Package bvpSolve, solving boundary value problems in R

Karline Soetaert
Centre for Estuarine and Marine Ecology
Netherlands Institute of Ecology
The Netherlands

Abstract

This document is about package **bvpSolve** (Soetaert 2009a), designed for the numerical solution of boundary value problems for ordinary differential equations (ODE) in R .
Package **bvpSolve** contains:

- function `bvpshoot` which implements the shooting method. This method makes use of the initial value problem solvers from packages **deSolve** (Soetaert, Petzoldt, and Setzer 2009) and the root-finding solver from package **rootSolve** (Soetaert 2009b).

- function `bvptwp`, the mono-implicit Runge-Kutta (MIRK) method with deferred corrections, code TWPBVP (Cash and Wright 1991), for solving two-point boundary value problems

The R functions have an interface which is similar to the interface of the solvers in package **deSolve**

Keywords: ordinary differential equations, boundary value problems, shooting method, mono-implicit Runge-Kutta, R.

1. Introduction

bvpSolve numerically solves boundary value problems (BVP) of ordinary differential equations (ODE), which for one (second-order) ODE can be written as:

$$\frac{d^2y}{dx^2} = f(x, y, \frac{dy}{dx})$$
$$a \le x \le b$$
$$g_1(y)|_a = 0$$
$$g_2(y)|_b = 0$$

where y is the dependent, x the independent variable, function f is the differential equation, g_a and g_b the boundary conditions at the end points a and b.

The problem must be specified as a first-order system. Thus, higher-order ODEs need to be rewritten as a set of first-order systems. For instance:

$$\frac{d^2y}{dx^2} = f(x, y, \frac{dy}{dx})$$

can be rewritten as:

$$\frac{dy}{dx} = z$$

$$\frac{dz}{dx} = f(x, y, z)$$

Note that in the current implementation, the boundary conditions must be defined at the end of the interval over which the ODE is specified (i.e. at `a` and/or `b`).

More examples of boundary value problems can be found in the packages `examples` subdirector. They include a.o. all problems found in `http://www.ma.ic.ac.uk/~jcash/BVP_software`.

2. A simple BVP example

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

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

The second-order ODE is expanded as two first-order ODEs as:

$$
\begin{aligned}
y_1' &= y_2 \\
y_2' &= 1/\xi \cdot (-xy_2 + y_1 - (1 + \xi\pi^2)\cos(\pi x) - \pi x \sin(\pi x))
\end{aligned}
$$

with boundary conditions

$$
\begin{aligned}
y_1(-1) &= -1 \\
y_1(1) &= 1
\end{aligned}
$$

This is implemented as:

```
> fun<- function(x,y,pars)
+ {
+  list(c(y[2],
+    1/ks*(-x*y[2]+y[1]-(1+ks*pi*pi)*cos(pi*x)-pi*x*sin(pi*x)))
+      )
+ }
```

and solved, using the two methods, as:

```
> ks <- 0.1
> x  <- seq(-1,1,by=0.01)
> print(system.time(
+ sol1  <- bvpshoot(yini=c(-1,NA),yend=c(1,NA),x=x,func=fun,guess=0)
+ ))

   user  system elapsed
   0.06    0.00    0.07


> print(system.time(
+ sol2  <- bvptwp(yini=c(-1,NA),yend=c(1,NA),x=x,func=fun, guess=0)
+ ))

   user  system elapsed
   0.14    0.00    0.14
```
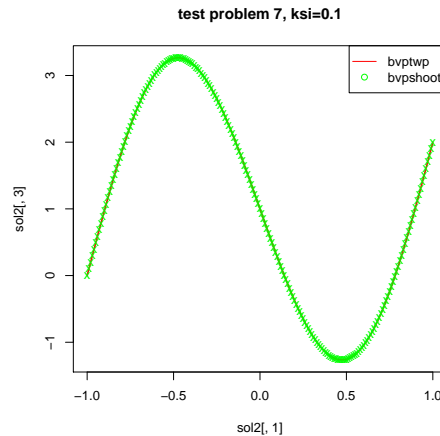
Figure 1: Solution of the simple BVP, for ksi=0.1 - see text for R -code

Note how the boundary conditions at the start (`yini`) and end `yend` of the integration interval are specified, where `NA` is used for boundary conditions that are not known.

A reasonable `guess` of the unknown initial condition is also inputted.

As is often the case, the shooting method is faster than the other method. However, there are particular problems where `bvpshoot` does not lead to a solution, whereas the MIRK method does (see below).

The plot shows that the two methods give the same solution:

```
> plot(sol2[,1],sol2[,3],type="l",main="test problem 7, ksi=0.1",
+       lwd=2,col="red")
> points(sol1[,1],sol1[,3],col="green",pch="x")
> legend("topright",c("bvptwp", "bvpshoot"),
+          lty=c(1,NA,NA), pch=c(NA,1,3),col=c("red", "green") )
```

When the parameter ξ is decreased, `bvpshoot` cannot solve the problem anymore, due to the presence of a zone of rapid change near x=0.

However, it can still easily be solved with the MIRK method:

```
> ks <-0.0001
> print(system.time(
+ sol2  <- bvptwp(yini=c(-1,NA),yend=c(1,NA),x=seq(-1,1,by=0.01),
+           func=fun, guess=0)
+ ))

   user   system elapsed
    1.8      0.0     1.8


> plot(sol2[,1],sol2[,3],type="l",main="test problem 7, ksi=0.0001",
+       lwd=2,col="red")
```
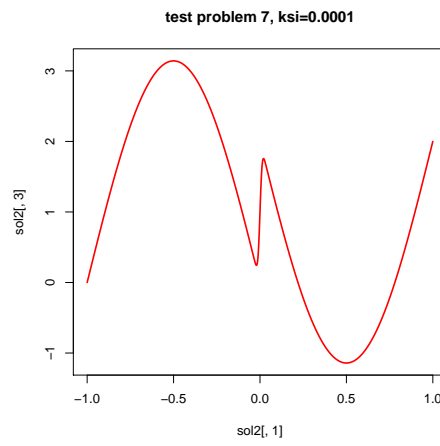
Figure 2: Solution of the simple BVP, for ksi=0.0001 - see text for R -code. Note that this problem cannot be solved with `bvpshoot`

3. A more complex BVP example

Here the test problem referred to as "swirling flow III" is solved (Ascher, Mattheij, and Russell 1995).

The original problem definition is:

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

on the interval [0,1] and subject to boundary conditions:

$$
\begin{aligned}
g(0) &= -1, f(0) = 0, f'(0) = 0 \\
g(1) &= 1, f(1) = 0, f'(1) = 0
\end{aligned}
$$

This is rewritten as a set of 1st order ODEs as follows:

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

Its implementation in R is:

```
> fsub <- function (t,Y,pars)
+ { return(list(c(f1 = Y[2],
+                 f2 = (Y[1]*Y[4] - Y[3]*Y[2])/eps,
+                   f3 = Y[4],
+                   f4 = Y[5],
+                   f5 = Y[6],
+                   f6 = (-Y[3]*Y[6] - Y[1]*Y[2])/eps)))
+ }
> eps <- 0.001
> x <- seq(0,1,len=100)
```

This model cannot be solved with the shooting method. However, it can be solved using bvptwp:

```
> print(system.time(
+ Soltwp <- bvptwp(x=x,func=fsub,guess= c(2,0,0),
+                     yini=c(y1=-1,y2=NA,y3=0,y4=0,y5=NA,y6=NA),
+                     yend=c(1,NA,0,0,NA,NA))
+ ))

   user  system elapsed
   2.26    0.00    2.26
```
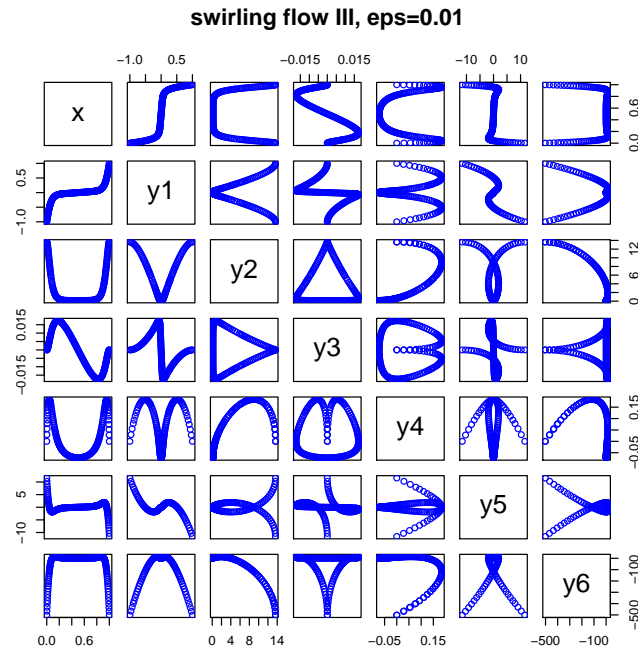
Figure 3: pairs plot of the swirling flow III problem - see text for R -code. Note that this problem cannot be solved with `bvpshoot`

where the reported system time is in seconds

The problem cannot be solved with too small values of eps:

```
> eps <- 1e-9
> Soltwp2 <- NA
> Soltwp2 <- try(bvptwp(x=x,func=fsub,guess= c(2,0,0),
+                     yini=c(y1=-1,y2=NA,y3=0,y4=0,y5=NA,y6=NA),
+                     yend=c(1,NA,0,0,NA,NA)),
+     silent = TRUE)
> cat(Soltwp2)

Error in bvptwp(x = x, func = fsub, guess = c(2, 0, 0), yini = c(y1 = -1,  :
  The Expected No. Of Subintervals Exceeds Storage Specifications.

> pairs(Soltwp, main="swirling flow III, eps=0.01", col="blue")
```
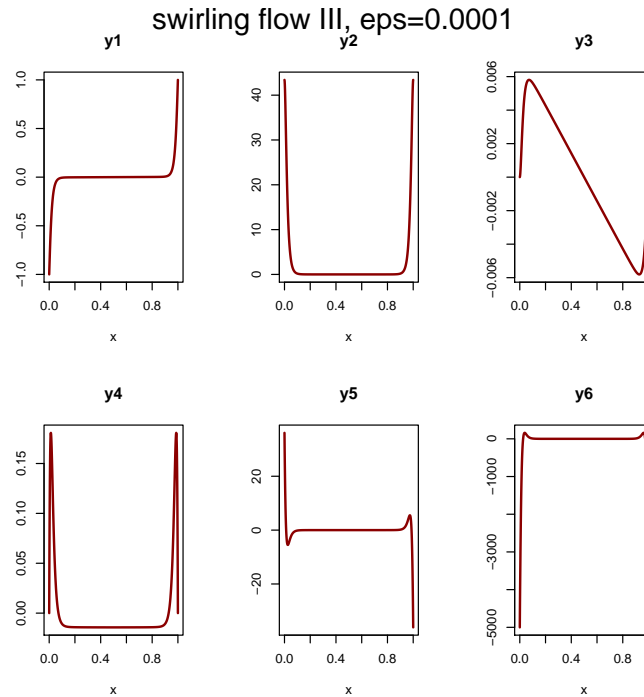
Figure 4: Solution of the swirling flow III problem with small eps, using continuation - see text for R -code.

4. Solving a boundary value problem using continuation

The previous -swirl- problem can be solved for small values of `eps` if the previous solution (`Soltwp`) with eps = 0.001, is used as an initial guess for smaller value of eps, 0.0001:

```
> eps <- 0.0001
> xguess <- Soltwp[,1]
> yguess <- t(Soltwp[,2:7])
> print(system.time(Sol2 <- bvptwp(x=x,func=fsub,guess= c(2,0,0),
+    xguess=xguess,yguess=yguess,yini=c(y1=-1,y2=NA,y3=0,y4=0,y5=NA,y6=NA),
+    yend=c(1,NA,0,0,NA,NA))))

   user  system elapsed
   3.46    0.00    3.46


> plot(Sol2, col="darkred", type="l", lwd=2)
> mtext(outer=TRUE, side=3, line=-1.5, cex=1.5,
+    "swirling flow III, eps=0.0001")
```

5. More complex initial or end conditions

Problem `musn` was described in (Ascher *et al.* 1995).

The problem is:

$$
\begin{aligned}
u' &= 0.5u(w-u)/v \\
v' &= -0.5(w-u) \\
w' &= (0.9 - 1000(w-y) - 0.5w(w-u))/z \\
z' &= 0.5(w-u) \\
y' &= -100(y-w)
\end{aligned}
$$

on the interval [0,1] and subject to boundary conditions:

$$
\begin{aligned}
u(0) = v(0) = w(0) &= 1 \\
z(0) &= -10 \\
w(1) &= y(1)
\end{aligned}
$$

Note the last boundary conditions which expresses `w` as a function of `y`.

Implementation of the ODE function is simple:

```
> musn <- function(x,Y,pars)
+ {
+   with (as.list(Y),
+   {
+     du=0.5*u*(w-u)/v
+     dv=-0.5*(w-u)
+     dw=(0.9-1000*(w-y)-0.5*w*(w-u))/z
+     dz=0.5*(w-u)
+     dy=-100*(y-w)
+     return(list(c(du,dv,dw,dz,dy)))
+   })
+ }
```

There are 4 boundary values specified at the start of the interval; a value for `y` is lacking:

```
> init <- c(u=1,v=1,w=1,z=-10,y=NA)
```

The boundary condition at the end of the integration interval (1) specifies the value of `w` as a function of `y`.

Because of that, `yend` cannot be simply inputted as a vector. It is rather implemented as a function that has as input the values at the end of the integration interval (`Y`), the values at the start (`yini`) and the parameters, and that returns the residual function (`w-y`):

```
> yend  <- function (Y,yini,pars)  with (as.list(Y), w-y)
```

This problem is most efficiently solved with `bvpshoot`: [1]

---

[1]Note that there are at least two solutions to this problem, the second solution can simply be found by setting `guess` equal to 0.9.
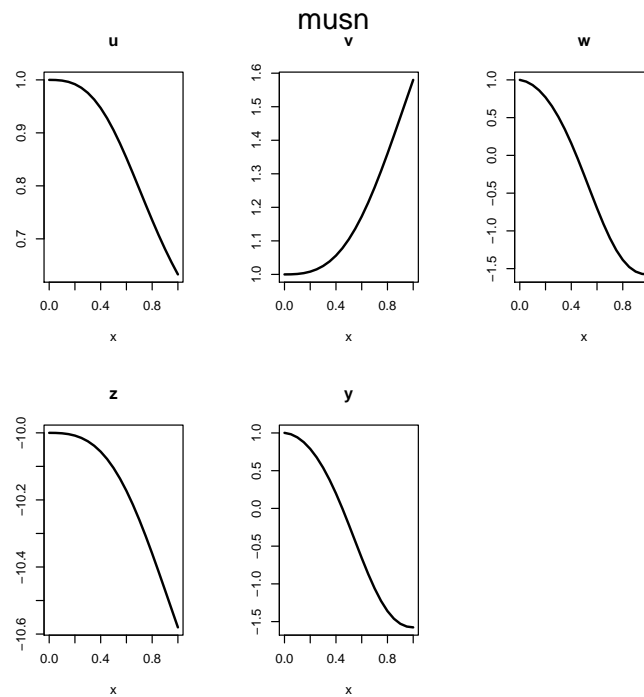
Figure 5: Solution of the musn model, using `bvpshoot` - see text for R -code.

```
> print(system.time(
+ sol    <-bvpshoot(yini= init, x=seq(0,1,by=0.05),func=musn,
+           yend=yend,guess=1,atol=1e-10,rtol=0)
+ ))

   user   system elapsed
   0.33     0.00    0.33

> plot(sol,type="l", lwd=2)
> mtext(outer=TRUE, side=3, line=-1.5, cex=1.5, "musn")
```

6. a BVP problem including an unknown parameter

In the next BVP problem, a parameter λ is to be found such that:

$$\frac{d^2y}{dt^2} + (\lambda - 10\cos(2t)) \cdot y = 0$$

on $[0,\pi]$ with boundary conditions $\frac{dy}{dt}(0) = 0$ and $\frac{dy}{dt}(\pi) = 0$ and $y(0) = 1$

Here all the initial values (at t=0) are prescribed. If λ would be known the problem would be overdetermined.

The 2nd order differential equation is first rewritten as two 1st-order equations:

$$
\begin{aligned}
\frac{dy}{dt} &= y2 \\
\frac{dy2}{dt} &= -(\lambda - 10\cos(2t)) \cdot y
\end{aligned}
$$

and the function that estimates these derivatives is written (`derivs`).

```
> mathieu <- function(x,y,lambda)
+     list(c(y[2],
+         -(lambda-10*cos(2*x))*y[1]))
```

which is easily solved using `bvpshoot`:

```
> init <- c(1,0)
> sol  <- bvpshoot(yini=init,yend=c(NA,0),x=seq(0,pi,by=0.01),
+         func=mathieu, guess=NULL, extra=15)
```

and plotted:

```
> plot(sol[,1:2])
> mtext(outer=TRUE, side=3, line=-1.5, cex=1.5, "elastica")
```

Note how the extra parameter to be fitted is passed (`extra`). The value of `lam` can be printed:

```
> attr(sol,"roots")  # root gives the value of "lam" (17.10684)

      root       f.root iter
2 17.10683 2.347269e-12    6
```
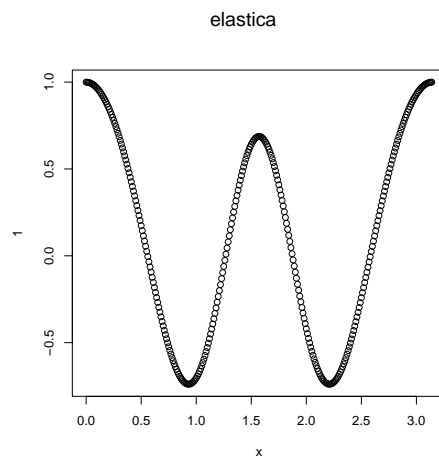
Figure 6: Solution of the BVP ODE problem including an unknown parameter, see text for R-code
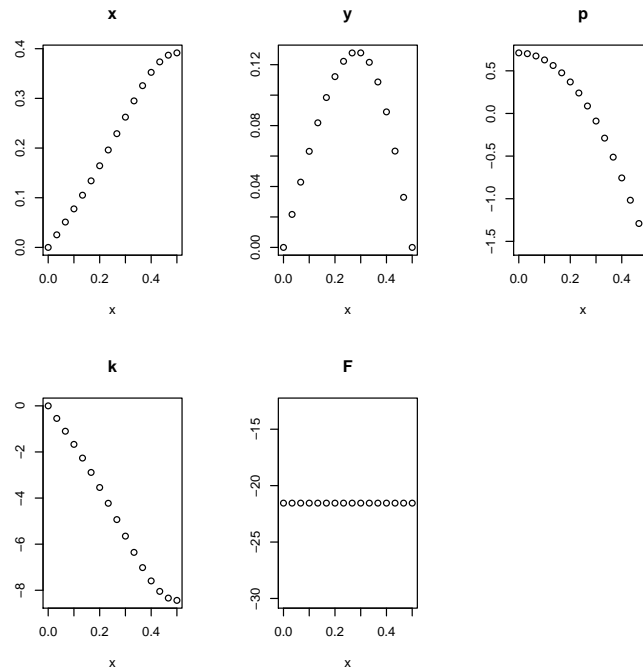
Figure 7: Solution of the elastica problem - see text for R-code

7. Specifying the analytic jacobians

By default, the Jacobian of the derivative function and of the boundary conditions, are estimated numerically. It is however possible - and faster - to provide the analytical solution of the jacobian.

As an example, the `elastica` problem is implemented. First, using the default specification:

```
> Elastica <- function (x, y, pars) {
+
+   list( c(cos(y[3]),
+           sin(y[3]),
+           y[4],
+           y[5]*cos(y[3]),
+           0))
+ }
> Sol <- bvptwp(func=Elastica,
+               yini = c(x=0, y=0, p=NA,    k=0, F=NA),
+               yend = c(x=NA,y=0, p=-pi/2,k=NA,F=NA),
+               x = seq(0,0.5,len=16),
+               guess=c(0,0) )

> plot(Sol)
```

Now several extra functions are defined, specifying

1. the analytic Jacobian for the derivative function (`jacfunc`)

2. the boundary function (`bound`). Here `i` is the boundary condition "number". The conditions at the left are enumerated first, then the ones at the right. For instance, i = 1 specifies the boundary for y(0) = 0, or BC(1) = y[1]-0; the fifth boundary condition is y[3] = -pi/2 or BC = y[3] + pi/2

3. the analytic Jacobian for the boundary function (`jacbound`)

This is done in the R -code below:

```
> Jac <- matrix(nr=5,nc=5,0)
> Jac[3,4]=1.0
> Jac[4,4]=1.0
> jacfunc <- function (x, y, pars) {
+        Jac[1,3]=-sin(y[3])
+        Jac[2,3]=cos(y[3])
+        Jac[4,3]=-y[5]*sin(y[3])
+        Jac[4,5]=Jac[2,3]
+        Jac
+ }
> bound <- function (i, y, pars)  {
+     if (i <=2) return(y[i])
+     else if (i == 3) return(y[4])
+     else if (i == 4) return(y[2])
+     else if (i == 5) return(y[3]+pi/2)
+ }
> jacbound <- function(i, y, pars)  {
+     JJ <- rep(0,5)
+         if (i <=2) JJ[i] =1.0
+     else if (i ==3) JJ[4] =1.0
+     else if (i ==4) JJ[2] =1.0
+     else if (i ==5) JJ[3] =1.0
+     JJ
+ }
```

If this input is used, the number of left boundary conditions `leftbc` needs to be specified.

```
> Sol4 <- bvptwp(leftbc = 3,
+             func=Elastica, jacfunc = dfsub,
+             bound = bound, jacbound=jacbound,
+             x = seq(0,0.5,len=16),
+             guess=c(0,0) )
```

Solving the model this way is about 3 times faster than the default.

8. implementing a BVP problem in compiled code

Even more computing time is saved by specifying the problem in lower-level languages such as FORTRAN or C, or C++, which are compiled into a dynamically linked library (DLL) and loaded into R .

This is similar as the differential equations from package **deSolve** (Soetaert *et al.* 2009).

Its vignette ("compiledCode") can be consulted for more information.

In order to create compiled models (.DLL = dynamic link libraries on Windows or .so = shared objects on other systems) you must have a recent version of the GNU compiler suite installed, which is quite standard for Linux.

Windows users find all the required tools on http://www.murdoch-sutherland.com/Rtools/. Getting DLLs produced by other compilers to communicate with R is much more complicated and therefore not recommended. More details can be found on http://cran.r-project.org/doc/manuals/R-admin.html.

The call to the derivative, boundary and Jacobian functions is more complex for compiled code compared to R -code, because it has to comply with the interface needed by the integrator source codes.

Below is an implementation of the elastica model in fortran:

```fortran
c  The differential system coded up in a suitable form is as follows:

      SUBROUTINE fsub(NCOMP,X,Z,F,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER NCOMP, IPAR  , I
      DOUBLE PRECISION F, Z, RPAR, X
      DIMENSION Z(*),F(*)
      DIMENSION RPAR(*), IPAR(*)

      F(1)=cos(Z(3))
      F(2)=sin(Z(3))
      F(3)=Z(4)
      F(4)=Z(5)*cos(Z(3))
      F(5)=0

      RETURN
      END


c The analytic Jacobian for the F-function:

      SUBROUTINE dfsub(NCOMP,X,Z,DF,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER NCOMP, IPAR, I, J
      DOUBLE PRECISION X, Z, DF, RPAR
      DIMENSION Z(*),DF(NCOMP,*)
      DIMENSION RPAR(*), IPAR(*)
      CHARACTER (len=50) str
```

```
      DO I=1,5
         DO J=1,5
            DF(I,J)=0.D0
         END DO
      END DO

      DF(1,3)=-sin(Z(3))
      DF(2,3)=cos(Z(3))
      DF(3,4)=1.0D0
      DF(4,3)=-Z(5)*sin(Z(3))
      DF(4,4)=1.0D0
      DF(4,5)=cos(Z(3))

      RETURN
      END


c The boundary conditions:

      SUBROUTINE gsub(I,NCOMP,Z,G,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER I, NCOMP, IPAR
      DOUBLE PRECISION Z, RPAR, G
      DIMENSION Z(*)
      DIMENSION RPAR(*), IPAR(*)



      IF (I.EQ.1) G=Z(1)
      IF (I.EQ.2) G=Z(2)
      IF (I.EQ.3) G=Z(4)
      IF (I.EQ.4) G=Z(2)
      IF (I.EQ.5) G=Z(3)+1.5707963267948966192313216916397514D0

      RETURN
      END


c The analytic Jacobian for the boundaries:

      SUBROUTINE dgsub(I,NCOMP,Z,DG,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER I, NCOMP, IPAR
      DOUBLE PRECISION Z, DG, RPAR
      DIMENSION Z(*),DG(*)
      DIMENSION RPAR(*), IPAR(*)

      DG(1)=0.D0
      DG(2)=0.D0
```

```
      DG(3)=0.D0
      DG(4)=0.D0
      DG(5)=0.D0

C     dG1/dZ1
      IF (I.EQ.1) DG(1)=1.D0
C     dG2/dZ2
      IF (I.EQ.2) DG(2)=1.D0
C     dG3/dZ4
      IF (I.EQ.3) DG(4)=1.D0
C     dG4/dZ2
      IF (I.EQ.4) DG(2)=1.D0
C     dG5/dZ3
      IF (I.EQ.5) DG(3)=1.D0

      RETURN
      END
```

The same model, implemented in C is:

```c
#include <math.h>

//  The differential system:
  void fsub(int *n, double *x, double *z, double *f,
       double * RPAR, int * IPAR)  {

      f[0]=cos(z[2]);
      f[1]=sin(z[2]);
      f[2]=z[3]      ;
      f[3]=z[4]*cos(z[2]);
      f[4]=0;
  }

// The analytic Jacobian for the F-function:
  void dfsub(int * n, double *x, double *z, double * df,
      double *RPAR, int *IPAR)  {

      int j;
      for (j = 0; j< *n * *n; j++) df[j] = 0;

      df[*n *2]    = -sin(z[2]);
      df[*n *2 +1] = cos(z[2]);
      df[*n *3 +2] = 1.0;
      df[*n *2 +3] = -z[4]*sin(z[2]);
      df[*n *3 +3] = 1.0;
      df[*n *4 +3] = cos(z[2]);
```

```
  }

// The boundary conditions:
  void gsub(int *i, int *n, double *z, double *g,
      double *RPAR, int *IPAR)  {

      if (*i==1) *g=z[0];
      else if (*i==2) *g=z[1];
      else if (*i==3) *g=z[3];
      else if (*i==4) *g=z[1];
      else if (*i==5) *g=z[2]+1.5707963267948966192313216916397514;
  }

// The analytic Jacobian for the G-function:
  void dgsub(int *i, int *n, double *z, double *dg,
      double *RPAR, int *IPAR)  {

      int j;
      for (j = 0; j< *n; j++) dg[j] = 0;

      if (*i == 1) dg[0] = 1.;
      else if (*i == 2) dg[1] = 1.;
      else if (*i == 3) dg[3] = 1.;
      else if (*i == 4) dg[1] = 1.;
      else if (*i == 5) dg[2] = 1.;
  }
```

In what follows, it is assumed that the codes are saved in a file called `elastica.f`, and `elasticaC.c` and that these files are in the working directory of R . (if not, use `setwd()`)

Before the functions can be executed, the fortran or C- code has to be compiled

This can simply be done in R:

```
 system("R CMD SHLIB elastica.f")
 system("R CMD SHLIB elasticaC.c")
```

or

```
system("gfortran -shared -o elastica.dll elastica.f")
system("gcc -shared -o elasticaC.dll elasticaC.c")
```

This will create a file called `elastica.dll` and `elasticaC.dll` respectively (on windows).

After loading the DLL, the model can be run, after which the dll is unloaded. For the Fortran, this is done as follows (the C code is similar, except for the name of the DLL):

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

outF <- bvptwp(ncomp=5,
               x = seq(0,0.5,len=16), leftbc = 3,
               func="fsub",jacfunc="dfsub",bound="gsub",jacbound="dgsub",
               dllname="elastica")

dyn.unload("elastica.dll")
```

Note that the number of components (equations) needs to be explicitly inputted (`ncomp`).

This model is about 8-10 times faster than the pure R implementation from previous section.

The solver recognizes that the model is specified as a DLL due to the fact that arguments `func`, `jacfunc`, `bound` and `jacbound` are not regular R -functions but character strings.

Thus, the solver will check whether these functions are loaded in the DLL with name "elastica.dll". Note that the name of the DLL should be specified without extension.

This DLL should contain all the compiled function or subroutine definitions needed.

Also, if `func` is specified in compiled code, then `jacfunc`, `bound` and `jacbound` should also be specified in a compiled language. It is not allowed to mix R-functions and compiled functions.

References

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

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

Soetaert K (2009a). *bvpSolve: solvers for boundary value problems of ordinary differential equations.* R package version 1.0.

Soetaert K (2009b). *rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations.* R package version 1.4.

Soetaert K, Petzoldt T, Setzer RW (2009). *deSolve: General solvers for initial value problems of ordinary differential equations (ODE), partial differential equations (PDE) and differential algebraic equations (DAE).* R package version 1.3.

Affiliation:

Karline 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