x, leftbc = 3, parms = parms, ncomp = 6,
     nmax = 100000, atol = 1e-8)
 ))
```

```
  user  system elapsed
  0.05    0.03    0.08
```

```
print(system.time(
   sol2 <- bvptwp(func = cMeasel,
     xguess = x, yguess = yguess,
     x=x, leftbc = 3, parms = parms * c(1, 2, 2) , ncomp = 6,
     nmax = 100000, atol = 1e-8)
 ))
```

```
  user  system elapsed
  0.08    0.03    0.11
```

```
plot(sol1, sol2)
```

8. Benchmarking

This is a quick test of where the time gain using compiled code is achieved. It appears that there is lots to be gained by having everything in compiled code. Compared to pure R compiled codes can be 20 to even 100 times faster - however, it is also possible that the gain is only a few percent. This a.o. depends on how many times a function is entered and how efficiently the R-code is written. Using compiled code from a call within R may be tens of % to twice faster than in pure R; compared to all-compiled this is still 10 to 20 times slower.

Here is how I tested several options, using the chaos differential equation model:

```
require(deSolve)
chaos.R <- function(t, state, parameters) {
     list(
     c(-8/3 * state[1] + state[2] * state[3],
       -10 * (state[2] - state[3]),
       -state[1] * state[2] + 28 * state[2] - state[3]))
 }
state <- c(xx = 1, yy = 1, zz = 1)
times <- seq(0, 200, 0.01)
print(system.time(
   out   <- vode(state, times, chaos.R, 0)
 ))

  user  system elapsed
   0.4     0.0     0.4

# -------------------------------full compiled code ------------------------
chaos.f95 <- "
     f(1)     = -8.d0/3 * y(1) + y(2) * y(3)
     f(2)     = -10.d0 * (y(2) - y(3))
     f(3)     = -y(1) * y(2) + 28d0 * y(2) - y(3)
 "
cChaos <- compile.ode(chaos.f95)
print(system.time(
   cout   <- vode(state, times, func = cChaos, parms = 0)
 ))

  user  system elapsed
  0.01    0.00    0.01

# ---------------------- calling compiled code in R ------------------------

rchaos <- function(t, state, parameters) {
    list(cChaos$func(3, t, state, f = 1:3, 1, 1)$f)
 }
print(system.time(
   cout2  <- vode(state, times, func = rchaos, parms = 0)
 ))
```

```
  user  system elapsed 
   0.5     0.0     0.5 
```

```
# ---------------------- bitwise compilation in R ------------------------
require(compiler)
bchaos <- cmpfun(chaos.R)
print(system.time(
   cout3  <- vode(state, times, func = bchaos, parms = 0)
 ))
```

```
  user  system elapsed 
  0.41    0.00    0.41 
```

9. Passing data

There are several ways to pass data to the compiled code.

- All subroutines in compiled code have the arguments `rpar` and `ipar`, a double precision and integer vector, that are passed with arguments of the same name when calling the solver. The elements in these vectors are unnamed, and can be used for input. For differential equation solvers, they are also used to contain the output variables (`compile.ode`, `compile.bvp`, `compile.dae`. See vignette ('compiledCode') (Soetaert *et al.* 2014b)

- `parms` is to contain the values of named parameters, whose length is known during compilation. They are declared in a common block (Fortran) or as global variables (C) and their value is set at the start of the solution procedure, as passed with argument `parms`, which is a (named) vector or list. Parameters are not supposed to be changed. They are not implemented for the minmization, uniroot and nonlinear lest squares methods (the name `parms` is too close to these function's argument `par` which refers to the variables to be solved for).

- forcings are only used in certain differential equation models. Their implementation is akin to the parameter implementation (common block or global variable). However, their values are updated by the solver at every time (or spatial, for BVP) step, by interpolating a given data set.

- `data`. This is to contain data, in `matrix` or `data.frame` format, to which a model needs to be fitted. During compilation, the names of the data columns are used to set variable names in the code. The data variables are declared in a module (Fortran) or as global variables (C). The length is not necessarily known at compile time, so memory is allocated and their values are set at the start of the simulation. Data are not meant to be changed in the compiled code.

10. Finally

To save time it can be a good idea to save the compiled object and load it for later use. To allow that, the package **inline** has been extended with two functions: `writeDynLib` and `readDynLib` to save and load the objects.

Note that it is more robust to put the compiled code in a package instead. To obtain the complete compiled code, there are several options:

- **inline**-function `code` will print the code to the screen; when setting the argument `linenumbers = FALSE`, this can be copy-pasted.
- The code of a compiled object x can be extracted (rather than printed), using: `x@code`. For the above example, the compiled code was called `cChaos`. The code can be extracted and written to object `x` as: `x <- cChaos[[1]]@code`

  To get rid of the new-line character and write to a file:

  ```
  write (strsplit(x, "\n")[[1]], file = "fn")}
  ```

One final warning: it is very easy to produce code that makes R crash! It is the responsibility of the package users to write code that is safe to use.

References

Ascher U, Mattheij R, Russell R (1995). *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. Philadelphia, PA.

Elzhov TV, Mullen KM, Spiess AN, Bolker B (2013). *minpack.lm: R interface to the Levenberg-Marquardt nonlinear least-squares algorithm found in MINPACK, plus support for bounds*. R package version 1.1-8, URL `http://CRAN.R-project.org/package=minpack.lm`.

Nash JC, Varadhan R (2011). "Unifying Optimization Algorithms to Aid Software System Users: optimx for R." *Journal of Statistical Software*, **43**(9), 1–14. URL `http://www.jstatsoft.org/v43/i09/`.

R Development Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

Sklyar O, Murdoch D, Smith M, Eddelbuettel D, Francois R, Soetaert K (2015). *inline: Functions to Inline C, C++, Fortran Function Calls from R*. R package version 0.3.14.

Soetaert K (2009). *rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations*. R package version 1.6.

Soetaert K (2014). *ccSolve: Solving numerical problems in compiled code*. R package version 0.01.

Soetaert K, Cash J, F M (2012). *Solving Differential Equations in R*. Springer. ISBN 978-3-642-28070-2.

Soetaert K, Cash J, Mazzia F (2010a). *bvpSolve: solvers for boundary value problems of ordinary differential equations*. R package version 1.2.

Soetaert K, Cash J, Mazzia F (2014a). *deTestSet: Testset for differential equations*. R package version 1.1.1.

Soetaert K, Herman PMJ (2009). *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform*. Springer. ISBN 978-1-4020-8623-6.

Soetaert K, Petzoldt T, Setzer RW (2010b). "Solving Differential Equations in R: Package deSolve." *Journal of Statistical Software*, **33**(9), 1–25. ISSN 1548-7660. URL `http://www.jstatsoft.org/v33/i09`.

Soetaert K, Petzoldt T, Setzer RW (2014b). *R Package deSolve, Writing Code in Compiled Languages*. DeSolve vignette.

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

Table 1: Summary of the compiled function interfaces; in *italics* is the variable that needs to be specified; `i, n, nroot, ndat, ldfjac, ipar, ires` are integers; the rest are doubles.

R-Function	Subroutine declaration	solvers

`compile.ode`, `compile.steady`	func(n, t, y(*), *f(*)*, rpar(*), ipar(*))
	jacfunc(n, t, y(*), *df(n, *)*, rpar(*), ipar(*))
	rootfunc(n, t, y(*), nroot, *root(*)*, rpar(*), ipar(*))
	eventfunc(n, t, *y(*)*)
		ode, ode.1D, ...
		steady, steady.1D,
		runsteady

`compile.bvp`	func(n, x, y(*), *f(*)*, rpar(*), ipar(*))
	jacfunc(n, x, y(n, *), *df(*)*, rpar(*), ipar(*))
	bound(i, n, y(*), *g(*)*, rpar(*), ipar(*))
	jacbound(i, n, y(*), *dg(*)*, rpar(*), ipar(*))
		bvpcol,
		bvptwp

`compile.dae`	res(t, y(*), dy(*), cj, *r(*)*, ires, rpar(*), ipar(*))
	rootfunc(n, t, y(*), nroot, *root(*)*, rpar(*), ipar(*))
	eventfunc(n, t, *y(*)*)
		daspk
		mebdfi

`compile.multiroot`	func(n, t, y(*), *f(*)*, rpar(*), ipar(*))
	jacfunc(n, t, y(*), *df(*)*, rpar(*), ipar(*))
		multiroot
		multiroot.1D

`compile.nls`	func(n, ndat, x(n), *f(ndat)*, rpar(*), ipar(*)) jacfunc(n, ndat, ldfjac, x(*), *df(ldfjac, *)*, rpar(*), ipar(*))	`ccnls`

`compile.optim`	func(n, x(n), *f*, rpar(*), ipar(*))
	jacfunc(n, x(n), *df(n,*)*, rpar(*), ipar(*))
		`ccoptim`

`compile.optimize`, `compile.uniroot`	func(x, *f*, rpar(*), ipar(*))	`ccoptimize`
		`ccuniroot`,
		`ccuniroot.all`

`compile.integrate`	func(n, x(n), *f(n)*, rpar(*), ipar(*))	`ccintegrate`