Package bvpSolve, solving boundary value problems in R

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

Jeff Cash
Department of mathematics
Imperial College London
U.K.

Francesca Mazzia
Dipartimento di Matematica
Universita' di Bari
Italy

Abstract

package **bvpSolve** (Soetaert, Cash, and Mazzia 2010a), in the open-source software R , (R Development Core Team 2010) is designed for the numerical solution of boundary value problems (BVP) for ordinary differential equations (ODE).

It comprizes:

- function `bvpshoot` which implements the shooting method. This method makes use of the initial value problem solvers from packages **deSolve** (Soetaert, Petzoldt, and Setzer 2010b) and the root-finding solver from package **rootSolve** (Soetaert 2009).
- function `bvptwp`, the mono-implicit Runge-Kutta (MIRK) method with deferred corrections, using conditioning in the mesh selection, based on FORTRAN code TWPBVPC (Cash and Wright 1991; Cash and Mazzia 2005), for solving two-point boundary value problems
- function `bvpcol`, the 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

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

The default input to the solvers is very simple, requiring specification of only one function, that calculates the derivatives, while the boundary conditions are represented as simple vectors.

However, in order to speed-up the simulations, and to increase the number of problems that can be solved, it is also possible to specify the boundary conditions by means of a function and provide analytical solutions for the derivative and boundary gradients.

This is one of two vignette of package **rootSolve** .

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

1. Introduction

1.1. The Boundary Value Problem Solvers

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

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

where `y` is the dependent, `x` the independent variable, function `f` is the differential equation, and $g_1(y)|_a$ and $g_2(y)|_b$ the boundary conditions at the end points a and b.

1.2. Package bvpSolve

Three BVP solvers are included in **bvpSolve** :

- `bvpshoot`, implementing the shooting method. This method combines solutions of initial value problems (IVP) with solutions of nonlinear algebraic equations; it makes use of solvers from packages **deSolve** and **rootSolve** .
- `bvptwp`, a mono-implicit Runge-Kutta (MIRK) method with deferred corrections, and using conditioning in the mesh selection, based on FORTRAN code TWPBVPC (Cash and Wright 1991; Cash and Mazzia 2005).
- `bvpcol`, a collocation method based on FORTRAN codes COLNEW (Bader and Ascher 1987), and COLSYS (Ascher *et al.* 1979) for solving Multi-point boundary value problems of mixed order.

An S3 method for plotting is included. This will plot all variables in separate figures.

All functions can solve higher-order ODEs without writing them as a set of first-order ODEs, but this makes the interface a bit more difficult.

Only `bvpcol` is more efficient if the higher-order input is used.

For functions `bvpshoot` and `bvptwp` it is slightly more efficient to write them as *first-order* ODEs.

For instance:

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

can be rewritten as:

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

where `y` and `z` are now the two dependent variables.

In `bvptwp` and `bvpshoot`, 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`). In contrast, `bvpcol` can also have the boundary conditions specified at intermediate points.

1.3. Simple and More Complex Input

When using `bvptwp` and `bvpcol`, the problem can be specified in several ways.

- By default, the partial derivatives of the differential equations and of the boundary conditions are approximated by the solver using *finite differences*. Then, the user need not be concerned with supplying functions that calculate the analytical partial derivatives. This makes the definition of the problem very simple: only one function, estimating the derivatives needs to be provided, while the boundary conditions are specified as vectors. However, some problems cannot be specified this way. It is also the slowest method.

- The `bvptwp` and `bvpcol` function is much more efficient if *analytical* partial derivatives of the differential equations and of boundary conditions are given.

- Even more simulation time will be gained if the problem is specified in *compiled code* (FORTRAN, C). In this case, R is used to trigger the solver `bvptwp` or `bvpcol`, and for post-processing (graphics), while solving the BVP itself entirely takes place in compiled code.

1.4. Examples in This Vignette

In this package vignette it is shown how to formulate and solve BVPs. We use well-known test cases as examples.

- We start with a simple example, comprising one second-order ODE (test problem 7 from the website of Jeff Cash (`http://wwwf.imperial.ac.uk/~jcash/BVP_software/PROBLEMS.PDF`)

- This is followed by a more complex example, which consists of 6 first-order ODEs, the "swirling flow III" problem (Ascher, Mattheij, and Russell 1995). This example is used to demonstrate how to continuate a solution, i.e. use the solution for one problem as initial guess for solving another, more complex problem.

- How to implement more complex initial conditions is then exemplified by means of problem "musn" (Ascher *et al.* 1995).

- Next, solving for the fourth eigenvalue of "Mathieu's equation" (Shampine, Kierzenka, and Reichelt 2000), illustrates how to solve a BVP including an unknown parameter.

- The "nerve impulse" model (Seydel 1988), is an example including periodic boundary conditions.

- The "fluid injection" problem (Ascher *et al.* 1995), is a set of higher-order ODEs, which is best solved using `bvpcol`.

- A simple multipoint examplis is solved using `bvpcol`.

- The "elastica" problem (Jeff Cash's website) is used to demonstrate how to specify the analytic jacobians, and how to implement problems in FORTRAN or C.

- Finally, a standard linear testcase (Shampine *et al.* 2000) which has a steep boundary layer is implemented in FORTRAN, and run with several values of a model parameter.

More examples of boundary value problems can be found in the packages `examples` subdirectory.

The `dynload` subdirectory includes models specified in compiled code.

See also document "bvpSolve: a set of 35 test Problems", which can be accessed as `vignette("bvpTests")` or is available from the package's site on CRAN: `http://cran.r-project.org/package=bvpSolve/`

2. A Simple BVP Example

Here is a simple BVP ODE (which is problem 7 from the test problems available from `http://wwwf.imperial.ac.uk/~jcash/BVP_software/PROBLEMS.PDF`):

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

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 three methods, as [1]:

```
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
  0.02    0.00    0.02

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

  user  system elapsed
  0.01    0.00    0.02
```

[1] the system time, in seconds is printed

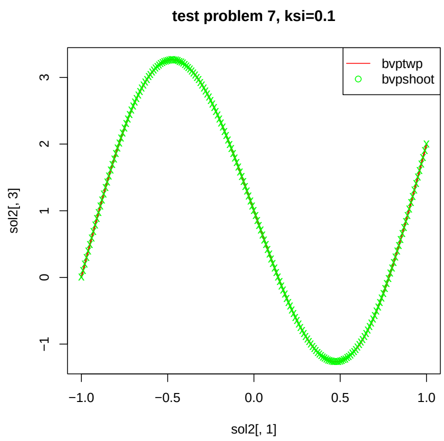

Figure 1: Solution of the simple BVP, for ksi=0.1 - see text for R -code

```
print(system.time(
 sol3  <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = x, func = fun)
 ))
```

```
   user  system elapsed 
   0.00    0.00    0.01 
```

Note how the boundary conditions at the start (`yini`) and at the 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 for the shooting method.

The shooting method is often faster than the other methods. However, there are particular problems where `bvpshoot` does not give a solution, whereas `bvptwp` or `bvpcol` do (see below).

The plot shows that the 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"))
```

For very small values of the parameter ξ, `bvpshoot` cannot solve the problem anymore, due to the presence of a zone of rapid change near x=0.

However, it can still be solved with the other methods, provided that good initial guesses are provided:

```
ks <- 0.0005
print(system.time(
 sol3  <- bvptwp(yini = c(-1, NA), yend = c(1, NA), x = seq(-1, 1, by = 0.01),
            func = fun, xguess = sol2[,1], yguess = t(sol2[,-1]))
 ))
```

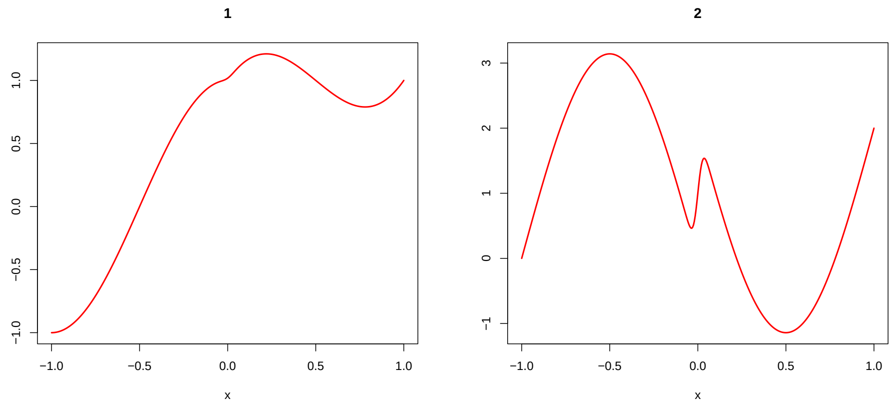

Figure 2: Solution of the simple BVP, for ksi=0.0001 - see text for R -code. Note that this problem cannot be solved with `bvpshoot`

```
   user  system elapsed
   0.08    0.00    0.09
```

```
print(system.time(
 sol3b <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = seq(-1, 1, by = 0.01),
            func = fun, xguess = sol2[,1], yguess = t(sol2[,-1]))
 ))
```

```
   user  system elapsed
   0.24    0.00    0.20
```

When using `bvpcol`, the simulation can also be continuated by simply passing the previous solution (as generated by `bvpcol` to the solver:

```
ks <- 0.0001
print(system.time(
 sol3c <- bvpcol(yini = c(-1, NA), yend = c(1, NA), x = seq(-1, 1, by = 0.01),
            func = fun, yguess = sol3b)
 ))
```

```
   user  system elapsed
   0.07    0.00    0.04
```

Here we produce the output with the S3 `plot` method, which depicts both dependent variables:

```
plot(sol3, type = "l", lwd = 2, col = "red")
```

3. A More Complex BVP Example

Now the test problem referred to as "swirling flow III" is solved (Ascher *et al.* 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:

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

3.1. Solving the Problem as a Set of 1st Order Equations

First the second and fourth order equations are rewritten as a set of 1_{st} 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` and `bvpcol`:

```
print(system.time(
 Soltwp <- bvptwp(x = x, func = fsub,
           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 
  0.21    0.00    0.22 
```

```
print(system.time(
 Solcol <- bvpcol(x = x, func = fsub,
            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 
  0.25    0.00    0.22 
```

where the reported system time is in seconds

The diagnostics of the output solved with `bvptwp` shows that the problem is quite stiff (high values of the conditioning parameters).

Both solvers require almost the same amount of steps:

```
diagnostics(Soltwp)
```

```
--------------------
solved with  bvptwp
--------------------
  Integration was successful.

  1 The return code                                : 0
  2 The number of function evaluations             : 28507
  3 The number of jacobian evaluations             : 3179
  4 The number of boundary evaluations             : 84
  5 The number of boundary jacobian evaluations    : 66
  6 The number of steps                            : 18
  7 The number of mesh resets                      : 1
  8 The maximal number of mesh points              : 1000
  9 The actual number of mesh points               : 199
 10 The size of the real work array                : 280660
 11 The size of the integer work array             : 14018


--------------------
conditioning pars
--------------------

  1 kappa1  : 12601.34
  2 gamma1  : 818.5373
  3 sigma   : 36.8086
  4 kappa   : 13175.8
  5 kappa2  : 574.46
```

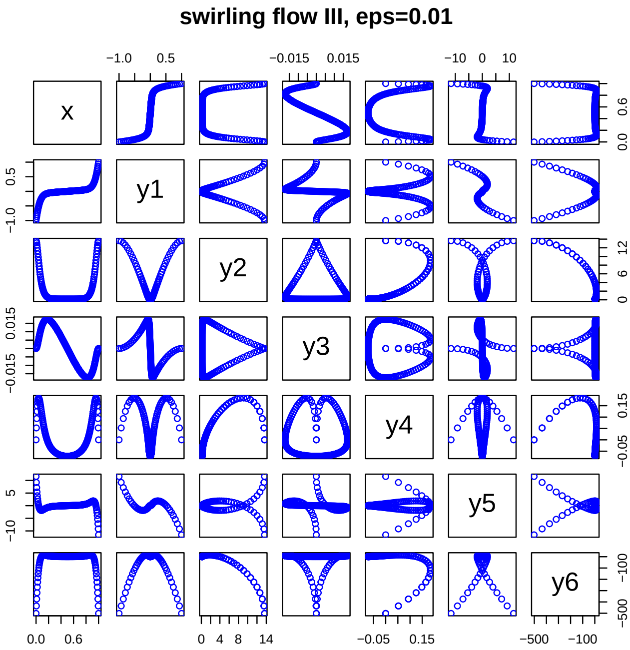

Figure 3: pairs plot of the swirling flow III problem - see text for R -code. Note that this problem cannot be solved with `bvpshoot`

```
diagnostics(Solcol)
```

```
--------------------
solved with  bvpcol
--------------------
  Integration was successful.

  1 The return code                                       : 1
  2 The number of function evaluations                    : 29821
  3 The number of jacobian evaluations                    : 2980
  4 The number of boundary evaluations                    : 294
  5 The number of boundary jacobian evaluations           : 120
  6 The number of steps                                   : 61
  7 The actual number of mesh points                      : 300
  8 The number of collocation points per subinterval      : 4
  9 The number of equations                               : 6
 10 The number of components (variables)                  : 6
```

A `pairs` plot produces a pretty picture.

```
pairs(Soltwp, main = "swirling flow III, eps=0.01", col = "blue")
```

Using the same input as above, the problem cannot be solved with too small values of eps (but see next section):

```
> eps <- 1e-9
> Soltwp2 <- NA
> Soltwp2 <- try(bvptwp(x=x,func=fsub,
+                         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, yini = c(y1 = -1,  :
  The Expected No. Of mesh points Exceeds Storage Specifications.
```

3.2. Solving the Problem in Higher Order Form

All functions can also solve the same problem without rewriting it in 1^{st} order form. However, only for function `bvpcol` will this lead to a significant improvement of performance. Thus, the derivative function becomes:

```
fsubhigh <- function (t,Y,pars)  {
   return(list(c(d2g = (Y[1]*Y[4] - Y[3]*Y[2])/eps,
                 d4f = (-Y[3]*Y[6] - Y[1]*Y[2])/eps)))
 }
eps <- 0.001
x   <- seq(0, 1, len = 100)
```

To solve the model in this form, we need to specify the order of each equation. The first equation is of order 2, the second of order 4.

```
print(system.time(
 Solcol2 <- bvpcol(x = x, func = fsubhigh, order = c(2, 4),
            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
  0.11    0.00    0.11
```

Whereas both ways of solving the system produce almost the same output (but not quite), the second way is significantly faster. It indeed solves the problem in less steps:

```
max(abs(Solcol2-Solcol))
```

```
[1] 2.65498e-08
```

```
diagnostics(Solcol2)
```

```
--------------------
solved with  bvpcol
--------------------
  Integration was successful.

  1 The return code                                    : 1
  2 The number of function evaluations                 : 17201
  3 The number of jacobian evaluations                 : 1725
  4 The number of boundary evaluations                 : 174
  5 The number of boundary jacobian evaluations        : 72
  6 The number of steps                                : 35
  7 The actual number of mesh points                   : 160
  8 The number of collocation points per subinterval   : 5
  9 The number of equations                            : 2
 10 The number of components (variables)               : 6
```

This is not case when we use `bvptwp`, where the higher-order specification is usually a bit slower than the first-order model:

```
 print(system.time(
  Soltwp2 <- bvptwp(x = x, func = fsubhigh, order = c(2, 4),
             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 
   0.24    0.00    0.27 
```

Here the output for the first-order and higher-order specification is exactly the same:

```
 max(abs(Soltwp2- Soltwp))
```

```
[1] 0
```

This problem is much too difficult to be solved with `bvpshoot`

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, e.g. 0.0001. When using `bvptwp`, we pass the previous output values in `xguess` and `xguess`

```
eps <- 0.0001
xguess <- Soltwp[,1]
yguess <- t(Soltwp[,2:7])

print(system.time(
   Sol2 <- bvptwp(x = x, func = fsub, 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
  0.73    0.00    0.76
```

Continuation using `bvpcol` can also be done in the same way, but it is more efficient to pass the entire previous solution produced by `bvpcol` in `xguess`.

```
print(system.time(
   Sol2b <- bvpcol(x = x, func = fsub, yguess = Solcol,
           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
  1.02    0.00    1.02
```

We use the S3 `plot` method to plot all dependent variables at once: These plots are to be compared with the first column of the "pairs" plot (figure 3).

```
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")
```

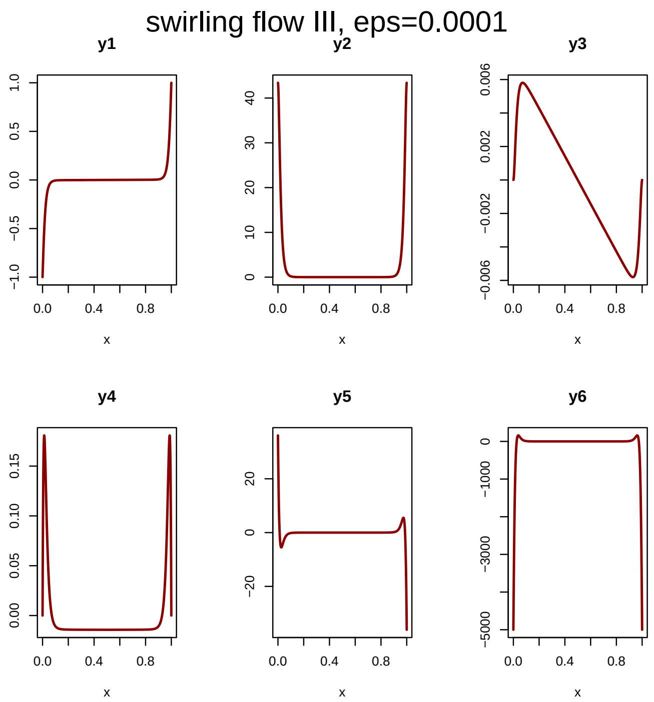

Figure 4: Solution of the swirling flow III problem with small eps, using continuation - see text for R -code.

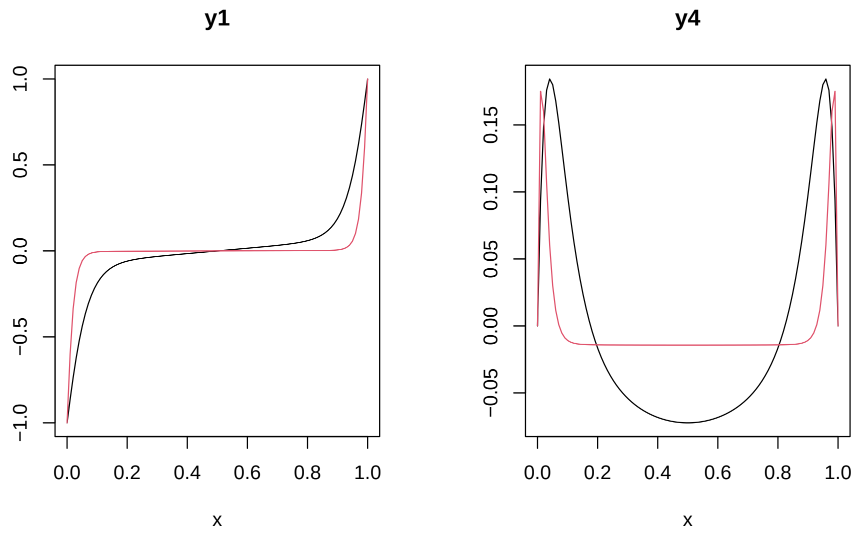

Figure 5: Plotting multiple scenario's at once - see text for R -code.

5. Plotting bvpSolve objects

It is possible to plot several outputs of the same model at once, and to add observed conditions.

```
plot (Solcol, Sol2b, which = c("y1","y4"), lty = 1)
```

It is also relatively simple to add observed data. If not specified, then only the variables, in common between the output and data will be plotted:

```
obsdat <- matrix (ncol = 2, data =
       c(seq(0, 1, 0.2), c(-1, -0.4, -0.1, 0.1, 0.4, 1)))
colnames (obsdat) <- c("x", "y1")
obsdat
```

```
       x   y1
[1,] 0.0 -1.0
[2,] 0.2 -0.4
[3,] 0.4 -0.1
[4,] 0.6  0.1
[5,] 0.8  0.4
[6,] 1.0  1.0
```

```
plot(Solcol, Sol2b, obs = obsdat)
```

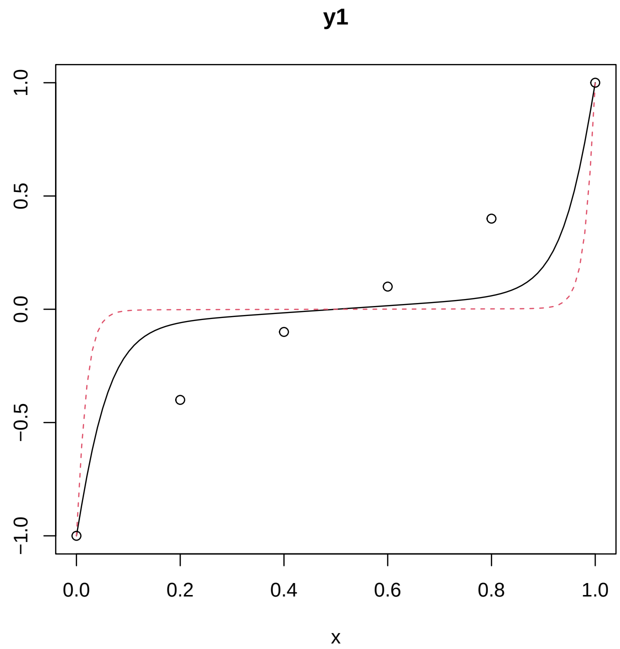

Figure 6: Plotting multiple scenario's with observed data - see text for R -code.

6. 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 condition 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)))
   })
 }
```

This model is solved differently whether bvpshoot or bvptwp is used.

6.1. Solving Problem musn with bvpshoot

It is easiest to solve the musn model with bvpshoot:

There are 4 boundary values specified at the start of the interval; a value for y is lacking (and set to NA):

```
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):

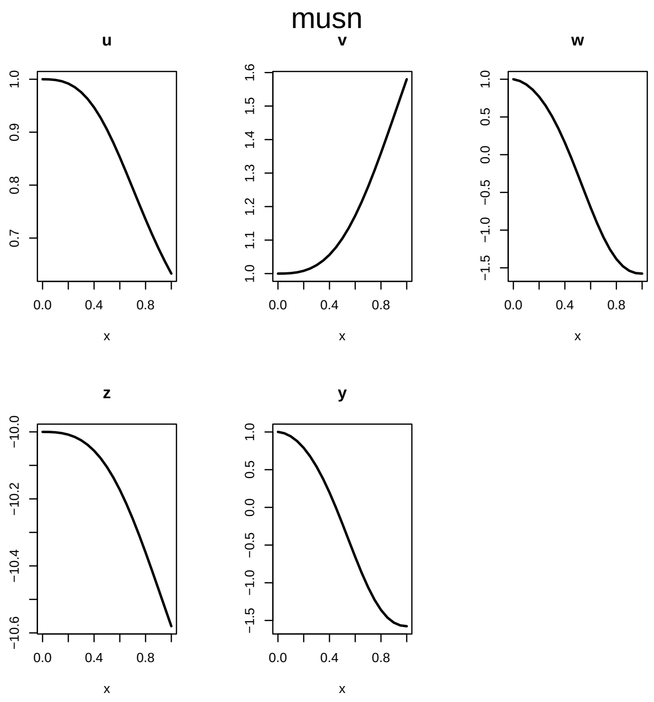

Figure 7: Solution of the musn model, using `bvpshoot` - see text for R -code.

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

Note that the specification of the boundaries for `bvptwp` are rather different (next section). The solution, using `bvpshoot` is obtained by: [2]

```
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
   0.07    0.00    0.06
```

and plotted as:

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

6.2. Solving Problem musn with bvptwp or bvpcol

Here the boundary function `bound` must be specified:

[2]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.

```
bound <- function(i,y,pars) {
   with (as.list(y), {
     if (i ==1) return (u-1)
     if (i ==2) return (v-1)
     if (i ==3) return (w-1)
     if (i ==4) return (z+10)
     if (i ==5) return (w-y)
  })
 }
```

Moreover, this problem can only be solved if good initial conditions are given:

```
xguess <- seq(0, 1, len = 5)
yguess <- matrix(ncol = 5, (rep(c(1, 1, 1, -10, 0.91), times = 5)) )
rownames(yguess) <- c("u","v","w","z","y")
xguess
```

```
[1] 0.00 0.25 0.50 0.75 1.00
```

```
yguess
```

```
    [,1]   [,2]   [,3]   [,4]   [,5]
u   1.00   1.00   1.00   1.00   1.00
v   1.00   1.00   1.00   1.00   1.00
w   1.00   1.00   1.00   1.00   1.00
z -10.00 -10.00 -10.00 -10.00 -10.00
y   0.91   0.91   0.91   0.91   0.91
```

Note that the rows of `yguess` have been given a name, such that this name can be used in the derivative and boundary function.

We specify that there are 4 left boundary conditions (`leftbc`).

```
print(system.time(
 Sol <- bvptwp(yini = NULL, x = x, func = musn, bound = bound,
               xguess = xguess, yguess = yguess, leftbc = 4,
               atol = 1e-10)
 ))
```

```
   user  system elapsed
   0.10    0.01    0.13
```

```
print(system.time(
 Sol2 <- bvpcol(yini = NULL, x = x, func = musn, bound = bound,
               xguess = xguess, yguess = yguess, leftbc = 4,
               atol = 1e-10)
 ))
```

```
user  system elapsed
0.12    0.00    0.11
```

7. A BVP Problem Including an Unknown Parameter

In the next BVP problem (Shampine *et al.* 2000), the fourth eigenvalue of the Mathieus equation (parameter λ) is computed. The equation is

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

defined on $[0,\pi]$, and with boundary conditions $\frac{dy}{dx}(0) = 0$ and $\frac{dy}{dx}(\pi) = 0$ and $y(0) = 1$
Here all the initial values (at x=0) are prescribed, in addition to one condition at the end of the interval. If λ would be known the problem would be overdetermined.
The 2^{nd} order differential equation is first rewritten as two 1^{st}-order equations:

$$\begin{aligned} \frac{dy}{dx} &= y_2 \\ \frac{dy_2}{dx} &= -(\lambda - 10\cos(2x)) \cdot y \end{aligned}$$

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

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

7.1. Solving For an Unknown Parameter Using bvpshoot

This problem is most easily solved using `bvpshoot`; an initial guess of the extra parameter to be solved is simply passed via argument `extra`.

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

The result is plotted:

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

The value of `lambda` can be printed:

```
attr(sol, "roots")  # root gives the value of "lambda" (17.10683)

      root        f.root iter
2 17.10683 -5.205281e-13    6
```

7.2. Solving For an Unknown Parameter Using bvptwp or bvpcol

To use `bvptwp` or `bvpcol`, we treat the unknown parameter as an extra variable, whose derivative = 0 (it is a parameter, and by definition does not change over the integration

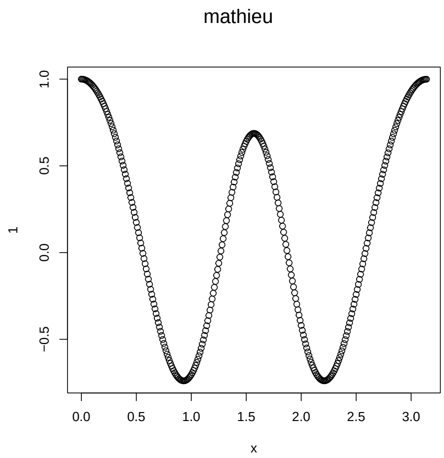

Figure 8: Solution of the BVP ODE problem including an unknown parameter, see text for R-code

interval). This is, the equations are:

$$\begin{aligned}\frac{dy}{dx} &= y_2\\ \frac{dy_2}{dx} &= -(\lambda - 10\cos(2x))\cdot y\\ \frac{d\lambda}{dx} &= 0\end{aligned}$$

for dependent variables `y, y_2` and λ

The model definition in R becomes:

```
mathieu2 <- function(x,y,p)
     list(c(y[2],
         -(y[3] - 10 * cos(2 * x)) * y[1],
         0) )
```

Note the third derivative, and the parameter `lambda` from previous chapter which is now `y[3]`, the third variable.

The initial condition, `yini` and final condition, `yend` now also provides a value, `NA`, for the parameter (y3) that is unknown. We also provide initial guesses for the x- and y-values (`xguess, yguess`). [3]

```
Sol <- bvptwp (yini = c(y = 1, dy = 0, lambda = NA), yend = c(NA, 0, NA),
         x = seq(0, pi, by = 0.01), func = mathieu2, xguess = c(0, pi),
         yguess = matrix(nrow = 3, data = rep(15, 6)) )
```

The y-value, its derivative, and `lambda`, are plotted

[3]This problem is not solved if the initial guess for the y-values is 0; yet any value different from 0 works

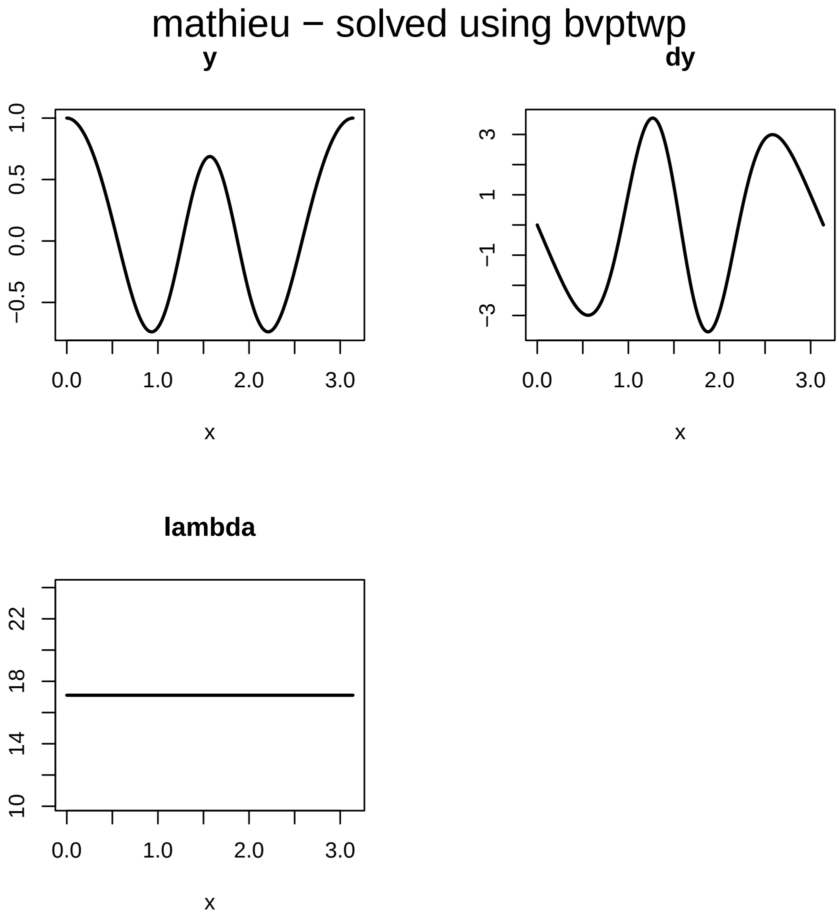

Figure 9: Solution of the BVP ODE problem including an unknown parameter, and using method bvptwp - see text for R-code

```
plot(Sol, type = "l", lwd = 2)
mtext(outer = TRUE, side = 3, line = -1.5, cex = 1.5,
      "mathieu - solved using bvptwp")
```

8. A Boundary Value Problem with Periodic Boundary Conditions

A BVP with cyclic boundary conditions is the nerve impulse model, a problem described in (Seydel 1988). The equations are:

$$y_1' = 3T(y_1 + y_2 - 1/3y_1^3 - 1.3)$$
$$y_2' = -T(y_1 - 0.7 + 0.8y_2)/3$$

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

$$y_1(0) = y_1(1)$$
$$y_2(0) = y_2(1)$$
$$1 = -T(y_1(0) - 0.7 + 0.8 * y_2(0))/3$$

8.1. Cyclic Boundary Conditions Solved Using bvpshoot

The problem is first solved using `bvpshoot`:

The derivative function (where `T` is the parameter) is:

```
nerve <- function (t,y,T)
   list(c( 3 * T * (y[1] + y[2] - 1/3 * (y[1]^3) - 1.3),
         (-1/3) * T * (y[1] - 0.7 + 0.8 * y[2]) ))
```

and the residual function, at the end of the interval is:

```
res<- function (Y,yini,T)
   c(Y[1] - yini[1],
     Y[2] - yini[2],
     T*(-1/3) * (yini[1] - 0.7 + 0.8 * yini[2]) - 1)
```

There are no initial conditions (`yini`); to solve this model, a reasonable `guess` of the missing initial conditions is necessary; the initial guess for the unknown parameter, T, is set to 2π (`extra`):

```
yini <- c(y1 = NA, y2 = NA)
sol  <- bvpshoot(yini = yini, x = seq(0, 1, by = 0.01),
         func = nerve, guess = c(0.5,0.5), yend = res, extra = 2 * pi)
```

T is estimated to be 10.710809; the root has been found in 11 iterations:

```
attributes(sol)$root
```

```
       root        f.root iter
1 -1.183453 -6.435181e-10   11
2  2.004203 -3.778911e-09   11
3 10.710809 -6.905587e-14   11
```

8.2. Cyclic Boundary Conditions Solved using bvptwp or bvpcol

Function `bvptwp` accepts only problems with separated boundary conditions, however, it is possible to use it also for solving boundary value problems with periodic boundary conditions.

This is done by considering the boundary conditions as "parameters", and using the strategy of defining these parameters as extra variables, with derivatives = 0, similar as in previous section.

The augmented derivative function is, with variable 3 the unknown parameter `T`, variables 4 and 5 the initial conditions of y_1, and y_2 respectively, is:

```
nerve3 <- function (t,y,p)
   list(c( 3 * y[3] * (y[1] + y[2] - 1/3 * (y[1]^3) - 1.3),
         (-1/3) * y[3] * (y[1] - 0.7 + 0.8 * y[2]) ,
         0,
         0,
         0)
   )
```

The required boundary function, with the first 3 boundary conditions at the left boundary is:

```
bound <- function(i,y,p) {
  if (i == 1) return ( y[3]*(-1/3) * (y[1] - 0.7 + 0.8 * y[2]) - 1 )
  if (i == 2) return ( y[1] - y[4] )
  if (i == 3) return ( y[2] - y[5] )          # left bnd
  if (i == 4) return ( y[1] - y[4] )          # right bnd
  if (i == 5) return ( y[2] - y[5] )
 }
```

boundary condition 2 sets the left boundary of y_1 equal to parameter y_4, boundary condition 4 does the same for the right boundary of y_1.

Solving this also requires good initial conditions to find a solution:

```
xguess = seq(0, 1, by = 0.1)
yguess = matrix(nrow = 5, ncol = length(xguess), data = 5.)
yguess[1,] <- sin(2 * pi * xguess)
yguess[2,] <- cos(2 * pi * xguess)
rownames(yguess) <- c("y1", "y2", "T", "y1ini", "y2ini")
```

We need to specify that there are three left boundary conditions (`leftbc`)

```
Sol  <- bvptwp(func = nerve3, bound = bound, x = seq(0, 1, by = 0.01),
         ynames = c("y", "dy", "T", "yi", "yj"),
         leftbc = 3, xguess = xguess, yguess = yguess)
```

The first row of `Sol` shows the initial conditions:

```
Sol[1,]
```

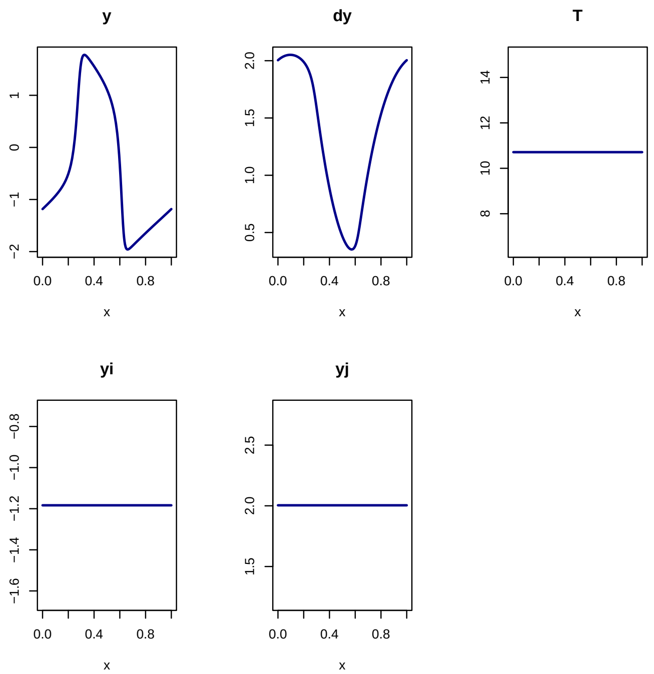

Figure 10: Solution of the nerve impulse problem, comprising two state variables and 3 parameters - see text for R-code

```
        x         y        dy         T        yi        yj
 0.000000 -1.183453  2.004203 10.710808 -1.183453  2.004203
```

```
plot(Sol, type = "l", lwd = 2, col = "darkblue")
```

9. A Set of Higher-Order BVPs

Here the "fluid injection problem" is solved (Ascher *et al.* 1995).

The original problem definition is:

$$\begin{aligned} f''' - R[(f')^2 - f * f''] + RA = 0 \\ h'' + R * f * h' + 1 = 0 \\ O'' + P * f * O' = 0 \end{aligned}$$

on the interval [0,1], and where A is an unknown constant.

The boundary conditions are:

$$\begin{aligned} f(0) = 0, f'(0) = 0, h(0) = 0, O(0) = 0 \\ f(1) = 1, f'(1) = 0, h(1) = 0, O(1) = 1 \end{aligned}$$

We first solve this equation as a set of 1st order ODEs, which is implemented and solved in R as:

```
fluid<-function(t, y, pars, R) {
  P    <- 0.7*R
  with(as.list(y),   {
   df = f1                        #f'
   df1= f2                        #f''
   df2= R * (f1^2 - f*f2)-A       #f'''
   dh = h1
   dh1= -R * f * h1 - 1
   dO = O1
   dO1= -P * f * O1
   dA = 0                     # the constant to be estimated
   return(list(c(df, df1, df2, dh, dh1, dO, dO1, dA)))
  })
 }
times  <- seq(0, 1, by = 0.01)
yini   <- c(f = 0, f1 = 0, f2 = NA, h = 0, h1 = NA, O = 0, O1 = NA, A = NA)
yend   <- c(1,          0,       NA,      0,       NA,      1,       NA,      NA)
print (system.time(
 Solcol1 <- bvpcol(func=fluid, x=times, parms=NULL, R=10000,
                  yini = yini, yend=yend)
 ))

  user  system elapsed
  0.39    0.05    0.43
```

where the reported system time is in seconds. For increasing values of R, the problem becomes more and more difficult to solve.

The implementation as a set of higher-order ODEs is as follows:

```
fluidHigh <- function(t, y, pars, R) {
  P    <- 0.7 * R
  with(as.list(y),   {
   d3f = R * (f1^2 - f * f2) - A       #f'''
   d2h = -R * f * h1 - 1
   d2O = -P * f * O1
   dA  = 0
   return(list(c(d3f, d2h, d2O, dA)))
  })
 }
times  <- seq(0, 1, by = 0.01)
yini   <- c(f = 0, f1 = 0, f2 = NA, h = 0, h1 = NA, O = 0, O1 = NA, A = NA)
yend   <- c(1,          0,      NA,     0,      NA,     1,      NA,     NA)
print (system.time(
 Solcol2 <- bvpcol(func = fluidHigh, x = times, parms = NULL, R = 10000,
                   order = c(3, 2, 2, 1), yini = yini, yend=yend)
 ))
```

```
   user  system elapsed
   0.27    0.00    0.23
```

Note that we have specified the `order` of each equation (3,2,2,1) in `fluidHigh`. The output is the same as the other specification:

```
head(Solcol1, n = 3)
```

```
        x           f       f1         f2            h           h1
[1,] 0.00 0.000000000 0.000000 244.549165 0.0000000000  0.030029302
[2,] 0.01 0.008423256 1.361821  53.204941 0.0002321423  0.014032413
[3,] 0.02 0.023462587 1.567566   3.822699 0.0002802765 -0.001566255
             O       O1        A
[1,] 0.0000000 62.30644 24932.52
[2,] 0.5894266 50.15712 24932.52
[3,] 0.9206130 16.61383 24932.52
```

```
head(Solcol2, n = 3)
```

```
        x           f       f1         f2            h           h1
[1,] 0.00 0.000000000 0.000000 244.549165 0.0000000000  0.030029301
[2,] 0.01 0.008423256 1.361821  53.204941 0.0002321423  0.014032413
[3,] 0.02 0.023462587 1.567566   3.822699 0.0002802765 -0.001566255
             O       O1        A
[1,] 0.0000000 62.30644 24932.52
[2,] 0.5894266 50.15712 24932.52
[3,] 0.9206130 16.61383 24932.52
```

```
plot(Solcol1, main="Fluid injection problem",
      which = "f1", type = "l", lwd = 2)
```

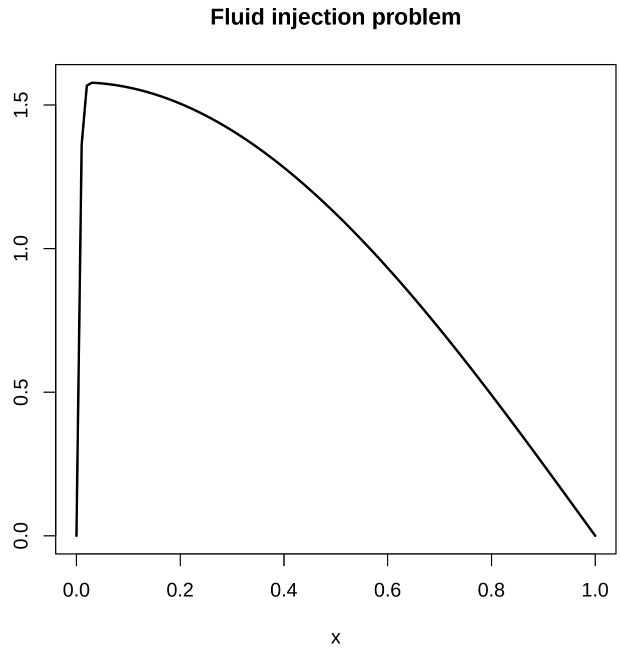

Figure 11: The fluid injection problem, a set of higher-order ODEs - see text for R -code.

10. A Multipoint Problem

Function `bvptwp` can only solve problems whose boundary conditions are located at the start and/or end of the integration interval.

Function `bvpcol` (and also `bvpshoot`) can also solve problems where the extra conditions are somewhere within the integration interval.

Consider the following problem:

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

defined in the interval [0,1] and with extra conditions:

$$y_1(1) = 0$$
$$y_2(0.5) = 1$$

As the second condition is specified within the integration interval, this is a multipoint problem.

Mulipoint problems can only be specified in R using a boundary function `bound`; as the boundary for y_2 is specified before y_1, it is treated first in the boundary function (because `posbound` has to be sorted).

```
multip <- function (x, y, p) {
   list(c((y[2] - 1)/2,
          (y[1]*y[2] - x)/mu))
 }
bound <- function (i, y, p) {
   if (i == 1) y[2] -1         # at x=0.5: y2=1
   else y[1]                   # at x=  1: y1=0
 }
mu  <- 0.1
sol <- bvpcol(func = multip, bound = bound,
              x = seq(0, 1, 0.01), posbound = c(0.5, 1))
```

We check the boundary values:

```
sol[sol[,1] %in% c(0.5,1),]
```

```
       x        1         2
[1,] 0.5 0.338895  1.000000
[2,] 1.0 0.000000 -2.658449
```

```
plot(sol)
```

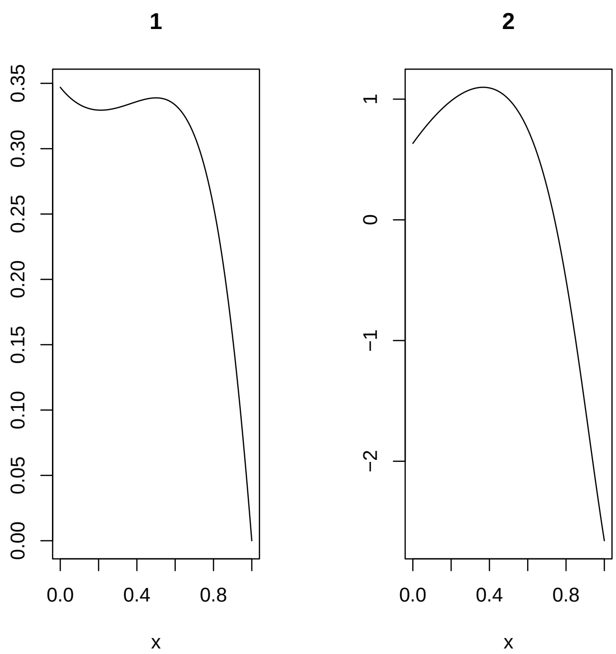

Figure 12: Solution of a multipoint problem - see text for R-code

11. Specifying the Analytic Jacobians

By default, the Jacobians 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 (http://wwwf.imperial.ac.uk/~jcash/BVP_software/readme.php).

The original system reads:

$$\frac{dx}{ds} = \cos(\phi) \tag{1}$$

$$\frac{dy}{ds} = \sin(\phi) \tag{2}$$

$$\frac{d\phi}{ds} = \kappa \tag{3}$$

$$\frac{d\kappa}{ds} = F\cos(\phi) \tag{4}$$

$$\frac{dF}{ds} = 0 \tag{5}$$

where F is an (unknown) constant, and with the following boundary conditions:

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 0 \\ \kappa(0) &= 0 \\ y(0.5) &= 0 \\ \phi(0.5) &= -\pi/2 \end{aligned}$$

First implementation uses 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 <- bvpcol(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))

plot(Sol)
```

Now several extra functions are defined, specifying

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

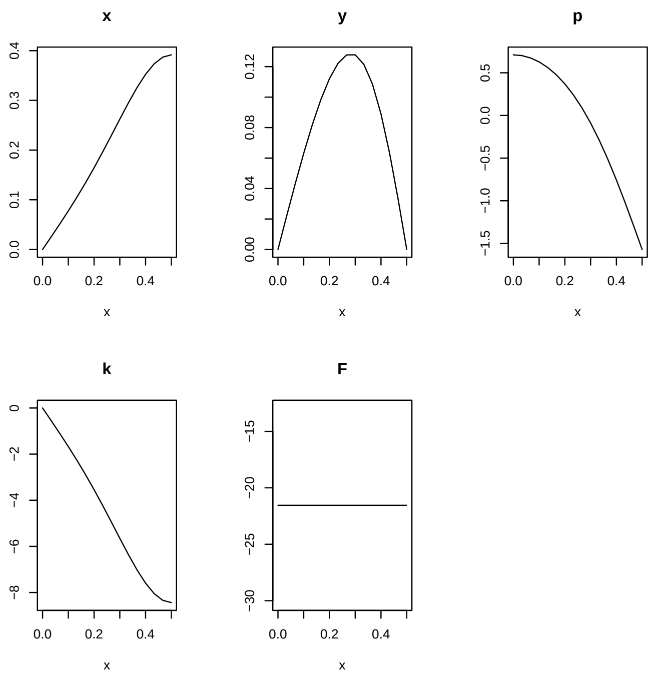

Figure 13: Solution of the elastica problem - see text for R-code

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 $BC_3 = y[3] + \pi/2$

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

This is done in the R -code below:

```
jacfunc <- function (x, y, pars) {
       Jac <- matrix(nrow = 5, ncol = 5, data = 0)
       Jac[3,4] <- 1.0
       Jac[4,4] <- 1.0
       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`), and either the number of state variables (`ncomp`), or their names (`ynames`) needs to be specified.

```
 Sol4 <- bvpcol(leftbc = 3, ynames = c("x", "y", "p", "k", "F"),
               func = Elastica, jacfunc = jacfunc,
               bound = bound, jacbound = jacbound,
               x = seq(0, 0.5, len=16))
```

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

12. 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.* 2010b).

Its vignette ("compiledCode") can be consulted for more information. (http://cran.r-project.org/package=deSolve/)

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.

12.1. The Elastica Problem in FORTRAN

Below is an implementation of the elastica model in FORTRAN: (slightly modified from http://wwwf.imperial.ac.uk/~jcash/BVP_software/readme.php):

```
c  The differential system:
      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
```

12.2. The Elastica Problem in C

The same model, implemented in C is:

```
#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.;
  }
```

12.3. Solving the Elastica Problem Specified in Compiled Code

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 version, this is done as follows (the C code is similar, except for the name of the DLL):

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

outF <- bvpcol(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.

13. Passing Parameters and External Data to Compiled Code

When using compiled code, it is possible to

- pass *parameters* from R to the compiled functions
- pass *forcing functions* from R to compiled functions. These are then updated to the correct value of the independent variable (`x`) at each step.

The implementation of this is similar as in package **deSolve**. How to do it has been extensively explained in deSolve's vignette, which can be consulted for details.

See http://cran.r-project.org/package=deSolve.

Here we implement a simple linear boundary value problem, which is a standard test problem for BVP code ((Scott and Watts 1977)). The model has a boundary layer at x=0.

The differential equation depends on a parameter `a` and `p`:

$$y'' + \frac{-apy}{(p+x^2)^2} = 0$$

and is solved on [-0.1, +0.1] with boundary conditions:

$$\begin{aligned} y(-0.1) &= -0.1\sqrt{p+0.01} \\ y(+0.1) &= 0.1\sqrt{p+0.01} \end{aligned}$$

where `a` $= 3$ and `p` is taken small.

This differential equation is written as a system of two first-order ODEs.

The implementation in pure R is given first:

```
fun <- function(t,y,pars)
   list(c( y[2],
         - a * p * y[1]/(p + t*t)^2
         ))
```

with parameter values:

```
p     <- 1e-5
a     <- 3
```

It is solved using `bvptwp`; note that the initial condition (`yini`) gives names to the variables; these names are used by the solver to label the output:

```
sol  <- bvptwp(yini = c(y = -0.1/sqrt(p+0.01), dy = NA),
                yend = c(      0.1/sqrt(p+0.01),       NA),
                x = seq(-0.1, 0.1, by = 0.001),
                func = fun)

plot(sol, type = "l")
```

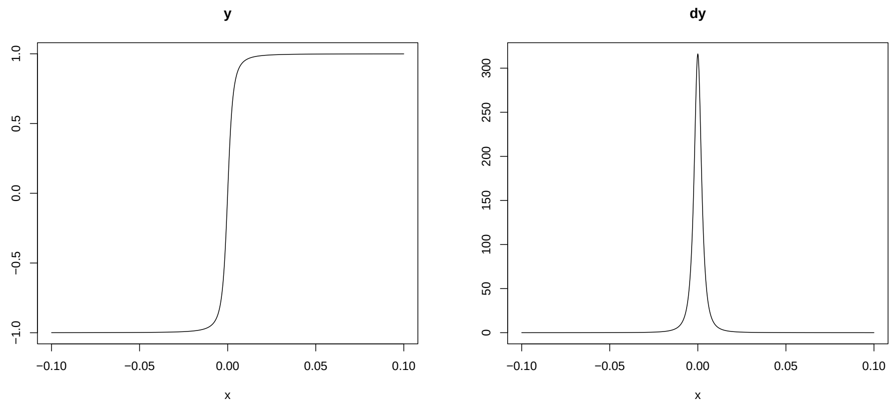

Figure 14: Solution of the linear boundary problem - see text for R-code

Next the FORTRAN implementation is given, which requires writing the boundary and jacobian functions (`bound`, `jacfunc` and `jacbound`)

The two parameters are initialised in a function called `initbnd`; its name is passed to function `bvptwp` via argument `initfunc`.

```
c FORTRAN implementation of the boundary problem
c Initialiser for parameter common block
      SUBROUTINE initbnd(bvpparms)
      EXTERNAL bvpparms

      DOUBLE PRECISION parms(2)
      COMMON / pars / parms

       CALL bvpparms(2, parms)
      END

c derivative function
      SUBROUTINE funbnd(NCOMP,X,Y,F,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER NCOMP, IPAR(*), I
      DOUBLE PRECISION F(2), Y(2), RPAR(*), X
      DOUBLE PRECISION a, p
      COMMON / pars / a, p

        F(1)= Y(2)
        F(2)= - a * p *Y(1)/(p+ x*x)**2
      END

c The analytic Jacobian for the derivative-function:
      SUBROUTINE dfbnd(NCOMP,X,Y,DF,RPAR,IPAR)
```

```
      IMPLICIT NONE
      INTEGER NCOMP, IPAR(*), I, J
      DOUBLE PRECISION X, Y(2), DF(2,2), RPAR(*)
      DOUBLE PRECISION a, p
      COMMON / pars / a, p

        DF(1,1)=0.D0
        DF(1,2)=1.D0
        DF(2,1)= - a *p /(p+x*x)**2
        DF(2,2)=0.D0
      END

c The boundary conditions:
      SUBROUTINE gbnd(I,NCOMP,Y,G,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER I, NCOMP, IPAR(*)
      DOUBLE PRECISION Y(2), RPAR(*), G
      DOUBLE PRECISION a, p
      COMMON / pars / a, p

        IF (I.EQ.1) THEN
          G=Y(1) + 0.1 / sqrt(p+0.01)
        ELSE IF (I.EQ.2) THEN
          G=Y(1) - 0.1 / sqrt(p+0.01)
        ENDIF
      END

c The analytic Jacobian for the boundaries:
      SUBROUTINE dgbnd(I,NCOMP,Y,DG,RPAR,IPAR)
      IMPLICIT NONE
      INTEGER I, NCOMP, IPAR(*)
      DOUBLE PRECISION Y(2), DG(2), RPAR(*)

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

Before running the model, the parameters are defined:

```
parms <- c(a = 3, p = 1e-7)
```

and the DLL created and loaded; This model has been made part of package **bvpSolve** , so it is available in DLL `bvpSolve`.

Assuming that this was not the case, and the code is in a file called `"boundary_for.f"`, this is how to compile this code and load the DLL (on windows):

```
system("R CMD SHLIB boundary_for.f")
dyn.load("boundary_for.dll")
```

We execute the model several times, for different values of parameter `p`; we create a sequence of parameter values (`pseq`), over which the model then iterates (`for (pp in pseq)`); the resulting `y`-values (2^{nd}) column) of each iteration are added to matrix `Out`.

```
Out  <- NULL
x    <- seq(-0.1, 0.1, by = 0.001)
pseq <- 10^-seq(0, 6, 0.5)
for (pp in pseq) {
   parms[2] <- pp
   outFor <- bvptwp(ncomp = 2, x = x, leftbc = 1,
      initfunc = "initbnd", parms = parms, func = "funbnd",
      jacfunc = "dfbnd", bound = "gbnd", jacbound = "dgbnd",
      allpoints = FALSE, dllname = "bvpSolve")
   Out <- cbind(Out, outFor[,2])
 }
```

It takes less than 0.06 seconds to do this.

Results are plotted, using R -function `matplot`:

```
matplot(x, Out, type = "l")
legend("topleft", legend = log10(pseq), title = "logp",
   col = 1 : length(pseq), lty = 1 : length(pseq), cex = 0.6)
```

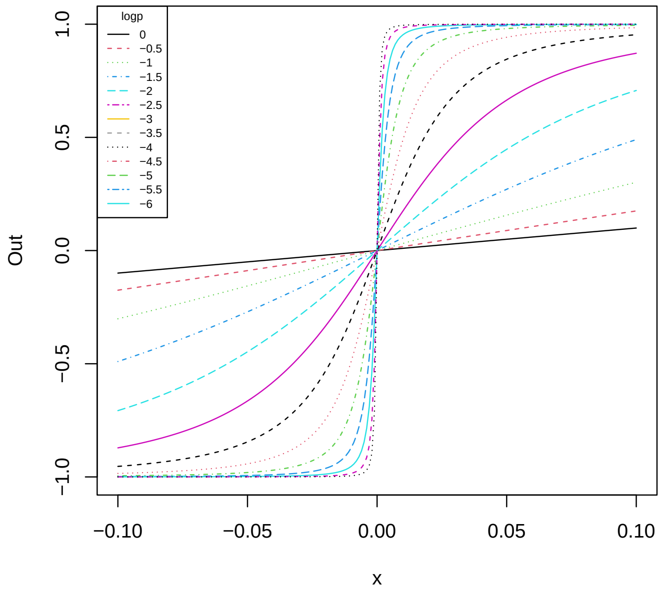

Figure 15: Multiple solutions of the linear problem - see text for R-code

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.

R Development Core Team (2010). *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.

Scott M, Watts H (1977). "Computational solution of linear two point boundary value problems via orthonormalization." *SIAM J. Numer. Anal.*, **14**, 40–70.

Seydel R (1988). *From equilibrium to Chaos.* Elsevier, New York.

Shampine L, Kierzenka J, Reichelt M (2000). *solving boundary value problems for ordinary differential equations in MATLAB with bvp4c.*

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

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

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.

Affiliation:

Karline Soetaert
Royal Netherlands Institute of
Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: karline.soetaert@nioz.nl
URL: https://www.nioz.nl/staff-detail?id=784400

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://www.dm.uniba.it/~mazzia/mazzia/