Package 'cOde'

September 7, 2017

Type Package

Title Automated C Code Generation for 'deSolve', 'bvpSolve' and 'Sundials'

Version 0.3

NeedsCompilation yes

Depends R (>= 3.0)

Suggests deSolve, bvpSolve

Imports Rcpp (>= 0.11.3)

LinkingTo Rcpp, RcppArmadillo

SystemRequirements C++11

Date 2017-09-04

Author Daniel Kaschek

Maintainer Daniel Kaschek <daniel.kaschek@gmail.com>

Description Generates all necessary C functions allowing the user to work with the compiled-code interface of ode() and bvptwp(). The implementation supports ``forcings'' and ``events''. Also provides functions to symbolically compute Jacobians, sensitivity equations and adjoint sensitivities being the basis for sensitivity analysis. Alternatively to 'deSolve', the Sundials 'CVODES' solver is implemented for computation of model sensitivities.

License GPL (>= 2)

RoxygenNote 6.0.1

Repository CRAN

Date/Publication 2017-09-07 21:44:06 UTC

R topics documented:

cOde-package 2
adjointSymb 3
bvptwpC 5
compileAndLoad 7

cvodesSyntax 8
forcData 8
funC 9
getSymbols 10
jacobianSymb 11
odeC 11
oxygenData 13
prodSymb 13
reduceSensitivities 14
replaceOperation 14
replaceSymbols 15
sensitivitiesSymb 15
setForcings 20
sumSymb 21
sundialsIncludes 22
sundialsJac 22
sundialsOde 23
sundialsSensOde 23
wrap_cvodes 24

Index **27**

`c0de-package` *Automated C Code Generation for Use with the "deSolve" and "bvpSolve" Packages*

Description

Generates all necessary C functions allowing the user to work with the compiled-code interface of ode() and bvptwp(). The implementation supports "forcings" and "events". The package also provides functions to symbolically compute Jacobians, sensitivity equations and adjoint sensitivities being the basis for sensitivity analysis.

Details

Package:	cOde
Type:	Package
Version:	0.2
Date:	2015-03-05
License:	GNU Public License 2 or above

The system of ordinary differential equations is defined as a named character vector. This character vector is translated in C syntax and the code output format is matched to the requirements of deSolve and bvpSolve.

Author(s)

Daniel Kaschek <daniel.kaschek@physik.uni-freiburg.de>

`adjointSymb` *Compute adjoint equations of a function symbolically*

Description

Compute adjoint equations of a function symbolically

Usage

```
adjointSymb(f, states = names(f), parameters = NULL, inputs = NULL)
```

Arguments

`f`	Named vector of type character, the functions
`states`	Character vector of the ODE states for which observations are available
`parameters`	Character vector of the parameters
`inputs`	Character vector of the "variable" input states, i.e. time-dependent parameters (in contrast to the forcings).

Details

The adjoint equations are computed with respect to the functional

$$(x, u) \mapsto \int_0^T \|x(t) - x^D(t)\|^2 + \|u(t) - u^D(t)\|^2 dt,$$

where x are the states being constrained by the ODE, u are the inputs and xD and uD indicate the trajectories to be best possibly approached. When the ODE is linear with respect to u, the attribute `inputs` of the returned equations can be used to replace all occurences of u by the corresponding character in the attribute. This guarantees that the input course is optimal with respect to the above function.

Value

Named vector of type character with the adjoint equations. The vector has attributes "chi" (integrand of the chisquare functional), "grad" (integrand of the gradient of the chisquare functional), "forcings" (character vector of the forcings necessary for integration of the adjoint equations) and "inputs" (the input expressed as a function of the adjoint variables).

Examples

```
## Not run:

#########################################################################
## Solve an optimal control problem:
#########################################################################

library(bvpSolve)

# O2 + O <-> O3
# O3 is removed by a variable rate u(t)
f <- c(
  O3 = " build_O3 * O2 * O - decay_O3 * O3 - u * O3",
  O2 = "-build_O3 * O2 * O + decay_O3 * O3",
  O  = "-build_O3 * O2 * O + decay_O3 * O3"
)

# Compute adjoints equations and replace u by optimal input
f_a <- adjointSymb(f, states = c("O3"), inputs = "u")
inputs <- attr(f_a, "inputs")
f_tot <- replaceSymbols("u", inputs, c(f, f_a))
forcings <- attr(f_a, "forcings")

# Initialize times, states, parameters
times <- seq(0, 15, by = .1)
boundary <- data.frame(
  name = c("O3", "O2", "O", "adjO3", "adjO2", "adjO"),
  yini = c(0.5, 2, 2.5, NA, NA, NA),
  yend = c(NA, NA, NA, 0, 0, 0))

pars <- c(build_O3 = .2, decay_O3 = .1, eps = 1)

# Generate ODE function
func <- funC(f = f_tot, forcings = forcings,
             jacobian = "full", boundary = boundary,
             modelname = "example5")

# Initialize forcings (the objective)
forcData <- data.frame(time = times,
                       name = rep(forcings, each=length(times)),
                       value = rep(
                         c(0.5, 0, 1, 1), each=length(times)))
forc <- setForcings(func, forcData)

# Solve BVP
out <- bvptwpC(x = times, func = func, parms = pars, forcings = forc)

# Plot solution
par(mfcol=c(1,2))
t <- out[,1]
M1 <- out[,2:4]
M2 <- with(list(uD = 0, O3 = out[,2],
```

```
                adjO3 = out[,5], eps = 1, weightuD = 1),
            eval(parse(text=inputs)))

matplot(t, M1, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="states")
abline(h = .5, lty=2)
legend("topright", legend = names(f), lty=1, col=1:3)
matplot(t, M2, type="l", lty=1, col=1,
        xlab="time", ylab="value", main="input u")
abline(h = 0, lty=2)


## End(Not run)
```

---

bvptwpC *Interface to bvptwp()*

Description

Interface to bvptwp()

Usage

```
bvptwpC(yini = NULL, x, func, yend = NULL, parms, xguess = NULL,
  yguess = NULL, ...)
```

Arguments

yini	named vector of type numeric. Initial values to be overwritten.
x	vector of type numeric. Integration times
func	return value from funC() with a boundary argument.
yend	named vector of type numeric. End values to be overwritten.
parms	named vector of type numeric. The dynamic parameters.
xguess	vector of type numeric, the x values
yguess	matrix with as many rows as variables and columns as x values
...	further arguments going to bvptwp()

Details

See bvpSolve-package for a full description of possible arguments

Value

matrix with times and states

Examples

```
## Not run:

#########################################################################
## Boundary value problem: Ozon formation with fixed ozon/oxygen ratio
## at final time point
#########################################################################

library(bvpSolve)

# O2 + O <-> O3
# diff = O2 - O3
# build_O3 = const.
f <- c(
  O3 = " build_O3 * O2 * O - decay_O3 * O3",
  O2 = "-build_O3 * O2 * O + decay_O3 * O3",
  O  = "-build_O3 * O2 * O + decay_O3 * O3",
  diff = "-2 * build_O3 * O2 * O + 2 * decay_O3 * O3",
  build_O3 = "0"
)

bound <- data.frame(
    name = names(f),
    yini = c(0, 3, 2, 3, NA),
    yend = c(NA, NA, NA, 0, NA)
)

# Generate ODE function
func <- funC(f, jacobian="full", boundary = bound, modelname = "example4")

# Initialize times, states, parameters and forcings
times <- seq(0, 15, by = .1)
pars <- c(decay_O3 = .1)
xguess <- times
yguess <- matrix(c(1, 1, 1, 1, 1), ncol=length(times),
                 nrow = length(f))

# Solve BVP
out <- bvptwpC(x = times, func = func, parms = pars,
               xguess = xguess, yguess = yguess)

# Solve BVP for different ini values, end values and parameters
yini <- c(O3 = 2)
yend <- c(diff = 0.2)
pars <- c(decay_O3 = .01)
out <- bvptwpC(yini = yini, yend = yend, x = times, func = func,
       parms = pars, xguess = xguess, yguess = yguess)

# Plot solution
par(mfcol=c(1,2))
t <- out[,1]
M1 <- out[,2:5]
```

```
M2 <- cbind(out[,6], pars)

matplot(t, M1, type="l", lty=1, col=1:4,
        xlab="time", ylab="value", main="states")
legend("topright", legend = c("O3", "O2", "O", "O2 - O3"),
       lty=1, col=1:4)
matplot(t, M2, type="l", lty=1, col=1:2,
        xlab="time", ylab="value", main="parameters")
legend("right", legend = c("build_O3", "decay_O3"), lty=1, col=1:2)


## End(Not run)
```

---

compileAndLoad *Compile and load shared object implementing the ODE system.*

Description

Compile and load shared object implementing the ODE system.

Usage

```
compileAndLoad(filename, dllname, fcontrol, verbose)
```

Arguments

filename	Full file name of the source file.
dllname	Base name for source and dll file.
fcontrol	Interpolation method for forcings.
verbose	Print compiler output or not.

Author(s)

Daniel Kaschek, <daniel.kaschek@physik.uni-freiburg.de>

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

cvodesSyntax *Sundials::cvodes: Replace powers and symbols to get correct C++ syntax for cvodes.*

Description

Sundials::cvodes: Replace powers and symbols to get correct C++ syntax for cvodes.

Usage

```
cvodesSyntax(f, variables, parameters, varSym = "y", parSym = "p",
  forcings = NULL)
```

Arguments

f	Named character vector.
variables	Variables appearing on the ODE, names(f).
parameters	Parameters appearing in the ODE.
varSym	Symbol of the variables to appear in the source code.
parSym	Symbol of the parameter to appear in the source code.
forcings	Inhomogenity of the ODE. Currently not used.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

forcData *Forcings data.frame*

Description

Forcings data.frame

funC *Generate C code for a function and compile it*

Description

Generate C code for a function and compile it

Usage

```
funC(f, forcings = NULL, fixed = NULL, outputs = NULL,
  jacobian = c("none", "full", "inz.lsodes", "jacvec.lsodes"),
  rootfunc = NULL, boundary = NULL, compile = TRUE,
  fcontrol = c("nospline", "einspline"), nGridpoints = 500,
  precision = 1e-05, modelname = NULL, verbose = FALSE,
  solver = c("deSolve", "Sundials"))
```

Arguments

f	Named character vector containing the right-hand sides of the ODE. You may use the key word time in your equations for non-autonomous ODEs.
forcings	Character vector with the names of the forcings
fixed	character vector with the names of parameters (initial values and dynamic) for which no sensitivities are required (will speed up the integration).
outputs	Named character vector for additional output variables, see arguments nout and outnames of lsode
jacobian	Character, either "none" (no jacobian is computed), "full" (full jacobian is computed and written as a function into the C file) or "inz.lsodes" (only the non-zero elements of the jacobian are determined, see lsodes)
rootfunc	Named character vector. The root function (see lsoda). Besides the variable names (names(f)) also other symbols are allowed that are treated like new parameters.
boundary	data.frame with columns name, yini, yend specifying the boundary condition set-up. NULL if not a boundary value problem
compile	Logical. If FALSE, only the C file is written
fcontrol	Character, either "nospline" (default, forcings are handled by deSolve) or "einspline" (forcings are handled as splines within the C code based on the einspline library).
nGridpoints	Integer, defining the number of grid points between tmin and tmax where the ODE is computed in any case. Indicates also the number of spline nodes if fcontrol = "einspline".
precision	Numeric. Only used when fcontrol = "einspline".
modelname	Character. The C file is generated in the working directory and is named <modelname>.c. If NULL, a random name starting with ".f" is chosen, i.e. the file is hidden on a UNIX system.

verbose	Print compiler output to R command line.
solver	Select the solver suite as either deSolve or Sundials. Defaults to deSolve.

Details

The function replaces variables by arrays y[i], etc. and replaces "^" by pow() in order to have the correct C syntax. The file name of the C-File is derived from f. I.e. funC(abc, ... will generate a file abc.c in the current directory. Currently, only explicit ODE specification is supported, i.e. you need to have the right-hand sides of the ODE.

Value

the name of the generated shared object file together with a number of attributes

Examples

```
## Not run:
# Exponential decay plus constant supply
f <- c(x = "-k*x + supply")
func <- funC(f, forcings = "supply")

# Example 2: root function
f <- c(A = "-k1*A + k2*B", B = "k1*A - k2*B")
rootfunc <- c(steadyState = "-k1*A + k2*B - tol")

func <- funC(f, rootfunc = rootfunc, modelname = "test")

yini <- c(A = 1, B = 2)
parms <- c(k1 = 1, k2 = 5, tol = 0.1)
times <- seq(0, 10, len = 100)

odeC(yini, times, func, parms)

## End(Not run)
```

---

getSymbols *Get symbols from a character*

Description

Get symbols from a character

Usage

```
getSymbols(char, exclude = NULL)
```

Arguments

char	Character vector (e.g. equation)
exclude	Character vector, the symbols to be excluded from the return value

Value

character vector with the symbols

Examples

```
getSymbols(c("A*AB+B^2"))
```

`jacobianSymb` *Compute Jacobian of a function symbolically*

Description

Compute Jacobian of a function symbolically

Usage

```
jacobianSymb(f, variables = NULL)
```

Arguments

`f`	named vector of type character, the functions
`variables`	other variables, e.g. paramters, f depends on. If variables is given, f is derived with respect to variables instead of `names(f)`

Value

named vector of type character with the symbolic derivatives

Examples

```
jacobianSymb(c(A="A*B", B="A+B"))
jacobianSymb(c(x="A*B", y="A+B"), c("A", "B"))
```

`odeC` *Interface to ode()*

Description

Interface to ode()

Usage

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

Arguments

y	named vector of type numeric. Initial values for the integration
times	vector of type numeric. Integration times
func	return value from funC()
parms	named vector of type numeric.
...	further arguments going to ode()

Details

See deSolve-package for a full description of possible arguments

Value

matrix with times and states

Examples

```
## Not run:

##########################################################################
## Ozone formation and decay, modified by external forcings
##########################################################################

library(deSolve)
data(forcData)
forcData$value <- forcData$value + 1

# O2 + O <-> O3
f <- c(
  O3 = " (build_O3 + u_build) * O2 * O - (decay_O3 + u_degrade) * O3",
  O2 = "-(build_O3 + u_build) * O2 * O + (decay_O3 + u_degrade) * O3",
  O  = "-(build_O3 + u_build) * O2 * O + (decay_O3 + u_degrade) * O3"
)

# Generate ODE function
forcings <- c("u_build", "u_degrade")
func <- funC(f, forcings = forcings, modelname = "test",
             fcontrol = "nospline", nGridpoints = 10)

# Initialize times, states, parameters and forcings
times <- seq(0, 8, by = .1)
yini <- c(O3 = 0, O2 = 3, O = 2)
pars <- c(build_O3 = 1/6, decay_O3 = 1)

forc <- setForcings(func, forcData)

# Solve ODE
out <- odeC(y = yini, times = times, func = func, parms = pars,
            forcings = forc)
```

```
# Plot solution

par(mfcol=c(1,2))
t1 <- unique(forcData[,2])
M1 <- matrix(forcData[,3], ncol=2)
t2 <- out[,1]
M2 <- out[,2:4]
M3 <- out[,5:6]

matplot(t1, M1, type="l", lty=1, col=1:2, xlab="time", ylab="value",
main="forcings", ylim=c(0, 4))
matplot(t2, M3, type="l", lty=2, col=1:2, xlab="time", ylab="value",
main="forcings", add=TRUE)

legend("topleft", legend = c("u_build", "u_degrade"), lty=1, col=1:2)
matplot(t2, M2, type="l", lty=1, col=1:3, xlab="time", ylab="value",
main="response")
legend("topright", legend = c("O3", "O2", "O"), lty=1, col=1:3)


## End(Not run)
```

`oxygenData` *Time-course data of O, O2 and O3*

Description

Forcings data.frame

`prodSymb` *Compute matrix product symbolically*

Description

Compute matrix product symbolically

Usage

```
prodSymb(M, N)
```

Arguments

`M`	matrix of type character
`N`	matrix of type character

Value

Matrix of type character, the matrix product of M and N

reduceSensitivities *reduceSensitivities*

Description

reduceSensitivities

Usage

```
reduceSensitivities(sens, vanishing)
```

Arguments

sens	Named character, the sensitivity equations
vanishing	Character, names of the vanishing sensitivities

Details

Given the set vanishing of vanishing sensitivities, the algorithm determins sensitivities that vanish as a consequence of the first set.

Value

Named character, the sensitivity equations with zero entries for vanishing sensitivities.

replaceOperation *Replace a binary operator in a string by a function*

Description

Replace a binary operator in a string by a function

Usage

```
replaceOperation(what, by, x)
```

Arguments

what	character, the operator symbol, e.g. "^"
by	character, the function string, e.g. "pow"
x	vector of type character, the object where the replacement should take place

Value

vector of type character

Examples

```
replaceOperation("^", "pow", "(x^2 + y^2)^.5")
```

`replaceSymbols` *Replace symbols in a character vector by other symbols*

Description

Replace symbols in a character vector by other symbols

Usage

```
replaceSymbols(what, by, x)
```

Arguments

`what`	vector of type character, the symbols to be replaced, e.g. c("A", "B")
`by`	vector of type character, the replacement, e.g. c("x[0]", "x[1]")
`x`	vector of type character, the object where the replacement should take place

Value

vector of type character, conserves the names of x.

Examples

```
replaceSymbols(c("A", "B"), c("x[0]", "x[1]"), c("A*B", "A+B+C"))
```

`sensitivitiesSymb` *Compute sensitivity equations of a function symbolically*

Description

Compute sensitivity equations of a function symbolically

Usage

```
sensitivitiesSymb(f, states = names(f), parameters = NULL, inputs = NULL,
  reduce = FALSE)
```

Arguments

f	named vector of type character, the functions
states	Character vector. Sensitivities are computed with respect to initial values of these states
parameters	Character vector. Sensitivities are computed with respect to initial values of these parameters
inputs	Character vector. Input functions or forcings. They are excluded from the computation of sensitivities.
reduce	Logical. Attempts to determine vanishing sensitivities, removes their equations and replaces their right-hand side occurences by 0.

Details

The sensitivity equations are ODEs that are derived from the original ODE f. They describe the sensitivity of the solution curve with respect to parameters like initial values and other parameters contained in f. These eqution are also useful for parameter estimation by the maximum-likelihood method. For consistency with the time-continuous setting provided by adjointSymb, the returned equations contain attributes for the chisquare functional and its gradient.

Value

Named vector of type character with the sensitivity equations. Furthermore, attributes "chi" (the integrand of the chisquare functional), "grad" (the integrand of the gradient of the chisquare functional), "forcings" (Character vector of the additional forcings being necessare to compute `chi` and `grad`) and "yini" (The initial values of the sensitivity equations) are returned.

Examples

```
## Not run:

#########################################################################
## Sensitivity analysis of ozone formation
#########################################################################

library(deSolve)

# O2 + O <-> O3
f <- c(
  O3 = " build_O3 * O2 * O - decay_O3 * O3",
  O2 = "-build_O3 * O2 * O + decay_O3 * O3",
  O  = "-build_O3 * O2 * O + decay_O3 * O3"
)

# Compute sensitivity equations
f_s <- sensitivitiesSymb(f)

# Generate ODE function
func <- funC(c(f, f_s))
```

```
# Initialize times, states, parameters and forcings
times <- seq(0, 15, by = .1)
yini <- c(O3 = 0, O2 = 3, O = 2, attr(f_s, "yini"))
pars <- c(build_O3 = .1, decay_O3 = .01)

# Solve ODE
out <- odeC(y = yini, times = times, func = func, parms = pars)

# Plot solution
par(mfcol=c(2,3))
t <- out[,1]
M1 <- out[,2:4]
M2 <- out[,5:7]
M3 <- out[,8:10]
M4 <- out[,11:13]
M5 <- out[,14:16]
M6 <- out[,17:19]

matplot(t, M1, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="solution")
legend("topright", legend = c("O3", "O2", "O"), lty=1, col=1:3)
matplot(t, M2, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O3)")
matplot(t, M3, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O2)")
matplot(t, M4, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O)")
matplot(t, M5, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d build_O3)")
matplot(t, M6, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d decay_O3)")


## End(Not run)
## Not run:

#########################################################################
## Sensitivity analysis of ozone formation with Sundials solver
#########################################################################


# O2 + O <-> O3
f <- c(
  O3 = " build_O3 * O2 * O - decay_O3 * O3",
  O2 = "-build_O3 * O2 * O + decay_O3 * O3",
  O  = "-build_O3 * O2 * O + decay_O3 * O3"
)

# Generate ODE function
func <- funC(f, solver = "Sundials", modelname = "ozon")

# Change attributes of func to cause the solver evaluate sensitivities
extended <- func
```

```
attr(extended, "deriv") <- TRUE
attr(extended, "variables") <- c(
  attr(func, "variables"),
  attr(func, "variablesSens"))

# Initialize times, states, parameters and forcings
times <- seq(0, 15, by = .1)
yini <- c(O3 = 0, O2 = 3, O = 2)
pars <- c(build_O3 = .1, decay_O3 = .01)


# Solve ODE without sensitivities
out <- odeC(yini, times, func, pars, method = "bdf")
# Solve ODE with sensitivities
out <- odeC(yini, times, extended, pars, method = "bdf")

# Plot solution
par(mfcol=c(2,3))
t <- out[,1]
M1 <- out[,2:4]
M2 <- out[,5:7]
M3 <- out[,8:10]
M4 <- out[,11:13]
M5 <- out[,14:16]
M6 <- out[,17:19]

matplot(t, M1, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="solution")
legend("topright", legend = c("O3", "O2", "O"), lty=1, col=1:3)
matplot(t, M2, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O3)")
matplot(t, M3, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O2)")
matplot(t, M4, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d O)")
matplot(t, M5, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d build_O3)")
matplot(t, M6, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="d/(d decay_O3)")


## End(Not run)
## Not run:

##############################################################################
## Estimate parameter values from experimental data
##############################################################################

library(deSolve)

# O2 + O <-> O3
# diff = O2 - O3
# build_O3 = const.
```

```
f <- c(
  O3 = " build_O3 * O2 * O - decay_O3 * O3",
  O2 = "-build_O3 * O2 * O + decay_O3 * O3",
  O  = "-build_O3 * O2 * O + decay_O3 * O3"
)

# Compute sensitivity equations and get attributes
f_s <- sensitivitiesSymb(f)
chi <- attr(f_s, "chi")
grad <- attr(f_s, "grad")
forcings <- attr(f_s, "forcings")

# Generate ODE function
func <- funC(f = c(f, f_s, chi, grad), forcings = forcings,
             fcontrol = "nospline", modelname = "example3")

# Initialize times, states, parameters
times <- seq(0, 15, by = .1)
yini <- c(O3 = 0, O2 = 2, O = 2.5)
yini_s <- attr(f_s, "yini")
yini_chi <- c(chi = 0)
yini_grad <- rep(0, length(grad)); names(yini_grad) <- names(grad)
pars <- c(build_O3 = .2, decay_O3 = .1)

# Initialize forcings (the data)
data(oxygenData)
forcData <- data.frame(time = oxygenData[,1],
                       name = rep(
                         colnames(oxygenData[,-1]),
                         each=dim(oxygenData)[1]),
                       value = as.vector(oxygenData[,-1]))
forc <- setForcings(func, forcData)

# Solve ODE
out <- odeC(y = c(yini, yini_s, yini_chi, yini_grad),
            times = times, func = func, parms = pars, forcings = forc,
            method = "lsodes")

# Plot solution
par(mfcol=c(1,2))
t <- out[,1]
M1 <- out[,2:4]
M2 <- out[,names(grad)]
tD <- oxygenData[,1]
M1D <- oxygenData[,2:4]

matplot(t, M1, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="states")
matplot(tD, M1D, type="b", lty=2, col=1:3, pch=4, add=TRUE)
legend("topright", legend = names(f), lty=1, col=1:3)
matplot(t, M2, type="l", lty=1, col=1:5,
        xlab="time", ylab="value", main="gradient")
legend("topleft", legend = names(grad), lty=1, col=1:5)
```

```
# Define objective function
obj <- function(p) {
  out <- odeC(y = c(p[names(f)], yini_s, yini_chi, yini_grad),
              times = times, func = func, parms = p[names(pars)],
      forcings = forc, method="lsodes")

  value <- as.vector(tail(out, 1)[,"chi"])
  gradient <- as.vector(
    tail(out, 1)[,paste("chi", names(p), sep=".")])
  hessian <- gradient%*%t(gradient)

  return(list(value = value, gradient = gradient, hessian = hessian))
}

# Fit the data
myfit <- optim(par = c(yini, pars),
               fn = function(p) obj(p)$value,
               gr = function(p) obj(p)$gradient,
               method = "L-BFGS-B",
               lower=0,
               upper=5)

# Model prediction for fit parameters
prediction <- odeC(y = c(myfit$par[1:3], yini_s, yini_chi, yini_grad),
                   times = times, func = func, parms = myfit$par[4:5],
   forcings = forc, method = "lsodes")

# Plot solution
par(mfcol=c(1,2))
t <- prediction[,1]
M1 <- prediction[,2:4]
M2 <- prediction[,names(grad)]
tD <- oxygenData[,1]
M1D <- oxygenData[,2:4]

matplot(t, M1, type="l", lty=1, col=1:3,
        xlab="time", ylab="value", main="states")
matplot(tD, M1D, type="b", lty=2, col=1:3, pch=4, add=TRUE)
legend("topright", legend = names(f), lty=1, col=1:3)
matplot(t, M2, type="l", lty=1, col=1:5,
        xlab="time", ylab="value", main="gradient")
legend("topleft", legend = names(grad), lty=1, col=1:5)


## End(Not run)
```

`setForcings` *Generate interpolation spline for the forcings and write into list of matrices*

Description

Generate interpolation spline for the forcings and write into list of matrices

Usage

```
setForcings(func, forcings)
```

Arguments

func	result from funC()
forcings	data.frame with columns name (factor), time (numeric) and value (numeric)

Details

Splines are generated for each name in forcings and both, function value and first derivative are evaluated at the time points of the data frame.

Value

list of matrices with time points and values assigned to the forcings interface of deSolve

Examples

```
f <- c(x = "-k*x + a - b")
func <- funC(f, forcings = c("a", "b"))
forcData <- rbind(
  data.frame(name = "a", time = c(0, 1, 10), value = c(0, 5, 2)),
  data.frame(name = "b", time = c(0, 5, 10), value = c(1, 3, 6)))
forc <- setForcings(func, forcData)
```

---

sumSymb *Compute matrix sumSymbolically*

Description

Compute matrix sumSymbolically

Usage

```
sumSymb(M, N)
```

Arguments

M	matrix of type character
N	matrix of type character

Value

Matrix of type character, the matrix sum of M and N

sundialsIncludes *Infos and includes for sundials::cvodes C++ code: ODE, Jacobian.*

Description

Infos and includes for sundials::cvodes C++ code: ODE, Jacobian.

Usage

```
sundialsIncludes()
```

Value

C++ source code as a character vector.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

sundialsJac *Implement the Jacobian of the ODE system for sundials::cvodes.*

Description

Implement the Jacobian of the ODE system for sundials::cvodes.

Usage

```
sundialsJac(f, variables, parameters, modelname)
```

Arguments

f	Named character vector containing the right-hand sides of the ODE. You may use the key word.
variables	Variables appearing on the ODE, `names(f)`.
parameters	Parameters appearing in the ODE.
modelname	Base name of the dll.

Value

C++ source code as a character vector.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

sundialsOde *Implement the ODE system for sundials::cvodes.*

Description

Implement the ODE system for sundials::cvodes.

Usage

```
sundialsOde(f, variables, parameters, modelname)
```

Arguments

f	Named character vector containing the right-hand sides of the ODE. You may use the key word.
variables	Variables appearing on the ODE, names(f).
parameters	Parameters appearing in the ODE.
modelname	Base name of the dll.

Value

C++ source code as a character vector.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

sundialsSensOde *Implement sensitivities for an ODE system, sundials::cvodes.*

Description

Implement sensitivities for an ODE system, sundials::cvodes.

Usage

```
sundialsSensOde(f, variablesOde, variablesSens, parameters, modelname)
```

Arguments

f	Named character vector containing the right-hand sides of the ODE. You may use the key word.
variablesOde	Variables appearing on the ODE of the dynamic system.
variablesSens	Variables appearing on the ODE of the sensitivities of the dynamic system.
parameters	Parameters appearing in the ODE.
modelname	Base name of the dll.

Value

C++ source code as a character vector.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`wrap_cvodes` *Solve an inital value problem with cvodes.*

Description

Wrapper around the solver cvodes from the Sundials suite.

Usage

```
wrap_cvodes(times, states_, parameters_, initSens_, events_, settings, model_,
  jacobian_, sens_)
```

Arguments

`times`	Numeric vector of time points at which integration results are returned.
`states_`	Numeric vector of inital values for states.
`parameters_`	Numeric vector of model parameters values.
`initSens_`	Numeric vector of inital values for sensitivities.
`events_`	Data.frame with columns `var` (index of the ODE state in the vector of states.), `time` (time point of the event), `method` (one of "replace", "add", or "multiply"), `value` (value associated with the event).

`settings`	List of setting passed to cvodes. For a detailed documentation of the supported setting please check the Sundials homepage. Supported settings are
	`"jacobian"`**, bool.** For `"jacobian" = TRUE`, a function returning the Jacobian matrix of the system must be provided by '`jacobian_`'.
	`"method"`**, string, can be** `"bdf"` **or** `"adams"`**.** The integration method used. For "bdf" `CVodeCreate(CV_BDF, CV_NEWTON)` is called, for "adams" `CVodeCreate(CV_ADAMS, CV_NEW`
	`"atol"`**, a scalar or a vector.** Specifies the absolute integration tolerance. If "atol" is scalar, each state is integrated with the same absolute tolerance. If the absolute error tolerance needs to be different for each state, "atol" can be a vector holding the tolerance for each state.
	`"rtol"`**, scalar.** Relative integration error tolerance.
	`"which_states"`**, vector.** Return the first `"which_states"`. If the model has `N` states, `which_states <= N` allows to dicard all states `> which_states`
	`"which_observed"`**, vector.** Same as `"which_states"`, but for observables.
	`"maxsteps" = 500`**, scalar.** Maximum number of internal steps allowed to reach the next output time. While not recommended, this test can be disabled by passing `"maxsteps" < 0`.

"maxord" = 12 (adams) or 5 (bdf), **scalar.** Maximum order of the linear multistep method. Can only be set to smaller values than default.

"hini" = "estimated", **scalar.** Inital step size.

"hmin" = 0.0, **scalar.** Minimum absolute step size.

"hmax" = infinity, **scalar.** Maximum absolute step size.

"maxerr" **= 7, scalar.** Permitted maximum number of failed error test per step.

"maxnonlin" = 3, **scalar.** Permitted nonlinear solver iterations per step.

"maxconvfail" = 10, **scalar.** Permitted convergence failures of the nonlinear solver per step.

"stability" **= FALSE, bool.** Stability limit detection for the "bdf" method.

"positive", **bool.** Issue an error (and abort?) in case a state becomes smaller than '"minimum"'.

"minimum", **scalar.** Lower bound below which a state is assumed negative and reported, in case '\code{"positive" = TRUE}'.

"sensitivities" **= FALSE, bool.** Integrate sensitivities of the dynamic system.

model_ The address of the ode model. The address is obtained as the attribute address of getNativeSymbolInfo. The signature of the model function must comply to

```
std::array<std::vector<double>, 2> (const double& t, const std::vector<double>&
states,   const std::vector<double>& parameters, const std::vector<double>& forcings)
```

Return vector std::array<std::vector<double>, 2>

1. First dimension holds the increments for all states.
2. Second dimension holds the observed state. Not sure what these are.

Argument list (const double& t, const std::vector<double>& states, const std::vector<d

t Most probably the requested time point, but I am not totally sure.

states Vector of current state values.

parameters Vector of parameters values.

forcings Vector of forcings acting on the model.

jacobian_ The address of the function which returns the Jacobian matrix of the model. Again, this address is the attribute address obtained from the call to getNativeSymbolInfo. The function must have the signature arma::mat (const double& t, const std::vector<double>& s
std::vector<double>& parameters, const std::vector<double>& forcings)
Returned is the Jacobian matrix as an arma::mat from the Armadillo package.

The list of arguments is the same as for 'model_'.

sens_ The address of the function which returns the right-hand side of the sensitivity equations. Again, this address is the attribute address obtained from the call to getNativeSymbolInfo. The list of arguments is the same as for 'model_'.

Details

This function sets up the cvodes integrator and loop over the vector 'times' of requested time points. On success, the states of the system are returend for these time points.

Write something about these observations, once you got a hold of them.

Value

Matrix with nrow = (no. timepoints) and ncol = (no. states + no. observed + [(no. states)x(no. parameters)]). [(no. states)x(no. parameters)] is only returned if sensitivity equations are calculated.

First column Integration time points as given in '`times`'.

Column 2 to no. of states + 1 The state for the respective time point.

no. of states + 1 to number of states + 1 + n. of observations Observation for the respective time point.

Author(s)

Alejandro Morales, <morales.s.alejandro@gmail.com>

Index

∗Topic **package**
cOde-package, 2

adjointSymb, 3, *16*

bvptwpC, 5

cOde *(*cOde-package*)*, 2
cOde-package, 2
compileAndLoad, 7
cvodesSyntax, 8

forcData, 8
funC, 9

getNativeSymbolInfo, *25*
getSymbols, 10

jacobianSymb, 11

lsoda, *9*
lsode, *9*
lsodes, *9*

odeC, 11
oxygenData, 13

prodSymb, 13

reduceSensitivities, 14
replaceOperation, 14
replaceSymbols, 15

sensitivitiesSymb, 15
setForcings, 20
sumSymb, 21
sundialsIncludes, 22
sundialsJac, 22
sundialsOde, 23
sundialsSensOde, 23

wrap_cvodes, 24