The rootSolve Package

September 28, 2008

Version 1.2

Title Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations

Author Karline Soetaert <k.soetaert@nioo.knaw.nl>

Maintainer Karline Soetaert <k.soetaert@nioo.knaw.nl>

Depends R (>= 2.01)

Description Routines to find the root of nonlinear functions, and to perform steady-state and equilibrium analysis of ordinary differential equations (ODE). Includes routines that: (1) generate gradient and Jacobian matrices (full and banded), (2) find roots of non-linear equations by the Newton-Raphson method, (3) estimate steady-state conditions of a system of (differential) equations in full, banded or sparse form, using the Newton-Raphson method, or by dynamically running, (4) solve the steady-state conditions for uni-and multicomponent 1-D and 2-D reactive transport models (boundary value problems of ODE) using the method of lines approach. Includes fortran code.

License GPL

LazyData yes

R topics documented:

`gradient` *Estimates the gradient matrix for a simple function*

Description

Given a vector of variables (x), and a function (f) that estimates one function value or a set of function values ($f(x)$), estimates the gadient matrix, containing, on rows i and columns j

$$d(f(x)_i)/d(x_j)$$

The gradient matrix is not necessarily square

Usage

```
gradient(f, x, centered = FALSE, ...)
```

Arguments

`f`	function returning one function value, or a vector of function values
`x`	either one value or a vector containing the x-value(s) at which the gradient matrix should be estimated
`centered`	if TRUE, uses a centered difference approximation, else a forward difference approximation
`...`	other arguments passed to function `f`

Details

the function `f` that estimates the function values will be called as f(x, ...). If `x` is a vector, then the first argument passed to `f` should also be a vector.

The gradient is estimated numerically, by perturbing the x-values.

Value

The gradient matrix where the number of rows equals the length of `f` and the number of columns equals the length of `x`.

the elements on i-th row and j-th column contain: $d((f(x))_i)/d(x_j)$

Note

`gradient` can be used to calculate so-called `sensitivity functions`, that estimate the effect of parameters on output variables.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

References

Soetaert, K. and P.M.J. Herman (2008). A practical guide to ecological modelling - using R as a simulation platform. Springer.

See Also

jacobian.full, for generating a full and **square** gradient (jacobian) matrix and where the function call is more complex

hessian, for generating the hessian matrix

Examples

```
############################################################
# 1. Sensitivity analysis of the logistic differential equation
# dN/dt = r*(1-N/K)*N  , N(t0)=N0.

# analytical solution of the logistic equation:
logistic <- function (x,times)
{
 with (as.list(x),
 {
  N=K/(1+(K-N0)/N0*exp(-r*times))
  return(c(N=N))
  })
}

# parameters for the US population from 1900
x=c(N0=76.1,r=0.02,K=500)

# Sensitivity function: SF: dfi/dxj at
# output intervals from 1900 to 1950
SF<-gradient(f=logistic,x,times=0:50)

# sensitivity, scaled for the value of the parameter:
# [dfi/(dxj/xj)]= SF*x (columnise multiplication)
sSF<-(t(t(SF)*x))
matplot(sSF,xlab="time",ylab="relative sensitivity ",
        main = "logistic equation",pch=1:3)
legend("topleft",names(x),pch=1:3,col=1:3)

# mean scaled sensitivity
colMeans(sSF)

############################################################
# 2. Stability of the budworm model, as a function of its
# rate of increase.
#
# Example from the book of Soetaert and Herman(2008)
# A practical guide to ecological modelling
# using R as a simulation platform. Springer
# code and theory are explained in this book
```

```
r   <- 0.05
K   <- 10
bet <- 0.1
alf <- 1

# density-dependent growth and sigmoid-type mortality rate
rate <- function(x,r=0.05) r*x*(1-x/K)-bet*x^2/(x^2+alf^2)

# Stability of a root ~ sign of eigenvalue of Jacobian
stability <- function (r)
{
  Eq <- uniroot.all(rate,c(0,10),r=r)
  eig  <- vector()
  for (i in 1:length(Eq))
      eig[i] <- sign (gradient(rate,Eq[i],r=r))
  return(list(Eq=Eq,Eigen=eig))
}

# bifurcation diagram
rseq <- seq(0.01,0.07,by=0.0001)

plot(0,xlim=range(rseq),ylim=c(0,10),type="n",
     xlab="r",ylab="B*",main="Budworm model, bifurcation",
     sub="Example from book of Soetaert and Herman")

for (r in rseq) {
  st <- stability(r)
  points(rep(r,length(st$Eq)),st$Eq,pch=22,
         col=c("darkblue","black","lightblue")[st$Eigen+2],
         bg =c("darkblue","black","lightblue")[st$Eigen+2])
}

legend("topleft",pch=22,pt.cex=2,c("stable","unstable"),
col=c("darkblue","lightblue"),pt.bg=c("darkblue","lightblue"))
```

`hessian` *Estimates the hessian matrix*

Description

Given a vector of variables (x), and a function (f) that estimates one function value, estimates the hessian matrix by numerical differencing. The hessian matrix is a square matrix of second-order partial derivatives of the function f with respect to x. It contains, on rows i and columns j

$$d^2(f(x))/d(x_i)/d(x_j)$$

Usage

```
hessian(f, x, centered = FALSE, ...)
```

Arguments

f	function returning one function value, or a vector of function values
x	either one value or a vector containing the x-value(s) at which the hessian matrix should be estimated
centered	if TRUE, uses a centered difference approximation, else a forward difference approximation
...	other arguments passed to function f

Details

Function hessian(f,x) returns a forward or centered difference approximation of the gradient, which itself is also estimated by differencing. Because of that, it is not very precise.

Value

The gradient matrix where the number of rows equals the length of f and the number of columns equals the length of x.
the elements on i-th row and j-th column contain: $d((f(x))_i)/d(x_j)$

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

gradient, for generating a gradient matrix

Examples

```
# the banana function
   fun <- function(x)  100*(x[2] - x[1]^2)^2 + (1 - x[1])^2
   mm  <-nlm(fun, p=c(0,0))$estimate
   (Hes <- hessian(fun,mm))
   # can also be estimated by nlm(fun, p=c(0,0), hessian=TRUE)
   solve(Hes)   # estimate of parameter uncertainty
```

---

jacobian.band *Banded jacobian matrix for a system of ODEs (ordinary differential equations)*

Description

Given a vector of (state) variables y, and a function that estimates a function value for each (state) variable (e.g. the rate of change), estimates the Jacobian matrix ($d(f(y))/d(y)$) .
Assumes a banded structure of the Jacobian matrix, i.e. where the non-zero elements are restricted to a number of bands above and below the diagonal.

Usage

```
jacobian.band(y, func, bandup=1, banddown=1,
dy=NULL, time=0, parms=NULL, ...)
```

Arguments

y	(state) variables, a vector; if y has a name attribute, the names will be used to label the jacobian matrix columns
func	function that calculates one function value for each element of y; if an ODE system, func calculates the rate of change (see details)
bandup	number of nonzero bands above the diagonal of the Jacobian matrix
banddown	number of nonzero bands below the diagonal of the Jacobian matrix
dy	reference function value; if not specified, it will be estimated by calling func
time	time, passed to function func
parms	parameter values, passed to function func
...	other arguments passed to function func

Details

The function func that estimates the rate of change of the state variables has to be consistent with functions called from R-package deSolve, which contains integration routines.

This function call is as: **function(time,y,parms,...)** where

y : (state) variable values at which the Jacobian is estimated.

parms: parameter vector - need not be used.

time: time at which the Jacobian is estimated - in general, time will not be used.

...: (optional) any other arguments

The Jacobian is estimated numerically, by perturbing the x-values.

Value

Jacobian matrix, in banded format, i.e. only the nonzero bands near the diagonal form the rows of the Jacobian.
this matrix has bandup+banddown+1 rows, while the number of columns equal the length of y.
Thus, if the full Jacobian is given by:

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

the banded jacobian will be:

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

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

jacobian.full, for a full jacobian matrix

Examples

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

---

jacobian.full *Full square jacobian matrix for a system of ODEs (ordinary differential equations)*

Description

Given a vector of (state) variables, and a function that estimates one function value for each (state) variable (e.g. the rate of change), estimates the Jacobian matrix ($d(f(x))/d(x)$)
Assumes a full and square Jacobian matrix

Usage

```
jacobian.full(y, func, dy =NULL, time=0, parms=NULL, ...)
```

Arguments

y	(state) variables, a vector; if y has a name attribute, the names will be used to label the Jacobian matrix columns
func	function that calculates one function value for each element of y; if an ODE system, func calculates the rate of change (see details)
dy	reference function value; if not specified, it will be estimated by calling func
time	time, passed to function func

parms	parameter values, passed to function func
...	other arguments passed to function func

Details

The function func that estimates the rate of change of the state variables has to be consistent with functions called from R-package deSolve, which contains integration routines.

This function call is as: **function(time,y,parms,...)** where

y : (state) variable values at which the Jacobian is estimated.

parms: parameter vector - need not be used.

time: time at which the Jacobian is estimated - in general, time will not be used.

...: (optional) any other arguments

The Jacobian is estimated numerically, by perturbing the x-values.

Value

The square jacobian matrix; the elements on i-th row and j-th column are given by: $d(f(x)_i)/d(x_j)$

Note

This function is useful for stability analysis of ODEs, which start by estimating the Jacobian at equilibrium points. The type of equilibrium then depends on the eigenvalue of the Jacobian.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

jacobian.band, for a banded jacobian matrix
gradient, for a full (not necessarily square) gradient matrix and where the function call is simpler

Examples

```
# 1. Structure of the Jacobian
#-----------------------------
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)

# 2. Stability properties of a physical model
```

```
#------------------------------
coriolis <- function (t,velocity,pars,f)
 {
  dvelx <- f*velocity[2]
  dvely <- -f*velocity[1]
  list(c(dvelx,dvely))
 }

# neutral stability; f is coriolis parameter
Jac <- jacobian.full(y=c(velx=0,vely=0),func=coriolis,
                     parms=NULL,f=1e-4)
print(Jac)
eigen(Jac)$values

# 3. Type of equilibrium
#---------------------------------------
# From Soetaert and Herman (2008). A practical guide to ecological
# modelling. Using R as a simulation platform. Springer

eqn <- function (t,state,pars)
 {
  with (as.list(c(state,pars)),
  {
  dx<-a*x + cc*y
  dy<-b*y + dd*x
  list(c(dx,dy))
  })
 }

# stable equilibrium
A<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=-0.1,b=-0.3,cc=0,dd=0)))$values
# unstable equilibrium
B<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=0.2,b=0.2,cc=0.0,dd=0.2)))$values
# saddle point
C<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=-0.1,b=0.1,cc=0,dd=0)))$values
# neutral stability
D<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=0,b=0,cc=-0.1,dd=0.1)))$values
# stable focal point
E<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=0,b=-0.1,cc=-0.1,dd=0.1)))$values
# unstable focal point
F<-eigen(jacobian.full(y=c(x=0,y=0),func=eqn,
                       parms=c(a=0.,b=0.1,cc=0.1,dd=-0.1)))$values

data.frame(type=c("stable","unstable","saddle","neutral",
           "stable focus","unstable focus"),
           eigenvalue_1=c(A[1],B[1],C[1],D[1],E[1],F[1]),
           eigenvalue_2=c(A[2],B[2],C[2],D[2],E[2],F[2]))
```

```
# 4. Limit cycles
#---------------------------------------
# From Soetaert and Herman (2008). A practical guide to ecological
# modelling. Using R as a simulation platform. Springer

eqn2 <- function (t,state,pars)
 {
  with (as.list(c(state,pars)),
  {
  dx<-  a*y   +e*x*(x^2+y^2-1)
  dy<-  b*x   +f*y*(x^2+y^2-1)
  list(c(dx,dy))
  })
 }

# stable limit cycle with unstable focus
eigen(jacobian.full(c(x=0,y=0),eqn2,parms=c(a=-1,b=1,e=-1,f=-1)))$values
# unstable limit cycle with stable focus
eigen(jacobian.full(c(x=0,y=0),eqn2,parms=c(a=-1,b=1,e=1,f=1)))$values
```

`multiroot` *Solves for n roots of n (nonlinear) equations*

Description

Given a vector of n variables, and a set of n (nonlinear) equations in these variables,

estimates the root of the equations, i.e. the variable values where all function values = 0.

Assumes a full Jacobian matrix, uses the Newton-Raphson method

Usage

```
multiroot(f, start, maxiter=100,
  rtol=1e-6, atol=1e-8, ctol=1e-8,
  useFortran=TRUE, positive=FALSE, ...)
```

Arguments

`f`	function for which the root is sought; it must return a vector with as many values as the length of `start`
`start`	vector containing initial guesses for the unknown x; if `start` has a name attribute, the names will be used to label the output vector.
`maxiter`	maximal number of iterations allowed
`rtol`	relative error tolerance, either a scalar or a vector, one value for each element in the unknown x
`atol`	absolute error tolerance, either a scalar or a vector, one value for each element in x

ctol	a scalar. If between two iterations, the maximal change in the variable values is less than this amount, then it is assumed that the root is found
useFortran	logical, if FALSE, then an R -implementation of the Newton-Raphson method is used - see details
positive	
...	additional arguments passed to function f

Details

start gives the initial guess for each variable; different initial guesses may return different roots.

The input parameters rtol, and atol determine the **error control** performed by the solver.

The solver will control the vector **e** of estimated local errors in **f**, according to an inequality of the form max-norm of (**e/ewt**) ≤ 1, where **ewt** is a vector of positive error weights. The values of rtol and atol should all be non-negative.

The form of **ewt** is:

$$\mathbf{rtol} \times \mathrm{abs}(\mathbf{f}) + \mathbf{atol}$$

where multiplication of two vectors is element-by-element.

In addition, the solver will stop if between two iterations, the maximal change in the values of **x** is less than ctol.

There is no checking whether the requested precision exceeds the capabilities of the machine.

Value

a list containing:

root	the location (x-values) of the root
f.root	the value of the function evaluated at the root
iter	the number of iterations used
estim.precis	the estimated precision for root. It is defined as the mean of the absolute function values (mean(abs(f.root)))

Note

The Fortran implementation of the Newton-Raphson method function (the default) is generally faster than the R implementation.

multiroot makes use of function steady. Technically, multiroot is just a wrapper around function steady.

The R implementation has been included for didactic purposes.

It is NOT guaranteed that the method will converge to the root.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

steady, steady.1D, and steady.band, the root solvers for a system of ordinary differential equations. Here variables are state-variables and the function estimates the rate of change.

Examples

```
# example 1
# 2 simultaneous equations
model <- function(x) c(F1=x[1]^2+ x[2]^2 -1,F2=x[1]^2- x[2]^2 +0.5)

(ss<-multiroot(f=model,start=c(1,1)))

# example 2
# 3 equations, two solutions
model <- function(x) c(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 )

# first solution
(ss<-multiroot(model,c(1,1,1),useFortran=FALSE))
(ss<-multiroot(f=model,start=c(1,1,1)))

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

# example 3: find a matrix
f2<-function(x)
 {
 X<-matrix(nr=5,x)
 X %*% X %*% X -matrix(nr=5,data=1:25,byrow=TRUE)
 }
x<-multiroot(f2, start= 1:25 )$root
X<-matrix(nr=5,x)

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

---

rootSolve-package *Roots and steady-states*

Description

Functions that:

(1) generate gradient and Jacobian matrices (full and banded),

(2) find roots of non-linear equations by the Newton-Raphson method,

(3) estimate steady-state conditions of a system of (differential) equations in full, banded or sparse form, using the Newton-Raphson method or by a dynamic run,

(4) solve the steady-state conditions for uni-and multicomponent 1-D and 2-D reactive transport models (boundary value problems of ODE) using the method-of-lines approach.

Details

Package:	rootSolve
Type:	Package
Version:	1.2
Date:	2008-09-20
License:	GNU Public License 2 or above

rootSolve is designed for solving n roots of n nonlinear equations.

Author(s)

Karline Soetaert

See Also

`uniroot.all`, to solve all roots of one equation

`multiroot`, to solve n roots of n equations

`steady`, `steady.1D`, `steady.2D` general steady-state solvers

`stode`, `stodes`, steady-state solvers for full, banded or arbitrary sparse models (Newton-Raphson method)

`runsteady` steady-state solver by dynamically running to steady-state

package vignette `rootSolve`

Examples

```
## Not run:

## run demos
demo("Jacobandroots")
demo("Steadystate")

## open the directory with documents
browseURL(paste(system.file(package="rootSolve"), "/doc", sep=""))

## main package vignette
vignette("rootSolve")
## End(Not run)
```

`runsteady` *Dynamically runs a system of ordinary differential equations (ODE) to steady-state*

Description

Solves the steady-state condition of ordinary differential equations (ODE) in the form:

$$dy/dt = f(t, y)$$

by dynamically running till the summed absolute values of the derivatives become smaller than some predefined tolerance.

The R function `runsteady` makes use of the FORTRAN ODE solver DLSODE, written by Alan C. Hindmarsh and Andrew H. Sherman

The system of ODE's is written as an R function or defined in compiled code that has been dynamically loaded. The user has to specify whether or not the problem is stiff and choose the appropriate solution method (e.g. make choices about the type of the Jacobian).

Usage

```
runsteady(y, times=c(0,Inf), func, parms, stol=1e-8, rtol=1e-6, atol=1e-6,
  jacfunc=NULL, jactype="fullint", mf=NULL, verbose=FALSE, tcrit=NULL,
  hmin=0, hmax=NULL, hini=0, ynames=TRUE, maxord=NULL, bandup=NULL,
 banddown=NULL, maxsteps=100000, dllname=NULL, initfunc=dllname,
  initpar=parms, rpar=NULL, ipar=NULL, nout=0, outnames=NULL, ...)
```

Arguments

`y`	the initial (state) values for the ODE system. If `y` has a name attribute, the names will be used to label the output matrix.
`times`	The simulation time. This should be a 2-valued vector, consisting of the initial time and the end time. The last time value should be large enough to make sure that steady-state is effectively reached in this period. The simulation will stop either when `times[2]` has been reached or when `maxsteps` have been performed.
`func`	either an R-function that computes the values of the derivatives in the ODE system (the *model definition*) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library. If `func` is an R-function, it must be defined as: `yprime = func(t, y, parms,...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `time`, and whose next elements are global values that are required at each point in `times`.
`parms`	vector or list of parameters used in `func` or `jacfunc`.
`stol`	steady-state tolerance; it is assumed that steady-state is reached if the average of absolute values of the derivatives drops below this number
`rtol`	relative error tolerance of integrator, either a scalar or an array as long as `y`. See details.

atol	absolute error tolerance of integrator, either a scalar or an array as long as y. See details.
jacfunc	if not NULL, an R function that computes the jacobian of the system of differential equations dydot(i)/dy(j), or a string giving the name of a function or subroutine in 'dllname' that computes the jacobian (see Details below for more about this option). In some circumstances, supplying jacfunc can speed up the computations, if the system is stiff. The R calling sequence for jacfunc is identical to that of func. If the jacobian is a full matrix, jacfunc should return a matrix dydot/dy, where the ith row contains the derivative of dy_i/dt with respect to y_j, or a vector containing the matrix elements by columns (the way R and Fortran store matrices). If the jacobian is banded, jacfunc should return a matrix containing only the nonzero bands of the jacobian, rotated row-wise. See first example of lsode.
jactype	the structure of the jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if mf is not NULL
mf	the "method flag" passed to function lsode - overrules jactype - provides more options than jactype - see details
verbose	if TRUE: full output to the screen, e.g. will output the settings of vectors *istate* and *rstate* - see details
tcrit	if not NULL, then lsode cannot integrate past tcrit. The Fortran routine lsode overshoots its targets (times points in the vector times), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in tcrit.
hmin	an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don't use hmin if you don't know why!
hmax	an optional maximum value of the integration stepsize. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified
hini	initial step size to be attempted; if 0, the initial step size is determined by the solver
ynames	if FALSE: names of state variables are not passed to function func ; this may speed up the simulation
maxord	the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method (meth=1), order 5 if BDF method (meth=2). Reduce maxord to save storage space
bandup	number of non-zero bands above the diagonal, in case the jacobian is banded
banddown	number of non-zero bands below the diagonal, in case the jacobian is banded
maxsteps	maximal number of steps. The simulation will stop either when maxsteps have been performed or when times[2] has been reached.
dllname	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in func and jacfunc. See help of stode.

initfunc	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See help of stode.
initpar	only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (fortran) or global variables (C, C++)
rpar	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc
ipar	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc
nout	only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculed in the dll - you have to perform this check in the code - See package vignette.
outnames	only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library
...	additional arguments passed to func and jacfunc allowing this to be a generic function

Details

The work is done by the Fortran subroutine dlsode, whose documentation should be consulted for details (it is included as comments in the source file 'src/lsode.f'). The implementation is based on the November, 2003 version of lsode, from Netlib.

Before using runsteady, the user has to decide whether or not the problem is stiff.

If the problem is nonstiff, use method flag mf = 10, which selects a nonstiff (Adams) method, no Jacobian used..
If the problem is stiff, there are four standard choices which can be specified with jactype or mf.

The options for **jactype** are

jactype = "fullint" : a full jacobian, calculated internally by lsode, corresponds to mf=22

jactype = "fullusr" : a full jacobian, specified by user function jacfunc, corresponds to mf=21

jactype = "bandusr" : a banded jacobian, specified by user function jacfunc; the size of the bands specified by bandup and banddown, corresponds to mf=24

jactype = "bandint" : a banded jacobian, calculated by lsode; the size of the bands specified by bandup and banddown, corresponds to mf=25

More options are available when specifying **mf** directly.
The legal values of mf are 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 24, 25.
mf is a positive two-digit integer, mf = (10*METH + MITER), where

METH indicates the basic linear multistep method: METH = 1 means the implicit Adams method. METH = 2 means the method based on backward differentiation formulas (BDF-s).

MITER indicates the corrector iteration method: MITER = 0 means functional iteration (no Jacobian matrix is involved). MITER = 1 means chord iteration with a user-supplied full (NEQ by NEQ) Jacobian. MITER = 2 means chord iteration with an internally generated (difference quotient) full Jacobian (using NEQ extra calls to `func` per df/dy value). MITER = 3 means chord iteration with an internally generated diagonal Jacobian approximation (using 1 extra call to `func` per df/dy evaluation). MITER = 4 means chord iteration with a user-supplied banded Jacobian. MITER = 5 means chord iteration with an internally generated banded Jacobian (using ML+MU+1 extra calls to `func` per df/dy evaluation).

If MITER = 1 or 4, the user must supply a subroutine `jacfunc`.

Inspection of the example below shows how to specify both a banded and full jacobian.

The input parameters `rtol`, and `atol` determine the **error control** performed by the solver.

See `stode` for details.

Models may be defined in compiled C or Fortran code, as well as in an R-function. See function `stode` for details.

The output will have the **attributes *istate*, and *rstate***, two vectors with several useful elements.
if `verbose` = TRUE, the settings of istate and rstate will be written to the screen.

the following elements of **istate** are meaningful:

el 1 : returns the conditions under which the last call to the integrator returned. 2 if lsode was successful, -1 if excess work done, -2 means excess accuracy requested. (Tolerances too small), -3 means illegal input detected. (See printed message.), -4 means repeated error test failures. (Check all input), -5 means repeated convergence failures. (Perhaps bad Jacobian supplied or wrong choice of MF or tolerances.), -6 means error weight became zero during problem. (Solution component i vanished, and atol or atol(i) = 0.)

el 12 : The number of steps taken for the problem so far.

el 13 : The number of evaluations for the problem so far.,

el 14 : The number of Jacobian evaluations and LU decompositions so far.,

el 15 : The method order last used (successfully).,

el 16 : The order to be attempted on the next step.,

el 17 : if el 1 =-4,-5: the largest component in the error vector,

rstate contains the following:

1: The step size in t last used (successfully).

2: The step size to be attempted on the next step.

3: The current value of the independent variable which the solver has actually reached, i.e. the current internal mesh point in t.

4: A tolerance scale factor, greater than 1.0, computed when a request for too much accuracy was detected.

For more information, see the comments in the original code lsode.f

Value

A list containing

y	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If y has a names attribute, it will be used to label the output values.
...	the number of "global" values returned

The output will have the attribute steady, which returns TRUE, if steady-state has been reached, the attribute precis with the precision attained at the last iteration estimated as the mean absolute rate of change (sum(abs(dy))/n), the attribute time with the simulation time reached and the attribute steps with the number of steps performed.

The output will also have the attributes istate, and rstate, two vectors with several useful elements of the dynamic simulation. See details. The first element of istate returns the conditions under which the last call to the integrator returned. Normal is istate[1] = 2. If verbose = TRUE, the settings of istate and rstate will be written to the screen

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

References

Alan C. Hindmarsh, "ODEPACK, A Systematized Collection of ODE Solvers," in Scientific Computing, R. S. Stepleman, et al., Eds. (North-Holland, Amsterdam, 1983), pp. 55-64.

See Also

stode, for steady-state estimation using the Newton-Raphson method, when the jacobian matrix is banded or full.

stodes, for steady-state estimation using the Newton-Raphson method, when the jacobian matrix is sparse.

steady.band, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need NOT to be rearranged

steady.1D, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need to be rearranged

Examples

```
# A simple sediment biogeochemical model

model<-function(t,y,pars)
{
```

```
with (as.list(c(y,pars)),{

  Min       = r*OM
  oxicmin   = Min*(O2/(O2+ks))
  anoxicmin = Min*(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)
})
}

# parameter values
pars <- c(D=1,Flux=100,r=0.1,rox =1,
          ks=1,ks2=1,BO2=100,BSO4=10000,BHS = 0)
# initial conditions
y<-c(OM=1,O2=1,SO4=1,HS=1)

# direct iteration
print( system.time(ST <- stode(y=y,fun=model,parms=pars,pos=TRUE)))

print( system.time(
ST2 <- runsteady(y=y,fun=model,parms=pars,times=c(0,1000))))

rbind("Newton Raphson"=ST$y,"Runsteady"=ST2$y)
```

`steady.1D` *Steady-state solver for multicomponent 1-D ordinary differential equations*

Description

Estimates the steady-state condition for a system of ordinary differential equations that result from 1-Dimensional reaction-transport models that include transport only between adjacent layers and that model many species.

Usage

```
steady.1D(y, time=0, func, parms=NULL, nspec=NULL, dimens=NULL,
                method = "stode",...)
```

Arguments

`y`	the initial guess of (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.

time	time for which steady-state is wanted; the default is time=0
func	either an R-function that computes the values of the derivatives in the ode system (the model defininition) at time time, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is an R-function, it must be defined as: yprime = func(t, y, parms, ...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values whose steady-state value is also required.
parms	parameters passed to func
nspec	the number of *species* (components) in the model. If NULL, then dimens should be specified
dimens	the number of *boxes* in the model. If NULL, then nspec should be specified
method	the solution method, one of "stode", "stodes" or "runsteady"
...	additional arguments passed to the solver function as defined by method

Details

This is the method of choice for multi-species 1-dimensional models, that are only subjected to transport between adjacent layers
More specifically, this method is to be used if the state variables are arranged per species:
A[1],A[2],A[3],....B[1],B[2],B[3],.... (for species A, B))

Two methods are implemented.

The default method rearranges the state variables as A[1],B[1],...A[2],B[2],...A[3],B[3],.... This reformulation leads to a banded Jacobian with (upper and lower) half bandwidth = number of species. Then function stode solves the banded problem.

The second method uses function stodes. Based on the dimension of the problem, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is only occurring between adjacent layers. Then stodes is called to solve the problem.
As stodes is used to estimate steady-state, it may be necessary to specify the length of the real work array, lrw.
Although a reasonable guess of lrw is made, it is possible that this will be too low. In this case, steady.1D will return with an error message telling the size of the work array actually needed. In the second try then, set lrw equal to this number.

For single-species 1-D models, use steady.band.

If state variables are arranged as (e.g. A[1],B[1],A[2],B[2],A[3],B[3],... then the model should be solved with steady.band

Value

A list containing

y	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If y has a names attribute, it will be used to label the output values.
...	the number of "global" values returned

The output will have the attribute steady, which returns TRUE, if steady-state has been reached and the attribute precis with the precision attained during each iteration.

Note

It is advisable though not mandatory to specify BOTH nspec and dimens. In this case, the solver can check whether the input makes sense (i.e. if nspec*dimens = length(y))

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

stode and stodes for the additional options

steady, for solving steady-state when the jacobian matrix is full

steady.2D, for steady-state estimation of 2-D models

steady.band, for steady-state solution, when the jacobian matrix is banded

Examples

```
######################################################################
######  EXAMPLE 1: BOD + O2                                     ######
######################################################################
# Biochemical Oxygen Demand (BOD) and oxygen (O2) dynamics
# in a river

#==================#
# Model equations  #
#==================#
O2BOD <- function(t,state,pars)

{
  BOD <- state[1:N]
  O2  <- state[(N+1):(2*N)]

# BOD dynamics
  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))

 }    # END O2BOD


#==================#
# Model application#
#==================#
# parameters
dx      <- 100       # grid size, meters
v       <- 1e2       # velocity, m/day
x       <- seq(dx/2,10000,by=dx)  # m, distance from river
N       <- length(x)
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

# initial guess:
state <- c(rep(200,N),rep(200,N))

# running the model
print(system.time(
 out   <- steady.1D (y=state,func=O2BOD,parms=NULL, nspec=2,pos=TRUE)))

#==================#
# Plotting output  #
#==================#
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)

# same, but now running dynamically to steady-state
print(system.time(
out   <- steady.1D (y=state,func=O2BOD,parms=NULL, nspec=2,
                    time=c(0,1000),method="runsteady")))

######################################################################
######  EXAMPLE 2: Silicate diagenesis                          ######
######################################################################
# Example from the book:
# Soetaert and Herman (2008).
# a practical guide to ecological modelling -
```

```
# using R as a simulation platform.
# Springer

#====================#
# Model equations    #
#====================#

SiDIAmodel <- function (time=0,     # time, not used here
                        Conc,      # concentrations: BSi, DSi
                        parms=NULL) # parameter values; not used
{
 BSi<- Conc[1:N]
 DSi<- Conc[(N+1):(2*N)]

# transport
# diffusive fluxes at upper interface of each layer

# upper concentration imposed (bwDSi), lower: zero gradient
 DSiFlux <- -SedDisp *   IntPor *diff(c(bwDSi ,DSi,DSi[N]))/thick
 BSiFlux <- -Db      *(1-IntPor)*diff(c(BSi[1],BSi,BSi[N]))/thick

 BSiFlux[1] <- BSidepo                # upper boundary flux is imposed

# BSi dissolution
 Dissolution <- rDissSi * BSi*(1.- DSi/EquilSi )^pow
 Dissolution <- pmax(0,Dissolution)

# Rate of change= Flux gradient, corrected for porosity + dissolution
 dDSi     <- -diff(DSiFlux)/thick/Porosity      +    # transport
             Dissolution * (1-Porosity)/Porosity    # biogeochemistry

 dBSi     <- -diff(BSiFlux)/thick/(1-Porosity)  - Dissolution

 return(list(c(dBSi=dBSi,dDSi=dDSi), # Rates of changes
        Dissolution=Dissolution,     # Profile of dissolution rates
        DSiSurfFlux =DSiFlux[1],     # DSi sediment-water exchange rate
        DSIDeepFlux =DSiFlux[N+1],   # DSi deep-water (burial) flux
        BSiDeepFlux =BSiFlux[N+1]))  # BSi deep-water (burial) flux
}

#====================#
# Model run          #
#====================#
# sediment parameters
thick    <- 0.1                      # thickness of sediment layers (cm)
Intdepth <- seq(0,10,by=thick)       # depth at upper interface of layers
Nint     <- length(Intdepth)         # number of interfaces
Depth    <- 0.5*(Intdepth[-Nint] +Intdepth[-1]) # depth at middle of layers
N        <- length(Depth)                       # number of layers

por0    <- 0.9                        # surface porosity (-)
pordeep <- 0.7                        # deep porosity    (-)
porcoef <- 2                          # porosity decay coefficient  (/cm)
```

```
# porosity profile, middle of layers
Porosity <- pordeep + (por0-pordeep)*exp(-Depth*porcoef)
# porosity profile, upper interface
IntPor   <- pordeep + (por0-pordeep)*exp(-Intdepth*porcoef)

dB0      <- 1/365           # cm2/day      - bioturbation coefficient
dBcoeff  <- 2
mixdepth <- 5               # cm
Db       <- pmin(dB0,dB0*exp(-(Intdepth-mixdepth)*dBcoeff))

# biogeochemical parameters
SedDisp  <- 0.4             # diffusion coefficient, cm2/d
rDissSi  <- 0.005           # dissolution rate, /day
EquilSi  <- 800             # equilibrium concentration
pow      <- 1
BSidepo  <- 0.2*100          # nmol/cm2/day
bwDSi    <- 150              # mmol/m3

# initial guess of state variables-just random numbers between 0,1
Conc     <- runif(2*N)

# three runs with different deposition rates
BSidepo  <- 0.2*100          # nmol/cm2/day
sol  <- steady.1D (Conc, func=SiDIAmodel, parms=NULL, nspec=2)
CONC <- sol$y

BSidepo  <- 2*100          # nmol/cm2/day
sol2 <- steady.1D (Conc, func=SiDIAmodel, parms=NULL, nspec=2)
CONC <- cbind(CONC,sol2$y)

BSidepo  <- 3*100          # nmol/cm2/day
sol3 <- steady.1D (Conc, func=SiDIAmodel, parms=NULL, nspec=2)
CONC  <- cbind(CONC,sol3$y)

DSi  <- CONC[(N+1):(2*N),]
BSi  <- CONC[1:N,]

#====================#
# plotting output    #
#====================#
par(mfrow=c(2,2))

matplot(DSi,Depth,ylim=c(10,0),xlab="mmolSi/m3 Liquid",
        main="DSi",type="l",lwd=c(1,2,1),col="black")
matplot(BSi,Depth,ylim=c(10,0),xlab="mmolSi/m3 Solid" ,
        main="BSi",type="l",lwd=c(1,2,1),col="black")
legend("right",c("0.2","2","3"),title="mmol/m2/d",
       lwd=c(1,2,1),lty=1:3)
plot(Porosity,Depth,ylim=c(10,0),xlab="-" ,
     main="Porosity",   type="l",lwd=2)
plot(Db,Intdepth,ylim=c(10,0),xlab="cm2/d",
     main="Bioturbation",type="l",lwd=2)
mtext(outer=TRUE,side=3,line=-2,cex=1.5,"SiDIAmodel")
```

steady.2D *Steady-state solver for 2-Dimensional ordinary differential equations*

Description

Estimates the steady-state condition for a system of ordinary differential equations that result from 2-Dimensional reaction-transport models that include transport only between adjacent layers

Usage

```
steady.2D(y, time=0, func, parms=NULL, nspec=NULL, dimens,...)
```

Arguments

y	the initial guess of (state) values for the ODE system, a vector. If y has a name attribute, the names will be used to label the output matrix.
time	time for which steady-state is wanted; the default is time=0
func	either an R-function that computes the values of the derivatives in the ode system (the model definition) at time time, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is an R-function, it must be defined as: yprime = func(t, y, parms,...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values whose steady-state value is also required.
parms	parameters passed to func
nspec	the number of *species* (components) in the model.
dimens	a 2-valued vector with the dimensionality of the model, i.e. the number of *boxes* in x- and y-direction
...	additional arguments passed to function stodes

Details

This is the method of choice for 2-dimensional models, that are only subjected to transport between adjacent layers.

Based on the dimension of the problem, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is onely occurring between adjacent layers. Then stodes is called to find the steady-state.

As stodes is used, it will probably be necessary to specify the length of the real work array, lrw.

Although a reasonable guess of lrw is made, it is likely that this will be too low. In this case, steady.2D will return with an error message telling the size of the work array actually needed. In the second try then, set lrw equal to this number.

See stodes for the additional options

Value

A list containing

y	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If y has a names attribute, it will be used to label the output values.
...	the number of "global" values returned

The output will have the attribute steady, which returns TRUE, if steady-state has been reached and the attribute precis with the precision attained during each iteration.

Note

It is advisable though not mandatory to specify BOTH nspec and dimens. In this case, the solver can check whether the input makes sense (as nspec*dimens[1]*dimens[2] = length(y))

do NOT use this method for problems that are not 2D

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

stodes for the additional options

steady, for solving steady-state when the jacobian matrix is full

steady.1D, for solving steady-state for 1-D models

steady.2D, for steady-state estimation of 2-D models

steady.band, for steady-state solution, when the jacobian matrix is banded

Examples

```
############################################################
# Diffusion in 2-D; imposed boundary conditions
############################################################
diffusion2D <- function(t,Y,par)
  {
   y    <- matrix(nr=n,nc=n,data=Y)  # vector to 2-D matrix
   dY   <- -r*y        # consumption
   BND   <- rep(1,n)   # boundary concentration

   #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   <- dY - (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    <- dY - (Flux[,2:(n+1)]-Flux[,1:n])/dy
```

```
 return(list(as.vector(dY)))
}

# parameters
dy    <- dx <- 1   # grid size
Dy    <- Dx <- 1   # diffusion coeff, X- and Y-direction
r     <- 0.025     # consumption rate

n  <- 100
y  <- matrix(nr=n,nc=n,10.)

ST3 <- steady.2D(y,func=diffusion2D,parms=NULL,pos=TRUE,dimens=c(n,n),
               lrw=1000000,atol=1e-10,rtol=1e-10,ctol=1e-10)
y <- matrix(nr=n,nc=n,data=ST3$y)
  filled.contour(y,color.palette=terrain.colors)
```

`steady` *General steady-state solver for a set of ordinary differential equations*

Description

Estimates the steady-state condition for a system of ordinary differential equations.
This is a wrapper around steady-state solvers `stode` and `stodes`.

Usage

```
steady(y, time=0, func, parms=NULL, method="stode",...)
```

Arguments

`y`	the initial guess of (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
`time`	time for which steady-state is wanted; the default is time=0
`func`	either an R-function that computes the values of the derivatives in the ode system (the model definination) at time `time`, or a character string giving the name of a compiled function in a dynamically loaded shared library. If `func` is an R-function, it must be defined as: `yprime = func(t, y, parms,...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `time`, and whose next elements are global values whose steady-state value is also required.
`parms`	parameters passed to `func`
`method`	the solution method to use, one of `stode`, `stodes` or `runsteady`
`...`	additional arguments passed to function `stode`, `stodes` or `runsteady`

Details

This is simply a wrapper around the various steady-state solvers.
See help file of stode for information about specifying the model in compiled code.
See the selected solver for the additional options

Value

A list containing

y	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If y has a names attribute, it will be used to label the output values.
...	the number of "global" values returned

The output will have the attribute steady, which returns TRUE, if steady-state has been reached and the attribute precis with the precision attained during each iteration.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

stode and stodes for the additional options

steady.1D, for steady-state estimation of 1-D models

steady.2D, for steady-state estimation of 2-D models

steady.band, for solving steady-state when the jacobian matrix is banded

Examples

```
########################################
### Bacteria growing on a substrate
########################################

  # Bacteria (Bac) are growing on a substrate (Sub)
  model <- function(t,state,pars)
  {
  with (as.list(c(state,pars)), {
  #       substrate uptake              death  respiration
  dBact = gmax*eff*Sub/(Sub+ks)*Bact - dB*Bact - rB*Bact
  dSub  =-gmax    *Sub/(Sub+ks)*Bact + dB*Bact          +input

  return(list(c(dBact,dSub)))
                                  })
  }

  pars <- list(gmax =0.5,eff = 0.5,
               ks =0.5, rB =0.01, dB =0.01, input=0.1)
  # Newton-Raphson
```

```
steady(y=c(Bact=0.1,Sub=0),time=0,
       func=model,parms=pars,pos=TRUE)

# Dynamic run to steady-state
as.data.frame(steady(y=c(Bact=0.1,Sub=0),time=c(0,1e5),
       func=model,parms=pars,method="runsteady"))
```

`steady.band` *Steady-state solver for ordinary differential equations; assumes a banded jacobian*

Description

Estimates the steady-state condition for a system of ordinary differential equations.
Assumes a banded jacobian matrix, but does not rearrange the state variables. This is in contrast to `steady.1D`. Suitable for 1-D models that include transport only between adjacent layers and that model only one species

Usage

```
steady.band(y, time=0, func, parms=NULL,
   nspec=NULL, bandup=nspec, banddown=nspec, ...)
```

Arguments

`y`	the initial guess of (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
`time`	time for which steady-state is wanted; the default is `time`=0
`func`	either an R-function that computes the values of the derivatives in the ode system (the model definition) at time `time`, or a character string giving the name of a compiled function in a dynamically loaded shared library. If `func` is an R-function, it must be defined as: `yprime = func(t, y, parms,...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `time`, and whose next elements are global values whose steady-state value is also required.
`parms`	parameters passed to `func`
`nspec`	the number of *species* (components) in the model.
`bandup`	the number of nonzero bands above the Jacobian diagonal
`banddown`	the number of nonzero bands below the Jacobian diagonal
`...`	additional arguments passed to function `stode`

Details

This is the method of choice for single-species 1-D models.

For multi-species 1-D models, this method can only be used if the state variables are arranged per box, per species (e.g. A[1],B[1],A[2],B[2],A[3],B[3],.... for species A, B).

Usually a 1-D *model* function will have the species arranged as A[1],A[2],A[3],....B[1],B[2],B[3],.... in this case, use `steady.1D`

Value

A list containing

`y`	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If `y` has a names attribute, it will be used to label the output values.
`...`	the number of "global" values returned

The output will have the attribute `steady`, which returns TRUE, if steady-state has been reached and the attribute `precis` with the precision attained during each iteration.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

`stode` for the additional options

`steady`, for solving steady-state when the jacobian matrix is full

`steady.1D`, for solving steady-state of 1-D models

`steady.2D`, for solving steady-state of 2-D models

Examples

```
############################################################
# 1000 simultaneous equations, solved 6 times for different
# values of parameter "decay"
############################################################
  model <- function (time,OC,parms,decay)
  {
   # model of particles (OC) that sink out of the water and decay
   # exponentially declining sinking rate, maximal 100 m/day
   sink <- 100*exp(-0.01*dist)

   # boundary flux; upper boundary=imposed concentration (100)
   Flux <- sink * c(100 ,OC)

   # Rate of change= Flux gradient and first-order consumption
   dOC  <- -diff(Flux)/dx - decay*OC
   list(dOC,maxC=max(OC))
```

```
  }
  dx   <- 1                          # thickness of boxes
  dist <- seq(0,1000,by=dx)          # water depth at each modeled box interface

  ss   <- NULL
  for (decay in seq(from=0.1,to=1.1,by=0.2))
    ss   <- cbind(ss,steady.band(runif(1000),func=model,
    parms=NULL,nspec=1,decay=decay)$y)

  matplot(ss,1:1000,type="l",lwd=2,main="steady.band", ylim=c(1000,0),
          ylab="water depth, m", xlab="concentration of sinking particles")
  legend("bottomright",legend=seq(from=0.1,to=1.1,by=0.2),lty=1:10,
          title="decay rate",col=1:10,lwd=2)

##############################################################
# 5001 simultaneous equations: solve
# dy/dt = 0 = d2y/dx2 + 1/x*dy/dx + (1-1/(4x^2)y - sqrx(x)*cos(x),
# over the interval [1,6]
# with boundary conditions: y(1)=1, y(6)=-0.5
##############################################################

  derivs <- function(t,y,parms, x,dx,N,y1,y6)
  {

  # Numerical approximation of derivates:
  # d2y/dx2 = (yi+1-2yi+yi-1)/dx^2
   d2y <- (c(y[-1],y6) -2*y + c(y1,y[-N])) /dx/dx

  # dy/dx = (yi+1-yi-1)/(2dx)
   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))
  }

dx     <- 0.001
x      <- seq(1,6,by=dx)
N      <- length(x)
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
plot(x,y,type="l",main="5001 nonlinear equations - banded Jacobian")

# add the analytic solution for comparison:
xx     <- seq(1,6,by=0.1)
ana <- 0.0588713*cos(xx)/sqrt(xx)+1/4*sqrt(xx)*cos(xx)+
       0.740071*sin(xx)/sqrt(xx)+1/4*xx^(3/2)*sin(xx)
points(xx,ana)
legend("topright",pch=c(NA,1),lty=c(1,NA),c("numeric","analytic"))
```

stode *Iterative steady-state solver for ordinary differential equations (ODE) and a full or banded Jacobian.*

Description

Estimates the steady-state condition for a system of ordinary differential equations (ODE) written in the form:

$$dy/dt = f(t, y)$$

i.e. finds the values of y for which f(t,y) = 0.

Uses a newton-raphson method, implemented in Fortran 77.

The system of ODE's is written as an R function or defined in compiled code that has been dynamically loaded.

Usage

```
stode(y, time=0, func, parms=NULL,
  rtol=1e-6, atol=1e-8, ctol=1e-8, jacfunc=NULL,
  jactype="fullint", verbose=FALSE, bandup=1, banddown=1,
  positive=FALSE, maxiter=100, ynames=TRUE,
  dllname=NULL, initfunc=dllname, initpar=parms,
  rpar=NULL, ipar=NULL, nout=0, outnames = NULL,...)
```

Arguments

y	the initial guess of (state) values for the ode system, a vector. If y has a name attribute, the names will be used to label the output matrix.
time	time for which steady-state is wanted; the default is time=0
func	either a user-supplied function that computes the values of the derivatives in the ode system (the *model definition*) at time time, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is a user-supplied function, it must be called as: yprime = func(t, y, parms, ...). t is the time point at which the steady-state is wanted, y is the current estimate of the variables in the ode system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector of parameters (which may have a names attribute). The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements (possibly with a names attribute) are global values that are required as output. If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before stode() is called. see Details for more information.
parms	other parameters passed to func and jacfunc
rtol	relative error tolerance, either a scalar or a vector, one value for each y

atol	absolute error tolerance, either a scalar or a vector, one value for each y
ctol	if between two iterations, the maximal change in y is less than this amount, steady-state is assumed to be reached
jacfunc	if not NULL, either a user-supplied R function that estimates the Jacobian of the system of differential equations dydot(i)/dy(j), or a character string giving the name of a compiled function in a dynamically loaded shared library as provided in dllname. In some circumstances, supplying jacfunc can speed up the computations. The R calling sequence for jacfunc is identical to that of func. If the Jacobian is a full matrix, jacfunc should return a matrix dydot/dy, where the ith row contains the derivative of dy_i/dt with respect to y_j, or a vector containing the matrix elements by columns (the way R and Fortran store matrices). If the Jacobian is banded, jacfunc should return a matrix containing only the nonzero bands of the jacobian, (dydot/dy), rotated row-wise.
jactype	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr", or "bandint" - either full or banded and estimated internally or by user
verbose	if TRUE: full output to the screen, e.g. will output the steady-state settings
bandup	number of non-zero bands above the diagonal, in case the Jacobian is banded
banddown	number of non-zero bands below the diagonal, in case the jacobian is banded
positive	if TRUE, the state variables y are forced to be non-negative numbers
maxiter	maximal number of iterations during one call to the solver
ynames	if FALSE: names of state variables are not passed to function func ; this may speed up the simulation especially for multi-D models
dllname	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func and jacfunc.
initfunc	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See details.
initpar	only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++)
rpar	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc
ipar	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc
nout	only used if 'dllname' is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculed in the dll - you have to perform this check in the code - see help of daspk or lsoda
outnames	only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library
...	additional arguments passed to func and jacfunc allowing this to be a generic function

Details

The work is done by a Fortran 77 routine that implements the Newton-Raphson method. It uses code from LINPACK.

The form of the **Jacobian** can be specified by `jactype` which can take the following values:

jactype = "fullint" : a full jacobian, calculated internally by the solver, the default

jactype = "fullusr" : a full jacobian, specified by user function `jacfunc`

jactype = "bandusr" : a banded jacobian, specified by user function `jacfunc`; the size of the bands specified by `bandup` and `banddown`

jactype = "bandint" : a banded jacobian, calculated by the solver; the size of the bands specified by `bandup` and `banddown`

if `jactype`= "fullusr" or "bandusr" then the user must supply a subroutine `jacfunc`.

The input parameters `rtol`, `atol` and `ctol` determine the **error control** performed by the solver.

The solver will control the vector **e** of estimated local errors in **y**, according to an inequality of the form max-norm of (**e/ewt**) $\leq$ 1, where **ewt** is a vector of positive error weights. The values of `rtol` and `atol` should all be non-negative. The form of **ewt** is:

$$\mathbf{rtol} \times \mathrm{abs}(\mathbf{y}) + \mathbf{atol}$$

where multiplication of two vectors is element-by-element.

In addition, the solver will stop if between two iterations, the maximal change in the values of **y** is less than `ctol`.

Models may be defined in compiled C or Fortran code, as well as in R.

If `func` or `jacfunc` are a string, then they are assumed to be compiled code.

In this case, `dllname` must give the name of the shared library (without extension) which must be loaded before lsode() is called.

If `func` is a user-supplied **R-function**, it must be called as: yprime = func(t, y, parms,...). t is the `time` at which the steady-state should be estimated, `y` is the current estimate of the variables in the ode system. The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `time`, and whose next elements contains output variables whose values at steady-state are also required .

An example is given below:

```
model<-function(t,y,pars)
{
with (as.list(c(y,pars)),{
Min = r*OM
oxicmin = Min*(O2/(O2+ks))
anoxicmin = Min*(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)
})
}
```

This model can be solved as follows:

```
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)
ST <- stode(y=y,fun=model,parms=pars,pos=TRUE))
```

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.

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

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);
}
```

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(*)
```

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,fun="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.

Value

A list containing

`y`	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If `y` has a names attribute, it will be used to label the output values.
`...`	the number of "global" values returned

The output will have the attribute `steady`, which returns TRUE, if steady-state has been reached and the attribute `precis` with an estimate of the precision attained during each iteration, the mean absolute rate of change (sum(abs(dy))/n).

Note

The implementation of `stode` and substantial parts of the help file is similar to the implementation of the integration routines (e.g. `lsode`) from package deSolve.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

References

For a description of the Newton-Raphson method, e.g.

Press, WH, Teukolsky, SA, Vetterling, WT, Flannery, BP, 1996. Numerical Recipes in FORTRAN. The Art of Scientific computing. 2nd edition. Cambridge University Press.

The algorithm uses LINPACK code:

Dongarra, J.J., J.R. Bunch, C.B. Moler and G.W. Stewart, 1979. LINPACK user's guide, SIAM, Philadelphia.

See Also

stodes, for steady-state estimation using the Newton-Raphson method, when the jacobian matrix is sparse.

runsteady, for steady-state estimation by dynamically running till the derivatives become 0.

steady.band, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need NOT to be rearranged

steady.1D, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need to be rearranged

Examples

```
# 1000 simultaneous equations
model <- function (time,OC,parms,decay,ing)
{
 # model describing organic Carbon (C) in a sediment,
 # Upper boundary = imposed flux, lower boundary = zero-gradient
 Flux  <- v * c(OC[1] ,OC) +              # advection
          -Kz*diff(c(OC[1],OC,OC[N]))/dx  # diffusion;
 Flux[1]<- flux     # imposed flux

 # Rate of change= Flux gradient and first-order consumption
 dOC   <- -diff(Flux)/dx - decay*OC

 # Fraction of OC in first 5 layers is translocated to mean depth
 dOC[1:5]  <- dOC[1:5] - ing*OC[1:5]
 dOC[N/2]  <- dOC[N/2] + ing*sum(OC[1:5])
 list(dOC)
}
v    <- 0.1    # cm/yr
flux <- 10
dx   <- 0.01
N    <- 1000
dist <- seq(dx/2,by=dx,len=N)
```

```
Kz   <- 1  #bioturbation (diffusion), cm2/yr
ss   <- stode(runif(N),func=model,parms=NULL,positive=TRUE, decay=5,ing=20)

plot(ss$y[1:N],dist,ylim=rev(range(dist)),type="l",lwd=2,
xlab="Nonlocal exchange",ylab="sediment depth",main="stode, full jacobian")
```

`stodes` *Steady-state solver for ordinary differential equations (ODE) with a sparse jacobian.*

Description

Estimates the steady-state condition for a system of ordinary differential equations (ODE) in the form:

$$dy/dt = f(t, y)$$

and where the jacobian matrix df/dy has an arbitrary sparse structure.

Uses a newton-raphson method, implemented in Fortran.

The system of ODE's is written as an R function or defined in compiled code that has been dynamically loaded.

Usage

```
stodes(y, time=0, func, parms=NULL,
rtol=1e-6, atol=1e-8, ctol=1e-8, sparsetype="sparseint",verbose=FALSE,
nnz=NULL, inz=NULL, lrw=NULL, ngp=NULL, positive=FALSE, maxiter=100,
ynames=TRUE, dllname=NULL, initfunc=dllname, initpar=parms,
rpar=NULL, ipar=NULL, nout=0, outnames = NULL, ...)
```

Arguments

`y`	the initial guess of (state) values for the ode system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
`time`	time for which steady-state is wanted; the default is `time`=0
`func`	either a user-supplied function that computes the values of the derivatives in the ode system (the *model definition*) at time `time`, or a character string giving the name of a compiled function in a dynamically loaded shared library. If `func` is a user-supplied function, it must be called as: `yprime = func(t, y, parms)`. `t` is the time point at which the steady-state is wanted, `y` is the current estimate of the variables in the ode system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector of parameters (which may have a names attribute). The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `time`, and whose next elements (possibly with a `names` attribute) are global values that are required as output. If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `stodes()` is called. see Details for more information.

parms	other parameters passed to func
rtol	relative error tolerance, either a scalar or a vector, one value for each y
atol	absolute error tolerance, either a scalar or a vector, one value for each y
ctol	if between two iterations, the maximal change in y is less than this amount, steady-state is reached
sparsetype	the sparsity pattern, to date only "sparseint", sparse jacobian, estimated internally by stodes
verbose	if TRUE: full output to the screen, e.g. will output the steady-state settings
nnz	the number of nonzero elements in the sparse Jacobian (if this is unknown, use an estimate); If NULL, a guess will be made, and if not sufficient, stodes will return with a message indicating the size actually required. If a solution is found, the minimal value of nnz actually required is returned by the solver (1st element of attribute dims)
inz	(row,column) indices to the nonzero elements in the sparse Jacobian. If this is NULL, the sparsity will be determined by stodes
lrw	the length of the work array of solver; due to the sparsicity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, stodes will return with a message indicating the size actually required. Therefore, some experimentation may be necessary to estimate the value of lrw If a solution is found, the minimal value of lrw actually required is returned by the solver (3rd element of attribute dims)
ngp	number of groups of independent state variables. Due to the sparsicity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, stodes will return with a message indicating the size actually required. Therefore, some experimentation may be necessary to estimate the value of ngp If a solution is found, the minimal value of ngp actually required is returned by the solver (2nd element of attribute dims
positive	if TRUE, the state variables are forced to be non-negative numbers
maxiter	maximal number of iterations during one call to the solver
ynames	if FALSE: names of state variables are not passed to function func ; this may speed up the simulation especially for multi-D models
dllname	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func.
initfunc	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See details.
initpar	only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++)
rpar	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func
ipar	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func
nout	only used if 'dllname' is specified: the number of output variables calculated in the compiled function func, present in the shared library

outnames	only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library
...	additional arguments passed to func allowing this to be a generic function

Details

The work is done by a Fortran 77 routine that implements the Newton-Raphson method.

stodes is to be used for problems, where the Jacobian has a sparse structure.

There are two choices for the sparsity specification, depending on whether inz is present.

If matrix inz is present, the sparsity is determined by the user. inz should contain indices (row, column) to the nonzero elements in the Jacobian matrix. In this case, nnz will be set equal to the number of rows in inz

If matrix inz is NOT present, the sparsity is estimated by the solver, based on numerical differences. In this case, it is advisable to provide an estimate of the number of non-zero elements in the Jacobian (nnz) This value can be approximate; upon return the number of nonzero elements actually required will be known (1st element of attribute dims)</nnz>

Either way, the Jacobian itself is always generated by the solver (i.e. there is no provision to provide an analytic Jacobian).
This is done by perturbing simulataneously a combination of state variables that do not affect each other. This significantly reduces computing time. The number of groups with independent state variables can be given by ngp

The input parameters rtol, atol and ctol determine the **error control** performed by the solver. See help for stode for details.

Models may be defined in compiled C or Fortran code, as well as in R. See help for stode for details.

Value

A list containing

y	A vector with the state variable values from the last iteration during estimation of steady-state condition of the system of equations. If y has a names attribute, it will be used to label the output values.
...	the number of "global" values returned

The output will have the attribute steady, which returns TRUE, if steady-state has been reached and the attribute precis with an estimate of the precision attained during each iteration, the mean absolute rate of change (sum(abs(dy))/n).

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

References

For a description of the Newton-Raphson method, e.g.
Press, WH, Teukolsky, SA, Vetterling, WT, Flannery, BP, 1996. Numerical Recipes in FORTRAN. The Art of Scientific computing. 2nd edition. Cambridge University Press.

The algorithm uses linear algebra routines from the Yale sparse matrix package:
Eisenstat, S.C., Gursky, M.C., Schultz, M.H., Sherman, A.H., 1982. Yale Sparse Matrix Package. i. The symmetric codes. Int. J. Num. meth. Eng. 18, 1145-1151.

See Also

stode, steady-state solver with full or banded jacobian.

runsteady, for steady-state estimation by dynamically running till the derivatives become 0.

steady.band, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need NOT to be rearranged

steady.1D, for steady-state estimation, when the jacobian matrix is banded, and where the state variables need to be rearranged

Examples

```
# 1000 simultaneous equations
model <- function (time,OC,parms,decay,ing)
{
 # model describing C in a sediment,
 # Upper boundary = imposed flux, lower boundary = zero-gradient
 Flux  <- v * c(OC[1] ,OC) +              # advection
          -Kz*diff(c(OC[1],OC,OC[N]))/dx  # diffusion;
 Flux[1]<- flux     # imposed flux

 # Rate of change= Flux gradient and first-order consumption
 dOC   <- -diff(Flux)/dx - decay*OC

 # Fraction of OC in first 5 layers is translocated to mean depth
 # (layer N/2)
 dOC[1:5]  <- dOC[1:5] - ing*OC[1:5]
 dOC[N/2]  <- dOC[N/2] + ing*sum(OC[1:5])
 list(dOC)
}
v    <- 0.1    # cm/yr
flux <- 10
dx   <- 0.01
N    <- 1000
dist <- seq(dx/2,by=dx,len=N)
Kz   <- 1  #bioturbation (diffusion), cm2/yr
ss   <- stodes(runif(N),func=model,parms=NULL,
               positive=TRUE, decay=5,ing=20,verbose=TRUE)

plot(ss$y[1:N],dist,ylim=rev(range(dist)),type="l",lwd=2,
```

```
xlab="Nonlocal exchange",ylab="sediment depth",
main="stodes, sparse jacobian")
```

`uniroot.all` *finds many (all?) roots of one equation*

Description

The function `uniroot.all` searches the interval from lower to upper for several roots (i.e., zero's) of a function `f` with respect to its first argument.
The number of roots found will depend on the number of subintervals in which the interval is subdivided
The function calls `uniroot`, the basic R-function.

Usage

```
uniroot.all(f, interval, lower=min(interval), upper=max(interval),
   tol=.Machine$double.eps^0.2, maxiter=1000, n=100, ...)
```

Arguments

`f`	the function for which the root is sought.
`interval`	a vector containing the end-points of the interval to be searched for the root.
`lower`	the lower end point of the interval to be searched.
`upper`	the upper end point of the interval to be searched.
`tol`	the desired accuracy (convergence tolerance).
`maxiter`	the maximum number of iterations.
`n`	number of subintervals in which the root is sought
`...`	additional named or unnamed arguments to be passed to `f` (but beware of partial matching to other arguments).

Details

`f` will be called as `f(x, ...)` for a numeric value of `x`.
Run `demo(Jacobandroots)` for an example of the use of uniroot.all for steady-state analysis.
See also second example of `gradient` This example is discussed in the book by Soetaert and Herman (2008).

Value

a vector with the roots found in the interval

Note

It is not guaranteed that all roots will be recovered.
This will depend on n, the number of subintervals in which the interval is divided.
If the function "touches" the X-axis (i.e. the root is a saddle point), then this root will generally not be retrieved. (but chances of this are pretty small).

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

uniroot for more information about input

Examples

```
##############################
##   Mathematical examples  ##
##############################

# a well-behaved case...
fun <- function (x) cos(2*x)^3
curve(fun(x),0,10,main="uniroot.all")
All <- uniroot.all(fun,c(0,10))
points(All,y=rep(0,length(All)),pch=16,cex=2)

# a pathetic case...
f <- function (x)1/cos(1+x^2)
AA <- uniroot.all(f,c(-5,5))
curve(f(x),-5,5,n=500,main="uniroot.all")
points(AA,rep(0,length(AA)),col="red",pch=16)
f(AA)  # !!!

##################################
## Ecological modelling example ##
##################################

# Example from the book of Soetaert and Herman(2008)
# A practical guide to ecological modelling
# using R as a simulation platform. Springer

r   <- 0.05
K   <- 10
bet <- 0.1
alf <- 1

# the model : density-dependent growth and sigmoid-type mortality rate
rate <- function(x,r=0.05) r*x*(1-x/K)-bet*x^2/(x^2+alf^2)

# find all roots within the interval [0,10]
Eq   <- uniroot.all(rate,c(0,10))
```

```
# jacobian evaluated at all roots:
# This is just one value - and therefore jacobian = eigenvalue
# the sign of eigenvalue: stability of the root: neg=stable, 0=saddle, pos=unstable

eig <- vector()
for (i in 1:length(Eq)) eig[i] <- sign (gradient(rate,Eq[i]))

curve(rate(x),ylab="dx/dt",from=0,to=10,
main="Budworm model, roots",sub= "Example from book of Soetaert and Herman")
abline(h=0)
points(x=Eq,y=rep(0,length(Eq)),pch=21,cex=2,bg=c("grey","black","white")[eig+2] )
legend("topleft",pch=22,pt.cex=2,c("stable","saddle","unstable"),
col=c("grey","black","white"),pt.bg=c("grey","black","white"))
```

Index