Package **rootSolve** : roots, gradients and steady-states in R

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)
Yerseke
The Netherlands

Abstract

R package **rootSolve** (Soetaert 2009) includes

- root-finding algorithms to solve for the roots of n nonlinear equations, using a Newton-Raphson method.
- An extension of R function `uniroot`
- Functions that find the steady-state condition of a set of ordinary differential equations (ODE). These functions are compatible with the solvers in package **deSolve** (Soetaert, Petzoldt, and Setzer 2010c), (Soetaert, Petzoldt, and Setzer 2010b), which solve initial value differential equatons. Separate solvers for full, banded and generally sparse problems are included. These allow to estimate the steady-state of 1-D, 2-D and 3-D partial differential equations (PDE) that have been rewritten as a set of ODEs by numerical differencing, e.g. using the functions available in package **ReacTran** (Soetaert and Meysman 2010).
- Functions that calculate the Hessian and Jaobian matrix or - more general - the gradient of functions with respect to independent variables.

Keywords: roots of nonlinear equations, gradient, Jacobian, Hessian, steady-state, boundary value ODE, method of lines, R.

The root of a function f(x) is the value of x for which f(x) = 0.

Package **rootSolve** deals with finding the roots of **n** nonlinear (or linear) equations. This is, it finds the values

$$x_i^* \quad (i = 1, n)$$

for which

$$f_j(\mathbf{x}^*) = \mathbf{0} \quad (j = 1, n)$$

Package **rootSolve** serves several purposes:

- it extends the root finding capabilities of R for non-linear functions.
- it includes functions for finding the steady-state of systems of ordinary differential equations (ODE) and partial differential equations (PDE).
- it includes functions to numerically estimate gradient matrices.

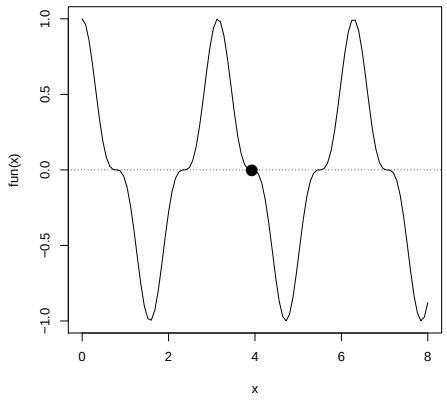

Figure 1: Root found with *uniroot*

The package was created to solve the steady-state and stability analysis examples in the book of Soetaert and Herman (2009). Please cite this work if you use the package.

The various functions in **rootSolve** are given in table (1).

1. Finding roots of nonlinear equations in R and rootSolve

The root-finding functions in R are:

- *uniroot*. Finds one root of one equation.
- *polyroot*. Finds the complex roots of a polynomial.

1.1. One equation

To find the root of function:

$$f(x) = \cos^3(2x)$$

in the interval [0,8] and plot the curve, we write:

```
>   fun <- function (x) cos(2*x)^3
>   curve(fun(x), 0, 8)
>   abline(h = 0, lty = 3)
>   uni <- uniroot(fun, c(0, 8))$root
>   points(uni, 0, pch = 16, cex = 2)
```

Although the graph (figure 1) clearly demonstrates the existence of many roots in the interval [0,8] R function *uniroot* extracts only one.

rootSolve function *uniroot.all* is a simple extension of *uniroot* which extracts many (presumably *all*) roots in the interval.

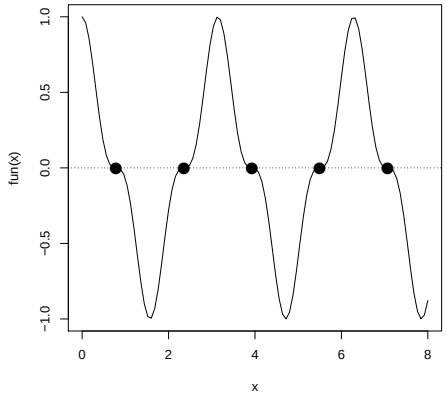

Figure 2: Roots found with uniroot.all

```
>   curve(fun(x), 0, 8)
>   abline(h = 0, lty = 3)
>   All <- uniroot.all(fun, c(0, 8))
>   points(All, y = rep(0, length(All)), pch = 16, cex = 2)
```

uniroot.all does that by first subdividing the interval into small sections and, for all sections where the function value changes sign, invoking *uniroot* to locate the root.

Note that this is not a full-proof method: in case subdivision is not fine enough some roots will be missed. Also, in case the curve does not cross the X-axis, but just "touches" it, the root will not be retrieved; (but neither will it be located by *uniroot*).

1.2. n equations in n unknowns

Except for polynomial root finding, to date R has no functions that retrieve the roots of multiple nonlinear equations.

Function *multiroot* in **rootSolve** implements the Newton-Raphson method (e.g. Press, Teukolsky, Vetterling, and Flannery (1992)) to solve this type of problem. As the Newton-Raphson method locates the root iteratively, the result will depend on the initial guess of the root. Also, it is not guaranteed that the root will actually be found (i.e. the method may fail).

The example below finds two different roots of a three-valued function:

$$f_1 = x_1 + x_2 + x_3^2 - 12$$
$$f_2 = x_1^2 - x_2 + x_3 - 2$$
$$f_3 = 2 \cdot x_1 - x_2^2 + x_3 - 1$$

```
> model <- function(x) {
+   F1 <- x[1]   + x[2]   + x[3]^2 -12
+   F2 <- x[1]^2 - x[2]   + x[3] -2
+   F3 <- 2*x[1] - x[2]^2 + x[3] -1
```

```
+ c(F1 = F1, F2 = F2, F3 = F3)
+ }
> # first solution
> (ss <- multiroot(f = model, start = c(1, 1, 1)))

$root
[1] 1 2 3

$f.root
           F1            F2            F3
 3.087877e-10  4.794444e-09 -8.678146e-09

$iter
[1] 6

$estim.precis
[1] 4.593792e-09

> # second solution; use different start values
> (ss2 <- multiroot(model, c(0, 0, 0)))$root

[1] -0.2337207  1.3531901  3.2985649

> model(ss2$root)   # the function value at the root

           F1            F2            F3
 1.092413e-08  1.920978e-07 -4.850423e-08
```

As another example, we seek the 5x5 matrix $\mathbf{X}$ for which

$$\mathbf{X} \cdot \mathbf{X} \cdot \mathbf{X} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

```
> f2<-function(x)  {
+  X <- matrix(nr = 5, x)
+  X %*% X %*% X - matrix(nrow = 5, data = 1:25, byrow = TRUE)
+ }
> print(system.time(
+   x<-multiroot(f2, start = 1:25)$root
+ ))

   user  system elapsed
   0.04    0.00    0.03
```

```
> (X<-matrix(nrow = 5, x))
```

```
            [,1]       [,2]        [,3]      [,4]      [,5]
[1,] -0.67506260 -0.3454778 -0.01800918 0.3123057 0.6404343
[2,] -0.03483809  0.1686488  0.37530821 0.5735846 0.7797107
[3,]  0.60668343  0.6835080  0.76025826 0.8442125 0.9159760
[4,]  1.24815964  1.1997437  1.15042418 1.1004603 1.0600742
[5,]  1.88640623  1.7145706  1.54265669 1.3697198 1.1937326
```

```
> X%*%X%*%X
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
```

1.3. n equations in n unknowns with known Jacobian

If the Jacobian is known, OR it has a known sparsity structure, then it is much more efficient to take that into account;

As an example, a set of linear equations, comprising 500 unknowns are solved. Of course, one would not do that using a nonlinear equation solver, but rather by using `solve`.

However, the example is included here to show how to make good use of the flexibility at which the Jacobian can be specified, and to show that the methods are quite fast!

We start by defining the linear system of equations: a (500x500) matrix `A` and a vector `B` of length 500. It is solved with `solve`, R 's default solver for systems of linear equations. The time it takes to do this is printed (in seconds):

```
> A <- matrix(nrow = 500, ncol = 500, runif(500*500))
> B <- runif(500)
> print(system.time(X1 <- solve(A, B)))
```

```
   user  system elapsed
   0.09    0.00    0.10
```

To use `multiroot` the nonlinear equation solver, it is noted that the Jacobian is equal to matrix `A`. A function is created that returns the Jacobian (or matrix A); the calling sequence of the Jacobian function is `function(x)`

```
> jfun <- function (x) A
```

The function whose root is to be solved receives the current estimate of `x`, and returns the difference `A x-B`:

```
> fun <- function(x) A %*%x - B
```

Next `multiroot` is called with jactype = "fullusr" (it is a full Jacobian, and specified by the user):

```
> print(system.time(
+ X <- multiroot(start = 1:500, f = fun,
+                 jactype = "fullusr", jacfunc = jfun)
+ ))

   user  system elapsed
   0.11    0.00    0.11
```

Finally, both solutions are the same

```
> sum( (X1 - X$y)^2)

[1] 0
```

2. Steady-state analysis

Ordinary differential equations are a special case of nonlinear equations, where $\mathbf{x}$ are called the state variables and the functions $\mathbf{f(x)}$ specify the derivatives of x with respect to some independent variable. This is:

$$f(\mathbf{x}, t) = \frac{d\mathbf{x}}{dt}$$

If "t", the independent variable is "time", then the root of the ODE system

$$\frac{d\mathbf{x}}{dt} = 0$$

is often referred to as the "steady-state" condition.

Within R , package **deSolve** (Soetaert *et al.* 2010b) is designed to solve so-called initial value problems (IVP) of ODEs and PDEs - partial differential equations by integration. **deSolve** includes integrators that deal efficiently with sparse and banded Jacobians or that are especially designed to solve initial value problems resulting from 1-Dimensional and 2-Dimensional partial differential equations. The latter are first written as ODEs using the method-of-lines approach.

To ensure compatibility, **rootSolve** offers the same functionalities as **deSolve** , and requires the ODE's to be similarly specified.

The function specifying the ordinary differential equations should thus be defined as:

```
deriv = function(x,t,parms,...)
```

where *parms* are the ODE parameters, x the state variables, t the independent variable and ... are any other arguments passed to the function (optional).

The return value of the function should be a list, whose first element is a vector containing the derivatives of x with respect to time.

Two different approaches are used to solve for the *steady-state* condition of ODE's:

- by dynamically running to steady-state.
- by solving for the root of the ODE using the Newton-Raphson method.

2.1. Running dynamically to steady-state

Function *runsteady* finds the steady-state condition by dynamically running (integrating) the ODE until the derivatives stop changing. This solves a particular case of an IVP, where the time instance for which the value of the state variable is sought equals infinity.

The implementation is based on **deSolve** solver function *lsode* (Hindmarsh (1983)).

Consider the following simple sediment biogeochemical model:

$$\begin{aligned}
\frac{dOM}{dt} &= Flux - r \cdot OM \cdot \frac{O_2}{O_2 + ks} - r \cdot OM \cdot (1 - \frac{O_2}{O_2 + ks}) \cdot \frac{SO_4}{SO_4 + ks2} \\
\frac{dO_2}{dt} &= -r \cdot OM \cdot \frac{O_2}{O_2 + ks} - 2rox \cdot HS \cdot \frac{O_2}{O_2 + ks} + D \cdot (BO2 - O_2) \\
\frac{dSO_4}{dt} &= -0.5 \cdot r \cdot OM \cdot (1 - \frac{O_2}{O_2 + ks}) \cdot \frac{SO_4}{SO_4 + ks2} + rox \cdot HS \cdot \frac{O_2}{O_2 + ks} + D \cdot (BSO4 - SO_4) \\
\frac{dHS}{dt} &= 0.5 \cdot r \cdot OM \cdot (1 - \frac{O_2}{O_2 + ks}) \cdot \frac{SO_4}{SO_4 + ks2} - rox \cdot HS \cdot \frac{O_2}{O_2 + ks} + D \cdot (BHS - HS)
\end{aligned}$$

In R this model is defined as:

```
>  model <- function(t, y, pars) {
+   with (as.list(c(y, pars)),{
+
+   oxicmin   = r*OM*(O2/(O2+ks))
+   anoxicmin = r*OM*(1-O2/(O2+ks))* SO4/(SO4+ks2)
+
+   dOM  = Flux - oxicmin - anoxicmin
+   dO2  = -oxicmin       -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2)
+   dSO4 = -0.5*anoxicmin   +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4)
+   dHS  =  0.5*anoxicmin   -rox*HS*(O2/(O2+ks)) + D*(BHS-HS)
+
+   list(c(dOM, dO2, dSO4, dHS), SumS = SO4+HS)
+ })
+ }
```

After defining the value of the parameters (`pars`) and the initial values (`y`), the model can be run to steady-state (`runsteady`). We specify the maximal length of time the simulation can take ($1e^5$)

```
> pars <- c(D = 1, Flux = 100, r = 0.1, rox = 1,
+           ks = 1, ks2 = 1, BO2 = 100, BSO4 = 10000, BHS = 0)
> y <- c(OM = 1, O2 = 1, SO4 = 1, HS = 1)
> print(system.time(
+   RS <- runsteady(y = y, fun = model,
+                   parms = pars, times = c(0, 1e5))
+ ))
```

```
   user  system elapsed
   0.17    0.00    0.22

> RS

$y
         OM          O2         SO4          HS
1000.012782    6.825178 9996.587411    3.412589

$SumS
[1] 10000

attr(,"istate")
 [1]      2      1      1      0      0      5 100000      0      0
[10]      0      0    210    374     32      2      1      0     74
[19]     24      0      0      1      2
attr(,"rstate")
[1]  34.76327 347.63269 308.10999   0.00000   0.00000
attr(,"precis")
[1] 1.080609e-09
attr(,"steady")
[1] TRUE
attr(,"time")
[1] 308.11
attr(,"steps")
[1] 210
```

The output shows the steady-state concentrations (`ST$y`), and the output variable (`ST$SumS`), the time and the number of timesteps it took to reach steady-state (attribute "time", "steps").

2.2. Using the Newton-Raphson method

Functions `stode`, `stodes`, `steady`, `steady.1D`, `steady.2D`, `steady.3D`, and `steady.band` find the steady-state by the iterative Newton-Raphson method (e.g. Press *et al.* (1992).

The same model as above can also be solved using `stode`. This is faster than running dynamically to steady-state, but not all models can be solved this way

```
> stode(y = y, fun = model, parms = pars, pos = TRUE)

$y
         OM          O2         SO4          HS
1000.012783    6.825178 9996.587411    3.412589

$SumS
[1] 10000

attr(,"precis")
```

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

Note that we set `pos=TRUE` to ensure that only positive values are found. Thus the outcome will be biologically realistic (negative concentrations do not exist).

2.3. Steady-state of 1-D models

Two special-purpose functions solve for the steady-state of 1-D models.

- Function `steady.band` efficiently estimates the steady-state condition for 1-D models that comprise one species only.
- Function `steady-1D` finds the steady-state for multi-species 1-D problems.

1-D models of 1 species

Consider the following 2nd order differential equation whose steady-state should be estimated:

$$\frac{\partial y}{\partial t} = 0 = \frac{\partial^2 y}{\partial dx^2} + \frac{1}{x}\frac{\partial y}{\partial x} + (1 - \frac{1}{4 \cdot x^2}) \cdot y - \sqrt(x) \cdot \ cos(x)$$

over the interval [1,6] and with boundary conditions:
$y(1) = 1$ and $y(6) = -0.5$
The spatial derivatives are approximated using centred differences [1]:

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{y_{i+1} - 2 \cdot y_i + y_{i-1}}{\Delta x^2}$$

and

$$\frac{\partial y}{\partial x} \approx \frac{y_{i+1} - y_{i-1}}{2 \cdot \Delta x}$$

First the model function is defined:

```
> derivs <- function(t, y, parms, x, dx, N, y1, y6)  {
+
+    d2y <- (c(y[-1],y6) -2*y + c(y1,y[-N])) /dx/dx
+    dy  <- (c(y[-1],y6) - c(y1,y[-N])) /2/dx
+
+    res <- d2y + dy/x + (1-1/(4*x*x))*y-sqrt(x)*cos(x)
+    return(list(res))
+ }
```

Then the interval [1,6] is subdivided in 5001 boxes (`x`) and the steady-state condition estimated, using `steady.band`; we specify that there is only one species (`nspec=1`).

[1] in a later section, an alternative approximation is used

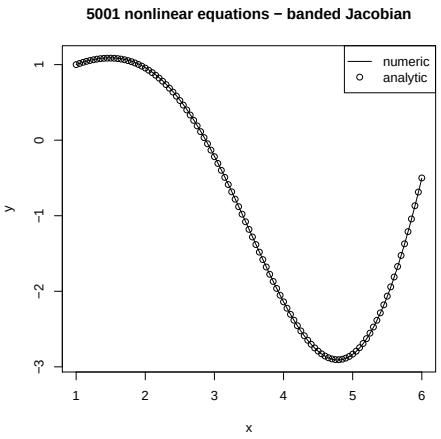

Figure 3: Solution of the 2nd order differential equation - see text for explanation

```
> dx     <- 0.001
> x      <- seq(1, 6, by = dx)
> N      <- length(x)
> print(system.time(
+ y  <- steady.band(y = rep(1, N), time = 0, func = derivs, x = x,
+                    dx = dx, N = N, y1 = 1, y6 = -0.5, nspec = 1)$y
+ ))
```

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

The steady-state of this system of 5001 nonlinear equations is retrieved in about 0.03 seconds [2].

The analytical solution of this equation is known; after plotting the numerical approximation, it is added to the figure (figure 3):

```
> plot(x, y, type = "l",
+      main = "5001 nonlinear equations - banded Jacobian")
> curve(0.0588713*cos(x)/sqrt(x)+1/4*sqrt(x)*cos(x)+
+       0.740071*sin(x)/sqrt(x)+1/4*x^(3/2)*sin(x),add=TRUE,type="p")
> legend("topright", pch = c(NA, 1), lty = c(1, NA),
+        c("numeric", "analytic"))
```

1-D models of many species

In the following model, dynamics of BOD (biochemical oxygen demand) and oxygen is mod-

[2] on my computer that dates from 2008

eled in a river. Both are transported downstream (velocity v)

$$\begin{aligned} \frac{\partial BOD}{\partial t} &= 0 = -\cdot\frac{\partial v\cdot BOD}{\partial x} - r\cdot BOD\cdot\frac{O_2}{O_2+ks} \\ \frac{\partial O_2}{\partial t} &= 0 = -\cdot\frac{\partial v\cdot O_2}{\partial x} - r\cdot BOD\cdot\frac{O_2}{O_2+ks} + p\cdot(O_2sat - O_2) \end{aligned}$$

subject to the boundary conditions $BOD(x=0) = BOD_0$ and $O_2(x=0) = O2_0$

First the advective fluxes (transport with velocity v) are calculated, taking into account the upstream concentrations (`FluxBOD`, `FluxO2`); then the rate of change is written as the sum of -1*Flux gradient and the consumption and production rate:

```
> O2BOD <- function(t, state, pars) {
+   BOD <- state[1:N]
+   O2  <- state[(N+1):(2*N)]
+
+   FluxBOD <-  v*c(BOD_0,BOD)  # fluxes due to water transport
+   FluxO2  <-  v*c(O2_0,O2)
+
+   BODrate <- r*BOD*O2/(O2+10)  # 1-st order consumption, Monod in oxygen
+
+ #rate of change = -flux gradient - consumption  + reaeration (O2)
+   dBOD         <- -diff(FluxBOD)/dx  - BODrate
+   dO2          <- -diff(FluxO2)/dx   - BODrate + p*(O2sat-O2)
+
+   return(list(c(dBOD = dBOD, dO2 = dO2), BODrate = BODrate))
+  }
```

After assigning values to the parameters, and setting up the computational grid (`x`), steady-state is estimated with function `steady.1D`; there are 2 species (BOD, O2) (`nspec=2`); we force the result to be positive (`pos=TRUE`).

```
> dx      <- 10        # grid size, meters
> v       <- 1e2       # velocity, m/day
> r       <- 0.1       # /day, first-order decay of BOD
> p       <- 0.1       # /day, air-sea exchange rate
> O2sat   <- 300       # mmol/m3 saturated oxygen conc
> O2_0    <- 50        # mmol/m3 riverine oxygen conc
> BOD_0   <- 1500      # mmol/m3 riverine BOD concentration
> x       <- seq(dx/2, 10000, by = dx)  # m, distance from river
> N       <- length(x)
> state <- c(rep(200, N), rep(200, N))    # initial guess of state variables:
> print(system.time(
+   out   <- steady.1D (y = state, func = O2BOD, parms = NULL,
+     nspec = 2, pos = TRUE)
+ ))

   user  system elapsed
   0.19    0.00    0.19
```

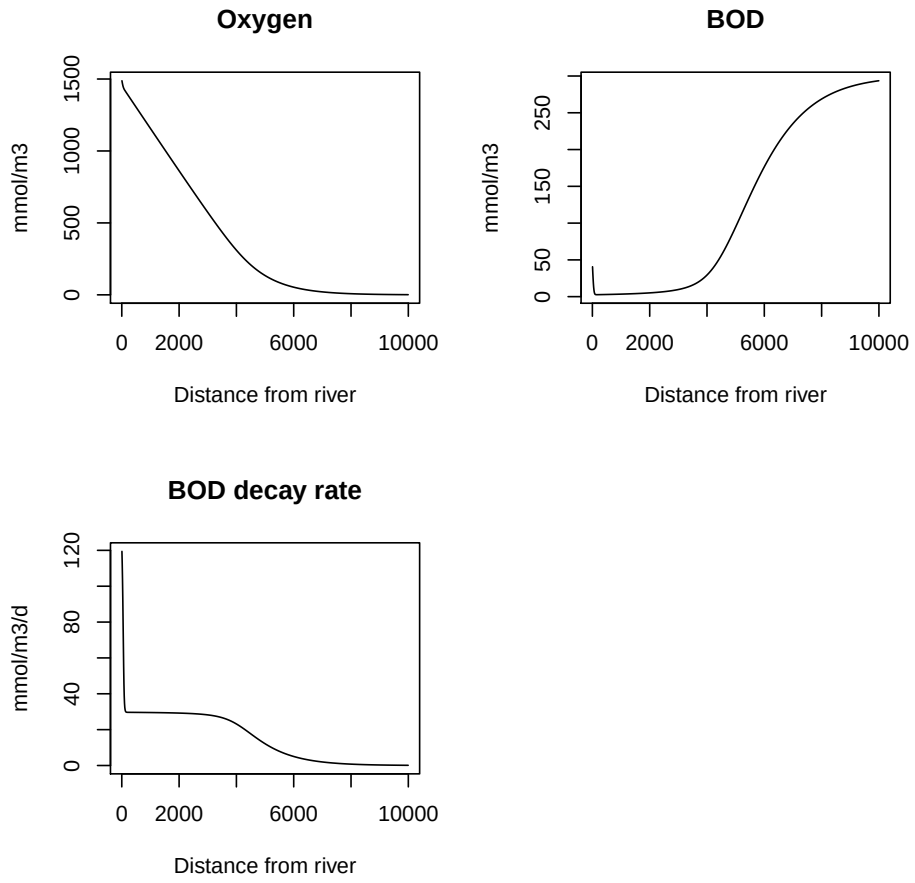

Figure 4: Steady-state solution of the BOD-O_2 model. See text for explanation

Although this model consists of 2000 nonlinear equations, it takes only 0.09 seconds to solve it [3].

Finally the results are plotted (figure 4), using `rootSolve`s plot functions:

```
> mf <- par(mfrow = c(2, 2))
> plot(out, grid = x, xlab = "Distance from river", mfrow = NULL,
+      ylab = "mmol/m3", main = c("Oxygen", "BOD"), type = "l")
> plot(out, which = "BODrate", grid = x, mfrow = NULL,
+      xlab = "Distance from river",
+      ylab = "mmol/m3/d", main = "BOD decay rate", type = "l")
> par(mfrow = mf)
```

Note: 1-D problems can also be run dynamically to steady-state. For some models this is the only way. See the help file of `steady.1D` for an example.

[3] on my computer that dates from 2008

2.4. Steady-state solution of 2-D PDEs

Function `steady.2D` efficiently finds the steady-state of 2-dimensional problems.

In the following model

$$\frac{\partial C}{\partial t} = D_x \cdot \frac{\partial^2 C}{\partial x^2} + D_y \cdot \frac{\partial^2 C}{\partial y^2} - r \cdot C^2 + p_{xy}$$

a substance C is consumed at a quadratic rate ($r \cdot C^2$), while dispersing in X- and Y-direction. At certain positions (x,y) the substance is produced (rate `p`).

The model is solved on a square (100*100) grid. There are zero-flux boundary conditions at the 4 boundaries.

The term $D_x \cdot \frac{\partial^2 C}{\partial x^2}$ is in fact shorthand for:

$$-\frac{\partial Flux}{\partial x}$$

where

$$Flux = -D_x \cdot \frac{\partial C}{\partial x}$$

i.e. it is the negative of the flux gradient, where the flux is due to diffusion.

In the numerical approximation fo the flux, the concentration gradient is approximated as the subtraction of two matrices, with the columns or rows shifted (e.g. `Conc[2:n,]-Conc[1:(n-1),]`).

The flux gradient is then also approximated by subtracting entire matrices

(e.g. `Flux[2:(n+1),]-Flux[1:(n),]`). This is very fast. The zero-flux at the boundaries is imposed by binding a column or row with 0-s.

```
> diffusion2D <- function(t, conc, par)  {
+    Conc     <- matrix(nrow = n, ncol = n, data = conc)  # vector to 2-D matrix
+    dConc    <- -r*Conc*Conc    # consumption
+    BND   <- rep(1, n)           # boundary concentration
+
+    # constant production in certain cells
+    dConc[ii]<-   dConc[ii]+  p
+
+    #diffusion in X-direction; boundaries=imposed concentration
+
+    Flux  <- -Dx * rbind(rep(0, n), (Conc[2:n,]-Conc[1:(n-1),]),
+                         rep(0, n) )/dx
+    dConc <- dConc - (Flux[2:(n+1),] - Flux[1:n,])/dx
+
+    #diffusion in Y-direction
+    Flux  <- -Dy * cbind(rep(0, n), (Conc[,2:n]-Conc[,1:(n-1)]),
+                         rep(0, n))/dy
+    dConc <- dConc - (Flux[,2:(n+1)]-Flux[,1:n])/dy
+
+    return(list(as.vector(dConc)))
+  }
```

After specifying the values of the parameters, 10 cells on the 2-D grid where there will be substance produced are randomly selected (`ii`).

```
>   # parameters
>   dy    <- dx <- 1    # grid size
>   Dy    <- Dx <- 1.5  # diffusion coeff, X- and Y-direction
>   r     <- 0.01       # 2-nd-order consumption rate (/time)
>   p     <- 20         # 0-th order production rate (CONC/t)
>   n     <- 100
>   # 10 random cells where substance is produced at rate p
>   ii    <- trunc(cbind(runif(10)*n+1, runif(10)*n+1))
```

The steady-state is found using function `steady.2D`. It takes as arguments a.o. the dimensionality of the problem (`dimens`) and `lrw=1000000`, the length of the work array needed by the solver. If this value is set too small, the solver will return with the size needed.

It takes about 0.5 second to solve this 10000 state variable model.

```
>   Conc0 <- matrix(nrow = n, ncol = n, 10.)
>   print(system.time(
+   ST3  <- steady.2D(Conc0, func = diffusion2D, parms = NULL,
+                     pos = TRUE, dimens = c(n, n), lrw = 1000000,
+                     atol = 1e-10, rtol = 1e-10, ctol = 1e-10)
+   ))

   user  system elapsed
   1.29    0.02    1.31
```

The S3 image method is used to generate the steady-state plot.

```
> image(ST3, main = "2-D diffusion+production", xlab = "x", ylab = "y",
+       legend = TRUE)
```

2.5. Steady-state solution of 3-D PDEs

Function `steady.3D` estimates the steady-state of 3-dimensional problems. We repeat the example from its help file, which models diffusion in 3-D, and with imposed boundary values.

```
> diffusion3D <- function(t, Y, par)   {
+    yy    <- array(dim=c(n,n,n),data=Y)  # vector to 3-D array
+    dY    <- -r*yy        # consumption
+    BND   <- rep(1,n)   # boundary concentration
+    for (i in 1:n) {
+      y <- yy[i,,]
+
+      #diffusion in X-direction; boundaries=imposed concentration
+      Flux <- -Dy * rbind(y[1,]-BND,(y[2:n,]-y[1:(n-1),]),BND-y[n,])/dy
```

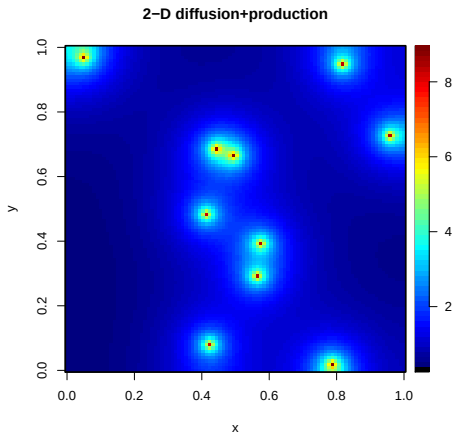

Figure 5: Steady-state solution of the nonlinear 2-Dimensional model

```
+      dY[i,,]   <- dY[i,,] - (Flux[2:(n+1),]-Flux[1:n,])/dy
+
+      #diffusion in Y-direction
+      Flux <- -Dz * cbind(y[,1]-BND,(y[,2:n]-y[,1:(n-1)]),BND-y[,n])/dz
+      dY[i,,]    <- dY[i,,] - (Flux[,2:(n+1)]-Flux[,1:n])/dz
+    }
+    for (j in 1:n) {
+      y <- yy[,j,]
+
+      #diffusion in X-direction; boundaries=imposed concentration
+      Flux <- -Dx * rbind(y[1,]-BND,(y[2:n,]-y[1:(n-1),]),BND-y[n,])/dx
+      dY[,j,]   <- dY[,j,] - (Flux[2:(n+1),]-Flux[1:n,])/dx
+
+      #diffusion in Y-direction
+      Flux <- -Dz * cbind(y[,1]-BND,(y[,2:n]-y[,1:(n-1)]),BND-y[,n])/dz
+      dY[,j,]    <- dY[,j,] - (Flux[,2:(n+1)]-Flux[,1:n])/dz
+    }
+    for (k in 1:n) {
+      y <- yy[,,k]
+
+      #diffusion in X-direction; boundaries=imposed concentration
+      Flux <- -Dx * rbind(y[1,]-BND,(y[2:n,]-y[1:(n-1),]),BND-y[n,])/dx
+      dY[,,k]   <- dY[,,k] - (Flux[2:(n+1),]-Flux[1:n,])/dx
+
+      #diffusion in Y-direction
+      Flux <- -Dy * cbind(y[,1]-BND,(y[,2:n]-y[,1:(n-1)]),BND-y[,n])/dy
+      dY[,,k]    <- dY[,,k] - (Flux[,2:(n+1)]-Flux[,1:n])/dy
+    }
+    return(list(as.vector(dY)))
```

```
+ }
>   # parameters
>   dy    <- dx <- dz <-1   # grid size
>   Dy    <- Dx <- Dz <-1   # diffusion coeff, X- and Y-direction
>   r     <- 0.025     # consumption rate
>   n  <- 10
>   y  <- array(dim=c(n, n, n), data = 10.)
>   print(system.time(
+   ST3 <- steady.3D(y, func =diffusion3D, parms = NULL, pos = TRUE,
+                    dimens = c(n, n, n), lrw = 100000,
+                    atol = 1e-10, rtol = 1e-10, ctol = 1e-10,
+                    verbose = TRUE)
+   ))

[1] "Steady-state settings"
  sparseType                                        message
1         3D sparse 3-D jacobian, calculated internally
[1] "estimated number of nonzero elements:  6910"
[1] "estimated number of function calls:  1001"
[1] "number of species:  1"
[1] "dimensions:  10 10 10"
[1] "cyclic boundaries:  0 0 0"
mean residual derivative 2.75179e-16
[1] "precision at each steady state step"
[1] 1.105000e+01 3.917674e-07 2.751792e-16
[1] ""
[1] "--------------------"
[1] " Memory requirements"
[1] "--------------------"
   par                                                  mess   val
1  nnz                         the number of nonzero elements  6400
2  ngp  the number of independent groups of state variables     12
3  nsp       the length of the work array actually required. 82366
   user  system elapsed
   0.34    0.00    0.34
```

The S3 image method is used to generate the steady-state plot. We can loop over either one of the dimensions. Here we loop over the first dimenstion, selecting a subset of the 10 cells (`dimselect`). We add contours and a legend.

```
>   image(ST3, mfrow = c(2, 2), add.contour = TRUE, legend = TRUE,
+         dimselect = list(x = c(1, 4, 8, 10)))
```

3. Boundary value problems

The previous example from functions `steady.1D, steady.2D and steady.3D` solved bundary

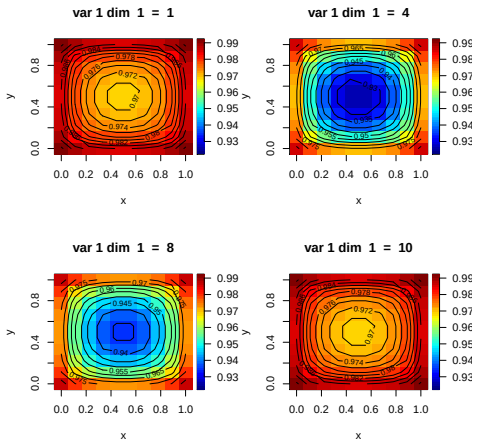

Figure 6: Steady-state solution of the 3-Dimensional model

value problems, in a way that the function call is compatible with initial value problem solvers from package **deSolve** .

Function `multiroot.1D` can also be used to solve boundary value problems of ordinary differential equations, but has a simpler function interface. It also uses the method-of-lines approach. Another package, **bvpSolve** provides two totally different methods to solve boundary values problems (Soetaert, Cash, and Mazzia 2010a).

The following differential equation:

$$0 = f(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2})$$

with boundary conditions

$y_{x=a} = ya$, at the start and $y_{x=b} = yb$ at the end of the integration interval [a,b] is solved as follows:

1. First the integration interval x is discretized,

   ```
   dx <- 0.01
   x <- seq(a,b,by=dx)
   ```

   where `dx` should be small enough, such as to keep the numerical discretisation error reasonable.

2. Then the first- and second-order derivatives are differenced on this numerical grid.

   R's `diff` function is very efficient in taking numerical differences, so it is used to approximate the first-, and second-order derivates as follows.

   A *first-order derivative y'* can be approximated either as:

   - y'=`diff(c(ya,y))/dx` if only the initial condition ya is prescribed,

- y'=`diff(c(y,yb))/dx` if only the final condition, yb is prescribed,
- y'=`0.5*(diff(c(ya,y))/dx+diff(c(y,yb))/dx)` if initial, ya, and final condition, yb are prescribed.

The latter (centered differences) is to be preferred.

A *second-order derivative y"* can be approximated by differencing twice.

y"=`diff(diff(c(ya,y,yb))/dx)/dx`

3. Finally, function `multiroot.1D` is used to locate the root.

3.1. test problem 22

As an example, the following boundary value problem will be solved:

$$\begin{aligned} \xi y'' + y' + y^2 &= 0 \\ y_{x=0} &= 0 \\ y_{x=1} &= 1/2 \end{aligned}$$

This is problem number 22 from a set of test boundary value problems which can be found at: http://www.ma.ic.ac.uk/~jcash/BVP_software/PROBLEMS.PDF.

First the function whose root has to solved is implemented:

```
> bvp22 <- function (y, xi) {
+   dy2 <- diff(diff(c(ya, y, yb))/dx)/dx
+   dy  <- 0.5*(diff(c(ya, y))/dx + diff(c(y, yb))/dx)
+
+   return(xi*dy2+dy+y^2)
+ }
```

Then the grid [0,1] is discretised (`x`) and the boundary values (`ya,yb`) defined

```
> dx <- 0.001
> x  <- seq(0, 1, by = dx)
> N  <- length(x)
> ya <- 0
> yb <- 0.5
```

The model is solved for different values of ξ and the output plotted. With the settings of dx, the root of 1001 equations needs to be found; the time it takes (in *mili*seconds) is printed for the first application.

```
> print(system.time(
+   Y1<- multiroot.1D(f = bvp22, start = runif(N), nspec = 1, xi = 0.1)
+ )*1000)

   user  system elapsed
     30       0      30
```

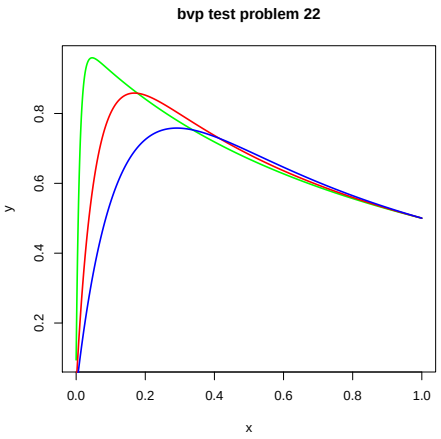

Figure 7: Solution of the boundary value problem, for three values of ξ

```
>   Y2<- multiroot.1D(f = bvp22, start = runif(N), nspec = 1, xi = 0.05)
> print(system.time(
+   Y3<- multiroot.1D(f = bvp22, start = runif(N), nspec = 1, xi = 0.01)
+ )*1000)

   user  system elapsed
     40       0      40

>

> plot(x, Y3$root, type = "l", col = "green", lwd = 2,
+      main = "bvp test problem 22" , ylab = "y")
> lines(x, Y2$root, col = "red", lwd = 2)
> lines(x, Y1$root, col = "blue", lwd = 2)
```

4. writing functions in compiled code

Similarly as for the models that are solved with integration routines from package **deSolve** , the models solved by the steady-state routines (`stode, stodes, steady, steady.1D, steady.2D, steady.3D`) can be written in compiled code (C or Fortran).

A vignette ("compiledCode") from package **deSolve** can be consulted for how to do that (Soetaert, Petzoldt, and Setzer 2008). Here the simple sediment biogeochemical model from chapter 2.1 is implemented in `C` and `Fortran`.

4.1. main function in C-code

For *code written in C*, the calling sequence for `func` must be as follows:

```
void anoxmod(int *neq, double *t, double *y, double *ydot,
double *yout, int *ip)

double OM, O2, SO4, HS;
double Min, oxicmin, anoxicmin;
if (ip[0] <1) error("nout should be at least 1");
OM  = y[0];
O2  = y[1];
SO4 = y[2];
HS  = y[3];
Min       = r*OM;
oxicmin   = Min*(O2/(O2+ks));
anoxicmin = Min*(1-O2/(O2+ks))* SO4/(SO4+ks2);

ydot[0]  = Flux - oxicmin - anoxicmin;
ydot[1]  = -oxicmin      -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2);
ydot[2]  = -0.5*anoxicmin  +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4);
ydot[3]  =  0.5*anoxicmin  -rox*HS*(O2/(O2+ks)) + D*(BHS-HS);

yout[0] = SO4+HS;
```

where `*neq` is the number of equations, `*t` is the value of the independent variable, `y` points to a double precision array of length `*neq` that contains the current value of the state variables, and `ydot` points to an array that will contain the calculated derivatives.

`yout` points to a double precision vector whose first `nout` values are other output variables (different from the state variables `y`), and the next values are double precision values as passed by parameter `rpar` when calling the steady-state solver. The key to the elements of `yout` is set in `*ip`.

`*ip` points to an integer vector whose length is at least 3; the first element contains the number of output values (which should be equal to `nout`), its second element contains the length of `*yout`, and the third element contains the length of `*ip`; next are integer values, as passed by parameter `ipar` when calling the steady-state solver.

4.2. main function in FORTRAN-code

For *code written in Fortran*, the calling sequence for `func` must be as in the following example:

```
subroutine model (neq, t, y, ydot, yout, ip)
double precision t, y(4), ydot(4), yout(*)
double precision OM,O2,SO4,HS
double precision min, oxicmin, anoxicmin

integer neq, ip(*)
double precision D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
common /myparms/D, Flux, r, rox, ks, ks2, BO2, BSO4, BHS
```

```
IF (ip(1) < 1) call rexit("nout should be at least 1")
OM  = y(1)
O2  = y(2)
SO4 = y(3)
HS  = y(4)
Min       = r*OM
oxicmin   = Min*(O2/(O2+ks))
anoxicmin = Min*(1-O2/(O2+ks))* SO4/(SO4+ks2)

ydot(1)  = Flux - oxicmin - anoxicmin
ydot(2)  = -oxicmin      -2*rox*HS*(O2/(O2+ks)) + D*(BO2-O2)
ydot(3)  = -0.5*anoxicmin  +rox*HS*(O2/(O2+ks)) + D*(BSO4-SO4
ydot(4)  =  0.5*anoxicmin  -rox*HS*(O2/(O2+ks)) + D*(BHS-HS)

yout(1) = SO4+HS
return
end
```

Note that we start by checking whether enough room is allocated for the output variables, else an error is passed to R (`rexit`) and the integration is stopped.

In this example, parameters are kept in a common block (called `myparms`) in the Fortran code

4.3. initialisation subroutine

In order to put parameters in the common block from the calling R code, an *initialisation subroutine* as specified in `initfunc` should be defined. This function has as its sole argument a function `steadyparms` that fills a double array with double precision values. In the example here, the initialisation subroutine is called `myinit`:

```
subroutine myinit(steadyparms)

external steadyparms
double precision parms(9)
common /myparms/parms

call steadyparms(9, parms)

return
end
```

Here `myinit` just calls `steadyparms` with the dimension of the parameter vector, and the array `parms` that will contain the parameter values.

The corresponding C-code is:

```
void initanox (void (* steadyparms)(int *, double *))
```

```
int N = 9;
steadyparms(&N, parms);
```

4.4. jacobian subroutine

If it is desired to supply a Jacobian to the solver, then the Jacobian must be defined in compiled code if the ode system is. The C function call for such a function must be as follows:

```
void myjac(int *neq, double *t, double *y, int *ml,
           int *mu, double *pd, int *nrowpd, double *yout, int *ip)
```

The corresponding subroutine call in Fortran is:

```
subroutine myjac (neq, t, y, ml, mu, pd, nrowpd, yout, ip)
integer neq, ml, mu, nrowpd, ip(*)
double precision y(*), pd(nrowpd,*), yout(*)
```

4.5. Estimating steady-state for models written in compiled code

To run the model using e.g. the Fortran code, the code in anoxmod.f must first be compiled.

This can be done in R itself:

`system("R CMD SHLIB anoxmod.f")`

which will create file anoxmod.dll

After loading the DLL, the model can be solved:

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

ST2 <- stode(y = y, func = "model", parms = pars,
       dllname = "anoxmod", initfunc = "myinit", pos = TRUE, nout = 1)
```

Examples in both C and Fortran are in the `dynload` subdirectory of the `rootSolve` package directory.

5. Gradients, Jacobians and Hessians

5.1. Gradient and Hessian matrices

Function `gradient` returns a forward difference approximation for the derivative of the function `f(y,...)` evaluated at the point specified by `x`.

Function `hessian` returns a forward difference approximation of the hessian matrix.

In the example below, the root of the "banana function" is first estimated (using R-function `nlm`), after which the gradient and the hessian at this point are taken.

All this can also be achieved using function `nlm`.

Note that, as `hessian` returns a (forward or centered) difference approximation of the gradient, which itself is also estimated by differencing, it is not very precise.

```
> # the banana function
>    fun <- function(x)  100*(x[2] - x[1]^2)^2 + (1 - x[1])^2
> # the minimum
>    mm  <-nlm(fun, p=c(0,0))$estimate
> # the Hessian
>    (Hes <- hessian(fun,mm))

          [,1]      [,2]
[1,]  801.9968 -399.9992
[2,] -399.9992  200.0000

> # the gradient
>    (grad <- gradient(fun,mm,centered=TRUE))

             [,1]         [,2]
[1,] -4.73282e-06 3.605689e-07

> # Hessian and gradient can also be estimated by nlm:
>    nlm(fun, p=c(0,0), hessian=TRUE)

$minimum
[1] 4.023727e-12

$estimate
[1] 0.999998 0.999996

$gradient
[1] -7.328280e-07  3.605689e-07

$hessian
          [,1]      [,2]
[1,]  802.2368 -400.0192
[2,] -400.0192  200.0000

$code
[1] 1

$iterations
[1] 26
```

The inverse of the Hessian gives an estimate of parameter uncertainty

```
>    solve(Hes)

          [,1]      [,2]
[1,] 0.4999936 0.9999853
[2,] 0.9999853 2.0049665
```

5.2. Jacobian matrices

Function `jacobian.full` and `jacobian.band` returns a forward difference approximation of the jacobian (the gradient matrix, where the function f is the derivative) for full and banded problems.

```
> mod <- function (t=0,y, parms=NULL,...)
+ {
+  dy1 <-   y[1] + 2*y[2]
+  dy2 <- 3*y[1] + 4*y[2] + 5*y[3]
+  dy3 <-          6*y[2] + 7*y[3] + 8*y[4]
+  dy4 <-                   9*y[3] +10*y[4]
+  return(as.list(c(dy1, dy2, dy3, dy4)))
+ }
> jacobian.full(y = c(1, 2, 3, 4), func = mod)

     [,1] [,2] [,3] [,4]
[1,]    1    2    0    0
[2,]    3    4    5    0
[3,]    0    6    7    8
[4,]    0    0    9   10

> jacobian.band(y = c(1, 2, 3, 4), func = mod)

     [,1] [,2] [,3] [,4]
[1,]    0    2    5    8
[2,]    1    4    7   10
[3,]    3    6    9    0
```

6. Finally

This vignette was created using Sweave (Leisch 2002).

It takes 2.2 seconds to "Sweave" the file; this is about the time needed to solve all the examples.

The computer on which this is run is an Intel Core (TM)2 Duo CPU T9300 2.5 GHz pentium PC with 3 GB of RAM

References

Hindmarsh AC (1983). "**ODEPACK**, a Systematized Collection of ODE Solvers." In R Stepleman (ed.), "Scientific Computing, Vol. 1 of IMACS Transactions on Scientific Computation," pp. 55–64. IMACS / North-Holland, Amsterdam.

Leisch F (2002). "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis." In W Härdle, B Rönz (eds.), "Compstat 2002 — Proceedings in Computational Statistics," pp. 575–580. Physica Verlag, Heidelberg. ISBN 3-7908-1517-9, URL http://www.stat.uni-muenchen.de/~leisch/Sweave.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992). *Numerical Recipes in FORTRAN. The Art of Scientific Computing*. Cambridge University Press, 2nd edition.

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

Soetaert K, Cash JR, Mazzia F (2010a). ***bvpSolve**: Solvers for Boundary Value Problems of Ordinary Differential Equations*. R package version 1.2, URL http://CRAN.R-project.org/package=bvpSolve.

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

Soetaert K, Meysman F (2010). *ReacTran: Reactive transport modelling in 1D, 2D and 3D*. R package version 1.3.

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

Soetaert K, Petzoldt T, Setzer RW (2008). *R package **deSolve**: Writing Code in Compiled Languages*. **deSolve** vignette - R package version 1.8.

Soetaert K, Petzoldt T, Setzer RW (2010c). *deSolve: General solvers for initial value problems of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE) and delay differential equations (DDE)*. R package version 1.9.

Affiliation:

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)

4401 NT Yerseke, Netherlands E-mail: karline.soetaert@nioz.nl
URL: http://www.nioz.nl

Table 1: Summary of the functions in package **rootSolve** ; values in **bold** are vectors

	Function	Description
$f(x) = 0$, $a < x < b$	`uniroot.all`	Finds many (all) roots of one (nonlinear) equation in an interval
$f(x) = 0$	`multiroot`	unconstrained root of one nonlinear equation
$\mathbf{f(x) = 0}$	`multiroot`	Finds the roots of n (nonlinear) equations
	`multiroot.1D`	Finds the roots of n (nonlinear) equations generated by discretisation of ODE, using the method-of-lines approach
$\frac{\mathbf{dx_{1..n}}}{\mathbf{dt}} = \mathbf{0}$	`stode`	Iterative steady-state solver for ordinary differential equations (ODE), assuming a full or banded Jacobian
	`stodes`	Iterative steady-state solver for ordinary differential equations (ODE), assuming an arbitrary *s*parse Jacobian
	`runsteady`	Steady-state solver for ODEs by dynamically running, assumes a full or banded Jacobian
	`steady`	General steady-state solver for ODEs; wrapper around *stode*, *stodes* and *runsteady*
	`steady.1D`	General steady-state solver for ODEs resulting from 1-dimensional partial differential equations
	`steady.band`	General steady-state solver for ODEs with banded Jacobian matrix
	`steady.2D`	General steady-state solver for ODEs resulting from 2-dimensional partial differential equations
	`steady.3D`	General steady-state solver for ODEs resulting from 3-dimensional partial differential equations
$\frac{df(\mathbf{x})}{d\mathbf{x}}$	`gradient`	Estimates the gradient matrix of a function with respect to one or more x-values
$\frac{\partial \frac{\partial(\mathbf{x})}{\partial t}}{\partial \mathbf{x}}$	`jacobian`	Estimates the jacobian matrix for a function f(x)
$\frac{\partial^2(f\mathbf{x})}{\partial x_i \partial x_j}$	`hessian`	Estimates the hessian matrix for a function f(x)