Package 'simecol'

October 7, 2021

Version 0.8-14

Title Simulation of Ecological (and Other) Dynamic Systems

Author Thomas Petzoldt [aut, cre] (<https://orcid.org/0000-0002-4951-6468>)

Depends R (>= 3.2), deSolve, methods

Imports graphics, grDevices, stats, utils, minqa

Suggests tcltk, FME, lattice

LazyLoad yes

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

Description An object oriented framework to simulate
ecological (and other) dynamic systems. It can be used for
differential equations, individual-based (or agent-based) and other
models as well. It supports structuring of simulation scenarios (to avoid copy
and paste) and aims to improve readability and re-usability of code.

License GPL (>= 2)

URL http://www.simecol.de/

NeedsCompilation yes

Repository CRAN

Date/Publication 2021-10-07 06:40:02 UTC

R topics documented:

`simecol-package` *Simulation of Ecological (and Other) Dynamic Systems*

Description

An object oriented framework to simulate ecological (and other) dynamic systems. It can be used for differential equations, individual-based (or agent-based) and other models as well. It supports structuring of simulation scenarios (to avoid copy and paste) and aims to improve readability and re-usability of code.

Details

The DESCRIPTION file:

Package:	simecol
Version:	0.8-14
Title:	Simulation of Ecological (and Other) Dynamic Systems
Authors@R:	c(person("Thomas","Petzoldt", role = c("aut", "cre"), email = "thomas.petzoldt@tu-dresden.de", comment = c(
Author:	Thomas Petzoldt [aut, cre] (<https://orcid.org/0000-0002-4951-6468>)
Depends:	R (>= 3.2), deSolve, methods
Imports:	graphics, grDevices, stats, utils, minqa

Suggests:	tcltk, FME, lattice
LazyLoad:	yes
Maintainer:	Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>
Description:	An object oriented framework to simulate ecological (and other) dynamic systems. It can be used for differenti
License:	GPL (>= 2)
URL:	http://www.simecol.de/

The **simecol** package is intended to give users (scientists and students) an interactive environment to implement, distribute, simulate and document ecological and other dynamic models without the need to write long simulation programs. An object oriented framework using the S4 class system provides a consistent but still flexible approach to implement simulation models of different types:

- differential equation (ODE, PDE) models (class `odeModel`),
- grid-oriented individual-based models (class `gridModel`), and
- particle diffusion-type models (class `rwalkModel`),
- individual-based models (class `indbasedModel`),
- other model types by deriving a user specified subclass from `simObj`.

Each simulation model is implemented as S4 object (superclass `simObj`) with the following slots:

- `main = function(time,init,parms,...)`: a function holding the main equations of the model,
- `equations`: an optional non-nested list holding arbitrary sub-equations (sub-models) of the model. Sub-equations can be interdependent and can be called directly from within `main` or `initfunc`.
- `parms`: a list (or vector for some classes) with constant model parameters,
- `times`: vector of time steps or vector with three named values `from`, `to`, `by` specifying the simulation time steps. The from-to-by form can be edited with `editParms`.
- `init`: initial state (start values) of the simulation. This is typically a named vector (state variables in `odeModels`) or matrix (e.g. initial grid of `gridModels`).
- `inputs`: time dependend or spatially resolved external inputs can be specified as data frame or matrix (more efficient). Performance optimized versions of `approx` (see `approxTime`) are available.
- `solver`: a function or a character string specifying the numerical algorithm used, e.g. `"lsoda"`, `"rk4"` or `"euler"` from package `deSolve`). In contrast to `"euler"` that can be used for difference equations (i.e. `main` returns derivatives), `"iterator"` is intended for models where main returns the new state (i.e for individual-based models). It is also possible to reference own algorithms (solvers) that are defined in the user workspace or to assign solver functions directly.
- `observer`: optional slot which determines the data stored during the simulation. A user-provided `observer` function can also be used to write logging information to the screen or to the hard-disk, to perform run-time visualisation, or statistical analysis during the simulation.

  The `observer`-mechanism works only with `iteration` solvers. It is not available for `odeModels`.

- `out`: this slot holds the simulation results after a simulation run as data frame (if the return value of `main` is a vector) or as list (otherwise). The type of data stored in `out` can be manipulated by providing a user-definded `observer` function.
- `initfunc`: this slot can hold an optional function which is called automatically when a new object is created by `new` or when it is re-initialized by `initialize` or `sim`.

`simObj` model objects should be defined and created using the common S4 mechanisms (`new`).

Normally, a `simObj` object can contain all data needed to run simulations simply by entering the model object via `source()` or `data()` and then to run and plot the model with `plot(sim(obj))`.

Accessor functions (with names identical to the slot names) are provided to get or set model parameters, time steps, initial values, inputs, the solver, the main and sub-equations, an observer or an initfunc and to extract the model outputs. It is also possible to modify the components of the simecol objects directly, e.g. the model equations of a model `lv` with `lv@main`, but this is normally not recommended as there is no guarantee that this will work in a compatible way in future versions.

Models of different type are provided as data and some more in source code (see directory examples).

The examples can be used as a starting point to write own `simObj` objects and to distribute them to whomever you wish.

The package is supplemented with several utility functions (e.g. `seedfill` or `neighbours`), which can be used independently from `simObj` objects.

Author(s)

Thomas Petzoldt [aut, cre] (<https://orcid.org/0000-0002-4951-6468>)

References

Petzoldt, T. and K. Rinke (2007) **simecol**: An Object-Oriented Framework for Ecological Modeling in R. *Journal of Statistical Software*, **22**(9). doi: 10.18637/jss.v022.i09

See Also

`CA`, `chemostat`, `conway`, `diffusion`, `lv`, `lv3`, `upca`.

Examples

```
## (1) Quick Start Examples ===========================================================

data(lv)        # load basic Lotka-Volterra model

## Not run:
require("tcltk")
lv <- editParms(lv)

## End(Not run)
parms(lv)
main(lv)
lv <- sim(lv)
plot(lv)
```

```
results <- out(lv)

## Not run:
data(conway)    # Conway's game of life
init(conway) <- matrix(0, 10, 10)
times(conway) <-  1:100
conway <- editInit(conway) # enter some "1"
sim(conway, animate=TRUE, delay=100)

## End(Not run)

## (2) Define and run your own  simecol model ==========================

lv <- new("odeModel",
  main = function (time, init, parms) {
    with(as.list(c(init, parms)), {
      dn1 <-   k1 * N1 - k2 * N1 * N2
      dn2 <- - k3 * N2 + k2 * N1 * N2
      list(c(dn1, dn2))
    })
  },
  parms  = c(k1 = 0.2, k2 = 0.2, k3 = 0.2),
  times  = c(from = 0, to = 100, by = 0.5),
  init   = c(N1 = 0.5, N2 = 1),
  solver = "lsoda"
)

lv <- sim(lv)
plot(lv)

## (3) The same in matrix notation; this allows generalization      ====
##     to multi-species interaction models with > 2 species.        ====

LVPP <- new("odeModel",
  main = function(t, n, parms) {
    with(parms, {
      dn <- r * n  + n * (A %*% n)
      list(c(dn))
    })
  },
  parms = list(
    # growth/death rates
    r = c(k1 = 0.2, k3 = -0.2),
    # interaction matrix
    A = matrix(c(0.0, -0.2,
                 0.2,  0.0),
                 nrow = 2, ncol = 2, byrow=TRUE)
  ),
  times  = c(from = 0, to = 100, by = 0.5),
  init   = c(N1 = 0.5, N2 = 1),
  solver = "lsoda"
)
```

```
plot(sim(LVPP))
```

---

addtoenv *Add Functions from a Non-nested List of Named Functions to a Common Environment*

Description

Create and set an environment where elements (e.g. functions) within a non-nested named list of functions see each other. This function is normally used within other functions.

Usage

```
addtoenv(L, p = parent.frame())
```

Arguments

L	a non-nested list of named functions.
p	the environment where the functions are assigned to. Defaults to the parent frame.

Details

This function is used by the 'solver functions' of `simecol`.

Value

The list of functions within a common environment.

Note

This is a very special function that uses environment manipulations. Its main purpose is to 'open' the access to interdependend functions within a common list structure (function list).

See Also

attach, environment

Examples

```
eq <- list(f1 = function(x, y)    x + y,
           f2 = function(a, x, y) a * f1(x, y)
          )

fx <- function(eq) {
  eq <- addtoenv(eq)
  print(ls())
  print(environment(eq$f1))
```

```
  f1(3,4) + f2(1,2,3)
}

fx(eq)
## eq$f2(2,3,4)       # should give an error outside fx
environment(eq$f2)    # should return R_GlobalEnv again
```

---

approxTime *Linear Interpolation with Complete Matrices or Data Frames*

Description

Return a data frame, matrix or vector which linearly interpolates data from a given matrix or data frame.

Usage

```
approxTime(x, xout, ...)
approxTime1(x, xout, rule = 1)
```

Arguments

x	a matrix or data frame with numerical values giving coordinates of points to be interpolated. The first column needs to be in ascending order and is interpreted as independent variable (e.g. time), the remaining columns are used as dependent variables.
xout	a vector (or single value for approxTime1) of independend values specifying where interpolation has to be done.
rule	an integer describing how interpolation is to take place outside the interval [min(x), max(x)]. If rule is 1 then NAs are returned for such points and if it is 2, the value at the closest data extreme is used.
...	optional parameters passed to approx.

Details

The functions can be used for linear interpolation with a complete matrix or data frame. This can be used for example in the main function of an odeModel to get input values at a specified time xout. The version approxTime1 is less flexible (only one single value for xout and only linear interpolation) but has increased performance. Both functions are faster if x is a matrix instead of a data frame.

Value

approxTime returns a matrix resp. data frame of the same structure as x containing data which interpolate the given data with respect to xout. approxTime1 is a performance optimized special version with less options than the original approx function. It returns an interpolated vector.

See Also

approxfun

Examples

```
inputs <- data.frame(time = 1:10, y1 = rnorm(10), y2 = rnorm(10, mean = 50))
input  <- approxTime(inputs, c(2.5, 3), rule = 2)
```

---

as.simObj *Coerce simObj Objects to Lists and Vice-Versa*

Description

These functions can be used to coerce (i.e. convert) **simecol** model objects (simObj objects) to ordinary lists.

Usage

```
## S4 method for signature 'list'
as.simObj(x, ...)
## S4 method for signature 'simObj'
as.list(x, ...)
## alternative usage:
# as(x, "list")
# as(x, "simObj")
```

Arguments

x	object to be coerced
...	for compatibility

Details

Function as.list converts a simObj model to an ordinary list with an additional element 'class' storing the class name of the original object.

Function as.simObj converts in the opposite direction where the type of the object to be created is determined by a class name stored in the list element 'class'. If it is missing or contains a non-existing class name, an error message is printed. Additional list elements which are not slot names of the corresponding S4 object are omitted.

See Also

odeModel, new, as, as.list, simecol-package

Examples

```
data(lv3)
llv3 <- as(lv3, "list")
olv3 <- as(llv3, "simObj")

llv3 <- as.list(lv3)
olv3 <- as.simObj(llv3)

dput(as.list(lv3), control="useSource")
## Not run:
## save human readable object representation
dput(as.list(lv3), file="lv3.R", control=c("all"))
## read it back and test it
l_lv3 <- dget("lv3.R")
o_lv3 <- as.simObj(l_lv3)
plot(sim(o_lv3))

## End(Not run)
```

CA *Stochastic Cellular Automaton*

Description

simecol example: This model simulates a stochastic cellular automaton.

Usage

```
data(conway)
```

Format

An S4 object according to the `gridModel` specification. The object contains the following slots:

`main` functions with the state transition rules of Coway's Game of Life.

`parms` a list with two vector elements:

- `pbirth` probability of birth,
- `pdeath` death probability, dependend on neighbors.

`times` number of time steps to be simulated.

`init` a matrix, giving the initial state of the cellular grid (default: rectangle in the middle of the grid).

Details

To see all details, please have a look into the implementation below.

See Also

sim, parms, init, times.

Examples

```
##============================================
## Basic Usage:
##   work with the example
##============================================
data(CA)
times(CA)["to"] <- 10
plot(sim(CA))

set.seed(345)
times(CA)["to"] <- 50
CA <- sim(CA)

library(lattice)
tcol <- (terrain.colors(13))[-13]
x <- out(CA, last=TRUE)
x <- ifelse(x == 0, NA, x)
levelplot(x,
  cuts = 11,
  col.regions = tcol,
  colorkey = list(at = seq(0, 55, 5))
)

##============================================
## Implementation:
##   The code of the CA model
##============================================
CA <- new("gridModel",
  main = function(time, init, parms) {
    z     <- init
    nb    <- eightneighbors(z)
    pgen  <- 1 - (1 - parms$pbirth)^nb
    zgen  <- ifelse(z == 0 &
                runif(z) < pgen, 1, 0)
    zsurv <- ifelse(z >= 1 &
                runif(z) < (1 - parms$pdeath),
                z + 1, 0)
    zgen + zsurv
  },
  parms = list(pbirth = 0.02, pdeath = 0.01),
  times = c(from = 1, to = 50, by = 1),
  init = matrix(0, nrow = 40, ncol = 40),
  solver = "iteration"
)
init(CA)[18:22,18:22] <- 1
##============================================
```

chemostat *Chemostat Model*

Description

simecol example: Model of continuos culture of microorganisms (chemostat).

Usage

```
data(chemostat)
```

Format

An S4 object according to the odeModel specification. The object contains the following slots:

`main` the differential equations for substrate (`S`) and cells (`X`).

`parms` a vector with the named parameters of the model:

- `vm` maximum growth rate of the cells,
- `km` half saturation constant,
- `Y` yield coefficient (conversion factor of substrate into cells).
- `D` dilution rate,
- `S0` substrate concentration in the inflow.

`times` simulation time and integration interval.

`init` vector with start values for substrate (`S`) and cells (`X`).

To see all details, please have a look into the implementation below.

See Also

simecol-package, sim, parms, init, times.

Examples

```
##=============================================
## Basic Usage:
##   work with the example
##=============================================
data(chemostat)
plot(sim(chemostat))

parms(chemostat)["D"] <- 0.9
plot(sim(chemostat))



##=============================================
## Implementation:
```

```
##   The code of the chemostat model
##=========================================
chemostat <- new("odeModel",
  main = function(time, init, parms, inputs = NULL) {
    with(as.list(c(init, parms)), {
      mu  <- vm * S/(km + S)              # Monod equation
      dx1 <- mu * X - D * X               # cells, e.g. algae
      dx2 <-  D *(S0 - S) - 1/Y * mu * X  # substrate, e.g. phosphorus
      list(c(dx1, dx2))
    })
  },
  parms = c(
    vm = 1.0,           # max growth rate, 1/d
    km = 2.0,           # half saturation constant, mumol / L
    Y  = 100,           # cells /mumol Substrate
    D  = 0.5,           # dilution rate, 1/d
    S0 = 10             # substrate in inflow, mumol / L
  ),
  times = c(from=0, to=40, by=.5),
  init  = c(X=10, S=10), # cells / L; Substrate umol / L
  solver = "lsoda"
)
```

---

conway *The Classical Coway's Game of Life*

Description

simecol example: This model simulates a deterministic cellular automaton.

Usage

```
data(conway)
```

Format

An S4 object according to the `gridModel` specification. The object contains the following slots:

`main` functions with the state transition rules.

`parms` A list with two vector elements:

- `srv` number of neighbours, necessary to survive,
- `gen` number of neighbours, necessary to generate a new cell.

`times` number of time steps to be simulated,

`init` matrix with the initial state of the cellular grid (default: random).

Details

To see all details, please have a look into the implementation below.

References

Gardner, Martin (1970) The Fantastic Combinations of John Conway's New Solitaire Game 'Life.' *Scientific American*, October 1970.

See Also

sim, parms, init, times.

Examples

```
##============================================
## Basic Usage:
##   explore the example
##============================================
data(conway)
plot(sim(conway))

## more interesting start conditions
m <- matrix(0, 40, 40)
m[5:35, 19:21] <- 1
init(conway) <- m
plot(sim(conway), col=c("white", "green"), axes = FALSE)

## change survival rules
parms(conway) <- list(srv = c(3,4), gen = c(3, 4))
plot(sim(conway), col = c("white", "green"), axes = FALSE)
## Not run:
require("tcltk")
init(conway) <- matrix(0, 10, 10)
conway <- editInit(conway) # enter some "1"
sim(conway, animate = TRUE, delay = 100)

##============================================
## Implementation:
##   The code of Conways Game of Life
##============================================
conway <- new("gridModel",
  main = function(time, init, parms) {
    x      <- init
    nb     <- eightneighbours(x)
    surviv <- (x >  0 & (nb %in% parms$srv))
    gener  <- (x == 0 & (nb %in% parms$gen))
    x      <- (surviv + gener) > 0
    return(x)
  },
  parms  = list(srv = c(2, 3), gen = 3),
  times  = 1:17,
  init   = matrix(round(runif(1000)), ncol = 40),
  solver = "iteration"
)

## End(Not run)
```

diffusion *A Random Walk Particle Diffusion Model*

Description

simecol example: This is a random walk (basic particle diffusion) model.

Usage

```
data(diffusion)
```

Format

An S4 object according to the `rwalkModel` specification. The object contains the following slots:

`main` A function with the movement rules for the particles.

`parms` A list with the following components:

- `ninds` number of simulated particles,
- `speed` speed of the particles,
- `area` vector with 4 elements giving the coordinates (left, bottom, right, top) of the coordinate system.

`times` Simulation time (discrete time steps, `by`-argument ignored).

`init` Data frame holding the start properties (Cartesian coordinates `x` and `y` and movement angle `a`) of the particles.

Details

To see all details, please have a look into the implementation.

See Also

`sim`, `parms`, `init`, `times`.

Examples

```
##=============================================
## Basic Usage:
##   explore the example
##=============================================
## Not run:
data(diffusion)
## (1) minimal example
plot(sim(diffusion))
## show "grid of environmental conditions"
image(inputs(diffusion))

## (2) scenario
```

```
##     with homogeneous environment (no "refuge" in the middle)
no_refuge <- diffusion # Cloning of the whole model object
inputs(no_refuge) <- matrix(1, 100, 100)
plot(sim(no_refuge))

##============================================
## Advanced Usage:
##   Assign a function to the observer-slot.
##============================================
observer(diffusion) <- function(state, ...) {
  ## numerical output to the screen
  cat("mean x=", mean(state$x),
      ", mean y=", mean(state$y),
      ", sd   x=", sd(state$x),
      ", sd   y=", sd(state$y), "\n")
  ## animation
  par(mfrow=c(2,2))
  plot(state$x, state$y, xlab="x", ylab="y", pch=16, col="red", xlim=c(0, 100))
  hist(state$y)
  hist(state$x)
  ## default case: return the state --> iteration stores it in "out"
  state
}

sim(diffusion)

## remove the observer and restore original behavior
observer(diffusion) <- NULL
diffusion <- sim(diffusion)

## End(Not run)

##============================================
## Implementation:
##   The code of the diffusion model.
##   Note the use of the "initfunc"-slot.
##============================================
diffusion <- rwalkModel(
  main = function(time, init, parms, inputs = NULL) {
    speed   <- parms$speed
    xleft   <- parms$area[1]
    xright  <- parms$area[2]
    ybottom <- parms$area[3]
    ytop    <- parms$area[4]

    x <- init$x  # x coordinate
    y <- init$y  # y coordinate
    a <- init$a  # angle (in radians)
    n <- length(a)

    ## Rule 1: respect environment (grid as given in "inputs")
    ## 1a) identify location on "environmental 2D grid" for each individual
    i.j <- array(c(pmax(1, ceiling(x)), pmax(1, ceiling(y))), dim=c(n, 2))
```

```
    ## 1b) speed dependend on "environmental conditions"
    speed <- speed * inputs[i.j]

    ## Rule 2: Random Walk
    a  <- (a + 2 * pi / runif(a))
    dx <- speed * cos(a)
    dy <- speed * sin(a)
    x  <- x + dx
    y  <- y + dy

    ## Rule 3: Wrap Around
    x <- ifelse(x > xright, xleft, x)
    y <- ifelse(y > ytop, ybottom, y)
    x <- ifelse(x < xleft, xright, x)
    y <- ifelse(y < ybottom, ytop, y)
    data.frame(x=x, y=y, a=a)
  },
  times  = c(from=0, to=100, by=1),
  parms  = list(ninds=50, speed = 1, area = c(0, 100, 0, 100)),
  solver = "iteration",
  initfunc = function(obj) {
    ninds   <- obj@parms$ninds
    xleft   <- obj@parms$area[1]
    xright  <- obj@parms$area[2]
    ybottom <- obj@parms$area[3]
    ytop    <- obj@parms$area[4]
    obj@init <- data.frame(x = runif(ninds) * (xright - xleft) + xleft,
                           y = runif(ninds) * (ytop - ybottom) + ybottom,
                           a = runif(ninds) * 2 * pi)
    inp <- matrix(1, nrow=100, ncol=100)
    inp[, 45:55] <- 0.2
    inputs(obj) <- inp
    obj
  }
)
```

`editParms` *Edit 'parms', 'init' or 'times' Slot of 'simecol' Objects*

Description

The functions invoke an editor dialog for parameters, initial values or time steps of `simObj` objects and then assign the new (edited) version of `x` in the user's workspace. A **Tcl/Tk** version or spreadsheet editor is displayed if possible, depending on the structure of the respective slot.

Usage

```
editParms(x)
editTimes(x)
editInit(x)
```

Arguments

x	A valid instance of the simObj class.

See Also

sEdit, simObj, parms, times, init, edit

Examples

```
## Not run:
require("tcltk")
data(lv)        # load basic Lotka-Volterra model
lv <- editParms(lv)
plot(sim(lv))

data(conway)    # Conway's game of life
init(conway) <- matrix(0, 10, 10)
conway <- editInit(conway) # enter some "1"
sim(conway, animate = TRUE, delay = 100)

## End(Not run)
```

---

eightneighbours *Count Number of Neighbours in a Rectangular Cellular Grid.*

Description

This function returns the sum of the eight neibours of a cell within a matrix. It can be used to simulate simple cellular automata, e.g. Conway's Game of Life.

Usage

```
eightneighbours(x)
eightneighbors(x)
```

Arguments

x	The cellular grid, which typically contains integer values of zero (dead cell) or one (living cell).

Value

A matrix with the same structure as x, but with the sum of the neighbouring cells of each cell.

See Also

seedfill, neighbours, conway

Examples

```
n <- 80; m <- 80
x <- matrix(rep(0, m*n), nrow = n)
x[round(runif(1500, 1, m*n))] <- 1
## uncomment this for another figure
#x[40, 20:60] <- 1

image(x, col=c("wheat", "grey", "red"))
x2 <- x
for (i in 2:10){
  nb <- eightneighbours(x)

  ## survive with 2 or 3 neighbours
  xsurv <- ifelse(x > 0 & (nb == 2 | nb ==3), 1, 0)

  ## generate for empty cells with 3 neigbours
  xgen <- ifelse(x == 0 & nb == 3, 1, 0)

  x  <- ((xgen + xsurv)>0)
  x2 <- ifelse(x2>1, 1, x2)
  x2 <- ifelse(x>0, 2, x2)

  image(x2, col=c("wheat", "grey", "red"), add=TRUE)
}
```

`fitOdeModel` *Parameter Fitting for odeModel Objects*

Description

Fit parameters of `odeModel` objects to measured data.

Usage

```
fitOdeModel(simObj, whichpar = names(parms(simObj)), obstime, yobs,
  sd.yobs = as.numeric(lapply(yobs, sd)), initialize = TRUE,
  weights = NULL, debuglevel = 0, fn = ssqOdeModel,
  method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "PORT",
   "newuoa", "bobyqa"),
  lower = -Inf, upper = Inf, scale.par = 1,
  control = list(), ...)
```

Arguments

`simObj`	a valid object of class `odeModel`,
`whichpar`	character vector with names of parameters which are to be optimized (subset of parameter names of the `simObj`),
`obstime`	vector with time steps for which observational data are available,

yobs	data frame with observational data for all or a subset of state variables. Their names must correspond exacly with existing names of state variables in the odeModel,
sd.yobs	vector of given standard deviations (or scale) for all observational variables given in yobs. If no standard deviations (resp. scales) are given, these are estimated from yobs,
initialize	optional boolean value whether the simObj should be re-initialized after the assignment of new parameter values. This can be necessary in certain models to assign consistent values to initial state variables if they depend on parameters.
weights	optional weights to be used in the fitting process. See cost function (currently only ssqOdeModel) for details.
debuglevel	a positive number that specifies the amount of debugging information printed,
fn	objective function, i.e. function that returns the quality criterium that is minimized, defaults to ssqOdeModel,
method	optimization method, see nlminb for the PORT algorithm, newuoa resp. bobyqa for the newuoa and bobyqa algorithms, and optim for all other methods,
lower, upper	bounds of the parameters for method L-BFGS-B, see optim, PORT see nlminb and bobyqa bobyqa. The bounds are also respected by other optimizers by means of an internal transformation of the parameter space (see p.constrain). In this case, *named vectors* are required.
scale.par	scaling of parameters for method PORT see nlminb. In many cases, automatic scaling (scale.par = 1) does well, but sometimes (e.g. if parameter ranges differ several orders of magnitude) manual adjustment is required. Often you get a reasonable choice if you set scale.par = 1/upper. The parameter is ignored by all other methods. For "Nelder-Mead", "BFGS", "CG" and "SANN" parameter scaling occurs as a side effect of parameter transformation with p.constrain.
control	a list of control parameters for optim resp. nlminb,
...	additional parameters passed to the solver method (e.g. to lsoda).

Details

This function works currently only with odeModel objects where parms is a vector, not a list.

Note also that the control parameters of the PORT algorithm are different from the control parameters of the other optimizers.

Value

A list with the optimized parameters and other information, see optim resp. nlminb for details.

References

Gay, D. M. (1990) Usage Summary for Selected Optimization Routines. Computing Science Technical Report No. 153. AT&T Bell Laboratories, Murray Hill, NJ.

Powell, M. J. D. (2009). The BOBYQA algorithm for bound constrained optimization without derivatives. Report No. DAMTP 2009/NA06, Centre for Mathematical Sciences, University of Cambridge, UK. https://www.damtp.cam.ac.uk/user/na/NA_papers/NA2009_06.pdf

See Also

ssqOdeModel, optim, nlminb, bobyqa

Note also that package **FME** function modFit has even more flexible means to fit model parameters.

Examples are given in the package vignettes.

Examples

```
## ======== load example model =========
data(chemostat)

#source("chemostat.R")

## derive scenarios
cs1 <- cs2 <- chemostat

## generate some noisy data
parms(cs1)[c("vm", "km")] <- c(2, 10)
times(cs1) <- c(from=0, to=20, by=2)
yobs <- out(sim(cs1))
obstime <- yobs$time
yobs$time <- NULL
yobs$S <- yobs$S + rnorm(yobs$S, sd= 0.1 * sd(yobs$S))*2
yobs$X <- yobs$X + rnorm(yobs$X, sd= 0.1 * sd(yobs$X))

## ======== optimize it! =========

## time steps for simulation, either small for rk4 fixed step
# times(cs2)["by"] <- 0.1
# solver(cs2) <- "rk4"

## or, faster: use lsoda and and return only required steps that are in the data
times(cs2) <- obstime
solver(cs2) <- "lsoda"

## Nelder-Mead (default)
whichpar  <- c("vm", "km")

res <- fitOdeModel(cs2, whichpar=whichpar, obstime, yobs,
  debuglevel=0,
  control=list(trace=TRUE))

coef(res)

## assign fitted parameters to the model, i.e. as start values for next step
parms(cs2)[whichpar] <- coef(res)

## alternatively, L-BFGS-B (allows lower and upper bounds for parameters)
res <- fitOdeModel(cs2, whichpar=c("vm", "km"), obstime, yobs,
  debuglevel=0, fn = ssqOdeModel,
  method = "L-BFGS-B", lower = 0,
```

```
  control=list(trace=TRUE),
  atol=1e-4, rtol=1e-4)

coef(res)

## alternative 2, transform parameters to constrain unconstrained method
## Note: lower and upper are *named* vectors
res <- fitOdeModel(cs2, whichpar=c("vm", "km"), obstime, yobs,
  debuglevel=0, fn = ssqOdeModel,
  method = "BFGS", lower = c(vm=0, km=0), upper=c(vm=4, km=20),
  control=list(trace=TRUE),
  atol=1e-4, rtol=1e-4)

coef(res)


## alternative 3a, use PORT algorithm
parms(cs2)[whichpar] <- c(vm=1, km=2)

lower <- c(vm=0, km=0)
upper <- c(vm=4, km=20)

res <- fitOdeModel(cs2, whichpar=c("vm", "km"), obstime, yobs,
  debuglevel=0, fn = ssqOdeModel,
  method = "PORT", lower = lower, upper = upper,
  control=list(trace=TRUE),
  atol=1e-4, rtol=1e-4)

coef(res)

## alternative 3b, PORT algorithm with manual parameter scaling
res <- fitOdeModel(cs2, whichpar=c("vm", "km"), obstime, yobs,
  debuglevel=0, fn = ssqOdeModel,
  method = "PORT", lower = lower, upper = upper, scale.par = 1/upper,
  control=list(trace=TRUE),
  atol=1e-4, rtol=1e-4)

coef(res)

## set model parameters to  fitted values and simulate again
parms(cs2)[whichpar] <- coef(res)
times(cs2) <- c(from=0, to=20, by=1)
ysim <- out(sim(cs2))

## plot results
par(mfrow=c(2,1))
plot(obstime, yobs$X, ylim = range(yobs$X, ysim$X))
lines(ysim$time, ysim$X, col="red")
plot(obstime, yobs$S, ylim= range(yobs$S, ysim$S))
lines(ysim$time, ysim$S, col="red")
```

fromtoby *Create Regular Sequence from 'from-to-by' Vector*

Description

This function creates a sequence from named vectors with the names `from`, `to` and `by`.

Usage

```
fromtoby(times)
```

Arguments

`times`	A named vector with the names `from`, `to` and `by`.

Details

Named vectors with `from`, `to` and `by` can be used in **simecol** to specify time steps.

Value

The appropriate vector with a sequence, generated by `seq`.

See Also

`seq`

Examples

```
times <- c(from=1, to=5, by=0.1)
fromtoby(times)
```

initialize-methods *Methods for Function 'initialize' in Package 'simecol'*

Description

This function is used to initialize objects derived from the `simObj` superclass, it is by default automatically called during object creation and by `sim`.

Usage

```
## S4 method for signature 'simObj'
initialize(.Object, ...)
```

Arguments

.Object	simObj instance which is to be re-initialized.
...	provided for compatibility with the default method of initialize, or slots of the object which is to be created (in case of new).

Methods

.Object = "ANY" Generic function: see new.

.Object = "simObj" The initialize function is normally called implicitly by new to create new objects. It may also be called explicitly to return a cloned and re-initialized object.

The **simecol** version of initialize provides an additonal mechanism to call a user specified function provided in the initfun slot of a simObj instance that can perform computations during the object creation process. The initfunc must have obj as its only argument and must return the modified version of this obj, see examples below. As a side effect end to ensure consistency, initialize clears outputs stored in slot out from former simulations.

See Also

simObj, new

Examples

```
## Note: new calls initialize and initialize calls initfunc(obj)
lv_efr <- new("odeModel",
  main = function (time, init, parms, ...) {
    x <- init
    p <- parms
    S <- approxTime1(inputs, time, rule=2)["s.in"]
    dx1 <-   S * p["k1"] * x[1] - p["k2"] * x[1] * x[2]
    dx2 <-     - p["k3"] * x[2] + p["k2"] * x[1] * x[2]
    list(c(dx1, dx2))
  },
  parms  = c(k1=0.2, k2=0.2, k3=0.2),
  times  = c(from=0, to=100, by=0.5),
  init   = c(prey=0.5, predator=1),
  solver = "lsoda",
  initfunc = function(obj) {
    tt <- fromtoby(times(obj))
    inputs(obj) <- as.matrix(data.frame(
            time = tt,
            s.in = pmax(rnorm(tt, mean=1, sd=0.5), 0)
          ))
    obj
  }
)
plot(sim(lv_efr))                    # initialize called automatically
plot(sim(lv_efr))                    # automatic initialization, different figure

lv_efr<- initialize(lv_efr)          # re-initialize manually
plot(sim(lv_efr, initialize = FALSE)) # simulation with that configuration
```

`iteration` *Discrete Simulation*

Description

Solver function to simulate discrete ecological (or other) dynamic models. It is normally called indirectly from `sim`.

Usage

```
iteration(y, times=FALSE, func=FALSE, parms=FALSE, animate = FALSE, ...)
```

Arguments

`y`	the initial values for the 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 *next time step* (not the derivatives !!!) in the system (the *model defininition*) at time t. The user-supplied function `func` must be called as: `yprime = func(t,y,parms)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ode system, 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 `names` 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.
`animate`	Animation during the simulation (if available for the specified class.
`...`	optional arguments passed to the `plot` function if `animate=TRUE`.

Details

The solver method `iteration` is used to simulate discrete event models. Normally, this function is run indirectly from `sim`.

In contrast to differential equation solvers, the `main` function of the model must not return the first derivative but instead and explicitly the new state at the specified times.

The actual value of time is available in the `main` function as `time` and the current increment as `parms["DELTAT"]` or `parms$DELTAT`. It is element of a vector if `parms` is a vector and it is a list if `parms` is a list.

If iteration is used for difference equations (see example `dlogist` below), it is mandatory to multiply the incremental part with `DELTAT` to ensure that variable time steps are correctly respected and that the first row of the simulation outputs contains the states at t_0.

The default `iteration` method of class `simObj` supports the `observer` mechanism. This means that a function stored in slot `observer` is called during each iteration step with the return value of `main` as its first argument. You can use this to control the amount of data stored during each iteration step (e.g. whole population or only mean values for individual based models), to do run-time animation or to write log files.

As an alternative for models of class `odeModel`, the `iteration` method of package **deSolve** may be used as a user-defined solver function. This is slightly faster and the output supports the extended plotting functions, but then no observers are possible and no implicit `DELTAT` variable.

Value

A list of the model outputs (states ...) for each timestep.

See Also

`sim`, `observer`, `parms`, `lsoda`, `rk4`, `euler`, `iteration`.

Examples

```
data(conway)


## plot after simulation:
plot(sim(conway), delay=100)

## plot during simulation
sim(conway, animate=TRUE, delay=100)


## discrete version of logistic growth equation
## Note: function main returns the *new value*, not the derivative

dlogist <- new("odeModel",
  main = function (time, init, parms, ...) {
    x <- init
    with(as.list(parms), {
      x <- x + r * x * (1 - x / K) * DELTAT
      #   ^^^ add to old value       ^^^^^^ special parameter with time step
      list(c(x))
    })
  },
  parms  = c(r=0.1, K=10),
  times  = seq(0, 100, 1),
  init   = c(population=0.1),
  solver = "iteration" #!!!
)

plot(sim(dlogist))
```

```
## alternative with function that returns the derivative
## discrete steps are realized with the euler method

dlogist <- new("odeModel",
  main = function (time, init, parms, ...) {
    x <- init
    with(as.list(parms), {
      x <- r * x * (1 - x / K)
      list(c(x))
    })
  },
  parms  = c(r=0.1, K=10),
  times  = seq(0, 100, 1),
  init   = c(population=0.1),
  solver = "euler"
)

plot(sim(dlogist))

## second alternative: use of the "iteration" solver from
## package deSolve, that supports extended plotting functions

dlogist <- new("odeModel",
  main = function (time, init, parms, ...) {
    x <- init[1]
    with(as.list(parms), {
      x <- x + r * x * (1 - x / K)
      #   ^^^ add to old value
      list(c(x))
    })
  },
  parms  = c(r=0.1, K=10),
  times  = seq(0, 100, 1),
  init   = c(population=0.1),
  solver =  function(y, times, func, parms, ...)
              ode(y, times, func, parms, ..., method="iteration")
)

plot(sim(dlogist))
```

`listOrNULL-class` *Helpful Union Classes*

Description

Classes representing either list or NULL (i.e. empty), function or NULL, function or character vector, numeric vector or list, or list or data.frame.

Objects from the Class

These classes are virtual: No objects may be created from it.

Methods

No methods exist for these classes.

See Also

simObj

---

lv *Lotka-Volterra Predator-Prey Model*

Description

simecol example: basic Lotka-Volterra predator prey-model.

Usage

```
data(lv)
```

Format

An S4 object according to the odeModel specification. The object contains the following slots:

`main` Lotka-Volterra equations for predator and prey.

`parms` Vector with the named parameters of the model:

- `k1` growth rate of the prey population,
- `k2` encounter rate of predator and prey,
- `k3` death rate of the predator population.

`times` Simulation time and integration interval.

`init` Vector with start values for predator and prey.

Details

To see all details, please have a look into the implementation.

References

Lotka, A. J. 1925. *Elements of physical biology.* Williams and Wilkins, Baltimore.

Volterra, V. (1926). Variazionie fluttuazioni del numero d'individui in specie animali conviventi. *Mem. Acad.Lincei*, **2**, 31-113.

See Also

simecol-package, sim, parms, init, times.

Examples

```
##=============================================
## Basic Usage:
##   explore the example
##=============================================
data(lv)
print(lv)
plot(sim(lv))

parms(lv) <- c(k1=0.5, k2=0.5, k3=0.5)
plot(sim(lv))

##=============================================
## Implementation:
##   The code of the Lotka-Volterra-model
##=============================================
lv <- new("odeModel",
  main = function (time, init, parms) {
    x <- init
    p <- parms
    dx1 <-   p["k1"] * x[1] - p["k2"] * x[1] * x[2]
    dx2 <- - p["k3"] * x[2] + p["k2"] * x[1] * x[2]
    list(c(dx1, dx2))
  },
  parms  = c(k1=0.2, k2=0.2, k3=0.2),
  times  = c(from=0, to=100, by=0.5),
  init   = c(prey=0.5, predator=1),
  solver = "rk4"
)
```

`lv3` *Lotka-Volterra-Type Model with Resource, Prey and Predator*

Description

simecol example: predator prey-model with three equations: predator, prey and resource (e.g. nutriens, grassland).

Usage

```
data(lv3)
```

Format

A valid S4 object according to the `odeModel` specification. The object contains the following slots:

`main` Lotka-Volterra equations for predator prey and resource.

`parms` Vector with named parameters of the model:

c growth rate of the prey population,

d encounter rate of predator and prey,

e yield factor (allows conversion with respect to d),

f death rate of the predator population,

g recycling parameter.

`inputs` Time series specifying external delivery of resource.

`times` Simulation time and integration interval.

`init` Vector with start values for s, p and k.

s Resource (e.g. grassland or phosphorus).

p Producer (prey).

k Consumer (predator).

`solver` Character string specifying the integration method.

See Also

simecol-package, sim, parms, init, times.

Examples

```
##============================================
## Basic Usage:
##   explore the example
##============================================
data(lv3)
plot(sim(lv3))
times(lv3)["by"] <- 5    # set maximum external time step to a large value
plot(sim(lv3))           # wrong! automatic time step overlooks internal inputs
plot(sim(lv3, hmax = 1)) # integration with correct maximum internal time step

##============================================
## Implementation:
##   The code of the model
##============================================
lv3 <- new("odeModel",
  main = function(time, init, parms, inputs) {
    s.in <- approxTime1(inputs, time, rule = 2)["s.in"]
    with(as.list(c(init, parms)),{
      ds <- s.in  - b*s*p + g*k
      dp <- c*s*p - d*k*p
      dk <- e*p*k - f*k
      list(c(ds, dp, dk), s.in = s.in)
    })
  },
  parms = c(b = 0.1, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0),
  times  = c(from = 0, to = 200, by = 1),
  inputs = as.matrix(
    data.frame(
      time = c(0,   99, 100,  101, 200),
      s.in = c(0.1, 0.1, 0.5, 0.1, 0.1)
```

```
    )
  ),
  init = c(s = 1, p = 1, k = 1), # substrate, producer, consumer
  solver = "lsoda"
)
```

`mixNamedVec` *Mix Two Named Vectors, Resolving Name Conflicts*

Description

The function mixes two named vectors. The resulting vectors contains all elements with unique name and only one of the two versions of the elements which have the same name in both vectors.

Usage

```
mixNamedVec(x, y, resolve.conflicts = c("x", "y"), warn = TRUE)
```

Arguments

`x`	first named vector,
`y`	second named vector,
`resolve.conflicts`	name of the vector from which all elements are taken,
`warn`	an indicator if a warning should be given if elements are not unique. This argument should usually set to FALSE, but the default is TRUE to be on the safe side.

Value

a vector with all elements from one vector and only these elements of the second, that have a unique name not contained in the other vector.

Author(s)

Thomas Petzoldt

See Also

`which`

Examples

```
x <- c(a=1, b=2, c=3)
y <- c(a=1, b=3, d=3)

mixNamedVec(x, y)
mixNamedVec(x, y, resolve.conflicts="x")

mixNamedVec(x, y, resolve.conflicts="x", warn=FALSE)
mixNamedVec(x, y, resolve.conflicts="y", warn=FALSE)

## without names, returns vector named in "resolve conflicts"
x <- as.vector(x)
y <- as.vector(y)
mixNamedVec(x, y)
mixNamedVec(x, y, resolve.conflicts="y")

## names partly
x <- c(4, a=1, b=2, c=3, 4, 9)
y <- c(a=1, 6, b=3, d=3, 8)

mixNamedVec(x, y)
mixNamedVec(x, y, resolve.conflicts="y")
```

---

modelFit-class *Class of Fitted Model Parameters*

Description

Class that contains parameters and other information returned by `fitOdeModel`.

Methods

`coef`, `deviance`, `print`

See Also

`fitOdeModel`

---

modelFit-method *Show Results of Model Fits*

Description

Functions to access the results of parameter fits.

Usage

```
## S4 method for signature 'modelFit'
coef(object, ...)

## S4 method for signature 'modelFit'
deviance(object, ...)

## S4 method for signature 'modelFit'
summary(object, ...)

## S4 method for signature 'modelFit'
x$name

## S4 method for signature 'modelFit'
x[i, j, ..., drop=TRUE]

## S4 method for signature 'modelFit'
x[[i, j, ...]]
```

Arguments

`object, x`	'modelFit' object from which to extract element(s).
`i, j`	indices specifying elements to extract. Indices are numeric or character vectors or empty (missing) or NULL.
`name`	a literal character string or a name (possibly backtick quoted). For extraction, this is normally partially matched to the names of the object.
`drop`	For matrices and arrays. If TRUE the result is coerced to the lowest possible dimension.
`...`	other arguments pased to the methods

See Also

`fitOdeModel`, `Extract`

`neighbours` *Count Number of Neighbours on a Rectangular Grid.*

Description

This is the base function for the simulation of deterministic and stochastic cellular automata on rectangular grids.

Usage

```
neighbours(x, state = NULL, wdist = NULL, tol = 1e-4, bounds = 0)
neighbors(x, state = NULL, wdist = NULL, tol = 1e-4, bounds = 0)
```

Arguments

x	Matrix. The cellular grid, in which each cell can have a specific state value, e.g. zero (dead cell) or one (living cell) or the age of an individual.
state	A value, whose existence is checked within the neighbourhood of each cell.
wdist	The neighbourhood weight matrix. It has to be a square matrix with an odd number of rows and columns).
tol	Tolerance value for the comparision of state with the state of each cell. If tol is a large value, then more than one state can be checked simultaneously.
bounds	A vector with either one or four values specifying the type of boundaries, where 0 means open boundaries and 1 torus-like boundaries. The values are specified in the order bottom, left, top, right.

Details

The performance of the function depends on the size of the matrices and the type of the boundaries, where open boundaries are faster than torus like boundaries. Function eightneighbours is even faster.

Value

A matrix with the same structure as x with the weighted sum of the neigbours with values between state -tol and state + tol.

See Also

seedfill, eightneighbours, conway

Examples

```
## ====================================================================
## Demonstration of the neighborhood function alone
## ====================================================================

## weight matrix for neighbourhood determination
wdist <- matrix(c(0.5,0.5,0.5,0.5,0.5,
                  0.5,1.0,1.0,1.0,0.5,
                  0.5,1.0,1.0,1.0,0.5,
                  0.5,1.0,1.0,1.0,0.5,
                  0.5,0.5,0.5,0.5,0.5), nrow=5)

## state matrix
n <- 20; m <- 20
x <- matrix(rep(0, m * n), nrow = n)

## set state of some cells to 1
x[10, 10] <- 1
x[1, 5]   <- 1
x[n, 15]  <- 1
x[5, 2]   <- 1
```

```
x[15, m]  <- 1
#x[n, 1]   <- 1 # corner

opar <- par(mfrow = c(2, 2))
## start population
image(x)
## open boundaries
image(matrix(neighbours(x, wdist = wdist, bounds = 0), nrow = n))
## torus (donut like)
image(matrix(neighbours(x, wdist = wdist, bounds = 1), nrow = n))
## cylinder (left and right boundaries connected)
image(matrix(neighbours(x, wdist = wdist, bounds = c(0, 1, 0, 1)), nrow = n))
par(opar) # reset graphics area

## ====================================================================
## The following example demonstrates a "plain implementation" of a
## stochastic cellular automaton i.e. without the simecol structure.
##
## A simecol implementation of this can be found in
## the example directory of this package (file: stoch_ca.R).
## ====================================================================
mycolors <- function(n) {
  col <- c("wheat", "darkgreen")
  if (n>2) col <- c(col, heat.colors(n - 2))
  col
}

pj <- 0.99  # survival probability of juveniles
pa <- 0.99  # survival probability of adults
ps <- 0.1   # survival probability of senescent
ci <- 1.0   # "seeding constant"
adult <- 5  # age of adolescence
old   <- 10 # age of senescence

## Define a start population
n <- 80
m <- 80
x <- rep(0, m*n)

## stochastic seed
## x[round(runif(20,1,m*n))] <- adult
dim(x)<- c(n, m)

## rectangangular seed in the middle
x[38:42, 38:42] <- 5

## plot the start population
image(x, col = mycolors(2))

## ------------------------------------------------------------------------
## Simulation loop (hint: increase loop count)
## ------------------------------------------------------------------------
for (i in 1:10){
```

```
  ## rule 1: reproduction
  ## 1.1 which cells are adult? (only adults can generate)
  ad <- ifelse(x >= adult & x < old, x, 0)

  ## 1.2 how much (weighted) adult neighbours has each cell?
  nb <- neighbours(ad, wdist = wdist)

  ## 1.3 a proportion of the seeds develops juveniles
  ## simplified version, you can also use probabilities
  genprob <- nb * runif(nb) * ci
  xgen  <- ifelse(x == 0 & genprob >= 1, 1, 0)

  ## rule 2: growth and survival of juveniles
  xsurvj <- ifelse(x >= 1 & x < adult & runif(x) <= pj, x+1, 0)
  ## rule 2: growth and survival of adults
  xsurva <- ifelse(x >= adult & x < old & runif(x) <= pa, x+1, 0)
  ## rule 2: growth and survival of senescent
  xsurvs <- ifelse(x >= old & runif(x) <= ps, x+1, 0)

  ## make resulting grid of complete population
  x     <- xgen + xsurvj + xsurva + xsurvs

  ## plot resulting grid
  image(x, col = mycolors(max(x) + 1), add = TRUE)
  if (max(x) == 0) stop("extinction", call. = FALSE)
}

## modifications:  pa<-pj<-0.9

## additional statistics of population structure
## with table, hist, mean, sd, ...
```

`observer` *Get or Set an Observer Functions to an 'simObj' Object*

Description

Get or set a user-defined observer to enable user-specified storage of simulation results, visualisation or logging.

Usage

```
observer(obj, ...)
observer(obj) <- value
```

Arguments

obj	A valid simObj instance.
value	A function specifying an observer, see Details.
...	Reserved for method consistency.

Details

The observer can be used with solver iteration or a user-defined solver function. It does not work with differential equations solvers.

The observer is a function with the following arguments:

```
function(state)
```

or:

```
function(state,time,i,out,y)
```

Where state is the actual state of the system, time and i are the simulation time and the indexof the time step respectively, out is the output of the actual simulation collected so far. The original object used in the simulation is passed via y and can be used to get access on parameter values or model equations.

If available, the observer function is called for every time step in the iteration. It can be used for calculations "on the fly" to reduce memory of saved data, for user-specified animation or for logging purposes.

If the value returned by observer is a vector, than resulting out will be a data.frame, otherwise it will be a list of all states.

Value

The observer function either modifies obj or it returns the assigned observer function or NULL (the default).

See Also

iteration for the iteration solver,

parms for accessor and replacement functions of other slots,

simecol-package for an overview of the package.

Examples

```
## load model "diffusion"
data(diffusion)

solver(diffusion) # solver is iteration, supports observer
times(diffusion) <- c(from=0, to=20, by=1) # to can be increased, to e.g. 100

### == Example 1 ================================================================

## assign an observer for visualisation
```

```
observer(diffusion) <- function(state) {
  ## numerical output to the screen
  cat("mean x=", mean(state$x),
      ", mean y=", mean(state$y),
      ", sd   x=", sd(state$x),
      ", sd   y=", sd(state$y), "\n")
  ## animation
  par(mfrow = c(2, 2))
  plot(state$x, state$y, xlab = "x", ylab = "y", pch = 16,
    col = "red", xlim = c(0, 100))
  hist(state$y)
  hist(state$x)

  ## default case:
  ## return the state --> iteration stores full state in "out"
  state
}

sim(diffusion)

### == Example 2 ================================================================

## an extended observer with full argument list
observer(diffusion) <- function(state, time, i, out, y) {
  ## numerical output to the screen
  cat("index =", i,
      ", time =", time,
      ", sd   x=", sd(state$x),
      ", sd   y=", sd(state$y), "\n")
  ## animation
  par(mfrow = c(2, 2))
  plot(state$x, state$y, xlab = "x", ylab = "y", pch = 16,
    col = "red", xlim = c(0, 100))
  hist(state$y)
  hist(state$x)
  if (is.matrix(out)) # important because out may be NULL for the first call
    matplot(out[,1], out[,-1]) # dynamic graph of sd in both directions

  ## return a vector with summary information
  c(times = time, sdx=sd(state$x), sdy=sd(state$y))
}

diffusion <- sim(diffusion)

### == Restore default ==========================================================

observer(diffusion) <- NULL # delete observer
diffusion <- sim(diffusion)
```

odeModel *Generating-functions (Constructors) to Create Objects of Classes 'odeModel', 'rwalkModel' and 'gridModel'.*

Description

These functions can be used to create simObj instances without using new explicitly.

Usage

```
odeModel(obj = NULL, main = NULL,
        equations = NULL, times = c(from = 0, to = 10, by = 1),
        init = numeric(0), parms = numeric(0),
        inputs = NULL, solver = "rk4", initfunc = NULL)

gridModel(obj = NULL, main = NULL,
        equations = NULL, times = c(from=0, to=10, by=1),
        init = matrix(0), parms = list(),
        inputs = NULL, solver = "iteration", initfunc = NULL)

rwalkModel(obj = NULL, main = NULL,
        equations = NULL, times = c(from = 0, to = 10, by = 1),
        init = NULL, parms = list(),
        inputs = NULL, solver = "iteration", initfunc = NULL)

indbasedModel(obj = NULL, main = NULL,
        equations = NULL, times = c(from = 0, to = 10, by = 1),
        init = NULL, parms = list(),
        inputs = NULL, solver = "iteration", initfunc = NULL)
```

Arguments

obj	Unnamed arguments are regarded as objects of the corresponding class. If obj is omitted, the new object is created from scratch.
main	The main equations of the model.
equations	The sub-models (sub-equations and of the model).
times	A vector of time steps or a vector with three named values from, to, by specifying the simulation time steps. The 'from-to-by' form can be edited with editParms.
init	Initial values (start values) of the state variable given as named vector.
parms	A vector or list (depending on the respective class) of constant parameters.
inputs	Optional time-dependend input variables (matrix or data frame).
solver	The solver used to integrate the model.
initfunc	The function is called by the initialize mechanism and allows direct access and manipulation of all slots of the object in creation

Details

These functions provide an alternative way to create `simObj` instances in addition to the standard S4 `new` mechanism. The functions are provided mainly for compatibility with older versions of **simecol**.

See `simecol-package` and the examples for details about the slots.

Value

The function returns an S4 object of type `odeModel`, `rwalkModel`, `gridModel`

See Also

`new`, `simecol-package`

Examples

```
## (1) Define and run your own simecol model with new ==================

lv <- new("odeModel",
  main = function (time, init, parms) {
    with(as.list(c(init, parms)), {
      dn1 <-   k1 * N1 - k2 * N1 * N2
      dn2 <- - k3 * N2 + k2 * N1 * N2
      list(c(dn1, dn2))
    })
  },
  parms  = c(k1 = 0.2, k2 = 0.2, k3 = 0.2),
  times  = c(from = 0, to = 100, by = 0.5),
  init   = c(N1 = 0.5, N2 = 1),
  solver = "lsoda"
)

## ... or use the generating function ---------------------------------

lv <- odeModel(
  main = function (time, init, parms) {
    with(as.list(c(init, parms)), {
      dn1 <-   k1 * N1 - k2 * N1 * N2
      dn2 <- - k3 * N2 + k2 * N1 * N2
      list(c(dn1, dn2))
    })
  },
  parms  = c(k1 = 0.2, k2 = 0.2, k3 = 0.2),
  times  = c(from = 0, to = 100, by = 0.5),
  init   = c(N1 = 0.5, N2 = 1),
  solver = "lsoda"
)

lv <- sim(lv)
plot(lv)
```

```
## (2) Conway's Game of Life ================================================

set.seed(23)  # to make it reproducible

conway <- new("gridModel",
  main = function(time, x, parms) {
    nb     <- eightneighbours(x)
    surviv <- (x >  0 & (nb %in% parms$srv))
    gener  <- (x == 0 & (nb %in% parms$gen))
    x      <- (surviv + gener) > 0
    return(x)
  },
  parms  = list(srv = c(2, 3), gen = 3),
  times  = 1:17,
  init   = matrix(round(runif(1000)), ncol=40),
  solver = "iteration"
)

sim(conway, animate=TRUE)
```

`p.constrain` *Transform Data Between Unconstrained and Box-constrained Scale*

Description

These functions can be used to transform a vector of data or parameters between unconstrained [-Inf, Inf] and box-constrained representation (interval [lower, upper]).

Usage

```
p.constrain(p, lower = -Inf, upper = Inf, f = 1)
p.unconstrain(p, lower = -Inf, upper = Inf, f = 1)
```

Arguments

`p`	vector of data (e.g. model parameters),
`lower, upper`	vectors with lower resp. upper bounds used for transformation,
`f`	optional scaling factor.

Details

These functions are employed by `fitOdeModel` `ssqOdeModel` in order to be able to use the unconstrained optimizers of `optim` for constrained optimization.

The transformation functions are

$$p' = \tan(\pi/2 \cdot (2p - upper - lower)/(upper - lower)) \cdot 1/f$$

and its inverse

$$p = (upper + lower)/2 + (upper - lower) \cdot \arctan(f \cdot p')/\pi$$

.

Value

vector with transformed (resp. back-transformed) values.

References

This trick seems to be quite common, but in most cases it is preferred to apply optimizers that can handle constraints internally.

Reichert, T. (1998) AQUASIM 2.0 User Manual. Computer Program for the Identification and Simulation of Aquatic Systems. Swiss Federal Institute for Environmental Science and Technology (EAWAG), CH - 8600 Duebendorf Switzerland, https://www.eawag.ch/de/abteilung/siam/software/.

See Also

fitOdeModel, ssqOdeModel

Examples

```
xx <- seq(-100, 100, 2)
plot(xx, yy<-p.constrain(xx, -20, 45), type="l")
points(p.unconstrain(yy, -20, 45), yy, col="red")
```

---

parms *Accessor Functions for 'simObj' Objects*

Description

Get or set simulation model parameters, main or sub-equations, initial values, time steps or solvers and extract simulation results.

Usage

```
parms(obj, ...)
parms(obj) <- value

main(obj, ...)
main(obj) <- value

equations(obj, ...)
equations(obj) <- value
```

```
init(obj, ...)
init(obj) <- value

inputs(obj, ...)
inputs(obj) <- value

times(obj, ...)
times(obj) <- value

solver(obj, ...)
solver(obj) <- value

#observer(obj, ...)
#observer(obj) <- value

initfunc(obj, ...)
initfunc(obj) <- value

out(obj, ...)
out(obj) <- value
```

Arguments

obj	A valid simObj instance.
value	Named list, vector, function or other data structure (depending on the slot and model class) with the same structure as the value returned by parms. Either all or a subset of values (e.g. single elements of vectors or lists) can be changed at once.
...	Reserved for method consistency.

Details

These are the accessing functions for parms, times etc.

Please take care of the semantics of your model when changing slots. All, element names, data structure and values have to correspond to you model object definition. For example in init the applied names must exactly correspond to the names and number (!) of state variables. The restrictions of parms or equations are less strict (additional values for "future use" are allowed).

The function times allows either to assign or to modify a special vector with three elements named from, to and by or to overwrite times with an un-named sequence (e.g. seq(1,100,0.1).

To ensure object consistency function out cannot assign arbitrary values. It can only extract or delete the contents (by assigning NULL) of the out-slot.

Value

A list, named vector, matrix or function (for main slot) or list of functions (equation slot) or other appropriate data structure depending on the class of the model object.

See Also

General explanation of the slots can be found in simecol-package.

Usage of the observer slot is found in the special help file observer.

Examples

```
data(lv)
parms(lv)
parms(lv)       <- c(k1 = 0.2, k2 = 0.5, k3 = 0.3)
parms(lv)["k2"] <- 0.5

data(conway)
parms(conway)
parms(conway)$srv <- c(2, 2)
parms(conway)

## add a new named parameter value
parms(lv)["dummy"] <- 1
## remove dummy parameter
parms(lv) <- parms(lv)[names(parms(lv)) != "dummy"]

## simulation and extraction of outputs
lv <- sim(lv)
o <- out(lv)

## remove outputs from object
out(lv) <- NULL

## store object persistently to the disk
## Not run:
save(lv, file = "lv.Rdata")           # in binary form
dput(as.list(lv), file = "lv-list.R") # in text form

## End(Not run)
```

---

pcuseries *Generate Plackett Bivariate Random Numbers*

Description

Generate bivariate uniform random numbers according to the Plackett distribution.

Usage

```
pcu(x, alpha = rho2alpha(rho), rho)
pcuseries(n, alpha = rho2alpha(rho), rho, min = 0, max = 1)
alpha2rho(alpha)
rho2alpha(rho)
```

Arguments

n	number of observations.
x	vector of uniformly [0, 1] distributed real numbers.
alpha	association coefficient of the Plackett distribution.
rho	Pearson correlation coefficient.
min, max	lower and upper limits of the distribution. Must be finite.

Details

The functions can be used to generate bivariate distributions with uniform marginals. Function pcu generates a vector of uniform random values of length(x) which are correlated to the corresponding vector x, pcuseries generates an auto-correlated series, and alpha2rho resp. rho2alpha convert between the Pearson correlation coefficient and the association measure of the Plackett distribution.

References

Johnson, M., Wang, C., & Ramberg, J. (1984). Generation of multivariate distributions for statistical applications. *American Journal of Mathematical and Management Sciences*, **4**, 225-248.

Nelsen, R. B. (2006). *An Introduction to Copulas*. Springer, New York.

See Also

runif

Examples

```
x <- runif(100)
y <- pcu(x, rho = 0.8)
plot(x, y)
cor(x, y)

x <- pcuseries(1000, rho=0.8)
plot(x, type="l")
acf(x)
pacf(x)
```

---

peaks *Find Peaks Within xy-Data*

Description

The function returns maxima (values which have only smaller neighbours) and minima (values which have only larger neighbours).

Usage

```
peaks(x, y=NULL, mode="maxmin")
```

Arguments

x, y	the coordinates of given points.
mode	specifies if both maxima and minima (mode="maxmin") or only maxima (mode="max") or minima (mode="min") are requested.

Value

A list with x and y coordinates of all peaks.

See Also

approx, upca

Examples

```
x <- sin(seq(0, 10, 0.1))
plot(x)
points(peaks(x), col="red", pch=15)
```

---

plot-methods *Methods for Function plot in Package 'simecol'*

Description

Methods for function plot in package **simecol**.

Usage

```
  ## S4 method for signature 'simObj,missing'
plot(x, y, ...)
  ## S4 method for signature 'odeModel,missing'
plot(x, y, ...)
  ## S4 method for signature 'odeModel,odeModel'
plot(x, y, ...)
  ## S4 method for signature 'gridModel,missing'
plot(x, y, index=1:length(x@out), delay=0, ...)
  ## S4 method for signature 'rwalkModel,missing'
plot(x, y, index=1:length(x@out), delay=0, ...)
```

Arguments

x	an object of class simObj,
y	either a second odeModel object or ignored,
index	index of time steps to be plotted,
delay	delay (in ms) between consecutive images (for gridModels) or xy-plots (for rwalkModels),
...	optional plotting parameters.

Methods

x = "ANY", y = "ANY" Generic function: see plot.

x = "simObj", ... template function, does nothing and only issues a warning.

x = "odeModel", ... plots time series of the state variables where one or more odeModel objects can be supplied. Optional plotting parameters are possible, too. See plot.deSolve for details.

x = "gridModel", ... displays a series of images for the simulated grid.

x = "rwalkModel", ... displays a series of x-y plots of the simulated individuals.

---

print-methods *Methods for Function 'print' in Package 'simecol'*

Description

Methods for function print in Package **simecol**.

Usage

```
  ## S4 method for signature 'simObj'
print(x, all = FALSE, ...)
```

Arguments

x	an object of class simObj or one of its subclasses.
all	specifies whether all slots are printed. Default is that only not-empty slots are printed and the contents of out are suppressed.
...	optional parameters passed to print.

Methods

x = "ANY" generic function: see print.

x = "simObj" prints the contents (slots) of the simObj object.

sEdit *Simple editing*

Description

Simple Editing of Vectors, Lists of Vectors and Other Objects.

Usage

```
sEdit(x, title = "Please enter values:")
```

Arguments

x	A named object that you want to edit.
title	A title for the dialog box.

Details

If called with a vector or list of vectors and if **Tcl/Tk** is installed, a dialog box is shown in which data can be entered. If the x is not of type vector or list of vectors, a default editing method is called.

Value

An object with the same type like x.

See Also

edit editParms

Examples

```
## Not run:
require("tcltk")
## named vector
vec  <- c(a = 1, b = 20, c = 0.03)
new  <- sEdit(vec)
## unnamed vector
sEdit(numeric(10))
## list of vectors
lst <- list(vec = vec, test = 1:10)
sEdit(lst)
## list with numeric and character vectors mixed
lst <- list(vec = vec, test = c("a", "b", "c"))
sEdit(lst)

## End(Not run)
```

seedfill *Color Fill Algorithm*

Description

Fills a bounded area within a numeric matrix with a given number (color).

Usage

```
seedfill(z, x=1, y=1, fcol=0, bcol=1, tol=1e-6)
```

Arguments

z	a matrix containing an image (double precision values are possible).
x, y	start coordinates of the filled area.
fcol	numeric value of the fill color.
bcol	numeric value of the border value.
tol	numeric value of border color tolerance.

Details

The function implements a basic color fill algorithm for use in image manipulation or cellular automata.

Value

A matrix with the same structure as z.

See Also

neighbours

Examples

```
# define a matrix
z<-matrix(0, nrow=20, ncol=20)

# draw some lines
z[10,]<-z[,10] <- 1
z[5,] <-z[,5]  <- 3

# plot matrix and filled variants
par(mfrow=c(2, 2))
image(z)
image(seedfill(z))
image(seedfill(z ,x=15, y=15, fcol=1, bcol=3))
image(seedfill(z, x=7, y=7, fcol=3, bcol=1))
```

sim-methods *Simulation of 'simObj' model objects*

Description

This function provides the core functionality of the 'simecol' package. Several methods depending on the class of the model are available.

Usage

```
sim(obj, initialize=TRUE, ...)
# sim(obj, animation=FALSE, delay=0, ...)
```

Arguments

obj	an object of class simObj or one of its subclasses.
initialize	re-initialize the object if the object contains a user-defined initializing function (initfunc). Setting initialize to FALSE can be useful to avoid time-consuming computations during initialization or to reproduce simulations of models which assign random values during the initialization process.
animation	logical value to switch animation on (for classes gridModel and rwalkModel.
delay	delay (in ms and in addition to the time needed for the simulation) between consecutive images (for gridModels) or xy-plots (for rwalkModels).
...	optional parameters passed to the solver function (e.g. hmax for lsoda).

Details

sim re-initializes the model object (if initialize==TRUE and calls the appropriate solver, specified in the solver-slot. Objects of class rwalkModel and indbasedModel are simulated by the default simObj method. If you derive own sublasses from simObj it may be neccessary to write an appropriate sim method and/or solver function.

Value

The function returns the complete simObj instance with the simulation results in the out slot.

Methods

obj = "simObj" simulates the respective model object with optional animation.

obj = "odeModel" simulates the respective model object.

obj = "indbasedModel" simulates the respective model object with optional animation.

obj = "gridModel" simulates the respective model object with optional animation.

Examples

```
data(lv)
plot(sim(lv))

lv2 <- lv
parms(lv2)["k1"] <- 0.5
lv2 <- sim(lv2)
plot(out(lv2))
```

---

ssqOdeModel *Sum of Squares Between odeModel and Data*

Description

Compute the sum of squares between a given data and an odeModel object.

Usage

```
ssqOdeModel(p, simObj, obstime, yobs,
  sd.yobs = as.numeric(lapply(yobs, sd)),
  initialize = TRUE, lower. = -Inf, upper. = Inf, weights = NULL,
  debuglevel = 0, ..., pnames = NULL)
```

Arguments

p	vector of named parameter values of the model (can be a subset),
simObj	a valid object of class odeModel,
obstime	vector with time steps for which observational data are available,
yobs	data frame with observational data for all or a subset of state variables. Their names must correspond exacly with existing names of state variables in the odeModel.
sd.yobs	vector of given standard deviations for all observational variables given in yobs. If no standard deviations are given, these are estimated from yobs.
initialize	optional boolean value whether the simObj should be re-initialized after the assignment of new parameter values. This can be necessary in certain models to assign consistent values to initial state variables if they depend on parameters.
lower., upper.	named vectors with lower and upper bounds used in the optimisation,
weights	optional weights to be used in the fitting process. Should be NULL or a data frame with the same structure as yobs. If non-NULL, weighted least squares is used with weights (that is, minimizing sum(w*e^2)); otherwise ordinary least squares is used.
debuglevel	a positive number that specifies the amount of debugging information printed,
...	additional parameters passed to the solver method (e.g. lsoda),
pnames	names of the parameters, optionally passed from fitOdeModel. This argument is a workaround for R versions below 2.8.1. It may be removed in future versions of **simecol**.

Details

This is the default function called by function fitOdeModel. The source code of this function can be used as a starting point to develop user-defined optimization criteria (cost functions).

Value

The sum of squared differences between yobs and simulation, by default weighted by the inverse of the standard deviations of the respective variables.

See Also

fitOdeModel, optim, p.constrain

Examples

```
data(chemostat)
cs1 <- chemostat

## generate some noisy data
parms(cs1)[c("vm", "km")] <- c(2, 10)
times(cs1) <- c(from = 0, to = 20, by = 2)
yobs <- out(sim(cs1))
obstime <- yobs$time
yobs$time <- NULL
yobs$S <- yobs$S + rnorm(yobs$S, sd = 0.1 * sd(yobs$S))*2
yobs$X <- yobs$X + rnorm(yobs$X, sd = 0.1 * sd(yobs$X))

## SSQ between model and data
ssqOdeModel(NULL, cs1, obstime, yobs)

## SSQ between model and data, different parameter set
ssqOdeModel(p=c(vm=1, km=2), cs1, obstime, yobs)

## SSQ between model and data, downweight second observation
## (both variables)
weights <- data.frame(X=rep(1, nrow(yobs)), S = rep(1, nrow=(yobs)))
ssqOdeModel(p=c(vm=1, km=2), cs1, obstime, yobs, weights=weights)

## downweight 3rd data set (row)
weights[3,] <- 0.1
ssqOdeModel(p=c(vm=1, km=2), cs1, obstime, yobs, weights=weights)

## give one value double weight (e.g. 4th value of S)
weights$S[4] <- 2
ssqOdeModel(p=c(vm=1, km=2), cs1, obstime, yobs, weights=weights)
```

upca *The Uniform Period Chaotic Amplitude Model*

Description

simecol example: resource-predator-prey model, which is able to exhibit chaotic behaviour.

Usage

```
data(upca)
```

Format

S4 object according to the odeModel specification. The object contains the following slots:

`main` The differential equations for predator prey and resource with:

- `u` resource (e.g. grassland or phosphorus),
- `v` producer (prey),
- `w` consumer (predator).

`equations` Two alternative (and switchable) equations for the functional response.

`parms` Vector with the named parameters of the model, see references for details.

`times` Simulation time and integration interval.

`init` Vector with start values for `u`, `v` and `w`.

`solver` Character string with the integration method.

Details

To see all details, please have a look into the implementation below and the original publications.

References

Blasius, B., Huppert, A., and Stone, L. (1999) Complex dynamics and phase synchronization in spatially extended ecological systems. *Nature*, **399** 354–359.

Blasius, B. and Stone, L. (2000) Chaos and phase synchronization in ecological systems. *International Journal of Bifurcation and Chaos*, **10** 2361–2380.

See Also

sim, parms, init, times.

Examples

```
##=============================================
## Basic Usage:
##   explore the example
##=============================================
data(upca)
plot(sim(upca))

# omit stabilizing parameter wstar
parms(upca)["wstar"] <- 0
plot(sim(upca))

# change functional response from
# Holling II (default) to Lotka-Volterra
equations(upca)$f <- function(x, y, k) x * y
plot(sim(upca))

##=============================================
## Implementation:
##   The code of the UPCA model
##=============================================
upca <- new("odeModel",
  main = function(time, init, parms) {
    u      <- init[1]
    v      <- init[2]
    w      <- init[3]
    with(as.list(parms), {
      du <-  a * u           - alpha1 * f(u, v, k1)
      dv <- -b * v           + alpha1 * f(u, v, k1) +
                             - alpha2 * f(v, w, k2)
      dw <- -c * (w - wstar) + alpha2 * f(v, w, k2)
      list(c(du, dv, dw))
    })
  },
  equations  = list(
    f1 = function(x, y, k){x*y},           # Lotka-Volterra
    f2 = function(x, y, k){x*y / (1+k*x)}  # Holling II
  ),
  times  = c(from=0, to=100, by=0.1),
  parms  = c(a=1, b=1, c=10, alpha1=0.2, alpha2=1,
    k1=0.05, k2=0, wstar=0.006),
  init = c(u=10, v=5, w=0.1),
  solver = "lsoda"
)

equations(upca)$f <- equations(upca)$f2
```

Index