

Solving Differential Algebraic Equations in R

Karline Soetaert

CEME

Netherlands Institute of Ecology

The Netherlands

Jeff Cash

Department of mathematics

Imperial College London

U.K.

Francesca Mazzia

Dipartimento di Matematica

Universita' di Bari

Italy

Abstract

Function `mebdfi` from R package **deTestSet** solves initial value problems (IVP) of ordinary differential equations (ODE) and of differential algebraic equations (DAE) of index up to 3.

It implements the FORTRAN code `mebdfi` written by T.J. Abdulla and J.R. Cash.

Here the implementation is tested on a set of DAE test problems from <http://www.dm.uniba.it/~testset>.

Keywords: differential algebraic equations, initial value problems, R.

1. Solving initial value problems in R

Several packages in R solve differential equations:

- **deSolve** (Soetaert, Petzoldt, Setzer, 2009) for initial value problems of ordinary and partial differential equations, and simple differential algebraic equations,
- **bvpSolve** (Soetaert, Cash, Mazzia, 2009) for boundary value problems, and
- **rootSolve** (Soetaert 2009), to solve steady-state conditions

Package **deTestSet** implements a solver for differential algebraic equations of index up to 3, `mebdfi`. This R function provides an interface to the Fortran DAE solver of the same name, written by T.J. Abdulla and J.R. Cash.

`mebdfi` implements the Modified Extended Backward Differentiation formulas for stiff fully implicit initial value problems. These formulas increase the absolute stability regions of the classical BDFs.

The orders of the implemented formulae range from 1 to 8. For details about this method, see Cash (1979, 1983), Cash and Considine (1992).

In this package vignette, the new solver is tested against the DAE test problems from the following site: <http://www.dm.uniba.it/~testset> (Mazzia and Magherini, 2008).

All problem descriptions can be found there.

If the model problem is small enough, then the model is implemented in pure R, and the code is shown in this vignette. For larger models, the problem specified in FORTRAN code at the website of Jeff Cash http://www.ma.ic.ac.uk/~jcash/IVP_software were used.

These implementations were compiled as DLLs, and included in the package. The code of these models can be found in the packages `inst/examples/dynload` subdirectory.

2. the car axis problem

This is an stiff DAE, of index 3, consisting of 8 differential and 2 algebraic equations. It models the behavior of a car axis on a bumpy road. The first 4 variables represent the positions of the 4 wheels, the next 4 the velocity vector, and the last two equations impose two position constraints.

This problem is used to show how to implement a DAE model in R and in a compiled language (FORTRAN and C).

2.1. model implemented in R

The equations are in an R -function, `car`, that take as input the time (`t`), the values of the variables (`y`) and their derivatives (`dy`), and the parameters (`pars`).

The function returns the residuals (`delt`) and a vector `f`, as a list.

```
> car <- function(t,y,dy,pars){
+   with(as.list(c(pars,y)), {
+     f <- rep(0,10)
+
+     yb <- r*sin(w*t)
+     xb <- sqrt(L*L-yb*yb)
+     Ll <- sqrt(xl^2+y1^2)
+     Lr <- sqrt((xr-xb)^2+(yr-yb)^2)
+
+     f[1:4] <- y[5:8]
+     k <- M*eps*eps/2
+
+     f[5] <- (L0-Ll)*xl/Ll +lam1*xb+2*lam2*(xl-xr)
+     f[6] <- (L0-Ll)*y1/Ll +lam1*yb+2*lam2*(y1-yr)-k*g
+     f[7] <- (L0-Lr)*(xr-xb)/Lr -2*lam2*(xl-xr)
+     f[8] <- (L0-Lr)*(yr-yb)/Lr -2*lam2*(y1-yr)-k*g
+
+     f[9] <- xb*xl+yb*y1
+     f[10] <- (xl-xr)^2+(y1-yr)^2-L*L
+
+     delt <- dy-f
+     delt[5:8] <- k*dy[5:8]-f[5:8]
+     delt[9:10] <- -f[9:10]
+
+     list(delt=delt,f=f)
+   })
+ }
```

After defining the parameters (`pars`) and the initial conditions of the variables (`yini`), a consistent set of initial values for the derivatives (`dyini`) is found:

```
> pars <- c(eps = 1e-2, M = 10, L = 1, L0 = 0.5,
+          r = 0.1, w = 10, g = 1)
> # initial conditions: state variables
> yini <- with(as.list(pars),
+             c(xl=0, yl=L0, xr=L, yr=L0, xla=-L0/L,
+             yla=0, xra=-L0/L, yra=0, lam1=0, lam2=0)
+             )
> # initial conditions: derivatives
> dyini <- rep(0,10)
> FF <- car(0,yini,dyini,pars)
> dyini[1:4] <- yini[5:8]
> dyini[5:8] <- 2/pars["M"]/(pars["eps"])^2*FF$f[5:8]
```

It is a good idea to check the consistency of the initial conditions. This can be done by calling the model function `car` with these values; if consistent then its return value in `delt` should be (virtually) equal to 0.

```
> car(0,yini,dyini,pars)$delt
```

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

The first 4 variables are of index 1, the next 4 of index 2 and the last two of index 2. This is inputted in a three-valued vector `nind`.

```
> nind <- c(4,4,2)
```

After specifying the times at which output is wanted, the model can be run; we print the time it takes, in seconds to do this (`print(system.time())`):

```
> times <- seq(0,3,by=0.01)
> print(system.time(
+ out <- mebdfi(y=yini,dy=dyini,times,res=car,parms=pars,nind=nind,
+             rtol=1e-5,atol=1e-5)
+ ))
```

```
user system elapsed
0.75    0.00    0.76
```

Output is plotted, using an S3-method:

```
> plot(out,which=1:4,type="l",lwd=2,ask=FALSE)
```

2.2. The car axis problem, implemented in FORTRAN

How to write models in compiled code is extensively explained in **deSolve** package vignette "compiledCode", which should be consulted for details.

Here we simply give the code as is. Note

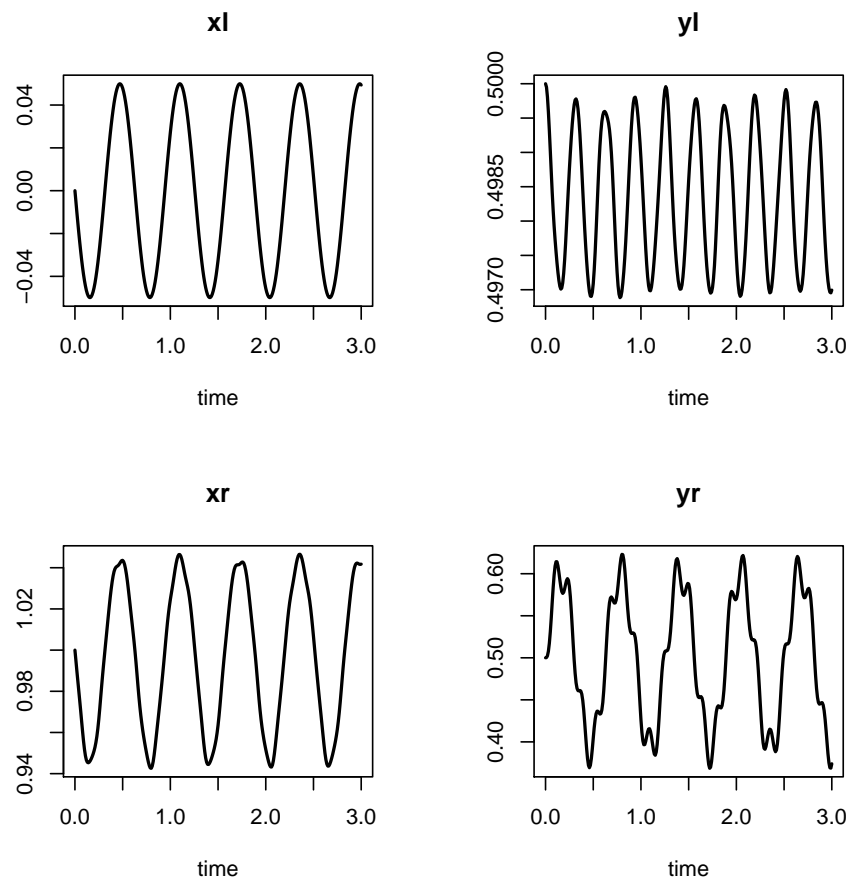


Figure 1: Solution of the car axis problem, an index 3 differential algebraic equation model - dimension 8 - see text for R-code

- parameter values are initialised in a subroutine called `carpar`. The parameters are stored in a common block (here called `myparms`). During the initialisation, this common block is defined to consist of a 7-valued vector (unnamed), but in the subroutine `carres`, the parameters are given a name (`eps,...`).
- output variables are passed via the `fout` argument; the number of output variables, an argument in the call to `mebdfi` is in `ipar(1)`. It is tested whether enough memory is allocated to the output variables; if this is not the case, then the model stops with an error message.

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

```

C-----
C initialising the parameter common block
C-----
      SUBROUTINE carpar(daeparms)

      EXTERNAL daeparms
      double precision parms(7)
      common /myparms/parms

      call daeparms(7, parms)
      return
      end

C-----
C the residual function
C-----
      SUBROUTINE carres (T,Y,YPRIME,CJ,DELTA,ier,FOUT,IPAR)
      IMPLICIT NONE
      INTEGER N, IPAR(3), IER, I
      PARAMETER (N=10)
      DOUBLE PRECISION T,Y(N), DELTA(N),YPRIME(N), FOUT(10)

      double precision eps, m, L, L0, r, w, g
      common / myparms/ eps, m, L, L0, R, W, G

      double precision k, cj, lam1, lam2
      double precision yl, yr, yb, xl, xr, xb, Lr, Ll

C
      if (ipar(1) < 10) call rexit("nout should be at least 10")
      k = m*eps*eps/2d0

      yb = r*sin(w*t)
      xb = sqrt(L*L-yb*yb)

```

```

do 10 i=1,4
    FOUT(i) = y(i+4)
10 continue

xl   = y(1)
yl   = y(2)
xr   = y(3)
yr   = y(4)
lam1 = y(9)
lam2 = y(10)

Ll = sqrt(xl**2+yl**2)
Lr = sqrt((xr-xb)**2+(yr-yb)**2)

FOUT(5) = (L0-Ll)*xl/Ll +lam1*xb+2d0*lam2*(xl-xr)
FOUT(6) = (L0-Ll)*yl/Ll +lam1*yb+2d0*lam2*(yl-yr)-k*g
FOUT(7) = (L0-Lr)*(xr-xb)/Lr -2d0*lam2*(xl-xr)
FOUT(8) = (L0-Lr)*(yr-yb)/Lr -2d0*lam2*(yl-yr)-k*g

FOUT(9) = xb*xl+yb*yl
FOUT(10) = (xl-xr)**2+(yl-yr)**2-L*L
do i=1,4
    delta(i) =yprime(i)-FOUT(i)
enddo
do i=5,8
    delta(i) = k*yprime(i)- FOUT(i)
enddo
do i=9,10
    delta(i) = -FOUT(i)
enddo

return
end
C-----

```

2.3. the car axis problem implemented in C

For code written in C, the calling sequence for `res` must be:

```

void myres(double *t, double *y, double *ydot, double *cj,
           double *delta, int *ires, double *yout, int *ip)

```

The code is given as-is:

```

/* suitable names for parameters and state variables */

```

```

#include <R.h>
static double parms[7];

#define eps parms[0]
#define m    parms[1]
#define L    parms[2]
#define L0   parms[3]
#define r    parms[4]
#define w    parms[5]
#define g    parms[6]

/*-----
  initialising the parameter common block
  -----
*/
void carparc(void (* daeparms)(int *, double *)) {
  int N = 7;
  daeparms(&N, parms);
}
/* Compartments */

#define xl y[0]
#define yl y[1]
#define xr y[2]
#define yr y[3]
#define lam1 y[8]
#define lam2 y[9]

/*-----
  the residual function
  -----
*/
void carresc (double *t, double *y, double *yprime, double *CJ,
              double *delta, int *ier, double *FOUT, int* ip) {

  double k, yb, xb, Lr, Ll;
  int i;

  if (ip[0] < 10) error("nout should be at least 10");

  k = m*eps*eps/2.;

  yb = r*sin(w* t) ;
  xb = sqrt(L*L-yb*yb);

  for (i = 0; i <4; i++)

```

```

    FOUT[i] = y[i+4];

    L1 = sqrt(xl*xl+yl*yl) ;
    Lr = sqrt((xr-xb)*(xr-xb)+(yr-yb)*(yr-yb));

    FOUT[4]  =(L0-L1)*xl/L1 +lam1*xb+2*lam2*(xl-xr)      ;
    FOUT[5]  =(L0-L1)*yl/L1 +lam1*yb+2*lam2*(yl-yr)-k*g;
    FOUT[6]  =(L0-Lr)*(xr-xb)/Lr -2*lam2*(xl-xr)        ;
    FOUT[7]  =(L0-Lr)*(yr-yb)/Lr -2*lam2*(yl-yr)-k*g    ;

    FOUT[8]  = xb*xl+yb*yl;
    FOUT[9]  = (xl-xr)*(xl-xr)+(yl-yr)*(yl-yr)-L*L;
    for (i = 0 ; i <4; i++)
        delta[i] =yprime[i]-FOUT[i];

    for (i = 4 ; i <8; i++)
        delta[i] = k*yprime[i]- FOUT[i];

    for (i = 8 ; i <10; i++)
        delta[i] = -FOUT[i];
}

```

2.4. running the car axis problem, implemented in compiled code

Before the model, implemented in compile code can be run, the fortran or C programme has to be compiled. This can be done in R , e.g.:

```

system("R CMD SHLIB car.f")
system("R CMD SHLIB carc.c")

```

(do make sure that this file is in the working directory... if this is not the case, use `setwd()`).

To run the C-code, we write:

```

require(deTestSet)
dyn.load("carc.dll")

# parameters
pars <- c(eps = 1e-2, M = 10, L = 1, L0 = 0.5,
          r   = 0.1,  w = 10, g = 1)

yini <- with (as.list(pars),
  c(xl=0, yl=L0, xr=L, yr=L0, xla=-L0/L,
    yla=0, xra=-L0/L, yra=0, lam1=0, lam2=0)
)

# initial conditions: derivates
dyini <- rep(0,10)

```



```

# 10 extra output variables (nout)...
FF <- DLLres(res="carresc",time=0.,y=yini,dy=dyini,parms= pars,
             initfunc="carparc", dllname="carc", nout=10)$var
dyini[1:4] <- yini[5:8]
dyini[5:8] <- 2/pars["M"]/(pars["eps"])^2*FF[5:8]

# check consistency of initial condition: delt should be = 0.
DLLres(res="carresc",time=0., y=yini,dy=dyini,parms= pars,
       initfunc="carparc", dllname="carc", nout=10)

# running the model
times <- seq(0,3,by=0.01)
nind <- c(4,4,2) # index 1, 2 and 3 variables
# 10 extra output variables...
out <- mebdfi(y=yini, dy=dyini, times, res="carresc", parms=parms,
              nind=nind, initfunc="carparc", dllname="carc", nout = 10,
              rtol=1e-10, atol=1e-10, outnames=c("f1","f2","f3","f4",
              "f5","f6","f7","f8","f9","f10"))

par(mar=c(3,3,3,1))
plot(out,type="l",lwd=2,mfrow=c(4,5))
dyn.unload("carc.dll")

```

The solver assumes that the model is defined in compiled code as `res` is a string rather than an R -function. Argument `dllname` gives the name (without extension) of the dynamically linkage library.

Note how the number of output variables (`nout`) is passed to the solver; also, the names of these output variables is given (`outnames`)

3. Andrews' squeezing mechanism

This is a non-stiff, second-order DAE of index 3, and comprising 14 differential and 13 algebraic equations. It describes the motion of 7 rigid bodies connected by joints without friction.

The model function, implemented in R is quite lengthy:

```
> Andrews <- function (t, y, dy, pars) {
+
+   with (as.list(pars), {
+     sibe = sin(y[1])
+     sith = sin(y[2])
+     siga = sin(y[3])
+     siph = sin(y[4])
+     side = sin(y[5])
+     siom = sin(y[6])
+     siep = sin(y[7])
+
+     cobe = cos(y[1])
+     coth = cos(y[2])
+     coga = cos(y[3])
+     coph = cos(y[4])
+     code = cos(y[5])
+     coom = cos(y[6])
+     coep = cos(y[7])
+
+     sibeth = sin(y[1]+y[2])
+     siphde = sin(y[4]+y[5])
+     siomep = sin(y[6]+y[7])
+
+     cobeth = cos(y[1]+y[2])
+     cophde = cos(y[4]+y[5])
+     coomep = cos(y[6]+y[7])
+
+     bep = y[8]
+     thp = y[9]
+     php = y[11]
+     dep = y[12]
+     omp = y[13]
+     epp = y[14]
+
+     m <- matrix(nr=7,nc=7,0)
+
+     m[1,1] = m1*ra^2 + m2*(rr^2-2*da*rr*coth+da^2) + i1 + i2
+     m[2,1] = m2*(da^2-da*rr*coth) + i2
+     m[2,2] = m2*da^2 + i2
+     m[3,3] = m3*(sa^2+sb^2) + i3
```

```

+      m[4,4] = m4*(e-ea)^2 + i4
+      m[5,4] = m4*((e-ea)^2+zt*(e-ea)*siph) + i4
+      m[5,5] = m4*(zt^2+2*zt*(e-ea)*siph+(e-ea)^2) + m5*(ta^2+tb^2) + i4 + i5
+      m[6,6] = m6*(zf-fa)^2 + i6
+      m[7,6] = m6*((zf-fa)^2-u*(zf-fa)*siom) + i6
+      m[7,7] = m6*((zf-fa)^2-2*u*(zf-fa)*siom+u^2) + m7*(ua^2+ub^2) + i6 + i7
+
+      for (j in 2:7)
+        for (i in 1:j-1)
+          m[i,j] = m[j,i]
+
+      xd = sd*coga + sc*siga + xb
+      yd = sd*siga - sc*coga + yb
+      lang = sqrt ((xd-xc)^2 + (yd-yc)^2)
+      force = - c0 * (lang - l0)/lang
+      fx = force * (xd-xc)
+      fy = force * (yd-yc)
+      f <- rep(0,7)
+      f[1] = mom - m2*da*rr*thp*(thp+2*bep)*sith
+      f[2] = m2*da*rr*bep^2*sith
+      f[3] = fx*(sc*coga - sd*siga) + fy*(sd*coga + sc*siga)
+      f[4] = m4*zt*(e-ea)*dep^2*coph
+      f[5] = - m4*zt*(e-ea)*php*(php+2*dep)*coph
+      f[6] = - m6*u*(zf-fa)*epp^2*coom
+      f[7] = m6*u*(zf-fa)*omp*(omp+2*epp)*coom
+
+      gp <- matrix(nr=6,nc=7,0)
+
+      gp[1,1] = - rr*sibe + d*sibeth
+      gp[1,2] = d*sibeth
+      gp[1,3] = - ss*coga
+      gp[2,1] = rr*cobe - d*cobeth
+      gp[2,2] = - d*cobeth
+      gp[2,3] = - ss*siga
+      gp[3,1] = - rr*sibe + d*sibeth
+      gp[3,2] = d*sibeth
+      gp[3,4] = - e*cophde
+      gp[3,5] = - e*cophde + zt*side
+      gp[4,1] = rr*cobe - d*cobeth
+      gp[4,2] = - d*cobeth
+      gp[4,4] = - e*siphde
+      gp[4,5] = - e*siphde - zt*code
+      gp[5,1] = - rr*sibe + d*sibeth
+      gp[5,2] = d*sibeth
+      gp[5,6] = zf*siomep
+      gp[5,7] = zf*siomep - u*coep
+      gp[6,1] = rr*cobe - d*cobeth

```

```

+      gp[6,2] = - d*cobeth
+      gp[6,6] = - zf*coomep
+      gp[6,7] = - zf*coomep - u*siep
+
+      g<-rep(0,7)
+      g[1] = rr*cobe - d*cobeth - ss*siga - xb
+      g[2] = rr*sibe - d*sibeth + ss*coga - yb
+      g[3] = rr*cobe - d*cobeth - e*siphde - zt*code - xa
+      g[4] = rr*sibe - d*sibeth + e*cophde - zt*side - ya
+      g[5] = rr*cobe - d*cobeth - zf*coomep - u*siep - xa
+      g[6] = rr*sibe - d*sibeth - zf*siomep + u*coep - ya
+
+      ff <- rep(0,21)
+      ff[1:14] <-y[8:21]
+
+      for (i in 15:21) {
+        ff[i] = -f[i-14]
+        for (j in 1:7)
+          ff[i] = ff[i]+m[i-14,j]*y[j+14]
+
+        for (j in 1:6)
+          ff[i] = ff[i]+gp[j,i-14]*y[j+21]
+      }
+      for (i in 22:27)
+        ff[i] = g[i-21]
+
+      ff[1:14] = dy[1:14]-ff[1:14]
+      ff[15:27] = - ff[15:27]
+      return(list(ff))
+    })
+  }

```

A consistent set of initial conditions is:

```

> yini <- c(-0.0617138900142764496358948458001, 0,
+ 0.455279819163070380255912382449, 0.222668390165885884674473185609,
+ 0.487364979543842550225598953530, -0.222668390165885884674473185609,
+ 1.23054744454982119249735015568 ,0,
+ 0,0, 0,0, 0,0,
+ 14222.4439199541138705911625887, -10666.8329399655854029433719415,
+ 0,0, 0,0,
+ 0,98.5668703962410896057654982170,
+ -6.12268834425566265503114393122,0,
+ 0,0, 0)
> yprime <- rep(0,27)
> yprime[1:14] <- yini[8:21]

```

Parameter values are given by:

```
> parameter <- c(m1=.04325, m2=.00365, m3=.02373 ,m4=.00706 ,
+               m5=.07050 ,m6=.00706 ,m7=.05498 ,
+               xa=-.06934 ,ya=-.00227 ,
+               xb=-0.03635 ,yb=.03273 ,
+               xc=.014 ,yc=.072 ,c0=4530 ,
+               i1=2.194e-6,i2=4.410e-7,i3=5.255e-6,i4=5.667e-7,
+               i5=1.169e-5,i6=5.667e-7,i7=1.912e-5,
+               d=28e-3,da=115e-4,e=2e-2,ea=1421e-5,
+               rr=7e-3,ra=92e-5,l0=7785e-5,
+               ss=35e-3,sa=1874e-5,sb=1043e-5,sc=18e-3,sd=2e-2,
+               ta=2308e-5,tb=916e-5,u=4e-2,ua=1228e-5,ub=449e-5,
+               zf=2e-2,zt=4e-2,fa=1421e-5,mom=33e-3)
```

The initial conditions are consistent:

```
> Andrews(0,yini,yprime,parameter)
```

```
[[1]]
 [1] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 [5] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 [9] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[13] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[17] -2.386980e-15 0.000000e+00 0.000000e+00 0.000000e+00
[21] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
[25] -2.168404e-18 1.387779e-17 3.035766e-18
```

The first 7 components are of index 1; the next 7 are of index 2, the remaining 13 of index 3:

```
> ind <- c(7,7,13)
```

After specifying the `times`, the model is solved and `diagnostics` printed:

```
> times <- seq(0,0.03,by=0.001)
> print (system.time(
+ MM <- mebdfi(y=yini, times=times, res=Andrews, parms=parameter, nind=ind,
+   atol=1e-8,rtol=1e-8)
+ ))
```

```
user  system elapsed
7.55   0.00   7.56
```

```
> diagnostics(MM)
```

```
-----
mebdfi return code
-----
```

```

return code (idid) = 0
Integration was successful.

```

```

-----
INTEGER values
-----

```

```

1 The return code : 0
2 The number of steps taken for the problem so far: 405
3 The number of function evaluations for the problem so far: 3032
4 The number of Jacobian evaluations so far: 49
5 The method order last used (successfully): 5
10 The number of matrix LU decompositions so far: 49
13 The number of error test failures of the integrator so far: 15
14 The number of Jacobian evaluations and LU decompositions so far: 98
18 The order (or maximum order) of the method: 7
19 The number of backsolves so far 1659
20 The number of times a new coefficient matrix has been formed so far 49
21 The number of times the order of the method has been changed so far 20

```

```

-----
RSTATE values
-----

```

```

1 The step size in t last used (successfully): 1.556857e-05

```

Finally, output is plotted:

```

> plot(MM, which=1:9, type="l", lwd=2, ask=FALSE)

```

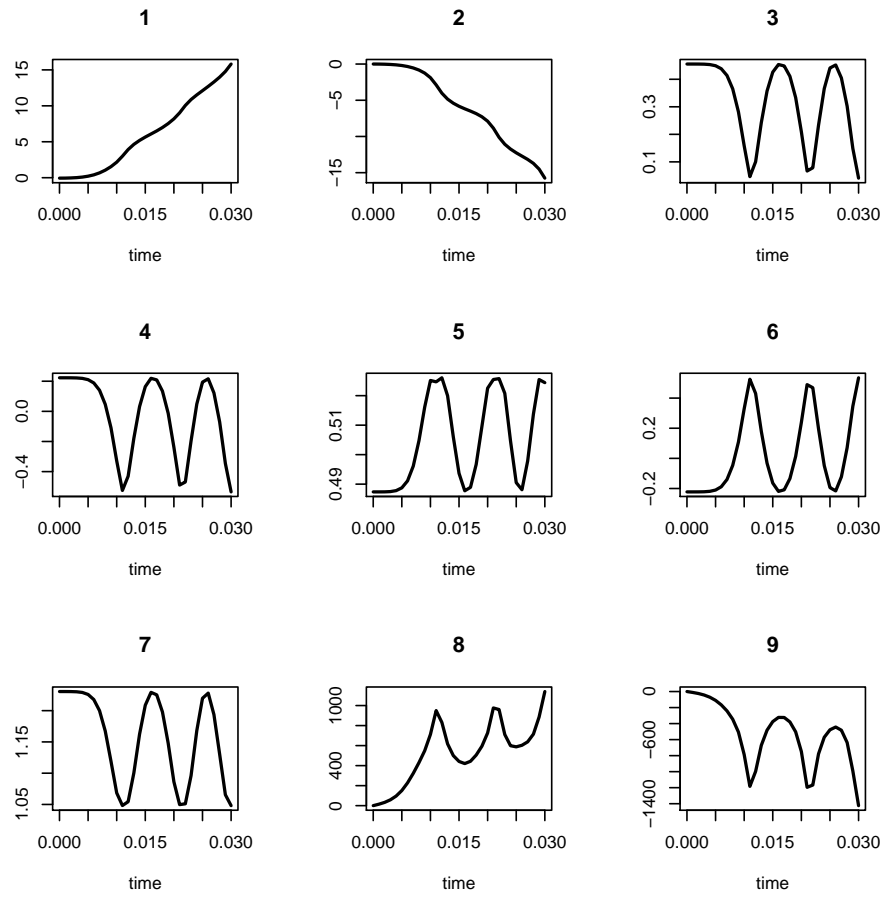


Figure 2: Solution of the Andrews squeezing mechanism problem, an index 3 differential algebraic equation model - dimension 27 - see text for R-code

4. Transistor amplifier

This is a stiff DAE of index 1, consisting of 8 equations. It originates from electrical circuit analysis. This model is still small enough to implement it in R -code

```
> Transistor<- function(t,y,dy,pars) {
+   deltt <- rep(0,8)
+   with (as.list(pars), {
+     uet    = 0.1*sin(200*pi*t)
+     fac1   = beta*(exp((y[2]-y[3])/uf)-1)
+     fac2   = beta*(exp((y[5]-y[6])/uf)-1)
+
+     deltt[1] = (y[1]-uet)/r0
+     deltt[2] = y[2]/r1+(y[2]-ub)/r2+(1-alpha)*fac1
+     deltt[3] = y[3]/r3-fac1
+     deltt[4] = (y[4]-ub)/r4+alpha*fac1
+     deltt[5] = y[5]/r5+(y[5]-ub)/r6+(1-alpha)*fac2
+     deltt[6] = y[6]/r7-fac2
+     deltt[7] = (y[7]-ub)/r8+alpha*fac2
+     deltt[8] = y[8]/r9
+
+     deltt[1] = -c1*dy[1]+c1*dy[2] -deltt[1]
+     deltt[2] =  c1*dy[1]-c1*dy[2] -deltt[2]
+     deltt[3] = -c2*dy[3]          -deltt[3]
+     deltt[4] = -c3*dy[4]+c3*dy[5] -deltt[4]
+     deltt[5] =  c3*dy[4]-c3*dy[5] -deltt[5]
+     deltt[6] = -c4*dy[6]          -deltt[6]
+     deltt[7] = -c5*dy[7]+c5*dy[8] -deltt[7]
+     deltt[8] =  c5*dy[7]-c5*dy[8] -deltt[8]
+
+     list(deltt)
+   })
+ }
```

An analytical jacobian can easily be calculated:

```
> tranjac <- function(t,y,dy,pars,cj) {
+
+   with (as.list(pars), {
+     fac1p = beta*exp((y[2]-y[3])/uf)/uf
+     fac2p = beta*exp((y[5]-y[6])/uf)/uf
+
+     dfdy <- matrix(nr=8,nc=8,0)
+
+     dfdy[1,3] = -(1-alpha)*fac1p
+     dfdy[1,6] = -(1-alpha)*fac2p
+     dfdy[2,1] = 1/r0
+     dfdy[2,2] = 1/r1+1/r2+(1-alpha)*fac1p
```



```

+      dfdy[2,3] = 1/r3+fac1p
+      dfdy[2,4] = 1/r4
+      dfdy[2,5] = 1/r5+1/r6+(1-alpha)*fac2p
+      dfdy[2,6] = 1/r7+fac2p
+      dfdy[2,7] = 1/r8
+      dfdy[2,8] = 1/r9
+      dfdy[3,2] = -fac1p
+      dfdy[3,3] = -alpha*fac1p
+      dfdy[3,5] = -fac2p
+      dfdy[3,6] = -alpha*fac2p
+      dfdy[4,2] = alpha*fac1p
+      dfdy[4,5] = alpha*fac2p
+      return(dfdy)
+    })
+  }

```

Parameter values and a consistent set of initial conditions are:

```

> parameter <- c(ub=6,uf=0.026,alpha=0.99,beta=1e-6,
+               r0=1000,r1=9000,r2=9000,r3=9000,
+               r4=9000,r5=9000,r6=9000,r7=9000,
+               r8=9000,r9=9000,c1=1e-6,c2=2e-6,c3=3e-6,
+               c4=4e-6,c5=5e-6)
> yini <- with(as.list(parameter), c(0,ub/(r2/r1+1),ub/(r2/r1+1),
+   ub, ub/(r6/r5+1), ub/(r6/r5+1),ub,0))
> dyini <- with(as.list(parameter),c(51.338775,51.338775,-yini[2]/(c2*r3),
+   -24.9757667,-24.9757667,-83.33333333,-10.00564453,-10.00564453))

```

A first application runs the model for 0.2 time units and compares the solution with the "true" solution:

```

> ind <- c(8,0,0)
> tran1 <- mebdfi(y=yini, dy = dyini, times=c(0,0.2), res=Transistor,
+   parms=parameter, nind=ind, hini=1e-6,
+   atol=1e-6, rtol=1e-6, maxsteps=100000)
> true <-c(-0.5562145012262709e-2,0.3006522471903042e1,0.2849958788608128e1,
+   0.2926422536206241e1,0.2704617865010554e1,0.2761837778393145e1,
+   0.4770927631616772e1,0.1236995868091548e1)
> tran1[2,-1]-true

```

	1	2	3	4	5
	4.211206e-13	1.175948e-12	2.166267e-12	-3.954734e-10	-3.877911e-10
	6	7	8		
	-4.428546e-10	-5.598460e-08	-6.702794e-08		

Next a dynamic simulation is performed, in a first run the jacobian is estimated numerically.

```
> times <- seq(0,0.2,0.005)
> print(system.time(
+ tran <- mebdfi(y=yini, dy = dyini, times=times, res=Transistor,
+   parms=parameter, nind=ind, atol=1e-6,rtol=1e-6,
+   hini = 1e-6, maxsteps=100000)
+ ))
```

```
user  system elapsed
4.53   0.00   4.57
```

A next run then uses the analytic jacobian (jacres = tranjac)

```
> print(system.time(
+ tran2 <- mebdfi(y=yini, dy = dyini, times=times, res=Transistor,
+   parms=parameter, nind=ind, jacres= tranjac,
+   hini=1e-6, atol=1e-6, rtol=1e-6, maxsteps=100000)
+ ))
```

```
user  system elapsed
4.42   0.00   4.43
```

```
> plot(tran, type="l", lwd=2,ask=FALSE)
```

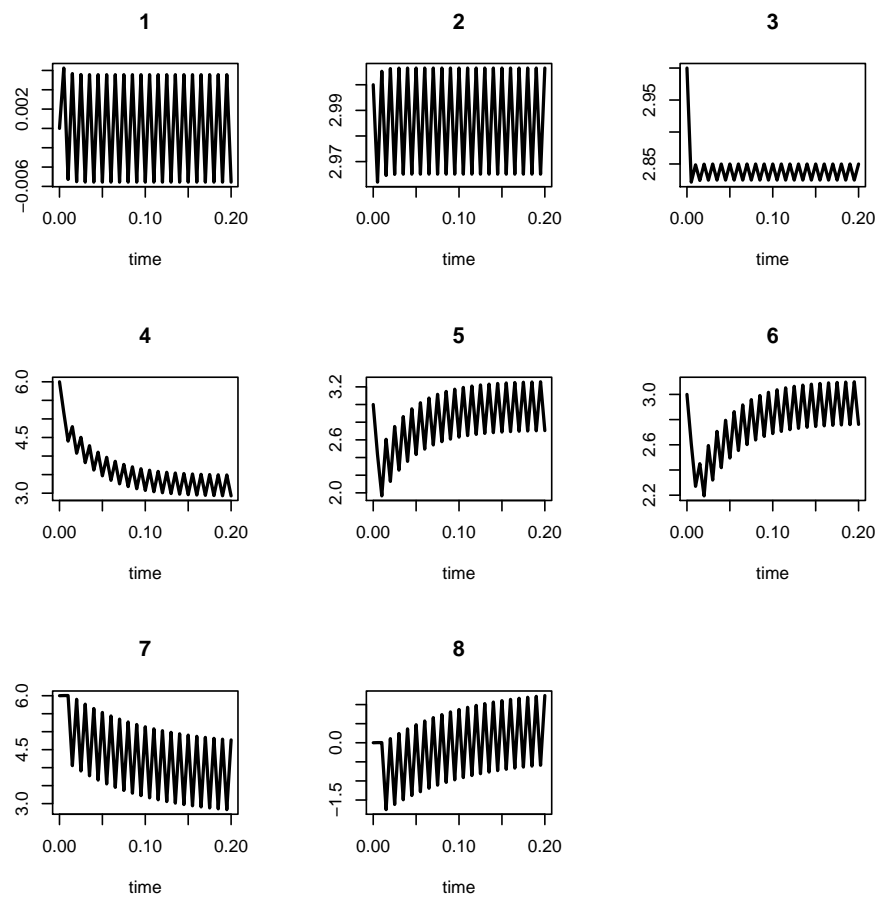


Figure 3: Solution of the transistor amplifier problem, a differential algebraic equation model of index 1, dimension 8 - see text for R-code

5. Charge pump problem

This problem cannot be solved by `mebdfi`.

6. Two bit adding unit

This is a stiff DAE problem of index 1, comprizing 175 differential and 175 algebraic equations which originates from circuit analysis. This problem is much too large to implement it in R -code, so the FORTRAN implementation is used instead.

The FORTRAN code contains two subroutines, one that initialises the problem (`twobinit`) and one that calculates the residuals (`twobres`).

The initial conditions are estimated by invoking R's `.Fortran` function; the subroutine is called "twobinit", and it requires as input the number of variables (`N`), an integer, and the time, variable and derivative vectors. It is important to ensure that integer and double precision values are passed as such; `as.integer` and `as.double` does that:

```
> N <- 350
> # Initial conditions in a fortran function
> Init <- .Fortran("twobinit",N=as.integer(N),
+               T=as.double(0),Y=as.double(rep(0.,N)),
+               dY=as.double(rep(0.,N)))
> yini <- Init$Y
> yprime <- Init$dY
```

All variables are of index 1:

```
> ind <- c(N,0,0)
```

The model is now solved by passing the *name* of the residual subroutine, `res` and the *DLL* ¹ where this subroutine can be found (`dllname`).

```
> times <- seq(0,320,by=0.1)
> print(system.time(
+ Twobit <- mebdfi(y=yini,dy=yprime,times=times,res="twobres", nind=ind,
+               dllname="deTestSet", initfunc=NULL, parms=NULL,
+               hini=1e-4,atol=1e-4,rtol=1e-4,maxsteps=100000)
+ ))
```

```
      user  system elapsed
13.29      0.04    13.38
```

The most interesting output is plotted:

```
> plot(Twobit,which=c(49,130,148),type="l",lwd=2,mfrow=c(3,1), ask=FALSE)
```

¹as all fortran examples are included in the package the DLL name is equal to the package name, `deTestSet`

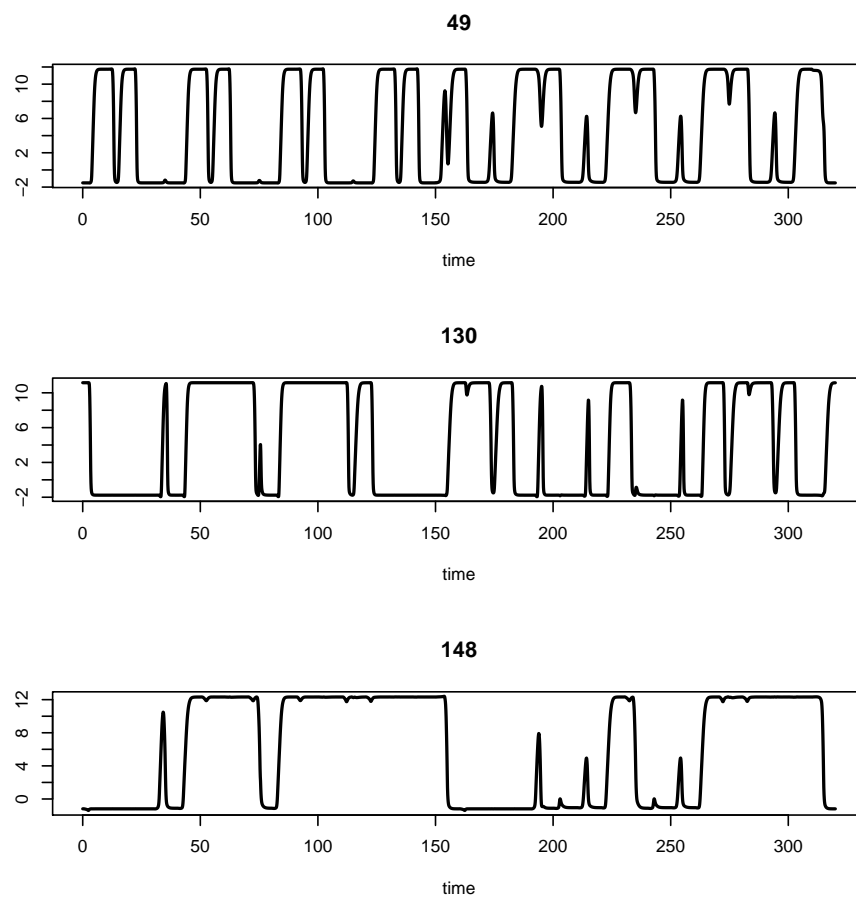


Figure 4: Solution of The two bit adding unit problem, an index 1 DAE problem, dimension 350 - see text for R-code

7. The fekete problem

This is an index 2 DAE from mechanics. The dimension is $8*N$, with $N = 20$, there are 160 variables.

The FORTRAN implementation, as made available within the package is used:

The initial conditions are in a Fortran function "fekinit".

```
> Init <- .Fortran("fekinit",N=as.integer(160),
+                  T=as.double(0),Y=as.double(rep(0.,160)),
+                  dY=as.double(rep(0.,160)))
> yini <- Init$Y
> yprime <- Init$dY
```

The first $6*20$ variables are of index 1, the remaining of index 2:

```
> ind <- c(6*20,2*20,0)
```

The model is run for 20 time units, and a plot of the first 6 variables is made:

```
> times <- seq(0,20,by=0.1)
> print(system.time(
+ Fekete <- mebdfi(y=yini,dy=yprime,times=times,res="fekres", nind=ind,
+                 dllname="deTestSet", initfunc=NULL, parms=NULL,
+                 hini=1e-8,atol=1e-8,rtol=1e-8,maxsteps=100000)
+ ))
```

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

```
> plot(Fekete,which=1:6,type="l",lwd=2,ask=FALSE)
```

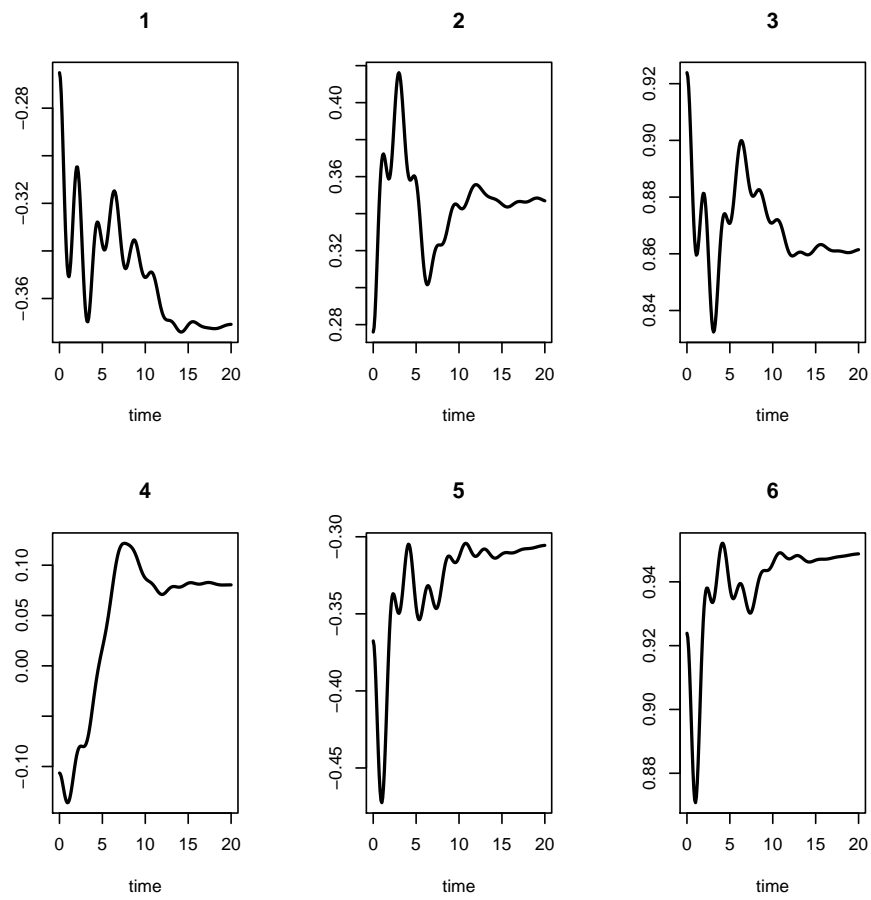


Figure 5: Solution of the fekte problem, an index 2 DAE model, dimension 160 - see text for R-code

8. The slider crank problem

This is a model of a constrained mechanical system, which includes both rigid and elastic bodies.

It is an index 2 DAE of dimension 24.

The FORTRAN implementation, as made available within the package is used.

A consistent set of initial values is:

```
> y <- c(0,0,0.450016933,0,0,0.103339863e-4,0.169327969e-4,
+       0.150000000e3,-.749957670e2,-.268938672e-5,0.444896105,
+       0.463434311e-2,-.178591076e-5,-.268938672e-5,
+       0,-1.344541576008661e-3,-5.062194923138079e3,
+       -6.833142732779555e-5,1.449382650173157e-8,
+       -4.268463211410861,2.098334687947376e-1,
+       -6.397251492537153e-08,3.824589508329281e2,
+       -4.376060460948886e-09)
> names(y) <- c("phi1","phi2","x3","q1","q2","q3","q4",
+       "vphi1","vphi2","vx3","vq1","vq2","vq3","vq4",
+       "aphi1","aphi2","ax3","aq1","aq2","aq3","aq4",
+       "la1","la2","la3")
> yprime <- c(y[8:21],rep(0,10))
```

The stiffness and damping options are set with ipar:

```
> nind <- c(14,10,0)
> times <- seq(0,0.1,by=0.001)
> ipar <- c(0,0)
> parameter <- c(M1 = 0.36, M2 = 0.151104, M3 = 0.075552, L1 = 0.15,
+       L2 = 0.3, J1 = 0.002727, J2 = 0.0045339259, EE = 2e+11,
+       NUE = 0.3, BB = 0.008, HH = 0.008, RHO = 7870, GRAV = 0,
+       OMEGA = 150)
> Crank <- mebdfi(y=y,dy=yprime,times=times,res="crankres", nind=nind,
+       dllname="deTestSet",initfunc="crankpar", parms=parameter,
+       ipar=ipar, maxsteps=100000)

> plot(Crank, type="l", lwd=2,
+       which=c("phi2","x3","q1","q2","q3","q4","la1","la2","la3"),
+       ask=FALSE)
```

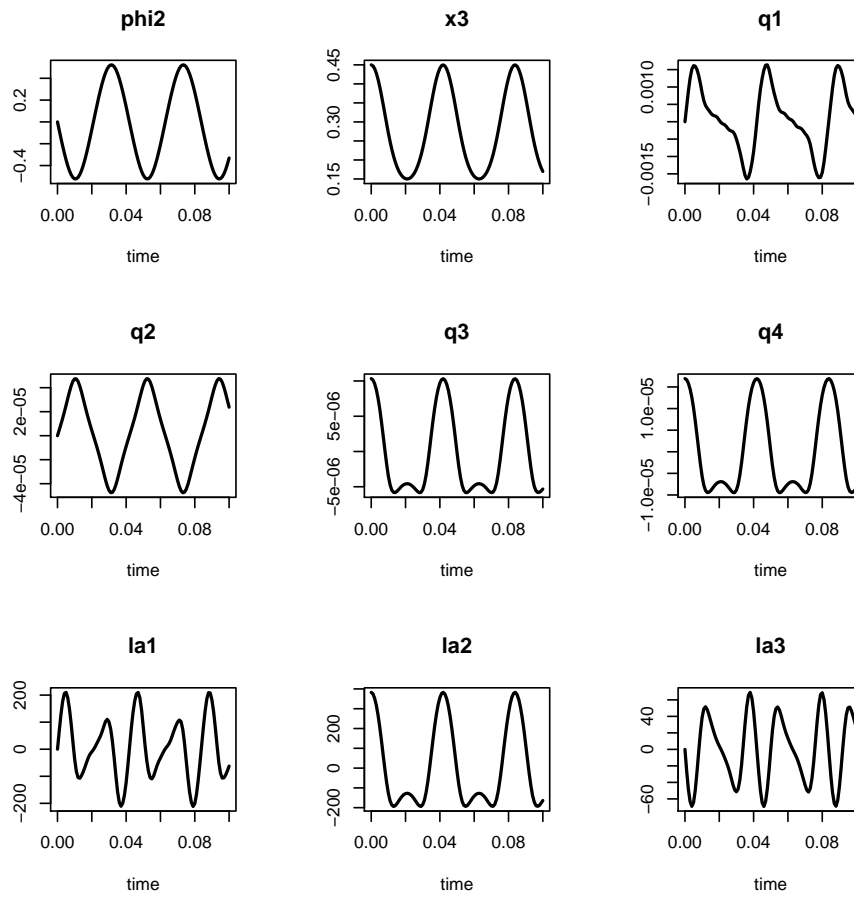


Figure 6: Solution of the crank-slider problem, an index 2 DAE model, dimension 24 - see text for R-code

9. Water tube system

This index 2 system comprising 49 non-linear DAEs, describes water flow through a tube system, taking into account turbulence and the roughness of the tube walls.

The initial conditions and parameters are specified first:

```
> yini <- rep(0,49)
> yprime <- rep(0,49)
> yini[19:36]<- 0.47519404529185289807e-1
> yini[37:49] <-109800
> parameter <- c(nu = 1.31e-6, g = 9.8, rho = 1.0e3, rcrit = 2.3e3,
+               length= 1.0e3, k = 2.0e-4, d= 1.0e0, b = 2.0e2)
```

The index of the system:

```
> ind <- c(38,11,0)
```

It is necessary to use different tolerances for the different variables

```
> atol <- rep(1e-10,49)
> atol[37:49] <- 10000
> rtol <-rep(1e-10,49)
```

The model is run for 17 hours:

```
> times <- seq(0,17.0*3600,by=10)
> print(system.time(
+ Water <- mebdfi(y=yini,dy=yprime,times=times,res="tuberes", nind=ind,
+               dllname="deTestSet",initfunc="tubepar", parms=parameter,
+               atol=atol,rtol=rtol,maxsteps=100000)
+ ))
```

```
user  system elapsed
0.8    0.0    0.8
```

The behavior of variable 4 for t in [6878,17*3600] is interesting:

```
> plot(Water, which=4, type="l", lwd=2, ask=FALSE,
+      main="water tube model, y4",
+      xlim=c(10000,60000), ylim=c(0.000145,0.000185))
```

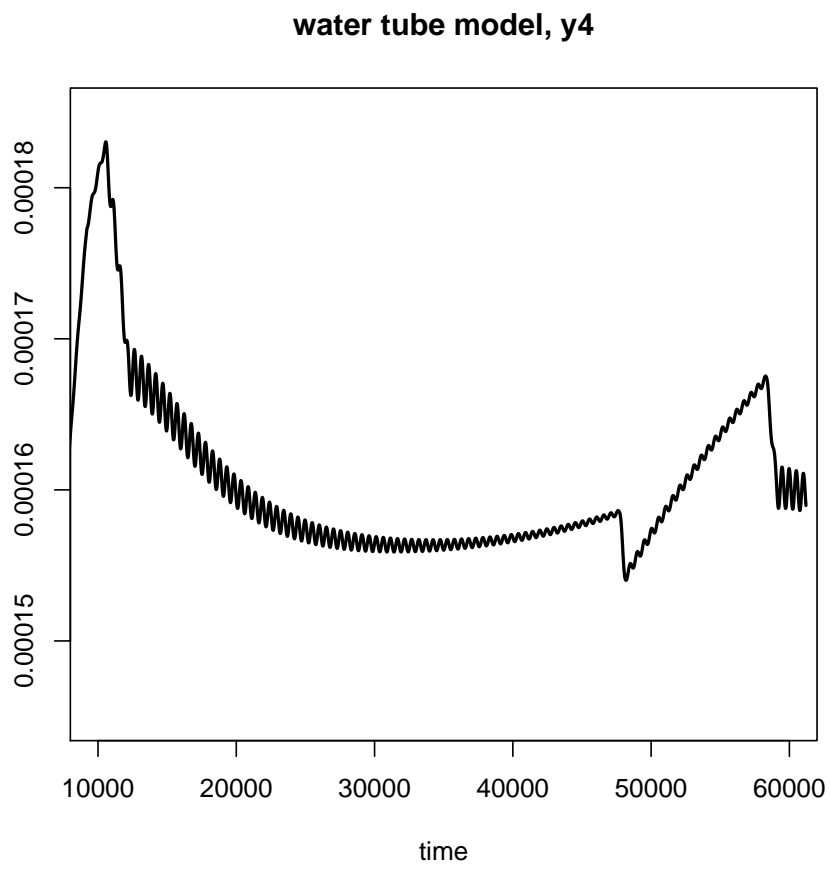


Figure 7: Solution of the water tube system problem, an an index 2 DAE model, dimension 49 - see text for R-code

10. Nand gate

This system of 14 stiff implicit differential equations of index 1 is below implemented in R .

```
> Nand <- function(t, Y, Yprime, pars)
+ {
+ #-----
+ # Voltage-dependent capacitance matrix C(Y) for the network equation
+ #          C(Y) * Y' - f(Y,t) = 0
+ #-----
+
+     CAP[1,1]=CGS
+     CAP[1,5]=-CGS
+     CAP[2,2]=CGD
+     CAP[2,5]=-CGD
+     CAP[3,3]=CBDBS(Y[3]-Y[5])
+     CAP[3,5]=-CBDBS(Y[3]-Y[5])
+     CAP[4,4]=CBDBS(Y[4]-VDD)
+     CAP[5,1]=-CGS
+     CAP[5,2]=-CGD
+     CAP[5,3]=-CBDBS(Y[3]-Y[5])
+     CAP[5,5]=CGS+CGD-CAP[5,3]+  CBDBS(Y[9]-Y[5])+C9
+     CAP[5,9]=-CBDBS(Y[9]-Y[5])
+     CAP[6,6]=CGS
+     CAP[7,7]=CGD
+     CAP[8,8]=CBDBS(Y[8]-Y[10])
+     CAP[8,10]=-CBDBS(Y[8]-Y[10])
+     CAP[9,5]=-CBDBS(Y[9]-Y[5])
+     CAP[9,9]=CBDBS(Y[9]-Y[5])
+     CAP[10,8]=-CBDBS(Y[8]-Y[10])
+     CAP[10,10]=-CAP[8,10]+CBDBS(Y[14]-Y[10])+C9
+     CAP[10,14]=-CBDBS(Y[14]-Y[10])
+     CAP[11,11]=CGS
+     CAP[12,12]=CGD
+     CAP[13,13]=CBDBS(Y[13])
+     CAP[14,10]=-CBDBS(Y[14]-Y[10])
+     CAP[14,14]=CBDBS(Y[14]-Y[10])
+
+ # -----
+ #          PULSE: Input signal in pulse form
+ # -----
+
+     P1  <- PULSE(t,0.0,5.0,5.0,5.0,5.0,5.0,20.0)
+     V1  <- P1$VIN
+     V1D <- P1$VIND
+
+     P2  <- PULSE(t,0.0,5.0,15.0,5.0,15.0,5.0,40.0)
+     V2  <- P2$VIN
```

```

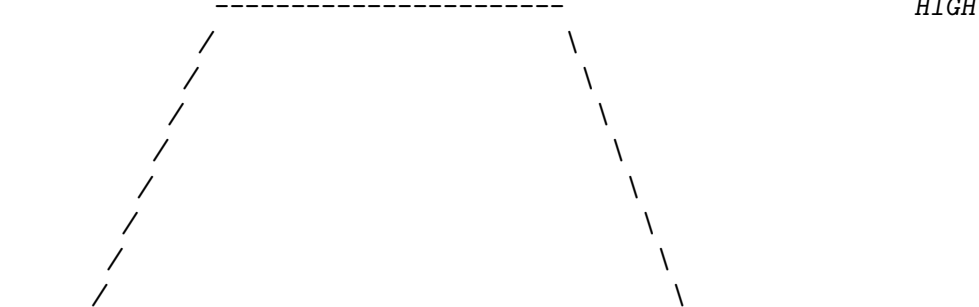
+       V2D <- P2$VIND
+
+ #-----
+ # Right-hand side f[X,t] for the network equation
+ #        $C[Y] * Y' - f[Y,t] = 0$ 
+ # External reference:
+ #       IDS: Drain-source current
+ #       IBS: Nonlinear current characteristic for diode between
+ #             bulk and source
+ #       IBD: Nonlinear current characteristic for diode between
+ #             bulk and drain
+ #-----
+
+       F[1]=-(Y[1]-Y[5])/RGS-IDS(1,Y[2]-Y[1],Y[5]-Y[1],Y[3]-Y[5],
+       Y[5]-Y[2],Y[4]-VDD)
+       F[2]=-(Y[2]-VDD)/RGD+IDS(1,Y[2]-Y[1],Y[5]-Y[1],Y[3]-Y[5],
+       Y[5]-Y[2],Y[4]-VDD)
+       F[3]=-(Y[3]-VBB)/RBS + IBS(Y[3]-Y[5])
+       F[4]=-(Y[4]-VBB)/RBD + IBD(Y[4]-VDD)
+       F[5]=-(Y[5]-Y[1])/RGS-IBS(Y[3]-Y[5])-(Y[5]-Y[7])/RGD-
+       IBD(Y[9]-Y[5])
+       F[6]=CGS*V1D-(Y[6]-Y[10])/RGS-
+       IDS(2,Y[7]-Y[6],V1-Y[6],Y[8]-Y[10],V1-Y[7],Y[9]-Y[5])
+       F[7]=CGD*V1D-(Y[7]-Y[5])/RGD+
+       IDS(2,Y[7]-Y[6],V1-Y[6],Y[8]-Y[10],V1-Y[7],Y[9]-Y[5])
+       F[8]=-(Y[8]-VBB)/RBS + IBS(Y[8]-Y[10])
+       F[9]=-(Y[9]-VBB)/RBD + IBD(Y[9]-Y[5])
+       F[10]=-(Y[10]-Y[6])/RGS-IBS(Y[8]-Y[10])-
+       (Y[10]-Y[12])/RGD-IBD(Y[14]-Y[10])
+       F[11]=CGS*V2D-Y[11]/RGS-IDS(2,Y[12]-Y[11],V2-Y[11],Y[13],
+       V2-Y[12],Y[14]-Y[10])
+       F[12]=CGD*V2D-(Y[12]-Y[10])/RGD+
+       IDS(2,Y[12]-Y[11],V2-Y[11],Y[13],V2-Y[12],Y[14]-Y[10])
+       F[13]=-(Y[13]-VBB)/RBS + IBS(Y[13])
+       F[14]=-(Y[14]-VBB)/RBD + IBD(Y[14]-Y[10])
+
+ #        $C[Y] * Y' - f[Y,t] = 0$ 
+       Delta <- colSums(t(CAP)*Yprime)-F
+       return(list(c(Delta),pulse1=P1$VIN,pulse2=P2$VIN))
+   }
+
+ # -----
+ # Function evaluating the drain-current due to the model of
+ # Shichman and Hodges
+ # -----
+
+ >
+ > IDS <- function (NED,      #   NED   Integer parameter for MOSFET-type
+ +                       VDS,  #   VDS   Voltage between drain and source

```

```

+           VGS,  #   VGS  Voltage between gate and source
+           VBS,  #   VBS  Voltage between bulk and source
+           VGD,  #   VGD  Voltage between gate and drain
+           VBD)  #   VBD  Voltage between bulk and drain
+
+ {
+   if ( VDS == 0 ) return(0)
+
+   if (NED== 1) { #--- Depletion-type
+     VTO      =-2.43
+     CGAMMA   = 0.2
+     PHI      = 1.28
+     BETA      = 5.35e-4
+   } else      { # --- Enhancement-type
+     VTO      = 0.2
+     CGAMMA   = 0.035
+     PHI      = 1.01
+     BETA      = 1.748e-3
+   }
+
+   if ( VDS > 0 ) # drain function for VDS>0
+   {
+     SQRT1<-ifelse (PHI-VBS>0,sqrt(PHI-VBS),0)
+     VTE = VTO + CGAMMA * ( SQRT1 - sqrt(PHI) )
+
+     if ( VGS-VTE <= 0.0) IDS = 0.  else
+     if ( 0.0 < VGS-VTE & VGS-VTE <= VDS )
+       IDS = - BETA * (VGS - VTE)^ 2.0 * (1.0 + DELTA*VDS) else
+     if ( 0.0 < VDS & VDS < VGS-VTE )
+       IDS = - BETA * VDS * (2 *(VGS - VTE) - VDS) * (1.0 + DELTA*VDS)
+
+     } else {
+
+     SQRT2<-ifelse (PHI-VBD>0,sqrt(PHI-VBD),0)
+     VTE = VTO + CGAMMA * (SQRT2 - sqrt(PHI) )
+
+     if ( VGD-VTE <= 0.0) IDS = 0.0  else
+     if ( 0.0 < VGD-VTE & VGD-VTE <= -VDS )
+       IDS = BETA * (VGD - VTE)^2.0 * (1.0 - DELTA*VDS) else
+     if ( 0.0 < -VDS & -VDS < VGD-VTE )
+       IDS = - BETA * VDS * (2 *(VGD - VTE) + VDS) *(1.0 - DELTA*VDS)
+     }
+   return(IDS)
+ }
+ # -----
+ # Function evaluating the current of the pn-junction between bulk and
+ # source due to the model of Shichman and Hodges

```

```
> # -----  
>  
> IBS <- function (VBS)      # VBS Voltage between bulk and source  
+       ifelse ( VBS <= 0.0, - CURIS * ( exp( VBS/VTH ) - 1.0 ) , 0.0)  
> # -----  
> # Function evaluating the current of the pn-junction between bulk and  
> # drain due to the model of Shichman and Hodges  
> # -----  
>  
> IBD <- function (VBD)      # VBD Voltage between bulk and drain  
+       ifelse ( VBD <= 0.0, - CURIS * ( exp( VBD/VTH ) - 1.0 ) , 0.0)  
> # -----  
> # Evaluating input signal at time point X  
> # -----  
>  
> PULSE <- function (X,      # Time-point at which input signal is evaluated  
+                     LOW,    # Low-level of input signal  
+                     HIGH,   # High-level of input signal  
+                     DELAY,T1,T2,T3,PERIOD) # Parameters to specify signal structure  
+ # -----  
+ # Structure of input signal:  
+ #  
+ #  
  
+ # -----  
+ {  
+     TIME = X%%PERIOD  
+     VIN = LOW  
+     VIND = 0.0  
+  
+     if (TIME > (DELAY+T1+T2))  
+     {  
+         VIN = ((HIGH-LOW)/T3)*(DELAY+T1+T2+T3-TIME) + LOW  
+         VIND= -((HIGH-LOW)/T3) } else  
+     if (TIME > (DELAY+T1))
```



```

+      {
+        VIN = HIGH
+        VIND= 0.0          } else
+      if (TIME > DELAY)
+      {
+        VIN = ((HIGH-LOW)/T1)*(TIME-DELAY) + LOW
+        VIND= ((HIGH-LOW)/T1)  }
+
+      return (list(VIN=VIN, # Voltage of input signal at time point X
+                   VIND=VIND)) # Derivative of VIN at time point X
+    }
+  }
+ # -----
+ # Function evaluating the voltage-dependent capacitance between bulk and
+ # drain gevalp. source due to the model of Shichman and Hodges
+ # -----
+
+ > CBDBS <- function (V) # Voltage between bulk and drain gevalp. source
+   ifelse ( V <= 0.0 ,CBD/sqrt(1.0-V/0.87),CBD*(1.0+V/(2.0*0.87)))

```

The parameter values are:

```

> RGS    = 4
> RGD    = 4
> RBS    = 10
> RBD    = 10
> CGS    = 0.6e-4
> CGD    = 0.6e-4
> CBD    = 2.4e-5
> CBS    = 2.4e-5
> C9     = 0.5e-4
> DELTA  = 0.2e-1
> CURIS  = 1.e-14
> VTH    = 25.85
> VDD    = 5.
> VBB    = -2.5

```

and a consistent set of initial conditions:

```

> VBB    <- -2.5
> Y      <- c(5,5,VBB,VBB,5,3.62385,5,VBB,VBB,3.62385,0,3.62385,VBB,VBB)
> Yprime <- rep(0,14)

```

After allocating memory to used matrices and vectors, and specifying the times for which output is wanted, the model is run; given the large number of equations, and the fact that it is implemented in an interpreted language, this takes a while...

```

> CAP    <- matrix(nrow=14,ncol=14,0)
> F      <- vector("double",14)

```

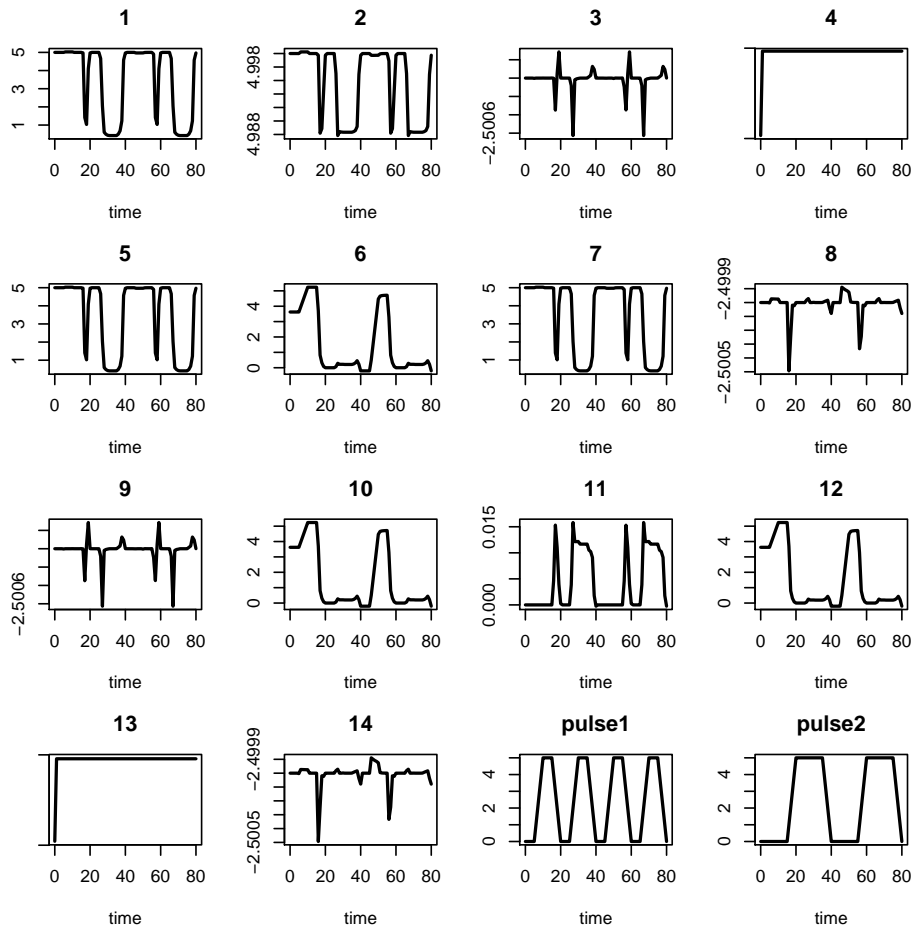


Figure 8: Solution of the NAND gate an index 1 implicit differential equation model of dimension 14 - see text for R-code

```
> times <- seq(0,80,by=1)
> print(system.time(
+ out<- mebdfi(y=Y,dy=Yprime,times,res=Nand,parms=0,
+             hini=1e-6,rtol=1e-6,atol=1e-6)
+ ))
```

```
user  system elapsed
34.14   0.00   34.34
```

The output is arranged in 4 rows and 4 columns (`mfrow=c(4,4)`)

```
> par(mar=c(4,2,3,2))
> plot(out, mfrow=c(4,4), type="l", lwd=2, ask=FALSE)
```

11. wheelset

The final example is an Implicit Differential Equation of index 2 and dimension 17. It describes motion of a simple wheelset on a rail track.

The model is written in fortran and included as a DLL in package deTestSet.

First the index of the variables, the initial conditions and the parameter values are specified:

```
> ind <- c(15,2,0)

> yini <- c( x=0.14941e-02, y=0.40089e-06, z=0.11241e-05,
+           theta=-.28573e-03, phi=0.26459e-03,
+           v1=0, v2=0, v3=0, v4=0, v5=0, beta=0,
+           q1=-7.4122380357667139e-06, q2=-0.1521364296121248,
+           q3=7.5634406395172940e-06, q4=0.1490635714733819,
+           lam1=-8.3593e-3, lam2=-7.4144e-3)
> yprime <- c(0,0,0,0,0,-1.975258894011285,-1.0898297102811276e-03,
+            7.8855083626142589e-02,-5.533362821731549,-0.3487021489546511,
+            -2.132968724380927,0,0,0,0,0,0)
> parameter <- c(MR = 16.08, G = 9.81, V = 30., RNO = 0.1, LI1 = 0.0605,
+               LI2 = 0.366, MA = 0.0, HA = 0.2, MU = 0.12, XL = 0.19,
+               CX = 6400., CZ = 6400.,
+               E = 1.3537956, GG = 0.7115218, SIGMA = 0.28, GM = 7.92e10,
+               C11 = 4.72772197, C22 = 4.27526987, C23 = 1.97203505,
+               DELTA0 = 0.0262, AR = 0.1506, RS = 0.06, EPS = 0.00001,
+               B1 = 0.0, B2 = 4.0)
```

... and it is tested whether the initial conditions are consistent:

```
> DLLres(res="wheelres", time=0., y=yini, dy=yprime,
+        initfunc="wheelpar", parms=parameter, dllname="deTestSet")
```

\$delta

	x	y	z	theta	phi
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
	v1	v2	v3	v4	v5
8.364143e-14	-1.033097e-14	-6.092913e-14	-1.230635e-14	8.523089e-15	
	beta	q1	q2	q3	q4
6.083296e-15	0.000000e+00	0.000000e+00	9.790710e-17	-3.358353e-22	
	lam1	lam2			
-6.691315e-16	-2.026593e-22				

\$var

[1] NA

The model tolerances are set, specifying larger values for the last two variables, and the model run:

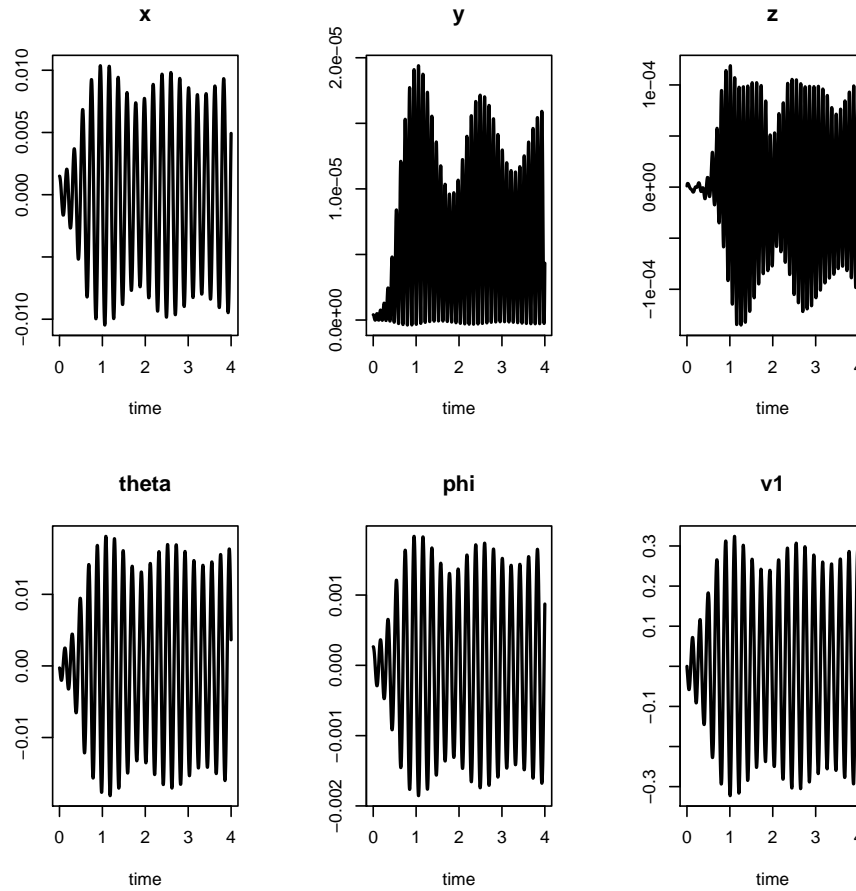


Figure 9: Solution of wheelset an implicit differential equation model - index 2, dimension 17
- see text for R-code

```
> atol      <- rep(1e-6,17)
> atol[16:17] <- 1e10
> rtol      <- rep(1e-6,17)
> rtol[16:17] <- 1e10

> times <- seq(0, 4, by=0.001)
> Wheel <- mebdfi(y=yini, dy=yprime, times=times, res="wheelres", nind=ind,
+               dllname="deTestSet", initfunc="wheelpar", parms=parameter,
+               atol=atol, rtol=rtol, maxsteps=100000)

> plot(Wheel, which=1:6, type="l", lwd=2, ask=FALSE)
```

12. References

- J.R. Cash (1979) Stable recursions with applications to the numerical solution of stiff systems, Academic Press.
- J. R. Cash (1983) The integration of stiff initial value problems in O.D.E.S using modified extended backward differentiation formulae, *Comp. and Maths. with applics.*, 9, 645-657, (1983).
- J.R. Cash and S. Considine, (1992). An MEBDF code for stiff initial value problems, *ACM Trans Math Software*, 142-158,
- F. Mazzia and C. Magherini. Test Set for Initial Value Problem Solvers, release 2.4. Department of Mathematics, University of Bari and INdAM, Research Unit of Bari, February 2008. Available at <http://www.dm.uniba.it/~testset>.
- Karine Soetaert, Francesca Mazzia, Jeff Cash (2010). *deTestSet: Solvers and Test Set for Initial Value Problems of Ordinary Differential Equations (ODE), Partial Differential Equations (PDE) and for Differential Algebraic Equations (DAE)*. R package version 0.0.
- Karine Soetaert, Thomas Petzoldt and R. Woodrow Setzer (2010). *deSolve: General solvers for initial value problems of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE) and delay differential equations*. R package version 1.7.
- Karine Soetaert, Jeff Cash and Francesca Mazzia (2009). *bvpSolve: Solvers for boundary value problems of ordinary differential equations*. R package version 1.1.
- Karine Soetaert (2009). *rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations*. R-package version 1.6

Affiliation:

Karine Soetaert
Centre for Estuarine and Marine Ecology (CEME)
Netherlands Institute of Ecology (NIOO)
4401 NT Yerseke, Netherlands
E-mail: k.soetaert@nioo.knaw.nl
URL: <http://www.nioo.knaw.nl/users/ksoetaert>

Jeff Cash
Imperial College London
South Kensington Campus
London SW7 2AZ, U.K.
E-mail: j.cash@imperial.ac.uk
URL: <http://www.ma.ic.ac.uk/~jcash>

Francesca Mazzia
Dipartimento di Matematica
Universita' di Bari
Via Orabona 4,

70125 BARI

Italy E-mail: mazzia@dm.uniba.it

URL: <http://pitagora.dm.uniba.it/~mazzia>