

Package ‘solarr’

March 20, 2025

Type Package

Title Stochastic models for solar radiation

Version 1.0.0

Author Beniamino Sartini

Maintainer Beniamino Sartini <beniamino.sartini2@unibo.it>

Description Implementation of stochastic models and option pricing on solar radiation data.

Depends R (>= 4.4.0),

ggplot2,

np,

dplyr,

mclust,

Imports stringr,

rugarch,

purrr,

tidyr,

lubridate,

nortest,

broom,

formula.tools,

numDeriv

Suggests knitr,

rmarkdown,

testthat

License GPL-3

VignetteBuilder knitr

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE, r6 = TRUE)

RoxygenNote 7.3.2

Contents

ARMA_modelR6	3
calibrate_dQdP_solar	5
control_hedging	6

control_seasonalClearsky	6
control_solarModel	7
control_solarOption	9
desscher	10
desscherMixture	11
detect_season	12
dgumbel	13
dinvgumbel	14
discountFactor	15
dkumaraswamy	15
dmixnorm	16
dmvmixnorm	17
dmvsolarGHI	18
dsnrm	19
dsolarGHI	20
dsolarK	21
dsolarX	23
GARCH_modelR6	24
gaussianMixture	27
havDistance	30
IDW	31
is_leap_year	31
kernelRegression	32
ks_test	33
ks_test_ts	34
loss_dQdP	34
loss_dQdP_tau	35
makeSemiPositive	35
monthlyParams	35
mvgaussianMixture	37
number_of_day	37
optionPayoff	38
PDF	38
plot.solarModelForecast	39
plot.solarScenario	39
pow_matrix	40
predict.solarModel	40
prepare_dQdP_calibration	40
prepare_dQdP_calibration_tau	41
radiationModel	41
riccati_root	47
seasonalClearsky	48
seasonalModel	49
seasonalSolarFunctions	52
solarEsscher	58
solarEsscher_probability	59
solarMixture	60
solarModel	62
solarModels_control_grid	67
solarModels_fit_control_grid	68
solarModel_forecast	68
solarModel_moments	68

solarModel_mvmmixture	69
solarModel_spec	69
solarModel_tests	70
solarModel_test_autocorr	71
solarModel_test_distribution	71
solarOption	72
solarOptionPayoffs	73
solarOption_calibrator	74
solarOption_contracts	75
solarOption_historical	75
solarOption_historical_bootstrap	76
solarOption_model	77
solarOption_pricing	77
solarOption_scenario	78
solarOption_structure	79
solarScenario	80
solarScenario_filter	81
solarScenario_residuals	81
solarScenario_spec	82
solarTransform	83
SoRadPorfolio	86
spatialCorrelation	87
spatialGrid	88
spatialModel	90
spatialParameters	92
spatialScenario_filter	93
spatialScenario_residuals	93
spatialScenario_spec	94
spectralDistribution	94
test_normality	95
tnorm	95

Index**97**

ARMA_modelR6

*ARMA(p, q) model implementation in a class R6***Description**

ARMA(p, q) model implementation in a class R6

ARMA(p, q) model implementation in a class R6

Active bindings

intercept Numeric named scalar. Intercept.

phi Numeric named vector. AR parameters.

theta Numeric named vector. MA parameters.

coefficients Numeric named vector. Intercept and ARMA parameters.

order Numeric named vector. ARMA order.

mean Numeric scalar. Long term expectation.

variance Numeric scalar. Long term variance.
 model Fitted ARMA model from the function arima.
 Phi Numeric, matrix. Companion matrix.
 b Numeric, vector. Vector for matrix form of the residuals.

Methods

Public methods:

- `ARMA_modelR6$new()`
- `ARMA_modelR6$fit()`
- `ARMA_modelR6$filter()`
- `ARMA_modelR6$next_step()`
- `ARMA_modelR6$update()`
- `ARMA_modelR6$print()`
- `ARMA_modelR6$clone()`

Method `new()`: Initialize an ARMA model

Usage:

```
ARMA_modelR6$new(arOrder = 1, maOrder = 1, include.intercept = FALSE)
```

Arguments:

arOrder Numeric, scalar. Order for Autoregressive component.
 maOrder Numeric, scalar. Order for Moving-Average component.
 include.intercept Logical. When TRUE the intercept will be included. The default is FALSE.

Method `fit()`: Fit the ARMA model with arima function.

Usage:

```
ARMA_modelR6$fit(x)
```

Arguments:

x Numeric, vector. Time series to fit.

Method `filter()`: Filter the time-series and compute fitted values and residuals.

Usage:

```
ARMA_modelR6$filter(x, eps0)
```

Arguments:

x Numeric, vector. Time series to filter.
 eps0 Numeric vector. Initial residuals of the same length of the MA order.

Method `next_step()`: Next step function

Usage:

```
ARMA_modelR6$next_step(x, n.ahead = 1, eps = 0)
```

Arguments:

x Numeric, vector. State vector with past observations and residuals.
 n.ahead Numeric, scalar. Number of steps ahead.
 eps Numeric vector. Optional realized residuals.

Method `update()`: Update the model's parameters

Usage:

```
ARMA_modelR6$update(coefficients)
```

Arguments:

`coefficients` Numeric, named vector. Model's coefficients. When missing will be used the fitted parameters.

Method `print()`: Print method for AR_modelR6 class.

Usage:

```
ARMA_modelR6$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
ARMA_modelR6$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

Examples

```
# Reference time series
x <- Bologna$data$Yt_tilde
# Initialize the model without intercept
arma <- ARMA_modelR6$new(arOrder = 2, maOrder = 2)
# Fit the model
arma$fit(x)
arma
arma$coefficients
arma$variance
# Next step prediction
arma$next_step(c(0.4, 0.2, 0.2, -0.2))
arma$next_step(c(0.4, 0.2, 0.2, -0.2), n.ahead = 10)
# Update coefficients
params <- arma$coefficients*1.01
arma$update(params)
arma
# All sample prediction
arma$filter(x, arma$model$residuals[c(1,2)])
```

calibrate_dQdP_solar *Calibration for the best lambda*

Description

Calibration for the best lambda

Usage

```
calibrate_dQdP_solar(data_dQdP, r = 0, nmonths = 1:12)
```

control_hedging	<i>Control for hedging</i>
-----------------	----------------------------

Description

Control for hedging

Usage

```
control_hedging(
  n_panels = 1,
  efficiency = 1,
  PUN = 1,
  tick = 1,
  n_contracts = 1
)
```

Arguments

n_panels	numeric, number of meters squared of solar panels.
efficiency	numeric, mean efficiency of the solar panels.
PUN	numeric, mean efficiency of the solar panels.
tick	numeric, conversion tick for the monetary payoff of a contract.
n_contract	numeric, number of contracts

control_seasonalClearsky	<i>Control parameters for a seasonalClearsky object</i>
--------------------------	---

Description

Control parameters for a seasonalClearsky object

Usage

```
control_seasonalClearsky(
  order = 1,
  period = 365,
  include.intercept = TRUE,
  delta0 = 1.4,
  lower = 0,
  upper = 3,
  by = 0.001,
  ntol = 0,
  quiet = FALSE
)
```

Arguments

order	Integer scalar, number of combinations of sines and cosines.
period	Integer scalar, seasonality period. The default is 365.
include.intercept	Logical, when TRUE, the default, the intercept will be included in the clear sky model.
delta0	Numeric scalar, initial value for delta.
lower	Numeric scalar. Lower bound for grid of delta parameters used for optimization. Default is 0.
upper	Numeric scalar. Upper bound for grid of delta parameters used for optimization. Default is 3.
by	Numeric scalar, step for the grid. Default is 0.01.
ntol	Integer scalar. Tolerance for the maximum number of violations admitted of the condition <code>clearsky > GHI</code> . Default is 0.
quiet	Logical, when FALSE, the default, the functions displays warning or messages.

Details

The parameters `ntol`, `lower`, `upper` and `by` are used exclusively in [clearsky_optimizer](#).

Value

Named list of control parameters.

Note

Version 1.0.0

Version 1.0.0

Examples

```
control = control_seasonalClearsky()
```

control_solarModel	<i>Control function for a solarModel object</i>
--------------------	---

Description

Control function for a `solarModel` object that contains all the setups used for the estimation.

Usage

```
control_solarModel(
  clearsky = control_seasonalClearsky(),
  stochastic_clearsky = FALSE,
  seasonal.mean = list(seasonalOrder = 1, include.H0 = FALSE, include.intercept = TRUE,
    monthly.mean = TRUE),
  mean.model = list(arOrder = 1, maOrder = 0, include.intercept = FALSE),
  seasonal.variance = list(seasonalOrder = 1, correction = TRUE, monthly.mean = TRUE),
```

```

variance.model = rugarch::ugarchspec(variance.model = list(garchOrder = c(1, 1),
  variance.targeting = 1), mean.model = list(armaOrder = c(0, 0), include.mean =
  FALSE)),
mixture.model = list(abstol = 1e-20, maxit = 100),
clearsky_threshold = 1.01,
threshold = 0.01,
outliers_quantile = 0,
garch_variance = TRUE,
quiet = FALSE
)

```

Arguments

clearsky List of control parameters for the clear sky model. See [control_seasonalClearsky](#) for more details.

seasonal.mean List of parameters to control the seasonal mean. Available choices are:
seasonalOrder Integer. Specify the order of the seasonal mean

$$\bar{Y}_t$$

. The default is 1.

include.intercept Logical. When TRUE the intercept

$$a_0$$

will be included in the seasonal model, otherwise will be excluded. The default is TRUE.

include.H0 Logical. When TRUE the extraterrestrial radiation will be included in the seasonal model, otherwise will be excluded. The default is FALSE.

monthly.mean Logical. When TRUE a vector of 12 monthly means will be computed on the deseasonalized series

$$\tilde{Y}_t = Y_t - \bar{Y}_t$$

and it is subtracted to ensure that the time series is centered around zero for all the months. The default if TRUE.

mean.model List of parameters to control the AR model. Available choices are:

arOrder Integer. An integer specifying the order of the AR component. The default is 1.

maOrder Integer. An integer specifying the order of the MA component. The default is 0.

include.intercept When TRUE an intercept will be included in the AR equation, otherwise is omitted. The default if FALSE.

seasonal.variance

List of parameters to control the seasonal variance. Available choices are:

seasonalOrder Integer. Specify the order of the seasonality of the seasonal variance. The default is 1.

correction Logical. When TRUE the parameters of seasonal variance are corrected to ensure that the standardize the residuals have exactly a unitary variance. The default if TRUE.

monthly.mean Logical. When TRUE a vector of 12 monthly std. deviations will be computed on the standardized residuals

$$\tilde{\varepsilon}_t$$

and used to standardize the time series such that it has unitary variance for all the months. The default if TRUE.

variance.model ugarchspec object for GARCH variance. Default is a GARCH(1,1) specification.

mixture.model List of parameters for the Gaussian mixture model. Available choices are:
 abstol Numeric. Absolute level for convergence of the EM-algorithm. The default is 1e-20.
 maxit Integer. Maximum number of iterations for EM-algorithm. The default is 100.

threshold Numeric. Threshold used to estimate the transformation parameters

$$\alpha$$

and

$$\beta$$

. The default is 0.01. See [solarTransform](#) for more details.

outliers_quantile Numeric, optional. Quantile for outliers detection that applies only when different from 0. In that case the observations that are below the quantile at confidence levels outliers_quantile and the observation above the quantile at confidence level 1-outliers_quantile will have a weight equal to zero and will be excluded from estimations of the parameters.

garch_variance Logical. When TRUE the GARCH model will be used to standardize the residuals otherwise will be excluded. The default if TRUE.

quiet Logical. When TRUE the function will not display any message. The default if TRUE.

Examples

```
control <- control_solarModel()
```

control_solarOption	<i>Control parameters for a solar option</i>
---------------------	--

Description

Control parameters for a solar option

Usage

```
control_solarOption(
  nyears = c(2005, 2023),
  K = 0,
  leap_year = FALSE,
  nsim = 200,
  ci = 0.05,
  seed = 1,
  B = discountFactor()
)
```

Arguments

nyears	numeric vector. Interval of years considered. The first element will be the minimum and the second the maximum years used in the computation of the fair payoff.
K	numeric, level for the strike with respect to the seasonal mean. The seasonal mean is multiplied by $\exp(K)$.
leap_year	logical, when FALSE, the default, the year will be considered of 365 days, otherwise 366.
nsim	integer, number of simulations used to bootstrap the premium's bounds. See solarOption_historical_bootstrap .
ci	numeric, confidence interval for bootstrapping. See solarOption_historical_bootstrap .
seed	integer, random seed for reproducibility. See solarOption_historical_bootstrap .
B	function. Discount factor function. Should take as input a number (in years) and return a discount factor.

Examples

```
control_options <- control_solarOption()
```

 desscher

Esscher transform of a density

Description

Given a function of x , i.e. $f_X(x)$, compute its Esscher transform and return again a function of x .

Usage

```
desscher(pdf, theta = 0, lower = -Inf, upper = Inf)
```

Arguments

pdf	density function.
theta	Esscher parameter.
lower	numeric, lower bound for integration, i.e. the lower bound for the pdf.
upper	numeric, lower bound for integration, i.e. the upper bound for the pdf.

Details

Given a pdf $f_X(x)$ the function computes its Esscher transform, i.e.

$$\mathcal{E}_\theta\{f_X(x)\} = \frac{e^{\theta x} f_X(x)}{\int_{-\infty}^{\infty} e^{\theta x} f_X(x) dx}$$

Examples

```
# Grid of points
grid <- seq(-3, 3, 0.1)
# Density function of x
pdf <- function(x) dnorm(x, mean = 0)
# Esscher density (no transform)
esscher_pdf <- desscher(pdf, theta = 0)
pdf(grid) - esscher_pdf(grid)
# Esscher density (transform)
esscher_pdf_1 <- function(x) dnorm(x, mean = -0.1)
esscher_pdf_2 <- desscher(pdf, theta = -0.1)
esscher_pdf_1(grid) - esscher_pdf_2(grid)
# Log-probabilities
esscher_pdf(grid, log = TRUE)
esscher_pdf_2(grid, log = TRUE)
```

desscherMixture

Density of the Esscher transform of a Gaussian Mixture

Description

Density of the Esscher transform of a Gaussian Mixture

Cdf of the Esscher transform of a Gaussian Mixture

Usage

```
desscherMixture(mean = c(0, 0), sd = c(1, 1), alpha = c(0.5, 0.5), theta = 0)
```

```
pesscherMixture(mean = c(0, 0), sd = c(1, 1), alpha = c(0.5, 0.5), theta = 0)
```

Arguments

mean	vector of means parameters.
sd	vector of std. deviation parameters.
alpha	vector of probability parameters for each component.
theta	Esscher parameter, the default is zero.

Examples

```
library(ggplot2)
grid <- seq(-5, 5, 0.01)
# Density
pdf_1 <- desscherMixture(mean = c(-3, 3), theta = 0)(grid)
pdf_2 <- desscherMixture(mean = c(-3, 3), theta = -0.5)(grid)
pdf_3 <- desscherMixture(mean = c(-3, 3), theta = 0.5)(grid)
ggplot()+
  geom_line(aes(grid, pdf_1), color = "black")+
  geom_line(aes(grid, pdf_2), color = "green")+
  geom_line(aes(grid, pdf_3), color = "red")
# Distribution
cdf_1 <- pesscherMixture(mean = c(-3, 3), theta = 0)(grid)
cdf_2 <- pesscherMixture(mean = c(-3, 3), theta = -0.2)(grid)
cdf_3 <- pesscherMixture(mean = c(-3, 3), theta = 0.2)(grid)
ggplot()+
  geom_line(aes(grid, cdf_1), color = "black")+
  geom_line(aes(grid, cdf_2), color = "green")+
  geom_line(aes(grid, cdf_3), color = "red")
```

detect_season

*Detect the season***Description**

Detect the season from a vector of dates

Usage

```
detect_season(x, invert = FALSE)
```

Arguments

x vector of dates in the format YYYY-MM-DD.

invert logical, when TRUE the seasons will be inverted.

Value

a character vector containing the correspondent season. Can be spring, summer, autumn, winter.

Examples

```
detect_season("2040-01-31")
detect_season(c("2040-01-31", "2023-04-01", "2015-09-02"))
```

dgumbel

Gumbel random variable

Description

Gumbel density, distribution, quantile and random generator.

Usage

```
dgumbel(x, location = 0, scale = 1, log = FALSE)

pgumbel(q, location = 0, scale = 1, log.p = FALSE, lower.tail = TRUE)

qgumbel(p, location = 0, scale = 1, log.p = FALSE, lower.tail = TRUE)

rgumbel(n, location = 0, scale = 1)
```

Arguments

x, q	vector of quantiles.
location	location parameter.
scale	scale parameter.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X < x]$ otherwise, $P[X > x]$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
# Grid
x <- seq(-5, 5, 0.01)

# Density function
p <- dgumbel(x, location = 0, scale = 1)
plot(x, p, type = "l")

# Distribution function
p <- pgumbel(x, location = 0, scale = 1)
plot(x, p, type = "l")

# Quantile function
qgumbel(0.1)
pgumbel(qgumbel(0.1))

# Random Numbers
rgumbel(1000)
plot(rgumbel(1000), type = "l")
```

dinvgumbel

*Inverted Gumbel random variable***Description**

Inverted Gumbel density, distribution, quantile and random generator.

Usage

```
dinvgumbel(x, location = 0, scale = 1, log = FALSE)

pinvgumbel(q, location = 0, scale = 1, log.p = FALSE, lower.tail = TRUE)

qinvgumbel(p, location = 0, scale = 1, log.p = FALSE, lower.tail = TRUE)

rinvgumbel(n, location = 0, scale = 1)
```

Arguments

x, q	vector of quantiles.
location	location parameter.
scale	scale parameter.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X < x]$ otherwise, $P[X > x]$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
# Grid
x <- seq(-5, 5, 0.01)

# Density function
p <- dinvgumbel(x, location = 0, scale = 1)
plot(x, p, type = "l")

# Distribution function
p <- pinvgumbel(x, location = 0, scale = 1)
plot(x, p, type = "l")

# Quantile function
qgumbel(0.1)
pinvgumbel(qinvgumbel(0.1))

# Random Numbers
rinvgumbel(1000)
plot(rinvgumbel(1000), type = "l")
```

discountFactor	<i>Discount factor function</i>
----------------	---------------------------------

Description

Discount factor function

Usage

```
discountFactor(r = 0.03, discrete = TRUE)
```

Arguments

r	level of yearly constant risk-free rate
discrete	logical, when TRUE, the default, discrete compounding will be used. Otherwise continuous compounding.

dkumaraswamy	<i>Kumaraswamy random variable</i>
--------------	------------------------------------

Description

Kumaraswamy density, distribution, quantile and random generator.

Usage

```
dkumaraswamy(x, a = 1, b = 1, log = FALSE)

pkumaraswamy(q, a = 1, b = 1, log.p = FALSE, lower.tail = TRUE)

qkumaraswamy(p, a = 1, b = 1, log.p = FALSE, lower.tail = TRUE)

rkumaraswamy(n, a = 1, b = 1)
```

Arguments

x, q	vector of quantiles.
a	parameter $a > 0$.
b	parameter $b > 0$.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE, the default, the computed probabilities are $P[X < x]$. Otherwise, $P[X > x]$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
# Grid
x <- seq(0, 1, 0.01)

# Density function
plot(x, dkumaraswamy(x, 0.2, 0.3), type = "l")
plot(x, dkumaraswamy(x, 2, 1.1), type = "l")

# Distribution function
plot(x, pkumaraswamy(x, 2, 1.1), type = "l")

# Quantile function
qkumaraswamy(0.2, 0.4, 1.4)
qkumaraswamy(pkumaraswamy(0.4, 2, 1.1), 2, 1.1)

# Random generator
rkumaraswamy(20, 0.4, 1.4)
```

dmixnorm

*Gaussian mixture random variable***Description**

Gaussian mixture density, distribution, quantile and random generator.

Usage

```
dmixnorm(x, mean = rep(0, 2), sd = rep(1, 2), alpha = rep(1/2, 2), log = FALSE)

pmixnorm(
  q,
  mean = rep(0, 2),
  sd = rep(1, 2),
  alpha = rep(1/2, 2),
  lower.tail = TRUE,
  log.p = FALSE
)

qmixnorm(
  p,
  mean = rep(0, 2),
  sd = rep(1, 2),
  alpha = rep(1/2, 2),
  lower.tail = TRUE,
  log.p = FALSE
)

rmixnorm(n, mean = rep(0, 3), sd = rep(1, 3), alpha = rep(1/3, 3))
```


Arguments

x, q	vector of quantiles.
mean	vector of means parameters.
sd	vector of std. deviation parameters.
alpha	vector of probability parameters for each component.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X < x]$ otherwise, $P[X > x]$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
# Parameters
mean = c(-3,0,3)
sd = rep(1, 3)
alpha = c(0.2, 0.3, 0.5)
# Density function
dmixnorm(3, mean, sd, alpha)
# Distribution function
dmixnorm(c(1.2, -3), mean, sd, alpha)
# Quantile function
qmixonorm(0.2, mean, sd, alpha)
# Random generator
rmixonorm(1000, mean, sd, alpha)
```

dmvmixnorm

*Multivariate Gaussian mixture random variable***Description**

Multivariate Gaussian mixture density, distribution, quantile and random generator.

Usage

```
dmvmixnorm(
  x,
  means = matrix(0, 2, 2),
  sigma2 = matrix(1, 2, 2),
  p = rep(1/2, 2),
  rho = c(0, 0),
  log = FALSE
)

pmvmixnorm(
  x,
  means = matrix(0, 2, 2),
```

```

    sigma2 = matrix(1, 2, 2),
    p = rep(1/2, 2),
    rho = c(0, 0),
    lower = -Inf,
    log.p = FALSE
  )

  qvmixnorm(
    x,
    means = matrix(0, 2, 2),
    sigma2 = matrix(1, 2, 2),
    p = rep(1/2, 2),
    rho = c(0, 0),
    log.p = FALSE
  )

```

Examples

```

# Means components
mean_1 = c(-1.8,-0.4)
mean_2 = c(0.6, 0.5)
# Dimension of the random variable
j = length(mean_1)
# Matrix of means
means = matrix(c(mean_1, mean_2), j,j, byrow = TRUE)

# Variance components
var_1 = c(1,1.4)
var_2 = c(1.3, 1.2)
# Matrix of variances
sigma2 = matrix(c(var_1, var_2), j,j, byrow = TRUE)

# Correlations
rho <- c(rho_1 = 0.2, rho_2 = 0.3)

# Probability for each component
p <- c(0.4, 0.6)

x <- matrix(c(0.1,-0.1), nrow = 1)
dmvmixnorm(x, means, sigma2, p, rho)
pmvmixnorm(x, means, sigma2, p, rho)
qvmixnorm(0.35, means, sigma2, p, rho)

```

dmvsolarGHI

Bivariate PDF GHI

Description

Bivariate PDF GHI

Usage

```
dmvsolarGHI(x, Ct, alpha, beta, joint_pdf_Yt)
```

Arguments

x	vector of quantiles.
Ct	clear sky radiation
alpha	parameters $\alpha > 0$.
beta	parameters $\beta > 0$ and $\alpha + \beta < 1$.
joint_pdf_Yt	joint density of Y1_t, Y2_t.

dsnorm	<i>Skewed Normal random variable</i>
--------	--------------------------------------

Description

Skewed Normal density, distribution, quantile and random generator.

Usage

```
dsnorm(x, location = 0, scale = 1, shape = 0, log = FALSE)

psnorm(q, location = 0, scale = 1, shape = 0, log.p = FALSE, lower.tail = TRUE)

qsnorm(p, location = 0, scale = 1, shape = 0, log.p = FALSE, lower.tail = TRUE)

rsnorm(n, location = 0, scale = 1, shape = 0)
```

Arguments

x, q	vector of quantiles.
location	location parameter.
scale	scale parameter.
shape	skewness parameter.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X < x]$ otherwise, $P[X > x]$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
# Grid of points
x <- seq(-5, 5, 0.01)

# Density function
# right tailed
plot(x, dsnorm(x, shape = 1.9), type = "l")
# left tailed
plot(x, dsnorm(x, shape = -1.9), type = "l")
```

```
# Distribution function
plot(x, psnorm(x, shape = 4.9), type = "l")
plot(x, psnorm(x, shape = -4.9), type = "l")

# Quantile function
dsnorm(0.1, shape = 4.9)
dsnorm(0.1, shape = -4.9)
psnorm(qsnorm(0.9, shape = 3), shape = 3)

# Random generator
set.seed(1)
plot(rsnorm(100, shape = 5), type = "l")
```

dsolarGHI

Solar radiation random variable

Description

Solar radiation density, distribution, quantile and random generator.

Usage

```
dsolarGHI(x, Ct, alpha, beta, pdf_Y, log = FALSE)

psolarGHI(x, Ct, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

qsolarGHI(p, Ct, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

rsolarGHI(n, Ct, alpha, beta, cdf_Y)
```

Arguments

x	vector of quantiles.
Ct	clear sky radiation
alpha	parameter $\alpha > 0$.
beta	parameter $\beta > 0$ and $\alpha + \beta < 1$.
pdf_Y	density of Y.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
cdf_Y	distribution of Y.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE, the default, the computed probabilities are $P[X < x]$. Otherwise, $P[X > x]$.
p	vector of probabilities.

Details

Consider a random variable $Y \in [-\infty, \infty]$ with a known density function pdf_Y. Then the funtion dsolarGHI compute the density function of the following transformed random variable, i.e.

$$GHI(Y) = C_t(1 - \alpha - \beta \exp(-\exp(Y)))$$

where $GHI(Y) \in [C_t(1 - \alpha - \beta), C_t(1 - \alpha)]$.

Examples

```
# Parameters
alpha = 0
beta = 0.9
Ct <- 7
# Grid of points
grid <- seq(Ct*(1-alpha-beta), Ct*(1-alpha), by = 0.01)

# Density
dsolarGHI(5, Ct, alpha, beta, function(x) dnorm(x))
dsolarGHI(5, Ct, alpha, beta, function(x) dnorm(x, sd=2))
plot(grid, dsolarGHI(grid, Ct, alpha, beta, function(x) dnorm(x, mean = -1, sd = 0.9)), type="l")

# Distribution
psolarGHI(3.993, 7, 0.001, 0.9, function(x) pnorm(x))
psolarGHI(3.993, 7, 0.001, 0.9, function(x) pnorm(x, sd=2))
plot(grid, psolarGHI(grid, Ct, alpha, beta, function(x) pnorm(x, sd = 0.2)), type="l")

# Quantile
qsolarGHI(c(0.05, 0.95), 7, 0.001, 0.9, function(x) pnorm(x))
qsolarGHI(c(0.05, 0.95), 7, 0.001, 0.9, function(x) pnorm(x, sd=2))

# Random generator (I)
Ct <- Bologna$seasonal_data$Ct
GHI <- purrr::map(Ct, ~rsolarGHI(1, .x, alpha, beta, function(x) pnorm(x, sd=1.4)))
plot(1:366, GHI, type="l")

# Random generator (II)
cdf <- function(x) pmixnorm(x, c(-0.8, 0.5), c(1.2, 0.7), c(0.3, 0.7))
GHI <- purrr::map(Ct, ~rsolarGHI(1, .x, 0.001, 0.9, cdf))
plot(1:366, GHI, type="l")
```

dsolarK

*Clearness index random variable***Description**

Clearness index density, distribution, quantile and random generator.

Usage

```
dsolarK(x, alpha, beta, pdf_Y, log = FALSE)

psolarK(x, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

qsolarK(p, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

rsolarK(n, alpha, beta, cdf_Y)
```

Arguments

x vector of quantiles.
alpha parameter $\alpha > 0$.

beta	parameter $\beta > 0$ and $\alpha + \beta < 1$.
pdf_Y	density function of Y.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
cdf_Y	distribution function of Y.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE, the default, the computed probabilities are $P[X < x]$. Otherwise, $P[X > x]$.
p	vector of probabilities.

Details

Consider a random variable $Y \in [-\infty, \infty]$ with a known density function pdf_Y. Then the function dsolarK compute the density function of the following transformed random variable, i.e.

$$K(Y) = 1 - \alpha - \beta \exp(-\exp(Y))$$

where $K(Y) \in [1 - \alpha - \beta, 1 - \alpha]$.

Examples

```
# Parameters
alpha = 0.001
beta = 0.9
# Grid of points
grid <- seq(1-alpha-beta, 1-alpha, length.out = 50)[-50]

# Density
dsolarK(0.4, alpha, beta, function(x) dnorm(x))
dsolarK(0.4, alpha, beta, function(x) dnorm(x, sd = 2))
plot(grid, dsolarK(grid, alpha, beta, function(x) dnorm(x, sd = 0.2)), type="l")

# Distribution
psolarK(0.493, alpha, beta, function(x) pnorm(x))
psolarK(0.493, alpha, beta, function(x) pnorm(x, sd = 2))
plot(grid, psolarK(grid, alpha, beta, function(x) pt(0.2*x, 3)), type="l")
plot(grid, psolarK(grid, alpha, beta, function(x) pnorm(x, sd = 0.2)), type="l")

# Quantile
qsolarK(c(0.05, 0.95), alpha, beta, function(x) pnorm(x))
qsolarK(c(0.05, 0.95), alpha, beta, function(x) pnorm(x, sd = 2))

# Random generator (I)
Kt <- rsolarK(366, alpha, beta, function(x) pnorm(x, sd = 1.3))
plot(1:366, Kt, type="l")

# Random generator (II)
pdf <- function(x) pmixnorm(x, c(-1.8, 0.8), c(0.5, 0.7), c(0.6, 0.4))
Kt <- rsolarK(36, alpha, beta, pdf)
plot(1:36, Kt, type="l")
```

dsolarX	<i>Solar risk driver random variable</i>
---------	--

Description

Solar risk driver density, distribution, quantile and random generator.

Usage

```
dsolarX(x, alpha, beta, pdf_Y, log = FALSE)

psolarX(x, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

qsolarX(p, alpha, beta, cdf_Y, log.p = FALSE, lower.tail = TRUE)

rsolarX(n, alpha, beta, cdf_Y)
```

Arguments

x	vector of quantiles.
alpha	parameter $\alpha > 0$.
beta	parameter $\beta > 0$ and $\alpha + \beta < 1$.
pdf_Y	density of Y.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
cdf_Y	distribution function of Y.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE, the default, the computed probabilities are $P[X < x]$. Otherwise, $P[X > x]$.
p	vector of probabilities.

Details

Consider a random variable $Y \in [-\infty, \infty]$ with a known density function pdf_Y. Then the funtion dsolarX compute the density function of the following transformed random variable, i.e.

$$X(Y) = \alpha + \beta \exp(-\exp(Y))$$

where $X(Y) \in [\alpha, \alpha + \beta]$.

Examples

```
# Parameters
alpha = 0.001
beta = 0.9
# Grid of points
grid <- seq(alpha, alpha+beta, length.out = 50)[-50]

# Density
dsolarX(0.4, alpha, beta, function(x) dnorm(x))
dsolarX(0.4, alpha, beta, function(x) dnorm(x, sd = 2))
```

```

plot(grid, dsolarX(grid, alpha, beta, function(x) dnorm(x, sd = 0.2)), type="l")

# Distribution
psolarX(0.493, alpha, beta, function(x) pnorm(x))
dsolarX(0.493, alpha, beta, function(x) pnorm(x, sd = 2))
plot(grid, psolarX(grid, alpha, beta, function(x) pnorm(x, sd = 0.2)), type="l")

# Quantile
qsolarX(c(0.05, 0.95), alpha, beta, function(x) pnorm(x))
qsolarX(c(0.05, 0.95), alpha, beta, function(x) pnorm(x, sd = 1.3))

# Random generator (I)
set.seed(1)
Kt <- rsolarX(366, alpha, beta, function(x) pnorm(x, sd = 0.8))
plot(1:366, Kt, type="l")

# Random generator (II)
cdf <- function(x) pmixnorm(x, c(-1.8, 0.9), c(0.5, 0.7), c(0.6, 0.4))
Kt <- rsolarX(366, alpha, beta, cdf)
plot(1:366, Kt, type="l")

```

GARCH_modelR6

*GARCH(p,q) model implementation in a class R6***Description**

GARCH(p,q) model implementation in a class R6

GARCH(p,q) model implementation in a class R6

Active bindings

spec model specification
 order model order
 coefficients model coefficients
 std.errors parameters std. errors
 omega intercept
 alpha arch parameters
 beta garch parameters
 vol model unconditional std. deviation
 loglik model loglik
 hessian hessian matrix

Methods**Public methods:**

- [GARCH_modelR6\\$new\(\)](#)
- [GARCH_modelR6\\$fit\(\)](#)
- [GARCH_modelR6\\$filter\(\)](#)
- [GARCH_modelR6\\$logLik\(\)](#)

- `GARCH_modelR6$update()`
- `GARCH_modelR6$next_step()`
- `GARCH_modelR6$Hessian()`
- `GARCH_modelR6$print()`
- `GARCH_modelR6$clone()`

Method `new()`: Initialize a GARCH model with rugarch specification

Usage:

```
GARCH_modelR6$new(spec, x, weights, sigma20)
```

Arguments:

`spec` GARCH specification from ugarchspec.

`x` Numeric, vector. Time series to be fitted.

`weights` Numeric, vector. Optional custom weights.

`sigma20` Numeric scalar. Target unconditional variance.

Method `fit()`: Fit the GARCH model with rugarch function.

Usage:

```
GARCH_modelR6$fit()
```

Method `filter()`: Filter method from rugarch package to compute GARCH variance, residuals and log-likelihoods.

Usage:

```
GARCH_modelR6$filter(x, coefficients, ...)
```

Arguments:

`x` Numeric, vector. Time series to be filtered.

`coefficients` Numeric, named vector. Model's coefficients. When missing will be used the fitted parameters.

`...` Other arguments passed to ugarchfilter function.

Method `logLik()`: Log-likelihoods function

Usage:

```
GARCH_modelR6$logLik(coefficients, x, weights, ...)
```

Arguments:

`coefficients` Numeric, named vector. Model's coefficients. When missing will be used the fitted parameters.

`x` Numeric, vector. Time series used to compute log-likelihoods.

`weights` Numeric, vector. Optional custom weights.

`...` Other arguments passed to ugarchfilter function.

Method `update()`: Update the coefficients of the model

Usage:

```
GARCH_modelR6$update(coefficients, ...)
```

Arguments:

`coefficients` Numeric, named vector. Model's coefficients.

`...` Other arguments passed to ugarchfilter function.

Method `next_step()`: Next step GARCH std. deviation forecast

Usage:

```
GARCH_modelR6$next_step(x = 1, sigma = 1, n.ahead = 1)
```

Arguments:

x Numeric, vector. Past residuals.

sigma Numeric, vector. Past garch std. deviations.

n.ahead Numeric, scalar. Number of steps ahead.

Method Hessian(): Numerical computation of the Hessian matrix.

Usage:

```
GARCH_modelR6$Hessian()
```

Method print(): Print method for GARCH_modelR6 class. Manual fit of the GARCH model
Manual computation of the parameters std. errors

Usage:

```
GARCH_modelR6$print()
```

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
GARCH_modelR6$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Note

Version 1.0.0

Examples

```
# Specification
GARCH_spec <- ugarchspec(variance.model = list(model = "sGARCH", garchOrder = c(1,1), variance.targeting = 1),
mean.model = list(armaOrder = c(0,0), include.mean = FALSE))
# Reference time series
x <- Bologna$data$eps_tilde

# Initialize the model
GARCH_model <- GARCH_modelR6$new(GARCH_spec, x, sigma20 = 1)
# Fit the model
GARCH_model$fit()
GARCH_model

# Next step prediction
GARCH_model$next_step(0.3, 0.9)
GARCH_model$next_step(0.3, 0.9, 7)

# Update coefficients
GARCH_model$update(c(0.05, 0.02, 0.9))
GARCH_model

# Next step prediction
GARCH_model$next_step(0.3, 0.9)
GARCH_model$next_step(0.3, 0.9, 7)

# All sample prediction
GARCH_model$filter()
```

gaussianMixture	<i>Gaussian mixture</i>
-----------------	-------------------------

Description

Gaussian mixture

Gaussian mixture

Details

Fit the parameters of a gaussian mixture with k-components.

Applied after updating the parameters

Applied after updating the parameters

Public fields

`maxit` Integer, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

`components` Integer, number of mixture components.

Active bindings

`means` Numeric vector containing the location parameter for each component.

`sd` Numeric vector containing the scale parameter for each component.

`p` Numeric vector containing the probability for each component.

`coefficients` named list with mixture coefficients.

`std.errors` named list with mixture parameters.

`model` Tibble with mixture parameters, in order means, sd, p.

`loglik` log-likelihood of the fitted series.

`fitted` fitted series

`moments` Tibble with the theoretic moments and the number of observations used for fit.

`summary` Tibble with estimated parameters, std.errors and statistics

Methods**Public methods:**

- `gaussianMixture$new()`
- `gaussianMixture$logLik()`
- `gaussianMixture$update_logLik()`
- `gaussianMixture$E_step()`
- `gaussianMixture$classify()`
- `gaussianMixture$fit()`
- `gaussianMixture$EM()`
- `gaussianMixture$update()`
- `gaussianMixture$filter()`

- `gaussianMixture$Hessian()`
- `gaussianMixture$compute_empiric_parameters()`
- `gaussianMixture$use_empiric_parameters()`
- `gaussianMixture$print()`
- `gaussianMixture$clone()`

Method `new()`: Initialize a gaussianMixture object

Usage:

```
gaussianMixture$new(components = 2, maxit = 500, abstol = 1e-09)
```

Arguments:

`components` Integer, number of components.

`maxit` `(integer(1))`

Numeric, maximum number of iterations.

`abstol` `(numeric(1))` Numeric, absolute level for convergence.

Method `logLik()`: Compute the log-likelihood

Usage:

```
gaussianMixture$logLik(x, params)
```

Arguments:

`x` vector

`params` Optional. Named list with mixture parameters.

Method `update_logLik()`: Update the log-likelihood with the current parameters

Usage:

```
gaussianMixture$update_logLik()
```

Method `E_step()`: Compute the posterior probabilities (E-step)

Usage:

```
gaussianMixture$E_step(x, params)
```

Arguments:

`x` vector

`params` a list of mixture parameters

Method `classify()`: Classify the time series in its components

Usage:

```
gaussianMixture$classify(x)
```

Arguments:

`x` vector

Method `fit()`: Fit the parameters with mclust package

Usage:

```
gaussianMixture$fit(x, weights)
```

Arguments:

`x` vector

`weights` observations weights, if a weight is equal to zero the observation is excluded, otherwise is included with unitary weight. When missing all the available observations will be used.

Method EM(): Fit the parameters with EM-algorithm

Usage:

```
gaussianMixture$EM(x, weights)
```

Arguments:

x vector

weights observations weights, if a weight is equal to zero the observation is excluded, otherwise is included with unitary weight. When missing all the available observations will be used.

Method update(): Update only the parameters (means, sd and p).

Usage:

```
gaussianMixture$update(means, sd, p)
```

Arguments:

means Numeric vector of means parameters.

sd Numeric vector of std. deviation parameters.

p Numeric vector of probability parameters.

Method filter(): Update the responsibilities, the log-likelihood, classify again the points and recompute empiric parameters.

Usage:

```
gaussianMixture$filter()
```

Method Hessian(): Hessian matrix gaussianMixture class.

Usage:

```
gaussianMixture$Hessian()
```

Method compute_empiric_parameters(): Compute the parameters on the classified time series.

Usage:

```
gaussianMixture$compute_empiric_parameters()
```

Method use_empiric_parameters(): Substitute the empiric parameters with EM parameters. If evaluated again the EM parameters will be substituted back.

Usage:

```
gaussianMixture$use_empiric_parameters()
```

Method print(): Print method for gaussianMixture class.

Usage:

```
gaussianMixture$print(label)
```

Arguments:

label Character, optional label.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
gaussianMixture$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Note

Version 1.0.0

Examples

```

means = c(0,0.5,2)
sd = rep(1, 3)
p = c(0.2, 0.3, 0.5)
# Grid
grid <- seq(-4, 4, 0.01)
plot(dmixnorm(grid, means, sd, p))
# Simulated sample
x <- rmixnorm(5000, means, sd, p)
# Gaussian mixture model
gm <- gaussianMixture$new(components=3)
# Fit the model
gm$fit(x$X)
# EM-algo
gm$EM(x$X)
# Model parameters
gm$coefficients
# Fitted series
gm$fitted
# Theoric moments
gm$moments
gm$update(means = c(-2, 0, 2))

```

havDistance

*Haversine distance***Description**

Compute the Haversine distance between two points.

Usage

```
havDistance(lat_1, lon_1, lat_2, lon_2)
```

Arguments

lat_1	numeric, latitude of first point.
lon_1	numeric, longitude of first point.
lat_2	numeric, latitude of second point.
lon_2	numeric, longitude of second point.

Value

Numeric vector the distance in kilometers.

Examples

```

havDistance(43.3, 12.1, 43.4, 12.2)
havDistance(43.35, 12.15, 43.4, 12.2)

```

IDW

*Inverse Distance Weighting Functions***Description**

Return a distance weighting function

Usage

```
IDW(beta, d0)
```

Arguments

beta	parameter used in exponential and power functions.
d0	parameter used only in exponential function.

Details

When the parameter d0 is not specified the function returned will be of power type otherwise of exponential type.

Examples

```
# Power weighting
IDW_pow <- IDW(2)
IDW_pow(c(2, 3,10))
IDW_pow(c(2, 3,10), normalize = TRUE)
# Exponential weighting
IDW_exp <- IDW(2, d0 = 5)
IDW_exp(c(2, 3,10))
IDW_exp(c(2, 3,10), normalize = TRUE)
```

is_leap_year

*Is leap year?***Description**

Check if a given year is leap (366 days) or not (365 days).

Usage

```
is_leap_year(x)
```

Arguments

x	numeric value or dates vector in the format YYYY-MM-DD.
---	---

Value

Boolean. TRUE if it is a leap year, FALSE otherwise.

Examples

```
is_leap_year("2024-02-01")
is_leap_year(c(2023:2030))
is_leap_year(c("2024-10-01", "2025-10-01"))
is_leap_year("2029-02-01")
```

kernelRegression	<i>Kernel regression</i>
------------------	--------------------------

Description

Kernel regression
Kernel regression

Details

Fit a kernel regression.

Active bindings

model an object of the class npreg.

Methods

Public methods:

- [kernelRegression\\$fit\(\)](#)
- [kernelRegression\\$predict\(\)](#)
- [kernelRegression\\$clone\(\)](#)

Method fit(): Fit a kernelRegression class

Usage:
kernelRegression\$fit(formula, data, ...)
Arguments:
formula formula, an object of class formula (or one that can be coerced to that class).
data an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
... other parameters to be passed to the function np::npreg.

Method predict(): Predict method for kernelRegression class

Usage:
kernelRegression\$predict(newdata)
Arguments:
newdata An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.

Method clone(): The objects of this class are cloneable with this method.

Usage:
kernelRegression\$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.

Note

Version 1.0.0

ks_test	<i>Kolmogorov Smirnov test for a distribution</i>
---------	---

Description

Test against a specific distribution

Usage

```
ks_test(  
  x,  
  cdf,  
  ci = 0.05,  
  min_quantile = 0.015,  
  max_quantile = 0.985,  
  k = 1000,  
  plot = FALSE  
)
```

Arguments

x	a vector.
ci	p.value for rejection.
min_quantile	minimum quantile for the grid of values.
max_quantile	maximum quantile for the grid of values.
k	finite value for approximation of infinite sum.
plot	when TRUE a plot is returned, otherwise a tibble.
pdf	a function. The theoretic density to use for comparison.
seed	random seed for two sample test.

Value

when plot = TRUE a plot is returned, otherwise a tibble.

ks_test_ts	<i>Two sample Kolmogorov Smirnov test for a time series</i>
------------	---

Description

Perform a two sample invariance test for a time series.

Usage

```
ks_test_ts(
  x,
  ci = 0.05,
  idx_split,
  min_quantile = 0.015,
  max_quantile = 0.985,
  seed = 1,
  plot = FALSE
)
```

Arguments

x	a vector.
ci	p.value for rejection.
idx_split	Index used for splitting the time series. If missing will be random sampled.
min_quantile	minimum quantile for the grid of values.
max_quantile	maximum quantile for the grid of values.
seed	random seed for two sample test.
plot	when TRUE a plot is returned, otherwise a tibble.

loss_dQdP	<i>Evaluate the loss for a set of times to maturity</i>
-----------	---

Description

Evaluate the loss for a set of times to maturity

Usage

```
loss_dQdP(lambda, data_dQdP, r = 0, nmonths = 1:12, quiet = FALSE)
```

loss_dQdP_tau	<i>Evaluate the loss for a specific time to maturity</i>
---------------	--

Description

Evaluate the loss for a specific time to maturity

Usage

```
loss_dQdP_tau(lambda, data, r = 0, nmonths = 1:12, quiet = FALSE)
```

makeSemiPositive	<i>Make a matrix positive semi-definite</i>
------------------	---

Description

Make a matrix positive semi-definite

Usage

```
makeSemiPositive(x, neg_values = 1e-05)
```

Arguments

x	matrix, squared and symmetric.
neg_values	numeric, the eigenvalues lower the zero will be substituted with this value.

Examples

```
m <- matrix(c(2, 2.99, 1.99, 2), nrow = 2, byrow = TRUE)
makeSemiPositive(m)
```

monthlyParams	<i>Create a function of time for monthly parameters</i>
---------------	---

Description

Create a function of time for monthly parameters
 Create a function of time for monthly parameters

Active bindings

parameters vector of parameters with length 12.

Methods

Public methods:

- `monthlyParams$new()`
- `monthlyParams$predict()`
- `monthlyParams$update()`
- `monthlyParams$clone()`

Method `new()`: Initialize a `monthlyParams` object

Usage:

```
monthlyParams$new(params)
```

Arguments:

`params` numeric vector of parameters with length 12.

Method `predict()`: Predict the monthly parameter

Usage:

```
monthlyParams$predict(x)
```

Arguments:

`x` date as character or month as numeric.

Method `update()`: Update the monthly parameters

Usage:

```
monthlyParams$update(params)
```

Arguments:

`params` numeric vector of parameters with length 12.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
monthlyParams$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

Examples

```
set.seed(1)
params <- runif(12)
mp <- monthlyParams$new(params)
t_now <- as.Date("2022-01-01")
t_hor <- as.Date("2024-12-31")
dates <- seq.Date(t_now, t_hor, by = "1 day")
plot(mp$predict(dates), type = "l")
```

mvgaussianMixture	<i>Multivariate gaussian mixture</i>
-------------------	--------------------------------------

Description

Multivariate gaussian mixture

Usage

```
mvgaussianMixture(
  x,
  means,
  sd,
  p,
  components = 2,
  maxit = 100,
  abstol = 1e-14,
  na.rm = FALSE
)
```

number_of_day	<i>Number of day</i>
---------------	----------------------

Description

Compute the number of day of the year given a vector of dates.

Usage

```
number_of_day(x)
```

Arguments

x dates vector in the format YYYY-MM-DD.

Value

Numeric vector with the number of the day during the year.

Examples

```
number_of_day("2040-01-31")
number_of_day(c("2015-03-31", "2016-03-31", "2017-03-31"))
number_of_day(c("2015-02-28", "2016-02-28", "2017-02-28"))
number_of_day(c("2015-03-01", "2016-03-01", "2017-03-01"))
```

optionPayoff	<i>Option payoff function</i>
--------------	-------------------------------

Description

Compute the payoffs of an option at maturity.

Usage

```
optionPayoff(x, strike = 0, c0 = 0, put = TRUE)
```

Arguments

x	numeric, vector of values at maturity.
strike	numeric, option strike.
put	logical, when TRUE, the default, the payoff function is a put otherwise a call.
v0	numeric, price of the option.

Examples

```
optionPayoff(10, 9, 1, put = TRUE)
mean(optionPayoff(seq(0, 20), 9, 1, put = TRUE))
```

PDF	<i>Density, distribution and quantile function</i>
-----	--

Description

Return a function of x given the specification of a function of x.

Usage

```
PDF(.f, ...)
```

```
CDF(.f, lower = -Inf, ...)
```

```
Quantile(cdf, interval = c(-100, 100))
```

Arguments

.f	density function
...	other parameters to be passed to .f.
lower	lower bound for integration (CDF).
cdf	cumulative distribution function.
interval	lower and upper bounds for unit root (Quantile).

Examples

```
# Density
pdf <- PDF(dnorm, mean = 0.3, sd = 1.3)
pdf(3)
dnorm(3, mean = 0.3, sd = 1.3)
# Distribution
cdf <- CDF(dnorm, mean = 0.3, sd = 1.3)
cdf(3)
pnorm(3, mean = 0.3, sd = 1.3)
# Numeric quantile function
pnorm(Quantile(pnorm)(0.9))
```

```
plot.solarModelForecast
```

Plot a forecast from a solarModel object

Description

Plot a forecast from a solarModel object

Usage

```
## S3 method for class 'solarModelForecast'
plot(object, type = "mix")
```

Examples

```
model <- Bologna
df_n <- model$moments$conditiona[23,]
plot(predict.solarModel(model, df_n, ci = 0.01))
```

```
plot.solarScenario
```

Plot scenarios from a solarScenario object

Description

Plot scenarios from a solarScenario object

Usage

```
## S3 method for class 'solarScenario'
plot(x, target = "GHI", nsim = 10, empiric = TRUE, ci = 0.05)
```

Examples

```
model <- Bologna
scenario <- solarScenario(model, nsim = 70)
plot(scenario)
```

pow_matrix	<i>Power of a matrix</i>
------------	--------------------------

Description

Compute the power of a matrix

Usage

```
pow_matrix(x, n = 0)
```

Arguments

x	Matrix
n	power

predict.solarModel	<i>Produce a forecast from a solarModel object</i>
--------------------	--

Description

Produce a forecast from a solarModel object

Usage

```
## S3 method for class 'solarModel'
predict(model, moments, theta = 0, ci = 0.01)
```

Examples

```
model <- Bologna
moments <- model$moments$conditional[14,]
object <- predict.solarModel(model, moments, ci = 0.01)
object
```

prepare_dQdP_calibration	<i>Preprocess the moments for a specific set of time to maturities to avoid recomputing them each time</i>
--------------------------	--

Description

Preprocess the moments for a specific set of time to maturities to avoid recomputing them each time

Usage

```
prepare_dQdP_calibration(tau = c(10, 40, 50), model_Rt)
```

```
prepare_dQdP_calibration_tau
```

Preprocess the moments for a specific tau to avoid recomputing them each time

Description

Preprocess the moments for a specific tau to avoid recomputing them each time

Usage

```
prepare_dQdP_calibration_tau(tau = 10, model_Rt)
```

```
radiationModel
```

Radiation model

Description

Radiation model

Radiation model

Public fields

theta Numeric, mean reversion parameter.

lambda_S Numeric, market risk premium Q-measure.

Active bindings

model model

measure Character, reference probability measure actually used.

lambda Numeric, market risk premium actually used.

Methods

Public methods:

- `radiationModel$new()`
- `radiationModel$change_measure()`
- `radiationModel$Ct()`
- `radiationModel$Yt_bar()`
- `radiationModel$Rt_bar()`
- `radiationModel$sigma_bar()`
- `radiationModel$mu_B()`
- `radiationModel$sigma_B()`
- `radiationModel$mu_Y()`
- `radiationModel$sigma_Y()`
- `radiationModel$mu_R()`
- `radiationModel$sigma_R()`

- radiationModel\$integral_expectation()
- radiationModel\$integral_variance()
- radiationModel\$e_mix_drift()
- radiationModel\$e_mix_diffusion()
- radiationModel\$M_Y()
- radiationModel\$S_Y()
- radiationModel\$pdf_Y()
- radiationModel\$cdf_Y()
- radiationModel\$pdf_R()
- radiationModel\$cdf_R()
- radiationModel\$e_GHI()
- radiationModel\$v_GHI()
- radiationModel\$print()
- radiationModel\$clone()

Method new(): Initialize a radiationModel object

Usage:

radiationModel\$new(model)

Arguments:

model solarModel model fitted

Method change_measure(): Change the reference probability measure

Usage:

radiationModel\$change_measure(measure)

Arguments:

measure Character, probability measure. Can be P or Q.

Method Ct(): Clear sky radiation for a day of the year.

Usage:

radiationModel\$Ct(t_now)

Arguments:

t_now Character, today date.

Returns: Clear sky radiation on date t_now.

Method Yt_bar(): Seasonal mean for the transformed variable Yt for a given day of the year.

Usage:

radiationModel\$Yt_bar(t_now)

Arguments:

t_now Character, today date.

Returns: Seasonal mean for Yt on date t_now.

Method Rt_bar(): Compute the seasonal mean for the solar radiation for a given day of the year.

Usage:

radiationModel\$Rt_bar(t_now)

Arguments:

`t_now` Character, today date.

Returns: Seasonal mean for R_t .

Method `sigma_bar()`: Instantaneous seasonal variance for the transformed variable for a given day of the year.

Usage:

```
radiationModel$sigma_bar(t_now)
```

Arguments:

`t_now` Character, today date.

Returns: Seasonal std. deviation for Y_t on date `t_now`.

Method `mu_B()`: Return the mixture drift if B is specified, otherwise it return the average drift.

Usage:

```
radiationModel$mu_B(t_now, B = 1)
```

Arguments:

`t_now` Character, today date.

B Integer, 1 for the first component, 0 for the second.

Returns: Mixture seasonal drift for Y_t on date `t_now`.

Method `sigma_B()`: Return the mixture diffusion with seasonal jump.

Usage:

```
radiationModel$sigma_B(t_now, B)
```

Arguments:

`t_now` Character, today date.

B Integer, 1 for the first component, 0 for the second.

Returns: Mixture seasonal diffusion for Y_t .

Method `mu_Y()`: Return the drift for the transformed variable Y_t .

Usage:

```
radiationModel$mu_Y(Yt, t_now, B = 1)
```

Arguments:

Y_t Numeric, transformed solar radiation.

`t_now` Character, today date.

B Integer, 1 for the first component, 0 for the second.

Returns: Mixture drift for Y_t .

Method `sigma_Y()`: Return the diffusion for solar radiation process

Usage:

```
radiationModel$sigma_Y(t_now, B = 1)
```

Arguments:

`t_now` Character, today date.

B Integer, 1 for the first component, 0 for the second.

R_t Numeric, solar radiation.

Returns: Diffusion for R_t .

Method `mu_R()`: Return the drift for solar radiation process

Usage:

```
radiationModel$mu_R(Rt, t_now, B = 1, dt = 1)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`B` Integer, 1 for the first component, 0 for the second.

`dt` Numeric, time step.

Returns: Drift for `Rt`.

Method `sigma_R()`: Return the diffusion for solar radiation process

Usage:

```
radiationModel$sigma_R(Rt, t_now, B = 1)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`B` Integer, 1 for the first component, 0 for the second.

Returns: Diffusion for `Rt`.

Method `integral_expectation()`: Compute the integral for expectation for constant mixture parameters

Usage:

```
radiationModel$integral_expectation(t_now, t_hor, df_date, last_day = TRUE)
```

Arguments:

`t_now` Character, today date.

`t_hor` Character, horizon date.

`df_date` Optional dataframe. See [create_monthly_sequence](#) for more details.

`last_day` Logical. When TRUE the last day will be treated as conditional variance otherwise not.

Method `integral_variance()`: Compute the integral for variance for constant mixture parameters

Usage:

```
radiationModel$integral_variance(t_now, t_hor, df_date, last_day = TRUE)
```

Arguments:

`t_now` Character, today date.

`t_hor` Character, horizon date.

`df_date` Optional dataframe. See [create_monthly_sequence](#) for more details.

`last_day` Logical. When TRUE the last day will be treated as conditional variance otherwise not.

Method `e_mix_drift()`: Return the value of the mixture drift of both component of `Yt`.

Usage:

```
radiationModel$e_mix_drift(t_now, t_hor, df_date)
```

Arguments:

t_now Character, today date.
 t_hor Character, horizon date.
 df_date Optional dataframe. See [create_monthly_sequence](#) for more details.
Returns: Mixture expected value for both component of Yt.

Method e_mix_diffusion(): Return the value of the mixture drift of both component of Yt.

Usage:
 radiationModel\$e_mix_diffusion(t_now, t_hor, df_date)

Arguments:
 t_now Character, today date.
 t_hor Character, horizon date.
 df_date Optional dataframe. See [create_monthly_sequence](#) for more details.
Returns: Mixture expected value for both component of Yt.

Method M_Y(): Return the conditional expectation for Yn for YN.

Usage:
 radiationModel\$M_Y(Rt, t_now, t_hor, df_date)
Arguments:
 Rt Numeric, solar radiation.
 t_now Character, today date.
 t_hor Character, horizon date.
 df_date Optional dataframe. See [create_monthly_sequence](#) for more details.
Returns: Conditional mean for Yt

Method S_Y(): Return the conditional variance for Yn for YN.

Usage:
 radiationModel\$S_Y(t_now, t_hor, df_date)
Arguments:
 t_now Character, today date.
 t_hor Character, horizon date.
 df_date Optional dataframe. See [create_monthly_sequence](#) for more details.
 Rt Numeric, solar radiation.
Returns: Conditional variance for Yt

Method pdf_Y(): Return the conditional density for Y_N given Yn.

Usage:
 radiationModel\$pdf_Y(Rt, t_now, t_hor, B)
Arguments:
 Rt Numeric, solar radiation.
 t_now Character, today date.
 t_hor Character, horizon date.
 B Integer, mixture component, if B is missing will be returned the mixture density, otherwise the component density non weighted.
Returns: Conditional density for Y_N

Method `cdf_Y()`: Return the conditional distribution for Y_N given Y_n .

Usage:

```
radiationModel$cdf_Y(Rt, t_now, t_hor, B)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`t_hor` Character, horizon date.

`B` Integer, mixture component, if `B` is missing will be returned the mixture distribution, otherwise the component distribution non weighted.

Returns: Conditional distribution for Y_N

Method `pdf_R()`: Return the conditional density for R_N given R_n .

Usage:

```
radiationModel$pdf_R(Rt, t_now, t_hor, B)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`t_hor` Character, horizon date.

`B` Integer, mixture component, if `B` is missing will be returned the mixture density, otherwise the component density non weighted.

Returns: Conditional density for R_N

Method `cdf_R()`: Return the conditional distribution for R_N given R_n .

Usage:

```
radiationModel$cdf_R(Rt, t_now, t_hor, B)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`t_hor` Character, horizon date.

`B` Integer, mixture component, if `B` is missing will be returned the mixture distribution, otherwise the component distribution non weighted.

Returns: Conditional distribution for R_N

Method `e_GHI()`: Return the conditional expected value for R_N given R_n .

Usage:

```
radiationModel$e_GHI(Rt, t_now, t_hor, B, moment = 1)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`t_hor` Character, horizon date.

`B` Integer, mixture component, if `B` is missing will be returned the mixture density, otherwise the component density non weighted.

`moment` Integer, scalar. Moment order. The default is 1, i.e. the expectation.

Returns: Conditional moment for solar radiation

Method `v_GHI()`: Return the conditional variance value for `R_N` given `Rn`.

Usage:

```
radiationModel$v_GHI(Rt, t_now, t_hor, B)
```

Arguments:

`Rt` Numeric, solar radiation.

`t_now` Character, today date.

`t_hor` Character, horizon date.

`B` Integer, mixture component, if `B` is missing will be returned the mixture density, otherwise the component density non weighted.

Returns: Conditional variance for `R_N`

Method `print()`: Method print

Usage:

```
radiationModel$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
radiationModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

riccati_root

Riccati Root

Description

Compute the square root of a symmetric matrix.

Usage

```
riccati_root(x)
```

Arguments

`x` squared and symmetric matrix.

Examples

```
cv <- matrix(c(1, 0.3, 0.3, 1), nrow = 2, byrow = TRUE)
riccati_root(cv)
```

seasonalClearsky	<i>Clear sky seasonal model</i>
------------------	---------------------------------

Description

Clear sky seasonal model

Clear sky seasonal model

Super class

`solarrr::seasonalModel` -> seasonalClearsky

Public fields

`control` Named list. Control parameters. See the function `control_seasonalClearsky` for details.

`lat` Numeric, scalar. Latitude of the location considered.

Methods

Public methods:

- `seasonalClearsky$new()`
- `seasonalClearsky$fit()`
- `seasonalClearsky$H0()`
- `seasonalClearsky$predict()`
- `seasonalClearsky$print()`
- `seasonalClearsky$clone()`

Method `new()`: Initialize a seasonalClearsky object.

Usage:

```
seasonalClearsky$new(control = control_seasonalClearsky())
```

Arguments:

`control` Named list. Control parameters. See the function `control_seasonalClearsky` for details.

Method `fit()`: Fit the seasonal model for clear sky radiation.

Usage:

```
seasonalClearsky$fit(x, date, lat, clearsky)
```

Arguments:

`x` Numeric vector. Time series of solar radiation.

`date` Character or Date vector. Time series of dates.

`lat` Numeric scalar. Reference latitude.

`clearsky` Numeric vector. Time series of CAMS clear sky radiation.

Method `H0()`: Compute the extraterrestrial radiation at a given location.

Usage:

```
seasonalClearsky$H0(n)
```


Arguments:

n Integer, scalar or vector. Number of day of the year.

Method `predict()`: Predict method for seasonalClearsky object.

Usage:

```
seasonalClearsky$predict(n)
```

Arguments:

n Integer, scalar or vector. number of day of the year.

Method `print()`: Print method for seasonalClearsky object.

Usage:

```
seasonalClearsky$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
seasonalClearsky$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
library(ggplot2)
# Arguments
place <- "Palermo"
# solarModel specification
spec <- solarModel_spec(place, target = "GHI")
# Extract the required elements
x <- spec$data$GHI
date <- spec$data$date
lat <- spec$coords$lat
clearsky <- spec$data$clearsky

# Initialize the model
model <- seasonalClearsky$new()
# Fit the model
model$fit(x, date, lat, clearsky)
# Predict the seasonal values
spec$data$Ct <- model$predict(spec$data$n)
```

Description

The `seasonalModel` class implements a seasonal regression model as a linear combination of sine and cosine functions. This model is designed to capture periodic effects in time series data, particularly for applications involving seasonal trends.

Details

The seasonal model is fitted using a specified formula, which allows for the inclusion of external regressors along with sine and cosine terms to model seasonal variations. The periodicity can be customized, and the model can be updated with new coefficients after the initial fit.

Public fields

`extra_params` List to contain custom extra parameters.

Active bindings

`coefficients` A named vector with the fitted coefficients.

`coefficients2` A named vector with the coefficients reparametrized to obtain a linear combination of only shifted sine functions.

`model` A slot with the fitted `lm` object.

`period` Integer, the seasonality period.

`order` Integer, number of combinations of sines and cosines.

Methods

Public methods:

- `seasonalModel$new()`
- `seasonalModel$fit()`
- `seasonalModel$predict()`
- `seasonalModel$differential()`
- `seasonalModel$update()`
- `seasonalModel$print()`
- `seasonalModel$clone()`

Method `new()`: Initialize a `seasonalModel` object.

Usage:

```
seasonalModel$new(order = 1, period = 365)
```

Arguments:

`order` Integer, number of combinations of sines and cosines.

`period` Integer, seasonality period. The default is 365.

Method `fit()`: Fit a seasonal model as a linear combination of sine and cosine functions and eventual external regressors specified in the formula. The external regressors used should have the same periodicity, i.e. not stochastic regressors are allowed.

Usage:

```
seasonalModel$fit(formula, data, ...)
```

Arguments:

`formula` formula, an object of class `formula` (or one that can be coerced to that class). It is a symbolic description of the model to be fitted and can be used to include or exclude the intercept or external regressors in data.

`data` an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in data, the variables are taken from `environment(formula)`, typically the environment from which `lm` is called.

... other parameters to be passed to the function `lm`.

Method `predict()`: Predict method for the class `seasonalModel`.

Usage:

```
seasonalModel$predict(n, newdata, dt = 1)
```

Arguments:

`n` integer, number of day of the year.

`newdata` An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.

`dt` Numeric, time step.

Method `differential()`: Compute the differential of the sinusoidal function. It do not consider the differential of eventual external regressors.

Usage:

```
seasonalModel$differential(n, newdata, dt = 1)
```

Arguments:

`n` Integer, number of day of the year.

`newdata` An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.

`dt` Numeric, time step.

Method `update()`: Update the parameters inside the model.

Usage:

```
seasonalModel$update(coefficients)
```

Arguments:

`coefficients` A named vector with coefficients.

Method `print()`: Print method for the class `seasonalModel`.

Usage:

```
seasonalModel$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
seasonalModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

seasonalSolarFunctions

Solar seasonal functions

Description

Solar seasonal functions

Solar seasonal functions

Public fields

legal_hour Logical, when TRUE the clock time will be corrected for the legal hour.

Active bindings

G0 solar constant, i.e, 1367.

Methods

Public methods:

- [seasonalSolarFunctions\\$new\(\)](#)
- [seasonalSolarFunctions\\$update_method\(\)](#)
- [seasonalSolarFunctions\\$B\(\)](#)
- [seasonalSolarFunctions\\$degree\(\)](#)
- [seasonalSolarFunctions\\$radiant\(\)](#)
- [seasonalSolarFunctions\\$E\(\)](#)
- [seasonalSolarFunctions\\$solar_time\(\)](#)
- [seasonalSolarFunctions\\$hour_angle\(\)](#)
- [seasonalSolarFunctions\\$incidence_angle\(\)](#)
- [seasonalSolarFunctions\\$azimut_angle\(\)](#)
- [seasonalSolarFunctions\\$G0n\(\)](#)
- [seasonalSolarFunctions\\$declination\(\)](#)
- [seasonalSolarFunctions\\$H0\(\)](#)
- [seasonalSolarFunctions\\$sunset_hour_angle\(\)](#)
- [seasonalSolarFunctions\\$sun_hours\(\)](#)
- [seasonalSolarFunctions\\$solar_altitude\(\)](#)
- [seasonalSolarFunctions\\$solar_angles\(\)](#)
- [seasonalSolarFunctions\\$clearsky\(\)](#)
- [seasonalSolarFunctions\\$clone\(\)](#)

Method new(): Initialize a seasonalSolarFunctions object

Usage:

```
seasonalSolarFunctions$new(method = "spencer", legal_hour = TRUE)
```

Arguments:

method character, method type for computations. Can be cooper or spencer.

legal_hour Logical, when TRUE the clock time will be corrected for the legal hour.

Method `update_method()`: Extract or update the method used for computations.

Usage:

`seasonalSolarFunctions$update_method(x)`

Arguments:

`x` character, method type. Can be cooper or spencer.

Returns: When `x` is missing it return a character containing the method that is actually used.

Method `B()`: Seasonal adjustment parameter.

Usage:

`seasonalSolarFunctions$B(n)`

Arguments:

`n` number of the day of the year

Details: The function computes

$$B(n) = \frac{2\pi}{365}n$$

Method `degree()`: Convert angles in radian into an angles in degrees.

Usage:

`seasonalSolarFunctions$degree(x)`

Arguments:

`x` numeric vector, angles in radian.

Details: The function computes:

$$\frac{x180}{\pi}$$

Method `radiant()`: Convert angles in degrees into an angles in radian

Usage:

`seasonalSolarFunctions$radiant(x)`

Arguments:

`x` numeric vector, angles in degrees.

Details: The function computes:

$$\frac{x\pi}{180}$$

Method `E()`: Compute the time adjustment in minutes.

Usage:

`seasonalSolarFunctions$E(n)`

Arguments:

`n` number of the day of the year

Details: The function implement Eq. 1.5.3 from Duffie (4th edition), i.e.

$$E = 229.2(0.000075 + 0.001868 \cos(B) - 0.032077 \sin(B) - 0.014615 \cos(2B) - 0.04089 \sin(2B))$$

Returns: The time adjustment in minutes.

Method `solar_time()`: Compute the solar time from a clock time.

Usage:

`seasonalSolarFunctions$solar_time(x, lon, lon_sd = 15)`

Arguments:

x datetime, clock hour.

lon longitude of interest in degrees.

lon_sd longitude of the Local standard meridian in degrees.

Details: The function implement Eq. 1.5.2 from Duffie (4th edition), i.e.

$$solartime = clocktime + 4(lon_s - lon) + E(n)$$

Returns: A datetime object

Method hour_angle(): Compute the solar angle for a specific hour of the day.

Usage:

```
seasonalSolarFunctions$hour_angle(x, lon, lon_sd = 15)
```

Arguments:

x datetime, clock hour.

lon longitude of interest in degrees.

lon_sd longitude of the Local standard meridian in degrees.

Returns: An angle in degrees

Method incidence_angle(): Compute the incidence angle

Usage:

```
seasonalSolarFunctions$incidence_angle(
  x,
  lat,
  lon,
  lon_sd = 15,
  beta = 0,
  gamma = 0
)
```

Arguments:

x datetime, clock hour.

lat latitude of interest in degrees.

lon longitude of interest in degrees.

lon_sd longitude of the Local standard meridian in degrees.

beta altitude

gamma orientation

Returns: An angle in degrees

Method azimuth_angle(): Compute the solar azimuth angle for a specific time of the day.

Usage:

```
seasonalSolarFunctions$azimut_angle(x, lat, lon, lon_sd = 15)
```

Arguments:

x datetime, clock hour.

lat latitude of interest in degrees.

lon longitude of interest in degrees.

lon_sd longitude of the Local standard meridian in degrees.

Details: The function implement Eq. 1.6.6 from Duffie (4th edition), i.e.

$$\gamma_s = \text{sign}(\omega) \left| \cos^{-1} \left(\frac{\cos \theta_z \sin \phi - \sin \delta}{\sin \theta_z \cos \phi} \right) \right|$$

Returns: The solar azimuth angle in degrees

Method G0n(): Compute the solar constant adjusted for the day of the year.

Usage:

seasonalSolarFunctions\$G0n(n)

Arguments:

n number of the day of the year

Details: When method is cooper the function implement Eq. 1.4.1a from Duffie (4th edition), i.e.

$$G_{0,n} = G_0(1 + 0.033 \cos(B))$$

otherwise when it is spencer it implement Eq. 1.4.1b from Duffie (4th edition):

$$G_{0,n} = G_0(1.000110 + 0.034221 \cos(B) + 0.001280 \sin(B) + 0.000719 \cos(2B) + 0.000077 \sin(2B))$$

Returns: The solar constant in W/m^2 for the day n.

Method declination(): Compute solar declination in degrees.

Usage:

seasonalSolarFunctions\$declination(n)

Arguments:

n number of the day of the year

Details: When method is cooper the function implement Eq. 1.6.1a from Duffie (4th edition), i.e.

$$\delta(n) = 23.45 \sin \left(\frac{2\pi(284 + n)}{365} \right)$$

otherwise when it is spencer it implement Eq. 1.6.1b from Duffie (4th edition):

$$\delta(n) = \frac{180}{\pi} (0.006918 - 0.399912 \cos(B) + 0.070257 \sin(B) - 0.006758 \cos(2B))$$

Returns: The solar declination in degrees.

Method H0(): Compute the solar extraterrestrial radiation

Usage:

seasonalSolarFunctions\$H0(n, lat)

Arguments:

n number of the day of the year

lat latitude of interest in degrees.

Returns: Extraterrestrial radiation on an horizontal surface in kilowatt hour for metres squared for day.

Method sunset_hour_angle(): Compute solar angle at sunset in degrees

Usage:

seasonalSolarFunctions\$sunset_hour_angle(n, lat)

Arguments:

n number of the day of the year

lat Numeric, latitude of interest in degrees.

Details: The function implement Eq. 1.6.10 from Duffie (4th edition), i.e.

$$\omega_s = \cos^{-1}(-\tan(\delta(n)) \tan(\phi))$$

Returns: The sunset hour angle in degrees.

Method sun_hours(): Compute number of sun hours for a day n.

Usage:

```
seasonalSolarFunctions$sun_hours(n, lat)
```

Arguments:

n number of the day of the year.

lat Numeric, latitude of interest in degrees.

Details: The function implement Eq. 1.6.11 from Duffie (4th edition), i.e.

$$\frac{2}{15} \omega_s$$

Method solar_altitude(): Compute solar altitude in degrees

Usage:

```
seasonalSolarFunctions$solar_altitude(n, lat)
```

Arguments:

n number of the day of the year

lat Numeric, latitude of interest in degrees.

Details: The function computes

$$\sin^{-1}(-\sin(\delta(n)) \sin(\phi) + \cos(\delta(n)) \cos(\phi))$$

Method solar_angles(): Compute the solar angle for a latitude in different dates.

Usage:

```
seasonalSolarFunctions$solar_angles(x, lat, lon, lon_sd, by = "1 min")
```

Arguments:

x datetime, clock hour.

lat Numeric, latitude of interest in degrees.

lon Numeric, longitude of interest in degrees.

lon_sd Numeric, longitude of the Local standard meridian in degrees.

by Character, time step. Default is 1 min.

Method clearsky(): Hottel clearsky

Usage:

```
seasonalSolarFunctions$clearsky(
  cosZ = NULL,
  G0 = NULL,
  altitude = 2.5,
  clime = "No Correction"
)
```

Arguments:

cosZ solar incidence angle
G0 solar constant
altitude altitude in km
clime clime correction

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
seasonalSolarFunctions$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Note

Version 1.0.0

References

Duffie, Solar Engineering of Thermal Processes Fourth Edition.

Examples

```
dates <- seq.Date(as.Date("2022-01-01"), as.Date("2022-12-31"), 1)
# Seasonal functions object
sf <- seasonalSolarFunctions$new()

# Adjustment parameter
sf$B(number_of_day(dates))

# Time adjustment in minutes
sf$E(dates)

# Declination
sf$declination(dates)

# Solar constant
sf$G0

# Solar constant adjusted
sf$G0n(dates)

# Extraterrestrial radiation
sf$H0(dates, 43)

# Number of hours of sun
sf$sun_hours(dates, 43)

# Sunset hour angle
sf$sunset_hour_angle(dates, 43)
```

solarEsscher

*Function to Calibrate Esscher Bounds and parameters***Description**

Function to Calibrate Esscher Bounds and parameters

Function to Calibrate Esscher Bounds and parameters

Public fields

control list containing the control parameters

grid list containing the grids

Active bindings

bounds calibrated bounds with respect to bootstrapped payoff.

model Models to predict the optimal theta given the expected return.

Methods**Public methods:**

- [solarEsscher\\$new\(\)](#)
- [solarEsscher\\$calibrator\(\)](#)
- [solarEsscher\\$calibrate_bounds\(\)](#)
- [solarEsscher\\$create_grid\(\)](#)
- [solarEsscher\\$predict\(\)](#)
- [solarEsscher\\$clone\(\)](#)

Method new(): Initialize the settings for calibration of Esscher parameter.*Usage:*

```
solarEsscher$new(
  n_key_points = 15,
  init_lambda = 0,
  lower_lambda = -1,
  upper_lambda = 1,
  put = TRUE,
  quiet = FALSE,
  control_options = control_solarOption()
)
```

Arguments:

n_key_points integer, number of key points used to create the grid for interpolation.

init_lambda numeric, initial value for the Esscher parameter.

lower_lambda numeric, lower value for the Esscher parameter.

upper_lambda numeric, upper value for the Esscher parameter.

put logical, when TRUE will be considered a put contract otherwise a call contract.

quiet logical

control_options control function. See [control_solarOption](#) for details.

Method `calibrator()`: Calibrate the optimal Esscher parameter given a target price

Usage:

```
solarEsscher$calibrator(model, target_price, nmonths = 1:12)
```

Arguments:

`model` solar model

`target_price` the target_price represent the model price under the target Q-measure.

`nmonths` month or months

Method `calibrate_bounds()`: Calibrate Esscher upper and lower bounds

Usage:

```
solarEsscher$calibrate_bounds(model, payoffs)
```

Arguments:

`model` object with the class `solarModel`. See the function `solarModel` for details.

`payoffs` object with the class `solarOptionPayoffs`. See the function `solarOptionPayoffs` for details.

Method `create_grid()`: Create a grid of optimal theta and expected returns with respect of the benchmark price. Fit the model to predict the optimal Esscher parameters given the grid.

Usage:

```
solarEsscher$create_grid(model, benchmark_price, lower_price, upper_price)
```

Arguments:

`model` object with the class `solarModel`. See the function `solarModel` for details.

`benchmark_price` benchmark price for an expected return equal to zero.

`lower_price` lower price in the grid.

`upper_price` upper price in the grid.

Method `predict()`: Predict the optimal Esscher parameters given a certain level of expected return.

Usage:

```
solarEsscher$predict(r)
```

Arguments:

`r` expected return

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
solarEsscher$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

solarEsscher_probability

Change probability according to Esscher parameters

Description

Change probability according to Esscher parameters

Usage

```
solarEsscher_probability(params = c(0, 0, 1, 1, 0.5), df_n, theta = 0)
```

solarMixture

*Monthly Gaussian mixture with two components***Description**

Monthly Gaussian mixture with two components

Monthly Gaussian mixture with two components

Public fields

maxit Integer, maximum number of iterations.

abstol Numeric, absolute level for convergence.

components Integer, number of components.

mu1 Function, see [monthlyParams](#).mu2 Function, see [monthlyParams](#).sd1 Function, see [monthlyParams](#).sd2 Function, see [monthlyParams](#).prob Function, see [monthlyParams](#).**Active bindings**

data A tibble with the following columns:

date Time series of dates.**Month** Vector of Month.**x** Time series used for fitting.**w** Time series of weights.

means Matrix of means where a row represents a month and a column a mixture component.

sd Matrix of std. deviations where a row represents a month and a column a mixture component.

p Matrix of probabilities where a row represents a month and a column a mixture component.

model Named List with 12 [gaussianMixture](#) objects.

loglik Numeric, total log-likelihood.

fitted A tibble with the classified series

moments A tibble with the theoric moments. It contains:

Month Month of the year.**mean** Theoric monthly expected value of the mixture model.**variance** Theoric monthly variance of the mixture model.**skewness** Theoric monthly skewness.**kurtosis** Theoric monthly kurtosis.**nobs** Number of observations used for fitting.**loglik** Monthly log-likelihood.

coefficients A tibble with the fitted parameters.

std.errors A tibble with the fitted std.errors

summary A tibble with the fitted std.errors

Methods

Public methods:

- `solarMixture$new()`
- `solarMixture$fit()`
- `solarMixture$update()`
- `solarMixture$filter()`
- `solarMixture$hessian()`
- `solarMixture$update_logLik()`
- `solarMixture$empiric_parameters()`
- `solarMixture$print()`
- `solarMixture$clone()`

Method `new()`: Initialize a `solarMixture` object

Usage:

```
solarMixture$new(components = 2, maxit = 100, abstol = 1e-14)
```

Arguments:

`components` Integer, number of components.

`maxit` Integer, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

Method `fit()`: Fit the parameters with `mclust` package

Usage:

```
solarMixture$fit(x, date, weights)
```

Arguments:

`x` vector

`date` date vector

`weights` observations weights, if a weight is equal to zero the observation is excluded, otherwise is included with unitary weight. When missing all the available observations will be used.

Method `update()`: Update means, sd, p and recompute log-likelihood and fitted data.

Usage:

```
solarMixture$update(means, sd, p)
```

Arguments:

`means` Numeric matrix of means parameters.

`sd` Numeric matrix of std. deviation parameters.

`p` Numeric matrix of probability parameters.

Method `filter()`: Filter the gaussianMixture models

Usage:

```
solarMixture$filter()
```

Method `Hessian()`: Compute the hessian of each gaussianMixture model

Usage:

```
solarMixture$hessian()
```

Method `update_logLik()`: Update the log-likelihoods of each gaussianMixture model.

Usage:

```
solarMixture$update_logLik()
```

Method `empiric_parameters()`: Substitute the empiric parameters with EM parameters. If evaluated again the EM parameters will be substituted back.

Usage:

```
solarMixture$empiric_parameters()
```

Method `print()`: Print method for solarMixture class.

Usage:

```
solarMixture$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
solarMixture$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

Examples

```
mode <- Bologna
x <- model$data$u_tilde
w <- model$data$weights
date <- model$data$date

sm <- solarMixture$new()
sm$fit(x, date, w)
params <- sm$parameters
sm$std.errors
# params[1,]$mu1 <- params[1,]$mu1*0.9
# sm$update(means = params[,c(2,3)])
```

solarModel

Solar Model in R6 Class

Description

The `solarModel` class allows for the step-by-step fitting and transformation of solar radiation data, from clear sky models to GARCH models for residual analysis. It utilizes various private and public methods to fit the seasonal clearsky model, compute risk drivers, detect outliers, and apply time-series models.

Details

The `solarModel` class is an implementation of a comprehensive solar model that includes fitting seasonal models, detecting outliers, performing transformations, and applying time-series models such as AR and GARCH. This model is specifically designed to predict solar radiation data, and it uses seasonal and Gaussian Mixture models to capture the underlying data behavior.

Public fields

place Character, optional name of the location considered.

target Character, name of the target variable to model. Can be "GHI" or "clearsky".

dates A named list, with the range of dates used in the model. Output of [solarModel_spec](#).

coords A named list with the coordinates of the location considered. Contains:

- lat** Numeric, reference latitude in degrees.
- lon** Numeric, reference longitude in degrees.
- alt** Numeric, reference altitude in metres.

Active bindings

data A data frame with the fitted data, and the seasonal and monthly parameters.

seasonal_data A data frame containing seasonal and monthly parameters.

monthly_data A data frame that contains monthly parameters.

loglik The log-likelihood computed on train data.

control A list of control parameters that govern the behavior of the model's fitting process.

location A data frame with coordinates of the location considered.

transform A [solarTransform](#) object with the transformation functions applied to the data.

seasonal_model_Ct The fitted model for clear sky radiation, used for predict the maximum radiation available.

seasonal_model_Yt The fitted seasonal model for the target variable.

ARMA The fitted ARMA model for the target variable.

seasonal_variance The fitted model for seasonal variance.

GARCH A model object representing the GARCH model fitted to the residuals.

NM_model A model object representing the Gaussian Mixture model fitted to the standardized residuals.

moments Get a list containing the conditional and unconditional moments.

coefficients Get the model parameters as a named list.

Methods**Public methods:**

- [solarModel\\$new\(\)](#)
- [solarModel\\$fit\(\)](#)
- [solarModel\\$update\(\)](#)
- [solarModel\\$filter\(\)](#)
- [solarModel\\$Moments\(\)](#)
- [solarModel\\$VaR\(\)](#)
- [solarModel\\$logLik\(\)](#)
- [solarModel\\$R_to_Y\(\)](#)
- [solarModel\\$Y_to_R\(\)](#)
- [solarModel\\$print\(\)](#)
- [solarModel\\$fit_clearsky_model\(\)](#)
- [solarModel\\$compute_risk_drivers\(\)](#)

- `solarModel$fit_solar_transform()`
- `solarModel$detect_outliers_Yt()`
- `solarModel$fit_seasonal_mean()`
- `solarModel$correct_monthly_mean()`
- `solarModel$fit_ARMA()`
- `solarModel$fit_seasonal_variance()`
- `solarModel$correct_monthly_variance()`
- `solarModel$correct_seasonal_variance()`
- `solarModel$fit_GARCH_model()`
- `solarModel$fit_mixture_model()`
- `solarModel$clone()`

Method `new()`: Initialize a `solarModel`

Usage:

`solarModel$new(spec)`

Arguments:

`spec` an object with class `solarModelSpec`. See the function `solarModel_spec` for details.

Method `fit()`: Initialize and fit a `solarModel` object given the specification contained in `$control`.

Usage:

`solarModel$fit()`

Method `update()`: Update the parameters inside object

Usage:

`solarModel$update(params)`

Arguments:

`params` List of parameters. See the slot `$coefficients` for a template.

Method `filter()`: Filter the time series when new parameters are supplied in the method `$update(params)`.

Usage:

`solarModel$filter()`

Returns: Update the slots `$data`, `$seasonal_data`, `$monthly_data`, `$moments$conditional`, `$moments$unconditional` and `$loglik`.

Method `Moments()`: Compute the conditional moments

Usage:

`solarModel$Moments(t_now, t_hor, quiet = FALSE)`

Arguments:

`t_now` Character date. Today date.

`t_hor` Character date. Horizon date.

`quiet` Logical for verbose messages.

Method `Var()`: Value at Risk for a `solarModel`

Usage:

`solarModel$Var(moments, t_now, t_hor, ci = 0.05)`

Arguments:

moments moments dataset
t_now Character date. Today date.
t_hor Character date. Horizon date.
ci Confidence interval (one tail).

Method logLik(): Compute the log-likelihood of the model and update the slot \$loglik.

Usage:

```
solarModel$logLik(moments, target = "Yt")
```

Arguments:

moments Dataset containing the moments to use for computation.
target Character. Target variable to use "Yt" or "GHI".

Method R_to_Y(): Convert solar radiation Rt into the transformed variable Yt for a given day of the year.

Usage:

```
solarModel$R_to_Y(Rt, t_now)
```

Arguments:

Rt Numeric, solar radiation.
t_now Character, today date.

Returns: Transformed variable on date t_now.

Method Y_to_R(): Convert the transformed variable Yt into solar radiation Rt for a given day of the year.

Usage:

```
solarModel$Y_to_R(Yt, t_now)
```

Arguments:

Yt Numeric, transformed variable.
t_now Character, today date.

Returns: Solar radiation Rt on date t_now.

Method print(): Print method for solarModel class.

Usage:

```
solarModel$print()
```

Method fit_clearsky_model(): Initialize and fit a [seasonalClearsky](#) model given the specification contained in \$control.

Usage:

```
solarModel$fit_clearsky_model()
```

Method compute_risk_drivers(): Compute the risk drivers and impute the observation that are greater or equal to the clear sky level.

Usage:

```
solarModel$compute_risk_drivers()
```

Method fit_solar_transform(): Fit the parameters of the [solarTransform](#) object.

Usage:

```
solarModel$fit_solar_transform()
```

Method `detect_outliers_Yt()`: Detect and assign a zero weight to the outliers to exclude them from the fit. The threshold to be outliers is specified with the control function.

Usage:

```
solarModel$detect_outliers_Yt()
```

Method `fit_seasonal_mean()`: Fit a [seasonalModel](#) the transformed variable (Y_t) and compute deseasonalized series (Y_{t_tilde}).

Usage:

```
solarModel$fit_seasonal_mean()
```

Method `correct_monthly_mean()`: Correct the deseasonalized series (Y_{t_tilde}) by subtracting its monthly mean ($Y_{t_tilde_uncond}$).

Usage:

```
solarModel$correct_monthly_mean()
```

Method `fit_ARMA()`: Fit an AR model (Y_{t_tilde}) and compute AR residuals (ϵ).

Usage:

```
solarModel$fit_ARMA()
```

Method `fit_seasonal_variance()`: Fit a [seasonalModel](#) on AR squared residuals (ϵ) and compute deseasonalized residuals ϵ_{t_tilde} .

Usage:

```
solarModel$fit_seasonal_variance()
```

Method `correct_monthly_variance()`: Correct the standardized series (ϵ_{t_tilde}) by subtracting its monthly mean (σ_{uncond}).

Usage:

```
solarModel$correct_monthly_variance()
```

Method `correct_seasonal_variance()`: Correct the parameters of the seasonal variance to ensure a unitary variance

Usage:

```
solarModel$correct_seasonal_variance()
```

Method `fit_GARCH_model()`: Fit a GARCH model on the deseasonalized residuals (ϵ_{t_tilde}). Compute the standardized (u) and monthly deseasonalized residuals (u_{t_tilde}).

Usage:

```
solarModel$fit_GARCH_model()
```

Method `fit_mixture_model()`: Initialize and fit a solarMixture object.

Usage:

```
solarModel$fit_mixture_model()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
solarModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0

Examples

```

# Control list
control <- control_solarModel(outliers_quantile = 0, seasonal.mean = list(monthly.mean = TRUE),
mean.model = list(arOrder = 1, maOrder = 1), garch_variance = TRUE)
# Model specification
spec <- solarModel_spec("Bologna", from="2005-01-01", control_model = control)
Bologna <- solarModel$new(spec)
# Model fit
Bologna$fit()
# save(Bologna, file = "data/Bologna.RData")

# Extract and update the parameters
model <- Bologna$clone(TRUE)
params <- model$coefficients
model$update(params)
model$filter()

# Fit a model with the realized clear sky
spec$control$stochastic_clearsky <- TRUE
# Initialize a new model
model <- solarModel$new(spec)
# Model fit
model$fit()

# Fit a model for the clearsky
spec_Ct <- spec
spec_Ct$control$stochastic_clearsky <- FALSE
spec_Ct$target <- "clearsky"
# Initialize a new model
model <- solarModel$new(spec)
# Model fit
model$fit()

```

solarModels_control_grid
*Best models control***Description**

Best models control

Usage

```

solarModels_control_grid(
  arOrder = 2,
  maOrder = 2,
  archOrder = 2,
  garchOrder = 2
)

```

solarModels_fit_control_grid
Best models fit

Description

Best models fit

Usage

```
solarModels_fit_control_grid(
  place,
  max_date,
  lag.max = 10,
  ci = 0.05,
  controls
)
```

solarModel_forecast *Iterate the forecast on multiple dates*

Description

Iterate the forecast on multiple dates

Usage

```
solarModel_forecast(model, moments, ci = 0.1, theta = 0)
```

solarModel_moments *Moments from a solarModel object*

Description

Moments from a solarModel object

Usage

```
solarModel_moments(
  t_now,
  t_hor,
  data,
  ARMA,
  GARCH,
  NM_model,
  transform,
  quiet = FALSE
)
```

Examples

```

model = Bologna
t_now = "2021-12-31"
t_hor = "2022-12-01"
data = model$data
ARMA = model$ARMA
GARCH = model$GARCH
NM_model = model$NM_model
transform = model$transform
solarModel_moments(t_now, t_hor, data, ARMA, GARCH, NM_model, transform, quiet = FALSE)

```

solarModel_mvmmixture	<i>Monthly multivariate Gaussian mixture with two components</i>
-----------------------	--

Description

Monthly multivariate Gaussian mixture with two components

Usage

```
solarModel_mvmmixture(model_Ct, model_GHI)
```

Arguments

model_Ct	arg
model_GHI	arg

solarModel_spec	<i>Specification function for a solarModel</i>
-----------------	--

Description

Specification function for a solarModel

Usage

```

solarModel_spec(
  place,
  target = "GHI",
  min_date,
  max_date,
  from,
  to,
  CAMS_data = solarr::CAMS_data,
  control_model = control_solarModel()
)

```

Arguments

place	Character, name of an element in the CAMS_data list.
target	Character, target variable to model. Can be GHI or clearsky.
min_date	Character. Date in the format YYYY-MM-DD. Minimum date for the complete data. If missing will be used the minimum data available.
max_date	Character. Date in the format YYYY-MM-DD. Maximum date for the complete data. If missing will be used the maximum data available.
from	Character. Date in the format YYYY-MM-DD. Starting date to use for training data. If missing will be used the minimum data available after filtering for min_date.
to	character. Date in the format YYYY-MM-DD. Ending date to use for training data. If missing will be used the maximum data available after filtering for max_date.
CAMS_data	named list with radiation data for different locations.
control_model	list with control parameters, see control_solarModel for details.

Examples

```
control_model <- control_solarModel(outliers_quantile = 0)
spec <- solarModel_spec("Bologna", from="2005-01-01", to="2022-01-01", control_model = control_model)
```

solarModel_tests	<i>Autocorrelation and Distribution tests</i>
------------------	---

Description

Evaluate a Kolmogorov-Smirnov test on the residuals of a solarModel model object against the estimated Gaussian mixture distribution and a Breush-pagan or Box-pierce test on the residuals.

Usage

```
solarModel_tests(
  model,
  lag.max = 3,
  ci = 0.05,
  min_quantile = 0.025,
  max_quantile = 0.985,
  method = "bp"
)
```

Arguments

model	An object of the class solarModel
lag.max	Numeric, scalar. Maximum lag to consider for the test.
ci	p.value for rejection.
min_quantile	minimum quantile for the grid of values.
max_quantile	maximum quantile for the grid of values.
method	Character. Type of test. Can be "bp" for breush-pagan or "lb" for Box-pierce.

Examples

```
model <- Bologna
solarModel_tests(model)
```

```
solarModel_test_autocorr
```

Autocorrelation test

Description

Evaluate the autocorrelation in the components of a solarModel object.

Usage

```
solarModel_test_autocorr(model, lag.max = 3, ci = 0.05, method = "bp")
```

Arguments

model	An object of the class solarModel
lag.max	Numeric, scalar. Maximum lag to consider for the test.
ci	Numeric, scalar. Minimum p-value to consider the test "passed".
method	Character. Type of test. Can be "bp" for breush-pagan or "lb" for Box-pierce.

Examples

```
model <- Bologna
solarModel_test_autocorr(model, method = "lb")
```

```
solarModel_test_distribution
```

Distribution test

Description

Evaluate a Kolmogorov-smirnov test on the residuals of a solarModel model object against the estimated Gaussian mixture distribution.

Usage

```
solarModel_test_distribution(  
  model,  
  ci = 0.05,  
  min_quantile = 0.025,  
  max_quantile = 0.985  
)
```

Arguments

model	An object of the class solarModel
ci	p.value for rejection.
min_quantile	minimum quantile for the grid of values.
max_quantile	maximum quantile for the grid of values.

Examples

```
model <- Bologna
solarModel_test_distribution(model)
```

solarOption	<i>Create a SoRad / SoREd contract specification</i>
-------------	--

Description

Create a SoRad / SoREd contract specification

Create a SoRad / SoREd contract specification

Public fields

ticker description

strike Strike price for solar radiation.

t_pricing Character, pricing date.

t_now Character, today date.

t_init Character, inception date.

t_hor Character, maturity date.

tick Numeric, monetary conversion tick.

contract_type Character, maturity date.

Active bindings

control control parameters

tau Numeric, scalar. Time from t_now till t_hor in days.

tau_accrued Numeric, scalar. Time from t_pricing till t_hor in days.

Methods**Public methods:**

- `solarOption$new()`
- `solarOption$set_contract()`
- `solarOption$set_control()`
- `solarOption$print()`
- `solarOption$clone()`

Method `new()`: Initialize the contract

Usage:

```
solarOption$new(contract_type = "SoRad")
```

Arguments:

contract_type Character, contract type "SoRad" or "SoREd"

Method set_contract(): Initialize the contract*Usage:*

```
solarOption$set_contract(t_pricing, t_init, t_hor, strike, tick = 1)
```

Arguments:

t_pricing Character, pricing date.

t_init Character, inception date.

t_hor Character, maturity date.

strike Numeric, strike price.

tick Numeric monetary tick.

Method set_control(): Store a list of custom control parameters*Usage:*

```
solarOption$set_control(control)
```

Arguments:

control List, control parameters.

Method print(): Print method*Usage:*

```
solarOption$print()
```

Method clone(): The objects of this class are cloneable with this method.*Usage:*

```
solarOption$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Note

Version 1.0.0

solarOptionPayoffs	<i>solarOptionPayoff</i>
--------------------	--------------------------

Description

solarOptionPayoff

Usage

```
solarOptionPayoffs(model, control_options = control_solarOption())
```

Arguments

`model` `solarModel`
`control_options` control list, see [control_solarOption](#) for more details.

Value

An object of the class `solarOptionPayoffs`.

`solarOption_calibrator`

Calibrator function for solarOptions Recalibrate and adjust the mixture parameters such that the model premium matches exactly the historical premium for all the months.

Description

Calibrator function for solarOptions Recalibrate and adjust the mixture parameters such that the model premium matches exactly the historical premium for all the months.

Usage

```
solarOption_calibrator(
  model,
  nmonths = 1:12,
  abstol = 1e-04,
  reltol = 1e-04,
  control_options = control_solarOption()
)
```

Arguments

`model` object with the class `solarModel`. See the function [solarModel](#) for details.
`nmonths` numeric vector of months in which the payoff is computed. Range from 1 to 12.
`abstol` The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.
`reltol` Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of $\text{reltol} * (\text{abs}(\text{val}) + \text{reltol})$ at a step. Defaults to $\sqrt{.Machine\$double.eps}$, typically about $1e-8$.
`control_options` control list, see [control_solarOption](#) for more details.

Examples

```
model <- Bologna
model_cal <- solarOption_calibrator(model, nmonths = 7, reltol=1e-3)
# Compare log-likelihoods
model$loglik
model_cal$loglik
# Compare parameters
model$NM_model$coefficients[7,]
model_cal$NM_model$coefficients[7,]
```

solarOption_contracts *Optimal number of contracts*

Description

Compute the optimal number of contracts given a particular setup.

Usage

```
solarOption_contracts(
  payoff,
  type = "model",
  premium = "Q",
  put = TRUE,
  nyear = 2021,
  control = control_hedging()
)
```

Arguments

type	character, method used for computing the premium. Can be model (Model with integral) or sim (Monte Carlo).
premium	character, premium used. Can be P, Qdw, Qup, or Q.
nyear	integer, actual year. The optimization will be performed excluding the year nyear and the following.
model	object with the class solarModel. See the function solarModel for details.
control_hedging	numeric, list of hedging parameters.

solarOption_historical
 Payoff on Historical Data

Description

Payoff on Historical Data

Usage

```
solarOption_historical(
  model,
  nmonths = 1:12,
  put = TRUE,
  control_options = control_solarOption()
)
```

Arguments

model	object with the class solarModel. See the function solarModel for details.
nmonths	numeric vector of months in which the payoff is computed. Range from 1 to 12.
put	logical, when TRUE, the default, the computations will consider a put contract. Otherwise a call.
control_options	control list, see control_solarOption for more details.

Value

An object of the class solarOptionPayoff.

Examples

```
model <- Bologna
solarOption_historical(model, put=TRUE)
solarOption_historical(model, put=FALSE)
```

solarOption_historical_bootstrap

Bootstrap a fair premium from historical data

Description

Bootstrap a fair premium from historical data

Usage

```
solarOption_historical_bootstrap(
  model,
  put = TRUE,
  control_options = control_solarOption()
)
```

Arguments

model	object with the class solarModel. See the function solarModel for details.
put	logical, when TRUE, the default, the computations will consider a put contract. Otherwise a call.
control_options	control list, see control_solarOption for more details.

Value

An object of the class solarOptionBoot.

Examples

```
model <- Bologna
solarOption_historical_bootstrap(model, control_options = control_solarOption(ci = 0.4, nsim = 1000))
```

solarOption_model	<i>Compute the price of a solarOptionPortfolio</i>
-------------------	--

Description

Compute the price of a solarOptionPortfolio

Usage

```
solarOption_model(
  model,
  moments,
  portfolio,
  theta = 0,
  put = TRUE,
  control = control_solarOption()
)
```

Arguments

moments	description
object	An object of the class solarOptionPortfolio.

Examples

```
# Model
model <- Bologna$clone(TRUE)
# Pricing date
t_now <- as.Date("2021-12-31")
# Inception date
t_init <- as.Date("2022-01-01")
# Maturity date
t_hor <- as.Date("2022-12-31")
# SoRad portfolio
portfolio <- SoRadPorfolio(model, t_now, t_init, t_hor)
# Moments
moments <- map_df(portfolio, ~model$Moments(t_now, .x$t_hor))
moments <- filter(model$moments$unconditional, date >= t_init & date <= t_hor)
# Premium
premium_Vt <- solarOption_model(model, moments, portfolio, theta = 0.0, put = TRUE)
premium_Vt$payoff_year$premium
```

solarOption_pricing	<i>Compute the price of a solarOption</i>
---------------------	---

Description

Compute the price of a solarOption

Usage

```
solarOption_pricing(
  object,
  moments,
  theta = 0,
  put = TRUE,
  control = control_solarOption()
)
```

Arguments

object	An object of the class solarOption.
moments	description

Examples

```
model <- Bologna$clone(TRUE)
moments <- filter(model$moments$conditional, Year == 2022)
sorad <- solarOption$new()
sorad$set_contract("2021-12-31", "2022-01-01", "2022-04-20", moments$GHI_bar[1])
solarOption_pricing(sorad, moments[1,])
solarOption_pricing(sorad, moments[1,], theta = 0.02)
solarOption_pricing(sorad, moments[1,], theta = -0.02)
```

solarOption_scenario	<i>Payoff on simulated Data</i>
----------------------	---------------------------------

Description

Payoff on simulated Data

Usage

```
solarOption_scenario(
  scenario,
  nmonths = 1:12,
  put = TRUE,
  measure = "P",
  nsim,
  control_options = control_solarOption()
)
```

Arguments

scenario	object with the class solarModelScenario. See the function solarModel_scenarios for details.
nmonths	numeric vector of months in which the payoff is computed. Range from 1 to 12.
put	logical, when TRUE, the default, the computations will consider a put contract. Otherwise a call.

nsim number of simulation to use for computation.
 control_options control function, see [control_solarOption](#) for details.

Value

An object of the class solarOptionPayoff.

Examples

```
model <- Bologna
scenario <- solarScenario(model, from = "2011-01-01", to = "2012-01-01", by = "1 month", nsim = 1, seed = 3)
solarOption_scenario(scenario)
solarOption_historical(model)
```

solarOption_structure *Structure payoffs*

Description

Structure payoffs

Usage

```
solarOption_structure(
  payoffs,
  type = "model",
  put = TRUE,
  exact_daily_premium = TRUE
)
```

Arguments

payoffs object with the class solarOptionPayoffs. See the function [solarOptionPayoffs](#) for details.

type method used for computing the premium. If model, the default will be used the analytic model, otherwise with scenarios the monte carlo scenarios stored inside the model\$scenarios\$P.

exact_daily_premium when TRUE the historical premium is computed as daily average among all the years. Otherwise the monthly premium is computed and then divided by the number of days of the month.

Value

The object payoffs with class solarOptionPayoffs.

solarScenario	<i>Simulate multiple scenarios</i>
---------------	------------------------------------

Description

Simulate multiple scenarios of solar radiation with a solarModel object.

Usage

```
solarScenario(
  model,
  from = "2010-01-01",
  to = "2011-01-01",
  by = "1 year",
  theta = 0,
  nsim = 1,
  seed = 1,
  quiet = FALSE
)
```

Arguments

model	object with the class solarModel. See the function solarModel for details.
from	character, start Date for simulations in the format YYYY-MM-DD.
to	character, end Date for simulations in the format YYYY-MM-DD.
by	character, steps for multiple scenarios, e.g. 1 day (day-ahead simulations), 15 days, 1 month, 3 months, ecc. For each step are simulated nsim scenarios.
theta	numeric, Esscher parameter.
nsim	integer, number of simulations.
seed	scalar integer, starting random seed.
quiet	logical

Examples

```
model <- Bologna
scen <- solarScenario(model, "2016-01-01", "2017-01-01", nsim = 10, by = "1 month")
# Plot
plot(scen, nsim = 3)
# Solar Option
solarOption_scenario(scen)
```

solarScenario_filter *Simulate trajectories from a a solarScenario_spec*

Description

Simulate trajectories from a a solarScenario_spec

Usage

```
solarScenario_filter(simSpec)
```

Arguments

simSpec object with the class solarScenario_spec. See the function [solarScenario_spec](#) for details.

Examples

```
model <- Bologna
simSpec <- solarScenario_spec(model, from = "2023-01-01", to = "2023-12-31")
simSpec <- solarScenario_residuals(simSpec, nsim = 1, seed = 1)
simSpec <- solarScenario_filter(simSpec)
# Empiric data
df_emp <- simSpec$emp
# First simulation
df_sim <- simSpec$simulations[[1]]
ggplot()+
  geom_line(data = df_emp, aes(date, GHI))+
  geom_line(data = df_sim, aes(date, GHI), color = "red")
```

solarScenario_residuals
 Simulate residuals for a solarScenario_spec

Description

Simulate residuals for a solarScenario_spec

Usage

```
solarScenario_residuals(simSpec, nsim = 1, seed = 1)
```

Arguments

simSpec object with the class solarScenario_spec. See the function [solarScenario_spec](#) for details.

nsim integer, number of simulations.

seed scalar integer, starting random seed.

Examples

```
model <- Bologna
simSpec <- solarScenario_spec(model)
simSpec <- solarScenario_residuals(simSpec, nsim = 10)
```

solarScenario_spec	<i>Specification of a solar scenario</i>
--------------------	--

Description

Specification of a solar scenario

Usage

```
solarScenario_spec(
  model,
  from = "2010-01-01",
  to = "2010-12-31",
  theta = 0,
  exclude_known = FALSE,
  quiet = FALSE
)
```

Arguments

model	object with the class <code>solarModel</code> . See the function solarModel for details.
from	character, start Date for simulations in the format YYYY-MM-DD.
to	character, end Date for simulations in the format YYYY-MM-DD.
theta	numeric, Esscher parameter.
exclude_known	when true the two starting points (equals for all the simulations) will be excluded from the output.
quiet	logical

Examples

```
model <- Bologna
simSpec <- solarScenario_spec(model)
```

solarTransform	<i>solarTransform Solar functions</i>
----------------	---------------------------------------

Description

Solar Model transformation functions

Public fields

epsilon Numeric, ϵ transformation parameter.

Active bindings

alpha Numeric, α transformation parameter.

beta Numeric, β transformation parameter.

Methods

Public methods:

- `solarTransform$new()`
- `solarTransform$GHI()`
- `solarTransform$GHI_y()`
- `solarTransform$iGHI()`
- `solarTransform$Y()`
- `solarTransform$iY()`
- `solarTransform$ieta()`
- `solarTransform$eta()`
- `solarTransform$fit()`
- `solarTransform$bounds()`
- `solarTransform$update()`
- `solarTransform$print()`
- `solarTransform$clone()`

Method `new()`: Initialize a solarTransform object.

Usage:

```
solarTransform$new(alpha = 0, beta = 1)
```

Arguments:

alpha Numeric, α transformation parameter.

beta Numeric, β transformation parameter.

Method `GHI()`: Map the risk driver X in solar radiation

Usage:

```
solarTransform$GHI(x, Ct)
```

Arguments:

x Numeric values in $(\alpha, \alpha + \beta)$.

Ct Numeric, clear sky radiation.

Details: The function computes:

$$GHI(x) = C_t(1 - x)$$

Returns: Numeric values in $C_t(1 - \alpha - \beta, 1 - \alpha)$.

Method GHI_y(): Map the transformed variable Y in solar radiation

Usage:

solarTransform\$GHI_y(y, Ct)

Arguments:

y Numeric values in $(-\infty, \infty)$.

Ct Numeric, clear sky radiation.

Details: The function computes:

$$GHI(y) = C_t(1 - \alpha - \beta \exp(-\exp(x)))$$

Returns: Numeric values in $[C_t(1 - \alpha - \beta), C_t(1 - \alpha)]$.

Method iGHI(): Map the solar radiation in the risk driver X

Usage:

solarTransform\$iGHI(x, Ct)

Arguments:

x Numeric values in $[C_t(1 - \alpha - \beta), C_t(1 - \alpha)]$.

Ct Numeric, clear sky radiation.

Details: The function computes the inverse of the GHI function

$$iGHI(x) = 1 - \frac{x}{C_t}$$

Returns: Numeric values in $[\alpha, \alpha + \beta]$.

Method Y(): Map the risk driver X in the transformed variable Y

Usage:

solarTransform\$Y(x)

Arguments:

x numeric vector in $[\alpha, \alpha + \beta]$.

Details: The function computes:

$$Y(x) = \log(\log(\beta) - \log(x - \alpha))$$

Returns: Numeric values in $[-\infty, \infty]$.

Method iY(): Map the transformed variable Y in the risk driver X.

Usage:

solarTransform\$iY(y)

Arguments:

y numeric vector in $[-\infty, \infty]$.

Details: The function computes:

$$iY(y) = \alpha + \beta \exp(-\exp(y))$$

Returns: Numeric values in $[\alpha, \alpha + \beta]$.

Method `ieta()`: Map the risk driver X in the normalized variable Z. Transformation function from X to Y

Usage:

`solarTransform$ieta(x)`

Arguments:

x numeric vector in $[\alpha, \alpha + \beta]$.

Details: The function computes:

$$\eta^{-1}(x) = \frac{x - \alpha}{\beta}$$

Returns: Numeric values in $[0, 1]$.

Method `eta()`: Map the normalized variable Z in the risk driver X.

Usage:

`solarTransform$eta(z)`

Arguments:

z numeric vector in $[0, 1]$.

Details: The function computes:

$$\eta(z) = \alpha + \beta \cdot z$$

Returns: Numeric values in $[\alpha, \alpha + \beta]$.

Method `fit()`: Fit the best parameters α and β from a given time series

Usage:

`solarTransform$fit(x, epsilon = 0.01)`

Arguments:

x time series of solar risk drivers in $(0, 1)$.

epsilon Numeric,

Details: Return a list that contains:

alpha Numeric, α transformation parameter.

beta Numeric, β transformation parameter.

epsilon Numeric, threshold used for fitting.

Xt_min Numeric, minimum value of the time series.

Xt_max Numeric, maximum value of the time series.

Returns: A named list.

Method `bounds()`: Compute the bounds for the transformed variables.

Usage:

`solarTransform$bounds(target = "Xt")`

Arguments:

target target variable. Available choices are:

"Xt" Solar risk driver, the bounds returned are $[\alpha, \alpha + \beta]$.

"Kt" Clearness index, the bounds returned are $[1 - \alpha - \beta, 1 - \alpha]$.

"Yt" Solar transform, the bounds returned are $[-\infty, \infty]$.

Returns: A numeric vector where the first element is the lower bound and the second the upper bound.

Method `update()`: Update the transformation parameters α and β .

Usage:

```
solarTransform$update(alpha, beta)
```

Arguments:

`alpha` Numeric, transformation parameter.

`beta` Numeric, transformation parameter.

Returns: Update the slots `$alpha` and `$beta`.

Method `print()`: Print method for the class `solarTransform`

Usage:

```
solarTransform$print()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
solarTransform$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
st <- solarTransform$new()
st$GHI(0.4, 3)
st$GHI(st$iGHI(0.4, 3), 3)
```

SoRadPorfolio

Create a SoRad / SoREd portfolio

Description

Create a SoRad / SoREd portfolio

Usage

```
SoRadPorfolio(model, t_now, t_init, t_hor)
```

spatialCorrelation *spatialCorrelation object*

Description

spatialCorrelation object
spatialCorrelation object

Active bindings

places Get a vector with the labels of all the places in the grid.
sigma_B Get a list of matrices with implied covariance matrix from joint probabilities.
cr_X Get a matrix with multivariate gaussian mixture correlations.
margprob Get a list of vectors with marginal probabilities.

Methods

Public methods:

- `spatialCorrelation$new()`
- `spatialCorrelation$get_sigma_B()`
- `spatialCorrelation$get_margprob()`
- `spatialCorrelation$get_cr_X()`
- `spatialCorrelation$get()`
- `spatialCorrelation$clone()`

Method `new()`: Initialize an object with class `spatialCorrelation`.

Usage:

```
spatialCorrelation$new(binprobs, mixture_cr)
```

Arguments:

binprobs param

mixture_cr param

Method `get_sigma_B()`: Extract the implied covariance matrix for a given month and places.

Usage:

```
spatialCorrelation$get_sigma_B(places, nmonth = 1)
```

Arguments:

places character, optional. Names of the places to consider.

nmonth integer, month considered from 1 to 12.

Method `get_margprob()`: Extract the marginal probabilities for a given month and places.

Usage:

```
spatialCorrelation$get_margprob(places, nmonth = 1)
```

Arguments:

places character, optional. Names of the places to consider.

nmonth integer, month considered from 1 to 12.

Method `get_cr_X()`: Extract the covariance matrix of the gaussian mixture for a given month and places.

Usage:

```
spatialCorrelation$get_cr_X(places, nmonth = 1)
```

Arguments:

`places` character, optional. Names of the places to consider.

`nmonth` integer, month considered from 1 to 12.

Method `get()`: Extract a list with `sigma_B`, `margprob` and `cr_X` for a given month.

Usage:

```
spatialCorrelation$get(places, nmonth = 1, date)
```

Arguments:

`places` character, optional. Names of the places to consider.

`nmonth` integer, month considered from 1 to 12.

`date` character, optional date. The month will be extracted from the date.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
spatialCorrelation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

spatialGrid

Spatial Grid

Description

Spatial Grid

Spatial Grid

Details

Create a grid from a range of latitudes and longitudes.

Value

a tibble with two columns `lat` and `lon`.

Methods

Public methods:

- `spatialGrid$new()`
- `spatialGrid$make_grid()`
- `spatialGrid$is_inside_bounds()`
- `spatialGrid$is_known_point()`
- `spatialGrid$known_point()`
- `spatialGrid$neighborhoods()`

- `spatialGrid$clone()`

Method new():

Usage:

```
spatialGrid$new(  
  lat_min = 43.7,  
  lat_max = 45.1,  
  lon_min = 9.2,  
  lon_max = 12.7,  
  lat_by = 0.1,  
  lon_by = 0.1,  
  weights = IDW(2)  
)
```

Method make_grid():

Usage:

```
spatialGrid$make_grid(labels)
```

Method is_inside_bounds(): Check if a point is inside the bounds of the spatial grid.

Usage:

```
spatialGrid$is_inside_bounds(lat, lon)
```

Arguments:

lat numeric, latitude of a location.

lon numeric, longitude of a location.

Returns: TRUE when the point is inside the limits and FALSE otherwise.

Method is_known_point(): Check if a point is already in the spatial grid

Usage:

```
spatialGrid$is_known_point(lat, lon)
```

Arguments:

lat numeric, latitude of a location.

lon numeric, longitude of a location.

Returns: TRUE when the point is a known point and FALSE otherwise.

Method known_point(): Return the ID and coordinates of a point that is already in the spatial grid

Usage:

```
spatialGrid$known_point(lat, lon)
```

Arguments:

lat numeric, latitude of a location.

lon numeric, longitude of a location.

Method neighborhoods(): Find the n-closest neighborhoods of a point

Usage:

```
spatialGrid$neighborhoods(lat, lon, n = 4)
```

Arguments:

lat numeric, latitude of a point in the grid.

lon numeric, longitude of a point in the grid.

n number of neighborhoods

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
spatialGrid$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
grid <- spatialGrid$new()
grid$make_grid()
grid$grid
grid$weights <- IDW(beta = 2)
grid$is_known_point(49.95, 12.15)
grid$known_point(c(44.85, 44.9), c(12.15, 11.2))
grid$is_inside_bounds(44.8, 10.9)
grid$neighborhoods(44.9, 12.1)
grid$neighborhoods(44.95, 12.15)
filter(grid$grid, lat == 44.95 & lon == 12.15)
```

spatialModel

Spatial model object

Description

Spatial model object

Spatial model object

Public fields

quiet logical, when TRUE the function will not display any message.

Active bindings

models list of solarModel objects

grid object with the spatial grid

parameters spatialParameters object

Methods

Public methods:

- `spatialModel$new()`
- `spatialModel$gridModel()`
- `spatialModel$interpolator()`
- `spatialModel$solarModel()`
- `spatialModel$combinations()`

- `spatialModel$clone()`

Method `new()`: Initialize the spatial model

Usage:

```
spatialModel$new(grid, models, paramsModels, quiet = FALSE)
```

Arguments:

`grid` A spatialGrid object.

`models` A list of solarModel objects

`paramsModels` A spatialParameters object.

`quiet` logical

`beta` numeric, used in exponential and power functions.

`d0` numeric, used only in exponential function.

Method `gridModel()`: Get a known model in the grid from place or coordinates.

Usage:

```
spatialModel$gridModel(id, lat, lon)
```

Arguments:

`id` character, id of the location.

`lat` numeric, latitude of a location.

`lon` numeric, longitude of a location.

Method `interpolator()`: Perform the bilinear interpolation for a target variable.

Usage:

```
spatialModel$interpolator(lat, lon, target = "GHI", n = 4, day_date)
```

Arguments:

`lat` numeric, latitude of the location to be interpolated.

`lon` numeric, longitude of the location to be interpolated.

`target` character, name of the target variable to interpolate.

`n` number of neighborhoods to use for interpolation.

`day_date` date for interpolation, if missing all the available dates will be used.

Method `solarModel()`: Interpolator function for a solarModel object

Usage:

```
spatialModel$solarModel(lat, lon, n = 4)
```

Arguments:

`lat` numeric, latitude of a point in the grid.

`lon` numeric, longitude of a point in the grid.

`n` number of neighborhoods

Method `combinations()`: Compute monthly moments for mixture with 16 components

Usage:

```
spatialModel$combinations(lat, lon, nmonths = 1:12, nobsm.min = 3)
```

Arguments:

`lat` numeric, latitude of a point in the grid.

`lon` numeric, longitude of a point in the grid.

`nmonths` numeric, months to consider

nobs.min numeric, minimum number of joint states under which the state is considered with 0 probability.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
spatialModel$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

spatialParameters	spatialParameters <i>object</i>
-------------------	---------------------------------

Description

spatialParameters object

spatialParameters object

Public fields

quiet Logical

Active bindings

models list of kernelRegression objects

data dataset with the parameters used for fitting

Methods

Public methods:

- [spatialParameters\\$new\(\)](#)
- [spatialParameters\\$fit\(\)](#)
- [spatialParameters\\$predict\(\)](#)
- [spatialParameters\\$clone\(\)](#)

Method new(): Initialize a spatialParameters object

Usage:

```
spatialParameters$new(data, params_names, models, sample)
```

Arguments:

data dataset with spatial parameters and lon, lat.

params_names Names of the parameters to fit.

models an optional list of kernelRegression models already fitted.

sample List of parameter used as sample.

Method fit(): Fit a kernelRegression object for a parameter or a group of parameters.

Usage:

```
spatialParameters$fit(params)
```

Arguments:

params list of parameters names to fit. When missing all the parameters will be fitted.

Method predict(): Predict all the parameters for a specified location.

Usage:

```
spatialParameters$predict(lat, lon, as_tibble = FALSE)
```

Arguments:

lat numeric, latitude in degrees.

lon numeric, longitude in degrees.

as_tibble logical, when TRUE will be returned a tibble.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
spatialParameters$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

spatialScenario_filter

Simulate trajectories from a spatialScenario_spec

Description

Simulate trajectories from a spatialScenario_spec

Usage

```
spatialScenario_filter(simSpec)
```

Arguments

simSpec	object with the class spatialScenario_spec. See the function spatialScenario_spec for details.
---------	--

spatialScenario_residuals

Simulate residuals from a a spatialScenario_spec

Description

Simulate residuals from a a spatialScenario_spec

Usage

```
spatialScenario_residuals(simSpec, nsim = 1, seed = 1)
```

Arguments

simSpec	object with the class spatialScenario_spec. See the function spatialScenario_spec for details.
nsim	integer, number of simulations.
seed	scalar integer, starting random seed.

spatialScenario_spec *Specification of a solar scenario*

Description

Specification of a solar scenario

Usage

```
spatialScenario_spec(
  sm,
  sc,
  places,
  from = "2010-01-01",
  to = "2010-01-31",
  exclude_known = FALSE,
  quiet = FALSE
)
```

Arguments

sm	spatialModel object
sc	spatialCorrelation object
places	target places
from	character, start Date for simulations in the format YYYY-MM-DD.
to	character, end Date for simulations in the format YYYY-MM-DD.
exclude_known	when true the two starting points (equals for all the simulations) will be excluded from the output.
quiet	logical

spectralDistribution *Compute the spectral distribution for a black body*

Description

Compute the spectral distribution for a black body

Usage

```
spectralDistribution(x, measure = "nanometer")
```

Arguments

measure	character, measure of the irradiated energy. If nanometer the final energy will be in W/m ² x nanometer, otherwise if micrometer the final energy will be in W/m ² x micrometer.
lambda	numeric, wave length in micrometers.

test_normality	<i>Perform normality tests</i>
----------------	--------------------------------

Description

Perform normality tests

Usage

```
test_normality(x = NULL, pvalue = 0.05)
```

Arguments

x	numeric, a vector of observation.
pvalue	numeric, the desiderd level of p.value at which the null hypothesis will be rejected.

Value

a tibble with the results of the normality tests.

Examples

```
set.seed(1)
x <- rnorm(1000, 0, 1) + rchisq(1000, 1)
test_normality(x)
x <- rnorm(1000, 0, 1)
test_normality(x)
```

tnorm	<i>Truncated Normal random variable</i>
-------	---

Description

Truncated Normal density, distribution, quantile and random generator.

Usage

```
dtnorm(x, mean = 0, sd = 1, a = -3, b = 3, log = FALSE)

ptnorm(x, mean = 0, sd = 1, a = -3, b = 3, log.p = FALSE, lower.tail = TRUE)

qtnorm(p, mean = 0, sd = 1, a = -3, b = 3, log.p = FALSE, lower.tail = TRUE)

rtnorm(n, mean = 0, sd = 1, a = -100, b = 100)
```

Arguments

x	vector of quantiles.
mean	vector of means.
sd	vector of standard deviations.
a	lower bound.
b	upper bound.
log	logical; if TRUE, probabilities are returned as $\log(p)$.
log.p	logical; if TRUE, probabilities p are given as $\log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X < x]$ otherwise, $P[X > x]$.
p	vector of probabilities.
n	number of observations. If $\text{length}(n) > 1$, the length is taken to be the number required.

Examples

```
x <- seq(-5, 5, 0.01)

# Density function
p <- dtnorm(x, mean = 0, sd = 1, a = -1)
plot(x, p, type = "l")

# Distribution function
p <- ptnorm(x, mean = 0, sd = 1, b = 1)
plot(x, p, type = "l")

# Quantile function
dtnorm(0.1)
ptnorm(qtnorm(0.1))

# Random Numbers
rtnorm(1000)
plot(rtnorm(100, mean = 0, sd = 1, a = 0, b = 1), type = "l")
```


Index

* 2beRevised

solarOption_contracts, 75

ARMA_modelR6, 3

calibrate_dQdP_solar, 5

CDF (PDF), 38

clearsky_optimizer, 7

control_hedging, 6

control_seasonalClearsky, 6, 8, 48

control_solarModel, 7, 70

control_solarOption, 9, 58, 74, 76, 79

create_monthly_sequence, 44, 45

desscher, 10

desscherMixture, 11

detect_season, 12

dgumbel, 13

dinvgumbel, 14

discountFactor, 15

dkumaraswamy, 15

dmixnorm, 16

dmvmixnorm, 17

dmvsolarGHI, 18

dsnrm, 19

dsolarGHI, 20

dsolarK, 21

dsolarX, 23

dtnorm (tnorm), 95

GARCH_modelR6, 24

gaussianMixture, 27, 60

havDistance, 30

IDW, 31

is_leap_year, 31

kernelRegression, 32

ks_test, 33

ks_test_ts, 34

ks_ts_test (ks_test), 33

loss_dQdP, 34

loss_dQdP_tau, 35

makeSemiPositive, 35

monthlyParams, 35, 60

mvgaussianMixture, 37

number_of_day, 37

optionPayoff, 38

PDF, 38

pesscherMixture (desscherMixture), 11

pgumbel (dgumbel), 13

pinvgumbel (dinvgumbel), 14

pkumaraswamy (dkumaraswamy), 15

plot.solarModelForecast, 39

plot.solarScenario, 39

pmixnorm (dmixnorm), 16

pmvmixnorm (dmvmixnorm), 17

pow_matrix, 40

predict.solarModel, 40

prepare_dQdP_calibration, 40

prepare_dQdP_calibration_tau, 41

psnorm (dsnrm), 19

psolarGHI (dsolarGHI), 20

psolarK (dsolarK), 21

psolarX (dsolarX), 23

ptnorm (tnorm), 95

qgumbel (dgumbel), 13

qinvgumbel (dinvgumbel), 14

qkumaraswamy (dkumaraswamy), 15

qmixnorm (dmixnorm), 16

qvmixnorm (dmvmixnorm), 17

qsnorm (dsnrm), 19

qsolarGHI (dsolarGHI), 20

qsolarK (dsolarK), 21

qsolarX (dsolarX), 23

qtnorm (tnorm), 95

Quantile (PDF), 38

radiationModel, 41

rgumbel (dgumbel), 13

riccati_root, 47

rinvgumbel (dinvgumbel), 14

rkumaraswamy (dkumaraswamy), 15

rmixnorm (dmixnorm), 16

rsnorm (dsnorm), 19
rsolarGHI (dsolarGHI), 20
rsolarK (dsolarK), 21
rsolarX (dsolarX), 23
rtnorm (tnorm), 95

seasonalClearsky, 48, 65
seasonalModel, 49, 66
seasonalSolarFunctions, 52
solarEsscher, 58
solarEsscher_probability, 59
solarMixture, 60
solarModel, 59, 62, 64, 74–76, 80, 82
solarModel_forecast, 68
solarModel_moments, 68
solarModel_mvmmixture, 69
solarModel_scenarios, 78
solarModel_spec, 63, 64, 69
solarModel_test_autocorr, 71
solarModel_test_distribution, 71
solarModel_tests, 70
solarModels_control_grid, 67
solarModels_fit_control_grid, 68
solarOption, 72
solarOption_calibrator, 74
solarOption_contracts, 75
solarOption_historical, 75
solarOption_historical_bootstrap, 10, 76
solarOption_model, 77
solarOption_pricing, 77
solarOption_scenario, 78
solarOption_structure, 79
solarOptionPayoffs, 59, 73, 79
solarr::seasonalModel, 48
solarScenario, 80
solarScenario_filter, 81
solarScenario_residuals, 81
solarScenario_spec, 81, 82
solarTransform, 9, 63, 65, 83
SoRadPorfolio, 86
spatialCorrelation, 87
spatialGrid, 88
spatialModel, 90
spatialParameters, 92
spatialScenario_filter, 93
spatialScenario_residuals, 93
spatialScenario_spec, 93, 94
spectralDistribution, 94

test_normality, 95
tnorm, 95