# 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 in R, more specifically differential equations, root solving problems, optimization and least squares problems. 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-base** (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, often the speed gain is just a factor two to an order of magnitude, while in certain cases the speed gain may be even negligible.

The problem also needs to be compiled before it can be solved; this often takes a few seconds; and this needs to be taken into account before deciding to solve via compiled code. So, these functions may not provide a good alternative to R-code for one-time use.

However, there are functions to save and load compiled problems, so compilation needs to be done only once and **ccSolve** may prove worthwhile for certain analyses that need to be repeated multiple times.

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

## 1. Introduction

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) and **minpack.lm** (Elzhov *et al.* 2013). It is meant to speed up the solution of intial 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 (multiple) or single nonlinear equations, and of least-squares and optimization problems.

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

the associated solver.

The package comprizes:

- function `compile.ode` to create compiled code of 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), **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, from the R-package **rootSolve**.
- function `compile.optim` and `compile.uniroot` to compile one dimensional optimization and root finding problems.

The solver packages (**deSolve** , **bvpSolve** , **rootSolve** , and **deTestSet**) already include the facility to write problems in compiled code; they have been extended to also accept compiled code generated by **ccSolve** [1].

The best description of how R solvers can be used in conjunction with compiled code is in the **deSolve** vignette ('compiledCode') (Soetaert, Petzoldt, and Setzer 2014b). it is clear from this vignette that this is a rather technical endeavour; it requires problem codes to be written in separate files and obeying 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 the user has to go throught when formulating the code as compiled problems.

**ccSolve** also contains functions `ccoptim`, `ccoptimize`, `ccuniroot` and `ccnls` that extend the `optim`, `uniroot` and `nls` functions to also support problems written in compiled code. To make these extensions, the original formulations from the **R-base** software and from the R-packge **minpack.lm** were altered.

Depending on the problem itself, 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 factor two. 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.bvp` are:

[1] so you may need to update your version

```
 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 to be used with extensions of certain numerical solvers from te R-base software 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 that represents the respective functions as used in the corresponding R-codes. These texts can be written in Fortran, F95 or C, and the functions will expand them, e.g. by adding the function or subroutine definitions, adding the declarations and code parts that perform some initialisation or finalisation.

The arguments `parms` and `forcings` allow to use the names of the model parameters and forcing functions in the codes. The compiling functions will then either create a common block (Fortran) or global variables (C) and include the parts of the code that allow the solvers to put the values of the parameters into these objects at the start of the model run (parameters) 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 [2] to the code; it adds code parts that map the state variable vector to these names, at the start of the code, while it ensures that the derivatives are written to the derivative vector at the end of the code.

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 `language` 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.

[2] for a state variable names "state", its derivative is defined as "dstate"

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

Single roots (`x` is one value) in compiled code are generated with function `compile.uniroot`, to be solved with `uniroot` from **R-base** or `uniroot.all` from **rootSolve**.

To compile problems to be used with the steady-state solvers, (that find the roots or steady-state solutions of differential equations), i.e. `stode` and `stodes` the function `compile.steady` should be used.

Functions `compile.multiroot` and `compile.uniroot` are called as:

```
 args(compile.multiroot)

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

 args(compile.uniroot)

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

### 3.1. The root of one function

```
 f <- function (x) 1/cos(1+x^2)
 cf <- compile.uniroot("f = 1.d0/cos(1.d0+x*x)")
 cf$func(1,f=1,1,1)$f

[1] -2.402998

 print(system.time(
   for (i in 1:100) AA <- uniroot.all(f, c(-10, 10))
  ))

   user  system elapsed
   1.65    0.00    1.65

 print(system.time(
   for (i in 1:100) A2 <- ccuniroot.all(cf, c(-10, 10))
  ))

   user  system elapsed
   0.28    0.00    0.28

 AA-A2
```

```
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[34] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

### 3.2. 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 for our 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, we write the problem as strings, the function values are put in a vector `f`. When using fortran, it is wise to write constants in their double precision notation, e.g. 3 becomes 3.d0. This ensures that the computation will be done in double precision.

```
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)
 "
```

The compiled function `cfun` contains the entire code that represents the problem; we can print it using `printCode`.

```
cfun <- compile.multiroot(fun.f95)
printCode(cfun)
```

```
Program source:
  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(*)
  9: INTEGER ipar(*)
 10:
 11:
 12:  f(1) = x(1) - 4.d0*x(1)**2. - x(1) *x(2)
 13:  f(2) = 2.d0*x(2) - x(2)**2 + 3.d0*x(1)*x(2)
 14:
 15:
 16: RETURN
 17: END
 18:
```

Note the function arguments, which are `n, t, x, f, rpar, ipar` [3].

Many of these arguments will not be used (n, t, rpar, ipar). The user must specify the values of `f` based on the inputted values `x`.

The problem is solved as for the R-problem:

```
multiroot(f = cfun, start = c(1, 1))
```

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

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

$iter
```

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

```
[1] 7

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

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.

```
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)
 "
cfun <- compile.multiroot(func = fun.f95, jacfunc = jacfun.f95)
printCode(cfun)

Program source:
  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(*)
  9: INTEGER ipar(*)
 10:
 11:
 12:  f(1) = x(1) - 4.d0*x(1)**2. - x(1) *x(2)
 13:  f(2) = 2.d0*x(2) - x(2)**2 + 3.d0*x(1)*x(2)
 14:
 15:
 16: RETURN
 17: END
 18:
 19:  SUBROUTINE jacfunc ( n, t, x, ml, mu, df, nrowpd, rpar, ipar )
 20: IMPLICIT none
 21: INTEGER n
 22: DOUBLE PRECISION t
 23: DOUBLE PRECISION x(*)
 24: INTEGER ml
 25: INTEGER mu
 26: DOUBLE PRECISION df(nrowpd,*)
 27: INTEGER nrowpd
 28: DOUBLE PRECISION rpar(*)
 29: INTEGER ipar(*)
```

```
 30:
 31:  integer ix, jx
 32:  do ix = 1, n
 33:   do jx = 1, n
 34:  df(ix,jx) = 0.d0
 35:  enddo
 36:  enddo
 37:
 38:   df (1, 1) = 1.d0 - 8.d0*x(1) - x(2)
 39:   df (1, 2) = -x(1)
 40:   df (2, 1) = 3.d0*x(2)
 41:   df (2, 2) = 2.d0 - 2.d0*x(2) + 3.d0*x(1)
 42:
 43:
 44: RETURN
 45: END
 46:
```

```
multiroot(f = cfun, 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
```

Subroutine jacfunc has as arguments: ( n, t, x, ml, mu, df, nrowpd, rpar, ipar ), and the user has to specify `df` ($=\partial fx/\partial x$) based on the inputted `x`; at the start of the subroutine, the jacobian matrix is put to 0.

### 3.3. six equations, using a parameter vector

We now solve for the root of 6 equations, using parameters by their names. They are passed to the rootsolving function via the `parms` argument.

```
sixeq <- "
  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)
 "
pars <- c(a = 2, b = 2, c = 3, d = 4, e = 8, ff = 0.1, g = 8, h = 50)
```

As the parameter vector is passed when compiling the function, its names are known in the compiled function. Function `compile.multiroot` creates the common block (fortran) or global vector (C) and assigns the parameter names in the compiled code. 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.

```
csixeq <- compile.multiroot(sixeq, parms = pars)
multiroot(f = csixeq, start = rep(1, 6), parms = pars)
```

```
$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
```

A printout of the fortran code shows how the parameters are declared and initialised in the function that is generated with `compile.multiroot`. The vignette "compiledCode" (Soetaert *et al.* 2014b) from the **deSolve** package gives more information to how this works.

```
printCode(csixeq)
```

```
Program source:
  1:
  2:  SUBROUTINE initpar(deparms)
  3: EXTERNAL deparms
  4: DOUBLE PRECISION parms(8)
  5: COMMON / xcbpar / parms
  6: CALL deparms(8, parms)
  7:
  8: END
  9:
 10:  SUBROUTINE func ( n, t, x, f, rpar, ipar )
 11: IMPLICIT none
 12: INTEGER n
 13: DOUBLE PRECISION t
 14: DOUBLE PRECISION x(*)
```

```
15: DOUBLE PRECISION f(*)
16: DOUBLE PRECISION rpar(*)
17: INTEGER ipar(*)
18:
19:   double precision  a, b, c, d, e, ff, g, h
20:   common /  xcbpar  /  a, b, c, d, e, ff, g, h
21:
22:
23:
24:  f(1) = x(1) + x(2)/x(6) + x(3) + a*x(4) - b
25:  f(2) = a*x(3) + c*x(4) + x(5) + x(6) - d
26:  f(3) = x(1) + b*x(2) + exp(x(4)) + x(5) + x(6) + e
27:  f(4) = a*x(3) + x(3)*x(5) - x(2)*x(3) - ff*(x(5)**2)
28:  f(5) = g*(x(3)**2) - x(4)*x(6)
29:  f(6) = h*x(1)*x(6) - x(2)*x(5)
30:
31:
32: RETURN
33: END
34:
```

### 3.4. variable number of equations, with and without jacobian

We now go to 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 and a 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 that the jacobian is banded. This is so, as the Jacobian has non-zero values below the diagonal, due to the dependence of f(i) on x(i-1).

We use function `multiroot.1D`, that assumes a banded Jacobian.

We will solve the model for 1e5 equations; as the R-code uses vectorised calculations, even though it is an interpreted code, it is very fast;

```
print(system.time(
```

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

```
   user  system elapsed
   0.27    0.05    0.33
```

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 <- 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]);
  "
 cRosenbrockC <- 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, start = runif(100000),  nspec = 1))
  )
```

```
   user  system elapsed
   0.08    0.02    0.10
```

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

```
 head(A$root)
```

```
[1] 1 1 1 1 1 1
```

# 4. optimization problems

Optimization problems are usually small problems, involving few equations and few iterations, or they are efficiently vectorised in R, so putting them in compiled code will not give large speed gains.

The `optim` function has been extended so that it also supports minimizing problems written in compiled code. The new function is called `ccoptim` and takes the same arguments as `optim`.

```
Nonlin.R <- function (x, ...) {
   sum(-4*x[1:(n-1)]) + 3*(n-1) + sum((x[1:(n-1)]^2 + x[n]^2)^2)
 }
n <- 500
ini <- runif( n)
print(system.time(A <- optim (fn = Nonlin.R, par = ini,
   method = "CG", control = list(maxit = 1000))))
```

```
  user  system elapsed
  5.35    0.00    5.37
```

```
Nonlin.f95 <- "
   integer i
   f = 0.d0
   do i = 1, n-1
      f = f - 4.d0*x(i) + 3.d0 + (x(i)*x(i) + x(n)*x(n))**2.
   enddo
 "
cNonlin <- compile.optim(Nonlin.f95)
print(system.time(AA <- ccoptim (fn = cNonlin, par = ini,
   method = "CG", control = list(maxit = 1000))))
```

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

There is less efficiency gain when the model is solved with the BFGS method:

```
print(system.time(A <- optim (fn = Nonlin.R, par = ini,
   method = "BFGS")))
```

```
  user  system elapsed
  0.14    0.00    0.14
```

```
print(system.time(AA <- ccoptim (fn = cNonlin, par = ini,
   method = "BFGS")))
```

```
  user  system elapsed
  0.03    0.00    0.04
```

```
max(abs(A$par-AA$par))
```

```
[1] 1.048742e-09
```

### 4.1. The Brown problem

Here is another minimzation problem:

```
brown.f <- 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.f, method = "BFGS")))
```

```
   user  system elapsed
   1.01    0.00    1.03
```

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
   0.55    0.00    0.55
```

## 5. Least squares

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

```
DNase1 <- subset(DNase, Run == 1)
print(system.time(
 for (i in 1:100)
 fm2DNase1 <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                  data = DNase1,
                  start = list(xmid = 0, scal = 1))
 ))
```

```
   user  system elapsed
   1.31    0.00    1.32
```

```
 summary(fm2DNase1)

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
```

The problme is compiled such that the compiler knows the names of the parameters to be solved for and the names of the data; the values of parms and data can differ during the actual application; the ordering of parameters and datacolumns should stay the same.

F95 works with vectors

```
 head (DNase1 [, -1])      # names conc, density

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

```
 f1 = "f = density - 1.0/(1.d0 + dexp((xmid - dlog(conc))/scal))"
 ccDNase <- compile.nls(func = f1, par = c(xmid = 0, scal = 1), data = DNase1[,-1])
 printCode(ccDNase)

Program source:
  1:
  2: module modnlsdata
  3:  implicit none
  4:  integer, parameter :: nvar = 2
  5:  double precision, dimension (:), allocatable ::conc, density
  6: end module modnlsdata
  7:
  8: subroutine initdat(nlsdat, m)
  9:  use modnlsdata
```

```
10:  external nlsdat
11:  integer m
12:
13:  if (allocated(conc)) deallocate(conc)
14:  allocate(conc( m))
15:  call nlsdat(1,conc )
16:  if (allocated(density)) deallocate(density)
17:  allocate(density( m))
18:  call nlsdat(2,density )
19:
20: return
21: end
22:
23:  SUBROUTINE func ( n, ndat, x, f, rpar, ipar )
24: USE modnlsdata
25: IMPLICIT none
26: INTEGER n
27: INTEGER ndat
28: DOUBLE PRECISION x(n)
29: DOUBLE PRECISION f(ndat)
30: DOUBLE PRECISION rpar(*)
31: INTEGER ipar(*)
32:
33:         double precision  xmid, scal
34:
35:         xmid = x(1)
36:         scal = x(2)
37:  f = density - 1.0/(1.d0 + dexp((xmid - dlog(conc))/scal))
38:
39: RETURN
40: END
41:
```

For C you need to write a loop:

```
ccDNase.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")

print(system.time(
 for (i in 1:100)
 fm2DNase2 <- ccnls(fn = ccDNase, data = DNase1[,-1],
   par = c(xmid = 0, scal = 1))
 ))
```

```
   user  system elapsed
   0.07    0.00    0.08
```

```
 summary(fm2DNase2)
```

```
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'.
```

Now it is straighforward to use the fast model version to fit all the Runs:

```
 for ( i in 1:11)
    print(ccnls(fn = ccDNase, par = c(xmid = 0, scal = 1),
      data = subset(DNase, Run == i)[, -1])$par)
```

```
      xmid       scal
-0.0288509  0.4564344
      xmid       scal
-0.1121940  0.4161821
      xmid       scal
-0.2301132  0.4654634
      xmid       scal
-0.0124683  0.4423296
       xmid        scal
-0.09223348  0.44735342
      xmid       scal
-0.2505654  0.5007696
      xmid       scal
-0.2565614  0.5142977
       xmid        scal
-0.09549687  0.45824798
      xmid       scal
-0.1474211  0.4723854
      xmid       scal
-0.2346889  0.4904728
      xmid       scal
-0.1871690  0.4893333
```

# 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. It is also to be used to find the steady-state (root) of differential equations.

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 problm

The famous Lorenz equations model chaos in the earth's atmosphere One possible implementation in R would be:

```
 require(deSolve)
 chaos <- function(t, state, parameters) {
    with(as.list(c(state)), {

      dxx     <- -8/3 * xx + yy * zz
      dyy     <- -10 * (yy - zz)
      dzz     <- -xx * yy + 28 * yy - zz

      list(c(dxx, dyy, dzz))
    })
  }
 state <- c(xx = 1, yy = 1, zz = 1)
 times <- seq(0, 100, 0.01)
 print(system.time(
    out   <- vode(state, times, chaos, 0)
  ))
```

```
   user  system elapsed
   1.41    0.00    1.42
```

In fortran we write:

```
 chaos.f95 <- "
      dxx     = -8.d0/3 * xx + yy * zz
      dyy     = -10.d0 * (yy - zz)
```

```
     dzz     = -xx * yy + 28d0 * yy - zz
 "
cChaos <- compile.ode(chaos.f95, y = state)
print(system.time(
   cout   <- vode(state, times, func = cChaos, parms = 0)
 ))
```

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

We can also implement it in C, now passing parameter values, but not the state variable names:

```
parms <- c(a = -8.0/3, b = -10.0, c = 28.0)
chaos.C <- "
    f[0] = a * y[0] + y[1]*y[2];
    f[1] = b*(y[1]-y[2]);
    f[2] = -y[0]*y[1] + c * y[1] - y[2];
 "
parms <- c(a = -8.0/3, b = -10.0, c = 28.0)
cChaos2 <- compile.ode(chaos.C, language = "C", parms = parms)
print(system.time(
   cout2  <- vode(state, times, func = cChaos2, parms = parms)
 ))
```

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

They all give (nearly) the same output:

```
plot(out, cout, cout2)
plot(out[,"xx"], out[,"yy"], type = "l", main = "Lorenz butterfly",
   xlab = "x", ylab = "y")
```

## 6.2. A discrete time model

In a difference equation, one spciefies 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)
```

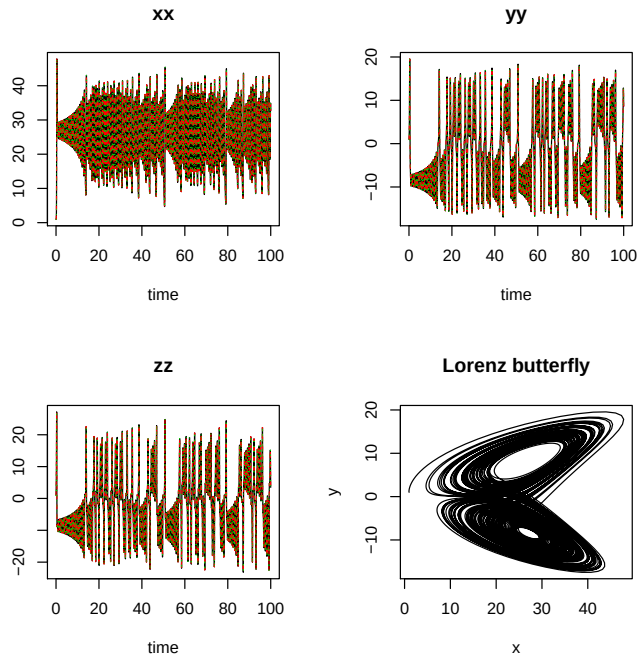


Figure 1: Solution of the chaos problem

```
    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)

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

  user  system elapsed
  0.11    0.00    0.11
```

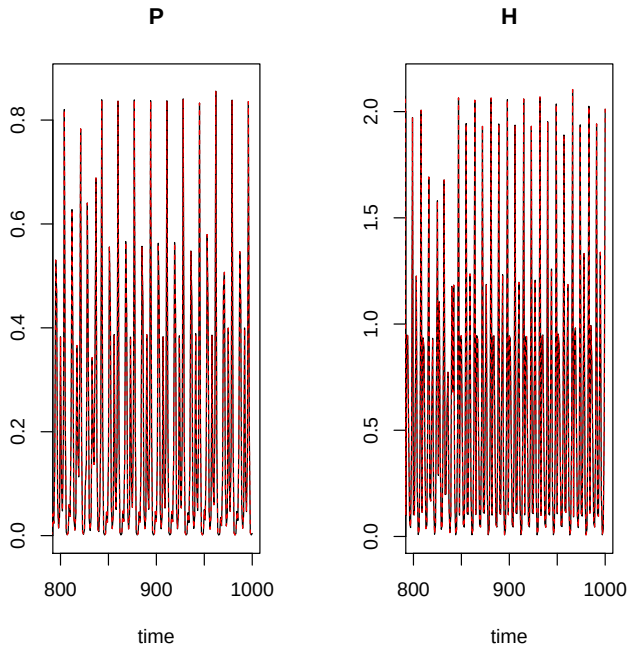


Figure 2: Solution of the iteration problem

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

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

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

### 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 (**?**). 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)

plot(outDLL, outDLL2, outDLL3, which = 1:4, 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
 "
```

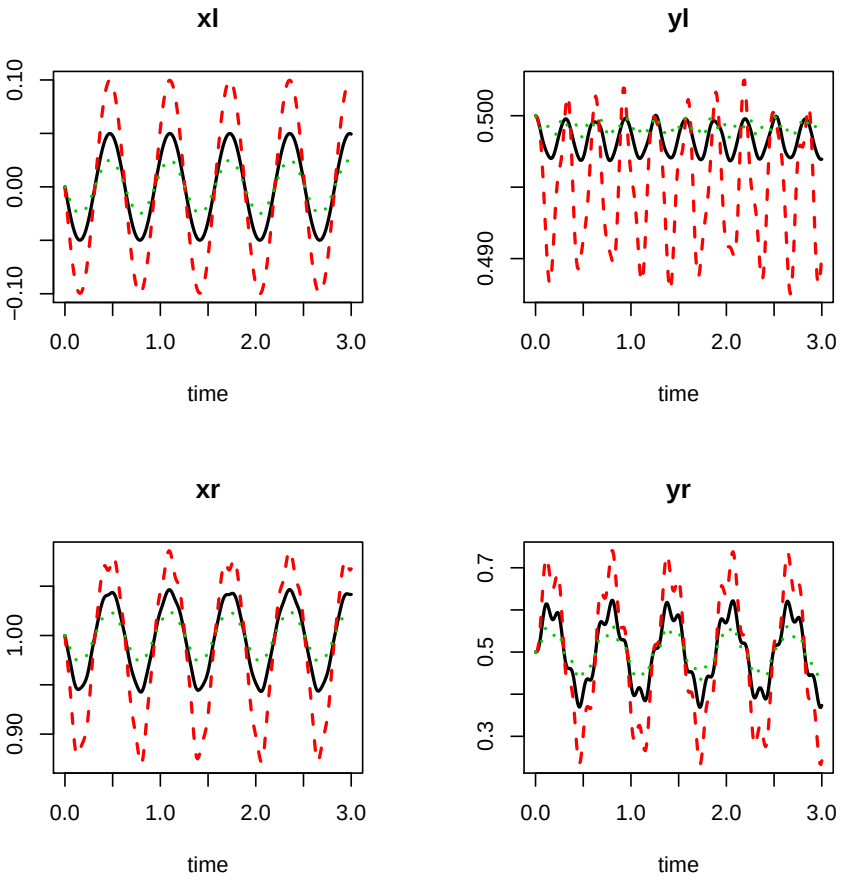


Figure 3: Solution of the linearly-implicit DAE problem

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 that are 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:

```
printCode(cBiogeo)
```

```
Program source:
  1:
  2:  SUBROUTINE initpar(deparms)
  3: EXTERNAL deparms
  4: DOUBLE PRECISION parms(9)
  5: COMMON / xcbpar / parms
  6: CALL deparms(9, parms)
  7:
  8: END
  9:
 10:  SUBROUTINE func ( n, t, y, f, rpar, ipar )
 11: IMPLICIT none
 12: INTEGER n
 13: DOUBLE PRECISION t
 14: DOUBLE PRECISION y(*)
 15: DOUBLE PRECISION f(*)
 16: DOUBLE PRECISION rpar(*)
 17: INTEGER ipar(*)
 18:
 19:   double precision  D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
 20:   common /  xcbpar  /  D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
 21:    double precision :: Min, oxicmin, anoxicmin
 22:
 23:         double precision  OM, O2, SO4, HS
 24:         double precision  dOM, dO2, dSO4, dHS
```

```
 25:
 26:         double precision SumS
 27:
 28:
 29:         if (ipar(1) <  1 ) call rexit('nout should be >=  1  ')
 30:
 31:          OM = y(1)
 32:          O2 = y(2)
 33:          SO4 = y(3)
 34:          HS = y(4)
 35:
 36:    Min       = r*OM
 37:    oxicmin   = Min*(O2/(O2+ks))
 38:    anoxicmin = Min*(1-O2/(O2+ks))* SO4/(SO4+ks2)
 39:
 40:    dOM  = Flux - oxicmin - anoxicmin
 41:    dO2  = -oxicmin      -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2)
 42:    dSO4 = -0.5*anoxicmin  +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4)
 43:    dHS  = 0.5*anoxicmin   -rox*HS*(O2/(O2+ks)) + D*(BHS-HS)
 44:
 45:    SumS = SO4 + HS
 46:
 47:          f(1) = dOM
 48:          f(2) = dO2
 49:          f(3) = dSO4
 50:          f(4) = dHS
 51:   rpar(1) = SumS
 52:
 53:
 54: RETURN
 55: END
 56:
```

The problem is solved by direct iteration; as there may be a -biologically unrealistic- negative solution, we enforce positivity (via argument `pos`). When we trigger the solver, we need to pass the parameters, initial conditions and names of the output variables, that are consistent with the ones we 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
```

We can also use the compiled model to run it in dynamic mode:

```
 out <- ode(y = y, func = cBiogeo, times = 0:50, parms = pars,
    outnames = "sumS", nout = 1)
 plot(out, which = 1:4)
```

setkeysGinwidth=0.6

## 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 (Cash and Wright 1991; Cash and Mazzia 2005).
- `bvpcol`, a collocation method based on FORTRAN codes COLNEW (Bader and Ascher 1987), and COLSYS (Ascher, Christiansen, and Russell 1979) for solving Multi-point boundary value problems of mixed order.

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

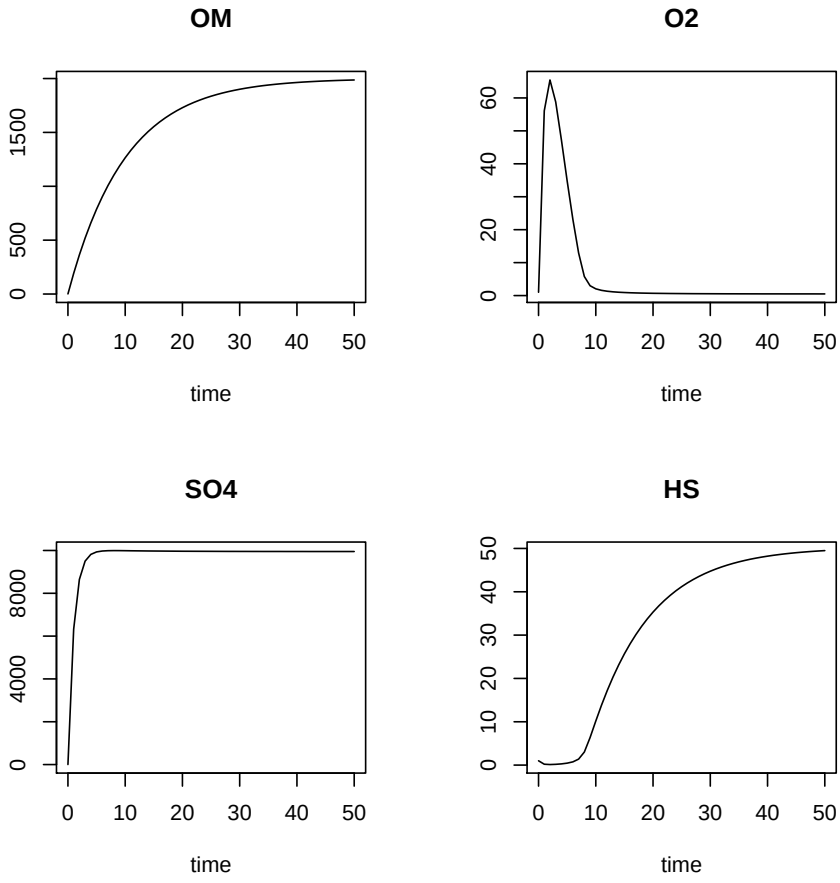


Figure 4: Solution of the biogeochemical problem

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. A Simple BVP Example implemented in fortran and in C

Here is a simple BVP ODE (which is problem 7 from the test problems available from http://www.ma.ic.ac.uk/~jcash/BVP_software/readme.php ):

$$\begin{aligned} \xi y'' + xy' - y &= -(1+\xi\pi^2)\cos(\pi x) - \pi x \sin(\pi x) \\ y(-1) &= -1 \\ y(1) &= 1 \end{aligned}$$

This is implemented in fortran 95 and in C. Note that this problem, as for most problems is much easier to read (and create) if F95 is the language chosen.

```
fun.f95 <- "
  f(1) = 1/ks * (-x * y(2) + y(1)-(1 + ks*3.14159**2)*cos(3.14159*x)-
     3.14159*x*sin(3.14159*x))
 "
```

To understand the C-code, it should be noted that the indedependent variable `x` is a pointer, hence its value is assessed either by `*x` or as `x[0]`. Also, array indexing in C starts from 0.

```
fun.C <- "
  f[0] = 1/ks * (-1 * *x *y[1]+y[0]-(1+ks*3.14159*3.14159)*cos(3.14159* *x)-
     3.14159* *x*sin(3.14159* *x));
 "
```

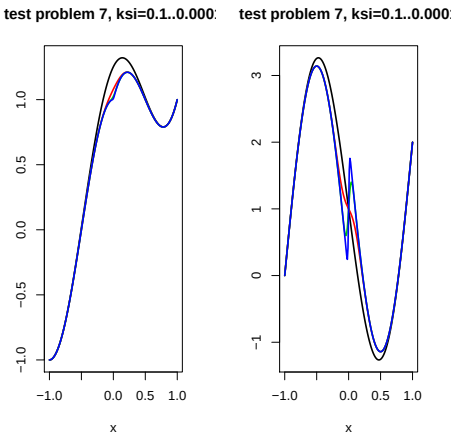


Figure 5: Solution of the simple BVP problem 7

The problem, in higher-order form can only be solved, using `bvpcol` Note that, when the problem would have been formulated in R-code, we could also have solved it with `bvptwp`.

```
ks <- 0.1
x  <- seq(-1, 1, by = 0.01)
cfun <- compile.bvp(fun.C, parms = c(ks = 0.1), language = "C")
sol1  <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = x,
   parms = 0.1, func = cfun, order = 2)
sol2  <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = x,
   parms = 0.01, func = cfun, order = 2)
sol3  <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = x,
   parms = 0.001, func = cfun, order = 2)
sol4  <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = x,
   parms = 0.0001, func = cfun, order = 2)

plot(sol1, sol2, sol3, sol4, type = "l", main = "test problem 7, ksi=0.1..0.0001",
      lwd = 2, lty = 1)
```

## 7.2. Specifying all functions in compiled code BVPs

In the previous example we only specified the derivative function in compiled code. It is possible to specify 3 other functions when solving BVPs: the jacobian function, the boundary function and the jacobian of the boundary.

We implement the measels problem as from (Ascher, Mattheij, and Russell 1995) and (**?**). 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
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
  5.79    0.07    5.85
```

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 = 3.14159265358979311 6d0\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 = "")
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
  0.14    0.06    0.22
```

```
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
  0.11    0.11    0.22
```

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

### 7.3. a multipoint boundary value problem

In `bvptwp`, the boundary conditions must be defined at the end of the interval over which the ODE is specified, but `bvpcol` can also have the boundary conditions specified at intermediate points.

We implement the multipoint example from `bvpcol`. The equations, defined over the interval [0,1] are:

$$\begin{aligned} y_1' &= (y_2 - 1)/2 \\ y_2' &= (y_1 * y_2 - x)/\mu \\ y_1(1) &= 0 \\ y_2(0.5) &= 1 \end{aligned}$$

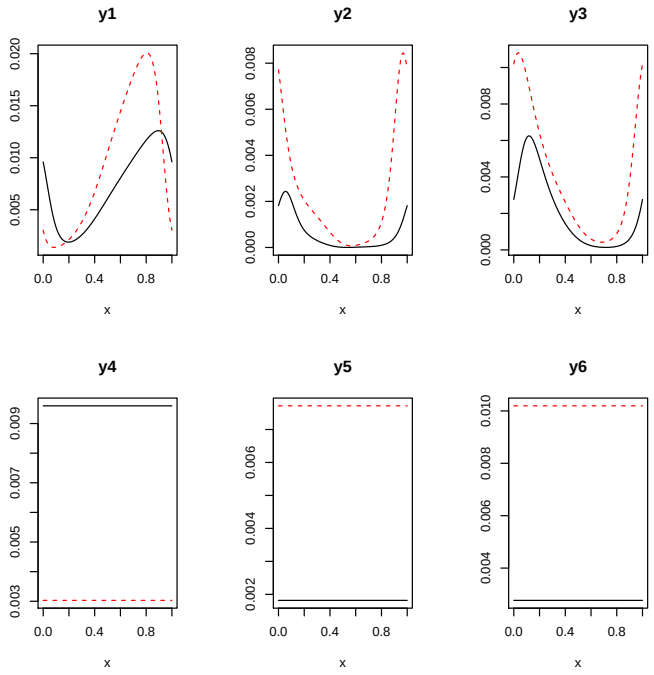


Figure 6: Two solutions of the measel BVP problem

```
multip <- "
 f(1) = (y(2) - 1)/2
 f(2) = (y(1)*y(2) - x)/mu
 "
bound <- "
   if (i == 1) then
     g = y(2) -1
   else
     g = y(1)
   endif
 "
cmultip <- compile.bvp(func = multip, bound = bound, parms = c(mu = 0.1))
mu  <- 0.1
sol <- bvpcol(func = cmultip, x = seq(0, 1, 0.01), posbound = c(0.5, 1), parms = mu)
# check boundary value
sol[sol[,1] == 0.5,]
```

```
      x        1        2
0.500000 0.338895 1.000000
```

```
plot(sol)
```

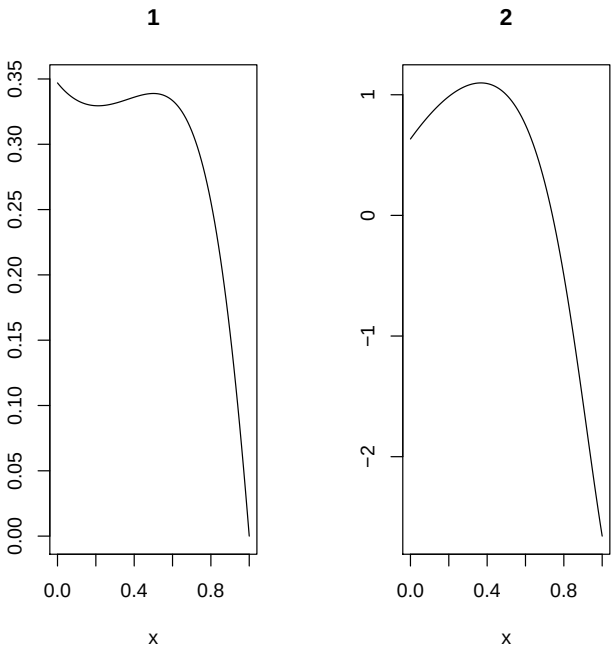


Figure 7: Solution of the multipoint problem

### 7.4. A boundary value differential algebraic equation

The following problem constitutes a simple DAE, where the last equation in the algebraic equation:

```
daebvp <- "
   f(1) = (ks + y(2) - sin(x))*y(4) + cos(x)
   f(2) = cos(x)
   f(3) = y(4)
   f(4) = (y(1) - sin(x))*(y(4) - exp(x))
 "
bounddae <- "
   if (i == 1) then
     g = (y(1) - sin(0.d0))
   else if (i == 2) then
     g = y(3) - 1
   else if (i == 3) then
     g = y(2) - sin(1.d0)
   else
     g = (y(1) - sin(1.d0))*(y(4) - exp(1.d0))
   endif
 "
cdaebvp <- compile.bvp(func = daebvp, bound = bounddae, parms = c(ks = 1e-4))
x <- seq(0, 1, by = 0.01)
mass <- diag(nrow = 4)  ; mass[4, 4] <- 0
out <- bvpcol (func = cdaebvp, x = x, atol = 1e-10, rtol = 1e-10,
               parms = 1e-4, ncomp = 4, leftbc = 2,
               dae = list(index = 2,  nalg = 1))
# the analytic solution
ana <- cbind(x, "1" = sin(x), "2" = sin(x), "3" = 1, "4" = 0, res = 0)

plot(out, obs = ana)
```

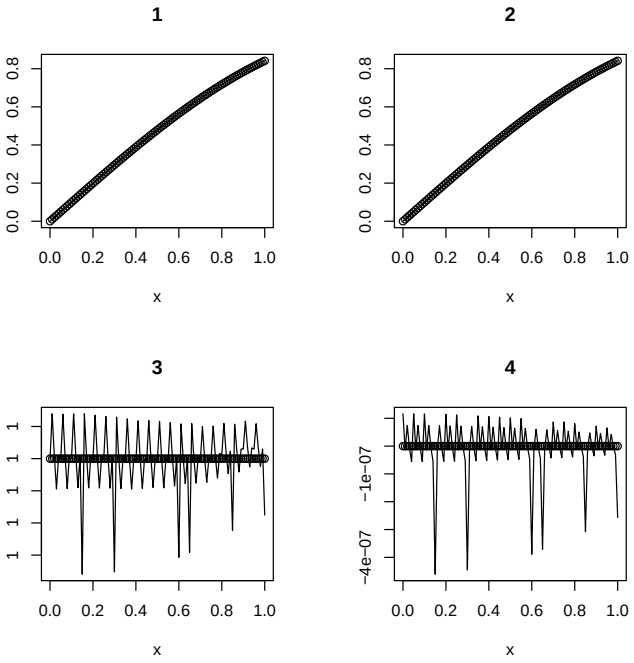


Figure 8: Solution of the BVP DAE problem

# 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 this 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 efficient 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 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
  1.36    0.00    1.36

# --------------------------------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
  0.06    0.00    0.06

# ---------------------- 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
   1.61    0.00    1.61

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

   user  system elapsed
   1.04    0.00    1.07
```

Karline:

To do: Fully implicit DAEs in compiled code - e.g. the pendulum problem.

roots and events?

PDEs but this is quite different, although promising - still under construction

## 9. Passing data

There are several ways to pass data to the compiled code. This relates to the solvers themselves.

- 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. They are unnamed, and can be used for input and output. For differential equation solvers, they are used to contain the output variables. See vignette ('compiledCode') (Soetaert *et al.* 2014b)

- `parms` is to contain the values of named variables, 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. They are not supposed to be changed. Not implemented for the minmization, uniroot and nonlinear lest squares methods (the name `parms` is too close to the official argument `par` which is completely different)

- forcings This is only used for differential equations. It is to contain the variables that are updated by the solver, every time (or spatial) step.

- `data`. This is to contain data in matrix or data.frame format, and whose length is not necessarily known at compile time. The names of the columns are used to set variable names in the code; the variables are declared in a module (Fortran) or as global variables (C). Their values are set at the start of the simulation. They are not meant to be changed.

## References

Ascher U, Christiansen J, Russell R (1979). "a collocation solver for mixed order systems of boundary value problems." *math. comp.*, **33**, 659–679.

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

Bader G, Ascher U (1987). "a new basis implementation for a mixed order boundary value ode solver." *siam j. scient. stat. comput.*, **8**, 483–500.

Cash JR, Mazzia F (2005). "A new mesh selection algorithm, based on conditioning, for two-point boundary value codes." *J. Comput. Appl. Math.*, **184**, 362–381.

Cash JR, Wright MH (1991). "A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation." *SIAM J. Sci. Stat. Comput.*, **12**, 971–989.

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.

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.

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, 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, 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