

# Package ‘deSolve’

November 23, 2009

**Version** 1.5-1

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

**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) and of differential algebraic equations (DAE). 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** 269

**Date/Publication** 2009-11-23 19:19:38

## R topics documented:

deSolve-package	2
aquaphy	4
ccl4data	8
ccl4model	9

daspk . . . . .	11
diagnostics . . . . .	20
diagnostics.deSolve . . . . .	21
DLLfunc . . . . .	22
DLLres . . . . .	24
forcings . . . . .	26
lsoda . . . . .	29
lsodar . . . . .	36
lsode . . . . .	42
lsodes . . . . .	49
ode . . . . .	56
ode.1D . . . . .	60
ode.2D . . . . .	65
ode.3D . . . . .	69
ode.band . . . . .	73
plot.deSolve . . . . .	75
rk . . . . .	77
rk4 . . . . .	82
rkMethod . . . . .	86
SCOC . . . . .	90
vode . . . . .	91
zvode . . . . .	98

**Index****104**


---

deSolve-package	<i>General Solvers for Initial Value Problems of Ordinary Differential Equations (ODE), Partial Differential Equations (PDE) and for Differential Algebraic Equations (DAE)</i>
-----------------	---

---

**Description**

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

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.

**Details**

Package:	deSolve
Type:	Package
Version:	1.5-1
Date:	2009-10-23
License:	GNU Public License 2 or above

The system of ODE's is written as an R function or be 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

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

L. R. Petzold, 1983. A Description of DASSL: A Differential/Algebraic System Solver, in Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, pp. 65-68.

P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, 1989. VODE: A Variable Coefficient ODE Solver, SIAM J. Sci. Stat. Comput., 10, pp. 1038-1051.

### 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  
`lsoda`, `lsode`, `lsodes`, `lsodar`, `vode`, `daspk`, for solvers of the Livermore family  
`rk`, `rk4`, `euler` for Runge-Kutta solvers.  
`DLLfunc`, `DLLres`, for testing model implementations in compiled code.

### Examples

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

## run demos
demo("rk_solvers") # comparison of lsoda with Runge-Kutta-Type Solvers
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  $\text{mmol C m}^{-3}$ . 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  $\text{mmol N m}^{-3}$ .

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**

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

**Author(s)**

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

**References**

Lancelot, C., Veth, C. and Mathot, S. (1991). Modelling ice-edge phytoplankton bloom in the Scotia-Weddell sea sector of the Southern Ocean during spring 1988. *Journal of Marine Systems* 2, 333–346.

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

**See Also**

[ccl4model](#), the CCl4 inhalation model.

**Examples**

```
## =====
##
## 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
##
## =====

times <- seq(0, 24*20, 1)

## -----
## create the forcing functions
## -----

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,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 CCl<sub>4</sub> (carbon tetrachloride) in rats.

## Usage

```
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

Evans, et al. 1994 Applications of sensitivity analysis to a physiologically based pharmacokinetic model for carbon tetrachloride in rats. *Toxicology and Applied Pharmacology* **128**: 36 – 44.

## Examples

```
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

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

## Arguments

<code>times</code>	time sequence for which the model has to be integrated.
<code>y</code>	the initial values for the state variables ("AI", "AAM", "AT", "AF", "AL", "CLT" and "AM"), in that order.
<code>parms</code>	vector or list holding the ccl4 model parameters; see the example for the order in which these have to be defined.
<code>...</code>	any other parameters passed to the integrator <code>ode</code> (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 '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**

```

## =====
## Parameter values
## =====

Pm <- c(
  ## Physiological parameters
  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 = 0.074, # 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)

  ## Chemical specific parameters for chemical
  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
## =====
y <- 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
## =====

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"))

```

---

daspk

*Solver for Differential Algebraic Equations (DAE)*


---

## Description

Solves either:

- a system of ordinary differential equations (ODE) of the form

$$y' = f(t, y, \dots)$$

or

- a system of differential algebraic equations (DAE) of the form

$$F(t, y, y') = 0$$

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", 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, ...)

```

**Arguments**

- |       |   |
|-------|---|
| y     | the initial (state) values for the DE system. If <code>y</code> has a <code>names</code> attribute, the names will be used to label the output matrix.  |
| times | time sequence for which output is wanted; the first value of <code>times</code> must be the initial time; if only one step is to be taken; set <code>times = NULL</code> .  |
| func  | cannot be used if the model is a DAE system. If an ODE system, <code>func</code> should be an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i> ) at time <code>t</code> .<br><code>func</code> must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> .<br><code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> , unless <code>ynames</code> is <code>FALSE</code> . <code>parms</code> is a vector or list of parameters. ... (optional) are any other arguments passed to the function.<br>The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code> , and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the specification of the state variables <code>y</code> .<br>Note that it is not possible to define <code>func</code> as a compiled function in a dynamically loaded shared library. Use <code>res</code> instead. |
| parms | vector or list of parameters used in <code>func</code> , <code>jacfunc</code> , or <code>res</code>   |
| dy    | the initial derivatives of the state variables of the DE system. Ignored if an ODE.   |
| res   | 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 <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library.<br>If <code>res</code> is a user-supplied R-function, it must be defined as: <code>res &lt;- function(t, y, dy, parms, ...)</code> .<br>Here <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system, <code>dy</code> are the corresponding derivatives. If the initial <code>y</code> or <code>dy</code> have a <code>names</code> attribute, the names will be available inside <code>res</code> , unless <code>ynames</code> is <code>FALSE</code> . <code>parms</code> is a vector of parameters.<br>The return value of <code>res</code> should be a list, whose first element is a vector containing the residuals of the DAE system, i.e. $\delta = F(t,y,y')$ , and whose next elements contain output variables that are required at each point in <code>times</code> .  |

	<p>If <code>res</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>daspk()</code> is called (see package vignette "compiledCode" for more information).</p>
<code>nalg</code>	<p>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.</p> <p>Only used if <code>estini = 1</code>.</p>
<code>rtol</code>	relative error tolerance, either a scalar or a vector, one value for each <code>y</code> ,
<code>atol</code>	absolute error tolerance, either a scalar or a vector, one value for each <code>y</code> .
<code>jacfunc</code>	<p>if not <code>NULL</code>, an <code>R</code> function that computes the Jacobian of the system of differential equations. Only used in case the system is an ODE (<math>y' = f(t,y)</math>), specified by <code>func</code>. The <code>R</code> calling sequence for <code>jacfunc</code> is identical to that of <code>func</code>.</p> <p>If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix <code>dydot/dy</code>, where the <i>i</i>th row contains the derivative of <math>dy_i/dt</math> with respect to <math>y_j</math>, or a vector containing the matrix elements by columns (the way <code>R</code> and <code>FORTRAN</code> store matrices).</p> <p>If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code>.</p>
<code>jacres</code>	<p><code>jacres</code> and not <code>jacfunc</code> should be used if the system is specified by the residual function <math>F(t,y,y')</math>, i.e. <code>jacres</code> is used in conjunction with <code>res</code>.</p> <p>If <code>jacres</code> is an <code>R</code>-function, the calling sequence for <code>jacres</code> is identical to that of <code>res</code>, but with extra parameter <code>cj</code>. Thus it should be called as: <code>jacres = func(t, y, dy, parms, cj, ...)</code>. Here <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system, <code>y'</code> are the corresponding derivatives and <code>cj</code> is a scalar, which is normally proportional to the inverse of the stepsize. If the initial <code>y</code> or <code>dy</code> have a <code>names</code> attribute, the names will be available inside <code>jacres</code>, unless <code>ynames</code> is <code>FALSE</code>. <code>parms</code> is a vector of parameters (which may have a <code>names</code> attribute).</p> <p>If the Jacobian is a full matrix, <code>jacres</code> should return the matrix <math>dG/dy + cj*dG/dyprime</math>, where the <i>i</i>th row is the sum of the derivatives of <math>G_i</math> with respect to <math>y_j</math> and the scaled derivatives of <math>G_i</math> with respect to <math>dy_j</math>.</p> <p>If the Jacobian is banded, <code>jacres</code> should return only the nonzero bands of the Jacobian, rotated rowwise. See details for the calling sequence when <code>jacres</code> is a string.</p>
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by the user.
<code>estini</code>	<p>only if a DAE system, and if initial values of <code>y</code> and <code>dy</code> are not consistent (i.e. <math>F(t,y,dy) \neq 0</math>), setting <code>estini = 1</code> or <code>2</code>, will solve for them. If <code>estini = 1</code>: <code>dy</code> and the algebraic variables are estimated from <code>y</code>; in this case, the number of algebraic equations must be given (<code>nalg</code>). If <code>estini = 2</code>: <code>y</code> will be estimated from <code>dy</code>.</p>
<code>verbose</code>	if <code>TRUE</code> : full output to the screen, e.g. will print the <code>diagnostics</code> of the integration - see details.
<code>tcrit</code>	the <code>FORTRAN</code> routine <code>daspk</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver
<code>ynames</code>	logical, if <code>FALSE</code> , names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for large models.
<code>maxord</code>	the maximum order to be allowed. Reduce <code>maxord</code> to save storage space ( $\leq 5$ )
<code>bandup</code>	number of non-zero bands above the diagonal, in case the Jacobian is banded (and <code>jactype</code> one of "bandint", "bandusr")
<code>banddown</code>	number of non-zero bands below the diagonal, in case the Jacobian is banded (and <code>jactype</code> one of "bandint", "bandusr")
<code>maxsteps</code>	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 <code>maxsteps</code> steps.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>res</code> and <code>jacres</code> . See package vignette "compiledCode".
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".
<code>initpar</code>	only when 'dllname' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>res</code> and <code>jacres</code> .
<code>ipar</code>	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>res</code> and <code>jacres</code> .
<code>nout</code>	only used if 'dllname' is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>res</code> , 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".
<code>outnames</code>	only used if 'dllname' is specified and <code>nout</code> $> 0$ : the names of output variables calculated in the compiled function <code>res</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	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 <a href="#">forcings</a> 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 <a href="#">forcings</a> or package vignette "compiledCode".
fcontrol	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette compiledCode.
...	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

$$y' = f(t, y, \dots)$$

for  $y = Y$ , or

- a DAE system of the form

$$F(t, y, y') = 0$$

for  $y = Y$  and  $y' = YPRIME$ .

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 be 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

L. R. Petzold, A Description of DASSL: A Differential/Algebraic System Solver, in *Scientific Computing*, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pp. 65-68.

K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier, New York, 1989.

P. N. Brown and A. C. Hindmarsh, Reduced Storage Matrix Methods in Stiff ODE Systems, *J. Applied Mathematics and Computation*, 31 (1989), pp. 40-91.

P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems, *SIAM J. Sci. Comp.*, 15 (1994), pp. 1467-1488.

P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, Consistent Initial Condition Calculation for Differential-Algebraic Systems, LLNL Report UCRL-JC-122175, August 1995; submitted to *SIAM J. Sci. Comp.*

Netlib: <http://www.netlib.org>

**See Also**

- [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**

```
## =====
## 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))
  })
}

times <- seq(0, 100, by = 2)

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

## ODE model solved with daspk
ODE <- as.data.frame(daspk(y = y, times = times, func = Fun_ODE,
                          parms = pars, atol = 1e-10, rtol = 1e-10))

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

## DAE model solved with daspk
DAE <- as.data.frame(daspk(y = y, dy = dy, times = times,
                          res = Res_DAE, parms = pars, atol = 1e-10, rtol = 1e-10))

## =====
## plotting output
## =====
opa <- par(mfrow = c(2,2))

for (i in 2:5)
{
  plot(ODE$time, ODE[,i], xlab = "time",
       ylab = "conc", main = names(ODE)[i], type = "l")
  points(DAE$time, DAE[,i], col = "red")
}
legend("bottomright", lty = c(1, NA), pch = c(NA, 1),

```

```

col = c("black","red"),legend = c("ODE","DAE"))

# difference between both implementations:
max(abs(ODE-DAE))

par(mfrow = opa)

## =====
## same DAE model, now with the Jacobian
## =====
jacres_DAE <- function (t, y, yprime, pars, cj)
{
  with (as.list(c(y, yprime, pars)), {
##   res1 = -dD - dA + prod
    PD[1,1] <- -1*cj      # d(res1)/d(A)-cj*d(res1)/d(dA)
    PD[1,2] <- 0         # d(res1)/d(B)-cj*d(res1)/d(dB)
    PD[1,3] <- -1*cj      # d(res1)/d(D)-cj*d(res1)/d(dD)
##   res2 = -dB + dA - r*B
    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 = 1e-10, rtol = 1e-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 = 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(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="initforcs", parms=parms, forcings=prod,
  atol=1e-10,rtol=1e-10,nout=2, outnames=c("CONC","Prod")))

opa <- 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)

```

---

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

```

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 equaton 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

```
## 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 <code>diagnostics</code> 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.

## Examples

```
## The famous Lorenz equations: chaos in the earth's atmosphere
## Lorenz 1963. J. Atmos. Sci. 20, 130-141.

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

<code>func</code>	the name of the function in the dynamically loaded shared library,
<code>times</code>	first value = the time at which the function needs to be evaluated,
<code>y</code>	the values of the dependent variables for which the function needs to be evaluated,
<code>parms</code>	the parameters that are passed to the initialiser function,
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains the compiled function or subroutine definitions referred to in <code>func</code> ,
<code>initfunc</code>	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘ <code>dllname</code> ’. See details,
<code>rpar</code>	a vector with double precision values passed to the dll-function <code>func</code> and <code>jacfunc</code> present in the dll, via argument <code>rpar</code> ,
<code>ipar</code>	a vector with integer values passed to the dll-function <code>func</code> and <code>jacfunc</code> present in the dll, via function argument <code>ipar</code> ,
<code>nout</code>	the number of output variables.
<code>outnames</code>	only used if ‘ <code>dllname</code> ’ is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library.
<code>forcings</code>	only used if ‘ <code>dllname</code> ’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval <code>[min(times), max(times)]</code> 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, 0.04, 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 = "derivsccl4",
        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

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

<code>res</code>	the name of the function in the dynamically loaded shared library,
<code>times</code>	first value = the time at which the function needs to be evaluated,
<code>y</code>	the values of the dependent variables for which the function needs to be evaluated,
<code>dy</code>	the derivative of the values of the dependent variables for which the function needs to be evaluated,
<code>parms</code>	the parameters that are passed to the initialiser function,

<code>dllname</code>	a string giving the name of the shared library (without extension) that contains the compiled function or subroutine definitions referred to in <code>func</code> ,
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in <code>'dllname'</code> . See details,
<code>rpar</code>	a vector with double precision values passed to the dll-function <code>func</code> and <code>jacfunc</code> present in the dll, via argument <code>rpar</code> ,
<code>ipar</code>	a vector with integer values passed to the dll-function <code>func</code> and <code>jacfunc</code> present in the dll, via function argument <code>ipar</code> ,
<code>nout</code>	the number of output variables.
<code>outnames</code>	only used if <code>'dllname'</code> is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library.
<code>forcings</code>	only used if <code>'dllname'</code> is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See package vignette "compiledCode".
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in <code>'dllname'</code> . It <b>MUST</b> be present if <code>forcings</code> has been given a value. See package vignette "compiledCode".
<code>fcontrol</code>	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:

<code>res</code>	the residual of derivative estimated by the function
<code>var</code>	the ordinary output variables of the function

### Author(s)

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

### See Also

[daspk](#) to solve DAE problems

## Examples

```
## =====
## 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(nc=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", parms=pars, forcings=prod,
        nout=2, outnames=c("CONC","Prod"))
```

---

forcings

*Passing forcing functions to models written in R or compiled code.*

---

## Description

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

If a model uses "external variables" (or "forcing variables"), their value at each time point needs to be estimated by interpolation of a data series.

If the models are defined in R-code, then forcing function interpolation is most efficiently done using R's function `approxfun`.

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

## Details

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`.
2. call this function within the model's derivative function, to interpolate at the current timestep.

See first example

If the model is defined in *compiled code*, the forcing function data series is provided by means of argument `forcings`, `initforc` is the name of the forcing function initialisation function, as

provided in `'dllname'`, while `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  $[\min(\text{times}), \max(\text{times})]$ . 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  $y_0 * (1-f) + y_1 * f$  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:

```
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

**Examples**

```

## =====
# The sediment oxygen consumption example - R-code:
## =====

# 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))

parms <- c(k=0.01)

times <- 1:365

# the model
sediment <- function( t, O2, k)
  list( c(Depo(t) - k*O2), depo = Depo(t))

# the forcing functions
Depo <- approxfun(x=Flux[,1], y= Flux[,2], method="linear")

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

# same forcing functions, now constant interpolation
Depo <- approxfun(x=Flux[,1], y= Flux[,2], method="constant", f=0.5)

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

mf <- par(mfrow=c(2,1))
plot (Out, which = "depo", type="l", lwd=2, mfrow=NULL)
lines(Out2[, "time"], Out2[, "depo"], col="red", lwd=2)

plot (Out, which = "O2", type="l", lwd=2, mfrow=NULL)
lines(Out2[, "time"], Out2[, "O2"], col="red", lwd=2)

## =====
## SCOC is the same model, as implemented in FORTRAN
## =====

out<- SCOC(times,parms=parms,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=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

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

---

## Description

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,
      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, ...)

```

### Arguments

<code>y</code>	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.
<code>times</code>	times at which explicit estimates for <code>y</code> are desired. The first value in <code>times</code> must be the initial time.
<code>func</code>	<p>either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i>) at time <code>t</code>, or a character string giving the name of a compiled function in a dynamically loaded shared library.</p> <p>If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code>. <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code>. <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function.</p> <p>The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to time, and whose next elements are global values that are required at each point in <code>times</code>. The derivatives should be specified in the same order as the state variables <code>y</code>.</p> <p>If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>lsoda()</code> is called. See package vignette "compiledCode" for more details.</p>
<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacfunc</code>	<p>if not <code>NULL</code>, an R function, that computes the Jacobian of the system of differential equations <math>dy_i/dy_j</math>, 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).</p> <p>In some circumstances, supplying <code>jacfunc</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jacfunc</code> is identical to that of <code>func</code>.</p> <p>If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix <math>dy_i/dy_j</math>, where the <math>i</math>th row contains the derivative of <math>dy_i/dt</math> with respect to <math>y_j</math>, or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).</p> <p>If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code>.</p>
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user.
<code>verbose</code>	a logical value that, when <code>TRUE</code> , will print the diagnostics of the integration - see details.

<code>tcrit</code>	if not <code>NULL</code> , then <code>lsoda</code> cannot integrate past <code>tcrit</code> . The FORTRAN routine <code>lsoda</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver.
<code>ynames</code>	logical, if <code>FALSE</code> : names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for large models.
<code>maxordn</code>	the maximum order to be allowed in case the method is non-stiff. Should be $\leq 12$ . Reduce <code>maxord</code> to save storage space.
<code>maxords</code>	the maximum order to be allowed in case the method is stiff. Should be $\leq 5$ . Reduce <code>maxord</code> to save storage space.
<code>bandup</code>	number of non-zero bands above the diagonal, in case the Jacobian is banded.
<code>banddown</code>	number of non-zero bands below the diagonal, in case the Jacobian is banded.
<code>maxsteps</code>	maximal number of steps per output interval taken by the solver.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette " <code>compiledCode</code> ".
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in ' <code>dllname</code> '. See package vignette " <code>compiledCode</code> ".
<code>initpar</code>	only when ' <code>dllname</code> ' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when ' <code>dllname</code> ' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>ipar</code>	only when ' <code>dllname</code> ' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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 " <code>compiledCode</code> ".
<code>outnames</code>	only used if ' <code>dllname</code> ' is specified and <code>nout</code> $> 0$ : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.

<code>forcings</code>	only used if <code>'dllname'</code> 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 <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>initforc</code>	if not NULL, the name of the forcing function initialisation function, as provided in <code>'dllname'</code> . It MUST be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>fcontrol</code>	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> 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/opkdomain.f'`). The implementation is based on the 12 November 2003 version of Isoda, 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 `jacstype` 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 `jacstype = "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  $\max\text{-norm of } (\mathbf{e}/\mathbf{ewt}) \leq 1$ , where **ewt** is a vector of positive error weights. The values of `rtol` and `atol` should all be non-negative. The form of **ewt** is:

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

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 be 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 an additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine 'Isoda' 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

Hindmarsh, Alan C. (1983) ODEPACK, A Systematized Collection of ODE Solvers; in p.55–64 of Stepleman, R.W. et al.[ed.] (1983) *Scientific Computing*, North-Holland, Amsterdam.

Petzold, Linda R. (1983) Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. *Siam J. Sci. Stat. Comput.* **4**, 136–148.

Netlib: <http://www.netlib.org>

### See Also

- [rk](#),
  - [rk4](#) and [euler](#) for Runge-Kutta integrators.
  - [lsode](#), [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

```
## =====
## Example 1:
##   A simple resource limited Lotka-Volterra-Model
##
## Note:
## 1. parameter and state variable names made
##    accessible via "with" function
## 2. function sigimp accessible through lexical scoping
##    (see also ode and rk examples)
## =====

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)
  })
}

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

## vector of timesteps
times <- seq(0, 100, 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
y <- xstart <- c(S = 1, P = 1, C = 1)

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

## Plotting
mf <- par(mfrow = c(2,2))
plot(out$time, out$S, type = "l", ylab = "substrate")
plot(out$time, out$P, type = "l", ylab = "producer")
plot(out$time, out$C, type = "l", ylab = "consumer")
plot(out$P, out$C, type = "l", xlab = "producer", ylab = "consumer")
par(mfrow = mf)
```

```

## =====
## Example 2:
## from lsoda source code
## =====

## names makes this easier to read, but may slow down execution.
parms <- c(k1 = 0.04, k2 = 1e4, k3 = 3e7)
my.atol <- c(1e-6, 1e-10, 1e-6)
times <- c(0, 4 * 10^(-1:10))

lsexamp <- function(t, y, p)
{
  yd1 <- -p["k1"] * y[1] + p["k2"] * y[2]*y[3]
  yd3 <- p["k3"] * y[2]^2
  list(c(yd1, -yd1-yd3, yd3), c(massbalance = sum(y)))
}

exampjac <- function(t, y, p)
{
  matrix(c(-p["k1"], p["k1"], 0,
          p["k2"]*y[3],
          - p["k2"]*y[3] - 2*p["k3"]*y[2],
          2*p["k3"]*y[2],
          p["k2"]*y[2], -p["k2"]*y[2], 0
          ), 3, 3)
}

## measure speed (here and below)
system.time(
  out <- lsoda(c(1, 0, 0), times, lsexamp, parms, rtol = 1e-4,
              atol = my.atol, hmax = Inf)
)
out

## This is what the authors of lsoda got for the example:

## the output of this program (on a cdc-7600 in single precision)
## is as follows..
##
## at t = 4.0000e-01 y = 9.851712e-01 3.386380e-05 1.479493e-02
## at t = 4.0000e+00 y = 9.055333e-01 2.240655e-05 9.444430e-02
## at t = 4.0000e+01 y = 7.158403e-01 9.186334e-06 2.841505e-01
## at t = 4.0000e+02 y = 4.505250e-01 3.222964e-06 5.494717e-01
## at t = 4.0000e+03 y = 1.831975e-01 8.941774e-07 8.168016e-01
## at t = 4.0000e+04 y = 3.898730e-02 1.621940e-07 9.610125e-01
## at t = 4.0000e+05 y = 4.936363e-03 1.984221e-08 9.950636e-01
## at t = 4.0000e+06 y = 5.161831e-04 2.065786e-09 9.994838e-01
## at t = 4.0000e+07 y = 5.179817e-05 2.072032e-10 9.999482e-01
## at t = 4.0000e+08 y = 5.283401e-06 2.113371e-11 9.999947e-01
## at t = 4.0000e+09 y = 4.659031e-07 1.863613e-12 9.999995e-01

```

```
## at t = 4.0000e+10   y = 1.404280e-08  5.617126e-14  1.000000e+00

## Using the analytic Jacobian speeds up execution a little :

system.time(
  outJ <- lsoda(c(1, 0, 0), times, lsexamp, parms, rtol = 1e-4,
               atol = my.atol, jacfunc = exampjac, jactype = "fullusr", hmax = Inf)
)

all.equal(as.data.frame(out), as.data.frame(outJ)) # TRUE
diagnostics(out)
diagnostics(outJ) # shows what lsoda did internally
```

---

 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.

## Usage

```
lsodar(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
       jacfunc = NULL, jactype = "fullint", rootfunc = NULL,
       verbose = FALSE, nroot = 0, tcrit = NULL, hmin = 0,
       hmax = NULL, hini = 0, ynames = TRUE, 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, ...)
```

**Arguments**

<code>y</code>	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.
<code>times</code>	times at which explicit estimates for <code>y</code> are desired. The first value in <code>times</code> must be the initial time.
<code>func</code>	<p>either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i>) at time <code>t</code>, or a character string giving the name of a compiled function in a dynamically loaded shared library.</p> <p>If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code>. <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code>. <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function.</p> <p>The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code>, and whose next elements are global values that are required at each point in <code>times</code>. The derivatives should be specified in the same order as the state variables <code>y</code>.</p> <p>If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>lsodar()</code> is called. See package vignette "compiledCode" for more details.</p>
<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacfunc</code>	<p>if not NULL, an R function, that computes the Jacobian of the system of differential equations <math>dy(i)/dy(j)</math>, 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).</p> <p>In some circumstances, supplying <code>jacfunc</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jacfunc</code> is identical to that of <code>func</code>.</p> <p>If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix <math>dydot/dy</math>, where the <math>i</math>th row contains the derivative of <math>dy_i/dt</math> with respect to <math>y_j</math>, or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).</p> <p>If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code>.</p>
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user.
<code>rootfunc</code>	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 <code>rootfunc</code> is identical to that of <code>func</code> . <code>rootfunc</code> should return a vector with the function values whose root is sought.
<code>verbose</code>	a logical value that, when TRUE, will print the <code>diagnostics</code> 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 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".

<code>outnames</code>	only used if <code>'dllname'</code> is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	only used if <code>'dllname'</code> is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in <code>'dllname'</code> . It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>fcontrol</code>	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> 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/opkmain.f'`). The implementation is based on the November, 2003 version of `lsodar`, 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 to 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 `lsodar`, the default,

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

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

**`jactype = "bandint"`**: banded Jacobian, calculated by `lsodar`; 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 be 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 an additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine 'Isoda' 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

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

Linda R. Petzold, Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, Siam J. Sci. Stat. Comput. 4 (1983), pp. 136-148.

Kathie L. Hiebert and Lawrence F. Shampine, Implicitly Defined Output Points for Solutions of ODEs, Sandia Report SAND80-0180, February 1980.

Netlib: <http://www.netlib.org>

### See Also

- [rk](#),
- [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**

```

## =====
## Example 1:
##   from lsodar source code
## =====

Fun <- function (t, y, parms)
{
  ydot <- vector(len = 3)
  ydot[1] <- -.04*y[1] + 1.e4*y[2]*y[3]
  ydot[3] <- 3.e7*y[2]*y[2]
  ydot[2] <- -ydot[1]-ydot[3]
  return(list(ydot,ytot = sum(y)))
}

rootFun <- function (t, y, parms)
{
  yroot <- vector(len = 2)
  yroot[1] <- y[1] - 1.e-4
  yroot[2] <- y[3] - 1.e-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)

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

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

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

```

```

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

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

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)

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

```

---

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:

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

.

The R function `lsode` 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.

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

`lsode` is very similar to `vode`, but uses a fixed-step-interpolate method rather than the variable-coefficient method in `vode`. 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 `lsode`, a copy is not kept.

## Usage

```

lsode(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
      jacfunc = NULL, jactype = "fullint", mf = NULL,
      verbose = FALSE, tcrit = NULL, hmin = 0, hmax = NULL, hini = 0,
      ynames = TRUE, maxord = NULL, bandup = NULL, banddown = NULL,
      maxsteps = 5000, dllname = NULL, initfunc = dllname,
      initpar = parms, rpar = NULL, ipar = NULL, nout = 0,
      outnames = NULL, forcings=NULL,
      initforc = NULL, fcontrol=NULL, ...)

```

**Arguments**

<code>y</code>	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.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time; if only one step is to be taken; set <code>times = NULL</code> .
<code>func</code>	<p>either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i>) at time <code>t</code>, or a character string giving the name of a compiled function in a dynamically loaded shared library.</p> <p>If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code>. <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code>. <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function.</p> <p>The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code>, and whose next elements are global values that are required at each point in <code>times</code>. The derivatives should be specified in the same order as the state variables <code>y</code>.</p> <p>If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>lsode()</code> is called. See package vignette "compiledCode" for more details.</p>
<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacfunc</code>	<p>if not <code>NULL</code>, an R function that computes the Jacobian of the system of differential equations <math>dy(i)/dy(j)</math>, 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).</p> <p>In some circumstances, supplying <code>jacfunc</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jacfunc</code> is identical to that of <code>func</code>.</p> <p>If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix <math>dydot/dy</math>, where the <math>i</math>th row contains the derivative of <math>dy_i/dt</math> with respect to <math>y_j</math>, or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).</p> <p>If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code>.</p>
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if <code>mfis</code> not <code>NULL</code> .
<code>mf</code>	the "method flag" passed to function <code>lsode</code> - overrules <code>jactype</code> - provides more options than <code>jactype</code> - see details.
<code>verbose</code>	if <code>TRUE</code> : full output to the screen, e.g. will print the <code>diagnostics</code> of the integration - see details.

<code>tcrit</code>	if not <code>NULL</code> , then <code>lsode</code> cannot integrate past <code>tcrit</code> . The FORTRAN routine <code>lsode</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver.
<code>yname</code> s	logical, if <code>FALSE</code> names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for multi-D models.
<code>maxord</code>	the maximum order to be allowed. <code>NULL</code> uses the default, i.e. order 12 if implicit Adams method ( <code>meth = 1</code> ), order 5 if BDF method ( <code>meth = 2</code> ). Reduce <code>maxord</code> to save storage space.
<code>bandup</code>	number of non-zero bands above the diagonal, in case the Jacobian is banded.
<code>banddown</code>	number of non-zero bands below the diagonal, in case the Jacobian is banded.
<code>maxsteps</code>	maximal number of steps per output interval taken by the solver.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette " <code>compiledCode</code> ".
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in ' <code>dllname</code> '. See package vignette " <code>compiledCode</code> ".
<code>initpar</code>	only when ' <code>dllname</code> ' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when ' <code>dllname</code> ' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>ipar</code>	only when ' <code>dllname</code> ' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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 " <code>compiledCode</code> ".
<code>outnames</code>	only used if ' <code>dllname</code> ' is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	only used if ' <code>dllname</code> ' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the

	interval [ <code>min(times)</code> , <code>max(times)</code> ] is done by taking the value at the closest data extreme.
	See <a href="#">forcings</a> or package vignette " <code>compiledCode</code> ".
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in ' <code>dllname</code> '. It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette " <code>compiledCode</code> ".
<code>fcontrol</code>	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> 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 `lsode`, 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 `jacstype` or `mf`.

The options for **`jacstype`** are

**`jacstype = "fullint"`** a full Jacobian, calculated internally by `lsode`, corresponds to `mf = 22`,

**`jacstype = "fullusr"`** a full Jacobian, specified by user function `jacfunc`, corresponds to `mf = 21`,

**`jacstype = "bandusr"`** a banded Jacobian, specified by user function `jacfunc`; the size of the bands specified by `bandup` and `banddown`, corresponds to `mf = 24`,

**`jacstype = "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 be 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 an 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

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

### See Also

- [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**

```

## =====
## Example 1:
##   Various ways to solve the same model.
## =====

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

  ydot[1] <- 0.1*y[1] -0.2*y[2]
  ydot[2] <- -0.3*y[1] +0.1*y[2] -0.2*y[3]
  ydot[3] <-          -0.3*y[2] +0.1*y[3] -0.2*y[4]
  ydot[4] <-          -0.3*y[3] +0.1*y[4] -0.2*y[5]
  ydot[5] <-          -0.3*y[4] +0.1*y[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 <- lsode(yini, times, f1, parms = 0, jactype = "fullint")

## stiff method, user-generated full Jacobian
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
out3 <- lsode(yini, times, f1, parms = 0, jactype = "bandint",
              bandup = 1, banddown = 1)

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

## non-stiff method
out5 <- lsode(yini, times, f1, parms = 0, mf = 10)

## =====
## Example 2:
## diffusion on a 2-D grid
## partially specified Jacobian
## =====

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
dy <- dx <- 1 # grid size
Dy <- Dx <- 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.)

pa <- par(ask = FALSE)

## initial condition
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)
  }
}

```

```

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

## =====
## Example 3:
##   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.
## =====

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)

```

---

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:

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

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

```

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, ...)

```

**Arguments**

<code>y</code>	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.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time; if only one step is to be taken; set <code>times = NULL</code> .
<code>func</code>	<p>either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i>) at time <code>t</code>, or a character string giving the name of a compiled function in a dynamically loaded shared library.</p> <p>If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code>. <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code>. <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function.</p> <p>The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code>, and whose next elements are global values that are required at each point in <code>times</code>. The derivatives should be specified in the same order as the state variables <code>y</code>.</p> <p>If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>lsodes()</code> is called. See package vignette "compiledCode" for more details.</p>
<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacvec</code>	<p>if not <code>NULL</code>, an R function that computes a column of the Jacobian of the system of differential equations <math>dy(i)/dy(j)</math>, 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).</p> <p>The R calling sequence for <code>jacvec</code> is identical to that of <code>func</code>, but with extra parameter <code>j</code>, denoting the column number. Thus, <code>jacvec</code> should be called as: <code>jacvec = func(t, y, j, parms)</code> and <code>jacvec</code> should return a vector containing column <code>j</code> of the Jacobian, i.e. its <code>i</code>-th value is <math>dy(i)/dy(j)</math>. If this function is absent, <code>lsodes</code> will generate the Jacobian by differences.</p>
<code>sparsetype</code>	the sparsity structure of the Jacobian, one of "sparseint" or "sparseusr", sparsity estimated internally by <code>lsodes</code> or given by user.
<code>nnz</code>	the number of nonzero elements in the sparse Jacobian (if this is unknown, use an estimate).
<code>inz</code>	(row,column) indices to the nonzero elements in the sparse Jacobian. Necessary if <code>sparsetype = "sparseusr"</code> ; else ignored.
<code>verbose</code>	if <code>TRUE</code> : full output to the screen, e.g. will print the diagnostics of the integration - see details.
<code>tcrit</code>	if not <code>NULL</code> , then <code>lsodes</code> cannot integrate past <code>tcrit</code> . The FORTRAN routine <code>lsodes</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .

<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver.
<code>ynames</code>	logical, if <code>FALSE</code> names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for multi-D models.
<code>maxord</code>	the maximum order to be allowed. <code>NULL</code> uses the default, i.e. order 12 if implicit Adams method ( <code>meth = 1</code> ), order 5 if BDF method ( <code>meth = 2</code> ). Reduce <code>maxord</code> to save storage space.
<code>maxsteps</code>	maximal number of steps per output interval taken by the solver.
<code>lrw</code>	the length of the real work array <code>rwork</code> ; due to the sparsicity, this cannot be readily predicted. If <code>NULL</code> , a guess will be made, and if not sufficient, <code>lsodes</code> will return with a message indicating the size of <code>rwork</code> actually required. Therefore, some experimentation may be necessary to estimate the value of <code>lrw</code> .
<code>liw</code>	the length of the integer work array <code>iwork</code> ; due to the sparsicity, this cannot be readily predicted. If <code>NULL</code> , a guess will be made, and if not sufficient, <code>lsodes</code> will return with a message indicating the size of <code>iwork</code> actually required. Therefore, some experimentation may be necessary to estimate the value of <code>liw</code> .
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette " <code>compiledCode</code> ".
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in ' <code>dllname</code> '. See package vignette " <code>compiledCode</code> ".
<code>initpar</code>	only when ' <code>dllname</code> ' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when ' <code>dllname</code> ' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>ipar</code>	only when ' <code>dllname</code> ' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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 " <code>compiledCode</code> ".
<code>outnames</code>	only used if ' <code>dllname</code> ' is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	only used if ' <code>dllname</code> ' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the

	interval <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme.
	See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in <code>'dllname'</code> . It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>fcontrol</code>	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> allowing this to be a generic function.

## Details

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

`lsodes` is applied for stiff problems, where the Jacobian has a sparse structure.

There are four choices depending on whether `jacvec` and `inz` is specified.

If function `jacvec` is present, then it should return the *j*-th column of the Jacobian matrix.

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

If `jacvec` and `inz` are present, then the Jacobian is fully specified by the user.

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

If `nnz` is present, but not `jacvec` then the Jacobian will be estimated internally, by differences.

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

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

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 be 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 an 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**

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

S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, Yale Sparse Matrix Package: I. The Symmetric Codes, *Int. J. Num. Meth. Eng.*, 18 (1982), pp. 1145-1151.

S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, Yale Sparse Matrix Package: II. The Nonsymmetric Codes, Research Report No. 114, Dept. of Computer Sciences, Yale University, 1977.

**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**

```
## Various ways to solve the same model.

## =====
## The example from lsodes source code
## A chemical model
## =====

n <- 12
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,parms)
{
  with (as.list(parms), {
    dy[1] <- -rk1 *Y[1]
    dy[2] <- rk1 *Y[1] + rk11*rk14*Y[4] + rk19*rk14*Y[5] -
             rk3 *Y[2]*Y[3] - rk15*Y[2]*Y[12] - rk2*Y[2]
    dy[3] <- rk2 *Y[2] - rk5 *Y[3] - rk3*Y[2]*Y[3] -
             rk7*Y[10]*Y[3] + rk11*rk14*Y[4] + rk12*rk14*Y[6]
    dy[4] <- rk3 *Y[2]*Y[3] - rk11*rk14*Y[4] - rk4*Y[4]
    dy[5] <- rk15*Y[2]*Y[12] - rk19*rk14*Y[5] - rk16*Y[5]
    dy[6] <- rk7 *Y[10]*Y[3] - rk12*rk14*Y[6] - rk8*Y[6]
    dy[7] <- rk17*Y[10]*Y[12] - rk20*rk14*Y[7] - rk18*Y[7]
    dy[8] <- rk9 *Y[10] - rk13*rk14*Y[8] - rk10*Y[8]
    dy[9] <- rk4 *Y[4] + rk16*Y[5] + rk8*Y[6] +
             rk18*Y[7]
    dy[10] <- rk5 *Y[3] + rk12*rk14*Y[6] + rk20*rk14*Y[7] +
             rk13*rk14*Y[8] - rk7 *Y[10]*Y[3] - rk17*Y[10]*Y[12] -
             rk6 *Y[10] - rk9*Y[10]
    dy[11] <- rk10*Y[8]
    dy[12] <- rk6 *Y[10] + rk19*rk14*Y[5] + rk20*rk14*Y[7] -
             rk15*Y[2]*Y[12] - rk17*Y[10]*Y[12]
    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 spl on rate of change of others
  2, 2, 3, 2, 4, 2, 5, 2, 12, 2,

```

```

2, 3, 3, 3, 4, 3, 6, 3, 10, 3,
2, 4, 3, 4, 4, 4, 9, 4, # d (dyi)/dy4
2, 5, 5, 5, 9, 5, 12, 5,
3, 6, 6, 6, 9, 6, 10, 6,
7, 7, 9, 7, 10, 7, 12, 7,
8, 8, 10, 8, 11, 8,
3,10, 6,10, 7,10, 10,10, 12,10,
2,12, 5,12, 7,12, 10,12, 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] <- -rk1
      PDJ[2] <- rk1
    } else if (j == 2) {
      PDJ[2] <- -rk3*Y[3] - rk15*Y[12] - rk2
      PDJ[3] <- rk2 - rk3*Y[3]
      PDJ[4] <- rk3*Y[3]
      PDJ[5] <- rk15*Y[12]
      PDJ[12] <- -rk15*Y[12]
    } else if (j == 3) {
      PDJ[2] <- -rk3*Y[2]
      PDJ[3] <- -rk5 - rk3*Y[2] - rk7*Y[10]
      PDJ[4] <- rk3*Y[2]
      PDJ[6] <- rk7*Y[10]
      PDJ[10] <- rk5 - rk7*Y[10]
    } else if (j == 4) {
      PDJ[2] <- rk11*rk14
      PDJ[3] <- rk11*rk14
      PDJ[4] <- -rk11*rk14 - rk4
      PDJ[9] <- rk4
    } else if (j == 5) {
      PDJ[2] <- rk19*rk14
      PDJ[5] <- -rk19*rk14 - rk16
      PDJ[9] <- rk16
      PDJ[12] <- rk19*rk14
    } 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[10] <- rk20*rk14
    PDJ[12] <- rk20*rk14
  } else if (j == 8) {
    PDJ[8] <- -rk13*rk14 - rk10
    PDJ[10] <- rk13*rk14
    PDJ[11] <- rk10
  } else if (j == 10) {
    PDJ[3] <- -rk7*Y[3]
    PDJ[6] <- rk7*Y[3]
    PDJ[7] <- rk17*Y[12]
    PDJ[8] <- rk9
    PDJ[10] <- -rk7*Y[3] - rk17*Y[12] - rk6 - rk9
    PDJ[12] <- rk6 - rk17*Y[12]
  } else if (j == 12) {
    PDJ[2] <- -rk15*Y[2]
    PDJ[5] <- rk15*Y[2]
    PDJ[7] <- rk17*Y[10]
    PDJ[10] <- -rk17*Y[10]
    PDJ[12] <- -rk15*Y[2] - rk17*Y[10]
  }
  return(PDJ)
})
}

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

## =====
## application 4. The structure of the Jacobian (nonzero elements) AND
##               the Jacobian (vector) function is input
##               not very efficient...
## =====
out4 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
               lrw = 351, sparsetype = "sparseusr", inz = nonzero,
               jacvec = chemjac, atol = atol, rtol = rtol,
               verbose = TRUE)

```

---

ode

---

*General Solver for Ordinary Differential Equations*


---

### 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"), ...)

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

**Arguments**

<code>y</code>	the initial (state) values for the ODE system, a vector. If <code>y</code> has a <code>name</code> attribute, the names will be used to label the output matrix.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code> , and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> . If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>ode</code> is called. See package vignette "compiledCode" for more details.
<code>parms</code>	parameters passed to <code>func</code> .
<code>method</code>	the integrator to use, either a string ("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "euler", "rk4", "ode23" or "ode45") or a function that performs integration, or a list of class <code>rkMethod</code> .
<code>x</code>	an object of class <code>deSolve</code> , as returned by the integrators, and to be printed.
<code>...</code>	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.

**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

- [plot.deSolve](#) for plotting the outputs,
- [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,
- [aquaphy](#), [ccl4model](#), where `ode` is used,
- [lsoda](#), [lsode](#), [lsodes](#), [lsodar](#), [vode](#), [daspk](#),
- [rk](#), [rkMethod](#)

[diagnostics](#) to print diagnostic messages.

### Examples

```
## =====
## Example1: Predator-Prey Lotka-Volterra model
## =====

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)))
  })
}

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
## =====

## Note:
## Function sigimp passed as an argument (input) to model
## (see also lsoda and rk examples)

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)
  })
}

## 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, dims = NULL,
       method = "lsode", ...)

```

**Arguments**

<code>y</code>	the initial (state) values for the ODE system, a vector. If <code>y</code> has a name attribute, the names will be used to label the output matrix.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to time, and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> . If <code>func</code> is a character string then integrator <code>lsodes</code> will be used. See details.
<code>parms</code>	parameters passed to <code>func</code> .
<code>nspec</code>	the number of <i>*species*</i> (components) in the model. If <code>NULL</code> , then <code>dims</code> should be specified.
<code>dims</code>	the number of <i>*boxes*</i> in the model. If <code>NULL</code> , then <code>nspec</code> should be specified.
<code>method</code>	the integrator to use, one of "vode", "lsode", "lsoda", "lsodar", "lsodes".
<code>...</code>	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:

A[1], A[2], A[3],.... B[1], B[2], B[3],.... (for species A, B))

Two methods are implemented.

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

Then 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.

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 is `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**

```
## =====
## 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
## =====

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

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)))
  })
}

## =====
```

```

## 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)
state[1] <- state[N+1] <- 10

## 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, 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, 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")

## =====

```

```

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

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

  ## rate of change = flux gradient - consumption + reaeration (O2)
  dBOD <- -diff(FluxBOD)/dx - BODrate
  dO2 <- -diff(FluxO2)/dx - BODrate + p*(O2sat-O2)

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

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

## initial conditions:
state <- c(rep(200, N), rep(200, N))
times <- seq(0, 20, by = 1)

## running the model
## step 1 : model spinup
out <- ode.1D(y = state, times, O2BOD, parms = NULL, nspec = 2)

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

```

```
## select oxygen (first column of out:time, then BOD, then O2
O2  <- out[, (N+2):(2*N+1)]
color = topo.colors

filled.contour(x = times, y = x, O2, color = color, nlevels = 50,
              xlab = "time, days", ylab = "Distance from river, m",
              main = "Oxygen")
```

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, dims,
       cyclicBnd = NULL, ...)
```

**Arguments**

<code>y</code>	the initial (state) values for the ODE system, a vector. If <code>y</code> has a name attribute, the names will be used to label the output matrix.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code> , and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> .
<code>parms</code>	parameters passed to <code>func</code> .
<code>nspec</code>	the number of <b>species</b> (components) in the model.
<code>dims</code>	2-valued vector with the number of <b>boxes</b> in two dimensions in the model.
<code>cyclicBnd</code>	if not <code>NULL</code> 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.
<code>...</code>	additional arguments passed to <code>lsodes</code> .

## Details

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

Based on the dimension of the problem, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is 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.

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.

## 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 is `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 (as `nspec * dimens[1] * dimens[2] == length(y)`).

Do **not** use this method for problems that are not 2D!

## Author(s)

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

## See Also

- [ode](#) for a general interface to most of the ODE solvers,
- [ode.band](#) for integrating models with a banded Jacobian

- [ode.1D](#) for integrating 1-D models
- [ode.3D](#) for integrating 3-D models
- [lsodes](#) for the integration options.

[diagnostics](#) to print diagnostic messages.

## Examples

```
## =====
## A Lotka-Volterra predator-prey model with predator and prey
## dispersing in 2 dimensions
## =====

## =====
## Model definitions
## =====

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
    dPrey <- rGrow* Prey *(1- Prey/K) - rIng* Prey *Pred
    dPred <- rIng* Prey *Pred*assEff -rMort* Pred

    zero <- rep(0,N)

    ## 1. Fluxes in x-direction; zero fluxes near boundaries
    FluxPrey <- -Da * rbind(zero, (Prey[2:N,]-Prey[1:(N-1),]), zero)/dx
    FluxPred <- -Da * rbind(zero, (Pred[2:N,]-Pred[1:(N-1),]), zero)/dx

    ## Add flux gradient to rate of change
    dPrey <- dPrey - (FluxPrey[2:(N+1),]-FluxPrey[1:N,])/dx
    dPred <- dPred - (FluxPred[2:(N+1),]-FluxPred[1:N,])/dx

    ## 2. Fluxes in y-direction; zero fluxes near boundaries
    FluxPrey <- -Da * cbind(zero, (Prey[,2:N]-Prey[,1:(N-1)]), zero)/dx
    FluxPred <- -Da * cbind(zero, (Pred[,2:N]-Pred[,1:(N-1)]), zero)/dx

    ## Add flux gradient to rate of change
    dPrey <- dPrey - (FluxPrey[,2:(N+1)]-FluxPrey[,1:N])/dx
    dPred <- dPred - (FluxPred[,2:(N+1)]-FluxPred[,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...)
times   <- seq(0, 50, by = 1)
out <- ode.2D(y = yini, times = times, func = lvmod2D, parms = pars,
             dims = c(N, N), N = N, dx = dx, Da = Da, lrw = 5000000)

## plot results
Col <- colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan",
                          "#7FFF7F", "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 = "x", ylab = "y", zlim = range(out[,2:(NN+1)]))

## 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(nr=nx,nc=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
y    <- matrix(nr=nx,nc=ny,1.)
print(system.time(
ST3 <- ode.2D(y, times=1:100, func=diffusion2D, parms=NULL,
             dims=c(nx,ny), verbose=TRUE,
             lrw=400000, atol=1e-10, rtol=1e-10, cyclicBnd=2)
))

## Not run:
zlim <- range(ST3[,-1])
for (i in 2:nrow(ST3)) {

y <- matrix(nr=nx,nc=ny,data=ST3[i,-1])
filled.contour(y,zlim=zlim,main=i)
}

## End(Not run)

```

### 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, dims,
      ...)
```

**Arguments**

<code>y</code>	the initial (state) values for the ODE system, a vector. If <code>y</code> has a <code>name</code> attribute, the names will be used to label the output matrix.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code> , and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> .
<code>parms</code>	parameters passed to <code>func</code> .
<code>nspec</code>	the number of <b>species</b> (components) in the model.
<code>dims</code>	3-valued vector with the number of <b>boxes</b> in three dimensions in the model.
<code>...</code>	additional arguments passed to <code>lsodes</code> .

**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.

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 `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 is `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 (as `nspec*dimens[1]*dimens[2]*dimens[3] == length(y)`).

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

### 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.1D](#) for integrating 1-D models
- [ode.2D](#) for integrating 2-D models
- [lsodes](#) for the integration options.

[diagnostics](#) to print diagnostic messages.

### Examples

```
## =====
## 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(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
dy    <- dx <- dz <-1   # grid size
Dy    <- Dx <- Dz <-1   # diffusion coeff, X- and Y-direction
r     <- 0.025         # consumption rate

n     <- 10
y     <- array(dim=c(n,n,n),data=10.)

print(system.time(
RES <- ode.3D(y, func=diffusion3D, parms=NULL, dims=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 2: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	<i>Solver for Ordinary Differential Equations; Assumes a Banded Jacobian</i>
----------	--

---

## 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

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	<p>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.</p> <p>If func is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code>. 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.</p> <p>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.</p>
parms	parameters passed to func.
nspec	the number of *species* (components) in the model.
bandup	the number of nonzero bands above the Jacobian diagonal.
banddown	the number of nonzero bands below the Jacobian diagonal.
method	the integrator to use, one of "vode", "lsode", "lsoda", "lsodar".
...	additional arguments passed to the integrator.

## 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

```
## =====
## The Aphid model from Soetaert and Herman, 20098.
## 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) # 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")

```

**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, ...)
```

**Arguments**

<code>x</code>	an object of class <code>deSolve</code> , as returned by the integrators, and to be plotted.
<code>which</code>	the name(s) or the index to the variables that should be plotted. Default = all variables.
<code>ask</code>	logical; if <code>TRUE</code> , the user is <i>asked</i> before each plot, if <code>NULL</code> the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see <code>par(ask=.)</code> and <code>dev.interactive</code> .
<code>...</code>	additional graphics arguments passed to <code>plot.default</code> or <code>hist</code>

**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`.

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.

**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(preyl = 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")

hist(out, col="darkblue", breaks = 50)
```

rk

*Explicit One-Step Solvers for Ordinary Differential Equations (ODE)***Description**

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

The R function `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 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`

**Usage**

```
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, ...)
```

**Arguments**

<code>y</code>	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.
<code>times</code>	times at which explicit estimates for <code>y</code> are desired. The first value in <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i> ) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> 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 `rk` is called. See package vignette "compiledCode" for more details.

<code>parms</code>	vector or list of parameters used in <code>func</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . Only applicable to methods with variable time step, see details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . Only applicable to methods with variable time step, see details.
<code>tcrit</code>	if not <code>NULL</code> , then <code>rk</code> cannot integrate past <code>tcrit</code> . The solver routines may overshoot their targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>verbose</code>	a logical value that, when <code>TRUE</code> , triggers more verbose output from the ODE solver.
<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the maximum of <code>hini</code> and the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified. Note that <code>hmin</code> and <code>hmax</code> are ignored by fixed step methods like "rk4" or "euler".
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined automatically by solvers with flexible time step. Setting <code>hini = 0</code> for fixed step methods forces setting of internal time steps identically to external time steps provided by <code>times</code> .
<code>ynames</code>	if <code>FALSE</code> : names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for large models.
<code>method</code>	the integrator to use. This can either be a string constant naming one of the pre-defined methods or a call to function <code>rkMethod</code> 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 to ensure compatibility with the Livermore-solvers, but the maximum number of steps in total is calculated as <code>length(times) * maxsteps</code> .
dllname	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette "compiledCode".
initfunc	if not <code>NULL</code> , 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 <code>initfunc</code> 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 <code>func</code> and <code>jacfunc</code> .
ipar	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
nout	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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 <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , 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 <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See <a href="#">forcings</a> or package vignette "compiledCode".
initforc	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in 'dllname'. It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette "compiledCode".
fcontrol	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
...	additional arguments passed to <code>func</code> 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  $\max\text{-norm}(\mathbf{e}/\mathbf{ewt}) \leq 1$ , where `ewt` is a vector of positive error weights. The values of `rtol` and `atol` should all be non-negative. The form of `ewt` is:

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

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 an 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`.

### Author(s)

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

### References

- Butcher, J. C. (1987) *The numerical analysis of ordinary differential equations, Runge-Kutta and general linear methods*, Wiley, Chichester and New York.
- Engeln-Muellges, G. and Reutter, F. (1996) *Numerik Algorithmen: Entscheidungshilfe zur Auswahl und Nutzung*. VDI Verlag, Duesseldorf.
- Hindmarsh, Alan C. (1983) ODEPACK, A Systematized Collection of ODE Solvers; in p.55–64 of Stepleman, R.W. et al.[ed.] (1983) *Scientific Computing*, North-Holland, Amsterdam.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2007) *Numerical Recipes in C*. Cambridge University Press.

### 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

```
## =====
## Example: Resource-producer-consumer Lotka-Volterra model
## =====

## Note:
## parameters are a list, names accessible via "with" function
## (see also ode and lsoda examples)

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
times <- seq(0, 100, 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

## Parameters for steady state conditions
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
y <- xstart <- c(S = 1, P = 1, C = 1)

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"))
})

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

<code>y</code>	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.
<code>times</code>	times at which explicit estimates for <code>y</code> are desired. The first value in <code>times</code> must be the initial time.
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i> ) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a <code>names</code> attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to <code>time</code> , and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> . If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>rk4</code> is called. See package vignette "compiledCode" for more details.
<code>parms</code>	vector or list of parameters used in <code>func</code> .
<code>verbose</code>	a logical value that, when TRUE, triggers more verbose output from the ODE solver.
<code>ynames</code>	if FALSE: names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for large models.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette "compiledCode".
<code>initfunc</code>	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode",
<code>initpar</code>	only when 'dllname' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>ipar</code>	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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".

<code>outnames</code>	only used if <code>'dllname'</code> is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library.
<code>forcings</code>	only used if <code>'dllname'</code> is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in <code>'dllname'</code> . It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette <code>"compiledCode"</code> .
<code>fcontrol</code>	A list of control parameters for the forcing functions. See <a href="#">forcings</a> or vignette <code>compiledCode</code> .
<code>...</code>	additional arguments passed to <code>func</code> 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) compared to the generic Runge-Kutta codes called by [rk](#).

If you need different internal and external time steps, you may 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 an 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](mailto:thomas.petzoldt@tu-dresden.de)>

### See Also

- [rkMethod](#) for a list of available Runge-Kutta parameter sets,
- [rk](#) and [euler](#)
- [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

```
## =====
## 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)
x <- c(N = N0)

## 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
time <- seq(0, 100, 2)
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
time <- seq(0, 100, 2)
out <- as.data.frame(euler(x, time, logist, parms))
points(out$time, out$N, pch = 1)

## unstable result
time <- seq(0, 100, 4)
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	<i>Collection of Parameter Sets (Butcher Arrays) for the Runge-Kutta Family of ODE Solvers</i>
----------	--

---

### 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") or "rk45dp7" (alias "ode45").
...	specification of a user-defined solver, see <i>Value</i> 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:

<b>rkMethod</b>		<b>Description</b>
"euler"		Euler's Method
"rk2"		2nd order Runge-Kutta, fixed time step (Heun's method)
"rk4"		classical 4th order Runge-Kutta, fixed time step
"rk23"		Runge-Kutta, order 2(3), Octave: <b>ode23</b>
"rk23bs", "ode23"		Bogacki-Shampine, order 2(3), Matlab: <b>ode23</b>
"rk34f"		Runge-Kutta-Fehlberg, order 3(4)
"rk45ck"		Runge-Kutta Cash-Karp, order 4(5)
"rk45f"		Runge-Kutta-Fehlberg, order 4(5), Octave: <b>ode45, pair=1</b>
"rk45e"		Runge-Kutta-England, order 4(5)
"rk45dp6"		Dormand-Prince, order 4(5), local order 6
"rk45dp7", "ode45"		Dormand-Prince 4(5), local order 7
		(also known as <b>dopri5</b> , MATLAB: <b>ode45</b> , Octave: <b>ode45, pair=0</b> )

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.

### 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 <code>link{rk}</code> 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	weighting coefficients for averaging the function evaluations of method 1.
b2	weighting coefficients for averaging the function evaluations of method 2 (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 <code>rk45dp7</code> (Dormand-Prince).
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 (default = 5th order). If <code>nknots &lt; 2</code> internal interpolation is switched off. Note that this works only if time steps match exactly. This may not work as expected for all cases because of floating point rounding errors (see R FAQ about “Why doesn’t R think these numbers are equal?”).
alpha	optional tuning parameter for stepsize adjustment. If <code>alpha</code> is omitted, it is set to $1/Qerr - 0.75beta$ . The default value is $1/Qerr$ (for <code>beta = 0</code> ).
beta	optional tuning parameter for stepsize adjustment. Typical values are 0 (default) or $0.4/Qerr$ .

### Note

- Adaptive stepsize Runge-Kuttas 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).
- 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. Interpolation for all other methods is done with Neville-Aitken polynomials (5th order by default) which needs at least 6 internal time steps. Local interpolation inaccuracies can be minimized if external and internal time steps are not too different.

### Author(s)

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

### References

- Bogacki, P. and Shampine L.F. (1989) A 3(2) pair of Runge-Kutta formulas, Appl. Math. Lett. **2**, 1–9.
- Butcher, J. C. (1987) The numerical analysis of ordinary differential equations, Runge-Kutta and general linear methods, Wiley, Chichester and New York.
- Cash, J. R. and Karp A. H., 1990. A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides, ACM Transactions on Mathematical Software **16**, 201–222.
- Dormand, J. R. and Prince, P. J. (1980) A family of embedded Runge-Kutta formulae, J. Comput. Appl. Math. **6**(1), 19–26.
- Dormand, J. R. and Prince, P. J. (1981) High order embedded Runge-Kutta formulae, J. Comput. Appl. Math. **7**(1), 67–75.
- Engeln-Muellges, G. and Reutter, F. (1996) Numerik Algorithmen: Entscheidungshilfe zur Auswahl und Nutzung. VDI Verlag, Duesseldorf.
- Fehlberg, E. (1967) Klassische Runge-Kutta-Formeln fuer fuenfter and siebenter Ordnung mit Schrittweiten-Kontrolle, Computing (Arch. Elektron. Rechnen) **4**, 93–106.
- Kutta, W. (1901) Beitrag zur naeherungsweise Integration totaler Differentialgleichungen, Z. Math. Phys. **46**, 435–453.
- Octave-Forge - Extra Packages for GNU Octave, Package OdePkg. <http://octave.sourceforge.net/doc/odepkg.html>
- Runge, C. (1895) Ueber die numerische Aufloesung von Differentialgleichungen, Math. Ann. **46**, 167–178.
- 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 fall back to Neville-Aitken polynomials
o2 <- ode(x, times, func, parms, method = rkMethod("rk45dp7", d = NULL))

## 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]-o0[,-1], type="l", lty="dashed", xlab="Time",
        main="difference: Neville-Aitken - Dense Output")
abline(h=0, col="grey")

## 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,
        Qerr    = 1
    )
)

```

**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:

$$\frac{dC}{dt} = Flux - kC$$

This model is written in FORTRAN.

**Usage**

```
SCOC(times, y=NULL, parms, Flux, ...)
```

**Arguments**

<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time,
<code>y</code>	the initial value of the state variable; if <code>NULL</code> it will be estimated based on <code>Flux</code> and <code>parms</code> ,
<code>parms</code>	the model parameter, <code>k</code> ,
<code>Flux</code>	a data set with the organic carbon deposition rates,
<code>...</code>	any other parameters passed to the integrator <code>ode</code> (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 ‘`dynload`’ subdirectory of the package.

**Author(s)**

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

**References**

Soetaert, K. and P.M.J. Herman, 2009. A Practical Guide to Ecological Modelling. Using R as a Simulation Platform. Springer, 372 pp.

**See Also**

[ccl4model](#), the CCl4 inhalation model.

[aquaphy](#), the algal growth model.

**Examples**

```
# 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))

parms <- c(k=0.01)

times <- 1:365
out<- SCOC(times,parms=parms,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=parms,Flux=Flux, fcontrol=fcontrol)

plot(out2$time,out2$Depo,type="l",col="red")
lines(out2$time,out2$Mineralisation,col="blue")
```

**Description**

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

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

The R function `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 R function or be defined in compiled code that has been dynamically loaded.

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

`vode` is very similar to `lsode`, but uses a variable-coefficient method rather than the fixed-step-interpolate methods in `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 `lsode`, a copy is not kept.

### Usage

```
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, ...)
```

### Arguments

<code>y</code>	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.
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time; if only one step is to be taken; set <code>times = NULL</code> .
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i> ) at time <code>t</code> , or a character string giving the name of a compiled function in a dynamically loaded shared library. If <code>func</code> is an R-function, it must be defined as: <code>func &lt;- function(t, y, parms, ...)</code> . <code>t</code> is the current time point in the integration, <code>y</code> is the current estimate of the variables in the ODE system. If the initial values <code>y</code> has a names attribute, the names will be available inside <code>func</code> . <code>parms</code> is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of <code>func</code> should be a list, whose first element is a vector containing the derivatives of <code>y</code> with respect to time, and whose next elements are global values that are required at each point in <code>times</code> . The derivatives should be specified in the same order as the state variables <code>y</code> . If <code>func</code> is a string, then <code>dllname</code> must give the name of the shared library (without extension) which must be loaded before <code>vode()</code> is called. See package vignette "compiledCode" for more details.
<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacfunc</code>	if not <code>NULL</code> , an R function that computes the Jacobian of the system of differential equations $dy(i)/dy(j)$ , 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 <code>jacfunc</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jacfunc</code> is identical to that of <code>func</code> .

	<p>If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix <code>dydot/dy</code>, where the <math>i</math>th row contains the derivative of <math>dy_i/dt</math> with respect to <math>y_j</math>, or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).</p> <p>If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code>.</p>
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if <code>mf</code> is not NULL.
<code>mf</code>	the "method flag" passed to function <code>vode</code> - overrules <code>jactype</code> - provides more options than <code>jactype</code> - see details.
<code>verbose</code>	if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.
<code>tcrit</code>	if not NULL, then <code>vode</code> cannot integrate past <code>tcrit</code> . The FORTRAN routine <code>dvode</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>hmin</code>	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 <code>hmin</code> if you don't know why!
<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver.
<code>yname</code>	logical; if FALSE: names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for multi-D models.
<code>maxord</code>	the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method ( <code>meth = 1</code> ), order 5 if BDF method ( <code>meth = 2</code> ). Reduce <code>maxord</code> to save storage space.
<code>bandup</code>	number of non-zero bands above the diagonal, in case the Jacobian is banded.
<code>banddown</code>	number of non-zero bands below the diagonal, in case the Jacobian is banded.
<code>maxsteps</code>	maximal number of steps per output interval taken by the solver.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette "compiledCode".
<code>initfunc</code>	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".
<code>initpar</code>	only when 'dllname' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .

<code>ipar</code>	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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".
<code>outnames</code>	only used if 'dllname' is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	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 <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See <a href="#">forcings</a> or package vignette "compiledCode".
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in 'dllname'. It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette "compiledCode".
<code>fcontrol</code>	A list of control parameters for the forcing functions. <a href="#">forcings</a> or package vignette "compiledCode"
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> 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*(10*METH + MITER)`, where

**JSV = 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 be 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 an 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

P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, 1989. VODE: A Variable Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, 10, pp. 1038-1051.  
Also, LLNL Report UCRL-98412, June 1988.

G. D. Byrne and A. C. Hindmarsh, 1975. A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations. ACM Trans. Math. Software, 1, pp. 71-96.

A. C. Hindmarsh and G. D. Byrne, 1977. EPISODE: An Effective Package for the Integration of Systems of Ordinary Differential Equations. LLNL Report UCID-30112, Rev. 1.

G. D. Byrne and A. C. Hindmarsh, 1976. EPISODEB: An Experimental Package for the Integration of Systems of Ordinary Differential Equations with Banded Jacobians. LLNL Report UCID-30132, April 1976.

A. C. Hindmarsh, 1983. ODEPACK, a Systematized Collection of ODE Solvers. in Scientific Computing, R. S. Stepleman et al., eds., North-Holland, Amsterdam, pp. 55-64.

K. R. Jackson and R. Sacks-Davis, 1980. An Alternative Implementation of Variable Step-Size Multistep Formulas for Stiff ODEs. ACM Trans. Math. Software, 6, pp. 295-318.

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

```
## =====
## ex. 1
## The famous Lorenz equations: chaos in the earth's atmosphere
## Lorenz 1963. J. Atmos. Sci. 20, 130-141.
## =====

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")

## =====
## ex. 2
## SCOC model, in FORTRAN - to see the FORTRAN code:
## browseURL(paste(system.file(package="deSolve"),
##                  "/doc/examples/dynload/scoc.f",sep=""))
## example from Soetaert and Herman, 2009, chapter 3. (simplified)
## =====

# 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))

parms <- 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:

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

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

```
zvode(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, ...)
```

## Arguments

<code>y</code>	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. <i>y has to be complex</i>
<code>times</code>	time sequence for which output is wanted; the first value of <code>times</code> must be the initial time; if only one step is to be taken; set <code>times = NULL</code> .
<code>func</code>	either an R-function that computes the values of the derivatives in the ODE system (the <i>model definition</i> ) at time <code>t</code> , 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 `vode()` is called. See package vignette "compiledCode" for more details.

<code>parms</code>	vector or list of parameters used in <code>func</code> or <code>jacfunc</code> .
<code>rtol</code>	relative error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>atol</code>	absolute error tolerance, either a scalar or an array as long as <code>y</code> . See details.
<code>jacfunc</code>	if not <code>NULL</code> , an R function that computes the Jacobian of the system of differential equations $dy(i)/dy(j)$ , or a string giving the name of a function or subroutine in ' <code>dllname</code> ' that computes the Jacobian (see vignette "compiledCode" for more about this option).  In some circumstances, supplying <code>jacfunc</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jacfunc</code> is identical to that of <code>func</code> .  If the Jacobian is a full matrix, <code>jacfunc</code> should return a matrix $dydot/dy$ , where the $i$ th 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 <i>complex numbers</i> .  If the Jacobian is banded, <code>jacfunc</code> should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of <code>lsode</code> .
<code>jactype</code>	the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if <code>mf</code> is not <code>NULL</code> .
<code>mf</code>	the "method flag" passed to function <code>vode</code> - overrules <code>jactype</code> - provides more options than <code>jactype</code> - see details.
<code>verbose</code>	if <code>TRUE</code> : full output to the screen, e.g. will print the diagnostics of the integration - see details.
<code>tcrit</code>	if not <code>NULL</code> , then <code>vode</code> cannot integrate past <code>tcrit</code> . The FORTRAN routine <code>dvode</code> overshoots its targets (times points in the vector <code>times</code> ), 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 <code>tcrit</code> .
<code>hmin</code>	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 <code>hmin</code> if you don't know why!

<code>hmax</code>	an optional maximum value of the integration stepsize. If not specified, <code>hmax</code> is set to the largest difference in <code>times</code> , to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<code>hini</code>	initial step size to be attempted; if 0, the initial step size is determined by the solver.
<code>yname</code> s	logical; if <code>FALSE</code> : names of state variables are not passed to function <code>func</code> ; this may speed up the simulation especially for multi-D models.
<code>maxord</code>	the maximum order to be allowed. <code>NULL</code> uses the default, i.e. order 12 if implicit Adams method ( <code>meth = 1</code> ), order 5 if BDF method ( <code>meth = 2</code> ). Reduce <code>maxord</code> to save storage space.
<code>bandup</code>	number of non-zero bands above the diagonal, in case the Jacobian is banded.
<code>banddown</code>	number of non-zero bands below the diagonal, in case the Jacobian is banded.
<code>maxsteps</code>	maximal number of steps per output interval taken by the solver.
<code>dllname</code>	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in <code>func</code> and <code>jacfunc</code> . See package vignette "compiledCode".
<code>initfunc</code>	if not <code>NULL</code> , the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".
<code>initpar</code>	only when 'dllname' is specified and an initialisation function <code>initfunc</code> is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
<code>rpar</code>	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>ipar</code>	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by <code>func</code> and <code>jacfunc</code> .
<code>nout</code>	only used if <code>dllname</code> is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function <code>func</code> , 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".
<code>outname</code> s	only used if 'dllname' is specified and <code>nout &gt; 0</code> : the names of output variables calculated in the compiled function <code>func</code> , present in the shared library. These names will be used to label the output matrix.
<code>forcings</code>	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 <code>[min(times), max(times)]</code> is done by taking the value at the closest data extreme. See <a href="#">forcings</a> or package vignette "compiledCode".
<code>initforc</code>	if not <code>NULL</code> , the name of the forcing function initialisation function, as provided in 'dllname'. It <b>MUST</b> be present if <code>forcings</code> has been given a value. See <a href="#">forcings</a> or package vignette "compiledCode".
<code>fcontrol</code>	A list of control parameters for the forcing functions. <a href="#">forcings</a> or package vignette "compiledCode"
<code>...</code>	additional arguments passed to <code>func</code> and <code>jacfunc</code> 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 an 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**

P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, 1989. VODE: A Variable Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, 10, pp. 1038-1051.  
Also, LLNL Report UCRL-98412, June 1988.

G. D. Byrne and A. C. Hindmarsh, 1975. A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations. *ACM Trans. Math. Software*, 1, pp. 71-96.

A. C. Hindmarsh and G. D. Byrne, 1977. EPISODE: An Effective Package for the Integration of Systems of Ordinary Differential Equations. LLNL Report UCID-30112, Rev. 1.

G. D. Byrne and A. C. Hindmarsh, 1976. EPISODEB: An Experimental Package for the Integration of Systems of Ordinary Differential Equations with Banded Jacobians. LLNL Report UCID-30132, April 1976.

A. C. Hindmarsh, 1983. ODEPACK, a Systematized Collection of ODE Solvers. in *Scientific Computing*, R. S. Stepleman et al., eds., North-Holland, Amsterdam, pp. 55-64.

K. R. Jackson and R. Sacks-Davis, 1980. An Alternative Implementation of Variable Step-Size Multistep Formulas for Stiff ODEs. *ACM Trans. Math. Software*, 6, pp. 295-318.

Netlib: <http://www.netlib.org>

**See Also**

[vode](#) for the double precision version

## Examples

```

## =====
## 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 <- li*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(li*times)

# compare numerical and analytical solution
tail(cbind(out[,2],analytical.solution))

## =====
## Example 2 - example in "zvode.f",
## df/dt = li*f (same as above ODE)
## dg/dt = -li*g*g*f (an additional ODE depending on f)
##
## Initial values are
## g(0) = 1/2.1 and
## z(0) = 1
## =====

ZODE2<-function(Time,State,Pars) {
  with(as.list(State), {
    df <- li * f
    dg <- -li*g*g*f
    return(list(c(df,dg)))
  })
}

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)

analytical <- cbind(f=exp(1i*times), g=1/(exp(1i*times)+1.1))

#compare numerical solution and the two analytical ones:
tail(cbind(out[,2],analytical[,1]))
```

# Index

## \*Topic **datasets**

ccl4data, 8

## \*Topic **hplot**

plot.deSolve, 74

## \*Topic **math**

daspk, 11

lsoda, 28

lsodar, 35

lsode, 41

lsodes, 48

ode, 55

ode.1D, 59

ode.2D, 64

ode.3D, 68

ode.band, 72

rk, 76

rk4, 81

rkMethod, 85

vode, 90

zvode, 97

## \*Topic **models**

aquaphy, 4

ccl4model, 8

SCOC, 89

## \*Topic **package**

deSolve-package, 2

## \*Topic **utilities**

diagnostics, 19

diagnostics.deSolve, 20

DLLfunc, 21

DLLres, 23

forcings, 25

.C, 11, 28, 76

.Call, 11, 28, 76

.Fortran, 28, 76

approxfun, 25, 26

aquaphy, 4, 9, 57, 90

ccl4data, 8, 9

ccl4model, 5, 8, 57, 90

daspk, 3, 11, 25, 32, 39, 45, 52, 57, 79, 83, 95  
deSolve, 75

deSolve (deSolve-package), 2

deSolve-package, 2

dev.interactive, 75

diagnostics, 15, 16, 19, 31, 32, 38, 39, 45,  
51, 52, 57, 61, 66, 70, 73, 80, 84, 94,  
95

diagnostics.deSolve, 20, 20

DLLfunc, 3, 21

DLLres, 3, 23

euler, 3, 16, 32, 39, 45, 52, 79, 83, 95

euler (rk4), 81

forcings, 14, 15, 25, 31, 32, 38, 39, 44, 45,  
51, 78, 83, 93, 94, 99

gnls, 32

hist, 75

hist.deSolve (plot.deSolve), 74

lsoda, 3, 15, 16, 28, 35, 38, 39, 41, 45, 51,  
52, 57, 61, 73, 79, 83, 91, 94, 95

lsodar, 3, 16, 32, 35, 45, 52, 57, 61, 73, 79,  
83, 95

lsode, 3, 16, 32, 39, 41, 52, 57, 61, 73, 79,  
83, 91, 94, 95

lsodes, 3, 16, 32, 39, 45, 48, 57, 61, 66, 70,  
79, 83, 95

nlm, 2, 28, 76

nlme, 2, 28, 76

nls, 2, 28, 76

ode, 3, 16, 22, 32, 39, 45, 52, 55, 61, 65, 70,  
73, 75, 80, 83, 85, 87, 95

ode.1D, 3, 16, 32, 39, 45, 52, 57, 59, 66, 70,  
73, 80, 84, 95  
ode.2D, 3, 16, 32, 39, 45, 52, 57, 61, 64, 70,  
73, 80, 84, 95  
ode.3D, 3, 16, 32, 39, 45, 52, 57, 61, 66, 68,  
73, 80, 84, 95  
ode.band, 3, 16, 32, 39, 45, 52, 57, 60, 61,  
65, 70, 72, 80, 83, 95  
optim, 2, 28, 76  
  
par, 75  
plot.default, 75  
plot.deSolve, 57, 74  
print.deSolve, 75  
print.deSolve(ode), 55  
  
rk, 3, 16, 32, 39, 45, 52, 57, 76, 83, 85, 87, 95  
rk4, 3, 16, 32, 39, 45, 52, 79, 81, 95  
rkMethod, 57, 77–79, 83, 85  
  
SCOC, 89  
  
vode, 3, 16, 32, 39, 41, 45, 52, 57, 61, 73, 79,  
83, 90, 100  
  
zvode, 97