Package ‘solarr’

October 16, 2025

Type Package

Title Stochastic models for solar radiation

Version 1.0.0

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

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

Depends R (>= 4.4.0),
ggplot2,
dplyr,
mclust

Imports np,
broom,
stringr,
rugarch,
purrr,
tidyr,
lubridate,
formula.tools,
numDeriv,
sp,
gstat

Suggests knitr,
rmarkdown,
testthat

License `use_gpl3_license()`

VignetteBuilder knitr

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE, r6 = TRUE)

RoxygenNote 7.3.2

Contents

`ARMA_modelR6` *R6 class for ARMA(p, q) model*

Description

R6 class for ARMA(p, q) model

R6 class for ARMA(p, q) model

Active bindings

`model` An object with the fitted ARMA model from the function `stats::arima()`.

`intercept` Numeric named scalar, intercept of the model.

`phi` Numeric named vector, AR parameters.

`theta` Numeric named vector, MA parameters.

`order` Numeric named vector, ARMA order. The first element is the AR order, while the second the MA order.

`coefficients` Numeric named vector, intercept and ARMA parameters.

`mean` Numeric scalar, long term expectation.

`variance` Numeric scalar, long term variance.

`Phi` Numeric matrix, companion matrix to govern the transition between two time steps.

`b` Numeric vector, unitary vector for the residuals.

`tidy` Tibble with estimated parameters and relative std. errors.

Methods

Public methods:

- `ARMA_modelR6$new()`
- `ARMA_modelR6$fit()`
- `ARMA_modelR6$filter()`
- `ARMA_modelR6$next_step()`
- `ARMA_modelR6$update()`
- `ARMA_modelR6$update_std.errors()`
- `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, forecasted 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. If missing nothing will be updated.

Method update_std.errors()**:** Update the standard errors of the parameters.

Usage:

ARMA_modelR6$update_std.errors(std.errors)

Arguments:

std.errors Numeric named vector, parameters' standard errors. If missing nothing will be updated.

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

See Also

stats::arima() which is wrapped in the method fit.

`as_solarScenario` *Extract and structure simulations from a* `solarScenarioSpec`

Description

Extract and structure simulations from a `solarScenarioSpec`

Usage

```
as_solarScenario(simSpec, by)
```

Arguments

`simSpec`	object with the class `solarScenario_spec`. See the function `solarScenario_spec` for details.
`by`	Optional character. Represent the steps used for multiple scenarios.

Note

Version 1.0.0.

`clearsky_optimizer` *Optimizer for clear sky model with restricted least squares (RLS).*

Description

Optimizer for clear sky model with restricted least squares (RLS).

Usage

```
clearsky_optimizer(seasonal_model_Ct, newdata, ntol = 0)
```

Arguments

`seasonal_model_Ct`	An object of the class `seasonalClearsky`. See the function `seasonalClearsky` for more details.
`newdata`	A data frame to input to the method `seasonal_model_Ct$predict()`.
`ntol`	Integer scalar. Tolerance for the maximum number of violations admitted of the condition `clearsky` > `GHI`. Default is `0`.

Note

Version 1.0.0

clearsky_outliers *Impute clear sky outliers*

Description

Detect and impute outliers with respect to a maximum level of radiation (Ct)

Usage

```
clearsky_outliers(x, Ct, date, threshold = 1e-04, quiet = FALSE)
```

Arguments

x	Numeric vector, time series of solar radiation.
Ct	Numeric vector, time series of fitted clear sky radiation.
date	Character or Date vector, time series of dates used to precisely impute solar radiation according to the realized values in the same day of the year.
threshold	Numeric, scalar, threshold value used for imputation. Default is `0.0001`.
quiet	Logical.

Details

The function will detect the observations for which `x > Ct`, `x < 0` or `is.na(x)`. Then, if

`x < 0` If a value is below 0 for a day it will be imputed to be equal to `min(x)` for that day.

`x > Ct` If a value is above the maximum clear sky Ct it will be imputed to be `Ct*(1-threshold)`.

`is.na(x)` If a value is NA it will be imputed to be the average `mean(x)` for that day..

Note

Version 1.0.0

Examples

```
clearsky_outliers(c(1,2,3), 2)
```

control_seasonalClearsky *Control parameters for a* `seasonalClearsky` *object*

Description

Control parameters for a `seasonalClearsky` object

Usage

```
control_seasonalClearsky(
  order = 1,
  order_H0 = 1,
  period = 365,
  include.intercept = TRUE,
  include.trend = FALSE,
  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 the optimization. The estimated clear sky is increased by delta0.
ntol	Integer scalar. Tolerance for the maximum number of violations admitted of the condition clearsky > GHI. 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

Examples

```
control = control_seasonalClearsky()
```

control_solarHedging *Control parameters for solar hedging*

Description

Control parameters for solar hedging

Usage

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

Arguments

`n_panels`	Numeric scalar, number of meters squared of solar panels.
`efficiency`	Numeric scalar, average electricity produced with 1 m^2 of solar panels given 1 kWh/m^2 of solar radiation received.
`PUN`	Numeric scalar, fixed electricity price at which the produced energy is sold.
`tick`	Numeric scalar, tick for the monetary conversion of the payoff of a solar derivative from kWh/m^2 to Euros.
`n_contract`	Numeric scalar, number of solar derivative contracts bought.

Note

Version 1.0.0.

control_solarOption *Control parameters for a solar option*

Description

Control parameters for a solar option

Usage

```
control_solarOption(
  nyears = c(2005, 2025),
  K = 0,
  leap_year = FALSE,
  nsim = 200,
  ci = 0.05,
  seed = 1
)
```

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`.

Note

Version 1.0.0.

Examples

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

`desscher` *Esscher-distorted density and distribution*

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)

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

Arguments

`pdf`	Function, density function to distort.
`theta`	Numeric, distortion parameter.
`lower, upper`	Numeric, lower and upper bounds for integration, i.e. the bounds of the pdf.

Details

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

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

Version 1.0.0.

Value

A function of x.

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)

# Esscher Distribution (transform)
esscher_cdf <- pesscher(pdf, theta = -0.1)
plot(esscher_cdf(grid))
```

---

desscherMixture *Esscher-distorted density and distribution of a Gaussian Mixture*

Description

Esscher-distorted density and distribution 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	Numeric, distortion parameter.

Details

Version 1.0.0.

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`	logica, 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 length(n) > 1, the length is taken to be the number required.

Details

Version 1.0.0.

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 `length(n) > 1`, the length is taken to be the number required.

Details

Version 1.0.0.

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")
```

---

dkumaraswamy *Kumaraswamy random variable*

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 length(n) > 1, the length is taken to be the number required.

Details

Version 1.0.0.

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 $\mathbb{P}(X < x)$, otherwise $\mathbb{P}(X \geq x)$.
`log.p`	logical; if `TRUE`, probabilities p are given as `log(p)`.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.

Details

Version 1.0.0.

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
qmixnorm(0.2, mean, sd, alpha)
# Random generator
rmixnorm(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
```

```
)

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

Details

Version 1.0.0.

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)
qmvmixnorm(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.

Details

Version 1.0.0.

---

dsnorm — *Skewed Normal random variable*

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 `length(n) > 1`, the length is taken to be the number required.

Details

Version 1.0.0.

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	*Density, distribution, quantile and random generator of Solar Radiation*

Description

Density, distribution, quantile and random generator of Solar Radiation

Distribution function for the GHI

Quantile function for the GHI

Random generator function for the GHI

Usage

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

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

qsolarGHI(
```

```
  p,
  Ct,
  alpha,
  beta,
  cdf_Y,
  log.p = FALSE,
  lower.tail = TRUE,
  link = "invgumbel"
)

rsolarGHI(n, Ct, alpha, beta, cdf_Y, link = "invgumbel")
```

Arguments

x, p	Numeric vector of quantiles or probabilities.
Ct	Numeric scalar, clear sky radiation
alpha	Numeric scalar, parameter alpha > 0.
beta	Numeric scalar, parameter beta > 0 and alpha + beta < 1.
pdf_Y	Function, density of Y.
log	Logical, when TRUE, probabilities are returned as log(p).
cdf_Y	Function, distribution of Y.
log.p	Logical, when TRUE, probabilities p are given as log(p).
lower.tail	Logical, when TRUE, the default, the computed probabilities are $\mathbb{P}(X < x)$, otherwise $\mathbb{P}(X \geq x)$.

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.

$$R_t(y) = C(t)(1 - \alpha - \beta \exp(-\exp(y)))$$

where $R_t(y) \in [C(t)(1 - \alpha - \beta), C(t)(1 - \alpha)]$.

Version 1.0.0.

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 funtion `dsolarK` compute the density function of the following transformed random variable, i.e.

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

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

Version 1.0.0.

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 *Solar risk driver random variable*

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

Version 1.0.0.

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")
```

`dsugeno` *Sugeno-distorted density and distribution*

Description

Compute the Sugeno-distorted distribution for a given distribution or density.

Usage

```
dsugeno(pdf, cdf, lambda = 0)

psugeno(cdf, lambda = 0)
```

Arguments

`pdf`	Function, density function.
`cdf`	Function, distribution function.
`lambda`	Numeric, distortion parameter.

Details

Version 1.0.0.

Examples

```
# Distribution and density
cdf <- function(x) pnorm(x)
pdf <- function(x) dnorm(x)
x <- seq(-4, 4, 0.01)
plot(x, psugeno(cdf, lambda = -0.2)(x))
plot(x, dsugeno(pdf, cdf, lambda = -0.2)(x))
```

`dtnorm` *Truncated Normal random variable*

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 `length(n) > 1`, the length is taken to be the number required.

Details

Version 1.0.0.

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")
```

GARCH_modelR6 *Implementation of* rugarch *methods for a GARCH(p,q) as R6 class*

Description

Implementation of rugarch methods for a GARCH(p,q) as R6 class

Implementation of rugarch methods for a GARCH(p,q) as R6 class

Active bindings

spec A rugarch object containing the model's specification.

omega Numeric scalar, intercept parameter.

alpha Numeric vector, ARCH parameters.

beta Numeric vector, GARCH parameters.

order Numeric scalar, model's order.

coefficients Numeric vector, model's coefficients.

vol Numeric scalar, long-term unconditional std. deviation.

loglik model log-likelihood

tidy Tibble with estimated parameters and relative std. errors.

Methods

Public methods:

- GARCH_modelR6$new()
- GARCH_modelR6$fit()
- GARCH_modelR6$filter()
- GARCH_modelR6$logLik()
- GARCH_modelR6$update()
- GARCH_modelR6$update_hessian()
- GARCH_modelR6$update_std.errors()
- GARCH_modelR6$next_step()
- 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, update = FALSE, ...)
```

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.

`update` Logical. When true the internal log-likelihood will be updated.

`...` 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.

Method `update_hessian()`**:** Numerical computation of the Hessian matrix.

Usage:

```
GARCH_modelR6$update_hessian(coefficients, logLik, ...)
```

Arguments:

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

`logLik` Function, log-likelihood function depending on the parameters and `x`.

`...` Other arguments passed to `logLik` function.

Method `update_std.errors()`**:** Numerical computation of the std. errors of the parameters.

Usage:

```
GARCH_modelR6$update_std.errors(std.errors)
```

Arguments:

`std.errors` Numeric std. errors.

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 `print()`**:** Print method for `GARCH_modelR6` class. Manual fit of the GARCH model

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

`gaussianMixture` *Gaussian mixture*

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.

`maxrestarts` Integer, maximum number of restarts.

`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.

`use_empiric` logical to denote if empiric parameters are currently used

`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 theoric 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$E_step()`
- `gaussianMixture$classify()`
- `gaussianMixture$fit()`
- `gaussianMixture$EM()`
- `gaussianMixture$update()`
- `gaussianMixture$update_logLik()`
- `gaussianMixture$update_empiric_parameters()`
- `gaussianMixture$filter()`
- `gaussianMixture$Hessian()`
- `gaussianMixture$use_empiric_parameters()`
- `gaussianMixture$print()`
- `gaussianMixture$clone()`

Method `new()`**:** Initialize a `gaussianMixture` object.

Usage:

```
gaussianMixture$new(
  components = 2,
  maxit = 5000,
  maxrestarts = 500,
  abstol = 1e-08
)
```

Arguments:

`components` `(integer(1))`, number of components.

`maxit` `(integer(1))` Numeric, maximum number of iterations.

`maxrestarts` `(integer(1))` Numeric, maximum number of restarts.

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

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

Usage:

```
gaussianMixture$logLik(x, params, per_obs = FALSE)
```

Arguments:

`x` vector

`params` Optional. Named list with mixture parameters.

`per_obs` Logical, when `TRUE` the log-likelihood is returned per observation, otherwise is summed.

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

Usage:

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

Arguments:

`x` Time series to fit.

`params` A named list with mixture parameters.

Method `classify()`**:** Classify the time series in the components with highest likelihood.

Usage:

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

Arguments:

`x` Time series to fit.

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

Usage:

```
gaussianMixture$fit(
  x,
  weights,
  method = "mixtools",
  mu_target = NA,
  var_target = NA
)
```

Arguments:

`x` vector

`weights` observations weights, if a weight is equal to zero the observation is excluded, otherwise is included with unitary weight.

`method` Character, package used to fit the parameters. Can be `mclust` or `mixtools`.

`mu_target` Numeric, target mean of the mixture to match.

`var_target` Numeric, target variance of the mixture to match. When `missing` all the available observations will be used.

Method `EM()`**:** Fit the parameters (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 the parameters inside the object.

Usage:

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

Arguments:

`means` Numeric vector, means parameters.

`sd` Numeric vector, std. deviation parameters.

`p` Numeric vector, probabilities.

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

Usage:

`gaussianMixture$update_logLik()`

Method `update_empiric_parameters()`**:** Compute the parameters on the classified time series.

Usage:

`gaussianMixture$update_empiric_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 `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(0.5,2)
sd = rep(1, 2)
p = c(0.2,  0.8)
# 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$new(components=2)
# Fit the model
gm$fit(x$X)
gm
self <- gm$.__enclos_env__$self
private <- gm$.__enclos_env__$private
# EM-algo
gm$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 vector, latitudes of first location.
lon_1	Numeric vector, longitudes of first location.
lat_2	Numeric vector, latitudes of second location.
lon_2	Numeric vector, longitudes of second location.

Value

Vector of distances in kilometers.

Examples

```
havDistance(43.3, 12.1, 43.4, 12.2)
havDistance(c(43.35, 43.35), c(12.15, 12.1), c(43.4, 44.5), c(12.2, 13.4))
```

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` *Kernel regression*

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 — *Kolmogorov Smirnov test for a distribution*

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.
cdf	a function. The theoric distribution to use for comparison.
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`.
seed	random seed for two sample test.

Value

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

`ks_test_ts` *Two sample Kolmogorov Smirnov test for a time series*

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`.

`makeSemiPositive` *Make a matrix positive semi-definite*

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 — *Create a function of time for monthly parameters*

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 paramete

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 *Multivariate gaussian mixture*

Description

Multivariate gaussian mixture

Usage

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

---

number_of_day *Number of day*

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"))
```

PDF *Density, distribution and quantile function*

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))
```

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` An object of the class `solarModel`.

`measure` Character, reference probability measure used.

`lambda` Numeric, market risk premium used.

Methods

Public methods:

- `radiationModel$new()`
- `radiationModel$parametrize_seasonal_variance()`
- `radiationModel$correct_NM_coefficients()`
- `radiationModel$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$Moments()`
- `radiationModel$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, correction = FALSE)
```

Arguments:

`model` A `solarModel` object. See solarModel.

`correction` Logical. When `TRUE` the mixture means will be corrected for the discrepancy between the integral seasonal std. deviation and variance.

Method `parametrize_seasonal_variance()`**:** Compute the parameters of the seasonal variance given OLS estimates.

Usage:

```
radiationModel$parametrize_seasonal_variance()
```

Method `correct_NM_coefficients()`**:** Correct for the discrepancy between the integral seasonal std. deviation and variance.

Usage:

```
radiationModel$correct_NM_coefficients()
```

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 at time t_now.

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

Usage:

```
radiationModel$Yt_bar(t_now)
```

Arguments:

`t_now` Character, today date.

Returns: Seasonal mean for Y_t at time t_now.

Method `Rt_bar()`**:** 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 Rt.

Method `sigma_bar()`**:** Transformed variable instantaneous seasonal std. deviation $\bar{\sigma_t}$.

Usage:

`radiationModel$sigma_bar(t_now)`

Arguments:

`t_now` Character, today date.

Returns: Seasonal std. deviation for Yt on date t_now.

Method `mu_B()`**:** Transformed variable mixture mean drift $\mu_(B)$.

Usage:

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

Arguments:

`t_now` Character, today date.

`B` Integer. If `B = 1` it is returned the mean of the first component, otherwise if `B = 1` the second.

Returns: Mixture seasonal drift for Y_t at time t_now.

Method `sigma_B()`**:** Transformed variable mixture diffusion drift $\sigma_(B)$

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()`**:** Transformed variable drift $\mu_(Y)$.

Usage:

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

Arguments:

`Yt` Numeric, transformed solar radiation.

`t_now` Character, today date.

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

`dt` Numeric, time step.

Returns: Mixture drift for Y_t.

Method `sigma_Y()`**:** Transformed variable diffusion $\sigma_(Y)$.

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.

`Rt` Numeric, solar radiation.

Returns: Diffusion for Y_t.

Method `mu_R()`: Solar radiation drift $\mu_(R)$.

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 R_t.

Method `sigma_R()`: Solar radiation diffusion $\sigma_(R)$.

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 R_t.

Method `integral_expectation()`: Compute the integral for expectation $\mu_(t, T)$.

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 $\sigma^2_(t, T)$.

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()`: Integral mixture drift of both component of Y_t.

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 Y_t.

Method `e_mix_diffusion()`**:** Integral mixture diffusion of both component of Y_t.

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 Y_t.

Method `M_Y()`**:** Conditional expectation of Y_T given Y_t.

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 Y_t

Method `S_Y()`**:** Conditional variance of Y_T.

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 Y_t

Method `Moments()`**:** Compute the conditional moments

Usage:

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

Arguments:

`t_now` Character, today date.

`t_hor` Character, horizon date.

`quiet` Logical, when `TRUE` there won't be displayed any messages.

Method `pdf_Y()`**:** Conditional density of Y_T given Y_t.

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_T.

Method `cdf_Y()`**:** Conditional distribution of Y_T given Y_t.

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_T.

Method `pdf_R()`**:** Conditional density of R_T given R_t.

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_T

Method `cdf_R()`**:** Conditional distribution of R_T given R_t.

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_T

Method `e_GHI()`**:** Conditional expected value of R_T given R_t.

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()`**:** Conditional variance value of R_T given R_t.

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_T

Method `print()`**:** Method print for `radiationModel` object.

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 *R6 implementation for a clear sky seasonal model*

Description

R6 implementation for a clear sky seasonal model

R6 implementation for a clear sky seasonal model

Super class

`solarr::seasonalModel` -> `seasonalClearsky`

Public fields

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

Active bindings

`control` Named list, control parameters.

`ssf` Solar Seasonal Functions

Methods

Public methods:

- `seasonalClearsky$new()`
- `seasonalClearsky$fit()`
- `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 more 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 target clear sky radiation.

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

Usage:

```
seasonalClearsky$predict(n, newdata)
```

Arguments:

`n` Integer, scalar or vector. 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.

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.

Note

Version 1.0.0

`seasonalModel` *Seasonal Model*

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` Named vector, fitted coefficients.

`coefficients2` Named vector, reparametrized coefficients into a linear combination of shifted sine functions.

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

`period` Integer scalar, periodicity of the seasonality.

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

`std.errors` Named vector, with the parameters' std. errors.

`tidy` A tibble with estimated parameters and std. errors.

Methods

Public methods:

- seasonalModel$new()
- seasonalModel$fit()
- seasonalModel$fit_differential()
- seasonalModel$predict()
- seasonalModel$differential()
- seasonalModel$update()
- seasonalModel$update_std.errors()
- seasonalModel$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 `fit_differential()`**:** Fit the differential of the sinusoidal function.

Usage:

```
seasonalModel$fit_differential(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 vector, numbers of day of the year.

`newdata` 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.

`dt` Numeric, time step.

Method `differential()`**:** Compute the differential of the sinusoidal function.

Usage:

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

Arguments:

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

`newdata` 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.

`dt` Numeric, time step.

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

Usage:

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

Arguments:

`coefficients` Named vector, new parameters.

Method `update_std.errors()`**:** Update the parameter's std. errors.

Usage:

```
seasonalModel$update_std.errors(std.errors)
```

Arguments:

`std.errors` Named vector, new standard errors of the parameters.

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

Examples

```
sm <- seasonalModel$new(1, 365)
formula <- "Yt ~ 1"
data = data.frame(Yt = rnorm(1000), n = 1:1000)
sm$fit(formula, data = data)
sm
sm$coefficients
sm$update(sm$coefficients*3)
sm$predict(20)
```

`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

`Gsc` solar constant, i,e, `1367`.

Methods

Public methods:

- `seasonalSolarFunctions$new()`
- `seasonalSolarFunctions$update_method()`
- `seasonalSolarFunctions$B()`
- `seasonalSolarFunctions$degree()`
- `seasonalSolarFunctions$radiant()`
- `seasonalSolarFunctions$E()`
- `seasonalSolarFunctions$elevation()`
- `seasonalSolarFunctions$solar_time()`
- `seasonalSolarFunctions$solar_hour()`
- `seasonalSolarFunctions$hour_angle()`
- `seasonalSolarFunctions$incidence_angle()`
- `seasonalSolarFunctions$azimut_angle()`
- `seasonalSolarFunctions$Gon()`
- `seasonalSolarFunctions$declination()`
- `seasonalSolarFunctions$Hon()`
- `seasonalSolarFunctions$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 implement Eq. 1.4.2 from Duffie (4th edition), i.e.

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

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

Usage:

```
seasonalSolarFunctions$degree(x)
```

Arguments:

`x` numeric vector, angles in radiant.

Details: The function computes:

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

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

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 `elevation()`**:** Compute the angle in the degree given a certain altitude in meters.

Usage:

`seasonalSolarFunctions$elevation(alt)`

Arguments:

`alt` Numeric, altitude in meters.

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

Usage:

`seasonalSolarFunctions$solar_time(x, lon, lon_st = 15, tz = "Europe/Rome")`

Arguments:

`x` datetime, clock hour.

`lon` longitude of interest in degrees.

`lon_st` longitude of the Local standard meridian in degrees.

`tz` Character, reference time zone.

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

$$solartime = clocktime + 4(lon - lon_{st}) + E(n)$$

Returns: A datetime object

Method `solar_hour()`**:** Compute the solar hour for a specific clock time.

Usage:

`seasonalSolarFunctions$solar_hour(LST)`

Arguments:

`LST` datetime, true solar time.

Returns: Hours

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

Usage:

`seasonalSolarFunctions$hour_angle(LST)`

Arguments:

`LST` datetime, true solar time.

Details: The function implement Eq. 1.42 from Comini (2013), i.e.

$$\omega = 15(solarhour - 12)$$

where the "solarhour" is expressed in hours.

Returns: An angle in degrees

Method `incidence_angle()`: Compute the incidence angle

Usage:

```
seasonalSolarFunctions$incidence_angle(LST, lat, alt = 0, beta = 0, gamma = 0)
```

Arguments:

`LST` datetime, true solar time.

`lat` latitude of interest in degrees.

`alt` Numeric, altitude in meters.

`beta` altitude

`gamma` orientation

Returns: An angle in degrees

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

Usage:

```
seasonalSolarFunctions$azimut_angle(LST, lat, alt, beta = 0, gamma = 0)
```

Arguments:

`LST` datetime, true solar time.

`lat` latitude of interest in degrees.

`alt` Numeric, altitude in meters.

`beta` altitude

`gamma` orientation

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

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

Returns: The solar azimut angle in degrees

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

Usage:

```
seasonalSolarFunctions$Gon(n, deriv = FALSE)
```

Arguments:

`n` number of the day of the year.

`deriv` Logical, when `TRUE` will return the first derivative with respect to time.

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

$$G_{o,n} = G_{sc}(1 + 0.033\cos(B))$$

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

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

When `deriv = TRUE` it will be returned the derivatives with respect to time. When the method is `cooper`:

$$\frac{\partial G_{o,n}}{\partial n} = -G_{sc}\frac{2\pi}{365}0.033\sin(B))$$

Otherwise if it is `spencer`:

$$\frac{\partial G_{o,n}}{\partial n} = G_{sc}\frac{2\pi}{365}(-0.034221\sin(B) + 0.001280\cos(B) - 0.001438\sin(2B) + 0.000154\cos(2B))$$

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

Method `declination()`**:** Compute solar declination in degrees.

Usage:

```
seasonalSolarFunctions$declination(n, deriv = FALSE)
```

Arguments:

`n` number of the day of the year

`deriv` Logical, when `TRUE` will return the first derivative with respect to time.

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):

$$\delta(n) = \frac{180}{\pi}(0.006918 - 0.399912\cos(B) + 0.070257\sin(B) - 0.006758\cos(2B) + 0.000907\sin(2B) - 0.002697\cos$$

When `deriv = TRUE` it will be returned the derivatives with respect to time. When the method is `cooper`:

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

otherwise when the method is `spencer`:

$$\frac{\partial \delta}{\partial n}(n) = \frac{360}{365}(0.399912\sin(B) + 0.070257\cos(B) + 0.013516\sin(2B) + 0.001814\cos(2B) + 0.008091\sin(3B) + 0.$$

Returns: The solar declination in degrees.

Method `Hon()`**:** Compute the solar extraterrestrial radiation

Usage:

```
seasonalSolarFunctions$Hon(n, lat, alt, deriv = FALSE)
```

Arguments:

`n` number of the day of the year

`lat` latitude of interest in degrees.

`alt` Numeric, altitude in meters.

`deriv` Logical, when `TRUE` will return the first derivative with respect to time.

Details: The function implement Eq. 1.10.3 from Duffie (4th edition):

$$H_{on} = G_{on}\frac{24 \times 3600}{\pi}(\cos(lat)\cos(\delta)\sin(\omega_s) + \frac{\pi}{180}\sin(lat)\sin(\delta))$$

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

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

Usage:

```
seasonalSolarFunctions$sunset_hour_angle(n, lat, alt, deriv = FALSE)
```

Arguments:

`n` number of the day of the year

`lat` Numeric, latitude of interest in degrees.

`alt` Numeric, altitude in meters.

`deriv` Logical, when `TRUE` will return the first derivative with respect to time.

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

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

When altitude is not missing it will implement a generalized version with altitude, i.e.

$$\omega_s = \cos^{-1}\left(\frac{\sin H - \sin\delta\sin\phi}{\cos\phi\cos\delta}\right)$$

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, alt)
```

Arguments:

`n` number of the day of the year.

`lat` Numeric, latitude of interest in degrees.

`alt` Numeric, altitude in meters.

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,
  alt,
  lon_st = 15,
  beta = 0,
  gamma = 0,
  by = "1 min",
  tz = "Europe/Rome"
)
```

Arguments:

`x` datetime, clock hour.

`lat` Numeric, latitude of interest in degrees.

`lon` Numeric, longitude of interest in degrees.

`alt` Numeric, altitude in meters.

`lon_st` Numeric, longitude of the Local standard meridian in degrees

`beta` Numeric angle, inclination of the solar panel.

`gamma` Numeric, angle orientation of the panel.

`by` Character, time step. Default is `1 min`.

`tz` Character, reference time zone.

Method `clearsky()`**:** Hottel clearsky

Usage:

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

Arguments:

`cosZ` solar incidence angle

`G0` solar constant

`alt` Numeric, altitude in meters.

`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$Gsc

# Solar constant adjusted
sf$Gon(dates)

# Extraterrestrial radiation
sf$Hon(dates, 43)

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

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

sf$solar_time("2022-01-01 12:00", 11, 10)
sf$hour_angle("2022-01-01 14:00", 11, 15)
sf$incidence_angle("2022-06-01 21:00", 31, 12, lon_st = 15, beta = 0, gamma = 0)
sf$azimut_angle("2022-01-01 14:00", 30, 17, lon_st = 15)
```

`solarDiscount` *Discount factor of a Solar Option*

Description

Discount factor of a Solar Option

Usage

```
solarDiscount(tau, P = 1, Gamma_h = 0, r = 0.03/365)
```

Arguments

`tau`	Numeric, time to maturity in days.
`P`	Numeric, price of the contract.
`Gamma_h`	Numeric, hedged payoff.
`r`	Numeric, daily risk-free rate.

Details

The discount factor reads:

$$B(\tau, P, \Gamma^h, r) = e^{-r\tau} + \frac{\Gamma^h}{P}(1 - e^{-r\tau})$$

Note

Version 1.0.0.

Examples

```
solarDiscount(365. 0.6, 2, 0.000008)
solarDiscount(365, 0.3, 2, 0.00008)
```

solarHedging_model *Global Minimum Variance number of contracts*

Description

Compute the optimal number of contracts, such that the variance of the cash flow of a solar power producer is minimum.

Usage

```
solarHedging_model(
  model,
  moments,
  P0_P,
  r_star,
  gamma = 0.01,
  put = TRUE,
  control_options = control_solarOption(),
  control_hedge = control_solarHedging()
)
```

Arguments

model	An object with the class solarModel. See the function solarModel for more details.
moments	Tibble containing the forecasted moments for different days ahead. See the function solarMoments for more details.
P0_P	Optional numeric scalar, expected value of 1 solar derivative with unitary tick under $\mathbb{P}$.
gamma	Numeric scalar, risk aversion parameter.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.
control_hedge	Named list, control parameters for hedging. See the function control_solarHedging for more details.

Note

Version 1.0.0.

solarHedging_scenarios — *Compute the optimal number of solar derivative index*

Description

Compute the optimal number of contracts, such that the variance of the cash flow of a solar power producer with a given setup is minimum.

Usage

```
solarHedging_scenarios(
  scenarios,
  P0_P,
  r_star = 0,
  gamma = 1e-04,
  put = TRUE,
  control_options = control_solarOption(),
  control_hedge = control_solarHedging()
)
```

Arguments

scenarios	Object with the class solarScenario. See the function solarScenario for more details.
P0_P	Optional numeric scalar, expected value of 1 solar derivative with unitary tick under $\mathbb{P}$.
gamma	Numeric scalar, risk aversion parameter.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.
control_hedge	Named list, control parameters for hedging. See the function control_solarHedging for more details.

Note

Version 1.0.0.

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.

`maxrestarts` Integer, maximum number of restarts.

`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.

`use_empiric` logical to denote if empiric parameters are currently used

`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$update()`
- `solarMixture$update_logLik()`

- solarMixture$update_empiric_parameters()
- solarMixture$filter()
- solarMixture$Hessian()
- solarMixture$use_empiric_parameters()
- solarMixture$logLik()
- solarMixture$grades()
- solarMixture$VaR()
- solarMixture$ES()
- solarMixture$print()
- solarMixture$clone()

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

Usage:

```
solarMixture$new(
  components = 2,
  maxit = 5000,
  maxrestarts = 500,
  abstol = 1e-08
)
```

Arguments:

`components` (`integer(1)`), number of components.

`maxit` (`integer(1)`) Numeric, maximum number of iterations.

`maxrestarts` (`integer(1)`) Numeric, maximum number of restarts.

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

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

Usage:

```
solarMixture$fit(
  x,
  date,
  weights,
  method = "mixtools",
  mu_target = rep(NA, 12),
  var_target = rep(NA, 12)
)
```

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` Character, package used to fit the parameters. Can be `mclust` or `mixtools`.

`mu_target` Numeric vector with length 12, target mean of the mixture to match.

`var_target` Numeric vector with length 12, target variance of the mixture to match.

Method `update()`**:** Update means, sd, p .

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 `update_logLik():` Apply the `$update_logLik()` method to all the `gaussianMixture` models

Usage:

`solarMixture$update_logLik()`

Method `update_empiric_parameters():` Apply the `$update_empiric_parameters()` method to all the `gaussianMixture` models

Usage:

`solarMixture$update_empiric_parameters()`

Method `filter():` Apply the `$filter()` method to all the `gaussianMixture` models

Usage:

`solarMixture$filter()`

Method `Hessian():` Apply the `$Hessian()` method to all the `gaussianMixture` models

Usage:

`solarMixture$Hessian()`

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

Usage:

`solarMixture$use_empiric_parameters()`

Method `logLik():` Log-likelihood

Usage:

`solarMixture$logLik(x, date)`

Arguments:

`x` vector

`date` dates

Method `grades():` Compute the grades

Usage:

`solarMixture$grades(x, date)`

Arguments:

`x` vector

`date` dates

Method `VaR():` Compute the VaR with certain confidence levels

Usage:

`solarMixture$VaR(date, alpha = 0.05)`

Arguments:

`date` dates

`alpha` confidence levels for the VaR

`x` vector

Method `ES()`**:** Compute the VaR with certain confidence levels

Usage:

`solarMixture$ES(date, alpha = 0.05)`

Arguments:

`date` dates

`alpha` confidence levels for the VaR

`x` vector

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

```
# Model fit
model <- solarModel$new(spec)
model$fit()
# Inputs
x <- model$data$u_tilde
w <- model$data$weights
date <- model$data$date
# Solar Mixture object
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)])
```

`solarMixture_VaR` *Compute the Value at Risk and Expected Shortfall of a SolarMixture*

Description

Compute the Value at Risk and Expected Shortfall of a SolarMixture

Usage

```
solarMixture_VaR(solarMix, date, x, alpha = 0.05, ci = 0.05, ES = FALSE)
```

Arguments

alpha	Numeric vector of confidence levels. Allows for more than one alpha.
ci	Numeric scalar, confidence levels used to state if the Null is rejected or not on VaR tests.
ES	Logical, when TRUE the expected shortfall will be also computed for each alpha.
model	solarMixture
type	Numeric, type of evaluation, full on the complete data, train on the train data, test on the test data.

---

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.

Active bindings

place Character, optional name of the location considered.

model_name Character, model's name.

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.

spec A list with the specification 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.

var_theta Variance-covariance matrix of the parameters with robust std. errors.

Methods

Public methods:

- solarModel$new()
- solarModel$fit()
- solarModel$fit_seasonal_model_Ct()
- solarModel$compute_risk_drivers()
- solarModel$fit_transform()
- solarModel$fit_seasonal_model_Yt()
- solarModel$fit_monthly_mean()
- solarModel$fit_ARMA()
- solarModel$fit_seasonal_variance()
- solarModel$fit_monthly_variance()
- solarModel$correct_seasonal_variance()
- solarModel$fit_GARCH()
- solarModel$fit_NM_model()
- solarModel$update()
- solarModel$update_moments()
- solarModel$update_logLik()
- solarModel$update_risk_drivers()
- solarModel$update_NM_classification()
- solarModel$filter()
- solarModel$Moments()
- solarModel$VaR()
- solarModel$logLik()
- solarModel$R_to_Y()
- solarModel$Y_to_R()
- solarModel$print()
- solarModel$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 `fit_seasonal_model_Ct()`**:** Initialize and fit a `seasonalClearsky` model given the specification contained in `$control`.

Usage:

```
solarModel$fit_seasonal_model_Ct()
```

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_transform()`**:** Fit the parameters of the `solarTransform` object.

Usage:

```
solarModel$fit_transform()
```

Method `fit_seasonal_model_Yt()`**:** Fit a `seasonalModel` the transformed variable (`Yt`) and compute deseasonalized series (`Yt_tilde`).

Usage:

```
solarModel$fit_seasonal_model_Yt()
```

Method `fit_monthly_mean()`**:** Correct the deseasonalized series (`Yt_tilde`) by subtracting its monthly mean (`Yt_tilde_uncond`).

Usage:

```
solarModel$fit_monthly_mean()
```

Method `fit_ARMA()`**:** Fit an AR model (`Yt_tilde`) and compute AR residuals (`eps`).

Usage:

```
solarModel$fit_ARMA()
```

Method `fit_seasonal_variance()`**:** Fit a `seasonalModel` on AR squared residuals (`eps`) and compute deseasonalized residuals `eps_tilde`.

Usage:

```
solarModel$fit_seasonal_variance()
```

Method `fit_monthly_variance()`**:** Correct the standardized series (`eps_tilde`) by subtracting its monthly mean (`sigma_uncond`).

Usage:

```
solarModel$fit_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()`**:** Fit a `GARCH` model on the deseasonalized residuals (`eps_tilde`). Compute the standardized (u) and monthly deseasonalized residuals (`u_tilde`).

Usage:

```
solarModel$fit_GARCH()
```

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

Usage:

```
solarModel$fit_NM_model()
```

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

Usage:

```
solarModel$update(params)
```

Arguments:

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

Method `update_moments()`**:** Update the moments inside object

Usage:

`solarModel$update_moments()`

Method `update_logLik()`**:** Update the log-likelihood inside object

Usage:

`solarModel$update_logLik()`

Method `update_risk_drivers()`**:** Update the clear sky and risk drivers

Usage:

`solarModel$update_risk_drivers()`

Method `update_NM_classification()`**:** Update the classification of the Bernoulli random variable.

Usage:

`solarModel$update_NM_classification(filter = FALSE)`

Arguments:

`filter` Logical, when `TRUE` before the classification will be runned the command `self$NM_model$filter()` to update the mixture classification.

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

Usage:

`solarModel$filter(fit = TRUE)`

Arguments:

`fit` Logical, when `TRUE`, if in the model's specification, the monthly mean and variances will be re estimated and the seasonal variance corrected such that the total variance of the deseasonalized residuals is zero.

Returns: Update the slots `$data`, `$seasonal_data`, `$monthly_data`

Method `Moments()`**:** Compute the conditional moments

Usage:

`solarModel$Moments(t_now, t_hor, theta = 0, quiet = FALSE)`

Arguments:

`t_now` Character date. Today date.

`t_hor` Character date. Horizon date.

`theta` Numeric, shift parameter for the mixture.

`quiet` Logical for verbose messages.

Method `VaR()`**:** Value at Risk for a `solarModel`

Usage:

`solarModel$VaR(moments, t_now, t_hor, theta = 0, ci = 0.05)`

Arguments:

`moments` moments dataset

`t_now` Character date. Today date.

`t_hor` Character date. Horizon date.

`theta` Numeric, shift parameter for the mixture.

`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", quasi = FALSE)
```

Arguments:

`moments` Dataset containing the moments to use for computation.

`target` Character. Target variable to use "Yt" or "GHI".

`quasi` Logical, when `TRUE` is computed the pseudo-likelihood with Gaussian link.

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 `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

```
# Model specification
spec <- solarModel_spec$new()
spec$set_mean.model(arOrder = 1, maOrder = 1)
spec$set_seasonal.mean(monthly.mean = FALSE)
spec$set_seasonal.variance(monthly.mean = FALSE)
spec$specification("Bologna")
spec
# Model fit
Bologna <- solarModel$new(spec)
Bologna$fit()
Bologna
# save(spec, 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_grid` *Fit a grid of solarModels*

Description

Fit a grid of solarModels

Usage

```
solarModels_grid(
  spec,
  arOrder = 2,
  maOrder = 2,
  archOrder = 1,
  garchOrder = 1,
  QMLE = FALSE
)
```

Arguments

`spec`	Specification
`arOrder`	Numeric, maximum AR order.
`maOrder`	Numeric, maximum MA order.
`archOrder`	Numeric, maximum ARCH order.
`garchOrder`	Numeric, maximum GARCH order.
`place`	Reference location

Note

Version 1.0.0.

Examples

```
spec <- solarModel_spec$new()
models <- solarModels_grid(spec, "Bologna", 1,1,1,1)
models[which.min(models$L),]
```

`solarModel_AIC_BIC` *Compute the AIC and BIC of a solarModel object*

Description

Compute the AIC and BIC of a solarModel object

Usage

```
solarModel_AIC_BIC(model, target = "GHI", type = c("train", "test", "full"))
```

Arguments

`model`	solarmodel

Note

Version 1.0.0.

`solarModel_calibrate_theta` *Calibrate theta to match a certain level of solar radiation*

Description

Calibrate theta to match a certain level of solar radiation

Usage

```
solarModel_calibrate_theta(model, moments, e_RT_target, quiet = FALSE)
```

Arguments

`e_RT_target`	Numeric, vector of target expectation to match.

Details

Version 1.0.0.

`solarModel_covariance` *Compute the covariance*

Description

Compute the covariance

Usage

```
solarModel_covariance(
  t_now,
  mom_t,
  mom_T,
  GARCH,
  NM_model,
  theta = 0,
  tol = 0.01
)
```

Note

Version 1.0.0.

solarModel_forecast *Iterate the forecast on multiple dates*

Description

Iterate the forecast on multiple dates

Usage

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

Note

Version 1.0.0.

solarModel_match_params *Match solarModel parameters in vector form*

Description

Match solarModel parameters in vector form

Usage

```
solarModel_match_params(vec_params, params)
```

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
vec_params <- c(theta = 1, alpha1 = 10)
solarModel_match_params(vec_params, model$coefficients)
```

solarModel_predict — *Produce a forecast from a solarModel object*

Description

Produce a forecast from a solarModel object

Usage

```
solarModel_predict(model, moments, lambda = 0, ci = 0.01)
```

Arguments

lambda	Numeric scalar, Sugeno parameter.

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
moments <- model$moments$conditional[14,]
object <- solarModel_predict(model, moments, ci = 0.01)
object
```

solarModel_predict_plot — *Plot a forecast from a solarModel object*

Description

Plot a forecast from a solarModel object

Usage

```
solarModel_predict_plot(object, pdf_components = FALSE, type = "mix")
```

Examples

```
model = solarModel$new(spec)
model$fit()
df_n <- model$moments$conditional[23,]
solarModel_predict_plot(solarModel_predict(model, df_n, ci = 0.01))
```

solarModel_QMLE *QMLE Estimate*

Description

QMLE Estimate

Usage

```
solarModel_QMLE(model, maxrestarts = 1, seed = 1, quiet = TRUE)
```

Note

Version 1.0.0.

solarModel_selection *Select the Best Model*

Description

Select the Best Model

Usage

```
solarModels_selection(
  spec,
  arOrder = 2,
  maOrder = 2,
  archOrder = 1,
  garchOrder = 1
)
```

Arguments

spec	specification

Note

Version 1.0.0.

`solarModel_spec` *Control function for a* `solarModel` *object*

Description

Control function for a `solarModel` object

Control function for a `solarModel` object

Details

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

Active bindings

`place` Character, optional name of the location considered.

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

`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.

`dates` A named list, with three sub-lists: `data` containing the information on the complete dataset, `train` containing the information on the train dataset and `test` containing the information on the test dataset. Each sub-list is structured as follows:

- **from** Character date, minmum date in the dataset.
- **to** Character date, maximum date in the dataset.
- **nobs** Integer scalar, number of observations contained in the dataset between `from` and `to`.
- **perc** Numeric scalar, percentage of data in the dataset with respect to the complete data.

`data` Tibble, dataset with CAMS solar radiation data.

`transform` Named list, specification of the solar transform.

`clearsky` Named list, specification of the clear sky model.

`seasonal.mean` Named list, specification of the seasonal model.

`mean.model` Named list, specification of the ARMA model.

`seasonal.variance` Named list, specification of the seasonal variance model.

`variance.model` Named list, specification of the GARCH model for deseasonalized residuals $\tilde{e}_t$.

`mixture.model` Named list, specification of the Mixture model for GARCH residuals u_t.

`garch_variance` Logical, when `TRUE` the GARCH model will be used otherwise no.

`clearsky_threshold` Numeric, parameter > 1, used to scale up CAMS clearsky.

`stochastic_clearsky` Logical, when `TRUE` the clear sky is considered stochastic.

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

Methods

Public methods:

- solarModel_spec$new()
- solarModel_spec$specification()
- solarModel_spec$set_params()
- solarModel_spec$set_transform()
- solarModel_spec$set_clearsky()
- solarModel_spec$set_seasonal.mean()
- solarModel_spec$set_mean.model()
- solarModel_spec$set_seasonal.variance()
- solarModel_spec$set_variance.model()
- solarModel_spec$set_mixture.model()
- solarModel_spec$print()
- solarModel_spec$clone()

Method `new()`**:** Initialize a `solarModel_spec` object.

Usage:

```
solarModel_spec$new()
```

Method `specification()`**:** Specification function for a `solarModel`

Usage:

```
solarModel_spec$specification(
  place,
  target = "GHI",
  min_date,
  max_date,
  from,
  to,
  data
)
```

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`.

`data` data for the selected location.

Method `set_params()`**:** Generic controls

Usage:

```
solarModel_spec$set_params(
  stochastic_clearsky = FALSE,
  clearsky_threshold = 1.01,
  quiet = FALSE
)
```

Arguments:

`stochastic_clearsky` Logical, when `TRUE` the clear sky will be considered stochastic.

`clearsky_threshold` Numeric, parameter > 1, used to scale up CAMS clearsky to avoid that clear sky radiaion and global horizontal radiation are equal.

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

Method `set_transform()`**:** Control parameters for the `solarTransform`. See `solarTransform` for more details.

Usage:

```
solarModel_spec$set_transform(
  min_pos = 1,
  max_pos = 1,
  link = "invgumbel",
  delta = 0.05,
  threshold = 0.01
)
```

Arguments:

`min_pos` Integer, position of the minimum. For example when `2` the minimum is the second lowest value.

`max_pos` Integer, position of the maximum. For example when `3` the maximum is the third greatest value.

`link` Character, link function.

`delta` transform params

`threshold` Numeric. Threshold used to estimate the transformation parameters

$$\alpha$$

and

$$\beta$$

. The default is `0.01`. See `solarTransform` for more details.

Method `set_clearsky()`**:** List with specification's parameters of the clear sky model.

Usage:

```
solarModel_spec$set_clearsky(control = control_seasonalClearsky())
```

Arguments:

`control` Named list with control parameters. See `control_seasonalClearsky` for more details.

Method `set_seasonal.mean()`**:** List with specification's parameters of the seasonal mean $\bar{Y}_t$ for Y_t.

Usage:

```
solarModel_spec$set_seasonal.mean(
  order = 1,
  period = 365,
  include.trend = FALSE,
  include.intercept = TRUE,
  monthly.mean = TRUE
)
```

Arguments:

`order` Integer. Specify the order of the seasonal mean

$$\bar{Y}_t$$

. The default is `1`.

`period` Integer, seasonal periodicity, the default is `365`.

`include.trend` Logical. When `TRUE` an yearly trend

$$t$$

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

`include.intercept` Logical. When `TRUE` the intercept

$$a_0$$

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

`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 dafault if `TRUE`.

Method `set_mean.model()`**:** List with specification's parameters of the ARMA model for deseasonalized series $\tilde{Y}_t = Y_t - \bar{Y}_t$.

Usage:

```
solarModel_spec$set_mean.model(
  arOrder = 1,
  maOrder = 0,
  include.intercept = FALSE
)
```

Arguments:

`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` Logical. When `TRUE` the intercept

$$\phi_0$$

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

Method `set_seasonal.variance()`**:** List with specification's parameters of the seasonal variance $\bar{\sigma}_t$ for ARMA's residuals e_t

Usage:

```
solarModel_spec$set_seasonal.variance(
  order = 1,
  period = 365,
  include.trend = FALSE,
  correction = TRUE,
  monthly.mean = TRUE
)
```

Arguments:

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

`period` Integer, seasonal periodicity, the default is `365`.

`include.trend` Logical. When `TRUE` an yearly trend

$$t$$

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

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

Method `set_variance.model()`**:** List with specification's parameters of the GARCH variance σ_t for deseasonalized residuals $\tilde{e}_t = e_t/\bar{\sigma}_t$.

Usage:

```
solarModel_spec$set_variance.model(
  archOrder = 1,
  garchOrder = 1,
  garch_variance = TRUE
)
```

Arguments:

`archOrder` Integer. An integer specifying the order of the ARCH component. The default is `1`.

`garchOrder` Integer. An integer specifying the order of the GARCH component. The default is `1`.

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

Method `set_mixture.model()`**:** List with specification's parameters of the Gaussian mixture model for GARCH residuals $u_t = \tilde{e}_t/\sigma_t$.

Usage:

```
solarModel_spec$set_mixture.model(
  abstol = 1e-20,
  match.expectation = TRUE,
  match.variance = FALSE,
  match.empiric = FALSE,
  method = "mclust",
  maxit = 5000,
  maxrestarts = 500
)
```

Arguments:

`abstol` Numeric. Absolute level for convergence of the EM-algorithm. The default is `1e-20`.

`match.expectation` Logical, when `TRUE` the mixture parameters ensures that the expected value is matched.

`match.variance` Logical, when `TRUE` the mixture parameters ensures that the variance is matched.

`match.empiric` Logical, when `TRUE` and `match.expectation = TRUE` or `match.variance = TRUE` the mixture parameters will be estimated ensuring that mean and variance matches the empirical parameters. Otherwise if `FALSE` and `match.expectation = TRUE` or `match.variance = TRUE` the target expectation will be zero and the target variance 1.

`method` Character, package used to fit the parameters. Can be `mclust` or `mixtools`.

`maxit` Integer. Maximum number of iterations for EM-algorithm. The default is `5000`.

`maxrestarts` Integer. Maximum number of restarts when EM-algorithm does not converge. The default is `500`.

Method `print()`**:** Print method for `solarModel_spec` class.

Usage:

```
solarModel_spec$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

Version 1.0.0.

Examples

```
control <- solarModel_spec$new()
```

`solarModel_tests` *Autocorrelation and distribution tests*

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,
  lags = c(7),
  ci = 0.05,
  min_quantile = 0.025,
  max_quantile = 0.985,
  method = "bg",
  type = c("train", "test", "full")
)
```

Arguments

`model`	An object of the class `solarModel`
`lags`	Numeric vector. Lags on which perform the autocorrelation tests. Can be more than one.
`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 `"bg"` for Breusch-Godfrey, `"bp"` for Box-pierce and `"lb"` for BLjung-Box.
`type`	Type of test.

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
solarModel_tests(model, train_data = TRUE)
```

`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 = c("bg", "bp", "lb"),
  type = c("train", "test", "full")
)
```

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 `"bg"` for Breusch-Godfrey, `"bp"` for Box-pierce and `"lb"` for BLjung-Box.
`train_data`	Logical, when `TRUE` only train data will be used to evaluate the test, otherwise all the available sample.

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
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,
  H0 = c("gm", "norm"),
  ci = 0.05,
  min_quantile = 0.025,
  max_quantile = 0.985,
  type = c("train", "test", "full")
)
```

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.
`type`	Type of test.
`test`	Character, null hypothesis for the residuals distribution `"gm"` for Gaussian mixture and `"norm"` for normality.

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
solarModel_test_distribution(model)
```

solarModel_test_forecast *Compute metrics to test forecasts*

Description

Compute metrics to test forecasts

Usage

```
solarModel_test_forecast(model, ci = 0.1, type = c("train", "test", "full"))
```

Note

Version 1.0.0.

solarModel_test_LPD *Compute the Log-predictive density of a solarModel*

Description

Compute the Log-predictive density of a solarModel

Usage

```
solarModel_test_LPD(model, type = c("train", "test", "full"))
```

Note

Version 1.0.0.

solarModel_test_PIT *Probability Integral Transform*

Description

Probability Integral Transform

Usage

```
solarModel_test_PIT(model, ci = 0.05, type = c("train", "test", "full"))
```

Note

Version 1.0.0.

solarModel_test_pricing

Compute metrics to test option pricing

Description

Compute metrics to test option pricing

Usage

```
solarModel_test_pricing(
  model,
  type = c("train", "test", "full"),
  control = control_solarOption()
)
```

Note

Version 1.0.0.

solarModel_VaR

Compute the Value at Risk and Expected Shortfall of a SolarModel

Description

Compute the Value at Risk and Expected Shortfall of a SolarModel

Usage

```
solarModel_VaR(model, alpha = 0.05, ci = 0.05, ES = FALSE, type = "full")
```

Arguments

`model`	solarModel
`alpha`	Numeric vector of confidence levels. Allows for more than one alpha.
`ci`	Numeric scalar, confidence levels used to state if the Null is rejected or not on VaR tests.
`ES`	Logical, when `TRUE` the expected shortfall will be also computed for each alpha.
`type`	Numeric, type of evaluation, `full` on the complete data, `train` on the train data, `test` on the test data.

solarMoments *Compute the generic conditional moments of a solarModel object*

Description

Compute the generic conditional moments of a solarModel object

Usage

```
solarMoments(
  t_now,
  t_hor,
  data,
  ARMA,
  GARCH,
  NM_model,
  transform,
  theta = 0,
  quiet = FALSE
)
```

Arguments

t_now	Date for today.
t_hor	Horizon date.
data	Slot data of a solarModel object. See the function solarModel for details.
ARMA	Slot ARMA of a solarModel object.
GARCH	Slot GARCH of a solarModel object.
NM_model	Slot NM_model of a solarModel object.
transform	Slot transform of a solarModel object.
theta	Numeric, vector of shift parameters for the Gaussian mixture residuals.

Details

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
t_now = "2019-07-11"
t_hor = "2019-10-19"
data = model$data
ARMA = model$ARMA
GARCH  = model$GARCH
NM_model = model$NM_model
transform = model$transform
theta = 0
solarMoments(t_now, t_hor, data, ARMA, GARCH, NM_model, transform, theta = 0, quiet = FALSE)
filter(model$moments$conditional, date == t_hor)
```

```
filter(model$moments$unconditional, date == t_hor)

t_seq <- seq.Date(as.Date("2013-01-01"), as.Date("2013-12-31"), 1)
mom <- purrr::map_df(t_seq, ~solarr::solarMoments(.x-150, .x, data, ARMA, GARCH, NM_model, transform, theta = (
solarOption_model(model, mom, control_options = control_solarOption(nyears = c(2005, 2024)))
solarOption_historical(model, control_options = control_solarOption(nyears = c(2005, 2024)))
```

---

solarMoments_conditional — *Compute the conditional moments*

Description

Compute the conditional moments

Usage

```
solarMoments_conditional(data, theta = 0, control_model)
```

Arguments

`data`	Slot `data` of a `solarModel` object. See the function `solarModel` for details.
`theta`	Numeric, shift parameter for the Gaussian mixture residuals.
`control_model`	An object with the class `solarModel_spec`. See the function `solarModel_spec` for details.

Details

Version 1.0.0.

---

solarMoments_path — *Condition the moments for a specific Bernoulli realization at a time t_cond*

Description

Condition the moments for a specific Bernoulli realization at a time t_cond

Usage

```
solarMoments_path(moments, GARCH, NM_model, theta = 0, B = 1, t_cond)
```

Arguments

`GARCH`	Slot `GARCH` of a `solarModel` object.
`NM_model`	Slot `NM_model` of a `solarModel` object.
`theta`	Numeric, vector of shift parameters for the Gaussian mixture residuals.
`B`	conditioning value for the Bernoulli state at time thor
`t_cond`	conditioning date

Details

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
GARCH  = model$GARCH
NM_model = model$NM_model
# Compute the moments
moments <- model$Moments("2012-01-01", "2012-03-05")

# Condition the moments on a realization of B
moments
solarOption_model(model, moments)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = 1, "2012-02-03")
solarOption_model(model, mom_cond)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = 1, "2012-03-01")
solarOption_model(model, mom_cond)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = 1, "2012-03-04")
solarOption_model(model, mom_cond)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = 0, "2012-03-04")
solarOption_model(model, mom_cond)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = c(0, 0), c("2012-03-03", "2012-03-04"))
solarOption_model(model, mom_cond)

mom_cond <- solarMoments_path(moments, GARCH, NM_model, theta = 0, B = c(1, 1), c("2012-03-03", "2012-03-04"))
solarOption_model(model, mom_cond)
```

---

solarMoments_unconditional

Compute the unconditional moments

Description

Compute the unconditional moments

Usage

```
solarMoments_unconditional(data, ARMA, GARCH, theta = 0)
```

Arguments

data	Slot data of a solarModel object. See the function solarModel for details.
ARMA	Slot ARMA of a solarModel object.
GARCH	Slot GARCH of a solarModel object.
theta	Numeric, shift parameter for the Gaussian mixture residuals.

Details

Version 1.0.0.

`solarOption` *Create a SoRad / SoREd contract specification*

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.

`solarOptionPayoff` *Structure the outputs of solarOption functions*

Description

Structure the outputs of solarOption functions

Usage

```
solarOptionPayoff(data, leap_year = FALSE)
```

Arguments

`data`	df_payoff
`leap_year`	control params

Value

An object of the class `solarOptionPayoff`.

Note

Version 1.0.0.

`solarOption_calibrate_theta` *Calibrate the time series of theta to match a certain level of Option price*

Description

Calibrate the time series of theta to match a certain level of Option price

Usage

```
solarOption_calibrate_theta(
  model,
  moments,
  P_target,
  put = TRUE,
  quiet = FALSE,
  control_options = control_solarOption()
)
```

Arguments

model	An object with the class solarModel. See the function solarModel for details.
P_target	Numeric, vector of target prices to match.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.

Details

Version 1.0.0.

`solarOption_choquet` *Compute Choquet price for a Solar Option*

Description

Compute Choquet price for a Solar Option

Usage

```
solarOption_choquet(
  model,
  moments,
  portfolio,
  nmonths = 1:12,
  lambda = 0,
```

```
    implvol = 1,
    put = TRUE,
    control_options = control_solarOption()
)
```

Arguments

model	An object with the class solarModel. See the function solarModel for details.
moments	Tibble containing the forecasted moments for different days ahead. See the function solarMoments for more details.
portfolio	Optional, A list of objects of the class solarOptionPortfolio.
nmonths	Numeric vector, months in which the payoff should be computed. Can vary from 1 (January) to 12 (December).
lambda	Numeric, Sugeno parameter.
implvol	Numeric, implied volatility.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.

Note

Version 1.0.0.

Examples

```
model = solarModel$new(spec)
model$fit()
moments <- model$moments$unconditional[1:365,]
lambda = 0
control_options = control_solarOption()
solarOption_choquet(model, moments[1:30,], lambda = 0.01)
solarOption_model(model, moments[1,])
solarOption_choquet(model, moments, lambda = 0.1, put = F)
```

---

solarOption_greeks *Compute the Greeks of a Solar Option*

Description

Compute the Greeks of a Solar Option

Usage

```
solarOption_greeks(model, moments, put = TRUE, elasticities = FALSE)
```

Note

Version 1.0.0.

Examples

```
# Model fit
model <- solarModel$new(spec)
model$fit()
solarOption_greeks(model, model$Moments("2023-01-01", "2024-05-01"))
```

---

solarOption_historical *Payoff of solar options on historical data*

Description

Payoff of solar options on historical data

Usage

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

Arguments

model	An object with the class solarModel. See the function solarModel for details.
nmonths	Numeric vector, months in which the payoff should be computed. Can vary from 1 (January) to 12 (December).
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.

Value

An object of the class solarOptionPayoff.

Note

Version 1.0.0.

Examples

```
# Solar model
model <- solarModel$new(spec)
model$fit()
# Historical payoff (put)
solarOption_historical(model, put=TRUE)
# Historical payoff (call)
solarOption_historical(model, put=FALSE)
```

solarOption_index_greeks

Compute the Greeks of a Solar Option index

Description

Compute the Greeks of a Solar Option index

Usage

```
solarOption_index_greeks(model, moments, elasticities = FALSE)
```

Note

Version 1.0.0.

Examples

```
# Model fit
model <- solarModel$new(spec)
model$fit()

t_now <- as.Date("2024-01-01")
t_hor <- as.Date("2024-12-31")
dates <- seq.Date(t_now+1, t_hor, 1)
mom <- purrr::map_df(dates, ~model$Moments(t_now, .x))
solarOption_index_greeks(model, mom, elasticities = TRUE)
mom_test <- mom
mom_test$beta <- mom_test$beta*1.01
solarOption_index_greeks(model, mom_test, elasticities = TRUE)
```

solarOption_lambda

Calibrate the implied Choquet parameter

Description

Calibrate the implied Choquet parameter

Usage

```
solarOption_lambda(
  model,
  moments,
  P_target,
  r_imp = 0,
  put = TRUE,
  quiet = FALSE,
  control_options = control_solarOption()
)
```

Arguments

model	An object with the class solarModel. See the function solarModel for details.
moments	Tibble containing the forecasted moments for different days ahead. See the function solarMoments for more details.
P_target	Numeric, target price to match.
r_imp	Numeric, implied return from measure under P.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.

Details

The optimization function will find the best parameter λ such that

$$\underset{-0.5<\lambda<1}{\text{argmin}} \left(P^{\mathbb{P}(\lambda)}(1 + r^{\text{imp}}) - P_{\text{target}} \right)^2$$

Note

Version 1.0.0.

---

solarOption_model *Compute the premium given the moments*

Description

Compute the premium given the moments

Usage

```
solarOption_model(
  model,
  moments,
  portfolio,
  nmonths = 1:12,
  lambda = 0,
  implvol = 1,
  put = TRUE,
  control_options = control_solarOption()
)
```

Arguments

model	An object with the class solarModel. See the function solarModel for details.
moments	Tibble containing the forecasted moments for different days ahead. See the function solarMoments for more details.
portfolio	Optional, A list of objects of the class solarOptionPortfolio.

nmonths	Numeric vector, months in which the payoff should be computed. Can vary from 1 (January) to 12 (December).
lambda	Numeric, Sugeno parameter.
implvol	Numeric, implied volatility.
put	Logical, when TRUE, the default, will be computed the price for a put contract, otherwise for a call contract.
control_options	Named list with control parameters. See control_solarOption for more details.

Note

Version 1.0.0.

Examples

```
# Model
model = solarModel$new(spec)
model$fit()
# Pricing without portfolio
moments <- model$moments$unconditional
# Premium
premium_Vt <- solarOption_model(model, moments, put = 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")
# solar options portfolio
portfolio <- SoRadPorfolio(model, t_now, t_init, t_hor)
# Moments
moments <- purrr::map_df(portfolio, ~model$Moments(t_now, .x$t_hor))
# Premium
premium_Vt <- solarOption_model(model, moments, portfolio, put = TRUE)
premium_Vt$payoff_year$premium
```

---

solarOption_moments *Compute the first fourth moments and variance of a SoRad*

Description

Compute the first fourth moments and variance of a SoRad

Usage

```
solarOption_moments(
  moments,
  transform,
  lambda = 0,
  put = TRUE,
  link = "invgumbel",
  control_options = control_solarOption()
)
```

Arguments

`transform`	slot transform of a solarModel
`lambda`	Sugeno parameter
`put`	Logical, when `TRUE`, the default, will be computed the price for a `put` contract, otherwise for a `call` contract.
`control_options`	Named list with control parameters. See `control_solarOption` for more details.

Note

Version 1.0.0.

Examples

```
# Solar model
model <- solarModel$new(spec)
model$fit()
moments <- model$moments$unconditional
solarOption_moments(moments[1:365,], model$transform)
```

`solarOption_pricing` *Compute the price of a* `solarOption`

Description

Compute the price of a `solarOption`

Usage

```
solarOption_pricing(
  moments,
  sorad,
  lambda = 0,
  put = TRUE,
  link = "invgumbel",
  control_options = control_solarOption()
)
```

Arguments

`moments`	Tibble containing the forecasted moments for different days ahead. See the function `solarMoments` for more details.
`sorad`	An object of the class `solarOption`.
`lambda`	Numeric, Sugeno parameter.
`put`	Logical, when `TRUE`, the default, will be computed the price for a `put` contract, otherwise for a `call` contract.
`control_options`	Named list with control parameters. See `control_solarOption` for more details.

Note

Version 1.0.0.

Examples

```
# Model
model = solarModel$new(spec)
model$fit()
moments <- filter(model$moments$conditional, Year == 2022)
# Pricing without contracts
solarOption_pricing(moments[1,])
# Pricing with contracts specification
sorad <- solarOption$new()
sorad$set_contract("2021-12-31", "2022-01-01", "2022-04-20", moments$GHI_bar[1])
solarOption_pricing(moments[1,], sorad)
solarOption_pricing(moments[1,], sorad, lambda = 0.02)
solarOption_pricing(moments[1,], sorad, lambda = -0.02)
```

`solarOption_scenario` *Compute average premium on simulated Data*

Description

Compute average premium on simulated Data

Usage

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

Arguments

`model`	An object with the class `solarModel`. See the function `solarModel` for details.
`scenario`	object with the class `solarModelScenario`. See the function `solarModel_scenarios` for details.
`nmonths`	Numeric vector, months in which the payoff should be computed. Can vary from 1 (January) to 12 (December).
`put`	Logical, when `TRUE`, the default, will be computed the price for a `put` contract, otherwise for a `call` contract.
`nsim`	number of simulation to use for computation.
`control_options`	control function, see `control_solarOption` for details.

Value

An object of the class `solarOptionPayoff`.

Note

Version 1.0.0.

Examples

```
# Solar model
model <- solarModel$new(spec)
model$fit()
# Simulate scenarios
scenario <- solarScenario(model, from = "2011-01-01", to = "2012-01-01", by = "1 month", nsim = 10, seed = 3)

solarOption_scenario(model, scenario)
solarOption_historical(model)
solarScenario_plot(scenario)
```

`solarPayoff` *Payoff function of a Solar Option*

Description

Payoff function of a Solar Option

Usage

```
solarPayoff(R, K = 0, put = TRUE)
```

Arguments

`R`	Numeric, vector of values of solar radiation at maturity.
`K`	Numeric, scalar or vector of strikes.
`put`	Logical, when `TRUE`, the default, the function will return the output of a put payoff otherwise a call payoff. See the details.

Details

The put option payoff reads:

$$(K - R)^+ = (K - R)1_{K>R}$$

Symmetrically a call option payoff reads:

$$(R - K)^+ = (R - K)1_{R\geq K}$$

Note

Version 1.0.0.

Examples

```
solarPayoff(10, 9, put = TRUE)
```

solarScenario *Simulate multiple scenarios*

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, shift parameter for the mixture.
`nsim`	integer, number of simulations.
`seed`	scalar integer, starting random seed.
`quiet`	logical

Note

Version 1.0.0.

Examples

```
model <- solarModel$new(spec)
model$fit()
scen <- solarScenario(model, "2005-01-10", "2025-01-10", theta = 0.1, nsim = 5, by = "1 year")
# Plot
solarScenario_plot(scen, nsim = 3)
# Solar Option
solarOption_scenario(model, scen)
solarOption_historical(model)
```

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.

Note

Version 1.0.0.

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_plot *Plot scenarios from a solarScenario object*

Description

Plot scenarios from a solarScenario object

Usage

```
solarScenario_plot(x, target = "GHI", nsim = 10, empiric = TRUE, ci = 0.05)
```

Examples

```
model = solarModel$new(spec)
model$fit()
scenario <- solarScenario(model, nsim = 70)
solarScenario_plot(scenario)
```

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.

Note

Version 1.0.0.

Examples

```
model <- solarModel$new(spec)
model$fit()
simSpec <- solarScenario_spec(model, from = "2010-01-01", to = "2010-01-01")
simSpec <- solarScenario_residuals(simSpec, nsim = 10)
```

solarScenario_spec *Specification of a solar scenario*

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 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.
theta	numeric, shift parameter for the mixture.
exclude_known	when true the two starting points (equals for all the simulations) will be excluded from the output.
quiet	logical

Note

Version 1.0.0.

Examples

```
model <- solarModel$new(spec)
model$fit()
simSpec <- solarScenario_spec(model)
```

---

solarScenario_VaR *Compute the Value at Risk from simulated values*

Description

Compute the Value at Risk from simulated values

Usage

```
solarScenario_VaR(scenarios, alpha = 0.05)
```

Arguments

scenarios	An object of the class solarScenario
alpha	Confidence level for the VaR

Note

Version 1.0.0.

Examples

```
model <- Bologna
scen <- solarScenario(model, "2016-01-01", "2017-01-01", nsim = 10, by = "1 month")
solarScenario_VaR(scen, 0.05)
```

solarTransform *solarTransform Solar functions*

Description

Solar Model transformation functions

Public fields

`epsilon` Numeric, ϵ transformation parameter.

Active bindings

`alpha` Numeric, α transformation parameter.

`beta` Numeric, β transformation parameter.

`F_H` Function, distribution function of the transform.

`Q_H` Function, quantile function of the transform.

Methods

Public methods:

- `solarTransform$new()`
- `solarTransform$GHI()`
- `solarTransform$iGHI()`
- `solarTransform$GHI_y()`
- `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, link = "invgumbel")
```

Arguments:

`alpha` Numeric, α transformation parameter.

`beta` Numeric, β transformation parameter.

`link` Character, link function.

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:

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

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

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.

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

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

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:

$$R_t(y) = C(t)(1 - \alpha - \beta \exp(-\exp(y)))$$

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

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:

$$Y^{-1}(y) = \alpha + \beta \exp(-\exp(y))$$

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, min_pos = 1, max_pos = 1)
```

Arguments:

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

`epsilon` Numeric

`min_pos` Integer, position of the minimum. For example when `2` the minimum is the second lowest value.

`max_pos` Integer, position of the maximum. For example when `3` the maximum is the third greatest value.

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_min 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.

Note

Version 1.0.0.

Examples

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

`SoRadPorfolio` *Create a Solar Option Index portfolio*

Description

Create a Solar Option Index portfolio

Usage

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

Note

Version 1.0.0.

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`.

Public fields

`weights` Weighting function for the distance.

Active bindings

`grid` A tibble with the spatial grid.

Methods

Public methods:

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

Method `new()`**:** Initialize a spatial grid

Usage:

```
spatialGrid$new(weights = IDW(2))
```

Arguments:

`weights` Weighting function for the distance.

Method `set_grid()`**:** Set a spatial grid

Usage:

```
spatialGrid$set_grid(grid)
```

Arguments:

`grid` Tibble with column `id`, `lat` and `lon`.

Method `make_grid()`**:** Create a spatial grid

Usage:

```
spatialGrid$make_grid(lat, lon, by, digits = 5)
```

Arguments:

`lat` Numeric vector, from which is extracted the minimum and maximum for latitude.

`lon` Numeric vector, from which is extracted the minimum and maximum for longitude.

`by` Numeric vector, the first element is used to establish the distance between two latitudes in the grid. The second element (if present) is used to establish the distance between two longitudes in the grid.

`digits` Integer scalar, number of digits for latitudes and longitudes.

Method `is_inside_bounds()`**:** Check if a location is inside the bounds of the grid.

Usage:

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

Arguments:

`lat` Numeric vector, reference latitudes.

`lon` Numeric vector, reference longitudes.

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

Method `is_known_point()`**:** Check if a location is a known point inside the grid.

Usage:

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

Arguments:

`lat` Numeric vector, reference latitudes.

`lon` Numeric vector, reference longitudes.

Returns: `TRUE` when the location is known and `FALSE` otherwise.

Method `known_point()`**:** Return the id and coordinates of a location inside the grid.

Usage:

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

Arguments:

`lat` Numeric vector, reference latitudes.

`lon` Numeric vector, reference longitudes.

Method `neighborhoods()`**:** Find the n-closest neighborhoods of a location.

Usage:

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

Arguments:

`lat` Numeric scalar, reference latitude.

`lon` Numeric scalar, reference longitude.

`n` number of neighborhoods

Method `print()`**:** Method print for a `spatialGrid` object.

Usage:

```
spatialGrid$print()
```

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

```
# Initialize a spatial grid
sp <- spatialGrid$new()

# Create an equally spaced grid
sp$make_grid(c(43.1, 44), c(9.2, 12.3), c(0.1, 0.1))
sp
# Check known point
sp$is_known_point(49.95, 12.15)
sp$is_known_point(43.9, 12)

# Check if a point is inside the bounds
sp$is_inside_bounds(44.8, 10.9)

# Extract a point
sp$neighborhoods(43.9, 12.1)

# Extract its neighborhoods
sp$neighborhoods(43.95, 12.15)
```

`spatialKringing` — `spatialKringing` *object*

Description

`spatialKringing` object

`spatialKringing` object

Public fields

`quiet` Logical

Active bindings

`models` list of `kernelRegression` objects

`data` dataset with the parameters used for fitting

Methods

Public methods:

- `spatialKringing$new()`
- `spatialKringing$fit()`
- `spatialKringing$predict()`
- `spatialKringing$clone()`

Method `new()`**:** Initialize a `spatialKringing` object

Usage:

```
spatialKringing$new(data, params_names, grid, vg_models, quiet = FALSE)
```

Arguments:

`data` dataset with spatial parameters and lon, lat.

`params_names` Names of the parameters to fit.

`grid` description

`vg_models` an optional list of kernelRegression models already fitted.

`quiet` description

`sample` List of parameter used as sample.

Method `fit()`**:** Fit a `kernelRegression` object for a parameter or a group of parameters.

Usage:

```
spatialKringing$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:

```
spatialKringing$predict(lat, lon, n = 4)
```

Arguments:

`lat` Numeric vector, latitudes in degrees.
`lon` Numeric vector, longitudes in degrees.
`n` Integer, number of neighborhoods to consider for interpolation.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`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

`grid` object with the spatial grid

`models` list of `solarModel` objects

`parameters` `spatialParameters` object

Methods

Public methods:

- `spatialModel$new()`
- `spatialModel$get()`
- `spatialModel$interpolate()`
- `spatialModel$solarModel()`
- `spatialModel$clone()`

Method `new()`: Initialize the spatial model

Usage:

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

Arguments:

`models` A list of `solarModel` objects
`paramsModels` A `spatialParameters` object.
`quiet` logical

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

Usage:

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

Arguments:

`id` character, id of the location.

`lat` numeric, latitude of a location.

`lon` numeric, longitude of a location.

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

Usage:

```
spatialModel$interpolate(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 `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
spatialModel$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/m2 x nanometer, otherwise if `micrometer` the final energy will be in W/m2 x micrometer.
`lambda`	numeric, wave length in micrometers.

`sp_cor_aniso` *Exponential and Spherical anisotropic spatial correlation*

Description

Exponential and Spherical anisotropic spatial correlation

Usage

```
sp_cor_aniso_exp(h1, h2, phi)

sp_cor_aniso_sph(h1, h2, phi)
```

Arguments

`h1`	Matrix of vector of distances.
`h2`	Matrix of vector of distances.
`phi`	Numeric vector, parameters.

Details

Version 1.0.0. Implement the functions:

$$\rho_{\exp}(h_1, h_2, \phi^{\exp}) = \exp\left(\frac{1}{\phi_1^{\exp}}\sqrt{\phi_2^{\exp} h_1^2 + h_2^2}\right)$$

$$\rho_{\text{sph}}(h_1, h_2, \phi^{\text{sph}}) = 1 - \frac{3}{2}\frac{\sqrt{\phi_2^{\text{sph}} h_1^2 + h_2}}{\phi_1^{\text{sph}}} + \frac{1}{2}\left(\frac{\sqrt{\phi_2^{\text{sph}} h_1^2 + h_2}}{\phi_1^{\text{sph}}}\right)^3$$

sp_cor_isotr — *Exponential, Gaussian and Spherical isotropic spatial correlation*

Description

Exponential, Gaussian and Spherical isotropic spatial correlation

Usage

```
sp_cor_isotr_exp(h, phi)

sp_cor_isotr_gau(h, phi)

sp_cor_isotr_sph(h, phi)
```

Arguments

`h`	Matrix of vector of distances.
`phi`	Numeric scalar, parameter.

Details

Version 1.0.0. Implement the functions:

$$\rho_{\mathrm{exp}}(h, \phi^{\mathrm{exp}}) = \exp\left(\frac{h}{\phi^{\mathrm{exp}}}\right)$$

$$\rho_{\mathrm{gau}}(h, \phi^{\mathrm{exp}}) = \exp\left\{\left(\frac{h}{\phi^{\mathrm{gau}}}\right)^2\right\}$$

$$\rho_{\mathrm{sph}}(h, \phi^{\mathrm{sph}}) = 1 - \frac{3}{2}\frac{h}{\phi^{\mathrm{sph}}} + \frac{1}{2}\left(\frac{h}{\phi^{\mathrm{sph}}}\right)^3$$

sugeno_bounds — *Sugeno upper and lower parameters.*

Description

Sugeno upper and lower parameters.

Usage

```
sugeno_bounds(lambda)
```

Arguments

`lambda`	Numeric, distortion parameter.

`VaR_test` *Evaluate VaR test*

Description

Evaluate VaR test

Usage

```
VaR_test(et, VaR, ci)
```

Index