Package ‘rodeo’

December 15, 2023

Type Package

NeedsCompilation yes

Title A Code Generator for ODE-Based Models

Version 0.7.8

Date 2023-12-15

Author David Kneis <david.kneis@tu-dresden.de>

Maintainer David Kneis <david.kneis@tu-dresden.de>

Description Provides an R6 class and several utility methods to facilitate the implementation of models based on ordinary differential equations. The heart of the package is a code generator that creates compiled 'Fortran' (or 'R') code which can be passed to a numerical solver. There is direct support for solvers contained in packages 'deSolve' and 'rootSolve'.

URL https://github.com/dkneis/rodeo

License GPL (>= 2)

Imports R6, deSolve

VignetteBuilder knitr

Suggests knitr, rmarkdown, xtable, rootSolve

SystemRequirements The tools to run 'R CMD SHLIB' on 'Fortran' code. The used 'Fortran' compiler must support pointer initialization which is a feature of the 2008 standard. The compiler from Oracle Developer Studio 12.6 on Solaris 10 currently does not meet this requirement.

RoxygenNote 7.2.3

Repository CRAN

Date/Publication 2023-12-15 13:20:02 UTC

R topics documented:

`rodeo-package` *Package to Facilitate ODE-Based Modeling*

Description

This package provides methods to

- import a conceptual ODE-based model stored in tabular form (i.e. as text files or spreadsheets).
- generate source code (either R or Fortran) to be passed to an ODE-solver.
- visualize and export basic information about a model, e.g. for documentation purposes.

Details

Consult the package vignette for details. The concept of writing an ODE system in tabular/matrix form is nicely introduced, e. g., in the book of Reichert, P., Borchardt, D., Henze, M., Rauch, W., Shanahan, P., Somlyody, L., and Vanrolleghem, P. A. (2001): River water quality model No. 1, IWA publishing, ISBN 9781900222822.

The current source code repository is `https://github.com/dkneis/rodeo`.

Class and class methods

See rodeo-class for the rodeo class and the corresponding class methods.

Non-class methods

Type help(package="rodeo") or see the links below to access the documentation of non-class methods contained in the package.

- forcingFunctions Generation of forcing functions in Fortran.
- exportDF Export of data frames as TEX or HTML code.
- stoiCreate Creates a stoichiometry matrix from a set of chemical reactions.
- stoiCheck Validates a stoichiometry matrix by checking for conservation of mass.

Author(s)

<david.kneis@tu-dresden.de>

---

compile *Generate Executable Code*

Description

Creates and 'compiles' a function for use with numerical methods from package deSolve or rootSolve.

Arguments

sources	Name(s) of source files(s) where functions appearing in process rates or stoichiometric factors are implemented. Can be NULL if no external functions are required, the name of a single file, or a vector of file names. See notes below.
fortran	If TRUE, Fortran code is generated and compiled into a shared library. If FALSE, R code is generated.
target	Name of a 'target environment'. Currently, 'deSolve' is the only supported value.
lib	File path to be used for the generated library (without the platform specific extension). Note that any uppercase characters will be converted to lowercase. By default, the file is created in R's temporary folder under a random name.
reuse	If TRUE, an already existing library file will be loaded. Use this to prevent unnecessary re-compilation but note that R is likely to crash in case of any mismatches between the object and the existing library. Default is FALSE, i.e. the library is unconditionally build from scratch.

Value

invisible(NULL)

Note

The expected language of the external code passed in `sources` depends on the value of `fortran`.

If `fortran` is `FALSE`, R code is generated and made executable by `eval` and `parse`. Auxiliary code passed via `sources` is made available via `source`. The created R function is stored in the object.

If `fortran` is `TRUE`, the external code passed in `sources` must implement a module with the fixed name 'functions'. This module must contain all user-defined functions referenced in process rates or stoichiometric factors.

If `fortran` is `TRUE`, a shared library is created. The library is immediately loaded with `dyn.load` and it is automatically unloaded with `dyn.unload` when the object's `finalize` method is called.

The name of the library (base name without extension) as well as the name of the function to compute the derivatives are stored in the object. These names can be queried with the `libName` and `libFunc` methods, respectively. Unless a file path is specified via the `lib` argument, the library is created in the folder returned by `tempdir` under a unique random name.

Author(s)

<david.kneis@tu-dresden.de>

See Also

This method internally calls `generate`.

Examples

```
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
# This would trigger compilation assuming that 'functionsCode.f95' contains
# a Fortran implementation of all functions; see vignette for full example
## Not run:
model$compile(sources="functionsCode.f95")

## End(Not run)
```

---

dynamics *Numerical Integration*

Description

Compute a dynamic solution with the numerical algorithms from package `deSolve`.

Arguments

`times`	Times of interest (numeric vector).
`fortran`	Switch between compiled Fortran and R code (logical). Default is `TRUE`.
`proNames`	Assign names to output columns holding the process rates? Default is `TRUE`.
`...`	Auxiliary arguments passed to `ode`. See notes below.

Value

The matrix returned by the integrator (see `ode`).

Note

This method can only be used after `compile` has been called.

The `...` argument should *not* be used to assign values to any of `y`, `parms`, `times`, `func`. If `fortran` is `TRUE` it should also not assign values to `dllname`, `nout`, or `outnames`. All these arguments of `ode` get their appropriate values automatically.

Author(s)

<david.kneis@tu-dresden.de>

See Also

Use `step` for integration over a single time step with a built-in, Fortran-based solver.

---

exportDF — *Export a Data Frame as HTML/TEX Code*

Description

Generates code to include tabular data in a tex document or web site.

Usage

```
exportDF(
  x,
  tex = FALSE,
  colnames = NULL,
  width = NULL,
  align = NULL,
  funHead = NULL,
  funCell = NULL,
  lines = TRUE,
  indent = 2
)
```

Arguments

`x`	The data frame being exported.
`tex`	Logical. Allows to switch between generation of TEX code and HTML.
`colnames`	Displayed column names. If `NULL`, the original names of `x` are used. Otherwise it must be a named vector with element names corresponding to column names in `x`. It is OK to supply alternative names for selected columns only.

width
: Either NULL (all columns get equal width) or a named vector with element names corresponding to column names in x. If tex == TRUE, values (between 0 and 1) are needed for columns with align code 'p' only. They are interpreted as a multiplier for '\textwidth'. If tex == FALSE, values (between 0 and 100) should be supplied for all columns of x.

align
: Either NULL (to use automatic alignment) or a named vector with element names corresponding to column names in x. If tex == FALSE valid alignment codes are 'left', 'right', 'center'. If tex == TRUE valid alignment codes are 'l', 'r', 'c', and 'p'. For columns with code 'p' a corresponding value of width should be set. It is OK to supply alignment codes for selected columns only.

funHead
: Either NULL or a list of functions whose names correspond to column names of x. The functions should have a single formal argument; the respective column names of x are used as the actual arguments. It is OK to supply functions for selected columns only (an empty function is applied to the remaining columns). See below for some typical examples.

funCell
: Like funHead but these functions are applied to the cells in columns rather that to the column names.

lines
: Logical. Switches table borders on/off.

indent
: Integer. Number of blanks used to indent the generated code.

Value

A character string (usually needs to be exported to a file).

Note

The functions funHead and funCell are useful to apply formatting or character replacement. For example, one could use

`function(x) {paste0("\\bold{",toupper(x),"}")}`

to generate bold, uppercase column names in a TEX table.

Author(s)

David Kneis <david.kneis@tu-dresden.de>

See Also

The xtable packages provides similar functionality with more sophisticated options. Consider the 'pandoc' software do convert documents from one markup language to another one. Finally, consider the latex package 'datatools' for direct inclusion of delimited text files (e.g. produced by write.table) in tex documents.

Examples

```
# Create example table
df <- data.frame(stringsAsFactors=FALSE, name= c("growth", "dead"),
  unit= c("1/d","1/d"), expression= c("r * N * (1 - N/K)"," d * N"))
```

```
# Export as TEX: header in bold, 1st colum in italics, last column as math
tex <- exportDF(df, tex=TRUE,
  colnames=c(expression="process rate expression"),
  width=c(expression=0.5),
  align=c(expression="p"),
  funHead=setNames(replicate(ncol(df),
    function(x){paste0("\\textbf{",x,"}")}),names(df)),
  funCell=c(name=function(x){paste0("\\textit{",x,"}")},
    expression=function(x){paste0("$",x,"$")})
)
cat(tex,"\n")

# Export as HTML: non-standard colors are used for all columns
tf <- tempfile(fileext=".html")
write(x= exportDF(df, tex=FALSE,
  funHead=setNames(replicate(ncol(df),
    function(x){paste0("<font color='red'>",x,"</font>")}),names(df)),
  funCell=setNames(replicate(ncol(df),
    function(x){paste0("<font color='blue'>",x,"</font>")}),names(df))
), file=tf)
## Not run:
  browseURL(tf)
  file.remove(tf)

## End(Not run)
```

`finalize` *Clean-up a* `rodeo` *Object*

Description

Clean-up method for objects of the `rodeo-class`.

Value

The method is called implicitly for its side effects when a `rodeo` object is destroyed.

Note

At present, the method just unloads the object-specific shared libraries created with the `compile` or `initStepper` methods.

Author(s)

`<david.kneis@tu-dresden.de>`

forcingFunctions *Generation of Forcing Functions in Fortran*

Description

Generates Fortran code to return the current values of forcing functions based on interpolation in tabulated time series data.

Usage

```
forcingFunctions(x)
```

Arguments

`x`	Data frame with colums 'name', 'file', 'column', 'mode', 'default'. See below for expected entries.

Value

A character string holding generated Fortran code. Must be written to disk, e.g. using `write`, prior to compilation.

Note

The fields of the input data frame are interpreted as follows:

- `name` Name of the forcing function as declared in the table of functions.
- `file` Name of the text file containing the time series data either as an absolute or relative path. Time information is expected as numeric values in the first column (e.g. as number of seconds after some reference date). The period is used as the decimal character in floating point numbers, numeric values can also be given in scientific format (e.g. as 0.314e+1). Allowed column delimiters are blank, tab, or comma. A sequence of white spaces collapses to a single delimiter but this is not the case for commas. It is strictly recommended to use a consistent delimiter character within a particular file. Blank lines are allowed everywhere in the file, comment lines must start with a '#'. The first non-blank, non-comment line is interpreted as column headers and the name of the first column (holding time info) is essentially ignored).
- `column` Name of the column in `file` from which data are to be read.
- `mode` Integer code to control how the interpolation is performed. Use 0 for constant interpolation with full weight given to the value at the end of a time interval. Use 1 for constant interpolation with full weight given to the value at the begin of a time interval. Any other values (< 0 or > 1) result in linear interpolation with weights being set automatically.
- `default` Logical. If `FALSE`, the generated function has the interface 'f(time)'. If `TRUE`, the generated function has a two-argument interface 'f(time, z)'. If the actual argument 'z' is `NaN`, the function behaves just like the single-argument version, i.e. interpolation in tabulated data is performed. If 'z' is not `NaN`, the function returns the value of 'z'.

The generated code provides a single module named 'forcings' which defines as many forcing functions as there are rows in x. The module 'forcings' needs to be made available to the compiler (either at the command line or via inclusion in another file with Fortran's include mechanism). In addition, it must be referenced in the module 'functions' with an appropriate 'use' statement (see example below).

The generated function return scalar values of type double precision. If an error condition is encountered, the return value of a functions equals the largest possible double precision value (generated by Fortran's 'huge' function). In addition, errors trigger calls of the subroutines 'rexit' (at top level) or 'rwarn' (at lower levels). These two functions are available automatically if the Fortran code is compiled using 'R CMD SHLIB'. Otherwise, the two functions need to be defined (see examples below).

In the two-argument version, the second argument is tested against NaN using 'ISNAN'. This function is not part of the Fortran standard but it is supported by most compilers, including gfortan. The Fortran 2003 standard conformal function would be 'IS_IEEE_NAN' which is not yet supported by compiler versions normally installed with R (March 2016).

Author(s)

David Kneis <david.kneis@tu-dresden.de>

Examples

```
## Not run: 
  ! Example of a Fortran file to define functions
  include 'forcings.f95'   ! include generated forcings file in compilation
  module functions
  use forcings             ! make forcings available as functions
  implicit none
  contains
  ! ... any non-forcing functions go here ...
  end module

## End(Not run)

## Not run: 
  ! Definition of 'rexit' and 'rwarn' for testing of the generated code
  ! outside of R
  subroutine rexit (x)
    character(len=*), intent(in):: x
    write(*,*) "ERROR: ",trim(adjustl(x))
    stop 1
  end subroutine

  subroutine rwarn (x)
    character(len=*), intent(in):: x
    write(*,*) "WARNING: ",trim(adjustl(x))
  end subroutine

## End(Not run)
```

funs — *Declaration of Functions*

Description

Declaration of functions referenced at the ODE's right hand sides of the bacteria growth example model.

Format

A data frame with the following fields:

name : Name of the function

unit : Unit of the return value

description : Short description (text)

generate — *Code Generator*

Description

This is a low-level method to translate the ODE-model specification into a function that computes process rates and the state variables derivatives (either in R or Fortran). You probably want to use the high-level method `compile` instead, which uses `generate` internally.

Arguments

`lang`	Character string to select the language of the generated source code. Currently either 'f95' (for Fortran) or 'r' (for R).
`name`	Name for the generated function (character string). It should start with a letter, optionally followed by letters, numbers, or underscores.

Value

The generated source code as a string. Must be written to disk, e.g. using `write`, prior to compilation.

Note

Details of this low-level method may change in the future.

Author(s)

<david.kneis@tu-dresden.de>

See Also

See other methods of the rodeo-class, especially compile which internally uses this method.

Examples

```
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
fortranCode <- model$generate(lang="f95")
## Not run:
write(fortranCode, file="")

## End(Not run)
```

---

getPars *Query Values of Parameters*

Description

Query values of parameters of a rodeo-based model.

Arguments

asArray	Logical. If FALSE, the values of parameters are returned as vector irrespective of the model's spatial resolution. If TRUE, the values are returned as an array with properly named dimensions. The array's last dimension represents the parameters and its first (fastest cycling) dimension, if any, refers to the model's highest spatial dimension.
useNames	Logical. Used to enable/disable element names for the return vector when asArray is FALSE. The names follow the pattern 'x.i.j' where 'x' is the parameter name and 'i', 'j' are indices of the sub-units in the first and second spatial dimension. The actual suffix is controlled by the number of dimensions and in the 0-dimensional case, no suffix is applied at all, i.e. the pure parameter names are used to label the elements of the vector. If isArray is TRUE, this argument is simply ignored, hence the dimensions of a returned array are always named.

Value

A numeric vector or array.

Author(s)

<david.kneis@tu-dresden.de>

See Also

The corresponding 'set' method is setPars and examples can be found there. Use getVars to query the values of variables rather than parameters.

getVars *Query Values of State Variables*

Description

Query values of state variables of a rodeo-based model.

Arguments

asArray	Logical. If FALSE, the values of variables are returned as vector irrespective of the model's spatial resolution. If TRUE, the values are returned as an array with properly named dimensions. The array's last dimension represents the variables and its first (fastest cycling) dimension, if any, refers to the model's highest spatial dimension.
useNames	Logical. Used to enable/disable element names for the return vector when asArray is FALSE. The names follow the pattern 'x.i.j' where 'x' is the variable name and 'i', 'j' are indices of the sub-units in the first and second spatial dimension. The actual suffix is controlled by the number of dimensions and in the 0-dimensional case, no suffix is applied at all, i.e. the pure variable names are used to label the elements of the vector. If isArray is TRUE, this argument is simply ignored, hence the dimensions of a returned array are always named.

Value

A numeric vector or array.

Author(s)

<david.kneis@tu-dresden.de>

See Also

The corresponding 'set' method is setVars and examples can be found there. Use getPars to query the values of parameters rather than variables.

initialize *Initialize a* rodeo *Object*

Description

Initializes an object of the rodeo-class with data frames holding the specification of an ODE system.

Arguments

- `vars` Declaration of state variables appearing in the ODE system. Data frame with mandatory columns 'name', 'unit', 'description'.
- `pars` Declaration of parameters (i.e. constants) appearing in the ODE system. Data frame with the same mandatory columns as `vars`.
- `funs` Declaration of functions being referenced in the ODE system. Data frame with the same mandatory columns as `vars` or `NULL` if no function calls are present at the ODEs' right-hand sides.
- `pros` Declaration of process rates. Data frame with mandatory columns 'name', 'unit', 'description', 'expression'.
- `stoi` Declaration of stoichiometric factors. A data frame with mandatory columns 'variable', 'process', 'expression', if `asMatrix` is `FALSE`. The 'expression' colum holds the stoichiometric factors. If `asMatrix` is `TRUE`, this must be a matrix of type character with row names (processes) and colum names (variables). Empty or `NA` matrix elements are interpreted as zero stoichiometry factors.
- `asMatrix` Logical. Specifies whether stoichiometry information is given in matrix or data base format.
- `dim` An integer vector, specifying the number of boxes in each spatial dimension. Use `c(1)` to create a zero-dimensional (i.e. single-box) model. This is the default. Use, e.g. `c(5)` to create a 1-dimensional model with 5 boxes. To create, e.g., a 2-dimensional model with 4 x 5 boxes, use `c(4,5)`.

Value

The method is called implicitly for its side effects when a `rodeo` object is instantiated with `new`. It has no accessible return value.

Note

The mandatory fields of the input data frames should be of type character. Additional fields may be present in these data frames and the contents becomes part of the `rodeo` object. The 'expression' fields of `pros` and `stoi` (or the contents of the stoichiometry matrix) should be valid mathematical expressions in R and Fortran. These can involve the names of declared state variables, parameters, and functions as well as numeric constants or basic math operators. Branching or loop constructs are not allowed (but these can appear inside referenced functions). There are currently few reserved words that cannot be used as variable, parameter, function, or process names. The reserved words are 'time', 'left', and 'right'.

Initialization does not assign numeric values to state variables or parameters. Use the decicated methods `setVars` and `setPars` for that purpose.

Author(s)

`<david.kneis@tu-dresden.de>`

See Also

See the package vignette for examples.

Examples

```
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
print(model)
```

---

initStepper *Initialize Internal ODE Solver*

Description

Initializes rodeo's built-in ODE solver. This method must be called prior to using step.

Arguments

sources	Name(s) of fortran source file(s) where a module with the fixed name 'functions' is implemented. This module must contain all user-defined functions referenced in process rates or stoichiometric factors. Can be NULL, the name of a single file, or a vector of file names if the Fortran code is split over several files.
method	Name of a the solver. Currently, 'rk5' is the only supported value (Runge-Kutta method of Cash and Karp).

Value

invisible(NULL)

Note

After this method was called, step can be used to perform the integration.

Author(s)

<david.kneis@tu-dresden.de>

See Also

To perform integration with the solvers from package deSolve use compile instead of this method.

libFunc — *Return name of library function*

Description

Return the name of the library function for use with deSolve or rootSolve methods.

Value

The name of the function to compute derivatives which is contained in the library built with compile. This name has to be supplied as the `func` argument of the solver methods in deSolve or rootSolve.

Author(s)

<david.kneis@tu-dresden.de>

libName — *Return library name*

Description

Return the pure name of the shared library for use with deSolve or rootSolve methods.

Value

The base name of the shared library file created with compile after stripping of the the platform specific extension. This name has to be supplied as the `dllname` argument of the solver methods in deSolve or rootSolve.

Author(s)

<david.kneis@tu-dresden.de>

pars — *Declaration of Parameters*

Description

Declaration of parameters of the bacteria growth example model.

Format

A data frame with the following fields:

name : Name of the parameter
unit : Unit
description : Short description (text)

`plotStoichiometry` *Plot Qualitative Stoichiometry Matrix*

Description

Visualizes the stoichiometry matrix using standard plot methods. The sign of stoichiometric factors is displayed as upward and downward pointing triangles. Also visualized are dependencies of process rates on variables.

Arguments

`box`	A positive integer pointing to a spatial sub-unit of the model.
`time`	Time. The value is ignored in the case of autonomous models.
`cex`	Character expansion factor.
`colPositive`	Color for positive stoichiometric factors.
`colNegative`	Color for negative stoichiometric factors.
`colInteract`	Color used to highlight dependencies.
`colBack`	Color of background.
`colGrid`	Color of a grid.
`lwdGrid`	Grid line width.
`translateVars`	Optional function to recode variable labels. Must take the original vector as argument and return the altered version.
`translatePros`	Optional function to recode process labels. Must take the original vector as argument and return the altered version.

Note

The values of state variables and parameters must have been set using the `setVars` and `setPars` methods. If the stoichiometric factors are mathematical expressions involving function references, these functions must be defined in R (even if the numerical computations are based on generated Fortran code).

Author(s)

`<david.kneis@tu-dresden.de>`

See Also

See other methods of the `rodeo-class` or `stoichiometry` for computing the stoichiometric factors only. Alternative options for displaying stoichiometry information are described in the package vignette.

Examples

```
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
model$setVars(c(bac=0.1, sub=0.5))
model$setPars(c(mu=0.8, half=0.1, yield= 0.1, vol=1000, flow=50, sub_in=1))
monod <- function(c,h) {c/(c+h)}
model$plotStoichiometry(box=c(1))
```

`pros` *Declaration of Processes*

Description

Definition of processes of the bacteria growth example model.

Format

A data frame with the following fields:

name : Name of the process

unit : Unit of the rate expression

description : Short description (text)

expression : Mathematical expression (as a string)

`rodeo-class` `rodeo` *Class*

Description

This documents the `rodeo` class to represent an ODE-based model. See the `rodeo-package` main page or type `help(package="rodeo")` for an introduction to the package of the same name.

Fields

`prosTbl` A data frame with fields 'name', 'unit', 'description', and 'expression' defining the process rates.

`stoiTbl` A data frame with fields 'variable', 'process', and 'expression' reprenting the stoichiometry matrix in data base format.

`varsTbl` A data frame with fields 'name', 'unit', 'description' declaring the state variables of the model. The declared names become valid identifiers to be used in the expression fields of `prosTbl` or `stoiTbl`.

`parsTbl` A data frame of the same structure as `vars` declaring the parameters of the model. The declared names become valid identifiers to be used in the expression fields of `prosTbl` or `stoiTbl`.

funsTbl A data frame of the same structure as vars declaring any functions referenced in the expression fields of prosTbl or stoiTbl.

dim Integer vector specifying the spatial dimensions.

vars Numeric vector, holding values of state variables.

pars Numeric vector, holding values of parameters.

Class methods

For most of the methods below, a separate help page is available describing its arguments and usage.

- initialize Initialization method for new objects.
- namesVars, namesPars, namesFuns, namesPros Functions returning the names of declared state variables, parameters, functions, and processes, respectively (character vectors). No arguments.
- lenVars, lenPars, lenFuns, lenPros Functions returning the number of declared state variables, parameters, functions and processes, respectively (integer). No arguments.
- getVarsTable, getParsTable, getFunsTable, getProsTable, getStoiTable Functions returning the data frames used to initialize the object. No arguments
- getDim Returns the spatial dimensions as an integer vector. No arguments.
- compile Compiles a Fortran library for use with numerical methods from packages deSolve or rootSolve.
- generate Translate the ODE-model specification into a function that computes process rates and the state variables' derivatives (either in R or Fortran). Consider to use the high-level method compile.
- setVars Assign values to state variables.
- setPars Assign values to parameters.
- getVars Returns the values of state variables.
- getPars Returns the values of parameters.
- stoichiometry Returns the stoichiometry matrix, either evaluated (numeric) or as text.
- plotStoichiometry Plots qualitative stoichiometry information.

See Also

See the rodeo-package main page or type help(package="rodeo") to find the documentation of any non-class methods contained in the rodeo package.

Examples

```
# Bacteria growth in a continuous flow culture
library("deSolve")

# Creation of model object
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
```

```
# Parameters, initial values
model$setPars(c(mu=0.8, half=0.1, yield= 0.1, vol=1000, flow=50, sub_in=1))
model$setVars(c(bac=0.01, sub=0))

# Implementation of functions declared in 'funs'
monod <- function(c,h) {c/(c+h)}

# Creation of derivatives function
code <- model$generate(name="derivs", lang="r")
derivs <- eval(parse(text=code))

# Integration
times <- 0:96
out <- deSolve::ode(y=model$getVars(), times=times, func=derivs,
  parms=model$getPars())
colnames(out) <- c("time", model$namesVars(), model$namesPros())
plot(out)
```

---

setPars *Assign Values to Parameters*

Description

Assign values to parameters of a rodeo-based model.

Arguments

x	A numeric vector or array, depending on the model's spatial dimensions. Consult the help page of the sister method setVars for details on the required input.

Value

NULL (invisible). The assigned numeric data are stored in the object and can be accessed by the getPars method.

Note

Look at the notes and examples for the setVars method.

Author(s)

<david.kneis@tu-dresden.de>

See Also

The corresponding 'get' method is getPars. Use setVars to assign values to variables rather than parameters. Consult the help page of the latter function for examples.

setVars *Assign Values to State Variables*

Description

Assign values to state variables of a rodeo-based model.

Arguments

x	A numeric vector or array, depending on the model's spatial dimensions. See details below.

Value

NULL (invisible). The assigned numeric data are stored in the object and can be accessed by the getVars method.

Note

For a 0-dimensional model (i.e. a model without spatial resolution), x must be a numeric vector whose length equals the number of state variables. The element names of x must match those returned by the object's namesVars method. See the examples for how to bring the vector elements into required order.

For models with a spatial resolution, x must be a numeric array of proper dimensions. The last dimension (cycling slowest) corresponds to the variables and the first dimension (cycling fastest) corresponds to the models' highest spatial dimension. Thus, dim(x) must be equal to c(rev(obj$getDim()), obj$namesVars()) where obj is the object whose variables are to be assigned. The names of the array's last dimension must match the return value of obj$namesVars().

In the common 1-dimensional case, this just means that x must be a matrix with column names matching the return value of obj$namesVars() and as many rows as given by obj$getDim().

Author(s)

<david.kneis@tu-dresden.de>

See Also

The corresponding 'get' method is getVars. Use setPars to assign values to parameters rather than variables.

Examples

```
data(vars, pars, funs, pros, stoi)
x0 <- c(bac=0.1, sub=0.5)

# 0-dimensional model
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
model$setVars(x0)
```

```
print(model$getVars())

# How to sort vector elements
x0 <- c(sub=0.5, bac=0.1)             # doesn't match order of variables
model$setVars(x0[model$namesVars()])

# 1-dimensional model with 3 boxes
nBox <- 3
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(nBox))
x1 <- matrix(rep(x0, each=nBox), nrow=nBox, ncol=model$lenVars(),
  dimnames=list(NULL, model$namesVars()))
model$setVars(x1)
print(model$getVars())
print(model$getVars(asArray=TRUE))

# 2-dimensional model with 3 x 4 boxes
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(3,4))
x2 <- array(rep(x0, each=3*4), dim=c(4,3,model$lenVars()),
  dimnames=list(dim2=NULL, dim1=NULL, variables=model$namesVars()))
model$setVars(x2)
print(model$getVars())
print(model$getVars(asArray=TRUE))
```

---

step *Numerical Integration Over a Single Time Step*

Description

Performs integration over a single time step using a built-in ODE solver. At present, a single solver is implement with limited options. The interface of this method may change when support for other solvers is added.

Arguments

`t0`	Numeric. Initial time.
`h`	Numeric. Length of time step of interest.
`hmin`	Minimum tolerated internal step size. The default of `NULL` sets this to 10 times the value of `.Machine$double.eps`.
`maxsteps`	Maximum tolerated number of sub-steps.
`tol`	Numeric. Relative accuracy requested. This is currently a global value, i.e. one cannot set the accuracy per state variable.
`method`	String. Currently, 'rk5' is the only method implemented. This is a Runge-Kutta Cash-Karp solver adapted from Press et al. (2002), Numerical recipes in Fortran 90. It is designed to handle non-stiff problems only.
`check`	Logical. Can be used to avoid repeated checks of arguments. This may increase performance in repeated calls.

Value

A named numeric vector holding the values of state variables and process rates in all boxes.

Note

This method can only be used after a call to `initStepper` has been made.

Author(s)

`<david.kneis@tu-dresden.de>`

See Also

Use `deSolve` for advanced solvers with more options and capabilities to handle stiff problems.

`stoi` *Specification of Stoichiometry*

Description

Definition of the links between simulated processes and state variables in the bacteria growth example model.

Format

A data frame with the following fields:

variable : Name of the state variable

process : Name of the process

expression : Mathematical expression (as a string)

`stoiCheck` *Validation of a Stoichiometry Matrix*

Description

Validates the stoichiometry matrix by checking for conservation of mass (more typically conservation of moles).

Usage

```
stoiCheck(stoi, comp, env = globalenv(), zero = .Machine$double.eps * 2)
```

Arguments

`stoi`	Stoichiometry matrix either in evaluated (`numeric`) or non-evaluated (`character`) form. A suitable matrix can be created with `stoiCreate`, for example.
`comp`	Matrix defining the elemental composition of compounds. Column names of `comp` need to match column names of `stoi` (but additional columns are allowed and columns can be in different order). There must be one row per element whose balance is to be checked and the elements' names must appear as row names. The elements of the matrix specify how much of an element is contained in a certain amount of a compound. Typically, these are molar ratios. If one works with mass ratios (not being a good idea), the information in `stoi` must be based on mass concentrations as well. The elements of `comp` are treated as mathematical expressions. Any variables, functions, or operators needed to evaluate those expressions must be provided by the specified environment `env`.
`env`	An environment or list supplying constants, functions, and operators needed to evaluate expressions in `comp` or `stoi`.
`zero`	A number close to zero. If the absolute result value of a mass balance computation is less than this, the result is set to 0 (exactly).

Value

A numeric matrix with the following properties:

- There is one row for each process, thus the number and names of rows are the same as in `stoi`.
- There is one column per checked element, hence column names are equal to the row names of `comp`.
- A matrix element at position $[i, k]$ represent the mass balance for the process in row i with respect to the element in column k. A value of zero indicates a close balance. Positive (negative) values indicate that mass is gained (lost) in the respective process.

Author(s)

David Kneis <david.kneis@tu-dresden.de>

See Also

Use `stoiCreate` to create a stoichiometry matrix from a set of reactions in common notation.

Examples

```
# Eq. 1 and 2 are from Soetaert et al. (1996), Geochimica et Cosmochimica
# Acta, 60 (6), 1019-1040. 'OM' is organic matter. Constants 'nc' and 'pc'
# represent the nitrogen/carbon and phosphorus/carbon ratio, respectively.
reactions <- c(
  oxicDegrad= "OM + O2 -> CO2 + nc * NH3 + pc * H3PO4 + H2O",
  denitrific= "OM + 0.8*HNO3 -> CO2 + nc*NH3 + 0.4*N2 + pc*H3PO4 + 1.4*H2O",
  dissPhosp1= "H3PO4 <-> H + H2PO4",
  dissPhosp2= "H2PO4 <-> H + HPO4"
)
```

```
# Non-evaluated stoichiometry matrix
stoi <- stoiCreate(reactions, toRight="_f", toLeft="_b")
# Parameters ('nc' and 'pc' according to Redfield ratio)
pars <- list(nc=16/106, pc=1/106)
# Elemental composition
comp <- rbind(
  OM=    c(C=1, N="nc", P="pc", H="2 + 3*nc + 3*pc"),
  O2=    c(C=0, N=0,    P=0,    H=0),
  CO2=   c(C=1, N=0,    P=0,    H=0),
  NH3=   c(C=0, N=1,    P=0,    H=3),
  H3PO4= c(C=0, N=0,    P=1,    H=3),
  H2PO4= c(C=0, N=0,    P=1,    H=2),
  HPO4=  c(C=0, N=0,    P=1,    H=1),
  H2O=   c(C=0, N=0,    P=0,    H=2),
  HNO3=  c(C=0, N=1,    P=0,    H=1),
  N2=    c(C=0, N=2,    P=0,    H=0),
  H=     c(C=0, N=0,    P=0,    H=1)
)
# We need the transposed form
comp <- t(comp)
# Mass balance check
bal <- stoiCheck(stoi, comp=comp, env=pars)
print(bal)
print(colSums(bal) == 0)
```

`stoichiometry` *Return the Stoichiometry Matrix*

Description

Return and optionally evaluate the mathematical expression appearing in the stoichiometry matrix.

Arguments

`box`	Either `NULL` or a vector of positive integers pointing to a spatial sub-unit of the model. If `NULL`, the mathematical expressions appearing in the stoichiometry matrix are not evaluated, hence, they are returned as character strings. If a spatial sub-unit is specified, a numeric matrix is returned. In the latter case, the values of state variables and parameters must have been set using the `setVars` and `setPars` methods.
`time`	Time. The value is ignored in the case of autonomous models.

Value

A matrix of numeric or character type, depending on the value of box.

Note

If the stoichiometric factors are mathematical expressions involving function references, these functions must be defined in R (even if the numerical computations are based on generated Fortran code).

Author(s)

<david.kneis@tu-dresden.de>

See Also

See other methods of the rodeo-class or plotStoichiometry for a graphical representation of the stoichiometric factors only.

Examples

```
data(vars, pars, funs, pros, stoi)
model <- rodeo$new(vars, pars, funs, pros, stoi, dim=c(1))
model$setPars(c(mu=0.8, half=0.1, yield= 0.1, vol=1000, flow=50, sub_in=1))
model$setVars(c(bac=0.1, sub=0.5))
monod <- function(c,h) {c/(c+h)}
print(model$stoichiometry(box=NULL))
print(model$stoichiometry(box=c(1)))
```

---

stoiCreate *Stoichiometry Matrix from Reaction Equations*

Description

Creates a stoichiometry matrix from a set of reaction equations.

Usage

```
stoiCreate(
  reactions,
  eval = FALSE,
  env = globalenv(),
  toRight = "_forward",
  toLeft = "_backward"
)
```

Arguments

reactions	A named vector of character strings, each representing a (chemical) reaction. See syntax details below.
eval	Logical. If FALSE (default), the returned matrix is of type character and any mathematical expressions are returned as text. If TRUE, an attempt is made to return a numeric matrix by evaluating the expression making use env.
env	Only relevant if eval is TRUE. Must be an environment or list supplying constants, functions, and operators needed to evaluate expressions in the generated matrix.

toRight — Only relevant for reversible reactions. The passed character string is appended to the name of the respective element of `reactions` to create a unique name for the forward reaction.

toLeft — Like `toRight`, but this is the suffix for the backward reaction.

Value

A matrix with the following properties:

- The number of columns equals the total number of components present in `reactions`. The components' names are used as column names.
- The number of rows equals the length of `reactions` plus the number of reversible reactions. Thus, a single row is created for each non-reversible reaction but two rows are created for reversible ones. The latter represent the forward and backward reaction (in that order). The row names are constructed from the names of `reactions`, making use of the suffixes `toRight` and `toLeft` in the case of reversible reactions.
- The matrix is filled with the stoichiometric factors extracted from `reactions`. Empty elements are set to zero.
- The type of the matrix (`character` or `numeric`) depends on the value of `eval`.

Note

The syntax rules for reaction equations are as follows (see examples):

- There must be a left hand side and a right hand side. Sides must be separated by one of the arrows '->', '<-', or '<->' with the latter indicating a reversible reaction.
- Names of component(s) must appear at each side of the reaction. These must be legal row/column names in R. If multiple components are consumed or produced, they must be separated by '+'.
- Any stoichiometric factors need to appear before the respective component name using '*' as the separating character. Stoichiometric factors being equal to unity can be omitted.
- A stoichiometric factor is treated as a mathematical expression. In common cases, it is just a numeric constant. However, the expression can also involve references to variables or functions. If such an expression contains math operators '*' or '+' it needs to be enclosed in parenthesis.

Author(s)

David Kneis <david.kneis@tu-dresden.de>

See Also

Use `stoiCheck` to validate the mass balance of the generated matrix.

Examples

```
# EXAMPLE 1: From https://en.wikipedia.org/wiki/Petersen_matrix (July 2016)
#
reactions <- c(
  formS=  "A + 2 * B -> S",
  equiES= "E + S <-> ES",
  decoES= "ES -> E + P"
)
stoi <- stoiCreate(reactions, eval=TRUE, toRight="_f", toLeft="_b")
print(stoi)

# EXAMPLE 2: Decomposition of organic matter (selected equations only)
#
# Eq. 1 and 2 are from Soetaert et al. (1996), Geochimica et Cosmochimica
# Acta, 60 (6), 1019-1040. 'OM' is organic matter. Constants 'nc' and 'pc'
# represent the nitrogen/carbon and phosphorus/carbon ratio, respectively.
reactions <- c(
  oxicDegrad= "OM + O2 -> CO2 + nc * NH3 + pc * H3PO4 + H2O",
  denitrific= "OM + 0.8*HNO3 -> CO2 + nc*NH3 + 0.4*N2 + pc*H3PO4 + 1.4*H2O",
  dissPhosp1= "H3PO4 <-> H + H2PO4",
  dissPhosp2= "H2PO4 <-> H + HPO4"
)
# Non-evaluated matrix
stoi <- stoiCreate(reactions, toRight="_f", toLeft="_b")
print(stoi)
# Evaluated matrix ('nc' and 'pc' according to Redfield ratio)
pars <- list(nc=16/106, pc=1/106)
stoi <- stoiCreate(reactions, eval=TRUE, env=pars, toRight="_f", toLeft="_b")
print(stoi)
```

`vars` *Declaration of Variables*

Description

Declaration of variables of the bacteria growth example model.

Format

A data frame with the following fields:

name : Name of the variable

unit : Unit

description : Short description (text)

Index