Package ‘deSolve’

September 11, 2010

Version 1.8

Title General solvers for initial value problems of ordinary differential equations (ODE), partial differential equations (PDE), differential algebraic equations (DAE), and delay differential equations (DDE).

Author Karline Soetaert <k.soetaert@nioo.knaw.nl>, Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>, R. Woodrow Setzer <setzer.woodrow@epa.gov>

Maintainer R. Woodrow Setzer <setzer.woodrow@epa.gov>

Depends R (>= 2.6.0)

Description Functions that solve initial value problems of a system of first-order ordinary differential equations (ODE), of partial differential equations (PDE), of differential algebraic equations (DAE), and of delay differential equations. The functions provide an interface to the FORTRAN functions lsoda, lsodar, lsode, lsodes of the ODEPACK collection, to the FORTRAN functions dvode and daspk and a C-implementation of solvers of the Runge-Kutta family with fixed or variable time steps. The package contains routines designed for solving ODEs resulting from 1-D, 2-D and 3-D partial differential equations (PDE) that have been converted to ODEs by numerical differencing.

License GPL (>= 2)

LazyData yes

Repository CRAN

Repository/R-Forge/Project desolve

Repository/R-Forge/Revision 368

Date/Publication 2010-09-11 09:11:31
R topics documented:

deSolve-package ................................................................. 2
aquaphy ................................................................. 5
ccl4data ................................................................. 9
ccl4model ................................................................. 9
daspk ................................................................. 12
dede ................................................................. 21
diagnostics ................................................................. 26
diagnostics.deSolve ................................................................. 26
DLLfunc ................................................................. 27
DLLres ................................................................. 30
events ................................................................. 32
forcings ................................................................. 35
lsoda ................................................................. 38
lsodar ................................................................. 45
lsode ................................................................. 52
lsodes ................................................................. 59
ode ................................................................. 66
ode.1D ................................................................. 70
ode.2D ................................................................. 76
ode.3D ................................................................. 81
ode.band ................................................................. 85
plot.deSolve ................................................................. 87
radau ................................................................. 90
rk ................................................................. 97
rk4 ................................................................. 103
rkMethod ................................................................. 106
SCOC ................................................................. 111
timelags ................................................................. 112
vode ................................................................. 116
zvode ................................................................. 123

Index 129


Description

Functions that solve initial value problems of a system of first-order ordinary differential equations (ODE), of partial differential equations (PDE), of differential algebraic equations (DAE) and delay differential equations.

The functions provide an interface to the FORTRAN functions lsoda, lsodar, lsode, lsodes of the ODEPACK collection, to the FORTRAN functions dvode and daspk and a C-implementation of solvers of the Runge-Kutta family with fixed or variable time steps.
The package contains routines designed for solving ODEs resulting from 1-D, 2-D and 3-D partial differential equations (PDE) that have been converted to ODEs by numerical differencing. It includes root-finding (or event location) and provides access to lagged variables and derivatives.

Details

Package: deSolve
Type: Package
Version: 1.8
Date: 2010-09-01
License: GNU Public License 2 or above

The system of differential equations is written as an R function or defined in compiled code that has been dynamically loaded, see package vignette(vignette(compiledCode)) for details. The solvers may be used as part of a modeling package for differential equations, or for parameter estimation using any appropriate modeling tool for non-linear models in R such as optim, nls, nlm or nlme.

Author(s)

Karline Soetaert,
Thomas Petzoldt,
R. Woodrow Setzer (Maintainer)

References


See also the references given on the specific help pages of the different methods.

See Also

ode for a general interface to most of the ODE solvers,
ode.band for solving models with a banded Jacobian,
ode.1D, ode.2D, ode.3D, for integrating 1-D, 2-D and 3-D models,
dede for a general interface to the delay differential equation solvers,
lsoda, lsode, lsodes, lsodar, vode, for ODE solvers of the Livermore family,
daspk, for a DAE solver up to index 1, of the Livermore family,
radau for integrating DAEs up to index 3 using an implicit Runge-Kutta,
rk, rkMethod, rk4, euler for Runge-Kutta solvers,
DLLfunc, DLLres, for testing model implementations in compiled code,
forcings, events, for how to implement forcing functions (external variables) and events (sudden changes in state variables),
lagvalue, lagderiv, for how to get access to lagged values of state variables and derivatives.

Examples

```
## Not run:
## show examples (see respective help pages for details)
example(aquaphy)
example(lsoda)
example(code.band)
example(code.1D)
example(code.2D)
example(code.3D)
example(dede)

## run demos
demo("odedim")  # partial differential equations
demo("CCL4model")  # a model fitting example (this will take some time)

## open the directory with source code of demos
browseURL(paste(system.file(package = "deSolve"), "/demo", sep = "))

## open the directory with R sourcecode examples
browseURL(paste(system.file(package = "deSolve"), "/doc/examples", sep = "))

## open the directory with C and FORTRAN sourcecode examples
browseURL(paste(system.file(package = "deSolve"), "/doc/examples/dynload", sep = "))

## show package vignette with how to use deSolve
## + source code of the vignette
vignette("deSolve")
edit(vignette("deSolve"))

## show package vignette with tutorial about how to use compiled models
## + source code of the vignette
## + directory with C and FORTRAN sources
vignette("compiledCode")
edit(vignette("compiledCode"))
browseURL(paste(system.file(package = "deSolve"), "/doc", sep = "))

## End(Not run)
```
Description

A phytoplankton model with uncoupled carbon and nitrogen assimilation as a function of light and Dissolved Inorganic Nitrogen (DIN) concentration.

Algal biomass is described via 3 different state variables:

• low molecular weight carbohydrates (LMW), the product of photosynthesis,
• storage molecules (RESERVE) and
• the biosynthetic and photosynthetic apparatus (PROTEINS).

All algal state variables are expressed in mmol C m$^{-3}$. Only proteins contain nitrogen and chlorophyll, with a fixed stoichiometric ratio. As the relative amount of proteins changes in the algae, so does the N:C and the Chl:C ratio.

An additional state variable, dissolved inorganic nitrogen (DIN) has units of mmol N m$^{-3}$.

The algae grow in a dilution culture (chemostat): there is constant inflow of DIN and outflow of culture water, including DIN and algae, at the same rate.

Two versions of the model are included.

• In the default model, there is a day-night illumination regime, i.e. the light is switched on and off at fixed times (where the sum of illuminated + dark period = 24 hours).
• In another version, the light is imposed as a forcing function data set.

This model is written in FORTRAN.

Usage

aquaphy(times, y, parms, PAR = NULL, ...)

Arguments

times time sequence for which output is wanted; the first value of times must be the initial time,
y the initial (state) values ("DIN", "PROTEIN", "RESERVE", "LMW"), in that order,
parms vector or list with the aquaphy model parameters; see the example for the order in which these have to be defined.
PAR a data set of the photosynthetically active radiation (light intensity), if NULL, on-off PAR is used,
... any other parameters passed to the integrator ode (which solves the model).
Details

The model is implemented primarily to demonstrate the linking of FORTRAN with R-code. The source can be found in the ‘doc/examples/dynload’ subdirectory of the package.

Author(s)

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

References


See Also

cc14model, the CCl4 inhalation model.

Examples

```r
## ======================================================
## Example 1. PAR an on-off function
## ======================================================
## -----------------------------
## the model parameters:
## -----------------------------

parameters <- c(maxPhotoSynt = 0.125,  # mol C/mol C/hr
                rMortPHY = 0.001,  # /hr
                alpha = -0.125/150,  # uEinst/m2/s/hr
                pExudation = 0.0,  # -
                maxProteinSynt = 0.136,  # mol C/mol C/hr
                ksDIN = 1.0,  # mmol N/m3
                minpLMW = 0.05,  # mol C/mol C
                maxpLMW = 0.15,  # mol C/mol C
                minQuotum = 0.075,  # mol C/mol C
                maxStorage = 0.23,  # /h
                respirationRate= 0.0001,  # /h
                pResp = 0.4,  # -
                catabolismRate = 0.06,  # /h
                dilutionRate = 0.01,  # /h
                rNCProtein = 0.2,  # mol N/mol C
                inputDIN = 10.0,  # mmol N/m3
                rChlN = 1,  # g Chl/mol N
                parMean = 250.,  # umol Phot/m2/s
```

dayLength = 15. # hours

# The initial conditions
state <- c(DIN = 6., # mmol N/m3
          PROTEIN = 20.0, # mmol C/m3
          RESERVE = 5.0, # mmol C/m3
          LMW = 1.0) # mmol C/m3

# Running the model

times <- seq(0, 24*20, 1)
out <- as.data.frame(aquaphy(times, state, parameters))

# Plotting model output

par(mfrow = c(2, 2), oma = c(0, 0, 3, 0))
col <- grey(0.9)
ii <- 1:length(out$PAR)

plot(times[ii], out$Chlorophyll[ii], type = "l",
     main = "Chlorophyll", xlab = "time, hours", ylab = "ug/l")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$Chlorophyll[ii], lwd = 2)

plot (times[ii], out$DIN[ii], type = "l",
      main = "DIN", xlab = "time, hours", ylab = "mmolN/m3")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$DIN[ii], lwd = 2)

plot (times[ii], out$NCratio[ii], type = "n",
      main = "NCratio", xlab = "time, hours", ylab = "molN/molC")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$NCratio[ii], lwd = 2)

plot (times[ii], out$PhotoSynthesis[ii],
      type = "l", main = "PhotoSynthesis", xlab = "time, hours",
      ylab = "mmolC/m3/hr")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$PhotoSynthesis[ii], lwd = 2)
mtext(outer = TRUE, side = 3, "AQUAPHY, PAR= on-off", cex = 1.5)
## Summary model output

t(summary(out))

## Example 2. PAR a forcing function data set

```r
times <- seq(0, 24*20, 1)
ftime <- seq(0,500,by=0.5)
parval <- pmax(0,250 + 350*sin(ftime*2*pi/24)+(runif(length(ftime))-0.5)*250)
Par <- matrix(nc=2,c(ftime,parval))

state <- c(DIN = 6., # mmol N/m3
           PROTEIN = 20.0, # mmol C/m3
           RESERVE = 5.0, # mmol C/m3
           LMW = 1.0) # mmol C/m3

out <- aquaphy(times, state, parameters, Par)
out2 <- as.data.frame(out) # facilitates printing...

plot (times, out2$PAR, type = "l",
      main = "PAR", xlab = "time, hours",ylab = "uEinst/m2/s")

plot (times, out2$Chlorophyll, type = "l",
      main = "Chlorophyll", xlab = "time, hours",ylab = "ug/l")

plot (times, out2$DIN, type = "l", main = "DIN",
      xlab = "time, hours",ylab = "mmolN/m3")

plot (times, out2$NCratio, type = "l", main = "NCratio",
      xlab = "time, hours", ylab = "molN/molC")

mtext(outer = TRUE, side = 3, "AQUAPHY, PAR=forcing", cex = 1.5)
```

# Now all variables plotted in one figure...
plot(out, which = 1:9, type = "l")
par(mfrow = c(1, 1))
**ccl4data**

**Closed Chamber Study of CCl4 Metabolism by Rats.**

**Description**

The results of a closed chamber experiment to determine metabolic parameters for CCl4 (carbon tetrachloride) in rats.

**Usage**

```r
data(ccl4data)
```

**Format**

This data frame contains the following columns:

- **time**: the time (in hours after starting the experiment).
- **initconc**: initial chamber concentration (ppm).
- **animal**: this is a repeated measures design; this variable indicates which animal the observation pertains to.
- **ChamberConc**: chamber concentration at time, in ppm.

**Source**


**Examples**

```r
plot(ChamberConc ~ time, data = ccl4data, xlab = "Time (hours)",
     xlim = range(c(0, ccl4data$time)),
     ylab = "Chamber Concentration (ppm)", log = "y")
ccl4data.avg <- aggregate(ccl4data$ChamberConc,
                            by = ccl4data[c("time", "initconc")], mean)
points(x ~ time, data = ccl4data.avg, pch = 16)
```

**ccl4model**

**The CCl4 Inhalation Model**

**Description**

The CCl4 inhalation model implemented in Fortran

**Usage**

```r
ccl4model(times, y, parms, ...)
```
Arguments

times
time sequence for which the model has to be integrated.
y
the initial values for the state variables ("AI", "AAM", "AT", "AF", "AL", "CLT"
and "AM"), in that order.

parms
vector or list holding the ccl4 model parameters; see the example for the order
in which these have to be defined.

... any other parameters passed to the integrator ode (which solves the model).

Details

The model is implemented primarily to demonstrate the linking of FORTRAN with R-code.
The source can be found in the ‘/doc/examples/dynload’ subdirectory of the package.

Author(s)

R. Woodrow Setzer <setzer.woodrow@epa.gov>

See Also

Try demo(CCL4model) for how this model has been fitted to the dataset ccl4data,
aquaphy, another FORTRAN model, describing growth in aquatic phytoplankton.

Examples

```r
## Parameter values
## -------------

Pm <- c(
  BW = 0.182, # Body weight (kg)
  QP = 4.0 , # Alveolar ventilation rate (hr^-1)
  QC = 4.0 , # Cardiac output (hr^-1)
  VFC = 0.08, # Fraction fat tissue (kg/(kg/BW))
  VLC = 0.04, # Fraction liver tissue (kg/(kg/BW))
  VMC = 0.74, # Fraction of muscle tissue (kg/(kg/BW))
  QFC = 0.05, # Fractional blood flow to fat ((hr^-1)/QC)
  QLC = 0.15, # Fractional blood flow to liver ((hr^-1)/QC)
  QMC = 0.32, # Fractional blood flow to muscle ((hr^-1)/QC)

  PLA = 16.17, # Liver/air partition coefficient
  PFA = 281.48, # Fat/air partition coefficient
  PMA = 13.3, # Muscle/air partition coefficient
  PTA = 16.17, # Viscera/air partition coefficient
  PB = 5.487, # Blood/air partition coefficient
  MW = 153.8, # Molecular weight (g/mol)
  VMAX = 0.04321671, # Max. velocity of metabolism (mg/hr) -calibrated
  KM = 0.4027255, # Michaelis-Menten constant (mg/l) -calibrated
```

## Parameters for simulated experiment
CONC = 1000, # Inhaled concentration
KL = 0.02, # Loss rate from empty chamber /hr
RATS = 1.0, # Number of rats enclosed in chamber
VCHC = 3.8 # Volume of closed chamber (l)

---

## State variables
##

```r
c
AI <- 21, # total mass , mg
AAM = 0,
AT = 0,
AF = 0,
AL = 0,
CLT = 0, # area under the conc.-time curve in the liver
AM = 0 # the amount metabolized (AM)
```

---

## Model application

```r
times <- seq(0, 6, by = 0.1)

# initial inhaled concentration-calibrated
conc <- c(26.496, 90.197, 245.15, 951.46)

plot(ChamberConc ~ time, data = ccl4data, xlab = "Time (hours)",
     xlim = range(c(0, ccl4data$time)),
     ylab = "Chamber Concentration (ppm)",
     log = "y", main = "ccl4model")

for (cc in conc) {
Pm["CONC"] <- cc

VCH <- Pm["VCHC"] - Pm["RATS"] * Pm["BW"]
AI0 <- VCH * Pm["CONC"] * Pm["MW"]/24450
y["AI"] <- AI0

# run the model:
out <- as.data.frame(ccl4model(times, y, Pm))
lines(out$time, out$CP, lwd = 2)
}

legend("topright", lty = c(NA, 1), pch = c(1, NA), lwd = c(NA, 2),
       legend = c("data", "model"))
```

---

## An example with tracer injection

---
## every day, a conc of 2 is added to AI.
## 1. implemented as a data.frame
eventdat <- data.frame(var = rep("AI", 6), time = 1:6 ,
  value = rep(1, 6), method = rep("add", 6))

eventdat

print(system.time(
  out <- ccl4model(times, y, Pm, events = list(data = eventdat))
))

plot(out, mfrow = c(3, 4), type = "l", lwd = 2)

# 2. implemented as a function in a DLL!
print(system.time(
  out2 <- ccl4model(times, y, Pm, events = list(func = "eventfun", time = 1:6))
))

plot(out2, mfrow = c(3, 4), type = "l", lwd = 2)

daspk

---

**Solver for Differential Algebraic Equations (DAE)**

### Description

Solves either:

- a system of ordinary differential equations (ODE) of the form
  \[ y' = f(t, y, ...) \]
  
or
- a system of differential algebraic equations (DAE) of the form
  \[ F(t, y, y') = 0 \]
  
or
- a system of linearly implicit DAES in the form
  \[ My' = f(t, y) \]

using a combination of backward differentiation formula (BDF) and a direct linear system solution method (dense or banded).

The R function `daspk` provides an interface to the FORTRAN DAE solver of the same name, written by Linda R. Petzold, Peter N. Brown, Alan C. Hindmarsh and Clement W. Ulrich.

The system of DE’s is written as an R function (which may, of course, use `.C`, `.Fortran`, `.Call`, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded.
Usage
daspk(y, times, func = NULL, parms, dy = NULL, res = NULL,
   nalg = 0, rtol = 1e-6, atol = 1e-8, jacfunc = NULL,
   jacres = NULL, jactype = "fullint", mass = NULL, estini = NULL,
   verbose = FALSE, tcrit = NULL, hmin = 0, hmax = NULL,
   hini = 0, ynames = TRUE, maxord = 5, bandup = NULL,
   banddown = NULL, maxsteps = 5000, dllname = NULL,
   initfunc = dllname, initpar = parms, rpar = NULL,
   ipar = NULL, nout = 0, outnames = NULL,
   forcings=NULL, initforc = NULL, fcontrol=NULL, lags = NULL, ...)

Arguments

y
the initial (state) values for the DE system. If y has a name attribute, the names
will be used to label the output matrix.
times
time sequence for which output is wanted; the first value of times must be the
initial time; if only one step is to be taken; set times = NULL.
func
cannot be used if the model is a DAE system. If an ODE system, func should
be an R-function that computes the values of the derivatives in the ODE system
(the model definition) at time t.
func must be defined as: func <- function(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, unless ynames is FALSE. 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 con-
   taining the derivatives of y with respect to time, and whose next elements are
   global values that are required at each point in times. The derivatives should
   be specified in the same order as the specification of the state variables y.
   Note that it is not possible to define func as a compiled function in a dynami-
   cally loaded shared library. Use res instead.
   parms
   vector or list of parameters used in func, jacfunc, or res
dy
   the initial derivatives of the state variables of the DE system. Ignored if an ODE.
   if a DAE system: either an R-function that computes the residual function
   F(t, y, y') of the DAE 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 res is a user-supplied R-function, it must be defined as: res <- function(t, y, dy, parms, ...).
   Here t is the current time point in the integration, y is the current estimate of the
   variables in the ODE system, dy are the corresponding derivatives. If the
   initial y or dy have a names attribute, the names will be available inside res,
   unless ynames is FALSE. parms is a vector of parameters.
The return value of res should be a list, whose first element is a vector con-
   taining the residuals of the DAE system, i.e. δ = F(t, y, y'), and whose next
   elements contain output variables that are required at each point in times.
If res is a string, then dllname must give the name of the shared library (without extension) which must be loaded before daspk() is called (see package vignette "compiledCode" for more information).

nalg

if a DAE system: the number of algebraic equations (equations not involving derivatives). Algebraic equations should always be the last, i.e. preceded by the differential equations.
Only used if estini = 1.

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.
jacfunc
if not NULL, an R function that computes the Jacobian of the system of differential equations. Only used in case the system is an ODE (y' = f(t,y)), specified by func. The R calling sequence for jacfunc is identical to that of func. If the Jacobian is a full matrix, jacfunc should return a matrix ∂y/∂y, 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.
jacres
jacres and not jacfunc should be used if the system is specified by the residual function F(t, y, y'), i.e. jacres is used in conjunction with res.
If jacres is an R-function, the calling sequence for jacres is identical to that of res, but with extra parameter cj. Thus it should be called as: jacres = func(t, y, dy, parms, cj,...). Here t is the current time point in the integration, y is the current estimate of the variables in the ODE system, y' are the corresponding derivatives and cj is a scalar, which is normally proportional to the inverse of the stepsize. If the initial y or dy have a names attribute, the names will be available inside jacres, unless ynames is FALSE. parms is a vector of parameters (which may have a names attribute).
If the Jacobian is a full matrix, jacres should return the matrix dG/dy + cj · dG/dy', where the ith row is the sum of the derivatives of G_i with respect to y_j, and the scaled derivatives of G_i with respect to dy_j.
If the Jacobian is banded, jacres should return only the nonzero bands of the Jacobian, rotated rowwise. See details for the calling sequence when jacres is a string.
jactype
the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by the user.
mass
the mass matrix. If not NULL, the problem is a linearly implicit DAE and defined as M dy/dt = f(t, y). The mass-matrix M should be of dimension n * n where n is the number of y-values.
If mass=NULL then the model is either an ODE or a DAE, specified with res only if a DAE system, and if initial values of y and dy are not consistent (i.e. F(t, y, dy) ≠ 0), setting estini = 1 or 2, will solve for them. If estini = 1: dy and the algebraic variables are estimated from y; in this case, the number of algebraic equations must be given (nalp). If estini = 2: y will be estimated from dy.
verbose

if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

tcrit

the FORTRAN routine daspk 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

logical, if FALSE, names of state variables are not passed to function func; this may speed up the simulation especially for large models.

maxord

the maximum order to be allowed. Reduce maxord to save storage space (<= 5)

bandup

number of non-zero bands above the diagonal, in case the Jacobian is banded (and jactype one of "bandint", "bandusr")

banddown

number of non-zero bands below the diagonal, in case the Jacobian is banded (and jactype one of "bandint", "bandusr")

maxsteps

maximal number of steps per output interval taken by the solver; will be recalculated to be at least 500 and a multiple of 500; the solver will give a warning if more than 500 steps are taken, but it will continue till maxsteps steps.

dllname

a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in res and jacres. See package vignette "compiledCode".

initfunc

if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

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 (FORTTRAN) 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 res and jacres.

ipar

only when ‘dllname’ is specified: a vector with integer values passed to the dll-functions whose names are specified by res and jacres.

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 res, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

outnames

only used if ‘dllname’ is specified and nout > 0: the names of output variables calculated in the compiled function res, present in the shared library. These names will be used to label the output matrix.
forcings  only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\text{min(times)}, \text{max(times)}]\) is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc  if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol  A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

lags  A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

...  additional arguments passed to func, jacfunc, res and jacres, allowing this to be a generic function.

Details

The daspk solver uses the backward differentiation formulas of orders one through five (specified with maxord) to solve either:

- an ODE system of the form
  
  \[ y' = f(t, y, \ldots) \]

  for \( y = Y \), or

- a DAE system of the form
  
  \[ F(t, y, y') = 0 \]

  for \( y = Y \) and \( y' = YPRIME \). The index of the DAE should be <=1.

ODEs are specified in func, DAEs are specified in res.

If a DAE system, Values for \( Y \) and \( YPRIME \) at the initial time must be given as input. Ideally, these values should be consistent, that is, if \( T, Y, YPRIME \) are the given initial values, they should satisfy \( F(T, Y, YPRIME) = 0 \).

However, if consistent values are not known, in many cases daspk can solve for them: when estini = 1, \( y' \) and algebraic variables (their number specified with nalg) will be estimated, when estini = 2, \( y \) will be estimated.

The form of the Jacobian can be specified by jactype. This is one of:

- jactype = "fullint": a full Jacobian, calculated internally by daspk, the default,
- jactype = "fullusr": a full Jacobian, specified by user function jacfunc or jacres,
- jactype = "bandusr": a banded Jacobian, specified by user function jacfunc or jacres; the size of the bands specified by bandup and banddown,
- jactype = "bandint": a banded Jacobian, calculated by daspk; the size of the bands specified by bandup and banddown.
If `jactype = "fullusr"` or `"bandusr"` then the user must supply a subroutine `jacfunc`.

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

The input parameters `rtol`, and `atol` determine the error control performed by the solver. If the request for precision exceeds the capabilities of the machine, daspk will return an error code. See `lsoda` for details.

`res` and `jacres` may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details. Examples in FORTRAN are in the `dynload` subdirectory of the deSolve package directory.

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiledCode"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the `dynload` subdirectory of the deSolve package directory.

Value

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func` or `res`, plus an additional column (the first) for the time value. There will be one row for each element in `times` unless the FORTRAN routine ‘daspk’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

Note

In this version, the Krylov method is not (yet) supported.

Author(s)

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

References


Netlib: http://www.netlib.org

See Also

- `radau` for integrating DAEs up to index 3,
- `rk`,
- `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda, lsode, lsodes, lsodar, vode`, for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,

`diagnostics` to print diagnostic messages.

Examples

```r
##Coupled chemical reactions including an equilibrium
##modeled as (1) an ODE and (2) as a DAE

## The model describes three chemical species A,B,D:
## subjected to equilibrium reaction D <-> A + B
## D is produced at a constant rate, prod
## B is consumed at 1s-t order rate, r
## Chemical problem formulation 1: ODE

## Dissociation constant
K <- 1

## parameters
pars <- c(
  ka = 1e6, # forward rate
  r = 1,
  prod = 0.1)

Fun_ODE <- function (t, y, pars)
{
  with (as.list(c(y, pars)), {
    ra <- ka*D # forward rate
    rb <- ka/K *A*B # backward rate
    # rates of changes
  }
}
```
dD <- -ra + rb + prod
dA <- ra - rb
dB <- ra - rb - r*B
return(list(dy = c(dA, dB, dD),
            CONC = A+B+D))
}

## Chemical problem formulation 2: DAE
## 1. get rid of the fast reactions ra and rb by taking
## linear combinations : dD+dA = prod (res1) and
## dB-dA = -r*B (res2)
## 2. In addition, the equilibrium condition (eq) reads:
## as ra = rb : ka*D = ka/K*A*B = > K*D = A*B
## =======================================================================
Res_DAE <- function (t, y, yprime, pars)
{
  with (as.list(c(y, yprime, pars)), {
    ## residuals of lumped rates of changes
    res1 <- -dD - dA + prod
    res2 <- -dB + dA - r*B
    ## and the equilibrium equation
    eq <- K*D - A*B
    return(list(c(res1, res2, eq),
                  CONC = A+B+D))
  })
}

## Chemical problem formulation 3: Mass * Func
## Based on the DAE formulation
## =======================================================================
Mass_FUN <- function (t, y, pars)
{
  with (as.list(c(y, pars)), {
    ## as above, but without the
    f1 <- prod
    f2 <- - r*B
    ## and the equilibrium equation
    f3 <- K*D - A*B
    return(list(c(f1, f2, f3),
                  CONC = A+B+D))
  })
}


\[
\begin{align*}
\text{Mass} & \leftarrow \text{matrix}(\text{nr}=3, \text{nc}=3, \text{byrow}=\text{TRUE}, \\
\text{data}=c(1, 0, 1, \quad \# dA + 0 + dB \\
-1, 1, 0, \quad \quad \quad \# -dA + dB +0 \\
0, 0, 0)) \quad \# \text{algebraic}
\end{align*}
\]

\[\text{times} \leftarrow \text{seq}(0, 100, \text{by} = 2)\]

\[
\begin{align*}
\text{## Initial conc; D is in equilibrium with A,B} \\
\text{y} & \leftarrow \text{c}(A = 2, B = 3, D = 2\times 3/K) \\
\text{## ODE model solved with daspk} \\
\text{ODE} & \leftarrow \text{as.data.frame(daspk(y = y, times = times, func = Fun_ODE,} \\
& \quad \text{parms} = \text{pars, atol = 1e-10, rtol = 1e-10)}) \\
\text{## Initial rate of change} \\
\text{dy} & \leftarrow \text{c}(\text{dA} = 0, dB = 0, \text{dD} = 0) \\
\text{## DAE model solved with daspk} \\
\text{DAE} & \leftarrow \text{as.data.frame(daspk(y = y, dy = dy, times = times,} \\
& \quad \text{res = Res_DAE, parms = pars, atol = 1e-10, rtol = 1e-10}) \\
\text{MASS} & \leftarrow \text{daspk(y = y, times = times, func = Mass\_FUN, parms = pars, mass = Mass)}
\end{align*}
\]

\[
\begin{align*}
\text{## plotting output} \\
\text{## ================} \\
\text{opa} & \leftarrow \text{par(mfrow = c(2,2))} \\
\text{for (i in 2:5)} \\
\{ \\
\text{plot(ODE\_time,ODE[,i],xlab = "time",} \\
\text{ylab = "conc",main = names(ODE)[i],type = "l")} \\
\text{points(DAE\_time,DAE[,i],col = "red")} \\
\text{legend("bottomright",lty = c(1,NA),pch = c(NA,1),} \\
\text{col = c("black","red"),legend = c("ODE","DAE"))}
\}
\text{# difference between both implementations:} \\
\text{max(abs(ODE\_DAE))}
\text{par(mfrow = opa)}
\end{align*}
\]

\[
\begin{align*}
\text{## ================} \\
\text{## same DAE model, now with the Jacobian} \\
\text{## ================} \\
\text{jacres\_DAE} & \leftarrow \text{function (t, y, yprime, pars, cj)} \\
\{ \\
\text{with (as.list(c(y, yprime, pars)),} \\
\text{\# res1 = -dD - dA + prod} \\
\text{PD[1,1]} & \leftarrow -1\times cj \quad \# d(res1)/d(A)-cj\times d(res1)/d(dA) \\
\text{PD[1,2]} & \leftarrow 0 \quad \quad \quad \quad \# d(res1)/d(B)\times cj\times d(res1)/d(dB) \\
\text{PD[1,3]} & \leftarrow -1\times cj \quad \# d(res1)/d(D)-cj\times d(res1)/d(dD) \\
\text{\# res2 = -dB + dA - r*B}
\}
\end{align*}
\]
```r
PD[2,1] <- 1*cj
PD[2,2] <- -r -1*cj
PD[2,3] <- 0
## eq = K*D - A*B
PD[3,1] <- -B
PD[3,2] <- -A
PD[3,3] <- K
return(PD)
)
}
PD <- matrix(nc = 3, nr = 3, 0)
DAE2 <- as.data.frame(daspk(y = y, dy = dy, times = times,
    res = Res_DAE, jacres = jacres_DAE, jactype = "fullusr",
    parms = pars, atol = le-10, rtol = le-10))
max(abs(DAE-DAE2))
## See \dynload subdirectory for a FORTRAN implementation of this model
##==================================================================
## The chemical model as a DLL, with production a forcing function
##==================================================================
times <- seq(0, 100, by = 2)
pars <- c(K = 1, ka = le6, r = 1)
## Initial conc; D is in equilibrium with A,B
y <- c(A = 2, B = 3, D = 2*3/pars["K"])
## Initial rate of change
dy <- c(dA = 0, dB = 0, dD = 0)
# production increases with time
prod <- matrix(nc=2,data=c(seq(0,100,by=10),0.1*(1+runif(11)*1)))
ODE_dll <- as.data.frame(daspk(y=y,dy=dy,times=times,res="chemres",
    dllname="deSolve", initfunc="initparms",
    initforc="initforces", parms=pars, forcings=prod,
    atol=le-10,rtol=le-10,nout=2, outnames=c("CONC","Prod")))
op <- par(mfrow = c(1,2))
plot(ODE_dll$time,ODE_dll$Prod,xlab = "time",
ylab = "/day",main = "production rate",type = "l")
plot(ODE_dll$time,ODE_dll$D,xlab = "time",
ylab = "conc",main = "D",type = "l")
par(mfrow = opa)
```

**General Solver for Delay Differential Equations.**
Description

Function **dde** is a general solver for delay differential equations, i.e. equations where the derivative depends on past values of the state variables or their derivatives.

Usage

```
dede(y, times, func=NULL, parms, method = c( "lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "bdf", "adams", "impAdams"), control = NULL, ...)
```

Arguments

- **y**
  - the initial (state) values for the DE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
- **times**
  - time sequence for which output is wanted; the first value of `times` must be the initial time.
- **func**
  - an R-function that computes the values of the derivatives in the ODE system (the model definition) at time `t`.
    - `func` must be defined as: `func <- function(t, y, parms, ...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the DE 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`. The derivatives should be specified in the same order as the state variables `y`.
    - If method "daspk" is used, then `func` can be `NULL`, in which case `res` should be used.
- **parms**
  - parameters passed to `func`.
- **method**
  - the integrator to use, either a string ("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "bdf", "adams", "impAdams") or a function that performs integration. The default integrator used is `lsoda`.
- **control**
  - a list that can supply (1) the size of the history array, as `control$mxhist`; the default is 1e4 and (2) how to interpolate, as `control$interpol`, where 1 is hermitian interpolation, 2 is variable order interpolation, using the Nordsieck history array. Only for the two Adams methods is the second option recommended.
  - ... additional arguments passed to the integrator.

Details

Functions `lagvalue` and `lagderiv` are to be used with `dde` as they provide access to past (lagged) values of state variables and derivatives. The number of past values that are to be stored in a history matrix, can be specified in `control$mxhist`. The default value (if unspecified) is 1e4.

Cubic Hermite interpolation is used to obtain an accurate interpolant at the requested lagged time.
**Value**

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the second element of the return from `func`, plus an additional column (the first) for the time value. There will be one row for each element in `times` unless the integrator returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

**Author(s)**

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

**See Also**

`lagvalue`, `lagderiv`, for how to specify lagged variables and derivatives.

**Examples**

```r
## A simple delay differential equation
## dy(t) = -y(t-1) ; y(t<0)=1
## _________________________________________________________________

##----------------------------
## the derivative function
##----------------------------
derivs <- function(t, y, parms) {
  if (t < 1)
    dy <- -1
  else
    dy <- - lagvalue(t - 1)
  list(c(dy))
}

##----------------------------
## initial values and times
##----------------------------
yinit <- 1
times <- seq(0, 30, 0.1)

##----------------------------
## solve the model
```
## The infectious disease model of Hairer; two lags.
## example 4 from Shampine and Thompson, 2000
## solving delay differential equations with dde23
#
## the derivative function
##
derivs <- function(t,y,parms) {
  if (t < 1)
    lag1 <- 0.1
  else
    lag1 <- lagvalue(t - 1,2)
  if (t < 10)
    lag10 <- 0.1
  else
    lag10 <- lagvalue(t - 10,2)
  dy1 <- -y[1] * lag1 + lag10
  dy3 <- y[2] - lag10
  list(c(dy1, dy2, dy3))
}

## initial values and times
##
yinit <- c(5, 0.1, 1)
times <- seq(0, 40, by = 0.1)

## solve the model
##
system.time(
  yout <- dede(y = yinit, times = times, func = derivs, parms = NULL)
)

## display, plot results
##
matplot(yout[,1], yout[,-1], type = "l", lwd = 2, lty = 1, 
  main = "Infectuous disease - Hairer")
## time lags + EVENTS triggered by a root function
## The two-wheeled suitcase model
## example 8 from Shampine and Thompson, 2000
## solving delay differential equations with dde23
## =============================================================================

### the derivative function

```r
derivs <- function(t, y, parms) {
  if (t < tau)
    lag <- 0
  else
    lag <- lagvalue(t - tau)

  dy1 <- y[2]
  dy2 <- -sign(y[1]) * gam * cos(y[1]) +
            sin(y[1]) - bet * lag[1] + A * sin(omega * t + mu)
  list(c(dy1, dy2))
}
```

### root and event function

```r
root <- function(t,y,parms) ifelse(t>0, return(y), return(1))
event <- function(t,y,parms) return(c(y[1], y[2]*0.931))
```

gam = 0.248; bet = 1; tau = 0.1; A = 0.75
omega = 1.37; mu = asin(gam/A)

### initial values and times

```r
yinit <- c(y = 0, dy = 0)
times <- seq(0, 12, len = 1000)
```

### solve the model

```r
yout <- dede(y = yinit, times = times, func = derivs, parms = NULL,
              method = "lsodar", rootfun = root, events = list(func = event, root = TRUE))
```

### display, plot results

```r
plot(yout, which = 1, type = "l", lwd = 2, main = "suitcase model", mfrow = c(1,2))
plot(yout[,2], yout[,3], xlab = "y", ylab = "dy", type = "l", lwd = 2)
```


---

**diagnostics**

*Print Diagnostic Characteristics of Solvers*

---

**Description**

Prints several diagnostics of the simulation to the screen, e.g. number of steps taken, the last step size, ...

**Usage**

```r
diagnostics(obj, ...)
## Default S3 method:
diagnostics(obj, ...)
```

**Arguments**

- `obj` is an output data structure produced by one of the solver routines.
- `...` optional arguments allowing to extend `diagnostics` as a generic function.

**Details**

Detailed information about the success of a simulation is printed, if a `diagnostics` function exists for a specific solver routine. A warning is printed, if no class-specific diagnostics exists.

Please consult the class-specific help page for details.

**See Also**

- `diagnostics.deSolve` for diagnostics of differential equation solvers.

---

**diagnostics.deSolve**

*Print Diagnostic Characteristics of ODE and DAE Solvers*

---

**Description**

Prints several diagnostics of the simulation to the screen, e.g. number of steps taken, the last step size, ...

**Usage**

```r
## S3 method for class 'deSolve':
diagnostics(obj, Full = FALSE, ...)
```
Arguments

obj is the output matrix as produced by one of the integration routines.

Full when TRUE then all messages will be printed, including the ones that are not relevant for the solver. If FALSE, then only the relevant messages will be printed.

... optional arguments allowing to extend diagnostics as a generic function.

Details

When the integration output is saved as a data.frame, then the required attributes are lost and method diagnostics will not work anymore.

Value

The integer and real vector with diagnostic values; for function lsodar also the root information.

See tables 2 and 3 in vignette("deSolve") for what these vectors contain.

Note: the number of function evaluations are *without* the extra calls performed to generate the ordinary output variables (if present).

Examples

```r
# The famous Lorenz equations: chaos in the earth's atmosphere

chaos <- function(t, state, parameters) {
  with(as.list(c(state)), {
    dx <- -8/3 * x + y * z
    dy <- -10 * (y - z)
    dz <- -x * y + 28 * y - z
    list(c(dx, dy, dz))
  })
}

state <- c(x = 1, y = 1, z = 1)
times <- seq(0, 50, 0.01)
out <- vode(state, times, chaos, 0)
pairs(out, pch = ".")
diagnostics(out)
```

---

**DLLfunc Evaluates a Derivative Function Represented in a DLL**

**Description**

Calls a function, defined in a compiled language as a DLL.
Usage

DLLfunc(func, times, y, parms, dllname, 
initfunc = dllname, rpar = NULL, ipar = NULL, nout = 0, 
outnames = NULL, forcings = NULL, initforc = NULL, 
fcontrol = NULL)

Arguments

func  the name of the function in the dynamically loaded shared library,
times  first value = the time at which the function needs to be evaluated,
y  the values of the dependent variables for which the function needs to be evaluated,
parms  the parameters that are passed to the initialiser function,
dllname  a string giving the name of the shared library (without extension) that contains
  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.
rpar  a vector with double precision values passed to the DLL-function func and
  jacfunc present in the DLL, via argument rpar,
ipar  a vector with integer values passed to the dll-function func and jacfunc
  present in the DLL, via function argument ipar,
nout  the number of output variables.
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.
forcings  only used if ‘dllname’ is specified: a list with the forcing function data sets, each
  present as a two-columned matrix, with (time, value); interpolation outside the
  interval [min(times), max(times)] is done by taking the value at the closest
  data extreme.
  See package vignette "compiledCode".
initforc  if not NULL, the name of the forcing function initialisation function, as provided
  in ‘dllname’. It MUST be present if forcings has been given a value. See
  package vignette "compiledCode".
fcontrol  A list of control parameters for the forcing functions. See package vignette
  "compiledCode".

Details

This function is meant to help developing FORTRAN or C models that are to be used to solve
ordinary differential equations (ODE) in packages deSolve and/or rootSolve.

Value

a list containing:
dy  the rate of change estimated by the function,
var  the ordinary output variables of the function.
Author(s)
Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also
ode for a general interface to most of the ODE solvers

Examples

```
# ex. 1
# ccl4model
# Parameter values and initial conditions
# see example(ccl4model) for a more comprehensive implementation
Parms <- c(0.182, 4.0, 4.0, 0.08, 0.4, 0.74, 0.05, 0.15, 0.32, 16.17, 281.48, 13.3, 16.17, 5.487, 153.8, 0.04321671, 0.4027255, 1000, 0.02, 1.0, 3.8)
yini <- c(AI = 21, AAM = 0, AT = 0, AF = 0, AL = 0, CLT = 0, AM = 0)

# the rate of change
DLLfunc(y = yini, dllname = "deSolve", func = "derivscc14", initfunc = "initccl4", parms = Parms, times = 1, nout = 3, outnames = c("DOSE", "MASS", "CP") )
```

```
# ex. 2
# SCOC model, in fortran - to see the FORTRAN code:
# Forcing function "data"
Flux <- matrix(ncol = 2, byrow = TRUE, data = c(1, 0.654, 2, 0.167))
parms <- c(k = 0.01)
Yini <- 60

# Forcing function "data"
DLLfunc(y=Yini, times=1, func = "scocder", parms = parms, dllname = "deSolve", initforc = "scocforc", forcings = Flux, initfunc = "scocpar", nout = 2, outnames = c("Mineralisation","Depo"))

# correct value = dy = flux - k * y = 0.654 - 0.01 * 60
DLLfunc(y = Yini, times = 2, func = "scocder", parms = parms, dllname = "deSolve", initforc = "scocforc", forcings = Flux, initfunc = "scocpar", nout = 2, outnames = c("Mineralisation", "Depo"))
```
DLLres  

Evaluates a Residual Derivative Function Represented in a DLL

Description

Calls a residual function, \( F(t, y, y') \) of a DAE system (differential algebraic equations) defined in a compiled language as a DLL.

To be used for testing the implementation of DAE problems in compiled code

Usage

```
DLLres(res, times, y, dy, parms, dllname, 
    initfunc = dllname, rpar = NULL, ipar = NULL, nout = 0, 
    outnames = NULL, forcings = NULL, initforc = NULL, 
    fcontrol = NULL)
```

Arguments

- `res`: the name of the function in the dynamically loaded shared library,
- `times`: first value = the time at which the function needs to be evaluated,
- `y`: the values of the dependent variables for which the function needs to be evaluated,
- `dy`: the derivative of the values of the dependent variables for which the function needs to be evaluated,
- `parms`: the parameters that are passed to the initialiser function,
- `dllname`: a string giving the name of the shared library (without extension) that contains 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,
- `rpar`: a vector with double precision values passed to the DLL-function `func` and `jacfunc` present in the DLL, via argument `rpar`,
- `ipar`: a vector with integer values passed to the DLL-function `func` and `jacfunc` present in the DLL, via function argument `ipar`,
- `nout`: the number of output variables,
- `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,
- `forcings`: only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time, value); interpolation outside the interval \([\min(times), \max(times)]\) is done by taking the value at the closest data extreme. See package vignette "compiledCode".
- `initforc`: if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if `forcings` has been given a value. See package vignette "compiledCode".
fcontrol  A list of control parameters for the forcing functions. See package vignette "compiledCode".

Details

This function is meant to help developing FORTRAN or C models that are to be used to solve differential algebraic equations (DAE) in package deSolve.

Value

a list containing:

res  the residual of derivative estimated by the function
var  the ordinary output variables of the function

Author(s)

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

See Also

daspk to solve DAE problems

Examples

```r
## =========================================================================
## Residuals from the daspk chemical model, production a forcing function
## =========================================================================
## Parameter values and initial conditions
## see example(daspk) for a more comprehensive implementation

pars <- c(K = 1, ka = 1e6, r = 1)

## Initial conc; D is in equilibrium with A,B
y <- c(A = 2, B = 3, D = 2 * 3/pars["K"])

## Initial rate of change
dy <- c(dA = 0, dB = 0, dD = 0)

## production increases with time
prod <- matrix(ncol = 2,
               data = c(seq(0, 100, by = 10), seq(0.1, 0.5, len = 11)))

DLLres(y = y, dy = dy, times = 5, res = "chemres",
       dllname = "deSolve", initfunc = "initparms",
       initforc = "initforcs", pars = pars, forcings = prod,
       nout = 2, outnames = c("CONC", "Prod"))
```
Implementing Events in Differential Equation Models.

Description

An event occurs when the value of a state variable is suddenly changed, e.g. because a value is added, subtracted, or multiplied. The integration routines cannot deal easily with such state variable changes. Typically these events occur only at specific times. In deSolve, events can be imposed by means of an input data.frame, that specifies at which time a certain state variable is altered, or via an event function.

Details

The events are specified by means of argument events passed to the integration routines. events should be a list that contains one of the following:

1. func: an R-function or the name of a function in compiled code that specifies the event,
2. data: a data.frame that specifies the variables, times, values and types of the events,
3. time: when events are specified by a function: the times at which the events take place,
4. root: when events are specified by a function and triggered by a root, this logical should be set equal to TRUE,
5. maxroot: when root = TRUE, the maximal number of times at which a root is found that are kept; defaults to 100. If the number of roots > maxroot, then only the first maxroot will be outputted.
6. ties: if events, as specified by a data.frame are "ordered", set to "ordered", the default is "notordered". This will save some computational time.

If specified by an R-function (argument events$func), this requires either input of the time of the events, a vector in events$time OR the specification of a root function. In the latter case, the model MUST be solved with integration routine lsodar

The R-function, must be defined as: function(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 events$func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The function should return the modified y-values, as a vector.

If events$func is a string, this indicates that the events are specified by a function in compiled code. This function has as arguments, the number of state variables, the time, and the state variable vector. See package vignette "compiledCode" for more details.

In addition, either the time at which the events take place should be specified as a vector (events$time). When the model is solved with lsodar, or with lsode, an event can also be triggered by a root function, as specified with argument rootfunc. In this case, the integrator is informed that the simulation it to be continued after a root is found by setting events$root equal to TRUE.

If in this case roots are found, then the output will have attribute troot which will contain the times at which a root was found. There will be at most events$maxroot such values. The default is 100.
See two last examples; also see example of `cc14model`.

If specified by a `data.frame` (argument `events$data`), this should contain the following columns (and in that order):

1. `var` the state variable name or number that is affected by the event
2. `time` the time at which the event is to take place; the solvers will check if the time is embraced by the simulation time
3. `value` the value, magnitude of the event
4. `method` which event is to take place; should be one of ("replace", "add", "multiply"); also allowed is to specify the number (1=replace, 2=add, 3=multiply)

For instance, the following line

"v1" 10 2 "add"

will cause the value 2 to be added to a state variable, called "v1" at time = 10.

### Author(s)

Karline Soetaert,

### See Also

`forcings`, for how to implement forcing functions.

### Examples

```r
## EVENTS in a data.frame

## model: rate of change set to 0
eventmod <- function(t, var, parms) {
  list(dvar = rep(0, 2))
}
yini <- c(v1 = 1, v2 = 2)
times <- seq(0, 10, by = 0.1)

eventdat <- data.frame(var = c("v1", "v2", "v2", "v1"),
                         time = c(1, 1, 5, 9),
                         value = c(1, 2, 3, 4),
                         method = c("add", "mult", "rep", "add"))

out <- vode(func = eventmod, y = yini, times = times, parms = NULL,
             events = list(data = eventdat))
plot(out, type = "l")

eventdat <- data.frame(var = c(rep("v1", 10), rep("v2", 10)),
                         time = c(1:10, 1:10),
                         value = runif(20),
                         method = c("add", "mult", "rep", "add"))
```
method = rep("add", 20))

out <- ode(func = eventmod, y = yini, times = times, parms = NULL,
   events = list(data = eventdat))

plot(out, type = "l")

## EVENTS in a function

## model: rate of change v1 = 0, v2 consumed at first-order rate

## derivative: simple first-order decay

## events: add 1 to v1, multiply v2 with 0.8

## EVENTS triggered by a root function

yini <- c(v1 = 1, v2 = 2)
times <- seq(0, 10, by = 0.1)

out <- ode(func = eventmod, y = yini, times = times, parms = NULL,
   events = list(func = eventfun, time = 1:9) )
plot(out, type="l")
 forcings

```r
yini <- 2
times <- seq(0, 100, 0.1)

## uses lsodar to solve; root = TRUE specifies that the event is
## triggered by a root.
out <- lsodar(times = times, y = yini, func = func, parms = NULL,
  events = list(func = eventfun, root = TRUE),
  rootfun = rootfun)

plot(out, type = "l")

## time of the root:
troot <- attributes(out)$troot
points(troot, rep(0.5, length(troot)))
```

forcings  

Passing Forcing Functions to Models Written in R or Compiled Code.

Description

A forcing function is an external variable that is essential to the model, but not explicitly modeled. Rather, it is imposed as a time-series. Thus, if a model uses forcing variables, their value at each time point needs to be estimated by interpolation of a data series.

Details

The forcing functions are imposed as a data series, that contains the values of the forcings at specified times.

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

If the model is defined in R code, it is most efficient to:

1. define a function that performs the linear interpolation, using R’s approxfun. It is generally recommended to use rule = 2, such as to allow extrapolation outside of the time interval, especially when using the Livermore solvers, as these may exceed the last time point.
2. call this function within the model’s derivative function, to interpolate at the current timestep.

See first example.

If the models are defined in compiled C or FORTRAN code, it is possible to use deSolve forcing function update algorithm. This is the compiled-code equivalent of approxfun or approx.

In this case:

1. the forcing function data series is provided by means of argument forcings,
2. initforc is the name of the forcing function initialisation function, as provided in ‘dllname’, while
3. fcontrol is a list used to finetune how the forcing update should be performed.

The fcontrol argument is a list that can supply any of the following components (conform the definitions in the approxfun function):
**method** specifies the interpolation method to be used. Choices are "linear" or "constant".  

**rule** an integer describing how interpolation is to take place outside the interval \([\text{min(times)}, \text{max(times)}]\). If rule is 1 then an error will be triggered and the calculation will stop if times extends the interval of the forcing function data set. If it is 2, the default, the value at the closest data extreme is used, a warning will be printed if verbose is TRUE.  

Note that the default differs from the approx default.

**f** For method = "constant" a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If \(y0\) and \(y1\) are the values to the left and right of the point then the value is \(y0 \times (1 - f) + y1 \times f\) so that \(f = 0\) is right-continuous and \(f = 1\) is left-continuous.

**ties** Handling of tied times values. Either a function with a single vector argument returning a single number result or the string "ordered".  

Note that the default is "ordered", hence the existence of ties will NOT be investigated; in the C code this will mean that -if ties exist, the first value will be used; if the dataset is not ordered, then nonsense will be produced.  

Alternative values for ties are mean, min etc

The defaults are:

```r
fcontrol = list(method = "linear", rule = 2, f = 0, ties = "ordered")
```

Note that only ONE specification is allowed, even if there is more than one forcing function data set.

More information about models defined in compiled code is in the package vignette ("compiled-Code").

**Note**

How to write compiled code is described in package vignette "compiledCode", which should be referred to for details.

This vignette also contains examples on how to pass forcing functions.

**Author(s)**

Karline Soetaert,
Thomas Petzoldt,
R. Woodrow Setzer (Maintainer)

**See Also**

approx or approxfun, the R function,
events for how to implement events.
## FORCING FUNCTION: The sediment oxygen consumption example - R-code:

### Forcing function data

```r
Flux <- matrix(ncol=2, byrow=TRUE, data=c(1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186, 93, 0.140, 103, 0.255, 113, 1.091, 163, 1.001, 173, 1.691, 183, 1.404, 194, 1.226, 204, 0.767, 214, 0.893, 224, 0.737, 234, 0.772, 244, 0.726, 254, 0.624, 264, 0.439, 274, 0.168, 284, 0.280, 294, 0.202, 304, 0.193, 315, 0.286, 325, 0.599, 335, 1.889, 345, 0.996, 355, 0.681, 365, 1.135))
``` 

```r
parms <- c(k=0.01)
times <- 1:365
```

### the model

```r
sediment <- function(t, O2, k)
  list(c(Depo(t) - k * O2), depo = Depo(t))
```

### the forcing functions; rule = 2 avoids NaNs in interpolation

```r
Depo <- approxfun(x = Flux[,1], y = Flux[,2], method = "linear", rule = 2)
```

```r
Out <- ode(times = times, func = sediment, y = c(O2 = 63), parms = parms)
```

```r
Out2 <- ode(times = times, func = sediment, y = c(O2 = 63), parms = parms)
```

### same forcing functions, now constant interpolation

```r
Depo <- approxfun(x = Flux[,1], y = Flux[,2], method = "constant", f = 0.5, rule = 2)
```

```r
Out2 <- ode(times = times, func = sediment, y = c(O2 = 63), parms = parms)
```

### Constant interpolation of forcing function - left side of interval

```r
fcontrol <- list(method = "constant")
```
lsoda <- SCOC(times, parms = parms, Flux = Flux, fcontrol = fcontrol)
plot(out2$time, out2$Depo, type = "l", col = "red")
lines(out2$time, out2$Mineralisation, col = "blue")

## Not run:
## ==============================================================
## show examples (see respective help pages for details)
## ==============================================================
example(aquaphy)

## show package vignette with tutorial about how to use compiled models
## + source code of the vignette
## + directory with C and FORTRAN sources
vignette("compiledCode")
edit(vignette("compiledCode"))
browseURL(paste(system.file(package = "deSolve"), "/doc", sep = ""))

## End(Not run)

lsoda

Solving initial value problems for stiff or non-stiff systems of first-order ordinary differential equations (ODEs).

The R function lsoda provides an interface to the FORTRAN ODE solver of the same name, written by Linda R. Petzold and Alan C. Hindmarsh.

The system of ODE’s is written as an R function (which may, of course, use .C, .Fortran, .Call, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded. A vector of parameters is passed to the ODEs, so the solver may be used as part of a modeling package for ODEs, or for parameter estimation using any appropriate modeling tool for non-linear models in R such as optim, nls, nlm or nlme.

lsoda differs from the other integrators (except lsodar) in that it switches automatically between stiff and nonstiff methods. This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

Usage

lsoda(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
      jacfunc = NULL, jactype = "fullint", verbose = FALSE,
      tcrit = NULL, hmin = 0, hmax = NULL, hini = 0, ynames = TRUE,
```r
code
maxordn = 12, maxords = 5, bandup = NULL, banddown = NULL,
maxsteps = 5000, dllname = NULL, initfunc = dllname,
initpar = parms, rpar = NULL, ipar = NULL, nout = 0,
outnames = NULL, forcings = NULL, initforc = NULL,
fcontrol = NULL, events = NULL, lags = NULL,...
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y</code></td>
<td>the initial (state) values for the ODE system. If <code>y</code> has a name attribute, the names will be used to label the output matrix.</td>
</tr>
<tr>
<td><code>times</code></td>
<td>times at which explicit estimates for <code>y</code> are desired. The first value in <code>times</code> must be the initial time.</td>
</tr>
</tbody>
</table>
| `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: `func <- function(t, y, parms,...)`.
| `parms`  | vector or list of parameters used in `func` or `jacfunc`. |
| `rtol`   | relative error tolerance, either a scalar or an array as long as `y`. See details. |
| `atol`   | absolute error tolerance, 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 \( \frac{\partial y_i}{\partial y_j} \), or a string giving the name of a function or subroutine in 'dllname' that computes the Jacobian (see vignette "compiledCode" 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`. |
| `jactype`| the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user. |

If the Jacobian is a full matrix, `jacfunc` should return a matrix \( \frac{\partial y_i}{\partial y_j} \), where the \( i \)th row contains the derivative of \( \frac{dy_i}{dt} \) 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.
verbose a logical value that, when TRUE, will print the diagnostics of the integration - see details.

tcrit if not NULL, then lsoda cannot integrate past tcrit. The FORTRAN routine lsoda 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 logical, if FALSE: names of state variables are not passed to function func; this may speed up the simulation especially for large models.

maxordn the maximum order to be allowed in case the method is non-stiff. Should be <= 12. Reduce maxord to save storage space.

maxords the maximum order to be allowed in case the method is stiff. Should be <= 5. 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 per output interval taken by the solver.

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 package vignette "compiledCode".

initfunc if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".

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 calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".

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. These names will be used to label the output matrix.
forcings  only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\min(times), \max(times)]\) is done by taking the value at the closest data extreme. See forcings or package vignette "compiledCode".

initforc  if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if forcings has been given a value. See forcings or vignette "compiledCode".

fcontrol  A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events  A list that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags  A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

...  additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

All the hard work is done by the FORTRAN subroutine lsoda, whose documentation should be consulted for details (it is included as comments in the source file `src/opkdmain.f`). The implementation is based on the 12 November 2003 version of lsoda, from Netlib.

lsoda switches automatically between stiff and nonstiff methods. This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

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

"fullint"  a full Jacobian, calculated internally by lsoda, the default,

"fullusr"  a full Jacobian, specified by user function jacfunc,

"bandusr"  a banded Jacobian, specified by user function jacfunc the size of the bands specified by bandup and banddown,

"bandint"  banded Jacobian, calculated by lsoda; the size of the bands specified by bandup and banddown.

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

The following description of error control is adapted from the documentation of the lsoda source code (input arguments rtol and atol, above):

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 \(y\), according to an inequality of the form max-norm of \((e/\text{ewt})\) \(\leq 1\), where \(\text{ewt}\) is a vector of positive error weights. The values of rtol and atol should all be non-negative. The form of \(\text{ewt}\) is:

\[
rtol \times \text{abs}(y) + atol
\]

where multiplication of two vectors is element-by-element.
If the request for precision exceeds the capabilities of the machine, the FORTRAN subroutine lsoda will return an error code; under some circumstances, the R function lsoda will attempt a reasonable reduction of precision in order to get an answer. It will write a warning if it does so.

The diagnostics of the integration can be printed to screen by calling diagnostics. If verbose = TRUE, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette("compiledCode"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the FORTRAN routine ‘lsoda’ returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

The ‘demo’ directory contains some examples of using gnls to estimate parameters in a dynamic model.

Author(s)

R. Woodrow Setzer <setzer.woodrow@epa.gov>

References


Netlib: http://www.netlib.org

See Also

- rk, rkMethod, rk4 and euler for Runge-Kutta integrators.
- lsode, which can also find a root
- lsodes, lsodar, vode, daspk for other solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
• ode.1D for integrating 1-D models,
• ode.2D for integrating 2-D models,
• ode.3D for integrating 3-D models,
diagnostics to print diagnostic messages.

Examples

```r
SPCmod <- function(t, x, parms) {
  with(as.list(c(parms, x)), {
    import <- sigimp(t)
    dS <- import - b*S*P + g*C  #substrate
    dP <- c*S*P - d*C*P         #producer
    dC <- e*P*C - f*C          #consumer
    res <- c(dS, dP, dC)
    list(res)
  })
}
```

```r
# Parameters
parms <- c(b = 0.0, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)
```

```r
# vector of timesteps
times <- seq(0, 100, length = 101)
```

```r
# external signal with rectangle impulse
signal <- as.data.frame(list(times = times,
                               import = rep(0,length(times))))
```

```r
signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2
```

```r
sigimp <- approxfun(signal$times, signal$import, rule = 2)
```

```r
# Start values for steady state
y <- xstart <- c(S = 1, P = 1, C = 1)
```

```r
# Solving
out <- as.data.frame(lsoda(xstart, times, SPCmod, parms))
```

```r
# Plotting
mf <- par(mfrow = c(2,2))
```
### Example 2:
```r
## from lsoda source code
## names makes this easier to read, but may slow down execution.

## measure speed (here and below)

## Isoda
lsodar

Solver for Ordinary Differential Equations (ODE), Switching Automatically Between Stiff and Non-stiff Methods and With Root Finding

Description

Solving initial value problems for stiff or non-stiff systems of first-order ordinary differential equations (ODEs) and including root-finding.

The R function lsodar provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Linda R. Petzold.

The system of ODE’s is written as an R function or be defined in compiled code that has been dynamically loaded. - see description of lsoda for details.

lsodar differs from lsode in two respects.

• It switches automatically between stiff and nonstiff methods (similar as lsoda).
• It finds the root of at least one of a set of constraint functions g(i) of the independent and dependent variables.

Two uses of lsodar are:

• To stop the simulation
• To trigger an events, i.e. a sudden change in one of the state variables.

when a particular condition is met.
Usage

\texttt{lsodar(y, times, func, parms, rtol = 1e-6, atol = 1e-6,}
\texttt{jacfunc = NULL, jactype = "fullint", rootfunc = NULL,}
\texttt{verbose = FALSE, nroot = 0, tcrit = NULL, hmin = 0,}
\texttt{hmax = NULL, hini = 0, ynames = TRUE, maxordn = 12,}
\texttt{maxords = 5, bandup = NULL, banddown = NULL, maxsteps = 5000,}
\texttt{dllname = NULL, initfunc = dllname, initpar = parms,}
\texttt{rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,}
\texttt{initforc = NULL, fcontrol=NULL, events=NULL, lags = NULL, ...)

Arguments

\texttt{y}
the initial (state) values for the ODE system. If \texttt{y} has a name attribute, the names will be used to label the output matrix.

\texttt{times}
times at which explicit estimates for \texttt{y} are desired. The first value in \texttt{times} must be the initial time.

\texttt{func}
either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the \textit{model definition}) at time \texttt{t}, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If \texttt{func} is an \texttt{R}-function, it must be defined as: \texttt{func <- function(t, y, parms,...).} \texttt{t} is the current time point in the integration, \texttt{y} is the current estimate of the variables in the ODE system. If the initial values \texttt{y} has a \texttt{names} attribute, the names will be available inside \texttt{func}. \texttt{parms} is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of \texttt{func} should be a list, whose first element is a vector containing the derivatives of \texttt{y} with respect to \texttt{time}, and whose next elements are global values that are required at each point in \texttt{times}. The derivatives should be specified in the same order as the state variables \texttt{y}.

If \texttt{func} is a string, then \texttt{dllname} must give the name of the shared library (without extension) which must be loaded before \texttt{lsodar()} is called. See package vignette "compiledCode" for more details.

\texttt{parms}
vector or list of parameters used in \texttt{func} or \texttt{jacfunc}.

\texttt{rtol}
relative error tolerance, either a scalar or an array as long as \texttt{y}. See details.

\texttt{atol}
absolute error tolerance, either a scalar or an array as long as \texttt{y}. See details.

\texttt{jacfunc}
if not \texttt{NULL}, an \texttt{R} function, that computes the Jacobian of the system of differential equations \(\partial y_i / \partial y_j\), or a string giving the name of a function or subroutine in \texttt{dllname} that computes the Jacobian (see vignette "compiledCode" for more about this option).

In some circumstances, supplying \texttt{jacfunc} can speed up the computations, if the system is stiff. The \texttt{R} calling sequence for \texttt{jacfunc} is identical to that of \texttt{func}.

If the Jacobian is a full matrix, \texttt{jacfunc} should return a matrix \(\partial y / \partial y\), where the ith row contains the derivative of \(d y_i / d t\) with respect to \(y_j\), or a vector containing the matrix elements by columns (the way \texttt{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.

`rootfunc` if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in ‘dllname’ that computes the root function. The R calling sequence for `rootfunc` is identical to that of `func`. `rootfunc` should return a vector with the function values whose root is sought.

`verbose` a logical value that, when TRUE, will print the diagnostic of the integration - see details.

`nroot` only used if ‘dllname’ is specified: the number of constraint functions whose roots are desired during the integration; if `rootfunc` is an R-function, the solver estimates the number of roots.

`tcrit` if not NULL, then lsodar cannot integrate past `tcrit`. The FORTRAN routine lsodar 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` logical, if FALSE: names of state variables are not passed to function `func`; this may speed up the simulation especially for large models.

`maxordn` the maximum order to be allowed in case the method is non-stiff. Should be <= 12. Reduce `maxord` to save storage space.

`maxords` the maximum order to be allowed in case the method is stiff. Should be <= 5. 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 per output interval taken by the solver.

`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 package vignette "compiledCode".

`initfunc` if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

`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 calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

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. These names will be used to label the output matrix.

forcings only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\text{min(times)}, \text{max(times)}]\) is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if forcings has been given a value. See forcings or vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette "compiledCode".

events A list that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

... additional arguments passed to `func` and `jacfunc` allowing this to be a generic function.

Details

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

lsodar switches automatically between stiff and nonstiff methods (similar as lsoda). This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

It finds the root of at least one of a set of constraint functions \(g(i)\) of the independent and dependent variables. It then returns the solution at the root if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.

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

`jactype = "fullint"`: a full Jacobian, calculated internally by Isodor, the default,

`jactype = "fullusr"`: a full Jacobian, specified by user function `jacfunc`,
**Isodar**

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

**jactype = "bandint":** banded Jacobian, calculated by Isodar; 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`, and `atol` determine the **error control** performed by the solver. See `lsoda` for details.

The output will have the attribute `iroot`, if a root was found `iroot` is a vector, its length equal to the number of constraint functions it will have a value of 1 for the constraint function whose root that has been found and 0 otherwise.

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiledCode"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.

**Value**

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine ‘lsoda’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

If a root has been found, the output will have the attribute `iroot`, an integer indicating which root has been found.

**Author(s)**

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

**References**


Netlib: [http://www.netlib.org](http://www.netlib.org)
See Also

- `rk`, `rkMethod`, `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda`, `lsode`, `lsodes`, `vode`, `daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.

Examples

```r
## Example 1: from lsodar source code

Fun <- function (t, y, parms) {
  ydot <- vector(len = 3)
  return(list(ydot, ytot = sum(y)))
}

rootFun <- function (t, y, parms) {
  yroot <- vector(len = 2)
  return(yroot)
}

y <- c(1, 0, 0)
times <- c(0, 0.4*10^(0:8))
Out <- NULL
ny <- length(y)
out <- lsodar(y = y, times = times, fun = Fun, rootfun = rootFun,
              rtol = 1e-4, atol = c(1e-6, 1e-10, 1e-6), parms = NULL)
print(paste("root is found for eqn", which(attributes(out)$iroot == 1)))
print(out[nrow(out),])
diagnostics(out)
```

```r
## Example 2: using lsodar to estimate steady-state conditions

y <- c(1, 0, 0)
times <- c(0, 0.4+10^(0:8))
Out <- NULL
ny <- length(y)
out <- lsodar(y = y, times = times, fun = Fun, rootfun = rootFun,
              rtol = 1e-4, atol = c(1e-6, 1e-10, 1e-6), parms = NULL)
print(paste("root is found for eqn", which(attributes(out)$iroot == 1)))
print(out[nrow(out),])
diagnostics(out)
```
## Bacteria (Bac) are growing on a substrate (Sub)

```r
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)))
  })
}
```

## root is the condition where sum of |rates of change|
## is very small

```r
rootfun <- function(t, state, pars) {
  dstate <- unlist(model(t, state, pars)) # rate of change vector
  return(sum(abs(dstate)) - 1e-10)
}
```

```r
pars <- list(Bini = 0.1, Sini = 100, gmax = 0.5, eff = 0.5,
             ks = 0.5, rB = 0.01, dB = 0.01, input = 0.1)
```

```r
tout <- c(0, 1e10)
state <- c(Bact = pars$Bini, Sub = pars$Sini)
out <- lsodar(state, tout, model, pars, rootfun = rootfun)
print(out)
```

## Example 3:
## using lsodar to trigger an event

```r
derivfun <- function(t, y, parms)
  list(-0.05 * y)
```

```r
rootfun <- function(t, y, parms)
  return(y - 0.1)
```

```r
eventfun <- function(t, y, parms)
  return(y + runif(1))
```

```r
yini <- 0.8
times <- 0:200
```

```r
out <- lsodar(func = derivfun, y = yini, times = times,
               rootfunc = rootfun, events = list(func = eventfun, root = TRUE))
```

```r
plot(out, type = "l", lwd = 2, main = "lsodar with event")
```
lsode

Solver for Ordinary Differential Equations (ODE)

Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]

The \texttt{R} function \texttt{lsode} provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Andrew H. Sherman.

It combines parts of the code \texttt{lsodar} and can thus find the root of at least one of a set of constraint functions \( g(i) \) of the independent and dependent variables. This can be used to stop the simulation or to trigger \texttt{events}, i.e. a sudden change in one of the state variables.

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

In contrast to \texttt{lsoda}, the user has to specify whether or not the problem is stiff and choose the appropriate solution method.

\texttt{lsode} is very similar to \texttt{vode}, but uses a fixed-step-interpolate method rather than the variable-coefficient method in \texttt{vode}. In addition, in \texttt{vode} it is possible to choose whether or not a copy of the Jacobian is saved for reuse in the corrector iteration algorithm; In \texttt{lsode}, a copy is not kept.

Usage

\[
\text{lsode}(y, \text{times}, \text{func}, \text{parms}, \text{rtol} = 1e-6, \text{atol} = 1e-6, \\
\text{jacfunc} = \text{NULL}, \text{jactype} = "fullint", \text{mf} = \text{NULL}, \text{rootfunc} = \text{NULL}, \\
\text{verbose} = \text{FALSE}, \text{nroot} = 0, \text{tcrit} = \text{NULL}, \text{hmin} = 0, \text{hmax} = \text{NULL}, \\
\text{hini} = 0, \text{ynames} = \text{TRUE}, \text{maxord} = \text{NULL}, \text{bandup} = \text{NULL}, \text{banddown} = \text{NULL}, \\
\text{maxsteps} = 5000, \text{dllname} = \text{NULL}, \text{initfunc} = \text{dllname}, \\
\text{initpar} = \text{parms}, \text{rpar} = \text{NULL}, \text{ipar} = \text{NULL}, \text{nout} = 0, \\
\text{outnames} = \text{NULL}, \text{forcings} = \text{NULL}, \text{initforc} = \text{NULL}, \\
\text{fcontrol} = \text{NULL}, \text{events} = \text{NULL}, \text{lags} = \text{NULL}, \ldots)
\]

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.

\( \text{times} \)

time sequence for which output is wanted; the first value of \( \text{times} \) must be the initial time; if only one step is to be taken; set \( \text{times} = \text{NULL} \).

\( \text{func} \)

either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the \textit{model definition}) at time \( t \), or a character string giving the name of a compiled function in a dynamically loaded shared library.

If \( \text{func} \) is an \texttt{R}-function, it must be defined as: \( \text{func} \leftarrow \text{function}(t, y, \text{parms}, \ldots) \). \( t \) is the current time point in the integration, \( y \) is the current
lsode 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. The derivatives should be specified in the same order as the state variables y.

If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before lsode() is called. See package vignette "compiledCode" for more details.

parms vector or list of parameters used in func or jacfunc.

rtol relative error tolerance, either a scalar or an array as long as y. See details.
atol absolute error tolerance, 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 \( \frac{\partial y_i}{\partial y_j} \), or a string giving the name of a function or subroutine in ‘dllname’ that computes the Jacobian (see vignette "compiledCode" 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 \( \frac{\partial y_i}{\partial y_j} \), 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.

rootfunc if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in ‘dllname’ that computes the root function. The R calling sequence for rootfunc is identical to that of func. rootfunc should return a vector with the function values whose root is sought.

verbose if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.
nroot only used if ‘dllname’ is specified: the number of constraint functions whose roots are desired during the integration; if rootfunc is an R-function, the solver estimates the number of roots.
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: logical, if FALSE names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.

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 per output interval taken by the solver.

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 package vignette "compiledCode".

initfunc: if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".

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 calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

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. These names will be used to label the output matrix.

forcings: only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc: if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".
**fcontrol**  
A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

**events**  
A list that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

**lags**  
A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

...  
additional arguments passed to func and jacfunc allowing this to be a generic function.

**Details**

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

Before using the integrator lsode, 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 lsoda for details.
The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.

**Value**

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine ‘lsoda’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

**Author(s)**

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

**References**


**See Also**

- `rk`,
- `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda`, `lsodes`, `lsodar`, `vode`, `daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.

**Examples**

```r
## Example 1:
## Various ways to solve the same model.
```
## the model, 5 state variables

```r
f1 <- function (t, y, parms) {
    ydot <- vector(len = 5)
    return(list(ydot))
}
```

## the Jacobian, written as a full matrix

```r
fulljac <- function (t, y, parms) {
    jac <- matrix(nrow = 5, ncol = 5, byrow = TRUE,
                  data = c(0.1, -0.2, 0, 0, 0,
                           -0.3, 0.1, -0.2, 0, 0,
                           0, -0.3, 0.1, -0.2, 0,
                           0, 0, -0.3, 0.1, -0.2,
                           0, 0, 0, -0.3, 0.1))
    return(jac)
}
```

## the Jacobian, written in banded form

```r
bandjac <- function (t, y, parms) {
    jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
                  data = c(0, -0.2, -0.2, -0.2, -0.2,
                           0.1, 0.1, 0.1, 0.1, 0.1,
                           -0.3, -0.3, -0.3, -0.3, 0))
    return(jac)
}
```

## initial conditions and output times

```r
yini <- 1:5
times <- 1:20
```

## default: stiff method, internally generated, full Jacobian

```r
out <- lsode(yini, times, f1, parms = 0, jactype = "fullint")
```

## stiff method, user-generated full Jacobian

```r
out2 <- lsode(yini, times, f1, parms = 0, jactype = "fullusr",
               jacfunc = fulljac)
```

## stiff method, internally-generated banded Jacobian

## one nonzero band above (up) and below(down) the diagonal

```r
out3 <- lsode(yini, times, f1, parms = 0, jactype = "bandint",
               bandup = 1, banddown = 1)
```

## stiff method, user-generated banded Jacobian

```r
out4 <- lsode(yini, times, f1, parms = 0, jactype = "bandusr",
               jacfunc = bandjac, bandup = 1, banddown = 1)
```
## non-stiff method

```r
glode <- lsode(yini, times, f1, parms = 0, mf = 10)
```

## Example 2:

### diffusion on a 2-D grid

### partially specified Jacobian

```r
diffusion2D <- function(t, Y, par) {
  y <- matrix(nr = n, nc = n, data = Y)
  dY <- r*y  # production

  ## diffusion in X-direction; boundaries = 0-concentration
  Flux <- -Dx * rbind(y[1,], (y[2:n,] - y[1:(n-1),]), -y[n,]) / dx
  dY <- dY - (Flux[2:(n+1),] - Flux[1:n,]) / dx

  ## diffusion in Y-direction
  Flux <- -Dy * cbind(y[,1], (y[,2:n] - y[,1:(n-1)]), -y[,n]) / dy
  dY <- dY - (Flux[,2:(n+1)] - Flux[,1:n]) / dy

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

### parameters

```r
dy <- dx <- 1  # grid size
Dx <- Dy <- 1  # diffusion coeff, X- and Y-direction
r <- 0.025  # production rate
times <- c(0, 1)
n <- 50
y <- matrix(nr = n, nc = n, 0)
```

```r
pa <- par(ask = FALSE)
```

### initial condition

```r
for (i in 1:n) {
  for (j in 1:n) {
    dst <- (i - n/2)^2 + (j - n/2)^2
    y[i, j] <- max(0, 1 - 1/(n*n) * (dst - n)^2)
  }
}
```

```r
filled.contour(y, color.palette = terrain.colors)
```

## jacfunc need not be estimated exactly

### a crude approximation, with a smaller bandwidth will do.

### Here the half-bandwidth 1 is used, whereas the true

### half-bandwidths are equal to n.

### This corresponds to ignoring the y-direction coupling in the ODEs.
lsodes

Solver for Ordinary Differential Equations (ODE) With Sparse Jacobian

Description

Solves the initial value problem for stiff systems of ordinary differential equations (ODE) in the form:

\[
\frac{dy}{dt} = f(t, y)
\]

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

The R function `lsodes` provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Andrew H. Sherman.

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

Usage

```r
lsodes(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
       jacvec = NULL, sparsetype = "sparseint", nnz = NULL,
       inz = NULL, verbose = FALSE, tcrit = NULL, hmin = 0,
       hmax = NULL, hini = 0, ynames = TRUE, maxord = NULL,
       maxsteps = 5000, lrw = NULL, liw = NULL, dllname = NULL,
       initfunc = dllname, initpar = parms, rpar = NULL,
       ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,
       initforc = NULL, fcontrol=NULL, events=NULL, lags = 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`: time sequence for which output is wanted; the first value of `times` must be the initial time; if only one step is to be taken; set `times = NULL`. 

print(system.time(
  for (i in 1:20) {
    out <- lsode(func = diffusion2D, y = as.vector(y), times = times,
                  parms = NULL, jactype = "bandint", bandup = 1, banddown = 1)

    filled.contour(matrix(nr = n, nc = n, out[2,-1]), zlim = c(0,1),
                   color.palette = terrain.colors, main = i)
    y <- out[2, -1]
  }
)
par(ask = pa)
**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: *func* <- function(*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`. The derivatives should be specified in the same order as the state variables *y*.

If *func* is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `lsodes()` is called. See package vignette "compiledCode" for more details.

**parms**

vector or list of parameters used in `func` or `jacfunc`.

**rtol**

relative error tolerance, either a scalar or an array as long as *y*. See details.

**atol**

absolute error tolerance, either a scalar or an array as long as *y*. See details.

**jacvec**

if not NULL, an R function that computes a column of the Jacobian of the system of differential equations \( \frac{\partial y_i}{\partial y_j} \), or a string giving the name of a function or subroutine in `dllname` that computes the column of the Jacobian (see vignette "compiledCode" for more about this option).

The R calling sequence for `jacvec` is identical to that of `func`, but with extra parameter *j*, denoting the column number. Thus, `jacvec` should be called as: `jacvec = func(t, y, j, parms)` and `jacvec` should return a vector containing column *j* of the Jacobian, i.e. its *i*-th value is \( \frac{\partial y_i}{\partial y_j} \). If this function is absent, `lsodes` will generate the Jacobian by differences.

**sparsetype**

the sparsity structure of the Jacobian, one of "sparseint" or "sparseusr", sparsity estimated internally by `lsodes` or given by user.

**nnz**

the number of nonzero elements in the sparse Jacobian (if this is unknown, use an estimate).

**inz**

(row,column) indices to the nonzero elements in the sparse Jacobian. Necessary if `sparsetype` = "sparseusr"; else ignored.

**verbose**

if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

**tcrit**

if not NULL, then `lsodes` cannot integrate past `tcrit`. The FORTRAN routine `lsodes` 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 logical, if FALSE names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.
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.
maxsteps maximal number of steps per output interval taken by the solver.
lrw the length of the real work array rwork; due to the sparsity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, lsodes will return with a message indicating the size of rwork actually required. Therefore, some experimentation may be necessary to estimate the value of lrw.

For instance, if you get the error:

DLSODES- RWORK length is insufficient to proceed.
Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
In above message, I1 = 27627 I2 = 25932

set lrw equal to 27627 or a higher value

liw the length of the integer work array iwork; due to the sparsity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, lsodes will return with a message indicating the size of iwork actually required. Therefore, some experimentation may be necessary to estimate the value of liw.
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 package vignette "compiledCode".

initfunc if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

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 calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".

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. These names will be used to label the output matrix.

forcings only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the
interval \([\min(\text{times}), \max(\text{times})]\) is done by taking the value at the closest data extreme. See \texttt{forcings} or package vignette "compiledCode".

\textbf{initforc} if not NULL, the name of the forcing function initialisation function, as provided in \texttt{dllname}. It MUST be present if \texttt{forcings} has been given a value. See \texttt{forcings} or package vignette "compiledCode".

\textbf{fcontrol} A list of control parameters for the forcing functions. See \texttt{forcings} or vignette compiledCode.

\textbf{events} A list that specifies events, i.e. when the value of a state variable is suddenly changed. See \texttt{events} for more information.

\textbf{lags} A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

\textbf{...} additional arguments passed to \texttt{func} and \texttt{jacfunc} allowing this to be a generic function.

\section*{Details}

The work is done by the FORTRAN subroutine \texttt{lsodes}, whose documentation should be consulted for details (it is included as comments in the source file \texttt{src/opkdmain.f}). The implementation is based on the November, 2003 version of \texttt{lsodes}, from Netlib.

\texttt{lsodes} is applied for stiff problems, where the Jacobian has a sparse structure.

There are four choices depending on whether \texttt{jacvec} and \texttt{inz} is specified.

If function \texttt{jacvec} is present, then it should return the \(j\)-th column of the Jacobian matrix.

If matrix \texttt{inz} is present, then it should contain indices (row, column) to the nonzero elements in the Jacobian matrix.

If \texttt{jacvec} and \texttt{inz} are present, then the Jacobian is fully specified by the user.

If \texttt{jacvec} is present, but not \texttt{nnz} then the structure of the Jacobian will be obtained from \texttt{NEQ + 1} calls to \texttt{jacvec}.

If \texttt{nnz} is present, but not \texttt{jacvec} then the Jacobian will be estimated internally, by differences.

If neither \texttt{nnz} nor \texttt{jacvec} is present, then the Jacobian will be generated internally by differences, its structure (indices to nonzero elements) will be obtained from \texttt{NEQ + 1} initial calls to \texttt{func}.

If \texttt{nnz} is not specified, it is advisable to provide an estimate of the number of non-zero elements in the Jacobian (\texttt{inz}).

The input parameters \texttt{rtol}, and \texttt{atol} determine the \textbf{error control} performed by the solver. See \texttt{lsoda} for details.

The diagnostics of the integration can be printed to screen by calling \texttt{diagnostics}. If \texttt{verbose} = \texttt{TRUE}, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

\textbf{Models} may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in \texttt{forcings}.  

Examples in both C and FORTRAN are in the ‘doc/examples/dynload’ subdirectory of the deSolve package directory.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of “global” values returned in the next elements of the return from func, plus additional column for the time value. There will be a row for each element in times unless the FORTRAN routine ‘lsoda’ returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Author(s)

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

References


See Also

- rk,
- rk4 and euler for Runge-Kutta integrators.
- lsoda, lsode, lsodar, vode, daspk for other solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
- ode.1D for integrating 1-D models,
- ode.2D for integrating 2-D models,
- ode.3D for integrating 3-D models,
- diagnostics to print diagnostic messages.

Examples

```r
## Various ways to solve the same model.
##
## =------------------------------------------------------------------------=
## The example from lsodes source code
## A chemical model
## =------------------------------------------------------------------------

n <- 12
```
```r
y <- rep(1, n)
dy <- rep(0, n)
times <- c(0, 0.1*(10^(0:4)))
rtol <- 1.0e-4
atol <- 1.0e-6

parms <- c(rk1 = 0.1, rk2 = 10.0, rk3 = 50.0, rk4 = 2.5, rk5 = 0.1,
rk6 = 10.0, rk7 = 50.0, rk8 = 2.5, rk9 = 50.0, rk10 = 5.0,
rk11 = 50.0, rk12 = 50.0,rk13 = 50.0, rk14 = 30.0,
rk15 = 100.0,rk16 = 2.5, rk17 = 100.0,rk18 = 2.5,
rk19 = 50.0, rk20 = 50.0)

# chemistry <- function (time, Y, pars) {
  with (as.list(pars), {
    dy[1] <- -rk1 *Y[1]
    rk18*Y[7]
    rk6*Y[10] - rk9*Y[10]
  })
  return(list(dy))
})

## application 1. lsodes estimates the structure of the Jacobian
## and calculates the Jacobian by differences
out <- lsodes(func = chemistry, y = y, parms = parms, times = times,
  atol = atol, rtol = rtol, verbose = TRUE)

## application 2. the structure of the Jacobian is input
## lsodes calculates the Jacobian by differences
## this is not so efficient...
## elements of Jacobian that are not zero
```

nonzero <- matrix(nc = 2, byrow = TRUE, data = c(
  1, 1, 2, 1,  # influence of sp1 on rate of change of others
  2, 2, 4, 2, 5, 2, 12, 2,
  2, 3, 4, 3, 6, 3, 10, 3,
  2, 4, 4, 4, 9, 4, # d(dy1)/dy4
  2, 5, 5, 9, 5, 12, 5,
  3, 6, 9, 6, 10, 6,
  7, 7, 10, 7, 12, 7,
  8, 8, 10, 8, 11, 8,
  3, 10, 7, 10, 10, 10,
  2, 12, 7, 12, 12, 12)
)

## when run, the default length of rwork is too small
## lsodes will tell the length actually needed
# out2 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
# inz = nonzero, atol = atol, rtol = rtol) # gives warning
out2 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
  sparsetype = "sparseusr", inz = nonzero,
  atol = atol, rtol = rtol, verbose = TRUE, lrw = 351)

# application 3. lsodes estimates the structure of the Jacobian
# the Jacobian (vector) function is input
chemjac <- function (time, Y, j, pars) {
  with (as.list(pars), {
    PDJ <- rep(0, n)
    if (j == 1){
      PDJ[1] <- -r1
      PDJ[2] <- r1
    } else if (j == 2){
      PDJ[12] <- -r15*Y[12]
    } else if (j == 3){
      PDJ[10] <- -r5 - r7*Y[10]
    } else if (j == 4){
      PDJ[2] <- r11*r14
      PDJ[3] <- r11*r14
      PDJ[4] <- -r11*r14 - r4
      PDJ[9] <- r4
    } else if (j == 5){
      PDJ[2] <- r19*r14
      PDJ[5] <- -r19*r14 - r16
      PDJ[9] <- r16
    }
  })
}
else if (j == 6) {
  PDJ[3] <- rk12*rk14
  PDJ[6] <- -rk12*rk14 - rk8
  PDJ[9] <- rk8
  PDJ[10] <- rk12*rk14
} else if (j == 7) {
  PDJ[7] <- -rk20*rk14 - rk18
  PDJ[9] <- rk18
  PDJ[12] <- rk20*rk14
} else if (j == 8) {
  PDJ[8] <- -rk13*rk14 - rk10
  PDJ[10] <- rk13*rk14
} else if (j == 10) {
  PDJ[8] <- rk9
  PDJ[12] <- rk6 - rk17*Y[12]
} else if (j == 12) {
  PDJ[10] <- -rk17*Y[10]
}
return(PDJ)
})
}

out3 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
               jacvec = chemjac, atol = atol, rtol = rtol)

out4 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
               lrw = 351, sparsetype = "sparseusr", inz = nonzero,
               jacvec = chemjac, atol = atol, rtol = rtol, verbose = TRUE)
Description

Solves a system of ordinary differential equations; a wrapper around the implemented ODE solvers

Usage

ode(y, times, func, parms,
    method = c("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk",
               "euler", "rk4", "ode23", "ode45", "radau",
               "bdf", "bdf_d", "adams", "impAdams", "impAdams_d"), ...)

## S3 method for class 'deSolve':
print(x, ...)

Arguments

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

times
- time sequence for which output is wanted; the first value of times must be the initial time.

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: func <- function(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. The derivatives should be specified in the same order as the state variables y.

If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before ode is called. See package vignette "compiledCode" for more details.

parms
- parameters passed to func.

method
- the integrator to use, either a function that performs integration, or a list of class rkMethod, or a string ("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "euler", "rk4", "ode23", "ode45", "radau", "bdf", "bdf_d", "adams", or "impAdams"). The latter three are the backward differentiation formula, the (explicit) Adams and the implicit Adams method respectively (see details). The default integrator used is lsoda.

x
- an object of class deSolve, as returned by the integrators, and to be printed.

... additional arguments passed to the integrator or to the methods.
Details

This is simply a wrapper around the various ode solvers.
See package vignette for information about specifying the model in compiled code.
See the selected integrator for the additional options.

The default integrator used is \texttt{lsoda}.

The option \texttt{method = \"bdf\"} provides a handle to the backward differentiation formula (it is equal to using \texttt{method = \"lsode\"}). It is best suited to solve stiff (systems of) equations.

The option \texttt{method = \"bdf\_d\"} selects the backward differentiation formula that uses Jacobi-Newton iteration (neglecting the off-diagonal elements of the Jacobian (it is equal to using \texttt{method = \"lsode\", mf = 23}). It is best suited to solve stiff (systems of) equations.

\texttt{method = \"adams\"} triggers the Adams method that uses functional iteration (no Jacobian used); (equal to \texttt{method = \"lsode\", mf = 10}. It is often the best choice for solving non-stiff (systems of) equations. Note: when functional iteration is used, the method is often said to be explicit, although it is in fact implicit.

\texttt{method = \"impAdams\"} selects the implicit Adams method that uses Newton- Raphson iteration (equal to \texttt{method = \"lsode\", mf = 12}.

\texttt{method = \"impAdams\_d\"} selects the implicit Adams method that uses Jacobi- Newton iteration, i.e. neglecting all off-diagonal elements (equal to \texttt{method = \"lsode\", mf = 13}.

For very stiff systems, \texttt{method = \"daspk\" may outperform \texttt{method = \"bdf\"}.

Value

A matrix of class \texttt{deSolve} with up to as many rows as elements in \texttt{times} and as many columns as elements in \texttt{y} plus the number of "global" values returned in the second element of the return from \texttt{func}, plus an additional column (the first) for the time value. There will be one row for each element in \texttt{times} unless the integrator returns with an unrecoverable error. If \texttt{y} has a names attribute, it will be used to label the columns of the output value.

Author(s)

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

See Also

- \texttt{plot.deSolve} for plotting the outputs,
- \texttt{ode.band} for solving models with a banded Jacobian,
- \texttt{ode.1D} for integrating 1-D models,
- \texttt{ode.2D} for integrating 2-D models,
- \texttt{ode.3D} for integrating 3-D models,
- \texttt{aquaphy, ccl4model}, where \texttt{ode} is used,
- \texttt{lsoda, lsode, lsodes, lsodar, vode, daspk, radau},
- \texttt{rk, rkMethod},
\texttt{diagnostics} to print diagnostic messages.
### Example1: Predator-Prey Lotka-Volterra model

```r
LVmod <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    Ingestion <- rIng * Prey * Predator
    GrowthPrey <- rGrow * Prey * (1 - Prey / K)
    MortPredator <- rMort * Predator
    dPrey <- GrowthPrey - Ingestion
    dPredator <- Ingestion * assEff - MortPredator
    return(list(c(dPrey, dPredator)))
  })
}
```

```r
pars <- c(rIng = 0.2, # /day, rate of ingestion
          rGrow = 1.0, # /day, growth rate of prey
          rMort = 0.2, # /day, mortality rate of predator
          assEff = 0.5, # -, assimilation efficiency
          K = 10) # mmol/m3, carrying capacity

yini <- c(Prey = 1, Predator = 2)
times <- seq(0, 200, by = 1)
out <- as.data.frame(ode(func = LVmod, y = yini,
                         parms = pars, times = times))

matplot(out$time, out[,, 2:3], type = "l", xlab = "time", ylab = "Conc",
        main = "Lotka-Volterra", lwd = 2)
legend("topright", c("prey", "predator"), col = 1:2, lty = 1:2)
```

### Example2: Substrate-Producer-Consumer Lotka-Volterra model

```r
SPCmod <- function(t, x, parms, input) {
  with(as.list(c(parms, x)), {
    import <- input(t)
    dS <- import - b*S*P + g*C # substrate
    dP <- c*S*P - d*C*P # producer
    dC <- e*P*C - f*C # consumer
    res <- c(dS, dP, dC)
    list(res)
  })
}
```

```r
## Note:
## Function sigimp passed as an argument (input) to model
## (see also lsoda and rk examples)
```
## The parameters
parms <- c(b = 0.001, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)

## vector of timesteps
times <- seq(0, 200, length = 101)

## external signal with rectangle impulse
signal <- as.data.frame(list(times = times,
                            import = rep(0, length(times))))

signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2

sigimp <- approxfun(signal$times, signal$import, rule = 2)

## Start values for steady state
xstart <- c(S = 1, P = 1, C = 1)

## Solve model
out <- ode(y = xstart, times = times,
            func = SPCmod, parms, input = sigimp)

## Default plot method
plot(out, type = "l")

## User specified plotting
mf <- par(mfrow = c(1, 2))
matplot(out[,1], out[,2:4], type = "l", xlab = "time", ylab = "state")
legend("topright", col = 1:3, lty = 1:3, legend = c("S", "P", "C"))
plot(out[,"P"], out[,"C"], type = "l", lwd = 2, xlab = "producer",
     ylab = "consumer")
par(mfrow = mf)

---

ode.1D  

**Solver For Multicomponent 1-D Ordinary Differential Equations**

**Description**

Solves a system of ordinary differential equations resulting from 1-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

**Usage**

ode.1D(y, times, func, parms, nspec = NULL, dimens = NULL,
        method= c("lsoda", "lsode", "lsodes", "lsodar", "vode", "dасpk",
                  "euler", "rk4", "ode23", "ode45", "radau", "bdf",
                  "adams", "impAdams"),
        names = NULL, bandwidth = 1, ...)

Arguments

- **y**: the initial (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
- **times**: time sequence for which output is wanted; the first value of `times` must be the initial time.
- **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: `func <- function(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`. The derivatives should be specified in the same order as the state variables `y`.

  If `func` is a character string then integrator `lsodes` will be used. See details.

- **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 integrator. Use "vode", "lsode", "lsoda", "lsodar", "daspk", or "lsodes" if the model is very stiff; "impAdams" or "radau" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via `rkMethod`. For instance, `method = rkMethod("ode45ck")` will trigger the Cash-Karp method of order 4(5).

- **names**: the names of the components; used for plotting.
- **bandwidth**: the number of adjacent boxes over which transport occurs. Normally equal to 1 (box i only interacts with box i-1, and i+1). Values larger than 1 will not work with `method = "lsodes"`. Ignored if the method is explicit.

- **...**: additional arguments passed to the integrator.

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:


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 the selected integrator solves the banded problem.

• The second method uses lsodes. 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 lsodes is called to solve the problem.

As lsodes is used to integrate, 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, ode.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 instance, if you get the error:

DLSODES- RWORK length is insufficient to proceed.
   Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
In above message,  I1 = 27627  I2 = 25932

set lrw equal to 27627 or a higher value

If the model is specified in compiled code (in a DLL), then option 2, based on lsodes is the only solution method.

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

See the selected integrator for the additional options.

Value

A matrix with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

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

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

- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for integrating models with a banded Jacobian
- `ode.2D` for integrating 2-D models
- `ode.3D` for integrating 3-D models
- `lsodes`, `lsode`, `lsoda`, `lsodar`, `vode` for the integration options.

`diagnostics` to print diagnostic messages.

Examples

```r
## example 1
## a predator and its prey diffusing on a flat surface
## in concentric circles
## 1-D model with using cylindrical coordinates
## Lotka-Volterra type biology
## -----------------------------------------------

time <- seq(from = 0, to = 100, by = 0.01)
params <- list(
  rIng = 0.1,
  rGrow = 0.2,
  rMort = 0.3,
  cap = 100,
  Da = 0.1
)

lvmod <- function (time, state, parms, N, rr, ri, dr, dri) {
  with (as.list(parms), {
    PREY <- state[1:N]
    PRED <- state[(N+1):(2*N)]

    ## Fluxes due to diffusion
    ## at internal and external boundaries: zero gradient
    FluxPrey <- -Da * diff(c(PREY[1], PREY, PREY[N]))/dri
    FluxPred <- -Da * diff(c(PRED[1], PRED, PRED[N]))/dri

    ## Biology: Lotka-Volterra model
    Ingestion <- rIng * PREY * PRED
    GrowthPrey <- rGrow * PREY * (1-PREY/cap)
    MortPredator <- rMort * PRED

    ## Rate of change = Flux gradient + Biology
    dPREY <- -diff(ri * FluxPrey)/rr/dr + GrowthPrey - Ingestion
    dPRED <- -diff(ri * FluxPred)/rr/dr + Ingestion * assEff - MortPredator

    return (list(c(dPREY, dPRED)))
  })
}
```

```r
## Model application
```
# model parameters:

R <- 20  # total radius of surface, m
N <- 100  # 100 concentric circles
dr <- R/N  # thickness of each layer
r <- seq(dr/2, by = dr, len = N)  # distance of center to mid-layer
ri <- seq(0, by = dr, len = N+1)  # distance to layer interface
dri <- dr  # dispersion distances

parms <- c(Da = 0.05,  # m2/d, dispersion coefficient
           rIng = 0.2,  # /day, rate of ingestion
           rGrow = 1.0,  # /day, growth rate of prey
           rMort = 0.2,  # /day, mortality rate of pred
           assEff = 0.5,  # -, assimilation efficiency
           cap = 10)  # density, carrying capacity

# Initial conditions: both present in central circle (box 1) only
state <- rep(0, 2*N)

# RUNNING the model:
times <- seq(0, 200, by = 1)  # output wanted at these time intervals

# the model is solved by the two implemented methods:
# 1. Default: banded reformulation
print(system.time(
  out <- ode.1D(y = state, times = times, func = lvmod, parms = parms,
               nspec = 2, names = c("PREY", "PRED"),
               N = N, rr = r, ri = ri, dr = dr, dri = dri)
))

# 2. Using sparse method
print(system.time(
  out2 <- ode.1D(y = state, times = times, func = lvmod, parms = parms,
                 nspec = 2, names = c("PREY", "PRED"),
                 N = N, rr = r, ri = ri, dr = dr, dri = dri,
                 method = "lsodes")
))

# Plotting output

# the data in 'out' consist of: 1st col times, 2-N+1: the prey
# N+2:2*N+1: predators
PREY <- out[, 2:(N+1)]

filled.contour(x = times, y = r, PREY, color = topo.colors,
               xlab = "time, days", ylab = "Distance, m",
               main = "Prey density")

# similar:
## Example 2.
## Biochemical Oxygen Demand (BOD) and oxygen (O2) dynamics
## in a river
## =======================================================================

### Model equations

### Model application

```r
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 # 1-st order consumption

  dBOD <- -diff(FluxBOD)/dx - BODrate
  dO2 <- -diff(FluxO2)/dx - BODrate + p * (O2sat-O2)

  return(list(c(dBOD = dBOD, dO2 = dO2)))
}
```

```r
dx <- 25 # grid size of 25 meters
v <- 1e3 # velocity, m/day
x <- seq(dx/2, 5000, by = dx) # m, distance from river
N <- length(x)
r <- 0.05 # /day, first-order decay of BOD
p <- 0.5 # /day, air-sea exchange rate
O2sat <- 300 # mmol/m3 saturated oxygen conc
O2_0 <- 200 # mmol/m3 riverine oxygen conc
BOD_0 <- 1000 # mmol/m3 riverine BOD concentration
state <- c(rep(200, N), rep(200, N))
times <- seq(0, 20, by = 0.1)
```

```r
out <- ode.1D(y = state, times, O2BOD, parms = NULL,
              method = "lsoda", method.args = list(
                n Dahl = 10000,
                nout = 10000,
                tout = 10000))
```
ode.2D

Solver for 2-Dimensional Ordinary Differential Equations

Description

Solves a system of ordinary differential equations resulting from 2-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

Usage

ode.2D(y, times, func, parms, nspec = NULL, dimens,
        method= c("lsodes", "euler", "rk4", "ode23", "ode45", "adams"),
        names = NULL, cyclicBnd = NULL, ...)

Arguments

y

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

times

time sequence for which output is wanted; the first value of \( \text{times} \) must be the initial time.

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 \( \text{func} \) is an \( R \)-function, it must be defined as: \( \text{func} \leftarrow \text{function}(t, y, \text{parms}, \ldots) \). \( t \) 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 \( \text{func} \).

\( \text{parms} \) is a vector or list of parameters; \( \ldots \) (optional) are any other arguments passed to the function.

The return value of \( \text{func} \) should be a list, whose first element is a vector containing the derivatives of \( y \) with respect to \( t \), and whose next elements are global values that are required at each point in \( \text{times} \). The derivatives should be specified in the same order as the state variables \( y \).
ode.2D

 parole  parameters passed to func.
 nspec   the number of species (components) in the model.
 dimens  2-valued vector with the number of boxes in two dimensions in the model.
 cyclicBnd if not NULL then a number or a 2-valued vector with the dimensions where a cyclic boundary is used - 1: x-dimension, 2: y-dimension; see details.
 names   the names of the components; used for plotting.
 method  the integrator. Use "lsodes" if the model is very stiff; "impAdams" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via rkMethod. For instance, method = rkMethod("ode45ck") will trigger the Cash-Karp method of order 4(5).
         If "lsodes" is used, then also the size of the work array should be specified (lrw) (see lsodes).
         ... additional arguments passed to lsodes.

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, and if lsodes is used as the integrator, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is only occurring between adjacent layers. Then lsodes is called to solve the problem.

If lsodes is used to integrate, 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, ode.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.

For instance, if you get the error:

DLSODES- RWORK length is insufficient to proceed.
         Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
         In above message, I1 = 27627  I2 = 25932

set lrw equal to 27627 or a higher value

In some cases, a cyclic boundary condition exists. This is when the first boxes in x-or y-direction interact with the last boxes. In this case, there will be extra non-zero fringes in the Jacobian which need to be taken into account. The occurrence of cyclic boundaries can be toggled on by specifying argument cyclicBnd. For instance, cyclicBnd = 1 indicates that a cyclic boundary is required only for the x-direction, whereas cyclicBnd = c(1,2) imposes a cyclic boundary for both x- and y-direction. The default is no cyclic boundaries.

See lsodes for the additional options.

If the model is not stiff, then it is more efficient to use one of the explicit integration routines
**Value**

A matrix with up to as many rows as elements in \( \text{times} \) and as many columns as elements in \( y \) plus the number of "global" values returned in the second element of the return from \( \text{func} \), plus an additional column (the first) for the time value. There will be one row for each element in \( \text{times} \) unless the integrator returns with an unrecoverable error. If \( y \) has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes \text{istate} and \text{rstate}, two vectors with several useful elements. The first element of istate returns the conditions under which the last call to the integrator returned. Normal is \text{istate} = 2. If \text{verbose} = \text{TRUE}, the settings of istate and rstate will be written to the screen. See the help for the selected integrator for details.

**Note**

It is advisable though not mandatory to specify both \text{nspec} and \text{dimens}. In this case, the solver can check whether the input makes sense (as \text{nspec} * \text{dimens}[1] * \text{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**

- \text{ode} for a general interface to most of the ODE solvers,
- \text{ode.band} for integrating models with a banded Jacobian
- \text{ode.1D} for integrating 1-D models
- \text{ode.3D} for integrating 3-D models
- \text{lsodes} for the integration options.

\text{diagnostics} to print diagnostic messages.

**Examples**

```r
lvmod2D <- function (time, state, pars, N, Da, dx) {
  NN <- N*N
  Prey <- matrix(nr = N,nc = N,state[1:NN])
  Pred <- matrix(nr = N,nc = N,state[(NN+1):(2*NN)])
  with (as.list(pars), }
```
## Biology

\[
d_{Prey} = r_{Grow} \times Prey \times (1 - \frac{Prey}{K}) - r_{Ing} \times Prey \times Pred
\]

\[
d_{Pred} = r_{Ing} \times Prey \times Pred \times assEff - r_{Mort} \times Pred
\]

zero <- rep(0, N)

## 1. Fluxes in x-direction; zero fluxes near boundaries

\[
Flux_{Prey} = -Da \times \begin{pmatrix} 0 \end{pmatrix}_{1:N} + \frac{Prey_{[2:N]} - Prey_{[1:(N-1)]}}{dx}
\]

\[
Flux_{Pred} = -Da \times \begin{pmatrix} 0 \end{pmatrix}_{1:N} + \frac{Pred_{[2:N]} - Pred_{[1:(N-1)]}}{dx}
\]

## Add flux gradient to rate of change

\[
d_{Prey} = d_{Prey} - \frac{Flux_{Prey}[2:(N+1),] - Flux_{Prey}[1:N,]}{dx}
\]

\[
d_{Pred} = d_{Pred} - \frac{Flux_{Pred}[2:(N+1),] - Flux_{Pred}[1:N,]}{dx}
\]

## 2. Fluxes in y-direction; zero fluxes near boundaries

\[
Flux_{Prey} = -Da \times \begin{pmatrix} 0 \end{pmatrix}_{1:N} + \frac{Prey_{[1:N]} - Prey_{[1:(N-1)]}}{dx}
\]

\[
Flux_{Pred} = -Da \times \begin{pmatrix} 0 \end{pmatrix}_{1:N} + \frac{Pred_{[1:N]} - Pred_{[1:(N-1)]}}{dx}
\]

## Add flux gradient to rate of change

\[
d_{Prey} = d_{Prey} - \frac{Flux_{Prey}[1:N,] - Flux_{Prey}[1:N,]}{dx}
\]

\[
d_{Pred} = d_{Pred} - \frac{Flux_{Pred}[1:N,] - Flux_{Pred}[1:N,]}{dx}
\]

return(list(c(as.vector(dPrey), as.vector(dPred))))

## Model applications

pars <- c(rIng = 0.2, # /day, rate of ingestion
          rGrow = 1.0, # /day, growth rate of prey
          rMort = 0.2 , # /day, mortality rate of predator
          assEff = 0.5, # -, assimilation efficiency
          K = 5 ) # mmol/m3, carrying capacity

R <- 20 # total length of surface, m
N <- 50 # number of boxes in one direction
dx <- R/N # thickness of each layer
Da <- 0.05 # m2/d, dispersion coefficient

NN <- N*N # total number of boxes

## initial conditions

yini <- rep(0, 2*N*N)
cc <- c((NN/2):(NN/2+1)+N/2, (NN/2):(NN/2+1)-N/2)
yini[cc] <- yini[NN+cc] <- 1

## solve model (5000 state variables... use Cash-Karp Runge-Kutta method

times <- seq(0, 50, by = 1)
out <- ode2D(y = yini, times = times, func = lvmod2D, parms = pars,
            dimens = c(N, N), names = c("Prey", "Pred"),
            ...)}

})
ode.2D

\[ N = N, \quad dx = dx, \quad Da = Da, \quad \text{method} = \text{rkMethod("rk45ck")}\]

## plot results
Col <- colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan", "#7FFFFF", "yellow", "#FF7F00", "red", "#7F0000"))

## Not run:
for (i in seq(1, length(times), by = 1))
  image(matrix(nr = N, nc = N, out[i, 2:(NN+1)]),
        col = Col(100), xlab = , zlim = range(out[,2:(NN+1)]))

## similar:
image(out, xlab = "x", ylab = "y")

## End(Not run)

## An example with a cyclic boundary condition.
## Diffusion in 2-D; extra flux on 2 boundaries, cyclic boundary in y

`diffusion2D <- function(t, Y, par) {`
  `y <- matrix(nrow = nx, ncol = ny, data = Y) # vector to 2-D matrix`
  `dY <- -r * y # consumption`
  `BNDx <- rep(1, nx) # boundary concentration`
  `BNDy <- rep(1, ny) # boundary concentration`

  ## diffusion in X-direction; boundaries=imposed concentration
  `Flux <- -Dx * rbind(y[1,]-BNDy,(y[2:nx,]-y[1:(nx-1),]),BNDy-y[nx,])/dx`
  `dY <- dY - (Flux[2:(nx+1),]-Flux[1:nx,])/dx`

  ## diffusion in Y-direction
  `Flux <- -Dy * cbind(y[,1]-BNDx,(y[,2:ny]-y[,1:(ny-1)]),BNDx-y[,ny])/dy`
  `dY <- dY - (Flux[2:(ny+1),]-Flux[1:ny,])/dy`

  ## extra flux on two sides
  `dY[,1] <- dY[,1] + 10`
  `dY[1,] <- dY[1,] + 10`

  ## and exchange between sides on y-direction
  `dY[,ny] <- dY[,ny] + (y[,1] - y[,ny]) * 10`
  `return(list(as.vector(dY)))`
`
## parameters
`dy <- dx <- 1 # grid size`
`Dy <- Dx <- 1 # diffusion coeff, X- and Y-direction`
`r <- 0.05 # consumption rate`

`nx <- 50`
`ny <- 100`
ode.3D

Solver for 3-Dimensional Ordinary Differential Equations

Description

Solves a system of ordinary differential equations resulting from 3-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

Usage

ode.3D(y, times, func, parms, nspec = NULL, dimens, method = c("lsodes", "euler", "rk4", "ode23", "ode45", "adams"), names = NULL, cyclicBnd = NULL, ...)

Arguments

y

times

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 \( \text{func} \) is an R-function, it must be defined as: \( \text{func} \leftarrow \text{function}(t, y, \text{parms}, \ldots) \). \( t \) 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.

The initial (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 sequence for which output is wanted; the first value of \( \text{times} \) must be the initial time.

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 \( \text{func} \) is an R-function, it must be defined as: \( \text{func} \leftarrow \text{function}(t, y, \text{parms}, \ldots) \). \( t \) 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`. The derivatives should be specified in the same order as the state variables `y`.

`parms` parameters passed to `func`.

`nspec` the number of species (components) in the model.

`dimens` 3-valued vector with the number of boxes in three dimensions in the model.

`names` the names of the components; used for plotting.

`cyclicBnd` if not NULL then a number or a 3-valued vector with the dimensions where a cyclic boundary is used - 1: x-dimension, 2: y-dimension, 3: z-dimension.

`method` the integrator. Use "lsodes" if the model is very stiff; "impAdams" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via `rkMethod`. For instance, `method = rkMethod("ode45ck")` will trigger the Cash-Karp method of order 4(5).

... additional arguments passed to `lsodes`.

**Details**

This is the method of choice for 3-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 only occurring between adjacent layers. Then `lsodes` is called to solve the problem.

As `lsodes` is used to integrate, 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, `ode.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.

For instance, if you get the error:

```
DLSODES- RWORK length is insufficient to proceed.
Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
In above message,  I1 = 27627  I2 = 25932
```

set `lrw` equal to 27627 or a higher value.

See `lsodes` for the additional options.
Value

A matrix with up to as many rows as elements in times and as many columns as elements in \( y \) plus the number of "global" values returned in the second element of the return from \( \text{func} \), plus an additional column (the first) for the time value. There will be one row for each element in \( \text{times} \) unless the integrator returns with an unrecoverable error. If \( y \) has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes \( \text{istate} \), and \( \text{rstate} \), two vectors with several useful elements. The first element of \( \text{istate} \) returns the conditions under which the last call to the integrator returned. Normal is \( \text{istate} = 2 \). If \( \text{verbose} = \text{TRUE} \), the settings of \( \text{istate} \) and \( \text{rstate} \) will be written to the screen. See the help for the selected integrator for details.

Note

It is advisable though not mandatory to specify both \( \text{nspec} \) and \( \text{dimens} \). In this case, the solver can check whether the input makes sense (as \( \text{nspec} \times \text{dimens}[1] \times \text{dimens}[2] \times \text{dimens}[3] = \text{length}(y) \)).

Do not use this method for problems that are not 3D!

Author(s)

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

See Also

- \texttt{ode} for a general interface to most of the ODE solvers,
- \texttt{ode.band} for integrating models with a banded Jacobian
- \texttt{ode.1D} for integrating 1-D models
- \texttt{ode.2D} for integrating 2-D models
- \texttt{lsodes} for the integration options.

\texttt{diagnostics} to print diagnostic messages.

Examples

```r
## Diffusion in 3-D; imposed boundary conditions
diffusion3D <- function(t, Y, par) {
  # function to bind two matrices to an array
  mbind <- function (Mat1, Array, Mat2, along = 1) {
    dimens <- dim(Array) + c(0, 0, 2)
    if (along == 3)
      array(dim = dimens, data = c(Mat1, Array, Mat2))
    else if (along == 1)
      aperm(array(dim = dimens,
                   data=c(Mat1, aperm(Array, c(3, 2, 1)), Mat2)), c(3, 2, 1))
    else if (along == 2)
      aperm(array(dim = dimens,
                   data=c(Mat1, aperm(Array, c(2, 1)), Mat2)), c(2, 1))
    else
      aperm(array(dim = dimens,
                   data=c(Mat1, aperm(Array, c(1, 1)), Mat2)), c(1, 1))
  }
  #...
ode.3D

```r
data = c(Mat1, aperm(Array, c(1, 3, 2)), Mat2), c(1, 3, 2))

yy <- array(dim=c(n,n,n),data=Y) # vector to 3-D array
dY <- -r*yy # consumption
BND <- matrix(nr=n,nc=n,data=1) # boundary concentration

## diffusion in x-direction
## new array including boundary concentrations in X-direction
BNDx <- mbind(BND, yy, BND, along = 1)
## diffusive Flux
Flux <- -Dx * (BNDx[2:(n+2),,,] - BNDx[1:(n+1),,,])/dx
## rate of change = - flux gradient
dY[] <- dY[] - (Flux[2:(n+1),,,] - Flux[1:n,,])/dx

## diffusion in y-direction
BNDy <- mbind(BND, yy, BND, along = 2)
Flux <- -Dy * (BNDy[,2:(n+2),] - BNDy[,1:(n+1),])/dy
dY[] <- dY[] - (Flux[,2:(n+1),] - Flux[,1:n])/dy

## diffusion in z-direction
BNDz <- mbind(BND, yy, BND, along = 3)
Flux <- -Dz * (BNDz[,,2:(n+2)] - BNDz[,,1:(n+1)])/dz
dY[] <- dY[] - (Flux[,,2:(n+1)] - Flux[,,1:n])/dz

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

## parameters
dx <- dy <- dz <-1 # grid size
Dx <- Dy <- Dz <-1 # diffusion coeff, X- and Y-direction
r <- 0.025 # consumption rate
n <- 10
y <- array(dim=c(n,n,n),data=10.)

## use lsodes, the default (for n>20, Runge-Kutta more efficient)
print(system.time(
  RES <- ode.3D(y, func = diffusion3D, parms = NULL, dimens = c(n, n, n),
                times = 1:20, lrw = 120000, atol = 1e-10,
                rtol = 1e-10, verbose = TRUE)
))
y <- array(dim = c(n, n, n), data = RES[nrow(RES), -1])
filled.contour(y[, , n/2], color.palette = terrain.colors)

## Not run:
for (i in 1:nrow(RES)) {
  y <- array(dim=c(n,n,n),data=RES[i,-1])
  filled.contour(y[,n/2],main=i,color.palette=terrain.colors)
}

## End(Not run)
```
ode.band

Solver for Ordinary Differential Equations; Assumes a Banded Jacobian

Description

Solves a system of ordinary differential equations.
Assumes a banded Jacobian matrix, but does not rearrange the state variables (in contrast to ode.1D).
Suitable for 1-D models that include transport only between adjacent layers and that model only one species.

Usage

ode.band(y, times, func, parms, nspec = NULL, bandup = nspec, banddown = nspec, method = "lsode", ...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>the initial (state) values for the ODE system, a vector. If y has a name attribute, the names will be used to label the output matrix.</td>
</tr>
<tr>
<td>times</td>
<td>time sequence for which output is wanted; the first value of times must be the initial time.</td>
</tr>
<tr>
<td>func</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>If func is an R-function, it must be defined as: func &lt;- function(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.</td>
</tr>
<tr>
<td></td>
<td>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. The derivatives should be specified in the same order as the state variables y.</td>
</tr>
<tr>
<td>parms</td>
<td>parameters passed to func.</td>
</tr>
<tr>
<td>nspec</td>
<td>the number of <em>species</em> (components) in the model.</td>
</tr>
<tr>
<td>bandup</td>
<td>the number of nonzero bands above the Jacobian diagonal.</td>
</tr>
<tr>
<td>banddown</td>
<td>the number of nonzero bands below the Jacobian diagonal.</td>
</tr>
<tr>
<td>method</td>
<td>the integrator to use, one of &quot;vode&quot;, &quot;lsode&quot;, &quot;lsoda&quot;, &quot;lsodar&quot;, &quot;radau&quot;.</td>
</tr>
<tr>
<td>...</td>
<td>additional arguments passed to the integrator.</td>
</tr>
</tbody>
</table>
Details

This is the method of choice for single-species 1-D reactive transport 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). By default, the model function will have the species arranged as A[1], A[2], A[3], ... B[1], B[2], B[3], ... in this case, use ode.1D.

See the selected integrator for the additional options.

Value

A matrix with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes istate and rstate, two vectors with several elements. See the help for the selected integrator for details. the first element of istate returns the conditions under which the last call to the integrator returned. Normal is istate = 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>

See Also

- ode for a general interface to most of the ODE solvers,
- ode.1D for integrating 1-D models
- ode.2D for integrating 2-D models
- ode.3D for integrating 3-D models
- lsode, lsoda, lsodar, vode for the integration options.
- diagnostics to print diagnostic messages.

Examples

```R
## The Aphid model from Soetaert and Herman, 2009.
## A practical guide to ecological modelling.
## Using R as a simulation platform. Springer.
##
## 1-D diffusion model

## 

```
Aphid <- function(t, APHIDS, parameters) {
  deltax <- c(0.5, rep(1, numboxes-1), 0.5)
  Flux <- -D*diff(c(0, APHIDS, 0))/deltax
  dAPHIDS <- -diff(Flux)/delx + APHIDS*r

  list(dAPHIDS) # the output
}

## ==================
## Model application
## ==================

## the model parameters:
D  <- 0.3 # m2/day diffusion rate
r  <- 0.01 # /day net growth rate
delx <- 1 # m thickness of boxes
numboxes <- 60

## distance of boxes on plant, m, 1 m intervals
Distance <- seq(from = 0.5, by = delx, length.out = numboxes)

## Initial conditions, ind/m2
## aphids present only on two central boxes
APHIDS <- rep(0, times = numboxes)
APHIDS[30:31] <- 1
state <- c(APHIDS = APHIDS) # initialise state variables

## RUNNING the model:
times <- seq(0, 200, by = 1) # output wanted at these time intervals
out <- ode.band(state, times, Aphid, parms = 0, nspec = 1)

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

## the data in 'out' consist of: 1st col times, 2-41: the density
## select the density data
DENSITY <- out[,2:(numboxes + 1)]

filled.contour(x = times, y = Distance, DENSITY, color = topo.colors,
               xlab = "time, days", ylab = "Distance on plant, m",
               main = "Aphid density on a row of plants")

---

**plot.deSolve**

*Plot and Histogram Method for deSolve Objects*

**Description**

Plot the output of numeric integration routines.
Usage

```
## S3 method for class 'deSolve':
plot(x, which = 1:(ncol(x)-1), ask = NULL, ...)
## S3 method for class 'deSolve':
hist(x, which = 1:(ncol(x)-1), ask = NULL, ...)
## S3 method for class 'deSolve':
image(x, which = NULL, ask = NULL,
      add.contour = FALSE, grid = NULL, method="image", ...)
plot.1D (x, which=NULL, ask=NULL, grid=NULL, xyswap = FALSE, ...)
```

Arguments

- **x**: an object of class `deSolve`, as returned by the integrators, and to be plotted.
- **which**: the name(s) or the index to the variables that should be plotted. Default = all variables.
- **ask**: logical; if TRUE, the user is asked before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see `par(ask=.)` and `dev.interactive`.
- **add.contour**: if TRUE, will add contours to the image plot.
- **method**: the name of the plotting method to use, one of "image", "filled.contour", "persp", "contour".
- **grid**: only for image plots and for `plot.1D`: the 1-D grid as a vector (output generated with `ode.1D`, or the x- and y-grid, as a list for output generated with `ode.2D`.
- **xyswap**: if TRUE, then x-and y-values are swapped and the y-axis is from top to bottom. Useful for drawing vertical profiles.
- **...**: additional graphics arguments passed to `plot.default`, `image` or `hist`.

Details

The number of panels per page is automatically determined up to 3 x 3 (par(mfrow = c(3, 3))). This default can be overwritten by specifying user-defined settings for `mfrow` or `mfcol`. Set `mfrow` equal to NULL to avoid the plotting function to change user-defined `mfrow` or `mfcol` settings.

Other graphical parameters can be passed as well. Parameters `xlab` and `ylab` are vectorized, so it is possible to assign specific axis labels to individual plots.

Image plots will only work for 1-D and 2-D variables, as solved with `ode.1D` and `ode.2D`. In the first case, an image with `times` as x- and the `grid` as y-axis will be created. in second case, an x-y plot will be created, for all times. Unless `ask = FALSE`, the user will be asked to confirm page changes. For images, it is possible to pass an argument `method` which can take the values "image" (default), "filled.contour", "contour" or "persp", in order to use the respective plotting method.

`plot` will always have `times` on the x-axis. For problems solved with `ode.1D`, it may be more useful to use `plot.1D` which will plot how spatial variables change with time. These plots will have the `grid` on the x-axis.
See Also

print.deSolve, ode, deSolve

Examples

## A Predator-Prey model with 4 species in matrix formulation

LVmatrix <- function(t, n, parms) {
  with(parms, {
    dn <- r * n + n * (A %*% n)
    return(list(c(dn)))
  })
}

parms <- list(
  r = c(r1 = 0.1, r2 = 0.1, r3 = -0.1, r4 = -0.1),
  A = matrix(c(0.0, 0.0, -0.2, 0.01, # prey 1
               0.0, 0.0, 0.02, -0.1, # prey 2
               0.2, 0.02, 0.0, 0.0, # predator 1; prefers prey 1
               0.01, 0.1, 0.0, 0.0), # predator 2; prefers prey 2
             nrow = 4, ncol = 4, byrow=TRUE)
)
times <- seq(from = 0, to = 500, by = 0.1)
y <- c(prey1 = 1, prey2 = 1, pred1 = 2, pred2 = 2)
out <- ode(y, times, LVmatrix, parms)

## Basic line plot
plot(out, type = "l")

## User-specified axis labels
plot(out, type = "l", ylab = c("Prey 1", "Prey 2", "Pred 1", "Pred 2"),
     xlab = "Time (d)", main = "Time Series")

## Set user-defined mfrow
pm <- par (mfrow = c(2, 2))
plot(out[, "prey1"], out[, "pred1"], xlab="prey1",
     ylab="pred1", type = "l", lwd = 2)
plot(out[, "prey2"], out[, "pred2"], xlab="prey2",
     ylab="pred2", type = "l", lwd = 2)

## restore graphics parameters
par ("mfrow" = pm)

hist(out, col = "darkblue", breaks = 50)
## The Aphid model from Soetaert and Herman, 2009.
## A practical guide to ecological modelling.
## Using R as a simulation platform. Springer.
## =======================================================================
## 1-D diffusion model
## ================
## Model equations
## ================
Aphid <- function(t, APHIDS, parameters) {
  deltax <- c(0.5, rep(1, numboxes - 1), 0.5)
  Flux <- -D * diff(c(0, APHIDS, 0))/deltax
  dAPHIDS <- -diff(Flux)/delx + APHIDS * r
  list(dAPHIDS)
}
## ==================
## Model application
## ==================
## the model parameters:
D <- 0.3 # m2/day diffusion rate
r <- 0.01 # /day net growth rate
delx <- 1 # m thickness of boxes
numboxes <- 60
## distance of boxes on plant, m, 1 m intervals
Distance <- seq(from = 0.5, by = delx, length.out = numboxes)
## Initial conditions, ind/m2
## aphids present only on two central boxes
APHIDS <- rep(0, times = numboxes)
APHIDS[30:31] <- 1
state <- c(APHIDS = APHIDS) # initialise state variables
## RUNNING the model:
times <- seq(0, 200, by = 1) # output wanted at these time intervals
out <- ode.1D(state, times, Aphid, parms = 0, nspec = 1)
image(out, grid = Distance, main = "Aphid model", ylab = "distance, m")
image(out, grid = Distance, main = "Aphid model", ylab = "distance, m",
  method = "persp", border = NA, theta = 30)
## Not run:
plot(out, ask = FALSE, mfrow = c(1, 1))
plot.1D(out, ask = FALSE, type = "l", lwd = 2, xyswap = TRUE)
## End(Not run)

Implicit Runge-Kutta RADAU IIA
\textbf{Description}

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]

or linearly implicit differential algebraic equations in the form:

\[ M \frac{dy}{dt} = f(t, y) \]

The \texttt{R} function \texttt{radau} provides an interface to the Fortran solver RADAU5, written by Ernst Hairer and G. Wanner, which implements the 3-stage RADAU IIA method. It implements the implicit Runge-Kutta method of order 5 with step size control and continuous output. The system of ODEs or DAEs is written as an \texttt{R} function or can be defined in compiled code that has been dynamically loaded.

\textbf{Usage}

\begin{verbatim}
  radau(y, times, func, parms, nind = c(length(y), 0, 0),
         rtol = 1e-6, atol = 1e-6, jacfunc = NULL, jactype = "fullint",
         mass = NULL, massup = NULL, massdown = NULL,
         verbose = FALSE, hmax = NULL, hini = 0, ynames = TRUE,
         bandup = NULL, banddown = NULL, maxsteps = 5000,
         dllname = NULL, initfunc = dllname, initpar = parms,
         rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
         forcings = NULL, initforc = NULL, fcontrol = NULL, ...)
\end{verbatim}

\textbf{Arguments}

- \texttt{y} the initial (state) values for the ODE system. If \texttt{y} has a name attribute, the names will be used to label the output matrix.
- \texttt{times} time sequence for which output is wanted; the first value of \texttt{times} must be the initial time; if only one step is to be taken; set \texttt{times} = \texttt{NULL}.
- \texttt{func} either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the model definition) at time \texttt{t}, or the right-hand side of the equation

  \[ M \frac{dy}{dt} = f(t, y) \]

  if a DAE. (if \texttt{mass} is supplied then the problem is assumed a DAE).

  \texttt{func} can also be a character string giving the name of a compiled function in a dynamically loaded shared library.

  If \texttt{func} is an \texttt{R}-function, it must be defined as:

  \begin{verbatim}
  func <- function(t, y, parms,...).
  \end{verbatim}

  \texttt{t} is the current time point in the integration, \texttt{y} is the current estimate of the variables in the ODE system. If the initial values \texttt{y} has a \texttt{name} attribute, the names will be available inside \texttt{func}. \texttt{parms} is a vector or list of parameters; ...

  (optional) are any other arguments passed to the function.

  The return value of \texttt{func} should be a list, whose first element is a vector containing the derivatives of \texttt{y} with respect to \texttt{time}, and whose next elements are
global values that are required at each point in times. The derivatives should
be specified in the same order as the state variables \( y \).

If \( \text{func} \) is a string, then \( \text{dllname} \) must give the name of the shared library
(without extension) which must be loaded before \( \text{radau}() \) is called. See deSolve package vignette "compiledCode" for more details.

\( \text{parms} \) vector or list of parameters used in \( \text{func} \) or \( \text{jacfunc} \).

\( \text{nind} \) if a DAE system: a three-valued vector with the number of variables of index 1, 2, 3 respectively. The equations must be defined such that the index 1 variables precede the index 2 variables which in turn precede the index 3 variables. The sum of the variables of different index should equal \( N \), the total number of variables.

\( \text{rtol} \) relative error tolerance, either a scalar or an array as long as \( y \). See details.

\( \text{atol} \) absolute error tolerance, either a scalar or an array as long as \( y \). See details.

\( \text{jacfunc} \) if not \( \text{NULL} \), an \( \text{R} \) function that computes the Jacobian of the system of differential equations \( \partial y_i / \partial y_j \), or a string giving the name of a function or subroutine in ‘dllname’ that computes the Jacobian (see vignette "compiledCode" from package deSolve, for more about this option).

In some circumstances, supplying \( \text{jacfunc} \) can speed up the computations, if the system is stiff. The \( \text{R} \) calling sequence for \( \text{jacfunc} \) is identical to that of \( \text{func} \).

If the Jacobian is a full matrix, \( \text{jacfunc} \) should return a matrix \( \partial y_i / \partial y_j \), 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 \( \text{R} \) and FORTRAN store matrices).

If the Jacobian is banded, \( \text{jacfunc} \) should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See example.

\( \text{jactype} \) the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user.

\( \text{mass} \) the mass matrix. If not \( \text{NULL} \), the problem is a linear implicit DAE and defined as \( M \, dy / dt = f(t, y) \). If the mass-matrix \( M \) is full, it should be of dimension \( n^2 \) where \( n \) is the number of \( y \)-values; if banded the number of rows should be less than \( n \), and the mass-matrix is stored diagonal-wise with element \((i, j)\) stored in \( \text{mass}(1 - j + \text{mumas} + 1, j) \).

If \( \text{mass} = \text{NULL} \) then the model is an ODE (default)

\( \text{massup} \) number of non-zero bands above the diagonal of the \( \text{mass} \) matrix, in case it is banded.

\( \text{massdown} \) number of non-zero bands below the diagonal of the \( \text{mass} \) matrix, in case it is banded.

\( \text{verbose} \) if \( \text{TRUE} \): full output to the screen, e.g. will print the diagnostics of the integration - see details.

\( \text{hmax} \) an optional maximum value of the integration stepsize. If not specified, \( \text{hmax} \) is set to the largest difference in \( \text{times} \), to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

\( \text{hini} \) initial step size to be attempted; if 0, the initial step size is set equal to 1e-6. Usually 1e-3 to 1e-5 is good for stiff equations
ynames logical, if FALSE names of state variables are not passed to function func; this
may speed up the simulation especially for multi-D models.

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 average maximal number of steps per output interval taken by the solver. This
argument is defined such as to ensure compatibility with the Livermore-solvers.
RADAU only accepts the maximal number of steps for the entire integration,
and this is calculated as length(times) * maxsteps.

dllname a string giving the name of the shared library (without extension) that con-
tains all the compiled function or subroutine definitions refered to in func and
jacfunc. See vignette "compiledCode" from package deSolve.

initfunc if not NULL, the name of the initialisation function (which initialises values of
parameters), as provided in 'dllname'. See vignette "compiledCode" from package deSolve.

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 calculated in the DLL - you have to perform this
check in the code - See vignette "compiledCode" from package deSolve.

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. These
names will be used to label the output matrix.

forcings only used if 'dllname' is specified: a list with the forcing function data sets, each
present as a two-columned matrix, with (time, value); interpolation outside the
interval [min(times), max(times)] is done by taking the value at the closest
data extreme.

See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided
in 'dllname'. It MUST be present if forcings has been given a value. See
forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette
compiledCode.

... additional arguments passed to func and jacfunc allowing this to be a generic
function.
Details

The work is done by the FORTRAN subroutine \texttt{RADAU5}, whose documentation should be consulted for details. The implementation is based on the Fortran 77 version from January 18, 2002.

There are four standard choices for the Jacobian which can be specified with \texttt{jactype}.

The options for \texttt{jactype} are

\begin{verbatim}
  jactype = "fullint" a full Jacobian, calculated internally by the solver.
  jactype = "fullusr" a full Jacobian, specified by user function \texttt{jacfunc}.
  jactype = "bandusr" a banded Jacobian, specified by user function \texttt{jacfunc}; the size of the
                 bands specified by \texttt{bandup} and \texttt{banddown}.
  jactype = "bandint" a banded Jacobian, calculated by \texttt{radau}; the size of the bands specified by
                  \texttt{bandup} and \texttt{banddown}.
\end{verbatim}

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

The input parameters \texttt{rtol}, and \texttt{atol} determine the error control performed by the solver, which roughly keeps the local error of $y(i)$ below $\texttt{rtol}(i) \times \text{abs}(y(i)) + \text{atol}(i)$.

The diagnostcs of the integration can be printed to screen by calling \texttt{diagnostics}. If \texttt{verbose} = \texttt{TRUE}, the diagnostcs will be written to the screen at the end of the integration.

See vignette("deSolve") from the \texttt{deSolve} package for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" from package \texttt{deSolve} for details.

Information about linking forcing functions to compiled code is in \texttt{forcings} (from package \texttt{deSolve}).

Value

A matrix of class \texttt{deSolve} with up to as many rows as elements in \texttt{times} and as many columns as elements in \texttt{y} plus the number of "global" values returned in the next elements of the return from \texttt{func}, plus and additional column for the time value. There will be a row for each element in \texttt{times} unless the FORTRAN routine returns with an unrecoverable error. If \texttt{y} has a names attribute, it will be used to label the columns of the output value.

Author(s)

Karline Soetaert

References

See Also

- `ode` for a general interface to most of the ODE solvers,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `daspk` for integrating DAE models up to index 1

```r
# Example 1: ODE
# Various ways to solve the same model.

## the model, 5 state variables
f1 <- function (t, y, parms) {
  ydot <- vector(len = 5)
  return(list(ydot))
}

## the Jacobian, written as a full matrix
fulljac <- function (t, y, parms) {
  jac <- matrix(nrow = 5, ncol = 5, byrow = TRUE,
                 data = c(0.1, -0.2, 0, 0, 0,
                          -0.3, 0.1, -0.2, 0, 0,
                          0, -0.3, 0.1, -0.2, 0,
                          0, 0, -0.3, 0.1, -0.2,
                          0, 0, 0, -0.3, 0.1))
  return(jac)
}

## the Jacobian, written in banded form
bandjac <- function (t, y, parms) {
  jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
                data = c(0, -0.2, -0.2, -0.2, -0.2,
                         0.1, 0.1, 0.1, 0.1, 0.1,
                         -0.3, -0.3, -0.3, -0.3, 0))
  return(jac)
}

## initial conditions and output times
yini <- 1:5
```
times <- 1:20

## default: stiff method, internally generated, full Jacobian
out <- radau(yini, times, f1, parms = 0)
plot(out)

## stiff method, user-generated full Jacobian
out2 <- radau(yini, times, f1, parms = 0, jactype = "fullusr",
              jacfunc = fulljac)

## stiff method, internally-generated banded Jacobian
## one nonzero band above (up) and below(down) the diagonal
out3 <- radau(yini, times, f1, parms = 0, jactype = "bandint",
              bandup = 1, banddown = 1)

## stiff method, user-generated banded Jacobian
out4 <- radau(yini, times, f1, parms = 0, jactype = "bandusr",
              jacfunc = bandjac, bandup = 1, banddown = 1)

## Example 2: ODE
## stiff problem from chemical kinetics

Chemistry <- function (t, y, p) {
  dy3 <- 3.e7*y[2]^2
  list(c(dy1, dy2, dy3))
}

times <- 10^(seq(0, 10, by = 0.1))
yini <- c(y1 = 1.0, y2 = 0, y3 = 0)
out <- radau(func = Chemistry, times = times, y = yini, parms = NULL)
plot(out, log = "x", type = "l", lwd = 2)

## Example 3: DAE
## Car axis problem, index 3 DAE, 8 differential, 2 algebraic equations
## from
## F. Mazzia and C. Magherini. Test Set for Initial Value Problem Solvers,
## release 2.4. Department
## of Mathematics, University of Bari and INdAM, Research Unit of Bari,
## February 2008.
## Available at http://www.dm.uniba.it/~testset.
##
## Problem is written as M*y = f(t,y,p).
## caraxisfun implements the right-hand side:

caraxisfun <- function(t, y, parms) {
  with(as.list(y), {
    M <- matrix(c(1, 2, -1, 0, 0, 0, 0, 0), 2, 4)
    f <- function(t, y) {
      with(as.list(y), {
        f2 <- 3.e7*y[2]^2
        list(c(f1, f2))
      })
    }
    f(t, y)
  })
}
yb <- r * sin(w * t)
xb <- sqrt(L * L - yb * yb)
Ll <- sqrt(xl^2 + yl^2)
Lr <- sqrt((xr - xb)^2 + (yr - yb)^2)
dxl <- ul; dyl <- vl; dxr <- ur; dyr <- vr
dul <- (L0-Ll) * xl/Ll + 2 * lam2 * (xl-xr) + lam1*xb
dvl <- (L0-Ll) * yl/Ll + 2 * lam2 * (yl-yr) + lam1*yb - k * g
dur <- (L0-Lr) * (xr-xb)/Lr - 2 * lam2 * (xl-xr)
dvr <- (L0-Lr) * (yr-yb)/Lr - 2 * lam2 * (yl-yr) - k * g
c1 <- xb * xl + yb * yl
c2 <- (xl - xr)^2 + (yl - yr)^2 - L * L

list(c(dxl, dyl, dxr, dyr, dul, dvl, dur, dvr, c1, c2))
}
}

eps <- 0.01; M <- 10; k <- M * eps^2/2;
L <- 1; L0 <- 0.5; r <- 0.1; w <- 10; g <- 1

yini <- c(xl = 0, yl = L0, xr = L, yr = L0,
ul = -L0/L, vl = 0,
ur = -L0/L, vr = 0,
lam1 = 0, lam2 = 0)

# the mass matrix
Mass <- diag(nrow = 10, 1)
Mass

# index of the variables: 4 of index 1, 4 of index 2, 2 of index 3
index <- c(4, 4, 2)
times <- seq(0, 3, by = 0.01)
out <- radau(y = yini, mass = Mass, times = times, func = caraxisfun,
parms = parameter, nind = index)
plot(out, which = 1:4, type = "l", lwd = 2)

---

**Description**

Solving initial value problems for non-stiff systems of first-order ordinary differential equations (ODEs).
The \texttt{R} function \texttt{rk} is a top-level function that provides interfaces to a collection of common explicit one-step solvers of the Runge-Kutta family with fixed or variable time steps. The system of ODE’s is written as an \texttt{R} function (which may, of course, use \texttt{.C}, \texttt{.Fortran}, \texttt{.Call}, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded. A vector of parameters is passed to the ODEs, so the solver may be used as part of a modeling package for ODEs, or for parameter estimation using any appropriate modeling tool for non-linear models in \texttt{R} such as \texttt{optim}, \texttt{nls}, \texttt{nlm} or \texttt{nlme}.

**Usage**

\begin{verbatim}
rk(y, times, func, parms, rtol = 1e-6, atol = 1e-6, verbose = FALSE, tcrit = NULL, hmin = 0, hmax = NULL, hini = hmax, ynames = TRUE, method = rkMethod("rk45dp7", ... ), maxsteps = 5000, dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings = NULL, initforc = NULL, fcontrol = NULL, events = NULL, ...)\end{verbatim}

**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**
  - times at which explicit estimates for \( y \) are desired. The first value in \( \text{times} \) must be the initial time.

- **func**
  - either an \texttt{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 \( \text{func} \) is an \texttt{R}-function, it must be defined as: \( \text{func} <- \text{function}(t, y, \text{parms}, ...) \). \( t \) 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 \texttt{names} attribute, the names will be available inside \( \text{func}. \text{parms} \) is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of \( \text{func} \) should be a list, whose first element is a vector containing the derivatives of \( y \) with respect to \( t \), and whose next elements are global values that are required at each point in \( \text{times} \). The derivatives should be specified in the same order as the state variables \( y \).
  
  If \( \text{func} \) is a string, then \( \text{dllname} \) must give the name of the shared library (without extension) which must be loaded before \( \text{rk} \) is called. See package vignette "compiledCode" for more details.

- **parms**
  - vector or list of parameters used in \( \text{func} \).

- **rtol**
  - relative error tolerance, either a scalar or an array as long as \( y \). Only applicable to methods with variable time step, see details.

- **atol**
  - absolute error tolerance, either a scalar or an array as long as \( y \). Only applicable to methods with variable time step, see details.

- **tcrit**
  - if not \texttt{NULL}, then \texttt{rk} cannot integrate past \texttt{tcrit}. This parameter is for compatibility with other solvers.
verbose

a logical value that, when TRUE, triggers more verbose output from the ODE solver.

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 maximum of hini and the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified. Note that hmin and hmax are ignored by fixed step methods like "rk4" or "euler".

hini

initial step size to be attempted; if 0, the initial step size is determined automatically by solvers with flexible time step. Setting hini = 0 for fixed step methods forces setting of internal time steps identically to external time steps provided by times.

ynames

if FALSE: names of state variables are not passed to function func; this may speed up the simulation especially for large models.

method

the integrator to use. This can either be a string constant naming one of the pre-defined methods or a call to function rkMethod specifying a user-defined method. The most common methods are the fixed-step methods "euler", second and fourth-order Runge Kutta ("rk2", "rk4"), or the variable step methods Bogacki-Shampine "rk23bs", Runge-Kutta-Fehlberg "rk34f", the fifth-order Cash-Karp method "rk45ck" or the fifth-order Dormand-Prince method with seven stages "rk45dp7". As a suggestion, one may use "rk23" (alias "ode23") for simple problems and "rk45dp7" (alias "ode45") for rough problems.

maxsteps

average maximal number of steps per output interval taken by the solver. This argument is defined such as to ensure compatibility with the Livermore-solvers. rk only accepts the maximal number of steps for the entire integration, and this is calculated as length(times) * maxsteps.

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 package vignette "compiledCode".

initfunc

if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

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 calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".
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.

forcings only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\text{min(times)}, \text{max(times)}]\) is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events A list that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information. Not also that if events are specified, then polynomial interpolation is switched off and integration takes place from one external time step to the next, with an internal step size less than or equal the difference of two adjacent points of `times`.

... additional arguments passed to `func` allowing this to be a generic function.

Details

Function `rk` is a generalized implementation that can be used to evaluate different solvers of the Runge-Kutta family of explicit ODE solvers. A pre-defined set of common method parameters is in function `rkMethod` which also allows to supply user-defined Butcher tables.

The input parameters `rtol`, and `atol` determine the error control performed by the solver. The solver will control the vector of estimated local errors in `y`, according to an inequality of the form

\[
\text{max-norm of } \left( \frac{e}{\text{ewt}} \right) \leq 1,
\]

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:

\[
\text{rtol} \times \text{abs}(y) + \text{atol}
\]

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

Models can be defined in R as a user-supplied R-function, that must be called 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.

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 second element contains output variables that are required at each point in time. Examples are given below.

Value

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the integration routine returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.
Note

Arguments rpar and ipar are provided for compatibility with lsoda.
Starting with version 1.8 implicit Runge-Kutta methods are also supported by this general rk interface, however their implementation is still experimental. Instead of this you may consider radau for a specific full implementation of an implicit Runge-Kutta method.

Author(s)

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

References


See Also

For most practical cases, solvers of the Livermore family (i.e. the ODEPACK solvers, see below) are superior. Some of them are also suitable for stiff ODEs, differential algebraic equations (DAEs), or partial differential equations (PDEs).

- rkMethod for a list of available Runge-Kutta parameter sets,
- rk4 and euler for special versions without interpolation (and less overhead),
- lsoda, lsode, lsodes, lsodar, vode, daspk for solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
- ode.1D for integrating 1-D models,
- ode.2D for integrating 2-D models,
- ode.3D for integrating 3-D models,

diagnostics to print diagnostic messages.

Examples

```r
## Example: Resource-producer-consumer Lotka-Volterra model
## Note:
## parameters are a list, names accessible via "with" function
```
## (see also ode and lsoda examples)

```r
SPCmod <- function(t, x, parms) {
  S <- x[1]  # substrate
  P <- x[2]  # producer
  C <- x[3]  # consumer

  with(parms, {
    import <- approx(signal$times, signal$import, t)$y
    dS <- import - b * S * P + g * C
    dP <- c * S * P - d * C * P
    dC <- e * P * C - f * C
    res <- c(dS, dP, dC)
    list(res)
  })
}
```

## vector of timesteps
```r
times <- seq(0, 100, length = 101)
```

## external signal with rectangle impulse
```r
signal <- as.data.frame(list(times = times,
                              import = rep(0, length(times))))
signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2
```

## Parameters for steady state conditions
```r
parms <- list(b = 0.0, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)
```

## Start values for steady state
```r
y <- xstart <- c(S = 1, P = 1, C = 1)
```

```r
system.time(
  # Euler method
  out1 <- as.data.frame(rk(xstart, times, SPCmod, parms,
                            hini = 0.1, method = "euler"))

  # classical Runge-Kutta 4th order
  out2 <- as.data.frame(rk(xstart, times, SPCmod, parms,
                            hini = 1, method = "rk4"))

  # Dormand-Prince method of order 5(4)
  out3 <- as.data.frame(rk(xstart, times, SPCmod, parms,
                            hmax = 1, method = "rk45dp7"))
)
```

```r
mf <- par(mfrow = c(2,2))
plot(out1$time, out1$S, type = "l", ylab = "Substrate")
lines(out2$time, out2$S, col = "red", lty = "dotted", lwd = 2)
lines(out3$time, out3$S, col = "green", lty = "dotted")

plot(out1$time, out1$P, type = "l", ylab = "Producer")
lines(out2$time, out2$P, col = "red", lty = "dotted")
```
lines(out3$time, out3$P, col = "green", lty = "dotted")

plot (out1$time, out1$C, type = "l", ylab = "Consumer")
lines(out2$time, out2$C, col = "red", lty = "dotted", lwd = 2)
lines(out3$time, out3$C, col = "green", lty = "dotted")

plot (out1$P, out1$C, type = "l", xlab = "Producer", ylab = "Consumer")
lines(out2$P, out2$C, col = "red", lty = "dotted", lwd = 2)
lines(out3$P, out3$C, col = "green", lty = "dotted")

legend("center", legend = c("euler", "rk4", "rk45dp7"),
    lty = c(1, 3, 3), lwd = c(1, 2, 1),
    col = c("black", "red", "green"))
par(mfrow = mf)

---

rk4

Solve System of ODE (Ordinary Differential Equation)s by Euler’s Method or Classical Runge-Kutta 4th Order Integration.

Description

Solving initial value problems for systems of first-order ordinary differential equations (ODEs) using Euler’s method or the classical Runge-Kutta 4th order integration.

Usage

euler(y, times, func, parms, verbose = FALSE, ynames = TRUE, dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings = NULL, initforc = NULL, fcontrol = NULL, ...)
rk4(y, times, func, parms, verbose = FALSE, ynames = TRUE, dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings = NULL, initforc = NULL, fcontrol = 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 times at which explicit estimates for y are desired. The first value in times must be the initial time.

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: func <- function(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`. The derivatives should be specified in the same order as the state variables `y`.

If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `rk4` is called. See package vignette "compiledCode" for more details.

`parms` vector or list of parameters used in `func`.

`verbose` a logical value that, when `TRUE`, triggers more verbose output from the ODE solver.

`ynames` if `FALSE`: names of state variables are not passed to function `func`; this may speed up the simulation especially for large 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`. See package vignette "compiledCode".

`initfunc` if not `NULL`, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode", and 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 calculated in the DLL - you have to perform this check in the code. See package vignette "compiledCode".

`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.

`forcings` only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time, value); interpolation outside the interval [min(`times`), max(`times`)] is done by taking the value at the closest data extreme.

See `forcings` or package vignette "compiledCode".

`initforc` if not `NULL`, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if `forcings` has been given a value. See `forcings` or package vignette "compiledCode".

`fcontrol` A list of control parameters for the forcing functions. See `forcings` or vignette compiledCode.

`...` additional arguments passed to `func` allowing this to be a generic function.
Details

rk4 and euler are special versions of the two fixed step solvers with less overhead and less functionality (e.g. no interpolation and no events) compared to the generic Runge-Kutta codes called by rk.

If you need different internal and external time steps or want to use events, please use: rk(y, times, func, parms, method = "rk4") or rk(y, times, func, parms, method = "euler").

See help pages of rk and rkMethod for details.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the integration routine returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

For most practical cases, solvers with flexible timestep (e.g. rk(method = "ode45")) and especially solvers of the Livermore family (ODEPACK, e.g. lsoda) are superior.

Author(s)

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

See Also

- rkMethod for a list of available Runge-Kutta parameter sets,
- rk for the more general Runge-Code,
- lsoda, lsode, lsodes, lsodar, vode, daspk for solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
- ode.1D for integrating 1-D models,
- ode.2D for integrating 2-D models,
- ode.3D for integrating 3-D models,
- dede for integrating models with delay differential equations,
- diagnostics to print diagnostic messages.

Examples

```r
## Example: Analytical and numerical solutions of logistic growth
##
## the derivative of the logistic
```

logist <- function(t, x, parms) {
  with(as.list(parms), {
    dx <- r * x[1] * (1 - x[1]/K)
    list(dx)
  })
}

time <- 0:100
N0 <- 0.1; r <- 0.5; K <- 100
parms <- c(r = r, K = K)

## analytical solution
plot(time, K/(1 + (K/N0-1) * exp(-r*time)), ylim = c(0, 120),
     type = "l", col = "red", lwd = 2)

## reasonable numerical solution
out <- as.data.frame(rk4(x, time, logist, parms))
points(out$time, out$N, pch = 16, col = "blue", cex = 0.5)

## same time step, systematic under-estimation
out <- as.data.frame(euler(x, time, logist, parms))
points(out$time, out$N, pch = 1)

## unstable result
out <- as.data.frame(euler(x, time, logist, parms))
points(out$time, out$N, pch = 8, cex = 0.5)

## method with automatic time step
out <- as.data.frame(lsoda(x, time, logist, parms))
points(out$time, out$N, pch = 1, col = "green")

legend("bottomright",
       c("analytical", "rk4, h=2", "euler, h=2",
          "euler, h=4", "lsoda"),
       lty = c(1, NA, NA, NA, NA), lwd = c(2, 1, 1, 1, 1),
       pch = c(NA, 16, 1, 8, 1),
       col = c("red", "blue", "black", "black", "green"))

rkMethod

Collection of Parameter Sets (Butcher Arrays) for the Runge-Kutta Family of ODE Solvers

Description

This function returns a list specifying coefficients and properties of ODE solver methods from the Runge-Kutta family.
Usage

rkMethod(method = NULL, ...)

Arguments

method a string constant naming one of the pre-defined methods of the Runge-Kutta family of solvers. The most common methods are the fixed-step methods "euler", "rk2", "rk4" or the variable step methods "rk23bs" (alias "ode23"), "rk45dp7" (alias "ode45") or "rk78f".

... specification of a user-defined solver, see Value and example below.

Details

This function supplies method settings for rk or ode. If called without arguments, the names of all implemented solvers of the Runge-Kutta family are returned.

The following comparison gives an idea how the algorithms of deSolve are related to similar algorithms of other simulation languages:

<table>
<thead>
<tr>
<th>rkMethod</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;euler&quot;</td>
<td>Euler's Method</td>
</tr>
<tr>
<td>&quot;rk2&quot;</td>
<td>2nd order Runge-Kutta, fixed time step (Heun’s method)</td>
</tr>
<tr>
<td>&quot;rk4&quot;</td>
<td>classical 4th order Runge-Kutta, fixed time step</td>
</tr>
<tr>
<td>&quot;rk23&quot;</td>
<td>Runge-Kutta, order 2(3), Octave: ode23</td>
</tr>
<tr>
<td>&quot;rk23bs&quot;, &quot;ode23&quot;</td>
<td>Bogacki-Shampine, order 2(3), Matlab: ode23</td>
</tr>
<tr>
<td>&quot;rk34f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 3(4)</td>
</tr>
<tr>
<td>&quot;rk45ck&quot;</td>
<td>Runge-Kutta Cash-Karp, order 4(5)</td>
</tr>
<tr>
<td>&quot;rk45f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 4(5), Octave: ode45, pair=1</td>
</tr>
<tr>
<td>&quot;rk45e&quot;</td>
<td>Runge-Kutta-England, order 4(5)</td>
</tr>
<tr>
<td>&quot;rk45dp6&quot;</td>
<td>Dormand-Prince, order 4(5), local order 6</td>
</tr>
<tr>
<td>&quot;rk45dp7&quot;, &quot;ode45&quot;</td>
<td>Dormand-Prince 4(5), local order 7</td>
</tr>
<tr>
<td>(also known as dopr5, MATLAB: ode45, Octave: ode45, pair=0)</td>
<td></td>
</tr>
<tr>
<td>&quot;rk78f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 7(8)</td>
</tr>
<tr>
<td>&quot;rk78dp&quot;</td>
<td>Dormand-Prince, order 7(8)</td>
</tr>
</tbody>
</table>

Note that this table is based on the Runge-Kutta coefficients only, but the algorithms differ also in their implementation, in their stepsize adaption strategy and interpolation methods.

The table reflects the state at time of writing and it is of course possible that implementations change.

Methods "rk45dp7" (alias "ode45") and "rk45ck" contain specific and efficient built-in interpolation schemes (dense output).

As an alternative, Neville-Aitken polynomials can be used to interpolate between time steps. This is available for all methods and may be useful to speed up computation if no dense-output formula is available. Note however, that this can introduce considerable local error; it is disabled by default (see nknots below).
Value

A list with the following elements:

- **ID**: name of the method (character)
- **varstep**: boolean value specifying if the method allows for variable time step (TRUE) or not (FALSE).
- **FSAL**: (first same as last) optional boolean value specifying if the method allows re-use of the last function evaluation (TRUE) or not (FALSE or NULL).
- **A**: coefficient matrix of the method. As `link{rk}` supports only explicit methods, this matrix must be lower triangular. A must be a vector for fixed step methods where only the subdiagonal values are different from zero.
- **b1**: coefficients of the lower order Runge-Kutta pair.
- **b2**: coefficients of the higher order Runge-Kutta pair (optional, for embedded methods that allow variable time step).
- **c**: coefficients for calculating the intermediate time steps.
- **d**: optional coefficients for built-in polynomial interpolation of the outputs from internal steps (dense output), currently only available for method `rk45dp7` (Dormand-Prince).
- **densetype**: optional integer value specifying the dense output formula; currently `densetype = 1` for `rk45dp7` (Dormand-Prince) and `densetype = 2` for `rk45ck` (Cash-Karp). Undefined values (e.g., `densetype = NULL`) disable dense output.
- **stage**: number of function evaluations needed (corresponds to number of rows in A).
- **Qerr**: global error order of the method, important for automatic time-step adjustment.
- **nknots**: integer value specifying the order of interpolation polynomials for methods without dense output. If `nknots < 2` (the default) then internal interpolation is switched off and integration is performed step by step between external time steps.

  If `nknots` is between 3 and 8, Neville-Aitken polynomials are used, which need at least `nknots + 1` internal time steps. Local interpolation may speed up integration but can lead to local errors higher than the tolerance, especially if external and internal time steps are very different.

- **alpha**: optional tuning parameter for stepsize adjustment. If `alpha` is omitted, it is set to `1/Qerr - 0.75beta`. The default value is `1/Qerr` (for `beta = 0`).
- **beta**: optional tuning parameter for stepsize adjustment. Typical values are 0 (default) or `0.4/Qerr`.

Note

- Adaptive stepsize Runge-Kutta are preferred if the solution contains parts where it changes fast, and parts where nothing much happens. They will take small steps over bumpy ground and long steps over uninteresting terrain.
- As a suggestion, one may use "rk23" (alias "ode23") for simple problems and "rk45dp7" (alias "ode45") for rough problems. The default solver is "rk45dp7" (alias "ode45"), because of its relatively high order (4), re-use of the last intermediate steps (FSAL = first same as last) and built-in polynomial interpolation (dense output).
rkMethod

- Solver "rk23bs", that supports also FSAL, may be useful for slightly stiff systems if demands on precision are relatively low.
- Another good choice, assuring medium accuracy, is the Cash-Karp Runge-Kutta method, "rk45ck".
- Classical "rk4" is traditionally used in cases where an adequate stepsize is known a-priori or if external forcing data are provided for fixed time steps only and frequent interpolation of external data needs to be avoided.
- Method "rk45dp7" (alias "ode45") contains an efficient built-in interpolation scheme (dense output) based on intermediate function evaluations.

Starting with version 1.8 implicit Runge-Kutta (irk) methods are also supported by the general rk interface, however their implementation is still experimental. Instead of this you may consider radau for a specific full implementation of an implicit Runge-Kutta method.

Author(s)

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

References

MATLAB (R) is a registered property of The Mathworks Inc. http://www.mathworks.com/

See Also

rk, ode
Examples

rkMethod() # returns the names of all available methods
rkMethod("rk45dp7") # parameters of the Dormand-Prince 5(4) method
rkMethod("ode45") # an alias for the same method

func <- function(t, x, parms) {
  with(as.list(c(parms, x)),{
    dP <- a * P - b * C * P
    dC <- b * P * C - c * C
    res <- c(dP, dC)
    list(res)
  })
}

times <- seq(0, 200, length = 101)
parms <- c(a = 0.1, b = 0.1, c = 0.1)
x <- c(P = 2, C = 1)

ode(x, times, func, parms, method = rkMethod("rk4"))
ode(x, times, func, parms, method = "ode45")

o0 <- ode(x, times, func, parms, method = "lsoda")
o1 <- ode(x, times, func, parms, method = rkMethod("rk45dp7"))

## disable dense-output interpolation
## and use Neville-Aitken polynomials only
o2 <- ode(x, times, func, parms,
  method = rkMethod("rk45dp7", densetype = NULL, nknots=5))

## show differences between methods
par(mfrow=c(3,1))
matplot(o1[,1], o1[,-1], type="l", xlab="Time", main="State Variables")
matplot(o1[,1], o2[,-1], type="p", pch=16, add=TRUE)
matplot(o0[,1], o1[,-1]-o0[,-1], type="l", lty="solid", xlab="Time",
  main="Difference: rk45dp7 - lsoda")
matplot(o1[,1], o2[,-1]-o1[,-1], type="l", lty="dashed", xlab="Time",
  main="difference: Neville-Aitken - Dense Output")
abline(h=0, col="grey")

## disable interpolation completely
## and integrate explicitly between external time steps
o3 <- ode(x, times, func, parms,
  method = rkMethod("rk45dp7", densetype = NULL, nknots = 0))

## define and use a new rk method
ode(x, times, func, parms,
  method = rkMethod(ID = "midpoint",
    varstep = FALSE,
    A = c(0, 1/2),
    b1 = c(0, 1),
    c = c(0, 1/2),
    stage = 2,
```r
Qerr = 1

## compare method diagnostics
times <- seq(0, 200, length = 10)
o1 <- ode(x, times, func, parms, method = rkMethod("rk45ck"))
o1b <- ode(x, times, func, parms, method = rkMethod("rk78dp"))
diagnostics(o1)
diagnostics(o1b)
```

---

**Description**

A model that describes oxygen consumption in a marine sediment.

One state variable:

- sedimentary organic carbon,

Organic carbon settles on the sediment surface (forcing function Flux) and decays at a constant rate. The equation is simple:

\[
\frac{dC}{dt} = \text{Flux} - kC
\]

This model is written in **FORTRAN**.

**Usage**

```r
SCOC(times, y = NULL, parms, Flux, ...)
```

**Arguments**

- `times`: time sequence for which output is wanted; the first value of `times` must be the initial time,
- `y`: the initial value of the state variable; if `NULL` it will be estimated based on `Flux` and `parms`,
- `parms`: the model parameter, `k`,
- `Flux`: a data set with the organic carbon deposition rates,
- `...`: any other parameters passed to the integrator `ode` (which solves the model).

**Details**

The model is implemented primarily to demonstrate the linking of FORTRAN with **R**-code.

The source can be found in the `doc/examples/dynload` subdirectory of the package.
Author(s)

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

References


See Also

ccl4model, the CCl4 inhalation model.
aquaphy, the algal growth model.

Examples

```r
## Forcing function data
Flux <- matrix(ncol = 2, byrow = TRUE, data = c(
  1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186,
  93, 0.140, 103, 0.255, 113, 0.231, 123, 0.309, 133, 1.127, 143, 1.923,
  153, 1.091, 163, 1.001, 173, 1.691, 183, 1.404, 194, 1.226, 204, 0.767,
  214, 0.893, 224, 0.737, 234, 0.772, 244, 0.726, 254, 0.624, 264, 0.439,
  274, 0.168, 284, 0.280, 294, 0.202, 304, 0.193, 315, 0.286, 325, 0.599,
  335, 1.889, 345, 0.996, 355, 0.681, 365, 1.135)

c <- c(k = 0.01)
times <- 1:365
out <- SCOC(times, parms = c, Flux = Flux)

plot(out$time, out$Depo, type = "l", col = "red")
lines(out$time, out$Mineralisation, col = "blue")

## Constant interpolation of forcing function - left side of interval
fcontrol <- list(method = "constant")

out2 <- SCOC(times, parms = c, Flux = Flux, fcontrol = fcontrol)

plot(out2$time, out2$Depo, type = "l", col = "red")
lines(out2$time, out2$Mineralisation, col = "blue")
```

---

Time Lagged Values of State Variables and Derivatives.
Description

Functions \texttt{lagvalue} and \texttt{lagderiv} provide access to past (lagged) values of state variables and derivatives.

They are to be used with function \texttt{dede}, to solve delay differential equations.

Function \texttt{timestep} gives the current and next timestep of the integration.

Usage

\begin{verbatim}
lagvalue(t, nr)
lagderiv(t, nr)
timestep(prev = TRUE)
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{t} the time for which the lagged value is wanted; this should be no larger than the current simulation time and no smaller than the initial simulation time.
  \item \texttt{nr} the number of the lagged value; if \texttt{NULL} then all state variables or derivatives are returned.
  \item \texttt{prev} if \texttt{TRUE} will return the timestep previously used; when \texttt{FALSE} will return the time step to be currently tried by the integrator.
\end{itemize}

Details

The \texttt{lagvalue} and \texttt{lagderiv} can only be called during the integration, the lagged time should not be smaller than the initial simulation time, nor should it be larger than the current simulation time.

Cubic Hermite interpolation is used to obtain an accurate interpolant at the requested lagged time.

Function \texttt{timestep} returns the current simulation timestep; if the simulation has not yet started, will return 1. Use with care.

Value

a scalar (or vector) with the lagged value(s).

Author(s)

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

See Also

\texttt{dede}, for how to implement delay differential equations.
Examples

```r
## exercise 6 from Shampine and Thompson, 2000
## solving delay differential equations with dde23
## two lag values
## ==============================================================

## the derivative function
## ==============================================================
derivs <- function(t, y, parms) {
    History <- function(t) c(cos(t), sin(t))
    if (t < 1)
        lag1 <- History(t - 1)[1]
    else
        lag1 <- lagvalue(t - 1)[1] # returns a vector; select first element
    if (t < 2)
        lag2 <- History(t - 2)[2]
    else
        lag2 <- lagvalue(t - 2,2) # faster than lagvalue(t - 2)[2]
    dy1 <- lag1 * lag2
    dy2 <- -y[1] * lag2
    list(c(dy1, dy2), lag1 = lag1, lag2 = lag2)
}

## parameters
## ==============================================================
r <- 3.5; m <- 19

## initial values and times
## ==============================================================
yinit <- c(y1 = 0, y2 = 0)
times <- seq(0, 20, by = 0.01)

## solve the model
## ==============================================================
yout <- dede(y = yinit, times = times, func = derivs,
              parms = NULL, atol = 1e-9)

## plot results
## ==============================================================
```
The predator-prey model with time lags, from Hale
problem 1 from Shampine and Thompson, 2000
solving delay differential equations with dde23

a vector with lag values

---

the derivative function
---

predprey <- function(t, y, parms) {
  tlag <- t - 1
  if (tlag < 0)
    ylag <- c(80, 30)
  else
    ylag <- lagvalue(tlag)  # returns a vector
  list(c(dy1, dy2))
}

---

parameters
---

a <- 0.25; b <- -0.01; c <- -1; d <- 0.01; m <- 200

---

initial values and times
---

yinit <- c(y1 = 80, y2 = 30)
times <- seq(0, 100, by = 0.01)

---

solve the model
---

yout <- dede(y = yinit, times = times, func = predprey, parms = NULL)

---

display, plot results
---

plot(yout, type = "l", lwd = 2, main = "Predator-prey model", mfrow = c(2, 2))
plot(yout[,2], yout[,3], xlab = "y1", ylab = "y2", type = "l", lwd = 2)
diagnostics(yout)
A neutral delay differential equation (lagged derivative)
\[ y'(t) = -y'(t-1), \quad y(t) \quad t < 0 = \frac{1}{t} \]

---

### the derivative function
```r
derivs <- function(t, y, parms) {
  tlag <- t - 1
  if (tlag < 0)
    dylag <- -1
  else
    dylag <- lagderiv(tlag)
  list(c(dy = -dylag), dylag = dylag)
}
```

### initial values and times
```r
yinit <- 0
times <- seq(0, 4, 0.001)
```

### solve the model
```r
yout <- dede(y = yinit, times = times, func = derivs, parms = NULL)
```

### display, plot results
```r
plot(yout, type = "l", lwd = 2)
```

---

**vode**

*Solver for Ordinary Differential Equations (ODE)*

**Description**

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]
The \texttt{R} function \texttt{vode} provides an interface to the FORTRAN ODE solver of the same name, written by Peter N. Brown, Alan C. Hindmarsh and George D. Byrne.

The system of ODE’s is written as an \texttt{R} function or be defined in compiled code that has been dynamically loaded.

In contrast to \texttt{lsoda}, the user has to specify whether or not the problem is stiff and choose the appropriate solution method.

\texttt{vode} is very similar to \texttt{lsode}, but uses a variable-coefficient method rather than the fixed-step-interpolate methods in \texttt{lsode}. In addition, in vode it is possible to choose whether or not a copy of the Jacobian is saved for reuse in the corrector iteration algorithm; In \texttt{lsode}, a copy is not kept.

**Usage**

\begin{verbatim}
vode(y, times, func, parms, rtol = 1e-6, atol = 1e-8,  
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 = 5000,  
dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL,  
ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,  
initforc = NULL, fcontrol=NULL, events=NULL, lags = NULL,...)
\end{verbatim}

**Arguments**

- \texttt{y} the initial (state) values for the ODE system. If \texttt{y} has a name attribute, the names will be used to label the output matrix.
- \texttt{times} time sequence for which output is wanted; the first value of \texttt{times} must be the initial time; if only one step is to be taken; set \texttt{times = NULL}.
- \texttt{func} either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the \textit{model definition}) at time \(t\), or a character string giving the name of a compiled function in a dynamically loaded shared library.
  
  If \texttt{func} is an \texttt{R}-function, it must be defined as: \texttt{func <- function(t, y, parms,...).} \(t\) is the current time point in the integration, \texttt{y} is the current estimate of the variables in the ODE system. If the initial values \texttt{y} has a \texttt{names} attribute, the names will be available inside \texttt{func}. \texttt{parms} is a vector or list of parameters; ... (optional) are any other arguments passed to the function.
  
  The return value of \texttt{func} should be a list, whose first element is a vector containing the derivatives of \texttt{y} with respect to \texttt{time}, and whose next elements are global values that are required at each point in \texttt{times}. The derivatives should be specified in the same order as the state variables \texttt{y}.
  
  If \texttt{func} is a string, then \texttt{dllname} must give the name of the shared library (without extension) which must be loaded before \texttt{vode()} is called. See package vignette "compiledCode" for more details.

- \texttt{parms} vector or list of parameters used in \texttt{func} or \texttt{jacfunc}.
- \texttt{rtol} relative error tolerance, either a scalar or an array as long as \texttt{y}. See details.
- \texttt{atol} absolute error tolerance, either a scalar or an array as long as \texttt{y}. See details.
if not NULL, an \texttt{R} function that computes the Jacobian of the system of differential equations $\partial \dot{y}_i / \partial y_j$, or a string giving the name of a function or subroutine in \texttt{dllname} that computes the Jacobian (see vignette "compiledCode" for more about this option).

In some circumstances, supplying \texttt{jacfunc} can speed up the computations, if the system is stiff. The \texttt{R} calling sequence for \texttt{jacfunc} is identical to that of \texttt{func}.

If the Jacobian is a full matrix, \texttt{jacfunc} should return a matrix $\partial \dot{y}_i / \partial y_j$, where the $i$th row contains the derivative of $\dot{y}_i$ with respect to $y_j$, or a vector containing the matrix elements by columns (the way \texttt{R} and FORTRAN store matrices).

If the Jacobian is banded, \texttt{jacfunc} should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of \texttt{lsode}.

the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if \texttt{mf} is not NULL.

the "method flag" passed to function \texttt{vode} - overrules \texttt{jactype} - provides more options than \texttt{jactype} - see details.

if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

if not NULL, then \texttt{vode} cannot integrate past \texttt{tcrit}. The FORTRAN routine \texttt{dvode} overshoots its targets (times points in the vector \texttt{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 \texttt{tcrit}.

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 \texttt{hmin} if you don’t know why!

an optional maximum value of the integration stepsize. If not specified, \texttt{hmax} is set to the largest difference in \texttt{times}, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

initial step size to be attempted; if 0, the initial step size is determined by the solver.

logical; if \texttt{FALSE}: names of state variables are not passed to function \texttt{func }; this may speed up the simulation especially for multi-D models.

the maximum order to be allowed. \texttt{NULL} uses the default, i.e. order 12 if implicit Adams method (\texttt{meth} = 1), order 5 if BDF method (\texttt{meth} = 2). Reduce \texttt{maxord} to save storage space.

number of non-zero bands above the diagonal, in case the Jacobian is banded.

number of non-zero bands below the diagonal, in case the Jacobian is banded.

maximal number of steps per output interval taken by the solver.

a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in \texttt{func} and \texttt{jacfunc}. See package vignette "compiledCode".
initfunc if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

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 calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

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. These names will be used to label the output matrix.

forcings only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. forcings or package vignette "compiledCode"

events A list that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

... additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

Before using the integrator vode, 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

jac = "fullint": a full Jacobian, calculated internally by vode, corresponds to mf = 22,

jac = "fullusr": a full Jacobian, specified by user function jacfunc, corresponds to mf = 21,
jac = "bandusr": a banded Jacobian, specified by user function jacfunc; the size of the bands specified by bandup and banddown, corresponds to mf = 24.

jac = "bandint": a banded Jacobian, calculated by vode; 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, -11, -12, -14, -15, -21, -22, -24, -25.

mf is a signed two-digit integer, \( mf = JSV \times (10 \times METH + MITER) \), where

- \( JSV = \text{SIGN}(mf) \) indicates the Jacobian-saving strategy: \( JSV = 1 \) means a copy of the Jacobian is saved for reuse in the corrector iteration algorithm. \( JSV = -1 \) means a copy of the Jacobian is not saved.

- 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.

The example for integrator lsode demonstrates how to specify both a banded and full Jacobian.

The input parameters rtol, and atol determine the error control performed by the solver. If the request for precision exceeds the capabilities of the machine, vode will return an error code. See lsoda for details.

The diagnostics of the integration can be printed to screen by calling diagnostics. If verbose = TRUE, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.
Value

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine 'lsoda' returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

Author(s)

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

References


Netlib: http://www.netlib.org

See Also

- `rk`
- `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda, lsode, lsodes, lsodar, daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.
Examples

```r
chaos <- function(t, state, parameters) {
  with(as.list(c(state)), {
    dx <- -8/3 * x + y * z
    dy <- -10 * (y - z)
    dz <- -x * y + 28 * y - z

    list(c(dx, dy, dz))
  })
}

state <- c(x = 1, y = 1, z = 1)
times <- seq(0, 100, 0.01)
out <- vode(state, times, chaos, 0)
plot(out, type = "l") # all versus time
plot(out[,"x"], out[,"y"], type = "l", main = "Lorenz butterfly",
     xlab = "x", ylab = "y")
```

```r
Flux <- matrix(ncol = 2, byrow = TRUE, data = c(
  1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186,
  93, 0.140,103, 0.255,113, 0.231,123, 0.309,133, 1.127,143, 1.923,
  153,1.091,163, 1.001,173, 1.691,183, 1.404,194, 1.226,204, 0.767,
  214,0.893,224, 0.737,234, 0.772,244, 0.726,254, 0.624,264, 0.439,
  274,0.168,284, 0.280,294, 0.202,304, 0.193,315, 0.286,325, 0.599,
  335,1.889,345, 0.996,355, 0.681,365, 1.135))

deps <- c(k = 0.01)
meanDepo <- mean(approx(Flux[,1], Flux[,2], xout = seq(1, 365, by = 1))$y)
Yini <- meanDepo/parms
times <- 1:365
out <- as.data.frame(vode(Yini, times, func = "scocder",}
parms = parms, dllname = "deSolve",
initforc = "scocforc", forcings = Flux,
initfunc = "scocpar", nout = 2,
outnames = c("Mineralisation", "Depo"))}

plot(out$time, out$Depo, type = "l", col = "red")
lines(out$time, out$Mineralisation, col = "blue")

## Constant interpolation of forcing function - left side of interval
fcontrol <- list(method = "constant")
out2 <- as.data.frame( vode(Yini, times, func = "scocder",
parms = parms, dllname = "deSolve",
initforc = "scocforc", forcings = Flux, fcontrol = fcontrol,
initfunc = "scocpar", nout = 2,
outnames = c("Mineralisation", "Depo")))

plot(out2$time, out2$Depo, type = "l", col = "red")
lines(out2$time, out2$Mineralisation, col = "blue")

## Constant interpolation of forcing function - middle of interval
fcontrol <- list(method = "constant", f = 0.5)
out3 <- as.data.frame( vode(Yini, times, func = "scocder",
parms = parms, dllname = "deSolve",
initforc = "scocforc", forcings = Flux, fcontrol = fcontrol,
initfunc = "scocpar", nout = 2,
outnames = c("Mineralisation", "Depo")))

lines(out3$time, out3$Depo, type = "l", col = "orange", lty = 2)

---

**zvode**  
_Solver for Ordinary Differential Equations (ODE) for COMPLEX variables_

**Description**

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]

where \( dy \) and \( y \) are complex variables.

The R function `zvode` provides an interface to the FORTRAN ODE solver of the same name, written by Peter N. Brown, Alan C. Hindmarsh and George D. Byrne.
Usage

\[ \text{zvode}(y, \text{times}, \text{func}, \text{parms}, \text{rtol} = 1e-6, \text{atol} = 1e-8, \]
\[ \text{jacfunc} = \text{NULL}, \text{jactype} = \text{"fullint"}, \text{mf} = \text{NULL}, \text{verbose} = \text{FALSE}, \]
\[ \text{tcrit} = \text{NULL}, \text{hmin} = 0, \text{hmax} = \text{NULL}, \text{hini} = 0, \text{ynames} = \text{TRUE}, \]
\[ \text{maxord} = \text{NULL}, \text{bandup} = \text{NULL}, \text{banddown} = \text{NULL}, \text{maxsteps} = 5000, \]
\[ \text{dllname} = \text{NULL}, \text{initfunc} = \text{dllname}, \text{initpar} = \text{parms}, \text{rpar} = \text{NULL}, \]
\[ \text{ipar} = \text{NULL}, \text{nout} = 0, \text{outnames} = \text{NULL}, \text{forcings} = \text{NULL}, \]
\[ \text{initforc} = \text{NULL}, \text{fcontrol} = \text{NULL}, \ldots) \]

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. `y` must be complex.
- `times` time sequence for which output is wanted; the first value of `times` must be the initial time; if only one step is to be taken; set `times = NULL`.
- `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: `func <- function(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`. The derivatives should be specified in the same order as the state variables `y`. They should be complex numbers.
  - If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `zvode()` is called. See package vignette "compiledCode" for more details.
- `parms` vector or list of parameters used in `func` or `jacfunc`.
- `rtol` relative error tolerance, either a scalar or an array as long as `y`. See details.
- `atol` absolute error tolerance, 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 \( \partial y_i / \partial y_j \), or a string giving the name of a function or subroutine in `dllname` that computes the Jacobian (see vignette "compiledCode" 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 \( dy/dy \), 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). Its elements should be complex numbers.
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 `zvode` - overrules `jactype` - provides more options than `jactype` - see details.

### verbose

If `TRUE`: full output to the screen, e.g. will print the diagnostics of the integration - see details.

### tcrit

If not `NULL`, then `zvode` cannot integrate past `tcrit`. The FORTRAN routine `dvode` 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

Logical; if `FALSE`: names of state variables are not passed to function `func`; this may speed up the simulation especially for multi-D models.

### 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 per output interval taken by the solver.

### 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`. See package vignette "compiledCode".

### initfunc

If not `NULL`, the name of the initialisation function (which initialises values of parameters), as provided in `dllname`. See package vignette "compiledCode".

### 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 calculated in the DLL - you have to perform this check in the code - See package vignette "compiledCode".

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. These names will be used to label the output matrix.

forcings only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\min(\text{times}), \max(\text{times})]\) is done by taking the value at the closest data extreme. See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in `dllname`. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. forcings or package vignette "compiledCode".

... additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

see vode, the double precision version, for details

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the FORTRAN routine 'lsoda' returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

(adapted from the zvode.f source code):

When using zvode for a stiff system, it should only be used for the case in which the function f is analytic, that is, when each f(i) is an analytic function of each y(j). Analyticity means that the partial derivative df(i)/dy(j) is a unique complex number, and this fact is critical in the way zvode solves the dense or banded linear systems that arise in the stiff case. For a complex stiff ODE system in which f is not analytic, zvode is likely to have convergence failures, and for this problem one should instead use ode on the equivalent real system (in the real and imaginary parts of y).

Author(s)

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

 Also, LLNL Report UCRL-98412, June 1988.


Netlib: http://www.netlib.org

See Also

vode for the double precision version

Examples

```R
# Example 1 - very simple example
# df/dt = li*f, where li is the imaginary unit
# The initial value is f(0) = 1 = 1+0i
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

ZODE<-function(Time, f, Pars) {
  df <- 1i * f
  return(list(df))
}

pars <- NULL
yini <- c(f = 1+0i)
times <- seq(0, 2*pi, length = 100)
out <- zvode(func = ZODE, y = yini, parms = pars, times = times, 
atol = 1e-10, rtol = 1e-10)

# The analytical solution to this ODE is the exp-function:
# f(t) = exp(li*t)
# = cos(t)+li*sin(t) (due to Euler's equation)

analytical.solution <- exp(1i * times)

# compare numerical and analytical solution
tail(cbind(out[,2], analytical.solution))
```

```
### Example 2 - example in "zvode.f",
### df/dt = 1i*f  (same as above ODE)
### dg/dt = -1i*g*g*f  (an additional ODE depending on f)

### Initial values are
### g(0) = 1/2.1 and
### z(0) = 1

```r
ZODE2<-function(Time,State,Pars) {
  with(as.list(State), {
    df <- 1i * f
    dg <- -1i * g*g * f
    return(list(c(df, dg)))
  })
}
```

```r
yini <- c(f = 1 + 0i, g = 1/2.1 + 0i)
times <- seq(0, 2*pi, length = 100)
out <- zvode(func = ZODE2, y = yini, parms = NULL, times = times,
atol = 1e-10, rtol = 1e-10)
```

### The analytical solution is
### f(t) = exp(1i*t)  (same as above)
### g(t) = 1/(f(t) + 1.1)

```r
analytical <- cbind(f = exp(1i * times), g = 1/(exp(1i * times) + 1.1))
```

### compare numerical solution and the two analytical ones:
```r
tail(cbind(out[,2], analytical[,1]))
```
Index

*Topic datasets
  ccl4data, 8

*Topic hplot
  plot.deSolve, 86

*Topic math
  daspk, 11
  lsoda, 37
  lsodar, 44
  lsode, 51
  lsodes, 58
  ode, 66
  ode.1D, 70
  ode.2D, 75
  ode.3D, 80
  ode.band, 84
  radau, 90
  rk, 97
  rk4, 102
  rkMethod, 106
  vode, 115
  zvode, 122

*Topic models
  aquaphy, 4
  ccl4model, 9
  SCOC, 110

*Topic package
  deSolve-package, 2

*Topic utilities
  dede, 21
  diagnostics, 25
  diagnostics.deSolve, 26
  DLLfunc, 27
  DLLres, 29
  events, 31
  forcings, 34
  timelags, 112
  .C, 12, 37, 97
  .Call, 12, 37, 97
  .Fortran, 37, 97
  approx, 36
  approxfun, 35, 36
  aquaphy, 4, 9, 68, 111
  ccl4data, 8, 9
  ccl4model, 5, 9, 32, 68, 111
  daspk, 3, 11, 30, 42, 49, 55, 62, 68, 94, 100, 104, 120
  dede, 3, 15, 21, 40, 47, 54, 61, 104, 112, 118
  deSolve, 88
  deSolve(deSolve-package), 2
  deSolve-package, 2
  dev.interactive, 87
  diagnostics, 16, 17, 25, 41, 42, 48, 49, 55, 56, 62, 63, 68, 72, 77, 82, 85, 93, 94, 100, 104, 119, 120
  diagnostics.deSolve, 25, 26
  DLLfunc, 3, 27
  DLLres, 3, 29
  euler, 3, 17, 42, 49, 55, 62, 100, 120
  euler(rk4), 102
  events, 3, 31, 36, 40, 45, 47, 51, 54, 61, 99, 118
  forcings, 3, 15, 16, 32, 34, 40, 41, 47, 48, 54, 55, 61, 62, 92, 93, 99, 103, 104, 118, 119, 125
  gnls, 41
  hist, 87
  hist.deSolve(plot.deSolve), 86
  image, 87
  image.deSolve(plot.deSolve), 86
  lagderiv, 3, 22
  lagderiv(timelags), 112
  lagvalue, 3, 22

129
lagvalue (timelags), 112
lsoda, 3, 16, 17, 21, 37, 44, 48, 49, 51, 55, 62, 67, 72, 85, 100, 104, 116, 119, 120
lsodar, 3, 17, 31, 32, 42, 44, 55, 62, 68, 72, 85, 100, 104, 120
lsode, 3, 17, 32, 42, 49, 51, 62, 68, 72, 85, 100, 116, 119, 120
lsodes, 3, 17, 42, 49, 55, 58, 68, 72, 76, 77, 82, 100, 104, 120
nlm, 2, 38, 97
nlme, 2, 38, 97
nls, 2, 38, 97
ode, 3, 17, 28, 42, 49, 56, 62, 66, 72, 77, 82, 85, 88, 94, 100, 104, 106, 109, 120
ode.1D, 3, 17, 42, 49, 56, 62, 68, 69, 77, 82, 85, 87, 94, 100, 104, 120
ode.2D, 3, 17, 42, 49, 56, 63, 68, 72, 75, 82, 85, 87, 94, 100, 104, 120
ode.3D, 3, 17, 42, 49, 56, 63, 68, 72, 77, 80, 85, 94, 100, 104, 120
ode.band, 3, 17, 42, 49, 56, 62, 68, 71, 72, 77, 82, 84, 100, 104, 120
optim, 2, 38, 97
par, 87
plot.1D (plot.deSolve), 86
plot.default, 87
plot.deSolve, 68, 86
print.deSolve, 88
print.deSolve (ode), 66
radau, 3, 17, 68, 90, 100, 108
rk, 3, 17, 42, 49, 55, 62, 68, 96, 104, 106, 109, 120
rk4, 3, 17, 42, 49, 55, 62, 100, 102, 120
rkMethod, 3, 42, 49, 68, 98–100, 104, 105
SCOC, 110
timelags, 15, 40, 47, 54, 61, 112, 118
timestep (timelags), 112
vode, 3, 17, 42, 49, 51, 55, 62, 68, 72, 85, 100, 104, 115, 125, 126
zvode, 122