

Package ‘odesolve’

March 19, 2012

Version 0.9-9

Date 2012/3/19

Title Solvers for Ordinary Differential Equations

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

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

Depends R (>= 2.3.1)

Description This package provides an interface for the ODE solver
lsoda. ODEs are expressed as R functions or as compiled code.
This is deprecated and will disappear from CRAN by the end of 2012! Use deSolve instead.

License GPL-2

Repository CRAN

Date/Publication 2012-03-19 17:53:35

R topics documented:

ccl4data	2
ccl4data.avg	2
lsoda	3
rk4	10

Index	13
--------------	-----------

`ccl4data`*Closed chamber study of CCl4 metabolism by rats.*

Description

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

Usage

```
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

```
data(ccl4data)
```

`ccl4data.avg`*Closed chamber study of CCl4 metabolism by rats.*

Description

The results of a closed chamber experiment to determine metabolic parameters for CCl₄ (carbon tetrachloride) in rats. This is the summary version of `ccl4data`. Each record is the average for the time point of all animals exposed to the given initial chamber concentration.

Usage

```
data(ccl4data.avg)
```

Format

This data frame contains the following columns:

time The time of the observation (hours after starting)

initconc Initial chamber concentration (in ppm)

ChamberConc Mean chamber concentration at the specified time (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

```
data(ccl4data.avg)
```

lsoda	<i>Solve System of ODE (ordinary differential equation)s.</i>
-------	---

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`, `nlm` or `nlme`.

Usage

```
lsoda(y, times, func, parms, rtol, atol, tcrit=NULL, jacfunc=NULL,
      verbose=FALSE, dllname=NULL, hmin=0, hmax=Inf, ...)
```

Arguments

<code>y</code>	the initial 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 a user-supplied 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 a user-supplied function, it must be called as: <code>yprime = func(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 of parameters (which may have a `names` attribute, desirable in a large 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 is a vector (possibly with a `names` attribute) of global values that are required at each point in `times`.

If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `lsoda()` is called. see Details for more information.

<code>parms</code>	any parameters used in <code>func</code> that should be modifiable without rewriting the function.
<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>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> . Note that it does not make sense (though it is not an error) to include times in <code>times</code> past <code>tcrit</code> , since the solver will stop and return at the last point in <code>times</code> that is earlier than <code>tcrit</code> .
<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 Details below for more about this option). In some circumstances, supplying <code>jac</code> can speed up the computations, if the system is stiff. The R calling sequence for <code>jac</code> is identical to that of <code>func</code> . <code>jac</code> should return a vector whose $((i-1)*length(y) + j)$ th value is $dy(i)/dy(j)$. That is, return the matrix $dydot/dy$, where the i th row is the derivative of dy_i/dt with respect to y_j , by columns (the way R and Fortran store matrices).
<code>verbose</code>	a logical value that, when <code>TRUE</code> , should trigger more verbose output from the ode solver. Currently does not do anything.
<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>jac</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. A maximum value may be necessary for non-autonomous models (with external inputs), otherwise the simulation possibly ignores short external events.
<code>...</code>	additional arguments, 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/lsoda.f'`). This is based on the Feb 24, 1997 version of `lsoda`, from Netlib. The following description of error control is adapted from that documentation (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 \mathbf{e} of estimated local errors in \mathbf{y} , according to an inequality of the form $\max\text{-norm}(\mathbf{e}/\mathbf{ewt}) \leq 1$, where \mathbf{ewt} is a vector of positive error weights. The values of `rtol` and `atol` should all be non-negative. The form of \mathbf{ewt} is:

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

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

If the request for precision exceeds the capabilities of the machine, the Fortran subroutine `Isoda` 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.

Models may be defined in compiled C or Fortran code, as well as in R. For C, the calling sequence for `func` must be as in the following example:

```
void myderivs(int *neq, double *t, double *y, double *ydot)
{
  ydot[0] = -k1*y[0] + k2*y[1]*y[2];
  ydot[2] = k3 * y[1]*y[1];
  ydot[1] = -ydot[0]-ydot[2];
}
```

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

In this example, parameters are kept in a global variable in the C code declared as

```
static double parms[3];
```

`#define` statements are used to make the code more readable, as in `#define k1 parms[0]`

This is the only way to pass parameters to a compiled C function from the calling R code. Functions that use this mechanism must be accompanied by a function with the same name as the shared library which has as its sole argument a pointer to a function (see declaration below) that fills a double array with double precision values, to copy the parameter values into the global variable. In the example here, the library is named `'mymod.so'`, a function such as:

```
void mymod(void (* odeparms)(int *, double *))
{
  int N=3;
  odeparms(&N, parms);
}
```

will be required to initialize the parameter vector. Here `mymod` just calls `odeparms` with a pointer to a `int` that contains the dimension of the parameter vector, and a pointer to the array that will contain the parameter values.

Models may also be defined in Fortran. For example:

```
subroutine myderivs (neq, t, y, ydot)
double precision t, y, ydot, parms(3)
integer neq
```

```

dimension y(3), ydot(3)
common /myparms/parms

ydot(1) = -parms(1)*y(1) + parms(2)*y(2)*y(3)
ydot(3) = parms(3)*y(2)*y(2)
ydot(2) = -ydot(1) - ydot(3)

return
end

```

In Fortran, parameters may be stored in a common block, in which case, the file that contains the model function definition must also contain a subroutine, again with the same name as the file which contains the model definition:

```

subroutine mymod(odeparms)
external odeparms
integer N
double precision parms(3)
common /myparms/parms

```

```

N = 3
call odeparms(N, parms)
return
end

```

When models are defined in compiled code, there is no provision for returning quantities that are not directly solutions of the odes (unlike models defined in R code).

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

```

void myjac(int *neq, double *t, double *y, int *ml,
int *mu, double *pd, int *nrowpd)
{
pd[0] = -k1;
pd[1] = k1;
pd[2] = 0.0;
pd[*nrowpd] = k2*y[2];
pd[*nrowpd] + 1] = -k2*y[2] - 2*k3*y[1];
pd[*nrowpd] + 2] = 2*k3*y[1];
pd[*nrowpd]*2] = k2*y[1];
pd[2*(*nrowpd) + 1] = -k2 * y[1];
pd[2*(*nrowpd) + 2] = 0.0;
}

```

The corresponding subroutine in Fortran is:

```

subroutine myjac (neq, t, y, ml, mu, pd, nrowpd)
integer neq, ml, mu, nrowpd

```

```
double precision y(*), pd(nrowpd,*), t, parms(3)
common /myparms/parms
```

```
pd(1,1) = -parms(1)
pd(2,1) = parms(1)
pd(3,1) = 0.0
pd(1,2) = parms(2)*y(3)
pd(2,2) = -parms(2)*y(3) - 2*parms(3)*y(2)
pd(3,2) = 2*parms(3)*y(2)
pd(1,3) = parms(2)*y(2)
pd(2,3) = -parms(2)*y(2)
pd(3,3) = 0.0
```

```
return
end
```

Examples in both C and Fortran are in the 'dynload' subdirectory of the odesolve package directory.

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 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. The output will have the attribute `istate` which returns the conditions under which the last call to `Isoda` returned. See the source code for an explanation of those values: normal is `istate = 2`.

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>

Examples

```

### lsexamp -- example 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)
{
  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
  )
}

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

all.equal(out, outJ) # TRUE

## Example for using hmax

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

## A simple resource limited Lotka-Volterra-Model
## Note passing parameters through using a closure
lvmodel <- with(as.list(parms), function(t, x, parms) {
  import <- sigimp(t)
  ds <- import - b*x["s"]*x["p"] + g*x["k"]
  dp <- c*x["s"]*x["p"] - d*x["k"]*x["p"]
  dk <- e*x["p"]*x["k"] - f*x["k"]
  res<-c(ds, dp, dk)
  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

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

## Start values for steady state
y<-xstart <- c(s=1, p=1, k=1)

## LSODA (default step size)
out2 <- as.data.frame(lsoda(xstart, times, lvmodel, parms))

## LSODA: with fixed maximum time step
out3 <- as.data.frame(lsoda(xstart, times, lvmodel, parms, hmax=1))

par(mfrow=c(2,2))
plot(out2$time, out2$s, type="l", ylim=c(0,3))
lines(out3$time, out3$s, col="green", lty="dotted")

plot(out2$time, out2$p, type="l", ylim=c(0,3))
lines(out3$time, out3$p, col="green", lty="dotted")

plot(out2$time, out2$k, type="l", ylim=c(0,3))
lines(out3$time, out3$k, col="green", lty="dotted")

```

```
plot(out2$p, out2$k, type="l", ylim=range(out2$k, out3$k))
lines(out3$p, out3$k, col="green", lty="dotted")
```

rk4 *Solve System of ODE (ordinary differential equation)s by classical Runge-Kutta 4th order integration.*

Description

Solving initial value problems for systems of first-order ordinary differential equations (ODEs) using the classical Runge-Kutta 4th order integration. The system of ODE's may be written as an R function (which may, of course, use `.C`, `.Fortran`, `.Call`, etc., to call foreign code). 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`, `nlm` or `nlme`.

Usage

```
rk4(y, times, func, parms, ...)
```

Arguments

y	the initial values for the ode system. If y has a name attribute, the names will be used to label the output matrix.
times	times at which explicit estimates for y are desired. The first value in times must be the initial time.
func	a user-supplied function that computes the values of the derivatives in the ode system (the <i>model definition</i>) at time t. The user-supplied function func must be called as: <code>yprime = func(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, and parms is a vector of parameters (which may have a names attribute, desirable in a large 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 is a vector (possibly with a <code>names</code> attribute) of global values that are required at each point in times.
parms	vector or list holding the parameters used in func that should be modifiable without rewriting the function.
...	additional arguments, allowing this to be a generic function

Details

The method is implemented primarily for didactic purposes. Please use `lsoda` for your real work!

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 for the time value. There will be a row for each element in `times`. If `y` has a `names` attribute, it will be used to label the columns of the output value.

Author(s)

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

References

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (1992) Numerical Recipes in C. Cambridge University Press.

See Also

[lsoda](#)

Examples

```
## A simple resource limited Lotka-Volterra-Model
lvmodel <- function(t, x, parms) {
  s <- x[1] # substrate
  p <- x[2] # producer
  k <- x[3] # consumer
  with(as.list(parms),{
    import <- approx(signal$times, signal$import, t)$y
    ds <- import - b*s*p + g*k
    dp <- c*s*p - d*k*p
    dk <- e*p*k - f*k
    res<-c(ds, dp, dk)
    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 <- c(a=0.0, 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, k=1)
```

```
## Classical RK4 with fixed time step
out1 <- as.data.frame(rk4(xstart, times, lvmodel, parms))

## LSODA (default step size)
out2 <- as.data.frame(lsoda(xstart, times, lvmodel, parms))

## LSODA: with fixed maximum time step
out3 <- as.data.frame(lsoda(xstart, times, lvmodel, parms, hmax=1))

par(mfrow=c(2,2))
plot(out1$time, out1$s, type="l", ylim=c(0,3))
lines(out2$time, out2$s, col="red", lty="dotted")
lines(out3$time, out3$s, col="green", lty="dotted")

plot(out1$time, out1$p, type="l", ylim=c(0,3))
lines(out2$time, out2$p, col="red", lty="dotted")
lines(out3$time, out3$p, col="green", lty="dotted")

plot(out1$time, out1$k, type="l", ylim=c(0,3))
lines(out2$time, out2$k, col="red", lty="dotted")
lines(out3$time, out3$k, col="green", lty="dotted")

plot(out1$p, out1$k, type="l")
lines(out2$p, out2$k, col="red", lty="dotted")
lines(out3$p, out3$k, col="green", lty="dotted")
```

Index

*Topic **datasets**

ccl4data, [2](#)

ccl4data.avg, [2](#)

*Topic **math**

lsoda, [3](#)

rk4, [10](#)

.C, [3](#), [10](#)

.Call, [3](#), [10](#)

ccl4data, [2](#), [2](#)

ccl4data.avg, [2](#)

gnls, [7](#)

lsoda, [3](#), [10](#), [11](#)

names, [4](#), [10](#)

nlm, [3](#), [10](#)

nlme, [3](#), [10](#)

optim, [3](#), [10](#)

rk4, [10](#)