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

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

Abstract

R package **rootSolve** (Soetaert 2008) includes

- root-finding algorithms to solve for the n 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 ODE solvers in package **deSolve** , which solve initial value ODEs. Separate solvers for full, banded and generally sparse problems are included.
- Functions calculate the 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, R.

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

Package **rootSolve** deals with finding **n** 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 steady-state of systems of ordinary differential equations (ODE) and partial differential equations (PDE)
- it includes functions to numerically solve gradients.

The package was created to solve the steady-state and stability analysis examples in the book of Soetaert and Herman (2009).

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

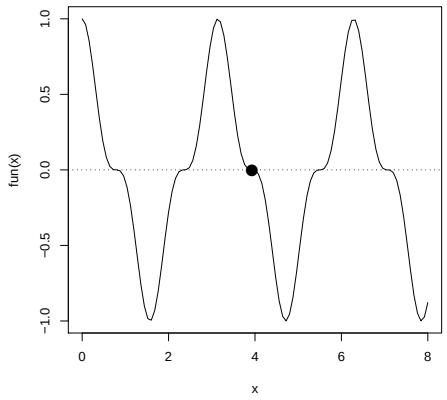

Figure 1: Root found with *uniroot*

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.

```
>   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)
```

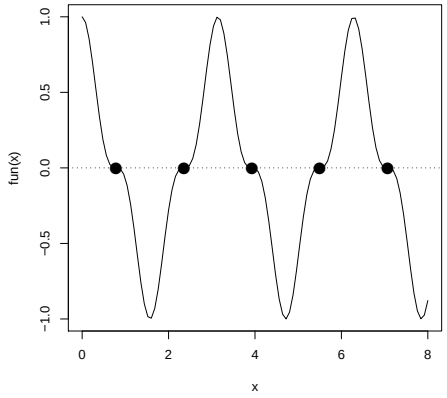

Figure 2: Roots found with uniroot.all

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 n roots of n 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:

$$\begin{aligned} f_1 &= x_1 + x_2 + x_3^2 - 12 \\ f_2 &= x_1^2 - x_2 + x_3 - 2 \\ f_3 &= 2 \cdot x_1 - x_2^2 + x_3 - 1 \end{aligned}$$

```
> 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

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

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

   user  system elapsed
  0.030   0.000   0.024

> (X<-matrix(nr=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
```

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, Petzoldt, and Setzer 2008) 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 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(
+ runsteady(y=y,fun=model,parms=pars,times=c(0,1e5))$y
+ ))
```

```
   user  system elapsed
  0.090   0.000   0.162
```

2.2. Using the Newton-Raphson method

Functions *stode*, *stodes*, *steady*, *steady.1D*, *steady.2D*, 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] with boundary conditions:
$y(1) = 1$ and $y(6) = -0.5$

The spatial derivatives are approximated using centred differences:

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

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`).

```
> 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
  0.050   0.000   0.056
```

The steady-state of this system of 5001 nonlinear equations is retrieved in about 0.03 seconds [1].

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] on my computer that dates from 2008

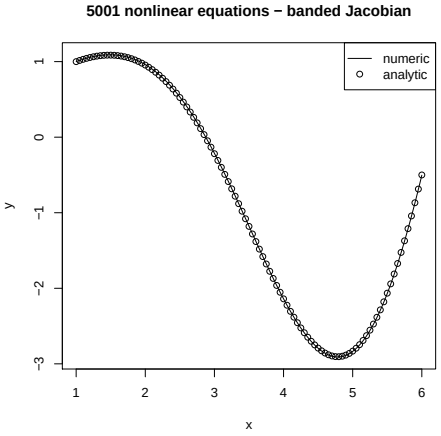

Figure 3: Solution of the 2nd order differential equation - see text for explanation

1-D models of many species

In the following model, dynamics of BOD (biochemical oxygen demand) and oxygen is modeled 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 -Flux gradient and 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
  0.100   0.000   0.104
```

Although this model consists of 2000 nonlinear equations, it takes only 0.09 seconds to solve it [2].

Finally the results are plotted (figure 4):

```
> mf <- par(mfrow=c(2,2))
> plot(x,out$y[(N+1):(2*N)],xlab= "Distance from river",
+      ylab="mmol/m3",main="Oxygen",type="l")
> plot(x,out$y[1:N],xlab= "Distance from river",
+      ylab="mmol/m3",main="BOD",type="l")
> plot(x,out$BODrate,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.

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`).

[2] on my computer that dates from 2008

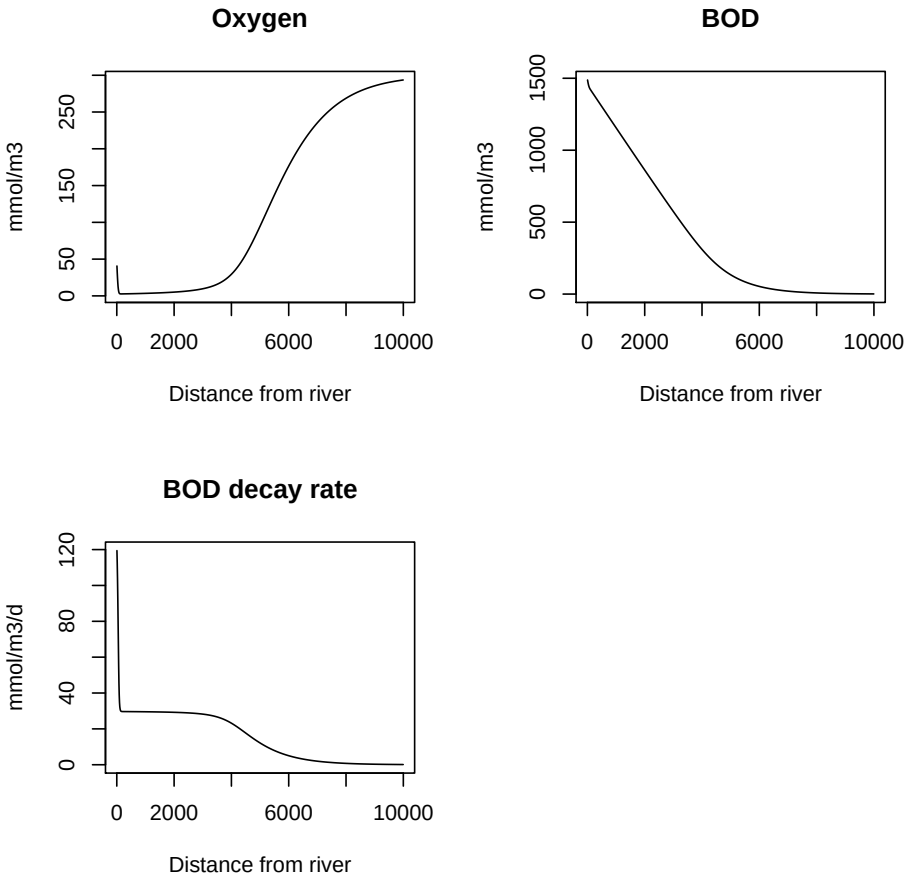

Figure 4: Steady-state solution of the BOD-O_2 model. See text for explanation

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(nr=n,nc=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`); `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.

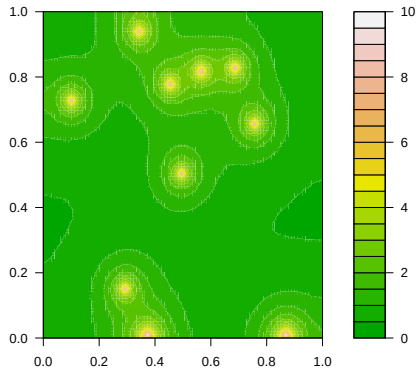

Figure 5: Steady-state solution of the nonlinear 2-Dimensional model

```
>   Conc0 <- matrix(nr=n,nc=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
  1.110   0.070   1.172

>   Conc <- matrix(nr=n,nc=n,data=ST3$y)

> filled.contour(Conc,color.palette=terrain.colors)
```

3. Gradients, Jacobians and Hessians

3.1. 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 3.605687e-07

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

$minimum
[1] 4.023726e-12

$estimate
[1] 0.999998 0.999996

$gradient
[1] -7.328277e-07  3.605687e-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
```

3.2. 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
```

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`	Solves for n roots of n (nonlinear) equations
$\frac{\mathbf{dx_{1..n}}}{\mathbf{dt}} = \mathbf{0}$	`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 multicomponent 1-dimensional reaction-transport problems
	`steady.band`	General steady-state solver for ODEs resulting from unicomponent 1-dimensional reaction-transport problems
	`steady.2D`	General steady-state solver for ODEs resulting from 2-dimensional reaction-transport problems
$\frac{df(\mathbf{x})}{d\mathbf{x}}$	`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)

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.

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 (2008). *rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations*. R package version 1.2.

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, Petzoldt T, Setzer RW (2008). *deSolve: General solvers for ordinary differential equations (ODE) and for differential algebraic equations (DAE)*. R package version 1.2.

Affiliation:

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