Package 'SoilR'

July 6, 2018

Title Models of Soil Organic Matter Decomposition

Version 1.1-53

Date 2017-05-04

Author Carlos A. Sierra, Markus Mueller

Maintainer Markus Mueller <mamueller@bgc-jena.mpg.de>

Description Functions for modeling Soil Organic Matter decomposition in terrestrial ecosystems with linear and nonlinear models.

License GPL-3

Depends deSolve,methods,parallel,expm,RUnit

Suggests FME,lattice,MASS

LazyData TRUE

R topics documented:

`SoilR-package` *SOILR*

Description

The package allows you to study compartmental Soil models.

Details

The typical workflow consists of the following steps:

- Create a model
- Inspect it

The simplest way of creating a model is to use one of the top level functions for predefined models: `predefinedModels`.
The objects returned by these functions can be of different type, usually either `Model` or `Model14`. To inspect the behaviour of a model object these classes provide several methods to be found in their respective descriptions: (`Model` or `Model_14`)
If none of the predefined models fits your needs you can assemble your own model. The functions that create it are the constructors of the classes Model or Model_14. By convention they have the same name as the class and are desribed here:
`Model`,
`Model_14`.

AbsoluteFractionModern_from_Delta14C *AbsoluteFractionModern_from_Delta14C S4 generic*

Description

no Description

Usage

```
AbsoluteFractionModern_from_Delta14C(delta14C)
```

Arguments

delta14C	see the method arguments for details

Methods

AbsoluteFractionModern_from_Delta14C,matrix-method
AbsoluteFractionModern_from_Delta14C,numeric-method

AbsoluteFractionModern_from_Delta14C,matrix-method *AbsoluteFractionModern_from_Delta14C,matrix-method Converts from Delta14C to Absolute Fraction Modern*

Description

This method produces a matrix of Delta14C values from a Matrix or number of Absolute Fraction Modern

Usage

```
## S4 method for signature 'matrix'
AbsoluteFractionModern_from_Delta14C(delta14C)
```

Arguments

delta14C	: of class matrix, An object of class matrix containing the values in Delta14C format

AbsoluteFractionModern_from_Delta14C,numeric-method *AbsoluteFractionModern_from_Delta14C,numeric-method Converts from Delta14C to Absolute Fraction Modern*

Description

Converts a number or vector containing Delta14C values to the appropriate Absolute Fraction Modern values. Have a look at the methods for details.

Usage

```
## S4 method for signature 'numeric'
AbsoluteFractionModern_from_Delta14C(delta14C)
```

Arguments

delta14C	: of class numeric, A numeric object containing the values in Delta14C format

add_plot *add_plot S4 generic*

Description

no Description

Usage

```
add_plot(x,
...)
```

Arguments

x	see the method arguments for details
...	see the method arguments for details

Methods

add_plot,TimeMap-method

add_plot,TimeMap-method *add_plot,TimeMap-method overview plot*

Description

The method adds a simple overview plot of a the scalar, vector, matrix or arrayvalued function plotting all time dependent component.

Usage

```
## S4 method for signature 'TimeMap'
add_plot(x,
...)
```

Arguments

x	: of class TimeMap, the TimeMap Object to be plotted
...	: other plot parameters not implemented yet

as.character *creates a character representation of the object in question*

Description

This function computes the carbon content of the pools as function of time

Usage

```
as.character(object)
```

Arguments

object	The object to be printed.

Author(s)

Carlos A. Sierra <csierra@bgc-jena.mpg.de>, Markus Mueller <mamueller@bgc-jena.mpg.de>

as.character,TimeMap-method

as.character,TimeMap-method convert TimeMap Objects to something printable.

Description

This method is needed to print a TimeMap object.

Usage

```
## S4 method for signature 'TimeMap'
as.character(x,
...)
```

Arguments

x	: of class TimeMap, An Object of class time map
...	: will be ignored

AWBmodel

Implementation of the microbial model AWB (Allison, Wallenstein, Bradford, 2010)

Description

This function implements the microbial model AWB (Allison, Wallenstein, Bradford, 2010), a four-pool model with a microbial biomass, enzyme, SOC and DOC pools. It is a special case of the general nonlinear model.

Usage

```
AWBmodel(t,
V_M=1e+08,
V_m=1e+08,
r_B=2e-04,
r_E=5e-06,
r_L=0.001,
a_BS=0.5,
epsilon_0=0.63,
epsilon_s=-0.016,
Km_0=500,
Km_u0=0.1,
Km_s=0.5,
Km_us=0.1,
Ea=47,
R=0.008314,
Temp1=20,
Temp2=20,
```

```
ival=c(B = 2.19159, E = 0.0109579, S = 111.876, D = 0.00144928),
I_S=0.005,
I_D=0.005)
```

Arguments

t	vector of times (in hours) to calculate a solution.
V_M	a scalar representing the maximum rate of uptake (mg DOC cm-3 h-1). Equivalent to V_maxuptake0 in original paper.
V_m	a scalar representing the maximum rate of decomposition of SOM (mg SOM cm-3 h-1). Equivalent to V_max0 in original paper.
r_B	a scalar representing the rate constant of microbial death (h-1). Equivalent to r_death in original publication.
r_E	a scalar representing the rate constant of enzyme production (h-1). Equivalent to r_EnzProd in original publication.
r_L	a scalar representing the rate constant of enzyme loss (h-1). Equivalent to r_EnzLoss in original publication.
a_BS	a scalar representing the fraction of the dead microbial biomass incorporated to SOC. MICtoSOC in original publication.
epsilon_0	a scalar representing the intercept of the CUE function (mg mg-1). CUE_0 in original paper.
epsilon_s	a scalar representing the slope of the CUE function (degree-1). CUE_slope in original paper.
Km_0	a scalar representing the intercept of the half-saturation constant of SOC as a function of temperature (mg cm-3).
Km_u0	a scalar representing the intercept of the half saturation constant of uptake as a function of temperature (mg cm-3).
Km_s	a scalar representing the slope of the half saturation constant of SOC as a function of temperature (mg cm-3 degree-1).
Km_us	a scalar representing the slope of the half saturation constant of uptake as a function of temperature (mg cm-3 degree-1).
Ea	a scalar representing the activation energy (kJ mol-1).
R	a scalar representing the gas constant (kJ mol-1 degree-1).
Temp1	a scalar representing the temperature in the output vector.
Temp2	a scalar representing the temperature in the transfer matrix.
ival	a vector of length 4 with the initial values for the pools (mg cm-3).
I_S	a scalar with the inputs to the SOC pool (mg cm-3 h-1).
I_D	a scalar with the inputs to the DOC pool (mg cm-3 h-1).

Details

This implementation containts default parameters presented in Allison et al. (2010).

Value

An object of class NlModel that can be further queried.

References

Allison, S.D., M.D. Wallenstein, M.A. Bradford. 2010. Soil-carbon response to warming dependent on microbial physiology. Nature Geoscience 3: 336-340.

Examples

```
hours=seq(0,800,0.1)

#Run the model with default parameter values
bcmodel=AWBmodel(t=hours)
Cpools=getC(bcmodel)
##Time solution
# fixme mm:
# the next line causes trouble on Rforge Windows patched build
matplot(hours,Cpools,type="l",ylab="Concentrations",xlab="Hours",lty=1,ylim=c(0,max(Cpool
##State-space diagram
plot(as.data.frame(Cpools))
```

---

bacwaveModel *Implementation of the microbial model Bacwave (bacterial waves)*

Description

This function implements the microbial model Bacwave (bacterial waves), a two-pool model with a bacterial and a substrate pool. It is a special case of the general nonlinear model.

Usage

```
bacwaveModel(t,
umax=0.063,
ks=3,
theta=0.23,
Dmax=0.26,
kd=14.5,
kr=0.4,
Y=0.44,
ival=c(S0 = 0.5, X0 = 1.5),
BGF=0.15,
ExuM=8,
ExuT=0.8)
```

Arguments

t	vector of times (in hours) to calculate a solution.
umax	a scalar representing the maximum relative growth rate of bacteria (hr-1)
ks	a scalar representing the substrate constant for growth (ug C /ml soil solution)
theta	a scalar representing soil water content (ml solution/cm3 soil)
Dmax	a scalar representing the maximal relative death rate of bacteria (hr-1)
kd	a scalar representing the substrate constant for death of bacteria (ug C/ml soil solution)

kr	a scalar representing the fraction of death biomass recycling to substrate (unitless)
Y	a scalar representing the yield coefficient for bacteria (ug C/ugC)
ival	a vector of length 2 with the initial values for the substrate and the bacterial pools (ug C/cm3)
BGF	a scalar representing the constant background flux of substrate (ug C/cm3 soil/hr)
ExuM	a scalar representing the maximal exudation rate (ug C/(hr cm3 soil))
ExuT	a scalar representing the time constant for exudation, responsible for duration of exudation (1/hr).

Details

This implementation containts default parameters presented in Zelenev et al. (2000). It produces nonlinear damped oscillations in the form of a stable focus.

Value

An object of class NlModel that can be further queried.

References

Zelenev, V.V., A.H.C. van Bruggen, A.M. Semenov. 2000. "BACWAVE," a spatial-temporal model for traveling waves of bacterial populations in response to a moving carbon source in soil. Microbail Ecology 40: 260-272.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
hours=seq(0,800,0.1)
#
#Run the model with default parameter values
bcmodel=bacwaveModel(t=hours)
Cpools=getC(bcmodel)
#
#Time solution
matplot(hours,Cpools,type="l",ylab="Concentrations",xlab="Hours",lty=1,ylim=c(0,max(Cpool
legend("topleft",c("Substrate", "Microbial biomass"),lty=1,col=c(1,2),bty="n")
#
#State-space diagram
plot(Cpools[,2],Cpools[,1],type="l",ylab="Substrate",xlab="Microbial biomass")
#
#Microbial biomass over time
plot(hours,Cpools[,2],type="l",col=2,xlab="Hours",ylab="Microbial biomass")
```

bind.C14curves *Binding of pre- and post-bomb Delta14C curves*

Description

This function takes a pre- and a post-bomb curve, binds them together, and reports the results back either in years BP or AD.

Usage

```
bind.C14curves(prebomb,
postbomb,
time.scale)
```

Arguments

prebomb	A pre-bomb radiocarbon dataset. They could be either IntCal09 or IntCal13.
postbomb	A post-bomb radiocarbon dataset. They could be any of the datasets in Hua2013.
time.scale	A character indicating whether to report the results in years before present BP or anno domini AD.

Value

A data.frame with 3 columns: years in AD or BP, the atmospheric Delta14C value, the standard deviation of the Delta14C value.

BoundFc *BoundFc S4 generic*

Description

no Description

Usage

```
BoundFc(format,
...)
```

Arguments

format	see the method arguments for details
...	see the method arguments for details

Methods

BoundFc,character-method
BoundFc,missing-method

`BoundFc,character-method`

BoundFc,character-method A constructor

Description

The method constructs an object from a format string and other parameters which are passed on to `TimeMap`

Usage

```
## S4 method for signature 'character'
BoundFc(format,
...)
```

Arguments

`format`	: of class character, a string that specifies the format used to represent the atmospheric fraction. Possible values are "Delta14C" which is the default or "afn" the Absolute Fraction Normal representation
`...`	: passed on to TimeMap

`BoundFc,missing-method`

BoundFc,missing-method A constructor

Description

The method constructs an object from a list which must contain a

Usage

```
## S4 method for signature 'missing'
BoundFc(format,
...)
```

Arguments

`format`	: of class missing, If this method is called the format argument was not present.
`...`	: passed on to TimeMap

BoundFc-class — *Objects containing the atmospheric 14C fraction and the format it is provided in.*

Description

no Description

Methods

No exported methods directly defined for class BoundFc:

Methods inherited from superclasses:

from class TimeMap:

GeneralDecompOp `signature(object = "TimeMap")`: ... `GeneralDecompOp,TimeMap-method`

GeneralInFlux `signature(object = "TimeMap")`: ... `GeneralInFlux,TimeMap-method`

TimeMap `signature(map = "TimeMap", starttime = "ANY", endtime = "ANY", times = "`
... `TimeMap,TimeMap,ANY,ANY,ANY,ANY-method`

add_plot `signature(x = "TimeMap")`: ... `add_plot,TimeMap-method`

as.character `signature(x = "TimeMap")`: ... `as.character,TimeMap-method`

getFunctionDefinition `signature(object = "TimeMap")`: ... `getFunctionDefinition,TimeMap-me`

getTimeRange `signature(object = "TimeMap")`: ... `getTimeRange,TimeMap-method`

Superclasses

`TimeMap-class`
`Fc-class`

Constructors

`BoundFc`

BoundInFlux — *constructor for BoundInFlux*

Description

The method internally calls `TimeMap` and expects the same kind of arguments

Usage

```
BoundInFlux(...)
```

Arguments

`...`	passed on to `TimeMap`

BoundInFlux-class *BoundInFlux S4 class*

Description

defines a time dependent inputrate as function of time and including the domain where the function is well defined. This can be used to avoid interpolations out of range when mixing different time dependent data sets

Methods

No exported methods directly defined for class BoundInFlux:

Methods inherited from superclasses:

from class InFlux:

GeneralInFlux `signature(object = "InFlux")`: ... GeneralInFlux,InFlux-method

from class TimeMap:

GeneralDecompOp `signature(object = "TimeMap")`: ... GeneralDecompOp,TimeMap-method

GeneralInFlux `signature(object = "TimeMap")`: ... GeneralInFlux,TimeMap-method

TimeMap `signature(map = "TimeMap", starttime = "ANY", endtime = "ANY", times = "`
... TimeMap,TimeMap,ANY,ANY,ANY,ANY-method

add_plot `signature(x = "TimeMap")`: ... add_plot,TimeMap-method

as.character `signature(x = "TimeMap")`: ... as.character,TimeMap-method

getFunctionDefinition `signature(object = "TimeMap")`: ... getFunctionDefinition,TimeMap-me

getTimeRange `signature(object = "TimeMap")`: ... getTimeRange,TimeMap-method

Superclasses

InFlux-class
TimeMap-class

Constructors

BoundInFlux

BoundLinDecompOp *BoundLinDecompOp S4 generic*

Description

no Description

Usage

```
BoundLinDecompOp(map,
...)
```

Arguments

map	see the method arguments for details
...	see the method arguments for details

Methods

BoundLinDecompOp,ANY-method
BoundLinDecompOp,UnBoundLinDecompOp-method

BoundLinDecompOp,ANY-method *BoundLinDecompOp,ANY-method a constructor*

Description

Creates a BoundLinDecompOp Object.

Usage

```
## S4 method for signature 'ANY'
BoundLinDecompOp(map,
...)
```

Arguments

map	: of class ANY, passed on to TimeMap
...	: passed on to TimeMap

`BoundLinDecompOp,UnBoundLinDecompOp-method`
BoundLinDecompOp,UnBoundLinDecompOp-method convert a UnBoundLinDecompOp to a BoundLinDecompOp

---

Description

The method creates a BoundLinDecompOp consisting of a constant time dependent function and the limits of its domain (starttime and endtime) set to -Inf and Inf respectively

Usage

```
## S4 method for signature 'UnBoundLinDecompOp'
BoundLinDecompOp(map,
starttime=-Inf,
endtime=Inf)
```

Arguments

`map`	: of class UnBoundLinDecompOp
`starttime`	: the left hand boundary of the valid time interval
`endtime`	: the right hand boundary of the valid time interval

`BoundLinDecompOp-class`
a decomposition operator described by a matrix valued function of time

---

Description

no Description

Methods

No exported methods directly defined for class BoundLinDecompOp:

Methods inherited from superclasses:

from class DecompOp:

GeneralDecompOp `signature(object = "DecompOp")`: ... `GeneralDecompOp,DecompOp-method`

from class TimeMap:

GeneralDecompOp `signature(object = "TimeMap")`: ... `GeneralDecompOp,TimeMap-method`
GeneralInFlux `signature(object = "TimeMap")`: ... `GeneralInFlux,TimeMap-method`
TimeMap `signature(map = "TimeMap", starttime = "ANY", endtime = "ANY", times = "`
... `TimeMap,TimeMap,ANY,ANY,ANY,ANY-method`
add_plot `signature(x = "TimeMap")`: ... `add_plot,TimeMap-method`
as.character `signature(x = "TimeMap")`: ... `as.character,TimeMap-method`
getFunctionDefinition `signature(object = "TimeMap")`: ... `getFunctionDefinition,TimeMap-me`
getTimeRange `signature(object = "TimeMap")`: ... `getTimeRange,TimeMap-method`

Superclasses

DecompOp-class
TimeMap-class

Constructors

BoundLinDecompOp

C14Atm *Atmospheric 14C fraction*

Description

Atmospheric 14C fraction in units of Delta14C for the bomb period in the northern hemisphere.

Usage

```
data(C14Atm)
```

Format

A data frame with 108 observations on the following 2 variables.

V1 a numeric vector

V2 a numeric vector

Note

This function will be deprecated soon. Please use C14Atm_NH or Hua2013 instead.

Examples

```
#Notice that C14Atm is a shorter version of C14Atm_NH
require("SoilR")
data("C14Atm_NH")
plot(C14Atm_NH,type="l")
lines(C14Atm,col=2)
```

C14Atm_NH *Post-bomb atmospheric 14C fraction*

Description

Atmospheric 14C concentrations for the post-bomb period expressed as Delta 14C in per mile. This dataset contains a combination of observations from locations in Europe and North America. It is representative for the Northern Hemisphere.

Usage

```
data(C14Atm_NH)
```

Format

A data frame with 111 observations on the following 2 variables.

`YEAR` a numeric vector with year of measurement.

`Atmosphere` a numeric vector with the Delta 14 value of atmospheric CO2 in per mil.

Examples

```
plot(C14Atm_NH,type="l")
```

CenturyModel *Implementation of the Century model*

Description

This function implements the Century model as described in Parton et al. (1987).

Usage

```
CenturyModel(t,
ks=c(k.STR = 0.094, k.MET = 0.35, k.ACT = 0.14, k.SLW = 0.0038, k.PAS = 0.00013)
C0=c(0, 0, 0, 0, 0),
In,
LN,
Ls,
clay=0.2,
silt=0.45,
xi=1,
solver=deSolve.lsoda.wrapper)
```

Arguments

t	A vector containing the points in time where the solution is sought.
ks	A vector of lenght 5 containing the values of the decomposition rates for the different pools. Units in per week.
C0	A vector of length 5 containing the initial amount of carbon for the 5 pools.
In	A scalar or data.frame object specifying the amount of litter inputs by time (mass per area per week).
LN	A scalar representing the lignin to nitrogen ratio of the plant residue inputs.
Ls	A scalar representing the fraction of structural material that is lignin.
clay	Proportion of clay in mineral soil.
silt	Proportion of silt in mineral soil.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.

Value

A Model Object that can be further queried

References

Parton, W.J, D.S. Schimel, C.V. Cole, and D.S. Ojima. 1987. Analysis of factors controlling soil organic matter levels in Great Plain grasslands. Soil Science Society of America Journal 51: 1173–1179. Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

RothCModel. There are other predefinedModels and also more general functions like Model.

Examples

```
t=seq(0,52*200,1) #200 years
LNcorn=0.17/0.004 # Values for corn clover reported in Parton et al. 1987
Ex=CenturyModel(t,LN=0.5,Ls=0.1,In=0.1)
Ct=getC(Ex)
Rt=getReleaseFlux(Ex)

matplot(t,Ct,type="l", col=1:5,lty=1,ylim=c(0,max(Ct)*2.5),
ylab=expression(paste("Carbon stores (kg C", ha^-1,")")),xlab="Time (weeks)")
lines(t,rowSums(Ct),lwd=2)
legend("topright", c("Structural litter","Metabolic litter",
"Active SOM","Slow SOM","Passive SOM","Total Carbon"),
lty=1,lwd=c(rep(1,5),2),col=c(1:5,1),bty="n")

matplot(t,Rt,type="l",lty=1,ylim=c(0,max(Rt)*3),ylab="Respiration (kg C ha-1 week-1)",xla
lines(t,rowSums(Rt),lwd=2)
legend("topright", c("Structural litter","Metabolic litter",
"Active SOM","Slow SOM","Passive SOM","Total Respiration"),
lty=1,lwd=c(rep(1,5),2),col=c(1:5,1),bty="n")
```

ConstFc — *creates an object containing the initial values for the 14C fraction needed to create models in SoilR*

Description

The function returns an object of class ConstFc which is a building block for any 14C model in SoilR. The building blocks of a model have to keep iformation about the formats their data are in, because the high level function dealing wiht the models have to know. This function is actually a convienient wrapper for a call to R's standard constructor new, to hide its complexity from the user.

Usage

```
ConstFc(values=c(0),
format=Delta14C)
```

Arguments

values	a numeric vector
format	a character string describing the format e.g. "Delta14C"

Value

An object of class ConstFc that contains data and a format description that can later be used to convert the data into other formats if the conversion is implemented.

ConstFc-class — *ConstFc S4 class*

Description

no Description

Methods

No exported methods directly defined for class ConstFc:

Superclasses

Fc-class

Constructors

ConstFc

`ConstInFlux` *ConstInFlux S4 generic*

Description

no Description

Usage

```
ConstInFlux(map)
```

Arguments

`map`	see the method arguments for details

Methods

`ConstInFlux,numeric-method`

`ConstInFlux,numeric-method` *ConstInFlux,numeric-method constructor*

Description

the method converts a vector of constant input rates into an object of Class `ConstInFlux`.

Usage

```
## S4 method for signature 'numeric'
ConstInFlux(map)
```

Arguments

`map`	: of class numeric

ConstInFlux-class *ConstInFlux S4 class*

Description

defines a constant inputrate

Methods

Exported methods directly defined for class ConstInFlux:

getFunctionDefinition `signature(object = "ConstInFlux")`: ... getFunctionDefinition,ConstI
getTimeRange `signature(object = "ConstInFlux")`: ... getTimeRange,ConstInFlux-method

Methods inherited from superclasses:
from class InFlux:

GeneralInFlux `signature(object = "InFlux")`: ... GeneralInFlux,InFlux-method

Superclasses

InFlux-class

Constructors

ConstInFlux

ConstLinDecompOp *ConstLinDecompOp S4 generic*

Description

no Description

Usage

```
ConstLinDecompOp(mat)
```

Arguments

mat	see the method arguments for details

Methods

ConstLinDecompOp,matrix-method

`ConstLinDecompOp,matrix-method`
ConstLinDecompOp,matrix-method construct from matric

Description

This method creates a ConstLinDecompOp from a matrix The operator is assumed to act on the vector of carbon stocks by multiplication of the (time invariant) matrix from the left.

Usage

```
## S4 method for signature 'matrix'
ConstLinDecompOp(mat)
```

Arguments

`mat`	: of class matrix

`ConstLinDecompOp-class`
constant decomposition operator

Description

no Description

Methods

Exported methods directly defined for class ConstLinDecompOp:

getFunctionDefinition `signature(object = "ConstLinDecompOp")`: ... `getFunctionDefinition,C`

getMeanTransitTime `signature(object = "ConstLinDecompOp")`: ... `getMeanTransitTime,Const`

getTimeRange `signature(object = "ConstLinDecompOp")`: ... `getTimeRange,ConstLinDecompO`

getTransitTimeDistributionDensity `signature(object = "ConstLinDecompOp")`: ... `getTransitTimeDistributionDensity,ConstLinDecompOp-method`

Methods inherited from superclasses:

from class DecompOp:

GeneralDecompOp `signature(object = "DecompOp")`: ... `GeneralDecompOp,DecompOp-method`

Superclasses

`DecompOp-class`

Constructors

`ConstLinDecompOp`

cycling *Cycling analysis of compartmental matrices*

Description

Computes the fundamental matrix N, and the expected number of steps from a compartmental matrix A

Usage

```
cycling(A)
```

Arguments

A	A compartmental linear square matrix with cycling rates in the diagonal and transfer rates in the off-diagonal.

Value

A list with 2 objects: the fundamental matrix N, and the expected number of steps Et.

See Also

systemAge

DecompOp-class *decomposition operator*

Description

The decomposition operator is a necessary ingredient of any Model. Very generally it describes the fluxes between the pools and to the exterior as functions of time and the pool contents. SoilR arranges different decomposition operators into different classes, to determine which computations can be performed with them (which methods can be called on them). The simplest and least general decomposition operator is ConstLinDecompOp-class which can be created from a constant reservoir matrix. Since it is the least general (most specific) the additional information can be used to compute more and more specific results (more methods) not available for the more abstract sub classes (for instance BoundLinDecompOp-class that are necessary to model the more general situations where the decomposition and transfer rates are functions of time. The different decomposition operators are created by class-specific functions, called constructors. (see the constructors section of this help page)

Methods

Exported methods directly defined for class DecompOp:

GeneralDecompOp signature(object = "DecompOp"): ... GeneralDecompOp,DecompOp-method

Subclasses

UnBoundLinDecompOp
BoundLinDecompOp
DecompositionOperator
ConstLinDecompOp

Constructors

The class is abstract (`contains "VIRTUAL"`). It can therefore not be instanciated directly. Look at non virtual subclasses and their constructors!

There is also an abstract factory that produces instances of different subclasses depending on the input:

GeneralDecompOp

DecompositionOperator-class

deprecated decomposition operator class

Description

no Description

Methods

Exported methods directly defined for class DecompositionOperator:

getFunctionDefinition `signature(object = "DecompositionOperator")`: ... getFunctionDefinit

getTimeRange `signature(object = "DecompositionOperator")`: ... getTimeRange,Decomposit

Methods inherited from superclasses:

from class DecompOp:

GeneralDecompOp `signature(object = "DecompOp")`: ... GeneralDecompOp,DecompOp-method

Superclasses

DecompOp-class

`Delta14C_from_AbsoluteFractionModern` *Delta14C_from_AbsoluteFractionModern S4 generic*

Description

no Description

Usage

```
Delta14C_from_AbsoluteFractionModern(AbsoluteFractionModern)
```

Arguments

`AbsoluteFractionModern` A numeric object

Methods

`Delta14C_from_AbsoluteFractionModern,matrix-method`
`Delta14C_from_AbsoluteFractionModern,numeric-method`

`Delta14C_from_AbsoluteFractionModern,matrix-method` *Delta14C_from_AbsoluteFractionModern,matrix-method Converts Absolute Fraction Modern values to Delta14C*

Description

This method produces a matrix of Delta14C values from a matrix of values in Absolute Fraction Modern.

Usage

```
## S4 method for signature 'matrix'
Delta14C_from_AbsoluteFractionModern(AbsoluteFractionModern)
```

Arguments

`AbsoluteFractionModern` : of class matrix, An object of class matrix containing the values in Absolute Fraction Modern format

Delta14C_from_AbsoluteFractionModern,numeric-method *Delta14C_from_AbsoluteFractionModern,numeric-method Converts to Delta14C format*

Description

This method produces Delta14C values from Absolute Fraction Modern Have a look at the methods for details.

Usage

```
## S4 method for signature 'numeric'
Delta14C_from_AbsoluteFractionModern(AbsoluteFractionModern)
```

Arguments

AbsoluteFractionModern	: of class numeric, A numeric object containing the values in Absolute Fraction Modern format

deSolve.lsoda.wrapper *deSolve.lsoda.wrapper*

Description

The function serves as a wrapper for lsoda using a much simpler interface which allows the use of matrices in the definition of the derivative. To use lsoda we have to convert our vectors to lists, define tolerances and so on. This function does this for us , so we don't need to bother about it.

Usage

```
deSolve.lsoda.wrapper(t,
ydot,
startValues)
```

Arguments

t	A row vector containing the points in time where the solution is sought.
ydot	The function of y and t that computes the derivative for a given point in time and a column vector y.
startValues	A column vector with the starting values.

Value

A matrix. Every column represents a pool and every row a point in time

eCO2 *Soil CO2 efflux from an incubation experiment*

Description

A dataset with soil CO2 efflux measurements from a laboratory incubation at controlled temperature and moisture conditions.

Usage

```
data(eCO2)
```

Format

A data frame with the following 3 variables.

`Days` A numeric vector with the day of measurement after the experiment started.

`eCO2mean` A numeric vector with the release flux of CO2. Units in ug C g-1 soil day-1.

`eCO2sd` A numeric vector with the standard deviation of the release flux of CO2-C. Units in ug C g-1 soil day-1.

Details

A laboratory incubation experiment was performed in March 2014 for a period of 35 days under controlled conditions of temperature (15 degrees Celsius), moisture (30 percent soil water content), and oxygen levels (20 percent). Soil CO2 measurements were taken using an automated system for gas sampling connected to an infrared gas analyzer. The soil was sampled at a boreal forest site (Caribou Poker Research Watershed, Alaska, USA). This dataset presents the mean and standard deviation of 4 replicates.

Examples

```
head(eCO2)

plot(eCO2[,1:2],type="o",ylim=c(0,50),ylab="CO2 efflux (ug C g-1 soil day-1)")
arrows(eCO2[,1],eCO2[,2]-eCO2[,3],eCO2[,1],eCO2[,2]+eCO2[,3], angle=90,length=0.3,code=3)
```

euler *euler*

Description

This function can solve arbitrary first order ode systems with the euler forward method and an adaptive time-step size control given a tolerance for the deviation of a coarse and fine estimate of the change in y for the next time step. It is an alternative to `deSolve.lsoda.wrapper` and has the same interface. It is much slower than ode and should probably be considered less capable in solving stiff ode systems. However it has one main advantage, which consists in its simplicity. It is quite easy to see what is going on inside it. Whenever you don't trust your implementation of another (more efficient but probably also more complex) ode solver, just compare the result to what this method computes.

Usage

```
euler(times,
ydot,
startValues)
```

Arguments

times	A row vector containing the points in time where the solution is sought.
ydot	The function of y and t that computes the derivative for a given point in time and a column vector y.
startValues	A column vector with the initial values.

---

example.2DBoundInFluxFromFunction
example.2DBoundInFluxFromFunction

Description

Create a 2-dimensionsonal example of a BoundInFlux object

Usage

```
example.2DBoundInFluxFromFunction()
```

Value

The returned object represents a time dependent Influx into a two pool model.

---

example.2DBoundLinDecompOpFromFunction
example.2DBoundLinDecompOpFromFunction

Description

An example used in tests and other examples.

Usage

```
example.2DBoundLinDecompOpFromFunction()
```

---

example.2DConstFc.Args
example.2DConstFc.Args

Description

Create a 2-dimensionsonal examples of a Influx objects from different arguments

Usage

```
example.2DConstFc.Args()
```

`example.2DConstInFluxFromVector`
2D example for constant Influx

Description

An example used in tests and other examples.

Usage

```
example.2DConstInFluxFromVector()
```

Value

The returned object represents a time invariant constant influx into a two pool model.

`example.2DGeneralDecompOpArgs`
example.2DGeneralDecompOpArgs

Description

We present all possibilities to define a 2D `DecompOp-class`

Usage

```
example.2DGeneralDecompOpArgs()
```

`example.2DInFlux.Args`
example.2DInFlux.Args

Description

Create a 2-dimensionsonal examples of a Influx objects from different arguments

Usage

```
example.2DInFlux.Args()
```

`example.2DUnBoundLinDecompOpFromFunction`
example.2DUnBoundLinDecompOpFromFunction

Description

An example used in tests and other examples.

Usage

```
example.2DUnBoundLinDecompOpFromFunction()
```

`example.ConstlinDecompOpFromMatrix`
example.ConstlinDecompOpFromMatrix

Description

An example used in tests and other examples.

Usage

```
example.ConstlinDecompOpFromMatrix()
```

`example.nestedTime2DMatrixList`
create an example nested list that can be

Description

An example used in tests and other examples.

Usage

```
example.nestedTime2DMatrixList()
```

`example.Time2DArrayList`
create an example TimeMap from 2D array

Description

An example used in tests and other examples.

Usage

```
example.Time2DArrayList()
```

example.Time3DArrayList *create an example TimeFrame from 3D array*

Description

An example used in tests and other examples.

Usage

```
example.Time3DArrayList()
```

example.TimeMapFromArray *create an example TimeFrame from 3D array*

Description

The function creates an example TimeMap that is used in other examples and tests.

Usage

```
example.TimeMapFromArray()
```

Fc-class *Fc S4 class*

Description

The ^{14}C fraction is a necessary ingredient of any `Model_14` object. In the most general case it is a real valued function of time, accompanied by a string describing the unit or format (i.e. "Delta14C" or "afn" for Absolute Fraction Modern) . In the most simple case it is constant real number plus format.

Methods

No exported methods directly defined for class Fc:

Subclasses

`BoundFc`
`ConstFc`

Constructors

The class is abstract (`contains "VIRTUAL"`). It can therefore not be instanciated directly. Look at non virtual subclasses and their constructors!

`FcAtm.from.Dataframe` *FcAtm.from.Dataframe*

Description

This function is deprecated constructor of the deprecatied class FcAtm

Usage

```
FcAtm.from.Dataframe(dframe,
lag=0,
interpolation=splinefun,
format)
```

Arguments

`dframe`	A data frame containing exactly two columns: the first one is interpreted as time the secon one is interpreted as atmospheric C14 fraction in the format mentioned
`lag`	a scalar describing the time lag. Positive Values shift the argument of the interpolation function forward in time. (retard its effect)
`interpolation`	A function that returns a function the default is splinefun. Other possible values are the linear interpolation approxfun or any self made function with the same interface.
`format`	a string that specifies the format used to represent the atmospheric fracton. Possible values are "Delta14C" which is the default or "afn" the Absolute Fraction Normal representation

Value

An object of the new class BoundFc that replaces FcAtm

`fT.Arrhenius` *Effects of temperature on decomposition rates according the Arrhenius equation*

Description

Calculates the effects of temperature on decomposition rates according to the Arrhenius equation.

Usage

```
fT.Arrhenius(Temp,
A=1000,
Ea=75000,
Re=8.3144621)
```

Arguments

Temp	A scalar or vector containing values of temperature (in degrees Kelvin) for which the effects on decomposition rates are calculated.
A	A scalar defining the pre-exponential factor.
Ea	A scalar defining the activation energy in units of J mol^-1.
Re	A scalar defining the universal gas contant in units of J K^-1 mol^-1.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

---

fT.Century1 *Effects of temperature on decomposition rates according the the CENTURY model*

Description

Calculates the effects of temperature on decomposition rates according to the CENTURY model.

Usage

```
fT.Century1(Temp,
Tmax=45,
Topt=35)
```

Arguments

Temp	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.
Tmax	A scalar defining the maximum temperature in degrees C.
Topt	A scalar defining the optimum temperature for the decomposition process in degrees C.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Burke, I. C., J. P. Kaye, S. P. Bird, S. A. Hall, R. L. McCulley, and G. L. Sommerville. 2003. Evaluating and testing models of terrestrial biogeochemistry: the role of temperature in controlling decomposition. Pages 235-253 in C. D. Canham, J. J. Cole, and W. K. Lauenroth, editors. Models in ecosystem science. Princeton University Press, Princeton.

fT.Century2 — *Effects of temperature on decomposition rates according the the CENTURY model*

Description

Calculates the effects of temperature on decomposition rates according to the CENTURY model.

Usage

```
fT.Century2(Temp,
Tmax=45,
Topt=35)
```

Arguments

Temp	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.
Tmax	A scalar defining the maximum temperature in degrees C.
Topt	A scalar defining the optimum temperature for the decomposition process in degrees C.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Adair, E. C., W. J. Parton, S. J. D. Grosso, W. L. Silver, M. E. Harmon, S. A. Hall, I. C. Burke, and S. C. Hart. 2008. Simple three-pool model accurately describes patterns of long-term litter decomposition in diverse climates. Global Change Biology 14:2636-2660.

fT.Daycent1 — *Effects of temperature on decomposition rates according to the DAYCENT model*

Description

Calculates the effects of temperature on decomposition rates according to the DAYCENT model.

Usage

```
fT.Daycent1(Temp)
```

Arguments

Temp	A scalar or vector containing values of soil temperature for which the effects on decomposition rates are calculated

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Kelly, R. H., W. J. Parton, M. D. Hartman, L. K. Stretch, D. S. Ojima, and D. S. Schimel (2000), Intra-annual and interannual variability of ecosystem processes in shortgrass steppe, J. Geophys. Res., 105.

---

fT.Daycent2 — *Effects of temperature on decomposition rates according to the DAYCENT model*

Description

Calculates the effects of temperature on decomposition rates according to the Daycent/Century models.

Usage

```
fT.Daycent2(Temp)
```

Arguments

Temp	A scalar or vector containing values of soil temperature for which the effects on decomposition rates are calculated.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Del Grosso, S. J., W. J. Parton, A. R. Mosier, E. A. Holland, E. Pendall, D. S. Schimel, and D. S. Ojima (2005), Modeling soil CO2 emissions from ecosystems, Biogeochemistry, 73(1), 71-91.

---

fT.Demeter — *Effects of temperature on decomposition rates according to the DEMETER model*

Description

Calculates the effects of temperature on decomposition rates according to the DEMETER model.

Usage

```
fT.Demeter(Temp,
Q10=2)
```

Arguments

`Temp`	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated
`Q10`	A scalar. Temperature coefficient Q10

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Foley, J. A. (1995), An equilibrium model of the terrestrial carbon budget, Tellus B, 47(3), 310-319.

`fT.KB` *Effects of temperature on decomposition rates according to a model proposed by M. Kirschbaum (1995)*

Description

Calculates the effects of temperature on decomposition rates according to a model proposed by Kirschbaum (1995).

Usage

```
fT.KB(Temp)
```

Arguments

`Temp`	a scalar or vector containing values of soil temperature for which the effects on decomposition rates are calculated

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Kirschbaum, M. U. F. (1995), The temperature dependence of soil organic matter decomposition, and the effect of global warming on soil organic C storage, Soil Biology and Biochemistry, 27(6), 753-760.

fT.LandT — *Effects of temperature on decomposition rates according to a function proposed by Lloyd and Taylor (1994)*

Description

Calculates the effects of temperature on decomposition rates according to a function proposed by Lloyd and Taylor (1994).

Usage

```
fT.LandT(Temp)
```

Arguments

Temp	A scalar or vector containing values of soil temperature for which the effects on decomposition rates are calculated

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Lloyd, J., and J. A. Taylor (1994), On the Temperature Dependence of Soil Respiration, Functional Ecology, 8(3), 315-323.

fT.linear — *Effects of temperature on decomposition rates according to a linear model*

Description

Calculates the effects of temperature on decomposition rates according to a linear model.

Usage

```
fT.linear(Temp,
a=0.198306,
b=0.036337)
```

Arguments

Temp	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.
a	A scalar defining the intercept of the linear function.
b	A scalar defining the slope of the linear function.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Adair, E. C., W. J. Parton, S. J. D. Grosso, W. L. Silver, M. E. Harmon, S. A. Hall, I. C. Burke, and S. C. Hart. 2008. Simple three-pool model accurately describes patterns of long-term litter decomposition in diverse climates. Global Change Biology 14:2636-2660.

`fT.Q10` — *Effects of temperature on decomposition rates according to a Q10 function*

Description

Calculates the effects of temperature on decomposition rates according to the modified Van't Hoff function (Q10 function).

Usage

```
fT.Q10(Temp,
k_ref=1,
T_ref=10,
Q10=2)
```

Arguments

`Temp`	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.
`k_ref`	A scalar representing the value of the decomposition rate at a reference temperature vaule.
`T_ref`	A scalar representing the reference temperature.
`Q10`	A scalar. Temperature coefficient Q10.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

fT.RothC — *Effects of temperature on decomposition rates according to the functions included in the RothC model*

Description

Calculates the effects of temperature on decomposition rates according to the functions included in the RothC model.

Usage

```
fT.RothC(Temp)
```

Arguments

Temp	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

Note

This function returns NA for Temp <= -18.3

References

Jenkinson, D. S., S. P. S. Andrew, J. M. Lynch, M. J. Goss, and P. B. Tinker (1990), The Turnover of Organic Carbon and Nitrogen in Soil, Philosophical Transactions: Biological Sciences, 329(1255), 361-368.

fT.Standcarb — *Effects of temperature on decomposition rates according to the StandCarb model*

Description

Calculates the effects of temperature on decomposition rates according to the StandCarb model.

Usage

```
fT.Standcarb(Temp,
Topt=45,
Tlag=4,
Tshape=15,
Q10=2)
```

Arguments

Temp	A scalar or vector containing values of temperature for which the effects on decomposition rates are calculated.
Topt	A scalar representing the optimum temperature for decomposition.
Tlag	A scalar that determines the lag of the response curve.
Tshape	A scalar that determines the shape of the response curve.
Q10	A scalar. Temperature coefficient Q10.

Value

A scalar or a vector containing the effects of temperature on decomposition rates (unitless).

References

Harmon, M. E., and J. B. Domingo (2001), A users guide to STANDCARB version 2.0: A model to simulate carbon stores in forest stands. Oregon State University, Corvallis.

---

fW.Candy — *Effects of moisture on decomposition rates according to the Candy model*

Description

Calculates the effects of water content and pore volume on decomposition rates.

Usage

```
fW.Candy(theta,
PV)
```

Arguments

theta	A scalar or vector containing values of volumetric soil water content.
PV	A scalar or vector containing values of pore volume.

References

J. Bauer, M. Herbst, J.A. Huisman, L. Weiherm\"uller, H. Vereecken. 2008. Sensitivity of simulated soil heterotrophic respiration to temperature and moisture reduction functions. Geoderma, Volume 145, Issues 1-2, 15 May 2008, Pages 17-27.

fW.Century — *Effects of moisture on decomposition rates according to the CENTURY model*

Description

Calculates the effects of precipitation and potential evapotranspiration on decomposition rates.

Usage

```
fW.Century(PPT,
PET)
```

Arguments

PPT	A scalar or vector containing values of monthly precipitation.
PET	A scalar or vector containing values of potential evapotranspiration.

Value

A scalar or a vector containing the effects of precipitation and potential evapotranspiration on decomposition rates (unitless).

References

Adair, E. C., W. J. Parton, S. J. D. Grosso, W. L. Silver, M. E. Harmon, S. A. Hall, I. C. Burke, and S. C. Hart (2008), Simple three-pool model accurately describes patterns of long-term litter decomposition in diverse climates, Global Change Biology, 14(11), 2636-2660. \ Parton, W. J., J. A. Morgan, R. H. Kelly, and D. S. Ojima (2001), Modeling soil C responses to environmental change in grassland systems, in The potential of U.S. grazing lands to sequester carbon and mitigate the greenhouse effect, edited by R. F. Follett, J. M. Kimble and R. Lal, pp. 371-398, Lewis Publishers, Boca Raton.

fW.Daycent1 — *Effects of moisture on decomposition rates according to the DAYCENT model*

Description

Calculates the effects of Soil Water Content on decomposition rates according to the Daycent Model.

Usage

```
fW.Daycent1(swc,
a=0.6,
b=1.27,
c=0.0012,
d=2.84,
partd=2.65,
bulkd=1,
width=1)
```

Arguments

swc	A scalar or vector with soil water content of a soil layer (cm).
a	Empirical coefficient. For fine textured soils a = 0.6. For coarse textured soils a = 0.55.
b	Empirical coefficient. For fine textured soils b = 1.27. For coarse textured soils b = 1.70.
c	Empirical coefficient. For fine textured soils c = 0.0012. For coarse textured soils c = -0.007.
d	Empirical coefficient. For fine textured soils d = 2.84. For coarse textured soils d = 3.22.
partd	Particle density of soil layer.
bulkd	Bulk density of soil layer (g/cm^3).
width	Thickness of a soil layer (cm).

Value

A data frame with values of water filled pore space (wfps) and effects of soil water content on decomposition rates. Both vectors are unitless.

References

Kelly, R. H., W. J. Parton, M. D. Hartman, L. K. Stretch, D. S. Ojima, and D. S. Schimel (2000), Intra-annual and interannual variability of ecosystem processes in shortgrass steppe, J. Geophys. Res., 105.

---

fW.Daycent2 — *Effects of moisture on decomposition rates according to the DAYCENT model*

Description

Calculates the effects of volumetric water content on decomposition rates according to the Daycent/Century models.

Usage

```
fW.Daycent2(W,
WP=0,
FC=100)
```

Arguments

W	A scalar or vector of volumetric water content in percentage.
WP	A scalar representing the wilting point in percentage.
FC	A scalar representing the field capacity in percentage.

Value

A data frame with values of relative water content (RWC) and the effects of RWC on decomposition rates (fRWC).

References

Del Grosso, S. J., W. J. Parton, A. R. Mosier, E. A. Holland, E. Pendall, D. S. Schimel, and D. S. Ojima (2005), Modeling soil CO2 emissions from ecosystems, Biogeochemistry, 73(1), 71-91.

`fW.Demeter` *Effects of moisture on decomposition rates according to the DEMETER model*

Description

Calculates the effects of soil moisture on decomposition rates according to the DEMETER model.

Usage

```
fW.Demeter(M,
Msat=100)
```

Arguments

`M`	A scalar or vector containing values of soil moisture for which the effects on decomposition rates are calculated.
`Msat`	A scalar representing saturated soil moisture.

Value

A scalar or a vector containing the effects of moisture on decomposition rates (unitless).

References

Foley, J. A. (1995), An equilibrium model of the terrestrial carbon budget, Tellus B, 47(3), 310-319.

`fW.Gompertz` *Effects of moisture on decomposition rates according to the Gompertz function*

Description

Calculates the effects of water content on decomposition rates.

Usage

```
fW.Gompertz(theta,
a=0.824,
b=0.308)
```

Arguments

`theta`	A scalar or vector containing values of volumetric soil water content.
`a`	Empirical parameter
`b`	Empirical parameter

References

I. Janssens, S. Dore, D. Epron, H. Lankreijer, N. Buchmann, B. Longdoz, J. Brossaud, L. Montagnani. 2003. Climatic Influences on Seasonal and Spatial Differences in Soil CO2 Efflux. In Valentini, R. (Ed.) Fluxes of Carbon, Water and Energy of European Forests. pp 235-253. Springer.

`fW.Moyano` *Effects of moisture on decomposition rates according to the function proposed by Moyano et al. (2013)*

Description

Calculates the effects of water content on decomposition rates.

Usage

```
fW.Moyano(theta,
a=3.11,
b=2.42)
```

Arguments

`theta`	A scalar or vector containing values of volumetric soil water content.
`a`	Empirical parameter
`b`	Empirical parameter

References

F. E. Moyano, S. Manzoni, C. Chenu. 2013 Responses of soil heterotrophic respiration to moisture availability: An exploration of processes and models. Soil Biology and Biochemistry, Volume 59, April 2013, Pages 72-85

`fW.RothC` *Effects of moisture on decomposition rates according to the RothC model*

Description

Calculates the effects of moisture (precipitation and pan evaporation) on decomposition rates according to the RothC model.

Usage

```
fW.RothC(P,
E,
S.Thick=23,
pClay=23.4,
pE=0.75,
bare=FALSE)
```

Arguments

P	A vector with monthly precipitation (mm).
E	A vector with same length with open pan evaporation or evapotranspiration (mm).
S.Thick	Soil thickness in cm. Default for Rothamsted is 23 cm.
pClay	Percent clay.
pE	Evaporation coefficient. If open pan evaporation is used pE=0.75. If Potential evaporation is used, pE=1.0.
bare	Logical. Under bare soil conditions, bare=TRUE. Dafault is set under vegetated soil.

Value

A data.frame with accumulated top soil moisture deficit (Acc.TSMD) and the rate modifying factor b.

References

Coleman, K., and D. S. Jenkinson (1999), RothC-26.3 A model for the turnover of carbon in soil: model description and windows user guide (modified 2008), 47 pp, IACR Rothamsted, Harpenden.

---

fW.Skopp *Effects of moisture on decomposition rates according to the function proposed by Skopp et al. 1990*

Description

Calculates the effects of relative soil water content on decomposition rates.

Usage

```
fW.Skopp(rwc,
alpha=2,
beta=2,
f=1.3,
g=0.8)
```

Arguments

rwc	relative water content
alpha	Empirical parameter
beta	Empirical parameter
f	Empirical parameter
g	Empirical parameter

References

J. Skopp, M. D. Jawson, and J. W. Doran. 1990. Steady-state aerobic microbial activity as a function of soil water content. Soil Sci. Soc. Am. J., 54(6):1619-1625

`fW.Standcarb` *Effects of moisture on decomposition rates according to the StandCarb model*

Description

Calculates the effects of moisture on decomposition rates according to the StandCarb model.

Usage

```
fW.Standcarb(Moist,
MatricShape=5,
MatricLag=0,
MoistMin=30,
MoistMax=350,
DiffuseShape=15,
DiffuseLag=4)
```

Arguments

`Moist`	A scalar or vector containing values of moisture content of a litter or soil pool (%).
`MatricShape`	A scalar that determines when matric limit is reduced to the point that decay can begin to occur.
`MatricLag`	A scalar used to offset the curve to the left or right.
`MoistMin`	A scalar determining the minimum moisture content.
`MoistMax`	A scalar determining the maximum moisture content without diffusion limitations.
`DiffuseShape`	A scalar that determines the range of moisture contents where diffusion is not limiting.
`DiffuseLag`	A scalar used to shift the point when moisture begins to limit diffusion.

Value

A data frame with limitation due to water potential (MatricLimit), limitation due to oxygen diffusion (DiffuseLimit), and the overall limitation of moisture on decomposition rates (MoistDecayIndex).

References

Harmon, M. E., and J. B. Domingo (2001), A users guide to STANDCARB version 2.0: A model to simulate carbon stores in forest stands. Oregon State University, Corvallis.

GaudinskiModel14 *Implementation of a the six-pool C14 model proposed by Gaudinski et al. 2000*

Description

This function creates a model as described in Gaudinski et al. 2000. It is a wrapper for the more general functions GeneralModel_14 that can handle an arbitrary number of pools.

Usage

```
GaudinskiModel14(t,
ks=c(kr = 1/1.5,
 koi = 1/1.5, koeal = 1/4, koeah = 1/80, kA1 = 1/3, kA2 = 1/75, kM = 1/110),
C0=c(FR0 = 390, C10 = 220, C20 = 390, C30 = 1370, C40 = 90, C50 = 1800, C60 = 56
F0_Delta14C=rep(0, 7),
LI=150,
RI=255,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.
ks	A vector of length 7 containing the decomposition rates for the 6 soil pools plus the fine-root pool.
C0	A vector of length 7 containing the initial amount of carbon for the 6 pools plus the fine-root pool.
F0_Delta14C	A vector of length 7 containing the initial amount of the radiocarbon fraction for the 7 pools as Delta14C values in per mil.
LI	A scalar or a data.frame object specifying the amount of litter inputs by time.
RI	A scalar or a data.frame object specifying the amount of root inputs by time.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive integer representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. An alternative to the default is euler or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Gaudinski JB, Trumbore SE, Davidson EA, Zheng S (2000) Soil carbon cycling in a temperate forest: radiocarbon-based estimates of residence times, sequestration rates and partitioning fluxes. Biogeochemistry 51: 33-69

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
years=seq(1901,2010,by=0.5)

Ex=GaudinskiModel14(
t=years,
ks=c(kr=1/3, koi=1/1.5, koeal=1/4, koeah=1/80, kA1=1/3, kA2=1/75, kM=1/110),
inputFc=C14Atm_NH
)
R14m=getF14R(Ex)
C14m=getF14C(Ex)

plot(
C14Atm_NH,
type="l",
xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),
xlim=c(1940,2010)
)
lines(years,C14m,col=4)
points(HarvardForest14CO2[1:11,1],HarvardForest14CO2[1:11,2],pch=19,cex=0.5)
points(HarvardForest14CO2[12:173,1],HarvardForest14CO2[12:173,2],pch=19,col=2,cex=0.5)
points(HarvardForest14CO2[158,1],HarvardForest14CO2[158,2],pch=19,cex=0.5)
lines(years,R14m,col=2)
legend(
"topright",
c("Delta 14C Atmosphere",
"Delta 14C SOM",
"Delta 14C Respired"
),
lty=c(1,1,1),
col=c(1,4,2),
bty="n"
)
## We now show how to bypass soilR s parameter sanity check if nacessary
## (e.g in for parameter estimation ) in functions
## wchich might call it with unreasonable parameters
years=seq(1800,2010,by=0.5)
Ex=GaudinskiModel14(
t=years,
ks=c(kr=1/3,koi=1/1.5,koeal=1/4,koeah=1/80,kA1=1/3,kA2=1/75,kM=1/110),
inputFc=C14Atm_NH,
pass=TRUE
```

```
)
```

`GeneralDecompOp` *GeneralDecompOp S4 generic*

Description

no Description

Usage

```
GeneralDecompOp(object)
```

Arguments

`object`	see the method arguments for details

Methods

`GeneralDecompOp,DecompOp-method`
`GeneralDecompOp,TimeMap-method`
`GeneralDecompOp,function-method`
`GeneralDecompOp,list-method`
`GeneralDecompOp,matrix-method`

`GeneralDecompOp,DecompOp-method`
GeneralDecompOp,DecompOp-method pass through factory

Description

This method handles the case that no actual construction is necessary since the argument is already of a subclass of DecompOp. See the `subclasses` section of `DecompOp-class`

Usage

```
## S4 method for signature 'DecompOp'
GeneralDecompOp(object)
```

Arguments

`object`	: of class DecompOp

`GeneralDecompOp,function-method`

GeneralDecompOp,function-method creates a UnBoundLinDecompOp from a matrix valued function

---

Description

The resulting operator is created by a call to the constructor of class UnBoundLinDecompOp

Usage

```
## S4 method for signature 'function'
GeneralDecompOp(object)
```

Arguments

`object`	: of class function

`GeneralDecompOp,list-method`

GeneralDecompOp,list-method creates a BoundLinDecompOp from a nested list of a times vector and a list of matrices (a matrix for each time step)

Description

The resulting operator is creted by a call to the constructor of class BoundLinDecompOp

Usage

```
## S4 method for signature 'list'
GeneralDecompOp(object)
```

Arguments

`object`	: of class list

`GeneralDecompOp,matrix-method`
GeneralDecompOp,matrix-method creates a ConstanDecompOp from a matrix

---

Description

The resulting operator is creted by a call to the constructor of class ConstLinDecompOp

Usage

```
## S4 method for signature 'matrix'
GeneralDecompOp(object)
```

Arguments

`object` : of class matrix

`GeneralDecompOp,TimeMap-method`
GeneralDecompOp,TimeMap-method creates a BoundLinDecompOp from a TimeMap object

---

Description

The resulting operator is creted by a call to the constructor of class BoundLinDecompOp

Usage

```
## S4 method for signature 'TimeMap'
GeneralDecompOp(object)
```

Arguments

`object` : of class TimeMap

`GeneralInFlux` *GeneralInFlux S4 generic*

Description

no Description

Usage

```
GeneralInFlux(object)
```

Arguments

`object`	see the method arguments for details

Methods

`GeneralInFlux,InFlux-method`
`GeneralInFlux,TimeMap-method`
`GeneralInFlux,function-method`
`GeneralInFlux,list-method`
`GeneralInFlux,numeric-method`

`GeneralInFlux,function-method`
GeneralInFlux,function-method creates a UnBoundInFlux from a vector valued function

Description

The resulting operator is created by a call to the constructor of class UnBoundInFlux. You should only use this if the domain of your function is the complete time axis (-Inf,+Inf). If your function has a finite domain create an object of class `BoundInFlux-class` ### by calling `BoundInFlux`. This will activeate checks on that avoid unintended extrapolation.

Usage

```
## S4 method for signature 'function'
GeneralInFlux(object)
```

Arguments

`object`	: of class function

`GeneralInFlux,InFlux-method`
GeneralInFlux,InFlux-method pass through conversion

Description

This method handles the case that no actual conversion is necessary since the argument is already of a subclass of InFlux-class

Usage

```
## S4 method for signature 'InFlux'
GeneralInFlux(object)
```

Arguments

`object`	: of class InFlux

`GeneralInFlux,list-method`
GeneralInFlux,list-method creates a BoundInFlux from a nested list of a times vector and a list of vectors (a vector for each time step)

Description

The resulting object is created by a call to the constructor of class BoundInFlux

Usage

```
## S4 method for signature 'list'
GeneralInFlux(object)
```

Arguments

`object`	: of class list

GeneralInFlux,numeric-method
GeneralInFlux,numeric-method conversion of a vector to an object of class ConstInFlux

Description

This method enables the model creating functions to handle constant input streams

Usage

```
## S4 method for signature 'numeric'
GeneralInFlux(object)
```

Arguments

object	: of class numeric

GeneralInFlux,TimeMap-method
GeneralInFlux,TimeMap-method create a BoundInFlux from a TimeMap object

Description

The method is used to ensure compatibility TimeMap class. The resulting BoundInFlux is created by a call to the constructor BoundInFlux(object) of that class.

Usage

```
## S4 method for signature 'TimeMap'
GeneralInFlux(object)
```

Arguments

object	: of class TimeMap

GeneralModel *additional function to create Models*

Description

In previous SoilR Version GeneralModel was the function to create linear models, a task now fulfilled by the function Model. To ensure backward compatibility this function remains as a wrapper. In future versions it might take on the role of an abstract factory that produces several classes of models (i.e linear or non-linear) depending on different combinations of arguments. It creates a Model object from any combination of arguments that can be converted into the required set of building blocks for a model for n arbitrarily connected pools.

Usage

```
GeneralModel(t,
A,
ivList,
inputFluxes,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
A	Anything that can be converted by GeneralDecompOp to any of the available DecompositionOperator classes
ivList	A vector containing the initial amount of carbon for the n pools. The length of this vector is equal to the number of pools and thus equal to the length of k. This is checked by an internal function.
inputFluxes	something that can be converted to any of the available InFlux classes
solverfunc	The function used by to actually solve the ODE system. This can be deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	Forces the constructor to create the model even if it is invalid

Value

A model object that can be further queried.

See Also

TwopParallelModel, TwopSeriesModel, TwopFeedbackModel

GeneralModel_14 *create objects of class Model_14*

Description

At the moment this is just a wrapper for the actual constructor Model_14 with additional support for some now deprecated parameters for backward compatibility. This role may change in the future to an abstract factory where the actual class of the created model will be determined by the supplied parameters.

Usage

```
GeneralModel_14(t,
A,
ivList,
initialValF,
inputFluxes,
Fc=NULL,
inputFc=Fc,
di=-0.0001209681,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
A	something that can be converted by GeneralDecompOp to any of the available subclasses of DecompOp.
ivList	A vector containing the initial amount of carbon for the n pools. The length of this vector is equal to the number of pools and thus equal to the length of k. This is checked by an internal function.
initialValF	An object equal or equivalent to class ConstFc containing a vector with the initial values of the radiocarbon fraction for each pool and a format string describing in which format the values are given.
inputFluxes	something that can be converted by GeneralInFlux to any of the available subclasses of InFlux.
Fc	deprecated keyword argument, please use inputFc instead
inputFc	An object describing the fraction of C_14 in per mille (different formats are possible)
di	the rate at which C_14 decays radioactivly. If you don't provide a value here we assume the following value: k=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year. Thus beside time itself it also affects decay rates the inputrates and the output
solverfunc	The function used by to actually solve the ODE system. This can be deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	Forces the constructor to create the model even if it is invalid

Value

A model object that can be further queried.

See Also

TwopParallelModel, TwopSeriesModel, TwopFeedbackModel

GeneralNlModel *Use this function to create objects of class NlModel.*

Description

The function creates a numerical model for n arbitrarily connected pools. It is one of the constructors of class NlModel. It is used by some more specialized wrapper functions, but can also be used directly.

Usage

```
GeneralNlModel(t,
TO,
ivList,
inputFluxes,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
TO	A object describing the model decay rates for the n pools, connection and feedback coefficients. The number of pools n must be consistent with the number of initial values and input fluxes.
ivList	A numeric vector containing the initial amount of carbon for the n pools. The length of this vector is equal to the number of pools.
inputFluxes	A TimeMap object consisting of a vector valued function describing the inputs to the pools as funtions of time TimeMap.new.
solverfunc	The function used by to actually solve the ODE system.
pass	Forces the constructor to create the model even if it is invalid. If set to TRUE, does not enforce the requirements for a biologically meaningful model, e.g. does not check if negative values of respiration are calculated.

Value

Tr=getTransferMatrix(Anl) #this is a function of C and t

##
# build the two models (linear and nonlinear) mod=GeneralModel(t, A,iv, inputrates, deSolve.lsoda.wrapper)
modnl=GeneralNlModel(t, Anl, iv, inputrates, deSolve.lsoda.wrapper)

Ynonlin=getC(modnl) lt1=2 lt2=4 plot(t,Ynonlin[,1],type="l",lty=lt1,col=1, ylab="Concentrations",xlab="Time",ylim=c
lines(t,Ynonlin[,2],type="l",lty=lt2,col=2) legend("topleft",c("Pool 1", "Pool 2"),lty=c(lt1,lt2),col=c(1,2))

See Also

GeneralModel.

Examples

```
t_start=0
t_end=20
tn=100
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
k1=1/2
k2=1/3
Km=0.5
nr=2

alpha=list()
alpha[["1_to_2"]]=function(C,t){
1/5
}
alpha[["2_to_1"]]=function(C,t){
1/6
}

f=function(C,t){
# The only thing to take care of is that we release a vector of the same
# size as C
S=C[[1]]
M=C[[2]]
O=matrix(byrow=TRUE,nrow=2,c(k1*M*(S/(Km+S)),
k2*M))
return(O)
}
Anl=new("TransportDecompositionOperator",t_start,Inf,nr,alpha,f)


c01=3
c02=2
iv=c(c01,c02)
inputrates=new("TimeMap",t_start,t_end,function(t){return(matrix(
nrow=nr,
ncol=1,
c( 2,  2)
))})
################################################################################
# we check if we can reproduce the linear decomposition operator from the
# nonlinear one
```

---

getAccumulatedRelease

getAccumulatedRelease S4 generic

Description

no Description

Usage

```
getAccumulatedRelease(object)
```

Arguments

object	A Model object (e.g. of class Model or Model14) Have a look at the methods for details.

Value

A n x m matrix of cummulative release fluxes with m columns representing the number of pools, and n rows representing the times as specified by the argument t in GeneralModel or other model creating functions.

Methods

getAccumulatedRelease,Model-method

---

getAccumulatedRelease,Model-method *getAccumulatedRelease,Model-method time integrals of release fluxes per pool*

Description

The method integrates the release flux of every pool for all times in the interval specified by the model definition.

Usage

```
## S4 method for signature 'Model'
getAccumulatedRelease(object)
```

Arguments

object	: of class Model

---

getC *getC S4 generic*

Description

no Description

Usage

```
getC(object,
as.closures=F)
```

Arguments

object	some model object, the actual class depends on the method used.
as.closures	if set to TRUE instead of a matrix a list of functions will be returned.

Value

A matrix with m columns representing the number of pools, and n rows representing the times as specified by the argument t in GeneralModel or another model creating function.

Methods

getC,Model-method
getC,NlModel-method

---

getC,Model-method *getC,Model-method*

Description

This function computes the value for C for each time and pool.

Usage

```
## S4 method for signature 'Model'
getC(object)
```

Arguments

object	: of class Model

Details

This function takes a Model object, which represents a system of ODEs of the form

$$\frac{d\mathbf{C}(t)}{dt} = \mathbf{I}(t) + \mathbf{A}(t)\mathbf{C}(t)$$

and solves the system for $\mathbf{C}(t)$. The numerical solver used can be specified in the constructor of the Model class e.g. Model, GeneralModel.

Value

A matrix with m columns representing the number of pools, and n rows representing the times as specified by the argument t in Model,GeneralModel or another model creating function.

See Also

See examples in GeneralModel, GeneralModel_14, TwopParallelModel, TwopSeriesModel, TwopFeedbackModel, etc.

`getC,NlModel-method` *getC,NlModel-method compute the solution for the content of the pools*

Description

This function computes the value for C for each time and pool.

Usage

```
## S4 method for signature 'NlModel'
getC(object,
as.closures=FALSE)
```

Arguments

`object`	: of class NlModel, the model
`as.closures`	: a flag that in TRUE will return an approximating function instead of the values

---

`getC14` *This function*

Description

no Description

Usage

```
getC14(object)
```

Arguments

`object`	see the method arguments for details

Methods

`getC14,Model_14-method`

getC14,Model_14-method *getC14,Model_14-method*

Description

This function computes the value for ^{14}C (mass or concentration) as function of time

Usage

```
## S4 method for signature 'Model_14'
getC14(object)
```

Arguments

object	: of class Model_14

getDecompOp *getDecompOp S4 generic*

Description

no Description

Usage

```
getDecompOp(object)
```

Arguments

object	see the method arguments for details

Methods

getDecompOp,Model-method
getDecompOp,NlModel-method

Examples

```
# suppose you have somehow created a model
# e.g. by one of the predefined functions
years=seq(1901,2010,by=0.5)

Ex=GaudinskiModel14(
t=years,
ks=c(kr=1/3, koi=1/1.5, koeal=1/4, koeah=1/80, kA1=1/3, kA2=1/75, kM=1/110),
inputFc=C14Atm_NH
)
op <- getDecompOp(Ex)
```

```
# Now you can get properties of the decompostion operator e.g.

func <- getFunctionDefinition(op)

#func is a x valued funtion of time
# In this example it is even constant, which is boring.
# you can evaluate it for different times
func(0)
func(5)
```

`getDecompOp,Model-method`
getDecompOp,Model-method

Description

Extracts the Operator from a model object

Usage

```
## S4 method for signature 'Model'
getDecompOp(object)
```

Arguments

`object`	: of class Model

`getDecompOp,NlModel-method`
getDecompOp,NlModel-method

Description

This method returns the Decomposition Operator of the Model

Usage

```
## S4 method for signature 'NlModel'
getDecompOp(object)
```

Arguments

`object`	: of class NlModel

getF14 *Computes the $\frac{\hat{}14C}{C}$ ratio*

Description

no Description

Usage

```
getF14(object)
```

Arguments

object	see the method arguments for details

Methods

getF14,Model_14-method

getF14,Model_14-method *getF14,Model_14-method radiocarbon*

Description

Calculates the radiocarbon fraction for each pool at each time step.

Usage

```
## S4 method for signature 'Model_14'
getF14(object)
```

Arguments

object	: of class Model_14, The model

getF14C *This function*

Description

no Description

Usage

```
getF14C(object)
```

Arguments

object	see the method arguments for details

Methods

getF14C,Model_14-method

getF14C,Model_14-method *getF14C,Model_14-method read access to the models F14C variable*

Description

The model was created with a F14C object to describe the atmospheric 14C content. This method serves to investigate those settings from the model.

Usage

```
## S4 method for signature 'Model_14'
getF14C(object)
```

Arguments

object	: of class Model_14

`getF14R` *This function*

Description

no Description

Usage

```
getF14R(object)
```

Arguments

`object`	see the method arguments for details

Methods

`getF14R,Model_14-method`

`getF14R,Model_14-method` *getF14R,Model_14-method average radiocarbon fraction weighted by carbonrelease*

Description

Calculates the average radiocarbon fraction weighted by the amount of carbon release at each time step. $\overline{F_R} = \frac{\sum_{i=1}^{n} {}^{14}R_i}{\sum_{i=1}^{n} R_i}$ Where ${}^{14}R_i(t)$ is the time dependent release of ${}^{14}C$ of pool i and $R_i(t)$ the release of all carbon isotops of pool i. Since the result is always in Absolute Fraction Modern format wie have to convert it to Delta14C

Usage

```
## S4 method for signature 'Model_14'
getF14R(object)
```

Arguments

`object`	: of class Model_14, an object

`getFunctionDefinition` *getFunctionDefinition S4 generic*

Description

no Description

Usage

```
getFunctionDefinition(object)
```

Arguments

`object`	see the method arguments for details

Methods

`getFunctionDefinition,ConstInFlux-method`
`getFunctionDefinition,ConstLinDecompOp-method`
`getFunctionDefinition,DecompositionOperator-method`
`getFunctionDefinition,TimeMap-method`
`getFunctionDefinition,TransportDecompositionOperator-method`
`getFunctionDefinition,UnBoundInFlux-method`
`getFunctionDefinition,UnBoundLinDecompOp-method`

`getFunctionDefinition,ConstInFlux-method` *getFunctionDefinition,ConstInFlux-method*

Description

create the (constant) function of time that is required by the models

Usage

```
## S4 method for signature 'ConstInFlux'
getFunctionDefinition(object)
```

Arguments

`object`	: of class ConstInFlux

Value

A constant function of time that is used by the Models to represent the input fluxes.

getFunctionDefinition,ConstLinDecompOp-method

getFunctionDefinition,ConstLinDecompOp-method creates a constant timedependent function and returns it

Description

The method creates a timedependent function from the existing matrix describing the operator

Usage

```
## S4 method for signature 'ConstLinDecompOp'
getFunctionDefinition(object)
```

Arguments

object	: of class ConstLinDecompOp

getFunctionDefinition,DecompositionOperator-method

getFunctionDefinition,DecompositionOperator-method

Description

extract the function definition (the R-function)

Usage

```
## S4 method for signature 'DecompositionOperator'
getFunctionDefinition(object)
```

Arguments

object	: of class DecompositionOperator

getFunctionDefinition,TimeMap-method

getFunctionDefinition,TimeMap-method

Description

extract the function definition (the R-function)

Usage

```
## S4 method for signature 'TimeMap'
getFunctionDefinition(object)
```

Arguments

object	: of class TimeMap

`getFunctionDefinition,TransportDecompositionOperator-method`
getFunctionDefinition,TransportDecompositionOperator-method

Description

extract the function definition (the R-function) from the object

Usage

```
## S4 method for signature 'TransportDecompositionOperator'
getFunctionDefinition(object)
```

Arguments

`object`	: of class TransportDecompositionOperator

`getFunctionDefinition,UnBoundInFlux-method`
getFunctionDefinition,UnBoundInFlux-method creates a constant timedependent function and returns it

---

Description

The method creates a timedependent function from the existing matrix describing the operator

Usage

```
## S4 method for signature 'UnBoundInFlux'
getFunctionDefinition(object)
```

Arguments

`object`	: of class UnBoundInFlux

`getFunctionDefinition,UnBoundLinDecompOp-method`
getFunctionDefinition,UnBoundLinDecompOp-method creates a constant timedependent function and returns it

---

Description

The method creates a timedependent function from the existing matrix describing the operator

Usage

```
## S4 method for signature 'UnBoundLinDecompOp'
getFunctionDefinition(object)
```

Arguments

object	: of class UnBoundLinDecompOp

`getMeanTransitTime` *This function*

Description

no Description

Usage

```
getMeanTransitTime(object,
inputDistribution)
```

Arguments

object	a DecompOp Object.
inputDistribution	a vector of length equal to the number of pools. The entries are weights, which must sum to 1.

Methods

`getMeanTransitTime,ConstLinDecompOp-method`

References

Manzoni, S., G.G. Katul, and A. Porporato. 2009. Analysis of soil carbon transit times and age distributions using network theories. Journal of Geophysical Research-Biogeosciences 114, DOI: 10.1029/2009JG001070.

Thompson, M.~V. and Randerson, J.~T.: Impulse response functions of terrestrial carbon cycle models: method and application, Global Change Biology, 5, 371–394, 10.1046/j.1365-2486.1999.00235.x, 1999.

Bolin, B. and Rodhe, H.: A note on the concepts of age distribution and transit time in natural reservoirs, Tellus, 25, 58–62, 1973.

Eriksson, E.: Compartment Models and Reservoir Theory, Annual Review of Ecology and Systematics, 2, 67–84, 1971.

`getMeanTransitTime,ConstLinDecompOp-method`
getMeanTransitTime,ConstLinDecompOp-method compute the mean transit time

Description

This method computes the mean transit time for the linear time invariant system that can be constructed from the given operator and input distribution.

It relies on the mehtod `getTransitTimeDistributionDensity` using the same arguments.

Usage

```
## S4 method for signature 'ConstLinDecompOp'
getMeanTransitTime(object,
inputDistribution)
```

Arguments

`object` : of class ConstLinDecompOp

`inputDistribution`
: distribution of the inputs

Details

To compute the mean transit time for the distribution we have to compute the integral

$$\bar{T} = \int_0^\infty T \cdot S_r \left(\frac{\vec{I}}{I}, 0, T \right) \, dT$$

for the numerically computed density. To avoid issues with numerical integration we dont use ∞ as upper limit but cut off the integragion interval prematurely. For this purpose we calculate a maximum response time of the system as *Lasaga*

$$\tau_{cycle} = \frac{1}{|\min(\lambda_i)|}$$

where λ_i are non-zero eigenvalues of the matrix $\mathbf{A}$.

References

Lasaga, A.: The kinetic treatment of geochemical cycles, Geochimica et Cosmochimica Acta, 44, 815 – 828, doi10.1016/0016-7037(80)90263-X, 1980.

`getReleaseFlux` *This function*

Description

no Description

Usage

```
getReleaseFlux(object)
```

Arguments

`object`	An model object (the actual class depends on the method e.g. Model or Model14

Value

A matrix. Every column represents a pool and every row a point in time

Methods

`getReleaseFlux,Model-method`
`getReleaseFlux,NlModel-method`

`getReleaseFlux,Model-method`
getReleaseFlux,Model-method get the release rate for all pools

---

Description

The method computes the release of carbon per time for all points in time specified in the Model objects time slot.

Usage

```
## S4 method for signature 'Model'
getReleaseFlux(object)
```

Arguments

`object`	: of class Model, an object of class Model created by a call to a constructor e.g. `Model`, `GeneralModel`or other model creating functions.

Details

This function takes a Model object, which represents a system of ODEs

$$\frac{d\mathbf{C}(t)}{dt} = \mathbf{I}(t) + \mathbf{A}(t)\mathbf{C}(t)$$

solves the system for $\mathbf{C}(t)$, calculates the release coefficients $\mathbf{R}(t)$, and computes the release flux as $\mathbf{R}(t)\mathbf{C}(t)$. The numerical solver used can be specified in the model creating functions like e.g. `Model`.

Value

A n x m matrix of release fluxes with m columns representing the number of pools, and n rows representing the values for each time step as specified by the argument `t` in `Model`, `GeneralModel` or another model creating function.

`getReleaseFlux,NlModel-method` *getReleaseFlux,NlModel-method get the release rate for all pools*

Description

The method computes the release of carbon per time for all points in time specified in the objects time slot.

Usage

```
## S4 method for signature 'NlModel'
getReleaseFlux(object)
```

Arguments

`object`	: of class NlModel, an object of class NlModel

`getReleaseFlux14` *This function*

Description

no Description

Usage

```
getReleaseFlux14(object)
```

Arguments

`object`	see the method arguments for details

Methods

`getReleaseFlux14,Model_14-method`

getReleaseFlux14,Model_14-method *getReleaseFlux14,Model_14-method 14C respiration rate for all pools*

Description

The function computes the ^{14}C release flux (mass per time) for all pools. Note that the respiration coefficients for ^{14}C do not change in comparison to the total C case. The fraction of ^{14}C lost by respiration is not greater for ^{14}C although the decay is faster due to the contribution of radioactivity.

Usage

```
## S4 method for signature 'Model_14'
getReleaseFlux14(object)
```

Arguments

object	: of class Model_14, the model.

getTimeRange *getTimeRange S4 generic*

Description

no Description

Usage

```
getTimeRange(object)
```

Arguments

object	see the method arguments for details

Methods

```
getTimeRange,ConstInFlux-method
getTimeRange,ConstLinDecompOp-method
getTimeRange,DecompositionOperator-method
getTimeRange,TimeMap-method
getTimeRange,UnBoundInFlux-method
getTimeRange,UnBoundLinDecompOp-method
```

`getTimeRange,ConstInFlux-method`
getTimeRange,ConstInFlux-method time domain of the function

Description

The method returns a vector containing the start and end time where the intepolation is valid. Since the class `ConstInFlux` represents an input stream constant in time it will return -infinity,+infinity

Usage

```
## S4 method for signature 'ConstInFlux'
getTimeRange(object)
```

Arguments

`object`	: of class ConstInFlux

`getTimeRange,ConstLinDecompOp-method`
getTimeRange,ConstLinDecompOp-method return an (infinite) time range since the operator is constant

---

Description

some functions dealing with DecompOps in general rely on this so we have to implement it even though the timerange is always the same: (-inf,inf)

Usage

```
## S4 method for signature 'ConstLinDecompOp'
getTimeRange(object)
```

Arguments

`object`	: of class ConstLinDecompOp

getTimeRange,DecompositionOperator-method

getTimeRange,DecompositionOperator-method ask for the boundaries of the underlying time interval

Description

The method returns the time range of the given object It is (probably mostly) used internally to make sure that time dependent functions retrieved from data are not used outside the interval where they are valid.

Usage

```
## S4 method for signature 'DecompositionOperator'
getTimeRange(object)
```

Arguments

object	: of class DecompositionOperator

getTimeRange,TimeMap-method

getTimeRange,TimeMap-method ask for the boundaries of the underlying time interval

Description

The method returns the time range of the given object It is probably mostly used internally to make sure that time dependent functions retrieved from data are not used outside the interval where they are valid.

Usage

```
## S4 method for signature 'TimeMap'
getTimeRange(object)
```

Arguments

object	: of class TimeMap, An object of class TimeMap or one that inherits from TimeMap

---

`getTimeRange,UnBoundInFlux-method`

getTimeRange,UnBoundInFlux-method return an (infinite) time range since the function is assumed to be valid for all times

---

Description

some functions dealing with DecompOps in general rely on this so we have to implement it even though the timerange is always the same: (-inf,inf)

Usage

```
## S4 method for signature 'UnBoundInFlux'
getTimeRange(object)
```

Arguments

`object`	: of class UnBoundInFlux

`getTimeRange,UnBoundLinDecompOp-method`

getTimeRange,UnBoundLinDecompOp-method return an (infinite) time range since the function is assumed to be valid for all times

---

Description

some functions dealing with DecompOps in general rely on this so we have to implement it even though the timerange is always the same: (-inf,inf)

Usage

```
## S4 method for signature 'UnBoundLinDecompOp'
getTimeRange(object)
```

Arguments

`object`	: of class UnBoundLinDecompOp

`getTimes` *getTimes S4 generic*

Description

no Description

Usage

```
getTimes(object)
```

Arguments

`object`	see the method arguments for details

Methods

`getTimes,Model-method`
`getTimes,NlModel-method`

`getTimes,Model-method` *getTimes,Model-method*

Description

This functions extracts the times argument from an argument of class Model

Usage

```
## S4 method for signature 'Model'
getTimes(object)
```

Arguments

`object`	: of class Model

`getTimes,NlModel-method`
getTimes,NlModel-method

Description

This functions extracts the times argument from an object of class NlModel

Usage

```
## S4 method for signature 'NlModel'
getTimes(object)
```

Arguments

`object`	: of class NlModel

`getTransitTimeDistributionDensity`
This function

Description

no Description

Usage

```
getTransitTimeDistributionDensity(object,
inputDistribution,
times)
```

Arguments

`object`	a protoDecompOp Object
`inputDistribution`	a vector of length equal to the number of pools. The entries are weights. That means that their sume must be equal to one!
`times`	the times for which the distribution density is sought

Methods

`getTransitTimeDistributionDensity,ConstLinDecompOp-method`

References

Manzoni, S., G.G. Katul, and A. Porporato. 2009. Analysis of soil carbon transit times and age distributions using network theories. Journal of Geophysical Research-Biogeosciences 114, DOI: 10.1029/2009JG001070.

`getTransitTimeDistributionDensity,ConstLinDecompOp-method`
getTransitTimeDistributionDensity,ConstLinDecompOp-method compute the TransitTimeDistributionDensity

Description

This mehtod computes the probability density of the transit time of the linear time invariant system that can be constructed from the given operator and input distribution.

Usage

```
## S4 method for signature 'ConstLinDecompOp'
getTransitTimeDistributionDensity(object,
inputDistribution,
times)
```

Arguments

`object`	: of class ConstLinDecompOp
`inputDistribution`	: distribution of the inputs
`times`	: the points in time where the solution is sought

Details

In a forthcoming paper *SoilRv1.2* we derive the algorithm used in this implementation under the assumption of steady conditions having prevailed infinitely. We arrive at a formulation well known from the literature about time invariant linlear systems, cited e.g. in *ManzoniJGR*.
The somehow amazing result is that the weight of the transit time density $\psi(T)$ for a *transit time* T for the steady state system is identical to the output $O(T)$ observed at time T of a *different* system which started with a normalized impulsive input $\frac{\vec{I}}{I}$ at time $T = 0$, where $I = \sum_{k=1}^{m} i_k$ is the cumulative input flux to all pools.
This fact simpliefies the computation considerably. Translated into the language of an ode solver an impulsive input becomes a start vector $\frac{\vec{I}}{I}$ at time $T = 0$ and $O(T)$ the respiration related to the solution of the initial value problem observed at time T.

$$\psi(T) = S_r\left(\frac{\vec{I}}{I}, 0, T\right)$$

Note that from the perspective of the ode solver S_r depends on the decomposition operator and the distribution of the input among the pools only. It is therefor possible to implement the transit time distribution as a function of the decomposition operator and the fixed input flux distribution. To insure steady state conditions the decomposition operator is not allowed to be a true function of time. We therefor implement the method only for the subclass `ConstLinDecompOp`
Remark:
The decision to implement this method for `transitTimeDensity` especially for objects of class `ConstLinDecompOp` reflects the fact that an arbitrary model in SoilR is by no means bound to end up in steady state. To insure this we would have to ignore the input part of a user created model which would be confusing.

Remark:
In future versions of SoilR it will be possible to compute a dynamic, time dependent transit time distribution for objects of class `Model` with a time argument specifying for which time the distribution is sought. The steady state results computed here could than be reproduced with the user responsible for providing a model actually reaching it.

References

Manzoni, S., Katul, G.~G., and Porporato, A.: Analysis of soil carbon transit times and age distributions using network theories, J. Geophys. Res., 114,

---

`HarvardForest14CO2` *Delta14C in soil CO2 efflux from Harvard Forest*

Description

Measurements of Delta14C in soil CO2 efflux conducted at Harvard Forest, USA, between 1996 and 2010.

Usage

```
data(HarvardForest14CO2)
```

Format

A data frame with the following 3 variables.

`Year` A numeric vector with the date of measurement in years

`D14C` A numeric vector with the value of the Delta 14C value measured in CO2 efflux in per mil

`Site` A factor indicating the site where measurements were made. NWN: Northwest Near, Drydown: Rainfall exclusion experiment.

Details

Samples for isotopic measurements of soil CO2 efflux were collected from chambers that enclosed an air headspace in contact with the soil surface in the absence of vegetation using a closed dynamic chamber system to collect accumulated CO2 in stainless steel traps with a molecular sieve inside. See Sierra et al. (2012) for additional details.

References

Sierra, C. A., Trumbore, S. E., Davidson, E. A., Frey, S. D., Savage, K. E., and Hopkins, F. M. 2012. Predicting decadal trends and transient responses of radiocarbon storage and fluxes in a temperate forest soil, Biogeosciences, 9, 3013-3028, doi:10.5194/bg-9-3013-2012

Examples

```
plot(HarvardForest14CO2[,1:2])
```

Hua2013 *Atmospheric radiocarbon for the period 1950-2010 from Hua et al. (2013)*

Description

Atmospheric radiocarbon for the period 1950-2010 reported by Hua et al. (2013) for 5 atmospheric zones.

Usage

```
data(Hua2013)
```

Format

A list containing 5 data frames, each representing an atmospheric zone. The zones are: NHZone1: northern hemisphere zone 1, NHZone2: northern hemisphere zone 2, NHZone3: northern hemisphere zone 3, SHZone12: southern hemisphere zones 1 and 2, SHZone3: southern hemisphere zone 3. Each data frame contains a variable number of observations on the following 5 variables.

`Year.AD` Year AD

`mean.Delta14C` mean value of atmospheric radiocarbon reported as Delta14C

`sd.Delta14C` standard deviation of atmospheric radiocarbon reported as Delta14C

`mean.F14C` mean value of atmospheric radiocarbon reported as fraction modern F14C

`sd.F14` standard deviation of atmospheric radiocarbon reported as fraction modern F14C

Details

This dataset corresponds to Table S3 from Hua et al. (2013). For additional details see the original publication.

Source

https://journals.uair.arizona.edu/index.php/radiocarbon/article/view/16177

References

Hua Q., M. Barbetti, A. Z. Rakowski. 2013. Atmospheric radiocarbon for the period 1950-2010. Radiocarbon 55(4):2059-2072.

Examples

```
plot(Hua2013$NHZone1$Year.AD, Hua2013$NHZone1$mean.Delta14C,
     type="l",xlab="Year AD",ylab=expression(paste(Delta^14,"C (\u2030)")))
lines(Hua2013$NHZone2$Year.AD,Hua2013$NHZone2$mean.Delta14C,col=2)
lines(Hua2013$NHZone3$Year.AD,Hua2013$NHZone3$mean.Delta14C,col=3)
lines(Hua2013$SHZone12$Year.AD,Hua2013$SHZone12$mean.Delta14C,col=4)
lines(Hua2013$SHZone3$Year.AD,Hua2013$SHZone3$mean.Delta14C,col=5)
legend(
"topright",
```

```
c(
"Norther hemisphere zone 1",
"Norther hemisphere zone 2",
"Norther hemisphere zone 3",
                "Southern hemisphere zones 1 and 2",
"Southern Hemispher zone 3"
),
lty=1,
col=1:5,
bty="n"
)
```

`ICBMModel` *Implementation of the Introductory Carbon Balance Model (ICBM)*

Description

This function is an implementation of the Introductory Carbon Balance Model (ICBM). This is simply a two pool model connected in series.

Usage

```
ICBMModel(t,
ks=c(k1 = 0.8, k2 = 0.00605),
h=0.13,
r=1.32,
c0=c(Y0 = 0.3, O0 = 3.96),
In=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sougth.
`ks`	A vector of length 2 with the decomposition rates for the young and the old pool.
`h`	Humufication coefficient (transfer rate from young to old pool).
`r`	External (environmental or edaphic) factor.
`c0`	A vector of length 2 with the initial value of carbon stocks in the young and old pool.
`In`	Mean annual carbon input to the soil.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE forces the constructor to create the model even if it is invalid

References

Andren, O. and T. Katterer. 1997. ICBM: The Introductory Carbon Balance Model for Exploration of Soil Carbon Balances. Ecological Applications 7:1226-1236.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
# examples from external files
# inst/examples/exICBMModel.R exICBMModel_paper:

    # This example reproduces the simulations
    # presented in Table 1 of Andren and Katterer (1997).
    # First, the model is run for different values of the
    # parameters representing different field experiments.
    times=seq(0,20,by=0.1)
    Bare=ICBMModel(t=times) #Bare fallow
    pNpS=ICBMModel(t=times, h=0.125, r=1,    c0=c(0.3,4.11),  In=0.19+0.095) #+N +Straw
    mNpS=ICBMModel(t=times, h=0.125, r=1.22, c0=c(0.3, 4.05), In=0.19+0.058) #-N +Straw
    mNmS=ICBMModel(t=times, h=0.125, r=1.17, c0=c(0.3, 3.99), In=0.057) #-N -Straw
    pNmS=ICBMModel(t=times, h=0.125, r=1.07, c0=c(0.3, 4.02), In=0.091) #+N -Straw
    FM=ICBMModel(t=times, h=0.250, r=1.10, c0=c(0.3, 3.99), In=0.19+0.082) #Manure
    SwS=ICBMModel(t=times, h=0.340, r=0.97, c0=c(0.3, 4.14), In=0.19+0.106) #Sewage Sludg
    SS=ICBMModel(t=times, h=0.125, r=1.00, c0=c(0.25, 4.16), In=0.2)  #Steady State

    #The amount of carbon for each simulation is recovered with the function getC
    CtBare=getC(Bare)
    CtpNpS=getC(pNpS)
    CtmNpS=getC(mNpS)
    CtmNmS=getC(mNmS)
    CtpNmS=getC(pNmS)
    CtFM=getC(FM)
    CtSwS=getC(SwS)
    CtSS=getC(SS)

    #This plot reproduces Figure 1 in Andren and Katterer (1997)
    plot(times,
      rowSums(CtBare),
      type="l",
      ylim=c(0,8),
      xlim=c(0,20),
      ylab="Topsoil carbon mass (kg m-2)",
      xlab="Time (years)"
    )
    lines(times,rowSums(CtpNpS),lty=2)
    lines(times,rowSums(CtmNpS),lty=3)
    lines(times,rowSums(CtmNmS),lty=4)
    lines(times,rowSums(CtpNmS),lwd=2)
    lines(times,rowSums(CtFM),lty=2,lwd=2)
    lines(times,rowSums(CtSwS),lty=3,lwd=2)
    #lines(times,rowSums(CtSS),lty=4,lwd=2)
    legend("topleft",
      c("Bare fallow",
        "+N +Straw",
        "-N +Straw",
        "-N -Straw",
        "+N -Straw",
        "Manure",
       "Sludge"
```

```
  ),
  lty=c(1,2,3,4,1,2,3),
  lwd=c(1,1,1,1,2,2,2),
  bty="n"
)
```

`InFlux-class` *InFlux S4 class*

Description

All models need to specify the influx of material to the pools. This parameter will be represented as an object of one of the subclasses of this class. The most general form of influx supported up to now is a vector valued function of time represented by `BoundInFlux-class`. In the most simple case it is constant and represented by an object of class `ConstInFlux-class`. Such an object can for instance be created from a numeric vector.

Methods

Exported methods directly defined for class InFlux:

GeneralInFlux `signature(object = "InFlux")`: ... `GeneralInFlux,InFlux-method`

Subclasses

`BoundInFlux`
`ConstInFlux`
`UnBoundInFlux`

Constructors

The class is abstract (`contains "VIRTUAL"`). It can therefore not be instanciated directly. Look at non virtual subclasses and their constructors!

There is also an abstract factory that produces instances of different subclasses depending on the input:

`GeneralInFlux`

`IntCal09` *Northern Hemisphere atmospheric radiocarbon for the pre-bomb period*

Description

Northern Hemisphere atmospheric radiocarbon calibration curve for the period 0 to 50,000 yr BP.

Usage

```
data(IntCal09)
```

Format

A data frame with 3522 observations on the following 5 variables.

`CAL.BP` Calibrated age in years Before Present (BP).

`C14.age` C14 age in years BP.

`Error` Error estimate for `C14.age`.

`Delta.14C` Delta.14C value in per mil.

`Sigma` Standard deviation of `Delta.14C` in per mil.

Details

`Deltal.14C` is age-corrected as per Stuiver and Polach (1977). All details about the derivation of this dataset are provided in Reimer et al. (2009).

Source

http://www.radiocarbon.org/IntCal09%20files/intcal09.14c

References

P. Reimer, M.Baillie, E. Bard, A. Bayliss, J. Beck, P. Blackwell, C. Ramsey, C. Buck, G. Burr, R. Edwards, et al. 2009. IntCal09 and Marine09 radiocarbon age calibration curves, 0 - 50,000 years cal bp. Radiocarbon, 51(4):1111 - 1150.

M. Stuiver and H. A. Polach. 1977. Rerporting of C-14 data. Radiocarbon, 19(3):355 - 363.

Examples

```
par(mfrow=c(2,1))
plot(IntCal09$CAL.BP, IntCal09$C14.age, type="l")
polygon(x=c(IntCal09$CAL.BP,rev(IntCal09$CAL.BP)),
y=c(IntCal09$C14.age+IntCal09$Error,rev(IntCal09$C14.age-IntCal09$Error)),
col="gray",border=NA)
lines(IntCal09$CAL.BP,IntCal09$C14.age)

plot(IntCal09$CAL.BP,IntCal09$Delta.14C,type="l")
polygon(x=c(IntCal09$CAL.BP,rev(IntCal09$CAL.BP)),
y=c(IntCal09$Delta.14C+IntCal09$Sigma,rev(IntCal09$Delta.14C-IntCal09$Sigma)),
col="gray",border=NA)
lines(IntCal09$CAL.BP,IntCal09$Delta.14C)
par(mfrow=c(1,1))
```

IntCal13 *Atmospheric radiocarbon for the 0-50,000 yr BP period*

Description

Atmospheric radiocarbon calibration curve for the period 0 to 50,000 yr BP. This is the most recent update to the internationally agreed calibration curve and superseds `IntCal09`.

Usage

```
data(IntCal13)
```

Format

A data frame with 5140 observations on the following 5 variables.

`CAL.BP` Calibrated age in years Before Present (BP).

`C14.age` C14 age in years BP.

`Error` Error estimate for `C14.age`.

`Delta.14C` Delta.14C value in per mil.

`Sigma` Standard deviation of `Delta.14C` in per mil.

Details

`Deltal.14C` is age-corrected as per Stuiver and Polach (1977). All details about the derivation of this dataset are provided in Reimer et al. (2013).

Source

http://www.radiocarbon.org/IntCal13%20files/intcal13.14c

References

Reimer PJ, Bard E, Bayliss A, Beck JW, Blackwell PG, Bronk Ramsey C, Buck CE, Cheng H, Edwards RL, Friedrich M, Grootes PM, Guilderson TP, Haflidason H, Hajdas I, Hatte C, Heaton TJ, Hogg AG, Hughen KA, Kaiser KF, Kromer B, Manning SW, Niu M, Reimer RW, Richards DA, Scott EM, Southon JR, Turney CSM, van der Plicht J. 2013. IntCal13 and MARINE13 radiocarbon age calibration curves 0-50000 years calBP. Radiocarbon 55(4): 1869-1887. DOI: 10.2458/azu_js_rc.55.16947

M. Stuiver and H. A. Polach. 1977. Rerporting of C-14 data. Radiocarbon, 19(3):355 - 363.

Examples

```
plot(IntCal13$CAL.BP,IntCal13$C14.age-IntCal13$Error,type="l",col=2,
     xlab="cal BP",ylab="14C BP")
lines(IntCal13$CAL.BP,IntCal13$C14.age+IntCal13$Error,col=2)

plot(IntCal13$CAL.BP,IntCal13$Delta.14C+IntCal13$Sigma,type="l",col=2,
     xlab="cal BP",ylab="Delta14C")
lines(IntCal13$CAL.BP,IntCal13$Delta.14C-IntCal13$Sigma,col=2)
```

linesCPool — *Add lines with the output of* getC14, getC, *or* getReleaseFlux *to an existing plot*

Description

This function adds lines to a plot with the C content, the C release, or Delta 14C value of each pool over time. Needs as input a matrix obtained after a call to getC14, getC, or getReleaseFlux.

Usage

```
linesCPool(t,
mat,
col,
...)
```

Arguments

t	A vector containing the time points for plotting.
mat	A matrix object obtained after a call to getC14, getC, or getReleaseFlux.
col	A color palette specifying color lines for each pool (columns of mat).
...	Other arguments passed to plot.

listProduct — *tensor product of lists*

Description

Creates a list of all combinations of the elements of the inputlists (like a "tensor product list " The list elements can be of any class. The function is used in examples and tests to produce all possible combinations of arguments to a function. look at the tests for example usage

Usage

```
listProduct(...)
```

Arguments

...	lists

Value

a list of lists each containing one combinations of the elements of the input lists

Examples

```
listProduct(list('a','b'),list(1,2))
```

`Model` *Constructor for class Model*

Description

This function creates an object of class Model, The arguments can be given in different form as long as they can be converted to the necessary internal building blocks. (See the links)

Usage

```
Model(t,
A,
ivList,
inputFluxes,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`A`	something that can be converted by GeneralDecompOp to any of the available subclasses of `DecompOp`.
`ivList`	A numeric vector containing the initial amount of carbon for the n pools. The length of this vector is equal to the number of pools. This is checked by an internal function.
`inputFluxes`	something that can be converted by GeneralInFlux to any of the available subclasses of InFlux.
`solverfunc`	The function used to actually solve the ODE system. The default is `deSolve.lsoda.wrapper` but you can also provide your own function that the same interface.
`pass`	Forces the constructor to create the model even if it does not pass internal sanity checks

Details

This function `Model` wraps the internal constructor of class Model. The internal constructor requires the argument `A` to be of class DecompOp and argument `inputFluxes` to be of class InFlux. Before calling the internal constructor `Model` calls GeneralDecompOp on its argument `A` and GeneralInFlux on its argument `inputFluxes` to convert them into the required classes. Both are generic functions. Follow the links to see for which kind of inputs conversion methods are available. The attempted conversion allows great flexibility with respect to arguments and independence from the actual implementation. However if your code uses the wrong argument the error will most likely occur in the delegate functions. If this happens analyse the errormassage (or use `traceback()`) to see which function was called and try to call the constructor of the desired subclass explicitly with your arguments. The subclasses are linked in the class documentation DecompOp or InFlux respectively.

Note also that this function checks its arguments quite elaborately and tries to detect accidental unreasonable combinations, especially concerning two kinds of errors.

1. unintended extrapolation of time series data

2. violoations of massbalance by the DecompOp argument.

SoilR has a lot of unit tests which are installed i with the package and are sometimes instructive as examples. To see example scenarios for parameter check look at:
> system.file("tests", "runit.correctness_of_Model.R", package = "SoilR")
[1] "/home/mm/R/x86_64-pc-linux-gnu-library/3.4/SoilR/tests/runit.correctness_of_Model.R"

Value

An object of class Model that can be queried by many methods to be found there.

See Also

This function is called by many of the predefinedModels.
Package functions called in the examples:
example.2DInFlux.Args,
example.2DGeneralDecompOpArgs,

Examples

```
# examples from external files
# inst/tests/requireSoilR/runit.all.possible.Model.arguments.R test.all.possible.Model.ar

  # This example shows different kinds of arguments to the function Model.
  # The model objects we will build will share some common features.
  #  - two pools
  #  - initial values

       iv<-  c(5,6)

  #  - times

       times <- seq(1,10,by=0.1)

  # The other parameters A and inputFluxes will be different
  # The function Model will transform these arguments
  # into objects of the classes required by the internal constructor.
  # This leads to a number of possible argument types.
  # We demonstrate some of the possibilities here.
  # Let us first look at the choeices for argument 'A'.

  #)
  possibleAs  <- example.2DGeneralDecompOpArgs()

  # Since "Model" will call "GeneralInFlux" on its "inputFluxes"
  # argument there are again different choices
  # we have included a function in SoilR that produces 2D examples

  possibleInfluxes <- example.2DInFlux.Args()
 print(possibleInfluxes$I.vec)
  # We can build a lot of  models from the possible combinations
  # for instance
  #m1 <- Model(
  #        t=times,
  #        A=matrix(nrow=2,byrow=TRUE,c(-0.1,0,0,-0.2)),
```

```
#         ivList=iv,
#         inputFluxes=possibleInfluxes$I.vec)
## We now produce that all combinations of As and InputFluxes
combinations <- listProduct(possibleAs,possibleInfluxes)
print(length(combinations))
# an a Model for each
models <- lapply(
            combinations,
            function(combi){
              #Model(t=times,A=combi$A,ivList=iv,inputFluxes=combi$I)
              Model(t=times,A=combi[[1]],ivList=iv,inputFluxes=combi[[2]])
            }
          )
## lets check that we can compute something#
lapply(models,getC)
```

`Model-class` *Model*

Description

The class Model is the focal point of SoilR.

1. It combines all the components that are needed to solve the initial value problem for the pool contents. $\vec{C}$.
2. It provides the single argument for the different functions that are available to compute various results from the solution of the initial value problem. See subsection `Methods` and the examples.)

Details

The initial value problem is given by:

- the ordinary differential equation $\dot{\vec{C}} = \mathbf{A}(t)\vec{C} = \vec{I}(t)$
- the intial Values $\vec{C}_0 = \vec{C}(t_0)$
- for the times $\{t_0, ...t_m\}$.

In an object of class Model the components are represented as follows:

- The time-dependent matrix valued function $\vec{A}(t)$ is represented by an object of a class that inherits from class `DecompOp`. Such objects can be created in different ways from functions, matrices or data. (see the subclasses of `DecompOp` and especially their `Constructors` sections. and the `examples` section of this help page.
- The vector-valued time-dependent function $\vec{I}(t)$ is in SoilR represented by an object of a class that inherits from class InFlux `InFlux`. Such objects can be created from functions, constant vectors and data. (see the subclasses of `InFlux` and especially their `Constructors` sections.
- The times for which the results are computed are represented by a numeric vector.
- The initial values are represented by a numeric vector

Methods

Exported methods directly defined for class Model:

[`signature(x = "Model", i = "character", j = "missing", drop = "missing"):` ... `[,Model,character,missing,missing-method`

getAccumulatedRelease `signature(object = "Model"):` ... `getAccumulatedRelease,Model-metho`

getC `signature(object = "Model"):` ... `getC,Model-method`

getDecompOp `signature(object = "Model"):` ... `getDecompOp,Model-method`

getReleaseFlux `signature(object = "Model"):` ... `getReleaseFlux,Model-method`

getTimes `signature(object = "Model"):` ... `getTimes,Model-method`

Subclasses

`Model_14`

Constructors

`Model`
Please also look at constructors of non virtual subclasses: `Model_14`.

Examples

```
# examples from external files
# inst/examples/ModelExamples.R CorrectNonautonomousLinearModelExplicit:

  # This example describes the creation and use of a Model object that
  # is defined by time dependent functions for decomposition and influx.
  # The constructor of the Model-class  (see  ?Model)
  # works for different combinations of
  # arguments.
  # Although Model (the constructor function for objects of this class
  # accepts many many more convienient kinds of arguments,
  # we will in this example call the constructor whith arguments which
  # are of the same type as one of hte current internal
  # representations in the
  # Model object and create these arguments explicitly beforehand
  # to demonstrate the approach with the most flexibility.
  # We start with the Decomposition Operator.
  # For this example we assume that we are able to describe the
  # decomposition ofperator  by explicit R functions that are valid
  # for a finite time interval.
  # Therefore we choose the appropriate  sub class BoundLinDecompOp
  # of DecompOp explicitly.  (see ?'BoundLinDecompOp-class')
  A=BoundLinDecompOp(
    ## We call the generic constructor (see ?BoundLindDcompOp)
    ## which has a method
    ## that takes a matrix-valued function of time as its first argument.
    ## (Although Model accepts time series data directly and
    ## will derive the internally used interpolating for you,
    ## the function argument could for instance represent the result
    ## of a very sophisticated interpolation performed by yourself)
    function(t){
      matrix(nrow=3,ncol=3,byrow=TRUE,
```

```
        c(
          -1,    0,        0,
         0.5,   -2,        0,
           0,    1, sin(t)-1
        )
      )
    },
    ## The other two arguments describe the time interval where the
    ## function is valid (the domain of the function)
    ## The interval will be checked against the domain of the InFlux
    ## argument of Model and against its 't' argument to avoid
    ## invalid computations outside the domain.
    ## (Inf and -Inf are possible values, but should only be used
    ## if the function is really valid for all times, which is
    ## especially untrue for functions resulting from interpolations,
    ## which are usually extremely misleading for arguments outside the
    ## domain covered by the data that has been used for the interpolation.)
    ## This is a safety net against wrong results origination from unitendet EXTRApolatic
    starttime=0,
    endtime=20
  )
  I=BoundInFlux(
     ## The first argument is a vector-valued function of time
     function(t){
       matrix(nrow=3,ncol=1,byrow=TRUE,
           c(-1,    0,    0)
       )
     },
     ## The other two arguments describe the time interval where the
     ## function is valid (the domain of the function)
     starttime=0,
     endtime=40
  )
  ## No we specify the points in time where we want
  ## to compute results
  t_start=0
  t_end=10
  tn=50
  timestep <- (t_end-t_start)/tn
  times <- seq(t_start,t_end,timestep)
  ## and the start values
  sv=c(0,0,0)
  mod=Model(t=times,A,sv,I)

  ## No we use the model to compute some results
  getC(mod)
  getReleaseFlux(mod)
  #also look at the methods section of Model-class
```

`Model_14` *general constructor for class Model_14*

Description

This method tries to create an object from any combination of arguments that can be converted into the required set of building blocks for the Model_14 for n arbitrarily connected pools.

Usage

```
Model_14(t,
A,
ivList,
initialValF,
inputFluxes,
inputFc,
c14DecayRate=-0.0001209681,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
A	something that can be converted by GeneralDecompOp to any of the available subclasses of DecompOp.
ivList	A vector containing the initial amount of carbon for the n pools. The length of this vector is equal to the number of pools and thus equal to the length of k. This is checked by an internal function.
initialValF	An object equal or equivalent to class ConstFc containing a vector with the initial values of the radiocarbon fraction for each pool and a format string describing in which format the values are given.
inputFluxes	something that can be converted by GeneralInFlux to any of the available subclasses of InFlux.
inputFc	An object describing the fraction of C_14 in per mille (different formats are possible)
c14DecayRate	the rate at which C_14 decays radioactivly. If you don't provide a value here we assume the following value: k=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year. Thus beside time itself it also affects decay rates the inputrates and the output
solverfunc	The function used by to actually solve the ODE system. This can be deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	Forces the constructor to create the model even if it is invalid

Value

A model object that can be further queried.

See Also

TwopParallelModel, TwopSeriesModel, TwopFeedbackModel

Examples

```
# examples from external files
# inst/tests/requireSoilR/runit.all.possible.Model.arguments.R test.all.possible.Model.ar

  # This example shows different kinds of arguments to the function Model.
  # The model objects we will build will share some common features.
  #  - two pools
  #  - initial values
```

```
      iv<-  c(5,6)

  #  - times

      times <- seq(1,10,by=0.1)

  # The other parameters A and inputFluxes will be different
  # The function Model will transform these arguments
  # into objects of the classes required by the internal constructor.
  # This leads to a number of possible argument types.
  # We demonstrate some of the possibilities here.
  # Let us first look at the choeices for argument 'A'.

  #)
  possibleAs  <- example.2DGeneralDecompOpArgs()

  # Since "Model" will call "GeneralInFlux" on its "inputFluxes"
  # argument there are again different choices
  # we have included a function in SoilR that produces 2D examples

  possibleInfluxes <- example.2DInFlux.Args()
 print(possibleInfluxes$I.vec)
  # We can build a lot of  models from the possible combinations
  # for instance
  #m1 <- Model(
  #        t=times,
  #        A=matrix(nrow=2,byrow=TRUE,c(-0.1,0,0,-0.2)),
  #        ivList=iv,
  #        inputFluxes=possibleInfluxes$I.vec)
  ## We now produce that all combinations of As and InputFluxes
  combinations <- listProduct(possibleAs,possibleInfluxes)
  print(length(combinations))
  # an a Model for each
  models <- lapply(
              combinations,
              function(combi){
                #Model(t=times,A=combi$A,ivList=iv,inputFluxes=combi$I)
                Model(t=times,A=combi[[1]],ivList=iv,inputFluxes=combi[[2]])
              }
            )
  ## lets check that we can compute something#
  lapply(models,getC)

# inst/examples/ModelExamples.R CorrectNonautonomousLinearModelExplicit:

  # This example describes the creation and use of a Model object that
  # is defined by time dependent functions for decomposition and influx.
  # The constructor of the Model-class  (see  ?Model)
  # works for different combinations of
  # arguments.
  # Although Model (the constructor function for objects of this class
  # accepts many many more convienient kinds of arguments,
  # we will in this example call the constructor whith arguments which
  # are of the same type as one of hte current internal
  # representations in the
  # Model object and create these arguments explicitly beforehand
```

```
# to demonstrate the approach with the most flexibility.
# We start with the Decomposition Operator.
# For this example we assume that we are able to describe the
# decomposition ofperator  by explicit R functions that are valid
# for a finite time interval.
# Therefore we choose the appropriate  sub class BoundLinDecompOp
# of DecompOp explicitly.  (see ?'BoundLinDecompOp-class')
A=BoundLinDecompOp(
  ## We call the generic constructor (see ?BoundLindDcompOp)
  ## which has a method
  ## that takes a matrix-valued function of time as its first argument.
  ## (Although Model accepts time series data directly and
  ## will derive the internally used interpolating for you,
  ## the function argument could for instance represent the result
  ## of a very sophisticated interpolation performed by yourself)
  function(t){
    matrix(nrow=3,ncol=3,byrow=TRUE,
       c(
         -1,    0,        0,
        0.5,   -2,        0,
          0,    1, sin(t)-1
      )
    )
  },
  ## The other two arguments describe the time interval where the
  ## function is valid (the domain of the function)
  ## The interval will be checked against the domain of the InFlux
  ## argument of Model and against its 't' argument to avoid
  ## invalid computations outside the domain.
  ## (Inf and -Inf are possible values, but should only be used
  ## if the function is really valid for all times, which is
  ## especially untrue for functions resulting from interpolations,
  ## which are usually extremely misleading for arguments outside the
  ## domain covered by the data that has been used for the interpolation.)
  ## This is a safety net against wrong results origination from unitendet EXTRApolatio
  starttime=0,
  endtime=20
)
I=BoundInFlux(
   ## The first argument is a vector-valued function of time
   function(t){
     matrix(nrow=3,ncol=1,byrow=TRUE,
         c(-1,    0,    0)
     )
   },
   ## The other two arguments describe the time interval where the
   ## function is valid (the domain of the function)
   starttime=0,
   endtime=40
)
## No we specify the points in time where we want
## to compute results
t_start=0
t_end=10
tn=50
timestep <- (t_end-t_start)/tn
times <- seq(t_start,t_end,timestep)
```

```
## and the start values
sv=c(0,0,0)
mod=Model(t=times,A,sv,I)

## No we use the model to compute some results
getC(mod)
getReleaseFlux(mod)
#also look at the methods section of Model-class
```

`Model_14-class` *Model_14*

Description

This class extends `Model`, to represent ^{14}C decay.

1.
   - As `Model` it contains all the components that are needed to solve the initial value problem for the pool contents $\vec{C}$.
   - It adds the components that are needed to solve the additional initial value problem for the ^{14}C contents of the pools $^{14}\vec{C}$.
2.
   - It provides the single argument for all the functions that are available for an argument of class `Model`.
   - and for additional functions that are available to compute various results from the solution of the additional initial value problem for ^{14}C. See subsection `Methods` and the examples.)

Details

The original initial value problem for $\vec{C}$ as decribed in the docomentation of the superclass `Model` was given by:

- $\frac{d\mathbf{C}(t)}{dt} = \mathbf{I}(t) + \mathbf{A}(t)\mathbf{C}(t)$
- the initial Values $\vec{C}_0 = \vec{C}(t_0)$
- for the times $\{t_0, ...t_m\}$.

The additional initial value problem for ^{14}C is represented by additional parameters:

- a second ordinary differential equation:

$$\frac{d^{14}\mathbf{C}(t)}{dt} = F(t)\mathbf{I}(t) + \mathbf{A}(t)^{14}\mathbf{C}(t) - k_{14}\ {}^{14}\mathbf{C}(t)$$

  with initial values $^{14}\mathbf{C}_0 = F_0\mathbf{C}_0$ with:
- the time dependent ^{14}C fraction $F(t)$,
- the constant ^{14}C fraction of the initial pool contents F_0,
- the ^{14}C decay rate k_{14}.

In an object of class `Model_14` the components are represented as follows:

- The time-dependent matrix valued function $\vec{A}(t)$ is represented by an object of a subclass of `DecompOp` (for decomposition operator). Such objects can be created in different ways from functions, matrices or data. (see the subclasses of `DecompOp` and especially their `Constructors` sections. and the `examples` section of this help page.

- The vector-valued time-dependent function $\vec{I}(t)$ is in SoilR represented by an object of a subclass of class `InFlux`. Such objects can also be created from functions, constant vectors and data. (see the subclasses of `InFlux` and especially their `Constructors` sections.
- The initial values for $\mathbf{C}_0$ are represented by a numeric vector
- The value for the $^{14}\mathbf{C}$ fraction of the initial $\mathbf{C}$ is represented as an object of class ConstFc which is a subclass of `Fc` representing the ^{14}C fraction F and its unit. (Either "Delta14C" or "afn" for Absolute Fraction Normal)
- The value for the $^{14}\mathbf{C}$ fraction of the input $\mathbf{I}(t)$ is also represented as an object of a subclass of `Fc`. It can be time dependent or constant.

Methods

Exported methods directly defined for class Model_14:

getC14 `signature(object = "Model_14")`: ... `getC14,Model_14-method`

getF14 `signature(object = "Model_14")`: ... `getF14,Model_14-method`

getF14C `signature(object = "Model_14")`: ... `getF14C,Model_14-method`

getF14R `signature(object = "Model_14")`: ... `getF14R,Model_14-method`

getReleaseFlux14 `signature(object = "Model_14")`: ... `getReleaseFlux14,Model_14-method`

Methods inherited from superclasses:

from class Model:

[`signature(x = "Model", i = "character", j = "missing", drop = "missing")`: ... `[,Model,character,missing,missing-method`

getAccumulatedRelease `signature(object = "Model")`: ... `getAccumulatedRelease,Model-metho`

getC `signature(object = "Model")`: ... `getC,Model-method`

getDecompOp `signature(object = "Model")`: ... `getDecompOp,Model-method`

getReleaseFlux `signature(object = "Model")`: ... `getReleaseFlux,Model-method`

getTimes `signature(object = "Model")`: ... `getTimes,Model-method`

Superclasses

`Model-class`

Constructors

`Model_14`

NlModel-class *NlModel*

Description

serves as a fence to the interface of SoilR functions. So that later implementations can differ

Methods

Exported methods directly defined for class NlModel:

$ `signature(x = "NlModel")`: ... `$,NlModel-method`

[`signature(x = "NlModel", i = "character", j = "ANY", drop = "ANY")`: ... `[,NlModel,character,ANY,ANY-method`

getC `signature(object = "NlModel")`: ... `getC,NlModel-method`

getDecompOp `signature(object = "NlModel")`: ... `getDecompOp,NlModel-method`

getReleaseFlux `signature(object = "NlModel")`: ... `getReleaseFlux,NlModel-method`

getTimes `signature(object = "NlModel")`: ... `getTimes,NlModel-method`

OnepModel *Implementation of a one pool model*

Description

This function creates a model for one pool. It is a wrapper for the more general function `GeneralModel`.

Usage

```
OnepModel(t,
k,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`k`	A scalar with the decomposition rate of the pool.
`C0`	A scalar containing the initial amount of carbon in the pool.
`In`	A scalar or a data.frame object specifying the amount of litter inputs by time.
`xi`	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE forces the constructor to create the model even if it is invalid

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
k=0.8
C0=100
In = 30


Ex=OnepModel(t,k,C0,In)
Ct=getC(Ex)
Rt=getReleaseFlux(Ex)
Rc=getAccumulatedRelease(Ex)

plot(
t,
Ct,
type="l",
ylab="Carbon stocks (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2
)

plot(
t,
Rt,
type="l",
ylab="Carbon released (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2
)

plot(
t,
Rc,
type="l",
ylab="Cummulative carbon released (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2
)
```

OnepModel14 *Implementation of a one-pool C14 model*

Description

This function creates a model for one pool. It is a wrapper for the more general function GeneralModel_14.

Usage

```
OnepModel14(t,
k,
C0,
F0_Delta14C,
In,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.
k	A scalar with the decomposition rate of the pool.
C0	A scalar containing the initial amount of carbon in the pool.
F0_Delta14C	A scalar containing the initial amount of the radiocarbon fraction in the pool in Delta_14C format.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object consisting of a function describing the fraction of C_14 in per mille. The first column will be assumed to contain the times.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A (positive) scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
years=seq(1901,2009,by=0.5)
LitterInput=700

Ex=OnepModel14(t=years,k=1/10,C0=500, F0=0,In=LitterInput, inputFc=C14Atm_NH)
C14t=getF14(Ex)

plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
legend(
"topright",
c("Delta 14C Atmosphere", "Delta 14C in SOM"),
lty=c(1,1),
col=c(1,4),
lwd=c(1,1),
bty="n"
)
```

ParallelModel *models for unconnected pools*

Description

This function creates a (linear) numerical model for n independent (parallel) pools that can be queried afterwards. It is used by the convinient wrapper functions `TwopParallelModel` and `ThreepParallelModel` but can also be used independently.

Usage

```
ParallelModel(times,
coeffs_tm,
startvalues,
inputrates,
solverfunc=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`times`	A vector containing the points in time where the solution is sought.
`coeffs_tm`	A TimeMap object consisting of a vector valued function containing the decay rates for the n pools as function of time and the time range where this function is valid. The length of the vector is equal to the number of pools.
`startvalues`	A vector containing the initial amount of carbon for the n pools. «The length of this vector is equal to the number of pools and thus equal to the length of k. This is checked by the function.
`inputrates`	An object consisting of a vector valued function describing the inputs to the pools as funtions of time `TimeMap.new`
`solverfunc`	The function used to actually solve the ODE system. This can be `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE forces the constructor to create the model even if it is invalid

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
k=TimeMap(
function(times){c(-0.5,-0.2,-0.3)},
t_start,
t_end
)
c0=c(1, 2, 3)
#constant inputrates
inputrates=BoundInFlux(
function(t){matrix(nrow=3,ncol=1,c(1,1,1))},
t_start,
t_end
)
mod=ParallelModel(t,k,c0,inputrates)
Y=getC(mod)
lt1=1 ;lt2=2 ;lt3=3
col1=1; col2=2; col3=3
plot(t,Y[,1],type="l",lty=lt1,col=col1,
ylab="C stocks",xlab="Time")
lines(t,Y[,2],type="l",lty=lt2,col=col2)
lines(t,Y[,3],type="l",lty=lt3,col=col3)
legend(
"topleft",
c("C in pool 1",
"C in 2",
"C in pool 3"
),
lty=c(lt1,lt2,lt3),
col=c(col1,col2,col3)
)
Y=getAccumulatedRelease(mod)
plot(t,Y[,1],type="l",lty=lt1,col=col1,ylab="C release",xlab="Time")
lines(t,Y[,2],lt2,type="l",lty=lt2,col=col2)
lines(t,Y[,3],type="l",lty=lt3,col=col3)
legend("topright",c("R1","R2","R3"),lty=c(lt1,lt2,lt3),col=c(col1,col2,col3))
```

`plotC14Pool` *Plots the output of* `getF14` *for each pool over time*

Description

This function produces a plot with the Delta14C in the atmosphere and the Delta14C of each pool obtained after a call to `getF14`.

Usage

```
plotC14Pool(t,
mat,
inputFc,
```

```
col,
...)
```

Arguments

t	A vector containing the time points for plotting.
mat	A matrix object obtained after a call to getF14
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
col	A color palette specifying color lines for each pool (columns of mat).
...	Other arguments passed to plot.

---

plotCPool — *Plots the output of getC or getReleaseFlux for each pool over time*

Description

This function produces a plot with the C content or released C for each pool over time. Needs as input a matrix obtained after a call to getC or getReleaseFlux.

Usage

```
plotCPool(t,
mat,
col,
...)
```

Arguments

t	A vector containing the time points for plotting.
mat	A matrix object obtained after a call to getC or getReleaseFlux
col	A color palette specifying color lines for each pool (columns of mat).
...	Other arguments passed to link{plot}.

predefinedModels *PREDEFINED MODELS*

Description

GaudinskiModel14
ICBMModel
OnepModel
OnepModel14
RothCModel
ThreepFeedbackModel
ThreepFeedbackModel14
ThreepParallelModel
ThreepParallelModel14
ThreepSeriesModel
ThreepSeriesModel14
TwopFeedbackModel
TwopFeedbackModel14
TwopParallelModel
TwopParallelModel14
TwopMMmodel
ThreepairMMmodel
TwopSeriesModel
TwopSeriesModel14
YassoModel
bacwaveModel
Yasso07Model
SeriesLinearModel
SeriesLinearModel14
CenturyModel

RespirationCoefficients *helper function to compute respiration coefficients*

Description

This function computes the respiration coefficients as function of time for all pools according to the given matrix A

Usage

```
RespirationCoefficients(A)
```

Arguments

A	A matrix valued function representing the model.

Value

A vector valued function of time containing the respiration coefficients for all pools.

RothCModel *Implementation of the RothCModel*

Description

This function implements the RothC model of Jenkinson et al. It is a wrapper for the more general function GeneralModel.

Usage

```
RothCModel(t,
ks=c(k.DPM = 10, k.RPM = 0.3, k.BIO = 0.66, k.HUM = 0.02, k.IOM = 0),
C0=c(0, 0, 0, 0, 2.7),
In=1.7,
FYM=0,
DR=1.44,
clay=23.4,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
ks	A vector of lenght 5 containing the values of the decomposition rates for the different pools
C0	A vector of length 5 containing the initial amount of carbon for the 5 pools.
In	A scalar or data.frame object specifying the amount of litter inputs by time.
FYM	A scalar or data.frame object specifying the amount of Farm Yard Manure inputs by time.
DR	A scalar representing the ratio of decomposable plant material to resistant plant material (DPM/RPM).
clay	Percent clay in mineral soil.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Jenkinson, D. S., S. P. S. Andrew, J. M. Lynch, M. J. Goss, and P. B. Tinker. 1990. The Turnover of Organic Carbon and Nitrogen in Soil. Philosophical Transactions: Biological Sciences 329:361-368. Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
t=0:500
Ex=RothCModel(t)
Ct=getC(Ex)
Rt=getReleaseFlux(Ex)

matplot(t,Ct,type="l",col=1:5, ylim=c(0,25),
ylab=expression(paste("Carbon stores (Mg C ", ha^-1,")")),
xlab="Time (years)", lty=1)
lines(t,rowSums(Ct),lwd=2)
legend("topleft",
c("Pool 1, DPM",
"Pool 2, RPM",
"Pool 3, BIO",
"Pool 4, HUM",
"Pool 5, IOM",
"Total Carbon"),
lty=1,
lwd=c(rep(1,5),2),
col=c(1:5,1),
bty="n"
)
```

SeriesLinearModel *General m-pool linear model with series structure*

Description

This function creates a model for m number of pools connected in series. It is a wrapper for the more general function `GeneralModel`.

Usage

```
SeriesLinearModel(t,
m.pools,
ki,
Tij,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
m.pools	An integer with the total number of pools in the model.
ki	A vector of lenght m containing the values of the decomposition rates for each pool i.
Tij	A vector of length m-1 with the transfer coefficients from pool j to pool i. The value of these coefficients must be in the range [0, 1].
C0	A vector of length m containing the initial amount of carbon for the m pools.
In	A scalar or data.frame object specifying the amount of litter inputs by time.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
#A five-pool model
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
ks=c(k1=0.8,k2=0.4,k3=0.2, k4=0.1,k5=0.05)
Ts=c(0.5,0.2,0.2,0.1)
C0=c(C10=100,C20=150, C30=50, C40=50, C50=10)
In = 50
#
Ex1=SeriesLinearModel(t=t,m.pools=5,ki=ks,Tij=Ts,C0=C0,In=In,xi=fT.Q10(15))
Ct=getC(Ex1)
#
matplot(t,Ct,type="l",col=2:6,lty=1,ylim=c(0,sum(C0)))
lines(t,rowSums(Ct),lwd=2)
legend("topright",c("Total C","C in pool 1", "C in pool 2","C in pool 3",
"C in pool 4","C in pool 5"),
lty=1,col=1:6,lwd=c(2,rep(1,5)),bty="n")
```

SeriesLinearModel14 *General m-pool linear C14 model with series structure*

Description

This function creates a radiocarbon model for m number of pools connected in series. It is a wrapper for the more general function GeneralModel_14.

Usage

```
SeriesLinearModel14(t,
m.pools,
ki,
Tij,
C0,
F0_Delta14C,
In,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
m.pools	An integer with the total number of pools in the model.
ki	A vector of lenght m containing the values of the decomposition rates for each pool i.
Tij	A vector of length m-1 with the transfer coefficients from pool j to pool i. The value of these coefficients must be in the range [0, 1].
C0	A vector of length m containing the initial amount of carbon for the m pools.
F0_Delta14C	A vector of length m containig the initial amount of the radiocarbon fraction for the m pools.
In	A scalar or data.frame object specifying the amount of litter inputs by time.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2014. Modeling radiocarbon dynamics in soils: SoilR version 1.1. Geoscientific Model Development 7, 1919-1931.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
years=seq(1901,2009,by=0.5)
LitterInput=700

Ex=SeriesLinearModel14(
t=years,ki=c(k1=1/2.8, k2=1/35, k3=1/100), m.pools=3,
C0=c(200,5000,500), F0_Delta14C=c(0,0,0),
In=LitterInput, Tij=c(0.5, 0.1),inputFc=C14Atm_NH
)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)

par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",
ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
lines(years, C14t[,3],col=4,lwd=3)
legend(
"topright",
c("Delta 14C Atmosphere", "Delta 14C pool 1", "Delta 14C pool 2", "Delta 14C pool 3"),
lty=rep(1,4),col=c(1,4,4,4),lwd=c(1,1,2,3),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

SoilR.F0.new *deprecated function that used to create an object of class SoilR.F0*

Description

The function internally calls the constructor of the replacement class ConstFc-class.

Usage

```
SoilR.F0.new(values=c(0),
format=Delta14C)
```

Arguments

values	a numeric vector
format	a character string describing the format e.g. "Delta14C"

Value

An object of class ConstFc-class that contains data and a format description that can later be used to convert the data into other formats if the conversion is implemented.

---

systemAge *System and pool age for compartment models*

Description

Computes the density distribution and mean for the system and pool ages of a SoilR model or a matrix representation of a compartmental model

Usage

```
systemAge(A,
u,
a=seq(0, 100),
q=c(0.05, 0.5, 0.95))
```

Arguments

A	A compartmental linear square matrix with cycling rates in the diagonal and transfer rates in the off-diagonal.
u	A one-column matrix defining the amount of inputs per compartment.
a	A sequence of ages to calculate density functions
q	A vector of probabilities to calculate quantiles of the system age distribution

Value

A list with 5 objects: mean system age, system age distribution, quantiles of system age distribution, mean pool-age, and pool-age distribution.

See Also

transitTime

`ThreepairMMmodel` *Implementation of a 6-pool Michaelis-Menten model*

Description

This function implements a 6-pool Michaelis-Meneten model with pairs of microbial biomass and substrate pools.

Usage

```
ThreepairMMmodel(t,
ks,
kb,
Km,
r,
Af=1,
ADD,
ival)
```

Arguments

`t`	vector of times to calculate a solution.
`ks`	a vector of length 3 representing SOM decomposition rate (m3 d-1 (gCB)-1)
`kb`	a vector of length 3 representing microbial decay rate (d-1)
`Km`	a vector of length 3 representing the Michaelis constant (g m-3)
`r`	a vector of length 3 representing the respired carbon fraction (unitless)
`Af`	a scalar representing the Activity factor; i.e. a temperature and moisture modifier (unitless)
`ADD`	a vector of length 3 representing the annual C input to the soil (g m-3 d-1)
`ival`	a vector of length 6 with the initial values of the SOM pools and the microbial biomass pools (g m-3)

Value

An object of class NlModel that can be further queried.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
days=seq(0,1000)
#Run the model with default parameter values
MMmodel=ThreepairMMmodel(t=days,ival=rep(c(100,10),3),ks=c(0.1,0.05,0.01),
kb=c(0.005,0.001,0.0005),Km=c(100,150,200),r=c(0.9,0.9,0.9),
ADD=c(3,1,0.5))
Cpools=getC(MMmodel)
#Time solution
matplot(days,Cpools,type="l",ylab="Concentrations",xlab="Days",lty=rep(1:2,3),
```

```
ylim=c(0,max(Cpools)*1.2),col=rep(1:3,each=2),
main="Multi-substrate microbial model")
legend("topright",c("Substrate 1", "Microbial biomass 1",
"Substrate 2", "Microbial biomass 2",
"Substrate 3", "Microbial biomass 3"),
lty=rep(1:2,3),col=rep(1:3,each=2),
bty="n")
#State-space diagram
plot(Cpools[,2],Cpools[,1],type="l",ylab="Substrate",xlab="Microbial biomass")
lines(Cpools[,4],Cpools[,3],col=2)
lines(Cpools[,6],Cpools[,5],col=3)
legend("topright",c("Substrate-Enzyme pair 1","Substrate-Enzyme pair 2",
"Substrate-Enzyme pair 3"),col=1:3,lty=1,bty="n")
#Microbial biomass over time
plot(days,Cpools[,2],type="l",col=2,xlab="Days",ylab="Microbial biomass")
```

`ThreepFeedbackModel`

Implementation of a three pool model with feedback structure

Description

This function creates a model for three pools connected with feedback. It is a wrapper for the more general function `GeneralModel`.

Usage

```
ThreepFeedbackModel(t,
ks,
a21,
a12,
a32,
a23,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of lenght 3 containing the values of the decomposition rates for pools 1, 2, and 3.
`a21`	A scalar with the value of the transfer rate from pool 1 to pool 2.
`a12`	A scalar with the value of the transfer rate from pool 2 to pool 1.
`a32`	A scalar with the value of the transfer rate from pool 2 to pool 3.
`a23`	A scalar with the value of the transfer rate from pool 3 to pool 2.
`C0`	A vector containing the initial concentrations for the 3 pools. The length of this vector is 3

In	A data.frame object specifying the amount of litter inputs by time.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE forces the constructor to create the model even if it is invalid

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
ks=c(k1=0.8,k2=0.4,k3=0.2)
C0=c(C10=100,C20=150, C30=50)
In = 60

Temp=rnorm(t,15,1)
TempEffect=data.frame(t,fT.Daycent1(Temp))

Ex1=ThreepFeedbackModel(t=t,ks=ks,a21=0.5,a12=0.1,a32=0.2,a23=0.1,C0=C0,In=In,xi=TempEffe
Ct=getC(Ex1)
Rt=getReleaseFlux(Ex1)

plot(
t,
rowSums(Ct),
type="l",
ylab="Carbon stocks (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2,
ylim=c(0,sum(Ct[51,]))
)
lines(t,Ct[,1],col=2)
lines(t,Ct[,2],col=4)
lines(t,Ct[,3],col=3)
legend(
"topleft",
c("Total C","C in pool 1", "C in pool 2","C in pool 3"),
lty=c(1,1,1,1),
col=c(1,2,4,3),
lwd=c(2,1,1,1),
bty="n"
)
```

```
plot(
t,
rowSums(Rt),
type="l",
ylab="Carbon released (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2,
ylim=c(0,sum(Rt[51,]))
)
lines(t,Rt[,1],col=2)
lines(t,Rt[,2],col=4)
lines(t,Rt[,3],col=3)
legend(
"topleft",
c("Total C release",
"C release from pool 1",
"C release from pool 2",
"C release from pool 3"),
lty=c(1,1,1,1),
col=c(1,2,4,3),
lwd=c(2,1,1,1),
bty="n"
)

Inr=data.frame(t,Random.inputs=rnorm(length(t),50,10))
plot(Inr,type="l")

Ex2=ThreepFeedbackModel(t=t,ks=ks,a21=0.5,a12=0.1,a32=0.2,a23=0.1,C0=C0,In=Inr)
Ctr=getC(Ex2)
Rtr=getReleaseFlux(Ex2)

plot(
t,
rowSums(Ctr),
type="l",
ylab="Carbon stocks (arbitrary units)",
xlab="Time (arbitrary units)",
lwd=2,
ylim=c(0,sum(Ctr[51,]))
)
lines(t,Ctr[,1],col=2)
lines(t,Ctr[,2],col=4)
lines(t,Ctr[,3],col=3)
legend("topright",c("Total C","C in pool 1", "C in pool 2","C in pool 3"),
lty=c(1,1,1,1),col=c(1,2,4,3),lwd=c(2,1,1,1),bty="n")

plot(t,rowSums(Rtr),type="l",ylab="Carbon released (arbitrary units)",
xlab="Time (arbitrary units)",lwd=2,ylim=c(0,sum(Rtr[51,])))
lines(t,Rtr[,1],col=2)
lines(t,Rtr[,2],col=4)
lines(t,Rtr[,3],col=3)
legend(
"topright",
c("Total C release",
"C release from pool 1",
"C release from pool 2",
"C release from pool 3"
```

```
),
lty=c(1,1,1,1),
col=c(1,2,4,3),
lwd=c(2,1,1,1),
bty="n")
```

`ThreepFeedbackModel14`

Implementation of a three-pool C14 model with feedback structure

---

Description

This function creates a model for three pools connected with feedback. It is a wrapper for the more general function `GeneralModel_14` that can handle an arbitrary number of pools with arbitrary connections. `GeneralModel_14` can also handle input data in different formats, while this function requires its input as Delta14C. Look at it as an example how to use the more powerful tool `GeneralModel_14` or as a shortcut for a standard task!

Usage

```
ThreepFeedbackModel14(t,
ks,
C0,
F0_Delta14C,
In,
a21,
a12,
a32,
a23,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset `C14Atm_NH` is 1900-2010.
`ks`	A vector of length 3 containing the decomposition rates for the 3 pools.
`C0`	A vector of length 3 containing the initial amount of carbon for the 3 pools.
`F0_Delta14C`	A vector of length 3 containing the initial fraction of radiocarbon for the 3 pools in Delta14C format. The format will be assumed to be Delta14C, so please take care that it is.
`In`	A scalar or a data.frame object specifying the amount of litter inputs by time.
`a21`	A scalar with the value of the transfer rate from pool 1 to pool 2.
`a12`	A scalar with the value of the transfer rate from pool 2 to pool 1.

a32	A scalar with the value of the transfer rate from pool 2 to pool 3.
a23	A scalar with the value of the transfer rate from pool 3 to pool 2.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE forces the constructor to create the model even if it is invalid. This is sometimes useful when SoilR is used by externel packages for parameter estimation.

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
#years=seq(1901,2009,by=0.5)
years=seq(1904,2009,by=0.5)
LitterInput=100
k1=1/2; k2=1/10; k3=1/50
a21=0.9*k1
a12=0.4*k2
a32=0.4*k2
a23=0.7*k3

Feedback=ThreepFeedbackModel14(
t=years,
ks=c(k1=k1, k2=k2, k3=k3),
C0=c(100,500,1000),
F0_Delta14C=c(0,0,0),
In=LitterInput,
a21=a21,
a12=a12,
a32=a32,
a23=a23,
inputFc=C14Atm_NH
)
F.R14m=getF14R(Feedback)
F.C14m=getF14C(Feedback)
F.C14t=getF14(Feedback)

Series=ThreepSeriesModel14(
t=years,
ks=c(k1=k1, k2=k2, k3=k3),
C0=c(100,500,1000),
F0_Delta14C=c(0,0,0),
```

```
In=LitterInput,
a21=a21,
a32=a32,
inputFc=C14Atm_NH
)
S.R14m=getF14R(Series)
S.C14m=getF14C(Series)
S.C14t=getF14(Series)

Parallel=ThreepParallelModel14(
t=years,
ks=c(k1=k1, k2=k2, k3=k3),
C0=c(100,500,1000),
F0_Delta14C=c(0,0,0),
In=LitterInput,
gam1=0.6,
gam2=0.2,
inputFc=C14Atm_NH,
lag=2
)
P.R14m=getF14R(Parallel)
P.C14m=getF14C(Parallel)
P.C14t=getF14(Parallel)

par(mfrow=c(3,2))
plot(
C14Atm_NH,
type="l",
xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),
xlim=c(1940,2010)
)
lines(years, P.C14t[,1], col=4)
lines(years, P.C14t[,2],col=4,lwd=2)
lines(years, P.C14t[,3],col=4,lwd=3)
legend(
"topright",
c("Atmosphere", "Pool 1", "Pool 2", "Pool 3"),
lty=rep(1,4),
col=c(1,4,4,4),
lwd=c(1,1,2,3),
bty="n"
)

plot(C14Atm_NH,type="l",xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),xlim=c(1940,2010))
lines(years,P.C14m,col=4)
lines(years,P.R14m,col=2)
legend("topright",c("Atmosphere","Bulk SOM", "Respired C"),
lty=c(1,1,1), col=c(1,4,2),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),xlim=c(1940,2010))
lines(years, S.C14t[,1], col=4)
lines(years, S.C14t[,2],col=4,lwd=2)
lines(years, S.C14t[,3],col=4,lwd=3)
legend("topright",c("Atmosphere", "Pool 1", "Pool 2", "Pool 3"),
```

```
lty=rep(1,4),col=c(1,4,4,4),lwd=c(1,1,2,3),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),xlim=c(1940,2010))
lines(years,S.C14m,col=4)
lines(years,S.R14m,col=2)
legend("topright",c("Atmosphere","Bulk SOM", "Respired C"),
lty=c(1,1,1), col=c(1,4,2),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),xlim=c(1940,2010))
lines(years, F.C14t[,1], col=4)
lines(years, F.C14t[,2],col=4,lwd=2)
lines(years, F.C14t[,3],col=4,lwd=3)
legend("topright",c("Atmosphere", "Pool 1", "Pool 2", "Pool 3"),
lty=rep(1,4),col=c(1,4,4,4),lwd=c(1,1,2,3),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",
ylab=expression(paste(Delta^14,"C ","(\u2030)")),xlim=c(1940,2010))
lines(years,F.C14m,col=4)
lines(years,F.R14m,col=2)
legend("topright",c("Atmosphere","Bulk SOM", "Respired C"),
lty=c(1,1,1), col=c(1,4,2),bty="n")


par(mfrow=c(1,1))
```

`ThreepParallelModel`

Implementation of a three pool model with parallel structure

Description

The function creates a model for three independent (parallel) pools. It is a wrapper for the more general function `ParallelModel` that can handle an arbitrary number of pools.

Usage

```
ThreepParallelModel(t,
ks,
C0,
In,
gam1,
gam2,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of length 3 containing the decomposition rates for the 3 pools.
`C0`	A vector of length 3 containing the initial amount of carbon for the 3 pools.

In	A scalar or a data.frame object specifying the amount of litter inputs by time.
gam1	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 1.
gam2	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 2.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	Logical that forces the Model to be created even if the chect suggest problems.

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)

Ex=ThreepParallelModel(t,ks=c(k1=0.5,k2=0.2,k3=0.1),
C0=c(c10=100, c20=150,c30=50),In=20,gam1=0.7,gam2=0.1,xi=0.5)
Ct=getC(Ex)

plot(t,rowSums(Ct),type="l",lwd=2,
ylab="Carbon stocks (arbitrary units)",xlab="Time",ylim=c(0,sum(Ct[1,])))
lines(t,Ct[,1],col=2)
lines(t,Ct[,2],col=4)
lines(t,Ct[,3],col=3)
legend("topright",c("Total C","C in pool 1", "C in pool 2","C in pool 3"),
lty=c(1,1,1,1),col=c(1,2,4,3),lwd=c(2,1,1,1),bty="n")

Rt=getReleaseFlux(Ex)
plot(t,rowSums(Rt),type="l",ylab="Carbon released (arbitrary units)",
xlab="Time",lwd=2,ylim=c(0,sum(Rt[1,])))
lines(t,Rt[,1],col=2)
lines(t,Rt[,2],col=4)
lines(t,Rt[,3],col=3)
legend("topright",c("Total C release","C release from pool 1",
"C release from pool 2","C release from pool 3"),
lty=c(1,1,1,1),col=c(1,2,4,3),lwd=c(2,1,1,1),bty="n")
```

ThreepParallelModel14 *Implementation of a three-pool C14 model with parallel structure*

Description

This function creates a model for two independent (parallel) pools. It is a wrapper for the more general function GeneralModel_14 that can handle an arbitrary number of pools.

Usage

```
ThreepParallelModel14(t,
ks,
C0,
F0_Delta14C,
In,
gam1,
gam2,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.
ks	A vector of length 3 containing the decomposition rates for the 3 pools.
C0	A vector of length 3 containing the initial amount of carbon for the 3 pools.
F0_Delta14C	A vector of length 3 containing the initial amount of the radiocarbon fraction for the 3 pools in Delta14C values in per mil.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
gam1	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 1.
gam2	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 2.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive scalar representing a time lag for radiocarbon to enter the system.

solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
years=seq(1903,2009,by=0.5) # note that we
LitterInput=700

Ex=ThreepParallelModel14(
t=years,
ks=c(k1=1/2.8, k2=1/35, k3=1/100),
C0=c(200,5000,500),
F0_Delta14C=c(0,0,0),
In=LitterInput,
gam1=0.7,
gam2=0.1,
inputFc=C14Atm_NH,
lag=2
)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)

par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
lines(years, C14t[,3],col=4,lwd=3)
legend(
"topright",
c(
"Delta 14C Atmosphere",
"Delta 14C pool 1",
"Delta 14C pool 2",
"Delta 14C pool 3"
),
lty=rep(1,4),
col=c(1,4,4,4),
lwd=c(1,1,2,3),
bty="n"
)

plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

`ThreepSeriesModel` *Implementation of a three pool model with series structure*

Description

This function creates a model for three pools connected in series. It is a wrapper for the more general function `GeneralModel`.

Usage

```
ThreepSeriesModel(t,
ks,
a21,
a32,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of lenght 3 containing the values of the decomposition rates for pools 1, 2, and 3.
`a21`	A scalar with the value of the transfer rate from pool 1 to pool 2.
`a32`	A scalar with the value of the transfer rate from pool 2 to pool 3.
`C0`	A vector of length 3 containing the initial amount of carbon for the 3 pools.
`In`	A scalar or data.frame object specifying the amount of litter inputs by time.
`xi`	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
ks=c(k1=0.8,k2=0.4,k3=0.2)
C0=c(C10=100,C20=150, C30=50)
In = 50

Ex1=ThreepSeriesModel(t=t,ks=ks,a21=0.5,a32=0.2,C0=C0,In=In,xi=fT.Q10(15))
Ct=getC(Ex1)
Rt=getReleaseFlux(Ex1)

plot(t,rowSums(Ct),type="l",ylab="Carbon stocks (arbitrary units)",
xlab="Time (arbitrary units)",lwd=2,ylim=c(0,sum(Ct[1,])))
lines(t,Ct[,1],col=2)
lines(t,Ct[,2],col=4)
lines(t,Ct[,3],col=3)
legend("topright",c("Total C","C in pool 1", "C in pool 2","C in pool 3"),
lty=c(1,1,1,1),col=c(1,2,4,3),lwd=c(2,1,1,1),bty="n")
```

ThreepSeriesModel14

Implementation of a three-pool C14 model with series structure

---

Description

This function creates a model for three pools connected in series. It is a wrapper for the more general function `GeneralModel_14` that can handle an arbitrary number of pools.

Usage

```
ThreepSeriesModel14(t,
ks,
C0,
F0_Delta14C,
In,
a21,
a32,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset `C14Atm_NH` is 1900-2010.

ks	A vector of length 3 containing the decomposition rates for the 3 pools.
C0	A vector of length 3 containing the initial amount of carbon for the 3 pools.
F0_Delta14C	A vector of length 3 containig the initial amount of the radiocarbon fraction for the 3 pools.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
a21	A scalar with the value of the transfer rate from pool 1 to pool 2.
a32	A scalar with the value of the transfer rate from pool 2 to pool 3 as Delta14C values in per mil.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
years=seq(1901,2009,by=0.5)
LitterInput=700

Ex=ThreepSeriesModel14(
t=years,ks=c(k1=1/2.8, k2=1/35, k3=1/100),
C0=c(200,5000,500), F0_Delta14C=c(0,0,0),
In=LitterInput, a21=0.1, a32=0.01,inputFc=C14Atm_NH
)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)

par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",
ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
lines(years, C14t[,3],col=4,lwd=3)
legend(
"topright",
```

```
c("Delta 14C Atmosphere", "Delta 14C pool 1", "Delta 14C pool 2", "Delta 14C pool 3"),
lty=rep(1,4),col=c(1,4,4,4),lwd=c(1,1,2,3),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

`TimeMap` *TimeMap S4 generic*

Description

no Description

Usage

```
TimeMap(map,
starttime,
endtime,
times,
data,
lag=0,
interpolation=splinefun)
```

Arguments

`map`	see the method arguments for details
`starttime`	see the method arguments for details
`endtime`	see the method arguments for details
`times`	see the method arguments for details
`data`	see the method arguments for details
`lag`	a time delay
`interpolation`	the interpolating function

Methods

`TimeMap,TimeMap,ANY,ANY,ANY,ANY-method`
`TimeMap,data.frame,missing,missing,missing,missing-method`
`TimeMap,function,numeric,numeric,missing,missing-method`
`TimeMap,list,missing,missing,missing,missing-method`
`TimeMap,missing,missing,missing,numeric,array-method`
`TimeMap,missing,missing,missing,numeric,list-method`
`TimeMap,missing,missing,missing,numeric,matrix-method`
`TimeMap,missing,missing,missing,numeric,numeric-method`

`TimeMap,data.frame,missing,missing,missing,missing-method`
TimeMap,data.frame,missing,missing,missing,missing-method *constructor*

Description

create a TimeMap object by interpolating the data.frame

Usage

```
## S4 method for signature 'data.frame,missing,missing,missing,missing'
TimeMap(map,
lag=0,
interpolation=splinefun)
```

Arguments

`map`	: of class data.frame, a data frame containing two columns
`lag`	: a time delay
`interpolation`	: the interpolating function

`TimeMap,function,numeric,numeric,missing,missing-method`
TimeMap,function,numeric,numeric,missing,missing-method *constructor*

Description

create a TimeMap object from the function definition and the time interval

Usage

```
## S4 method for signature 'function,numeric,numeric,missing,missing'
TimeMap(map,
starttime,
endtime)
```

Arguments

`map`	: of class function
`starttime`	: of class numeric
`endtime`	: of class numeric

`TimeMap,list,missing,missing,missing,missing-method`
TimeMap,list,missing,missing,missing,missing-method Create a TimeMap from a nested list

---

Description

The method creates an instance of `TimeMap-class` from a list that contains data and a vector of times referring to it.

Usage

```
## S4 method for signature 'list,missing,missing,missing,missing'
TimeMap(map,
lag=0,
interpolation=splinefun)
```

Arguments

`map`	: of class list
`lag`	: either a scalar or an element of the same shape as the elements of the data entry that refer to one time step
`interpolation`	: the interpolation method to be used

Details

The list must have two entries If the entries are not named, the first one is supposed to be a numeric vector of `times` and the second to contain the data referring to those times. The `data` entry of the list can itself be a list with the same length as the `times` entry or an array whose last dimension is equal to the length of the `times` entry. If the `data` entry is a list the elements must be `vectors`,`matrices` or `arrays`.

`TimeMap,missing,missing,missing,numeric,array-method`
TimeMap,missing,missing,missing,numeric,array-method Create a TimeMap from times and array

---

Description

The method creates an instance of `TimeMap-class` from a vector of times and an array whose last dimension ### is referring to it.

Usage

```
## S4 method for signature 'missing,missing,missing,numeric,array'
TimeMap(times,
data,
lag=0,
interpolation=splinefun)
```

Arguments

`times`	: of class numeric
`data`	: of class array
`lag`	: Either a scalar or an object of the same size as a slice of the array per timestep. It describes the time lag of the array elements
`interpolation`	: the interpolation method to be used

`TimeMap,missing,missing,missing,numeric,list-method`

TimeMap,missing,missing,missing,numeric,list-method Create a TimeMap from a nested list

---

Description

The method creates an instance of `TimeMap-class` from a vector of times and a list of the same length, containing vectors matrices or arrays

Usage

```
## S4 method for signature 'missing,missing,missing,numeric,list'
TimeMap(times,
data,
lag=0,
interpolation=splinefun)
```

Arguments

`times`	: of class numeric, A vector of times
`data`	: of class list, A list of the same length as times with every list element refering to one time in times
`lag`	: Either a scalar or an object of the same size as the list elements describing the time lag of the data
`interpolation`	: the interpolation method to be used

`TimeMap,missing,missing,missing,numeric,matrix-method`

TimeMap,missing,missing,missing,numeric,matrix-method Create a TimeMap from a nested list

---

Description

The method creates an instance of `TimeMap-class` from a vector of times and an array referring to it.

Usage

```
## S4 method for signature 'missing,missing,missing,numeric,matrix'
TimeMap(times,
data,
lag=0,
interpolation=splinefun)
```

Arguments

`times`	: of class numeric
`data`	: of class matrix, a matrix, every column corresponds to one time step
`lag`	: a scalar or a vector describing how much the data is laging behind the times
`interpolation`	: the interpolation method to be used

`TimeMap,missing,missing,missing,numeric,numeric-method` *TimeMap,missing,missing,missing,numeric,numeric-method Create a TimeMap from a nested list*

Description

The method creates an instance of `TimeMap-class` from a vector of times and an array referring to it.

Usage

```
## S4 method for signature 'missing,missing,missing,numeric,numeric'
TimeMap(times,
data,
lag=0,
interpolation=splinefun)
```

Arguments

`times`	: of class numeric
`data`	: of class numeric
`lag`	: a scalar or a vector,matrix or array describing how much the data lag behind the times. If lag is a vector, matrix or array, the resulting ## function will be vector, matrix or array valued accordingly.
`interpolation`	: the interpolation method to be used

`TimeMap,TimeMap,ANY,ANY,ANY,ANY-method` *TimeMap,TimeMap,ANY,ANY,ANY,ANY-method pass through constructor*

Description

The function just returns its argument. So any function that has to convert one of its argument can just call TimeMap on it even if the argument is allready one.

Usage

```
## S4 method for signature 'TimeMap,ANY,ANY,ANY,ANY'
TimeMap(map)
```

Arguments

`map`	: of class TimeMap, the object that will be returned unchanged

---

`TimeMap-class` *TimeMap S4 class*

Description

This class enhances a time dependent function by information about its domain. The information about the delay is especially usefull for functions that interpolate data. Assume that you are given time series data in two vectors `times`, `values`. You can create an interpolating function with `splinefun` or `approxfun` `f <- splinefun(x=times,y=values)` `f(t)` will yield sensible values for $\min_{t \in times} \leq t \leq max_{t \in times}$. but will produce unreasonable values for any t outside these limits. Unfortunately the interpolating functions produced by `splinefun` or `approxfun` do not retain any information about their domain which makes it possible to accidentatly apply them to times not at all supported by the original data. This would not even cause errors in the code but silently corrupt the results. To help you to keep track of the domains of the many time dependent functions used in SoilR's Models this class `TimeMap` stores the `starttime` and `endtime` values along with the function represented by `map`. SoilR functions that accept time series data will normally convert it to subclasses `TimeMap-class` automatically but you can do it explicitly.

Methods

Exported methods directly defined for class TimeMap:

GeneralDecompOp `signature(object = "TimeMap")`: ... `GeneralDecompOp,TimeMap-method`

GeneralInFlux `signature(object = "TimeMap")`: ... `GeneralInFlux,TimeMap-method`

TimeMap `signature(map = "TimeMap", starttime = "ANY", endtime = "ANY", times = "`
... `TimeMap,TimeMap,ANY,ANY,ANY,ANY-method`

add_plot `signature(x = "TimeMap")`: ... `add_plot,TimeMap-method`

as.character `signature(x = "TimeMap")`: ... `as.character,TimeMap-method`

getFunctionDefinition `signature(object = "TimeMap")`: ... `getFunctionDefinition,TimeMap-me`

getTimeRange `signature(object = "TimeMap")`: ... `getTimeRange,TimeMap-method`

Subclasses

BoundInFlux
TransportDecompositionOperator
BoundLinDecompOp
BoundFc

Constructors

TimeMap
Please also look at constructors of non virtual subclasses: BoundInFlux, , BoundLinDecompOp, BoundFc.

TimeMap.from.Dataframe

TimeMap.from.Dataframe

Description

This function is a deprecated constructor of the class TimeMap.

Usage

```
TimeMap.from.Dataframe(dframe,
lag=0,
interpolation=splinefun)
```

Arguments

dframe	A data frame containing exactly two columns: the first one is interpreted as time
lag	a scalar describing the time lag. Positive Values shift the argument of the interpolation function forward in time. (retard its effect)
interpolation	A function that returns a function the default is splinefun. Other possible values are the linear interpolation approxfun or any self made function with the same interface.

Value

An object of class TimeMap that contains the interpolation function and the limits of the time range where the function is valid. Note that the limits change according to the time lag this serves as a saveguard for Model which thus can check that all involved functions of time are actually defined for the times of interest

TimeMap.new *deprecated constructor of the class TimeMap.*

Description

deprecated functions ################## use the generic TimeMap(...) instead

Usage

```
TimeMap.new(t_start,
t_end,
f)
```

Arguments

t_start	A number marking the begin of the time domain where the function is valid
t_end	A number the end of the time domain where the function is valid
f	The time dependent function definition (a function in R's sense)

Value

An object of class TimeMap that can be used to describe models.

transitTime *Transit times for compartment models*

Description

Computes the density distribution and mean for the transit time of a compartmental model

Usage

```
transitTime(A,
u,
a=seq(0, 100),
q=c(0.05, 0.5, 0.95))
```

Arguments

A	A compartmental linear square matrix with cycling rates in the diagonal and transfer rates in the off-diagonal.
u	A one-column matrix defining the amount of inputs per compartment.
a	A sequence of ages to calculate density functions
q	Vector of probabilities to calculate quantiles of the transit time distribution

Value

A list with 3 objects: mean transit time, transit time density distribution, and quantiles.

See Also

systemAge

`TransportDecompositionOperator-class`
TransportDecompositionOperator S4 class

Description

no Description

Methods

Exported methods directly defined for class TransportDecompositionOperator:

getFunctionDefinition `signature(object = "TransportDecompositionOperator"):`
... `getFunctionDefinition,TransportDecompositionOperator-method`

Methods inherited from superclasses:

from class TimeMap:

GeneralDecompOp `signature(object = "TimeMap"):` ... `GeneralDecompOp,TimeMap-method`

GeneralInFlux `signature(object = "TimeMap"):` ... `GeneralInFlux,TimeMap-method`

TimeMap `signature(map = "TimeMap", starttime = "ANY", endtime = "ANY", times = "`
... `TimeMap,TimeMap,ANY,ANY,ANY,ANY-method`

add_plot `signature(x = "TimeMap"):` ... `add_plot,TimeMap-method`

as.character `signature(x = "TimeMap"):` ... `as.character,TimeMap-method`

getFunctionDefinition `signature(object = "TimeMap"):` ... `getFunctionDefinition,TimeMap-me`

getTimeRange `signature(object = "TimeMap"):` ... `getTimeRange,TimeMap-method`

Superclasses

TimeMap-class

`turnoverFit` *Estimation of the turnover time from a soil radiocarbon sample.*

Description

This function finds the best possible value of turnover time from a soil radiocarbon sample assuming a one pool model and annual litter inputs.

Usage

```
turnoverFit(obsC14,
obsyr,
In,
C0=0,
yr0=1900,
Zone=NHZone2,
plot=TRUE,
by=0.5)
```

Arguments

`obsC14`	a scalar with the observed radiocarbon value in Delta14C of the soil sample.
`obsyr`	a scalar with the year in which the soil sample was taken.
`In`	a scalar or data.frame with the annual amount of litter inputs to the soil.
`C0`	a scalar with the initial amount of carbon stored at the begning of the simulation.
`yr0`	The year at which simulations will start.
`Zone`	the hemispheric zone of atmospheric radiocarbon. Possible values are NHZone1: northern hemisphere zone 1, NHZone2: northern hemisphere zone 2, NHZone3: northern hemisphere zone 3, SHZone12: southern hemisphere zones 1 and 2, SHZone3: southern hemisphere zone 3. See `Hua2013` for additional information.
`plot`	logical. Should the function produce a plot?
`by`	numeric. The increment of the sequence of years used in the simulations.

Details

This algorithm takes the observed values and a given amount of litter inputs, runs `OnepModel14`, calculates the squared difference between predictions and observations, and uses `optimize` to find the minimum difference. If the turnover time is relatively short (< 50 yrs), it is safe to assume C0=0 because the soil will reach steady state within the simulation time. However, for longer turnover times it is recommended to use a value of C0 close to the steady state value.

Value

A scalar with the value of the turnover time that minimizes the difference between the prediction of a one pool model and the observed radiocarbon value.

`TwopFeedbackModel` *Implementation of a two pool model with feedback structure*

Description

This function creates a model for two pools connected with feedback. It is a wrapper for the more general function `GeneralModel`.

Usage

```
TwopFeedbackModel(t,
ks,
a21,
a12,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of length 2 with the values of the decomposition rate for pools 1 and 2.
`a21`	A scalar with the value of the transfer rate from pool 1 to pool 2.
`a12`	A scalar with the value of the transfer rate from pool 2 to pool 1.
`C0`	A vector of length 2 containing the initial amount of carbon for the 2 pools.
`In`	A data.frame object specifying the amount of litter inputs by time.
`xi`	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
#This example show the difference between the three types of two-pool models
times=seq(0,20,by=0.1)
ks=c(k1=0.8,k2=0.00605)
C0=c(C10=5,C20=5)

Temp=rnorm(times,15,2)
WC=runif(times,10,20)
TempEffect=data.frame(times,fT=fT.Daycent1(Temp))
MoistEffect=data.frame(times, fW=fW.Daycent2(WC)[2])

Inmean=1
InRand=data.frame(times,Random.inputs=rnorm(length(times),Inmean,0.2))
InSin=data.frame(times,Inmean+0.5*sin(times*pi*2))

Parallel=TwopParallelModel(t=times,ks=ks,C0=C0,In=Inmean,gam=0.9,
xi=(fT.Daycent1(15)*fW.Demeter(15)))
Series=TwopSeriesModel(t=times,ks=ks,a21=0.2*ks[1],C0=C0,In=InSin,
xi=(fT.Daycent1(15)*fW.Demeter(15)))
Feedback=TwopFeedbackModel(t=times,ks=ks,a21=0.2*ks[1],a12=0.5*ks[2],C0=C0,
In=InRand,xi=MoistEffect)

CtP=getC(Parallel)
CtS=getC(Series)
CtF=getC(Feedback)

RtP=getReleaseFlux(Parallel)
RtS=getReleaseFlux(Series)
RtF=getReleaseFlux(Feedback)

par(mfrow=c(2,1),mar=c(4,4,1,1))
plot(times,rowSums(CtP),type="l",ylim=c(0,20),ylab="Carbon stocks (arbitrary units)",xlab
lines(times,rowSums(CtS),col=2)
lines(times,rowSums(CtF),col=3)
legend("topleft",c("Two-pool Parallel","Two-pool Series","Two-pool Feedback"),
lty=c(1,1,1),col=c(1,2,3),bty="n")

plot(times,rowSums(RtP),type="l",ylim=c(0,3),ylab="Carbon release (arbitrary units)", xla
lines(times,rowSums(RtS),col=2)
lines(times,rowSums(RtF),col=3)
par(mfrow=c(1,1))
```

`TwopFeedbackModel14` *Implementation of a two-pool C14 model with feedback structure*

Description

This function creates a model for two pools connected with feedback. It is a wrapper for the more general function `GeneralModel_14` that can handle an arbitrary number of pools.

Usage

```
TwopFeedbackModel14(t,
```

```
ks,
C0,
F0_Delta14C,
In,
a21,
a12,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.
ks	A vector of length 2 containing the decomposition rates for the 2 pools.
C0	A vector of length 2 containing the initial amount of carbon for the 2 pools.
F0_Delta14C	A vector of length 2 containing the initial amount of the radiocarbon fraction for the 2 pools as Delta14C values in per mil.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
a21	A scalar with the value of the transfer rate from pool 1 to pool 2.
a12	A scalar with the value of the transfer rate from pool 2 to pool 1.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive integer representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
years=seq(1901,2009,by=0.5)
LitterInput=700

Ex=TwopFeedbackModel14(t=years,ks=c(k1=1/2.8, k2=1/35),C0=c(200,5000),
F0_Delta14C=c(0,0),In=LitterInput, a21=0.1,a12=0.01,inputFc=C14Atm_NH)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)

par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
legend("topright",c("Delta 14C Atmosphere", "Delta 14C pool 1", "Delta 14C pool 2"),
lty=c(1,1,1),col=c(1,4,4),lwd=c(1,1,2),bty="n")

plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

TwopMMmodel *Implementation of a two-pool Michaelis-Menten model*

Description

This function implements a two-pool Michaelis-Meneten model with a microbial biomass and a substrate pool.

Usage

```
TwopMMmodel(t,
ks=1.8e-05,
kb=0.007,
Km=900,
r=0.6,
Af=1,
ADD=3.2,
ival)
```

Arguments

t	vector of times (in days) to calculate a solution.
ks	a scalar representing SOM decomposition rate (m3 d-1 (gCB)-1)
kb	a scalar representing microbial decay rate (d-1)
Km	a scalar representing the Michaelis constant (g m-3)
r	a scalar representing the respired carbon fraction (unitless)

Af	a scalar representing the Activity factor; i.e. a temperature and moisture modifier (unitless)
ADD	a scalar representing the annual C input to the soil (g m-3 d-1)
ival	a vector of length 2 with the initial values of the SOM pool and the microbial biomass pool (g m-3)

Details

This implementation is similar to the model described in Manzoni and Porporato (2007).

Value

Microbial biomass over time

References

Manzoni, S, A. Porporato (2007). A theoretical analysis of nonlinearities and feedbacks in soil carbon and nitrogen cycles. Soil Biology and Biochemistry 39: 1542-1556.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
days=seq(0,1000,0.5)
MMmodel=TwopMMmodel(t=days,ival=c(100,10))
Cpools=getC(MMmodel)
matplot(days,Cpools,type="l",ylab="Concentrations",xlab="Days",lty=1,ylim=c(0,max(Cpools)
legend("topleft",c("SOM-C", "Microbial biomass"),lty=1,col=c(1,2),bty="n")
ks=0.000018
kb=0.007
r=0.6
ADD=3.2
#Analytical solution of fixed points
#Cs_=kb/((1-r)*ks) wrong look at the sympy test print twopMModel.pdf
Km=900
Af=1
Cs=kb*Km/(Af*ks*(1-r)-kb)
abline(h=Cs,lty=2)
Cb=(ADD*(1-r))/(r*kb)
abline(h=Cb,lty=2,col=2)
#State-space diagram
plot(Cpools[,2],Cpools[,1],type="l",ylab="SOM-C",xlab="Microbial biomass")
plot(days,Cpools[,2],type="l",col=2,xlab="Days",ylab="Microbial biomass")

#The default parameterization exhaust the microbial biomass.
#A different behavior is obtained by increasing ks and decreasing kb
MMmodel=TwopMMmodel(t=days,ival=c(972,304) ,Af=3,kb=0.0000001)
Cpools=getC(MMmodel)

matplot(days,Cpools,type="l",ylab="Concentrations",xlab="Days",lty=1,ylim=c(0,max(Cpools)
legend("topleft",c("SOM-C", "Microbial biomass"),lty=1,col=c(1,2),bty="n")

plot(Cpools[,2],Cpools[,1],type="l",ylab="SOM-C",xlab="Microbial biomass")
```

```
plot(days,Cpools[,2],type="l",col=2,xlab="Days",ylab="Microbial biomass")
```

`TwopParallelModel` *Implementation of a linear two pool model with parallel structure*

Description

This function creates a model for two independent (parallel) pools. It is a wrapper for the more general function `ParallelModel` that can handle an arbitrary number of pools.

Usage

```
TwopParallelModel(t,
ks,
C0,
In,
gam,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of length 2 containing the decomposition rates for the 2 pools.
`C0`	A vector of length 2 containing the initial amount of carbon for the 2 pools.
`In`	A scalar or a data.frame object specifying the amount of litter inputs by time.
`gam`	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 1.
`xi`	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
Ex=TwopParallelModel(t,ks=c(k1=0.5,k2=0.2),C0=c(c10=100, c20=150),In=10,gam=0.7,xi=0.5)
Ct=getC(Ex)
plot(t,rowSums(Ct),type="l",lwd=2,
ylab="Carbon stocks (arbitrary units)",xlab="Time",ylim=c(0,sum(Ct[1,])))
lines(t,Ct[,1],col=2)
lines(t,Ct[,2],col=4)
legend("topright",c("Total C","C in pool 1", "C in pool 2"),
lty=c(1,1,1),col=c(1,2,4),lwd=c(2,1,1),bty="n")

Rt=getReleaseFlux(Ex)
plot(t,rowSums(Rt),type="l",ylab="Carbon released (arbitrary units)",
xlab="Time",lwd=2,ylim=c(0,sum(Rt[1,])))
lines(t,Rt[,1],col=2)
lines(t,Rt[,2],col=4)
legend("topleft",c("Total C release","C release from pool 1", "C release from pool 2"),
lty=c(1,1,1),col=c(1,2,4),lwd=c(2,1,1),bty="n")
```

TwopParallelModel14

Implementation of a two-pool C14 model with parallel structure

---

Description

This function creates a model for two independent (parallel) pools. It is a wrapper for the more general function GeneralModel_14 that can handle an arbitrary number of pools.

Usage

```
TwopParallelModel14(t,
ks,
C0,
F0_Delta14C,
In,
gam,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.

ks	A vector of length 2 containing the decomposition rates for the 2 pools.
C0	A vector of length 2 containing the initial amount of carbon for the 2 pools.
F0_Delta14C	A vector of length 2 containing the initial amount of the fraction of radiocarbon for the 2 pools as Delta14C values in per mil.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
gam	A scalar representing the partitioning coefficient, i.e. the proportion from the total amount of inputs that goes to pool 1.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A positive scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
lag <- 2
years=seq(1901+lag,2009,by=0.5)
LitterInput=700
Ex=TwopParallelModel14(t=years,ks=c(k1=1/2.8, k2=1/35),C0=c(200,5000),
F0_Delta14C=c(0,0),In=LitterInput, gam=0.7,inputFc=C14Atm_NH,lag=lag)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)
par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
legend("topright",c("Delta 14C Atmosphere", "Delta 14C pool 1", "Delta 14C pool 2"),
lty=c(1,1,1),col=c(1,4,4),lwd=c(1,1,2),bty="n")
plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

`TwopSeriesModel` *Implementation of a two pool model with series structure*

Description

This function creates a model for two pools connected in series. It is a wrapper for the more general function `GeneralModel`.

Usage

```
TwopSeriesModel(t,
ks,
a21,
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of length 2 with the values of the decomposition rate for pools 1 and 2.
`a21`	A scalar with the value of the transfer rate from pool 1 to pool 2.
`C0`	A vector of length 2 containing the initial amount of carbon for the 2 pools.
`In`	A scalar or a data.frame object specifying the amount of litter inputs by time.
`xi`	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Sierra, C.A., M. Mueller, S.E. Trumbore. 2012. Models of soil organic matter decomposition: the SoilR package version 1.0. Geoscientific Model Development 5, 1045-1060.

See Also

There are other `predefinedModels` and also more general functions like `Model`.

Examples

```
t_start=0
t_end=10
tn=50
timestep=(t_end-t_start)/tn
t=seq(t_start,t_end,timestep)
ks=c(k1=0.8,k2=0.4)
a21=0.5
C0=c(C10=100,C20=150)
In = 30
#
Temp=rnorm(t,15,1)
TempEffect=data.frame(t,fT.Daycent1(Temp))
#
Ex1=TwopSeriesModel(t,ks,a21,C0,In,xi=TempEffect)
Ct=getC(Ex1)
Rt=getReleaseFlux(Ex1)
#
plot(t,rowSums(Ct),type="l",ylab="Carbon stocks (arbitrary units)",
xlab="Time (arbitrary units)",lwd=2,ylim=c(0,sum(Ct[1,])))
lines(t,Ct[,1],col=2)
lines(t,Ct[,2],col=4)
legend("bottomright",c("Total C","C in pool 1", "C in pool 2"),
lty=c(1,1,1),col=c(1,2,4),lwd=c(2,1,1),bty="n")
```

TwopSeriesModel14 *Implementation of a two-pool C14 model with series structure*

Description

This function creates a model for two pools connected in series. It is a wrapper for the more general function GeneralModel_14 that can handle an arbitrary number of pools.

Usage

```
TwopSeriesModel14(t,
ks,
C0,
F0_Delta14C,
In,
a21,
xi=1,
inputFc,
lambda=-0.0001209681,
lag=0,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought. It must be specified within the same period for which the Delta 14 C of the atmosphere is provided. The default period in the provided dataset C14Atm_NH is 1900-2010.

ks	A vector of length 2 containing the decomposition rates for the 2 pools.
C0	A vector of length 2 containing the initial amount of carbon for the 2 pools.
F0_Delta14C	A vector of length 2 containing the initial amount of the radiocarbon fraction for the 2 pools as Delta14C values in per mil.
In	A scalar or a data.frame object specifying the amount of litter inputs by time.
a21	A scalar with the value of the transfer rate from pool 1 to pool 2.
xi	A scalar or a data.frame specifying the external (environmental and/or edaphic) effects on decomposition rates.
inputFc	A Data Frame object containing values of atmospheric Delta14C per time. First column must be time values, second column must be Delta14C values in per mil.
lambda	Radioactive decay constant. By default lambda=-0.0001209681 y^-1 . This has the side effect that all your time related data are treated as if the time unit was year.
lag	A (positive) scalar representing a time lag for radiocarbon to enter the system.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE Forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

See Also

There are other predefinedModels and also more general functions like Model_14.

Examples

```
years=seq(1901,2009,by=0.5)
LitterInput=700
#
Ex=TwopSeriesModel14(t=years,ks=c(k1=1/2.8, k2=1/35),
C0=c(200,5000), F0_Delta14C=c(0,0),
In=LitterInput, a21=0.1,inputFc=C14Atm_NH)
R14m=getF14R(Ex)
C14m=getF14C(Ex)
C14t=getF14(Ex)
#
par(mfrow=c(2,1))
plot(C14Atm_NH,type="l",xlab="Year",
ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years, C14t[,1], col=4)
lines(years, C14t[,2],col=4,lwd=2)
legend("topright",c("Delta 14C Atmosphere", "Delta 14C pool 1", "Delta 14C pool 2"),
lty=c(1,1,1),col=c(1,4,4),lwd=c(1,1,2),bty="n")
#
plot(C14Atm_NH,type="l",xlab="Year",ylab="Delta 14C (per mil)",xlim=c(1940,2010))
lines(years,C14m,col=4)
lines(years,R14m,col=2)
legend("topright",c("Delta 14C Atmosphere","Delta 14C SOM", "Delta 14C Respired"),
lty=c(1,1,1), col=c(1,4,2),bty="n")
par(mfrow=c(1,1))
```

`UnBoundInFlux-class`
constant decomposition operator

Description

no Description

Methods

Exported methods directly defined for class UnBoundInFlux:

getFunctionDefinition `signature(object = "UnBoundInFlux"): ...` `getFunctionDefinition,UnBo`

getTimeRange `signature(object = "UnBoundInFlux"): ...` `getTimeRange,UnBoundInFlux-meth`

Methods inherited from superclasses:

from class InFlux:

GeneralInFlux `signature(object = "InFlux"): ...` `GeneralInFlux,InFlux-method`

Superclasses

`InFlux-class`

`UnBoundLinDecompOp` *UnBoundLinDecompOp S4 generic*

Description

no Description

Usage

```
UnBoundLinDecompOp(matFunc)
```

Arguments

`matFunc`	see the method arguments for details

Methods

`UnBoundLinDecompOp,function-method`

`UnBoundLinDecompOp,function-method`
UnBoundLinDecompOp,function-method construct from matrix valued function

---

Description

This method creates a UnBoundLinDecompOp from a matrix The operator is assumed to act on the vector of carbon stocks by multiplication of the (time dependent) matrix from the left.

Usage

```
## S4 method for signature 'function'
UnBoundLinDecompOp(matFunc)
```

Arguments

`matFunc` : of class function

`UnBoundLinDecompOp-class`
constant decomposition operator

Description

no Description

Methods

Exported methods directly defined for class UnBoundLinDecompOp:

BoundLinDecompOp `signature(map = "UnBoundLinDecompOp")`: ... `BoundLinDecompOp,UnBound`

getFunctionDefinition `signature(object = "UnBoundLinDecompOp")`: ... `getFunctionDefinition`

getTimeRange `signature(object = "UnBoundLinDecompOp")`: ... `getTimeRange,UnBoundLinDec`

Methods inherited from superclasses:

from class DecompOp:

GeneralDecompOp `signature(object = "DecompOp")`: ... `GeneralDecompOp,DecompOp-method`

Superclasses

`DecompOp-class`

Constructors

`UnBoundLinDecompOp`

`Yasso07Model` *Implementation of the Yasso07 model*

Description

This function creates a model for five pools as described in Tuomi et al. (2009)

Usage

```
Yasso07Model(t,
ks=c(kA = 0.66, kW = 4.3, kE = 0.35, kN = 0.22, kH = 0.0033),
p=c(p1 = 0.32,
 p2 = 0.01,
 p3 = 0.93,
 p4 = 0.34,
 p5 = 0,
 p6 = 0,
 p7 = 0.035, p8 = 0.005, p9 = 0.01, p10 = 5e-04, p11 = 0.03, p12 = 0.92, pH = 0.
C0,
In,
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

`t`	A vector containing the points in time where the solution is sought.
`ks`	A vector of lenght 5 containing the values of the decomposition rates for each pool.
`p`	A vector of length 13 containing transfer coefficients among different pools.
`C0`	A vector containing the initial amount of carbon for the 5 pools. The length of this vector must be 5.
`In`	A single scalar or data.frame object specifying the amount of litter inputs by time
`xi`	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
`solver`	A function that solves the system of ODEs. This can be `euler` or `deSolve.lsoda.wrapper` or any other user provided function with the same interface.
`pass`	if TRUE forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Tuomi, M., Thum, T., Jarvinen, H., Fronzek, S., Berg, B., Harmon, M., Trofymow, J., Sevanto, S., and Liski, J. (2009). Leaf litter decomposition-estimates of global variability based on Yasso07 model. Ecological Modelling, 220:3362 - 3371.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
years=seq(0,50,0.1)
C0=rep(100,5)
In=0

Ex1=Yasso07Model(t=years,C0=C0,In=In)
Ct=getC(Ex1)
Rt=getReleaseFlux(Ex1)

plotCPool(years,Ct,col=1:5,xlab="years",ylab="C pool",
ylim=c(0,max(Ct)))
legend("topright",c("xA","xW","xE","xN","xH"),lty=1,col=1:5,bty="n")

plotCPool(years,Rt,col=1:5,xlab="years",ylab="Respiration",ylim=c(0,50))
legend("topright",c("xA","xW","xE","xN","xH"),lty=1,col=1:5,bty="n")
```

YassoModel *Implementation of the Yasso model.*

Description

This function creates a model for seven pools as described in Liski et al. (2005). Model not yet implemented due to lack of data in original publication: values of vector p not completely described in paper. 0.1 was assumed.

Usage

```
YassoModel(t,
ks=c(a_fwl = 0.54,
 a_cwl = 0.03, k_ext = 0.48, k_cel = 0.3, k_lig = 0.22, k_hum1 = 0.012, k_hum2 =
p=c(fwl_ext = 0.1,
 cwl_ext = 0.1,
 fwl_cel = 0.1,
 cwl_cel = 0.1,
 fwl_lig = 0.1, cwl_lig = 0.1, pext = 0.05, pcel = 0.24, plig = 0.77, phum1 = 0.
C0,
In=c(u_fwl = 0.0758,
 u_cwl = 0.0866,
 u_nwl_cnwl_ext = 0.251 * 0.3,
 u_nwl_cnwl_cel = 0.251 * 0.3, u_nwl_cnwl_lig = 0.251 * 0.3, 0, 0),
xi=1,
solver=deSolve.lsoda.wrapper,
pass=FALSE)
```

Arguments

t	A vector containing the points in time where the solution is sought.
ks	A vector of lenght 7 containing the values of the exposure and decomposition rates for each pool.
p	A vector of containing transfer coefficients among different pools.
C0	A vector containing the initial amount of carbon for the 7 pools. The length of this vector must be 7.
In	A vector of constatn litter inputs.
xi	A scalar or data.frame object specifying the external (environmental and/or edaphic) effects on decomposition rates.
solver	A function that solves the system of ODEs. This can be euler or deSolve.lsoda.wrapper or any other user provided function with the same interface.
pass	if TRUE forces the constructor to create the model even if it is invalid

Value

A Model Object that can be further queried

References

Liski, J., Palosuo, T., Peltoniemi, M., and Sievanen, R. (2005). Carbon and decomposition model Yasso for forest soils. Ecological Modelling, 189:168-182.

See Also

There are other predefinedModels and also more general functions like Model.

Examples

```
years=seq(0,500,0.5)
C0=rep(100,7)
#
Ex1=YassoModel(t=years,C0=C0)
Ct=getC(Ex1)
Rt=getReleaseFlux(Ex1)
#
plotCPool(years,Ct,col=1:7,xlab="years",ylab="C pool",ylim=c(0,200))
legend("topright",c("fwl","cwl","ext","cel","lig","hum1","hum2"),lty=1,col=1:7,bty="n")
#
plotCPool(years,Rt,col=1:7,xlab="years",ylab="Respiration",ylim=c(0,50))
legend("topright",c("fwl","cwl","ext","cel","lig","hum1","hum2"),lty=1,col=1:7,bty="n")
```

[,Model,character,missing,missing-method

[,Model,character,missing,missing-method overload the [ ] operator for models and character vector

---

Description

This method is experimental and might change. User code should at the moment not rely on it. It overloads the [] operator for Model objects and acts as an alternative to the get methods so m["times"] is equivalent to getTimes(m), m["C"] to getC(m) and so on.

Usage

```
## S4 method for signature 'Model,character,missing,missing'
x[
i]
```

Arguments

x	: of class Model
i	: of class character

[,NlModel,character,ANY,ANY-method *[,NlModel,character,ANY,ANY-method overloads the [] operator*

Description

This function is still experimental and might change considerably It is meant as an alternative to all the get methods to provide a more R like interface.

Usage

```
## S4 method for signature 'NlModel,character,ANY,ANY'
x[
i]
```

Arguments

x	: of class NlModel, the model
i	: of class character, the property to access

[[,MCSim-method *[[,MCSim-method overload the [[operator*

Description

This function is still experimental and should no be used in production code yet.

Usage

```
## S4 method for signature 'MCSim'
x[[
i]]
```

Arguments

x	: of class MCSim, the simulator
i	: the index

[[<-,MCSim-method *[[<-,MCSim-method overloads the [[<- operator*

Description

This function is still experimental and should no be used in production code yet.

Arguments

x	: of class MCSim, The simulator
i	: the index
j	: ignored
...	: ignored
value	: the tasklist

$,NlModel-method *$,NlModel-method overload the $ operator*

Description

This function is still experimental and might change considerably It is meant as an alternative to all the get methods to provide a more R like interface.

Usage

```
## S4 method for signature 'NlModel'
x$
name
```

Arguments

x	: of class NlModel, the model
name	: the property to access

Index