Package ‘solarr’

October 12, 2024

Type Package

Title Stochastic models for solar radiation

Version 0.2.0

Author Beniamino Sartini

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

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

Depends R (>= 3.5.0),
ggplot2,
np,
dplyr (>= 1.1.3),
mclust,

Imports assertive (>= 0.3-6),
stringr (>= 1.5.0),
rugarch (>= 1.4.1),
purrr (>= 1.0.2),
tidyr (>= 1.2.0),
lubridate (>= 1.8.0),
nortest,
broom,
formula.tools

Suggests knitr,
rmarkdown,
testthat (>= 2.1.0)

License GPL-3

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

R topics documented:

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

Description

Control parameters for a 'seasonalClearsky' object

Usage

```
control_seasonalClearsky(
  method = "II",
  include.intercept = TRUE,
  order = 1,
  period = 365,
  delta0 = 1.4,
  lower = 0,
  upper = 3,
  by = 0.001,
  ntol = 30,
  quiet = FALSE
)
```

Arguments

`method`	character, method for clear sky estimate, can be 'I' or 'II'.
`include.intercept`	logical. When 'TRUE', the default, the intercept will be included in the model.
`order`	numeric, of sine and cosine elements.
`period`	numeric, periodicity. The default is '365'.
`delta0`	Value for delta init in the clear sky model.
`lower`	numeric, lower bound for delta grid.
`upper`	numeric, upper bound for delta grid.

by	numeric, step for delta grid.
ntol	integer, tolerance for ‘clearsky > GHI‘ condition. Maximum number of violations admitted.
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.

Examples

```
control = control_seasonalClearsky()
```

---

control_solarModel *Control parameters for a ‘solarModel‘ object*

Description

Control function for a solarModel

Usage

```
control_solarModel(
  clearsky = control_seasonalClearsky(),
  stochastic_clearsky = FALSE,
 seasonal.mean = list(seasonalOrder = 1, include.H0 = FALSE, include.intercept = TRUE,
    monthly.mean = TRUE),
  mean.model = list(arOrder = 2, include.intercept = FALSE),
 seasonal.variance = list(seasonalOrder = 1, correction = TRUE, monthly.mean = TRUE),
 variance.model = rugarch::ugarchspec(variance.model = list(garchOrder = c(1, 1)),
    mean.model = list(armaOrder = c(0, 0), include.mean = FALSE)),
  mixture.model = list(abstol = 0.001, maxit = 150, EM = TRUE),
  threshold = 0.01,
  outliers_quantile = 0,
  quiet = FALSE
)
```

Arguments

clearsky
: list with control parameters for the clear sky seasonal model. See control_seasonalClearsky for details.

seasonal.mean
: a list of parameters. Available choices are:

‘seasonalOrder‘ An integer specifying the order of the seasonal component. The default is ‘1‘.

‘include.intercept‘ Logical, when ‘TRUE‘ the intercept will be included in the seasonal model, otherwise will be omitted. The dafault if ‘TRUE‘.

‘include.H0‘ Logical, when ‘TRUE‘ the extraterrestrial radiation will be included in the seasonal model, otherwise will be omitted. The dafault if ‘FALSE‘.

- **‘monthly.mean‘** Logical, when ‘TRUE‘ a vector of 12 monthly means is computed on the deseasonalized series and it is subtracted to ensure that it is perfectly centered around zero.

`mean.model` a list of parameters for the AR model.

- **‘arOrder‘** An integer specifying the order of the AR component. The default is ‘2‘.
- **‘include.intercept‘** When ‘TRUE‘ an intercept will be included in the AR equation, otherwise is omitted. The dafault if ‘FALSE‘.

`seasonal.variance`
a list of parameters for the seasonal variance. Available choices are:

- **‘seasonalOrder‘** Integer, it specify the order of the seasonal component in the model. The default is ‘1‘.
- **‘correction‘** Logical, when ‘TRUE‘ the parameters of seasonal variance are corrected to ensure that the standardize the residuals have exactly a unitary variance.
- **‘monthly.mean‘** Logical, when ‘TRUE‘ a vector of 12 monthly std. deviations is computed on the GARCH residuals. Then, they are divided by this quantity to ensure that the monthly variance is one.

`variance.model` an ‘ugarchspec‘ object for GARCH variance. Default is ‘GARCH(1,1)‘ specification.

`mixture.model` a list of parameters for the monthly Gaussian mixture model. Available choices are:

- **‘abstol‘** Numeric, absolute level for convergence. The default is ‘1e-3‘.
- **‘maxit‘** Integer, maximum number of iterations. The default is ‘200‘.
- **‘EM‘** Logical, when ‘TRUE‘ the estimated parameters from ‘mclust‘ function will be used to initialize the EM-routine. The default is ‘TRUE‘.

`threshold` numeric, threshold used to estimate the transformation parameters alpha and beta. See `solarTransform` for details.

`outliers_quantile`
quantile for outliers detection. If different from 0, the observations that are below the quantile at confidence levels ‘outliers_quantile‘ and the observation above the quantile at confidence level 1-‘outliers_quantile‘ will have a weight equal to zero and will be excluded from estimations.

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

Examples

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

`control_solarOption` *Control parameters for a solar option*

Description

Control parameters for a solar option

Usage

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

Arguments

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

Examples

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

`desscher` *Esscher transform of a density*

Description

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

Usage

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

Arguments

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

Details

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

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

Examples

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

---

desscherMixture *Esscher transform of a Gaussian Mixture*

Description

Esscher transform of a Gaussian Mixture

Usage

```
desscherMixture(means = c(0, 0), sd = c(1, 1), p = c(0.5, 0.5), theta = 0)

pesscherMixture(means = c(0, 0), sd = c(1, 1), p = c(0.5, 0.5), theta = 0)
```

Arguments

`means`	vector of means parameters.
`sd`	vector of std. deviation parameters.
`p`	vector of probability parameters.
`theta`	Esscher parameter, the default is zero.

Examples

```
library(ggplot2)
grid <- seq(-5, 5, 0.01)
# Density
pdf_1 <- desscherMixture(means = c(-3, 3), theta = 0)(grid)
pdf_2 <- desscherMixture(means = c(-3, 3), theta = -0.5)(grid)
pdf_3 <- desscherMixture(means = 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 <- psscherMixture(means = c(-3, 3), theta = 0)(grid)
cdf_2 <- pesscherMixture(means = c(-3, 3), theta = -0.2)(grid)
cdf_3 <- pesscherMixture(means = 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(x, 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`	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.

References

Gumbel distribution [W].

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(x, 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	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.

Examples

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

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

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

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

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

`discountFactor` *Discount factor function*

Description

Discount factor function

Usage

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

Arguments

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

`dkumaraswamy` *Kumaraswamy random variable*

Description

Kumaraswamy density, distribution, quantile and random generator.

Usage

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

pkumaraswamy(x, 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`	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.

References

Kumaraswamy Distribution [W].

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)

# 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, means = rep(0, 2), sd = rep(1, 2), p = rep(1/2, 2), log = FALSE)

pmixnorm(
  x,
  means = rep(0, 2),
  sd = rep(1, 2),
  p = rep(1/2, 2),
  lower.tail = TRUE,
  log.p = FALSE
)

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

rmixnorm(n, means = rep(0, 3), sd = rep(1, 3), p = rep(1/3, 3))
```

Arguments

x	vector of quantiles or probabilities.
means	vector of means parameters.
sd	vector of std. deviation parameters.
p	vector of probability parameters.
log	logical; if 'TRUE', probabilities are returned as 'log(p)'.
lower.tail	logical; if TRUE (default), probabilities are 'P[X < x]' otherwise, 'P[X > x]'.
log.p	logical; if 'TRUE', probabilities p are given as 'log(p)'.
n	number of observations. If 'length(n) > 1', the length is taken to be the number required.

References

Mixture Models [W].

Examples

```
# Parameters
means = c(-3,0,3)
sd = rep(1, 3)
p = c(0.2, 0.3, 0.5)
# Density function
dmixnorm(3, means, sd, p)
# Distribution function
dmixnorm(c(1.2, -3), means, sd, p)
# Quantile function
qmixnorm(0.2, means, sd, p)
# Random generator
rmixnorm(1000, means, sd, p)
```

---

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
)
```

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.

---

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(x, 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	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.

References

Skewed Normal Distribution [W].

Examples

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

# Density function
# right tailed
plot(x, dsnorm(x, shape = 4.9), type = "l")
# left tailed
plot(x, dsnorm(x, shape = -4.9), 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.seet(1)
plot(rsnorm(100, shape = 4), type = "l")
```

---

dsolarGHI — *Solar radiation random variable*

Description

Solar radiation density, distribution, quantile and random generator.

Usage

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

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

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

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

Arguments

x	vector of quantiles.
Ct	clear sky radiation
alpha	parameter ‘alpha > 0‘.
beta	parameter ‘beta > 0‘ and ‘alpha + beta < 1‘.
pdf_Y	density of Y.
log	logical; if ‘TRUE‘, probabilities are returned as ‘log(p)‘.
cdf_Y	distribution of Y.

`log.p`	logical; if ‘TRUE‘, probabilities p are given as ‘log(p)‘.
`lower.tail`	logical; if ‘TRUE‘, the default, the computed probabilities are ‘P[X < x]‘. Otherwise, ‘P[X > x]‘.
`p`	vector of probabilities.

Details

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

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

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

Examples

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

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

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

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

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

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

`dsolarK` *Clearness index random variable*

Description

Clearness index density, distribution, quantile and random generator.

Usage

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

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

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

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

Arguments

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

Details

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

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

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

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

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")
```

`gaussianMixture` *Gaussian mixture*

Description

Gaussian mixture

Gaussian mixture

Details

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

Public fields

`maxit` Integer, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

`components` Integer, number of components.

`means` Numeric vector of means parameters.

`sd` Numeric vector of std. deviation parameters.

`p` Numeric vector of probability parameters.

Active bindings

`parameters` 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.

Methods

Public methods:

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

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

Usage:

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

Arguments:

`components` Integer, number of components.

`maxit` Numeric, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

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

Usage:

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

Arguments:

`x` vector

`params` Named list of mixture parameters.

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

Usage:

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

Arguments:

`x` vector

`params` a list of mixture parameters

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

Usage:

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

Arguments:

`x` vector

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

Usage:

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

Arguments:

`x` vector

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

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

Usage:

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

Arguments:

`x` vector

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

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

Usage:

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

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.

`means` Numeric vector of means parameters.

`sd` Numeric vector of std. deviation parameters.

`p` Numeric vector of probability parameters.

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.

Examples

```
means = c(-3,0,3)
sd = rep(1, 3)
p = c(0.2, 0.3, 0.5)
# Density function
pdf <- dmixnorm(means, sd, p)
# Distribution function
cdf <- pmixnorm(means, sd, p)
# Random numbers
```

```
x <- rmixnorm(5000, means, sd, p)
gm <- gaussianMixture$new(components=3)
gm$fit(x$X)
gm$parameters
gm$EM(x$X)
gm$parameters
gm$fitted
```

---

havDistance *Haversine distance*

Description

Compute the Haversine distance between two points.

Usage

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

Arguments

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

Value

Numeric vector the distance in kilometers.

Examples

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

IDW *Inverse Distance Weighting Functions*

Description

Return a distance weighting function

Usage

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

Arguments

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

Details

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

Examples

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

---

is_leap_year *Is leap year?*

Description

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

Usage

```
is_leap_year(x)
```

Arguments

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

Value

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

Examples

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

kernelRegression *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$new()`
- `kernelRegression$predict()`
- `kernelRegression$clone()`

Method `new()`**:** Initialize a 'kernelRegression' object

Usage:

```
kernelRegression$new(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

Usage:

```
kernelRegression$predict(...)
```

Arguments:

`...` arguments to fit.

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.

ks_test *Kolmogorov Smirnov test for a distribution*

Description

Test against a specific distribution with ‘ks_test‘ and perform a two sample invariance test for a time series with ‘ks_ts_test‘

Usage

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

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

Arguments

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

Value

when ‘plot = 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)
```

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("2040-01-31", "2023-04-01", "2015-09-02"))
number_of_day(c("2029-02-28", "2029-03-01", "2020-12-31"))
number_of_day(c("2020-02-29", "2020-03-01", "2020-12-31"))
```

optionPayoff *Option payoff function*

Description

Compute the payoffs of an option at maturity.

Usage

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

Arguments

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

Examples

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

PDF — *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))
```

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` *Clear sky seasonal model*

Description

Clear sky seasonal model

Clear sky seasonal model

Super class

`solarr::seasonalModel` -> `seasonalClearsky`

Public fields

`control` See the function `control_seasonalClearsky` for details.

`lat` latitude of the place considered.

Methods

Public methods:

- `seasonalClearsky$new()`
- `seasonalClearsky$fit()`
- `seasonalClearsky$updateH0()`
- `seasonalClearsky$clone()`

Method `new()`**:** Initialize a seasonalClearsky model

Usage:

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

Arguments:

`control` See the function `control_seasonalClearsky` for details.

Method `fit()`**:** Fit a seasonal model for clear sky radiation

Usage:

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

Arguments:

`x` time series of solar radiation

`date` time series of dates

`lat` reference latitude

`clearsky` optional time series of observed clerasky radiation.

Method `updateH0()`**:** Update the time series of Extraterrestrial radiation

Usage:

```
seasonalClearsky$updateH0(lat)
```

Arguments:

`lat` reference latitude

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

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

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

`seasonalModel` *Seasonal Model Object*

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 fitting.

Public fields

`seasonal_data` Slot that contains eventual externals seasonal regressors used for fitting.

`extra_params` Slot used for containing eventual extra parameters.

Active bindings

`coefficients` Get a vector with the fitted coefficients.

`model` Get the fitted ‘lm‘ object.

`period` Get the seasonality in days.

`order` Get the number of combinations of sines and cosines used.

Methods

Public methods:

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

Method `new()`**:** Initialize an object of the class ‘seasonalModel‘.

Usage:

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

Arguments:

`order` numeric, number of sine and cosine used in fitting.

`period` numeric, seasonal periodicity. The default is $\frac{2\pi}{365}$.

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

Usage:

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

Arguments:

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

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

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

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

Usage:

```
seasonalModel$predict(n)
```

Arguments:

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

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

Usage:

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

Arguments:

`coefficients` vector of parameters.

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.

`seasonalRadiation` *Seasonal model for solar radiation radiation*

Description

Fit a seasonal model for solar radiation

Usage

```
seasonalRadiation(spec)
```

Arguments

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

Examples

```
library(ggplot2)
# Seasonal model for GHI
spec <- solarModel_spec("Oslo", target = "GHI")
model <- seasonalRadiation(spec)
spec$data$GHI_bar <- model$predict(spec$data$n)
ggplot(spec$data)+
 geom_line(aes(n, GHI))+
 geom_line(aes(n, GHI_bar), color = "blue")

# Seasonal model for clear sky
spec <- solarModel_spec("Oslo", target = "clearsky")
model <- seasonalRadiation(spec)
spec$data$Ct_bar <- model$predict(spec$data$n)
ggplot(spec$data)+
 geom_line(aes(n, clearsky))+
 geom_line(aes(n, Ct_bar), color = "blue")
```

`seasonalSolarFunctions`

Solar seasonal functions

Description

Solar seasonal functions

Solar seasonal functions

Public fields

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

Active bindings

`G0` solar constant, i,e, '1367'.

Methods

Public methods:

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

Method `new()`**:** Initialize a 'seasonalSolarFunctions' object

Usage:

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

Arguments:

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

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

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

Usage:

```
seasonalSolarFunctions$update_method(x)
```

Arguments:

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

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

Method `B()`: Seasonal adjustment parameter.

Usage:

```
seasonalSolarFunctions$B(n)
```

Arguments:

`n` number of the day of the year

Details: The function computes

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

Method `degree()`: Convert angles in 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 `solar_time()`: Compute the solar time from a clock time.

Usage:

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

Arguments:

`x` datetime, clock hour.

`lon` longitude of interest in degrees.

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

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

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

Returns: A datetime object

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

Usage:

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

Arguments:

`x` datetime, clock hour.

`lon` longitude of interest in degrees.

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

Returns: An angle in degrees

Method `incidence_angle()`**:** Compute the incidence angle

Usage:

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

Arguments:

`x` datetime, clock hour.

`lat` latitude of interest in degrees.

`lon` longitude of interest in degrees.

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

`beta` altitude

`gamma` orientation

Returns: An angle in degrees

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

Usage:

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

Arguments:

`x` datetime, clock hour.

`lat` latitude of interest in degrees.

`lon` longitude of interest in degrees.

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

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

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

Returns: The solar azimut angle in degrees

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

Usage:

```
seasonalSolarFunctions$G0n(n)
```

Arguments:

`n` number of the day of the year

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

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

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

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

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

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

Usage:

```
seasonalSolarFunctions$declination(n)
```

Arguments:

`n` number of the day of the year

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

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

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

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

Returns: The solar declination in degrees.

Method `H0()`**:** Compute the solar extraterrestrial radiation

Usage:

```
seasonalSolarFunctions$H0(n, lat)
```

Arguments:

`n` number of the day of the year

`lat` latitude of interest in degrees.

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

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

Usage:

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

Arguments:

`n` number of the day of the year

`lat` Numeric, latitude of interest in degrees.

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

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

Returns: The sunset hour angle in degrees.

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

Usage:

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

Arguments:

`n` number of the day of the year.

`lat` Numeric, latitude of interest in degrees.

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

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

Method `solar_altitude()`**:** Compute solar altitude in degrees

Usage:

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

Arguments:

`n` number of the day of the year

`lat` Numeric, latitude of interest in degrees.

Details: The function computes

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

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

Usage:

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

Arguments:

`x` datetime, clock hour.

`lat` Numeric, latitude of interest in degrees.

`lon` Numeric, longitude of interest in degrees.

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

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

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

Usage:

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

Arguments:

`cosZ` solar incidence angle

`G0` solar constant

`altitude` altitude in km

`clime` clime correction

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

Usage:

`seasonalSolarFunctions$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Duffie, Solar Engineering of Thermal Processes Fourth Edition.

Examples

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

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

# Time adjustment in minutes
sf$E(dates)

# Declination
sf$declination(dates)

# Solar constant
sf$G0

# Solar constant adjusted
sf$G0n(dates)

# Extraterrestrial radiation
sf$H0(dates, 43)

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

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

`solarEsscher` *Calibrate Esscher Bounds and parameters*

Description

Calibrate Esscher Bounds and parameters

Calibrate Esscher Bounds and parameters

Public fields

`control` list containing the control parameters

`grid` list containing the grids

Active bindings

`bounds` calibrated bounds with respect to bootstrapped payoff.

`models` models to predict the optimal theta given the expected return.

Methods

Public methods:

- `solarEsscher$new()`
- `solarEsscher$calibrator()`
- `solarEsscher$calibrate_bounds()`
- `solarEsscher$create_grid()`
- `solarEsscher$fit_theta()`
- `solarEsscher$predict()`
- `solarEsscher$clone()`

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

Usage:

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

Arguments:

`n_key_points` integer, number of key points for interpolation.

`init_lambda` numeric, initial value for the Esscher parameter.

`lower_lambda` numeric, lower value for the Esscher parameter.

`upper_lambda` numeric, upper value for the Esscher parameter.

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

`target.Yt` logical, when 'TRUE' will be distorted with esscher parameter the pdf of Yt otherwise the pdf of the GHI.

`quiet` logical

`control_options` control function. See `control_solarOption` for details.

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

Usage:

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

Arguments:

`model` solar model

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

`nmonths` month or months

`target.Yt` logical, when ‘TRUE‘ will be distorted with esscher parameter the pdf of Yt otherwise the pdf of the GHI.

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

Usage:

```
solarEsscher$calibrate_bounds(model, payoffs, target.Yt)
```

Arguments:

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

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

`target.Yt` logical, when ‘TRUE‘ will be distorted with esscher parameter the pdf of Yt otherwise the pdf of the GHI.

Method `create_grid()`**:** Create a grid of optimal theta and expected returns with respect of the benchmark price.

Usage:

```
solarEsscher$create_grid(
  model,
  benchmark_price,
  lower_price,
  upper_price,
  target.Yt
)
```

Arguments:

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

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

`lower_price` lower price in the grid.

`upper_price` upper price in the grid.

`target.Yt` logical, when ‘TRUE‘ will be distorted with esscher parameter the pdf of Yt otherwise the pdf of the GHI.

Method `fit_theta()`**:** Fit the models to predict the optimal Esscher parameters given the grid.

Usage:

```
solarEsscher$fit_theta()
```

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

Usage:

```
solarEsscher$predict(r, target.Yt = FALSE)
```

Arguments:

`r` expected return

`target.Yt` logical, when ‘TRUE‘ will be distorted with esscher parameter the pdf of Yt otherwise the pdf of the GHI.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`solarEsscher_probability` *Change probability according to Esscher parameters*

Description

Change probability according to Esscher parameters

Usage

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

`solarMixture` *Monthly Gaussian mixture with two components*

Description

Monthly Gaussian mixture with two components

Monthly Gaussian mixture with two components

Public fields

`maxit` Integer, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

`components` Integer, number of components.

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.

`model` Named List with 12 `gaussianMixture` objects.

`loglik` Numeric, total log-likelihood.

`fitted` A 'tibble' with the classified series

`moments` A 'tibble' with the theoric moments. It contains: #'

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

`parameters` A 'tibble' with the fitted parameters.

Methods

Public methods:

- solarMixture$new()
- solarMixture$fit()
- solarMixture$clone()

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

Usage:

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

Arguments:

`components` Integer, number of components.

`maxit` Integer, maximum number of iterations.

`abstol` Numeric, absolute level for convergence.

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

Usage:

```
solarMixture$fit(x, date, weights, EM = FALSE)
```

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.

`EM` logical when TRUE will be applied EM algo

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.

`solarModel` *Solar Model in R6 Class*

Description

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

Details

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

Public fields

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

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

`dates` A named list, with the range of dates used in the model.

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

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

Active bindings

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

`seasonal_data` A data frame containing seasonal and monthly parameters.

`monthly_data` A data frame that contains monthly parameters.

`loglik` The log-likelihood computed on train data.

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

`location` A data frame with coordinates of the location considered.

`transform` A `solarTransform` object with the transformation functions applied to the data.

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

`seasonal_model_Yt` The fitted seasonal model for the target variable.

`AR_model_Yt` The fitted Autoregressive (AR) 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.

`combinations` mixture combination moments.

`parameters` Get the model parameters as a named list.

Methods

Public methods:

- `solarModel$new()`
- `solarModel$fit()`
- `solarModel$fit_clearsky_model()`
- `solarModel$compute_risk_drivers()`
- `solarModel$fit_solar_transform()`
- `solarModel$detect_outliers_Yt()`
- `solarModel$fit_seasonal_mean()`
- `solarModel$corrective_monthly_mean()`
- `solarModel$fit_AR_model()`
- `solarModel$fit_seasonal_variance()`
- `solarModel$fit_GARCH_model()`

- solarModel$corrective_monthly_variance()
- solarModel$fit_mixture_model()
- solarModel$update()
- solarModel$filter()
- solarModel$conditional_moments()
- solarModel$unconditional_moments()
- solarModel$logLik()
- solarModel$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_clearsky_model()`**:** Initialize and fit a `seasonalClearsky` model given the specification contained in ‘$control‘.

Usage:

```
solarModel$fit_clearsky_model()
```

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

Usage:

```
solarModel$compute_risk_drivers()
```

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

Usage:

```
solarModel$fit_solar_transform()
```

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

Usage:

```
solarModel$detect_outliers_Yt()
```

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

Usage:

```
solarModel$fit_seasonal_mean()
```

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

Usage:

```
solarModel$corrective_monthly_mean()
```

Method `fit_AR_model()`: Fit an AR model (‘Yt_tilde‘) and compute AR residuals (‘eps‘).

Usage:

```
solarModel$fit_AR_model()
```

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_GARCH_model()`: Fit a ‘GARCH‘ model on the deseasonalized residuals (‘eps_tilde‘). Compute the standardized (‘u‘) and monthly deseasonalized residuals (‘u_tilde‘).

Usage:

```
solarModel$fit_GARCH_model()
```

Method `corrective_monthly_variance()`: Correct the standardized GARCH residuals (‘u‘) by dividing them by the monthly std. deviation (‘sigma_m‘).

Usage:

```
solarModel$corrective_monthly_variance()
```

Returns: Update the slots ‘$data$u_tilde‘ and ‘$monthly_data$sigma_m‘.

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

Usage:

```
solarModel$fit_mixture_model()
```

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

Usage:

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

Arguments:

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

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

Usage:

```
solarModel$filter()
```

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

Method `conditional_moments()`: Compute the conditional moments and update the slot ‘$moments$conditional‘.

Usage:

```
solarModel$conditional_moments()
```

Method `unconditional_moments()`: Compute the unconditional seasonal moments and update the slot ‘$moments$unconditional‘.

Usage:

```
solarModel$unconditional_moments()
```

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

Usage:

```
solarModel$logLik()
```

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.

Examples

```
# Control list
control <- control_solarModel(outliers_quantile = 0)
# Model specification
spec <- solarModel_spec("Bologna", from="2005-01-01", control_model = control)
Bologna <- solarModel$new(spec)
# Model fit
Bologna$fit()
# save(Bologna, file = "data/Bologna.RData")

# Extract and update the parameters
params <- sm$parameters
sm$update(params)
sm$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()
```

---

solarModel_conditional_moments *Compute conditional moments from a 'solarModel' object*

Description

Compute conditional moments from a 'solarModel' object

Usage

```
solarModel_conditional_moments(model, date)
```

Examples

```
model <- Bologna
solarModel_conditional_moments(model)
solarModel_conditional_moments(model, date = "2022-01-01")
```

`solarModel_empiric_GM` *Empiric Gaussian Mixture parameters*

Description

Empiric Gaussian Mixture parameters

Usage

```
solarModel_empiric_GM(model, match_moments = FALSE)
```

`solarModel_forecast` *Iterate the forecast on multiple dates*

Description

Iterate the forecast on multiple dates

Usage

```
solarModel_forecast(model, date, ci = 0.1, unconditional = FALSE)
```

Examples

```
model <- Bologna
dates <- seq.Date(as.Date("2020-01-01"), as.Date("2020-01-31"), 1)
solarModel_forecast(model, date = dates)
```

`solarModel_forecaster` *Produce a forecast from a solarModel object*

Description

Produce a forecast from a solarModel object

Usage

```
solarModel_forecaster(
  model,
  date = "2020-01-01",
  ci = 0.1,
  unconditional = FALSE
)
```

Examples

```
model <- Bologna
solarModel_forecaster(model, date = "2010-04-01")
object <- solarModel_forecaster(model, date = "2020-04-01", unconditional = TRUE)
object
```

`solarModel_forecaster_plot` *Plot a forecast from a solarModel object*

Description

Plot a forecast from a solarModel object

Usage

```
solarModel_forecaster_plot(
  model,
  date = "2021-05-29",
  ci = 0.1,
  type = "mix",
  unconditional = FALSE
)
```

Examples

```
model <- Bologna
day_date <- "2013-01-13"
solarModel_forecaster_plot(model, date = day_date)
solarModel_forecaster_plot(model, date = day_date, unconditional = TRUE)
solarModel_forecaster_plot(model, date = day_date, type = "dw")
solarModel_forecaster_plot(model, date = day_date, type = "dw", unconditional = TRUE)
solarModel_forecaster_plot(model, date = day_date, type = "up")
solarModel_forecaster_plot(model, date = day_date, type = "up", unconditional = TRUE)
```

`solarModel_mvmixture` *Monthly multivariate Gaussian mixture with two components*

Description

Monthly multivariate Gaussian mixture with two components

Usage

```
solarModel_mvmixture(model_Ct, model_GHI)
```

Arguments

`model_Ct`	arg
`model_GHI`	arg

solarModel_spec *Specification function for a 'solarModel'*

Description

Specification function for a 'solarModel'

Usage

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

Arguments

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

Examples

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

`solarModel_test_residuals`
Stationarity and distribution test (Gaussian mixture) for a ‘solarModel‘

Description

Stationarity and distribution test (Gaussian mixture) for a ‘solarModel‘

Usage

```
solarModel_test_residuals(
  model,
  nrep = 50,
  ci = 0.05,
  min_quantile = 0.015,
  max_quantile = 0.985,
  seed = 1
)
```

Examples

```
model <- Bologna
solarModel_test_residuals(model)
```

`solarModel_unconditional_moments`
Compute conditional moments from a ‘solarModel‘ object

Description

Compute conditional moments from a ‘solarModel‘ object

Usage

```
solarModel_unconditional_moments(model, nmonths, ndays, date)
```

Examples

```
model <- Bologna
solarModel_unconditional_moments(model)
solarModel_unconditional_moments(model, nmonths = 1)
solarModel_unconditional_moments(model, nmonths = 1, ndays = 1)
solarModel_unconditional_moments(model, date = "2022-01-01")
```

`solarOptionPayoffs` *solarOptionPayoff*

Description

solarOptionPayoff

Usage

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

Arguments

`model`	solarModel
`control_options`	control list, see `control_solarOption` for more details.

Value

An object of the class ‘solarOptionPayoffs‘.

`solarOption_calibrator` *Calibrator for solar Options*

Description

Calibrator for solar Options

Usage

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

Arguments

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

Examples

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

---

solarOption_contracts *Optimal number of contracts*

Description

Compute the optimal number of contracts given a particular setup.

Usage

```
solarOption_contracts(
  payoff,
  type = "model",
  premium = "Q",
  put = TRUE,
  nyear = 2021,
  tick = 0.06,
  efficiency = 0.2,
  n_panels = 2000,
  pun = 0.06
)
```

Arguments

type	character, method used for computing the premium. Can be ‘model‘ (Model with integral) or ‘sim‘ (Monte Carlo).
premium	character, premium used. Can be ‘P‘, ‘Qdw‘, ‘Qup‘, or ‘Q‘.
nyear	integer, actual year. The optimization will be performed excluding the year ‘nyear‘ and the following.
tick	numeric, conversion tick for the monetary payoff of a contract.
efficiency	numeric, mean efficiency of the solar panels.
n_panels	numeric, number of meters squared of solar panels.
pun	numeric, reference electricity price at which the energy is sold for computing the cash-flows.
model	object with the class ‘solarModel‘. See the function solarModel for details.

---

solarOption_historical
Payoff on Historical Data

Description

Payoff on Historical Data

Usage

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

Arguments

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

Value

An object of the class 'solarOptionPayoff'.

Examples

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

---

solarOption_historical_bootstrap
Bootstrap a fair premium from historical data

Description

Bootstrap a fair premium from historical data

Usage

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

Arguments

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

Value

An object of the class ‘solarOptionBoot‘.

Examples

```
model <- Bologna
solarOption_historical_bootstrap(model)
```

---

solarOption_model *Pricing function under the solar model*

Description

Pricing function under the solar model

Usage

```
solarOption_model(
  model,
  nmonths = 1:12,
  theta = 0,
  combinations = NA,
  implvol = 1,
  put = TRUE,
  target.Yt = TRUE,
  control_options = control_solarOption()
)
```

Arguments

model	object with the class ‘solarModel‘. See the function solarModel for details.
nmonths	numeric vector of months in which the payoff is computed. Range from 1 to 12.
theta	Esscher parameter
combinations	list of 12 elements with gaussian mixture components.
implvol	implied unconditional GARCH variance, the default is ‘1‘.
put	logical, when ‘TRUE‘, the default, the computations will consider a ‘put‘ contract. Otherwise a ‘call‘.
target.Yt	logical, when ‘TRUE‘, the default, the computations will consider the pdf of ‘Yt‘ otherwise the pdf of solar radiation.
control_options	control list, see control_solarOption for more details.

Value

An object of the class ‘solarOptionPayoff‘.

Examples

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

---

solarOption_model_test

Test errors solar Option model

Description

Test errors solar Option model

Usage

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

Examples

```
model <- Bologna
solarOption_model_test(model)
solarOption_model_test(model, put = FALSE)
solarOption_model_test(model, nmonths = 6, put = FALSE)
solarOption_model_test(model, nmonths = 6, put = TRUE)
```

solarOption_scenario *Payoff on simulated Data*

Description

Payoff on simulated Data

Usage

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

Arguments

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

Value

An object of the class 'solarOptionPayoff'.

Examples

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

solarOption_structure *Structure payoffs*

Description

Structure payoffs

Usage

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

Arguments

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

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

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

Value

The object ‘payoffs‘ with class ‘solarOptionPayoffs‘.

`solarScenario` *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, Esscher parameter.
`nsim`	integer, number of simulations.
`seed`	scalar integer, starting random seed.
`quiet`	logical

Examples

```
model <- Bologna
scen <- solarScenario(model, "2024-01-01", "2025-12-31")
```

`solarScenario_filter` *Simulate trajectories from a a ‘solarScenario_spec‘*

Description

Simulate trajectories from a a ‘solarScenario_spec‘

Usage

```
solarScenario_filter(simSpec)
```

Arguments

`simSpec`	object with the class ‘solarScenario_spec‘. See the function `solarScenario_spec` for details.

Examples

```
model <- Bologna
simSpec <- solarScenario_spec(model, from = "2024-06-01", to = "2024-12-31")
simSpec <- solarScenario_residuals(simSpec, nsim = 1)
simSpec <- solarScenario_filter(simSpec)
# Empiric data
df_emp <- simSpec$emp
# First simulation
df_sim <- simSpec$simulations[[1]]
ggplot()+
geom_line(data = df_emp, aes(date, GHI))+
geom_line(data = df_sim, aes(date, GHI), color = "red")
```

solarScenario_residuals *Simulate residuals for a ‘solarScenario_spec‘*

Description

Simulate residuals for a ‘solarScenario_spec‘

Usage

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

Arguments

simSpec	object with the class ‘solarScenario_spec‘. See the function solarScenario_spec for details.
nsim	integer, number of simulations.
seed	scalar integer, starting random seed.

Examples

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

solarScenario_spec *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, Esscher parameter.
exclude_known	when true the two starting points (equals for all the simulations) will be excluded from the output.
quiet	logical

Examples

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

---

solarTransform *solarTransform Solar functions*

Description

Solar Model transformation functions

Active bindings

`alpha` Numeric, the first transformation parameter.

`beta` Numeric, the second transformation parameter.

Methods

Public methods:

- solarTransform$new()
- solarTransform$GHI()
- solarTransform$GHI_y()
- solarTransform$iGHI()
- solarTransform$Y()
- solarTransform$iY()
- solarTransform$fit()
- solarTransform$bounds()
- solarTransform$update()
- solarTransform$clone()

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

Usage:

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

Arguments:

`alpha` Numeric, transformation parameter.

`beta` Numeric, transformation parameter.

Method `GHI()`**:** Solar radiation function

Usage:

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

Arguments:

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

`Ct` Numeric, clear sky radiation.

Details: The function computes:

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

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

Method `GHI_y()`**:** Solar radiation function in terms of y

Usage:

```
solarTransform$GHI_y(y, Ct)
```

Arguments:

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

`Ct` Numeric, clear sky radiation.

Details: The function computes:

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

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

Method `iGHI()`**:** Compute the risk driver process

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‘funcion

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

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

Method `Y()`**:** Transformation function from X to 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()`**:** Inverse transformation from Y to X.

Usage:

```
solarTransform$iY(y)
```

Arguments:

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

Details: The function computes:

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

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

Method `fit()`**:** Fit the best parameters from a time series

Usage:

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

Arguments:

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

`threshold` for minimum

Details: Return a list that contains:

alpha Numeric, first transformation parameter.

beta Numeric, second transformation parameter.

epsilon Numeric, threshold used for fitting.

Xt_min Numeric, minimum value of the supplied time series.

Xt_min Numeric, maximum value of the supplied 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

Usage:

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

Arguments:

`alpha` Numeric, transformation parameter.

`beta` Numeric, transformation parameter.

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

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

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

---

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

Create a grid from a range of latitudes and longitudes.

Usage

```
spatialGrid(lat = c(43.7, 45.1), lon = c(9.2, 12.7), by = c(0.1, 0.1))
```

Arguments

`by`	step for longitudes and latitudes. If two values are specified the first will be used for latitudes and the second for longitudes
`range_lat`	vector with latitudes. Only the minimum and maximum values are considered.
`range_lon`	vector with longitudes. Only the minimum and maximum values are considered.

Value

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

Examples

```
spatialGrid(lat = c(43.7, 43.8), lon = c(12.5, 12.7), by = 0.1)
spatialGrid(lat = c(43.7, 43.8), lon = c(12.6, 12.7), by = c(0.05, 0.01))
```

`spatialModel` *Spatial model object*

Description

Spatial model object

Spatial model object

Active bindings

`models` list of ‘solarModel‘ objects

`locations` dataset with all the locations.

`parameters` ‘spatialParameters‘ object

Methods

Public methods:

- `spatialModel$new()`
- `spatialModel$neighborhoods()`
- `spatialModel$is_known_location()`
- `spatialModel$gridModel()`
- `spatialModel$is_inside_limits()`
- `spatialModel$interpolator()`
- `spatialModel$solarModel()`
- `spatialModel$combinations()`
- `spatialModel$clone()`

Method `new()`**:** Initialize the spatial model

Usage:

```
spatialModel$new(locations, models, paramsModels, beta = 2, d0, quiet = FALSE)
```

Arguments:

`locations` A tibble with columns ‘place‘, ‘lat‘, ‘lon‘, ‘from‘, ‘to‘, ‘nobs‘.

`models` A list of ‘solarModel‘ objects

`paramsModels` A ‘spatialParameters‘ object.

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

`d0` numeric, used only in exponential function.

`quiet` logical

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

Usage:

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

Arguments:

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

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

`n` number of neighborhoods

Method `is_known_location()`**:** Check if a point is already in the spatial grid

Usage:

```
spatialModel$is_known_location(lat, lon)
```

Arguments:

`lat` numeric, latitude of a location.

`lon` numeric, longitude of a location.

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

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

Usage:

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

Arguments:

`place` character, id of the location.

`lat` numeric, latitude of a location.

`lon` numeric, longitude of a location.

Method `is_inside_limits()`**:** Check if a point is inside the limits of the spatial grid.

Usage:

```
spatialModel$is_inside_limits(lat, lon)
```

Arguments:

`lat` numeric, latitude of a location.

`lon` numeric, longitude of a location.

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

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

Usage:

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

Arguments:

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

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

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

`n` number of neighborhoods to use for interpolation.

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

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

Usage:

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

Arguments:

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

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

`n` number of neighborhoods

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

Usage:

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

Arguments:

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

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

`nmonths` numeric, months to consider

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

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`spatialParameters` *‘spatialParameters‘ object*

Description

‘spatialParameters‘ object

‘spatialParameters‘ object

Active bindings

`models` list of ‘kernelRegression‘ objects

`data` dataset with the parameters used for fitting

Methods

Public methods:

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

Method `new()`: Initialize a ‘spatialParameters‘ object

Usage:

```
spatialParameters$new(solarModels, models, quiet = FALSE)
```

Arguments:

`solarModels` list of ‘solarModel‘ objects.

`models` an optional list of models.

`quiet` logical

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

Usage:

```
spatialParameters$fit(params)
```

Arguments:

`params` list of parameters names to fit. When missing all the parameters will be fitted.

Method `predict()`**:** Predict all the parameters for a specified location.

Usage:

```
spatialParameters$predict(lat, lon, as_tibble = FALSE)
```

Arguments:

`lat` numeric, latitude in degrees.

`lon` numeric, longitude in degrees.

`as_tibble` logical, when ‘TRUE‘ will be returned a ‘tibble‘.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

---

spatialScenario_filter *Simulate trajectories from a ‘spatialScenario_spec‘*

Description

Simulate trajectories from a ‘spatialScenario_spec‘

Usage

```
spatialScenario_filter(simSpec)
```

Arguments

`simSpec`	object with the class ‘spatialScenario_spec‘. See the function `spatialScenario_spec` for details.

`spatialScenario_residuals` *Simulate residuals from a a 'spatialScenario_spec'*

Description

Simulate residuals from a a 'spatialScenario_spec'

Usage

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

Arguments

simSpec	object with the class 'spatialScenario_spec'. See the function spatialScenario_spec for details.
nsim	integer, number of simulations.
seed	scalar integer, starting random seed.

`spatialScenario_spec` *Specification of a solar scenario*

Description

Specification of a solar scenario

Usage

```
spatialScenario_spec(
  sm,
  sc,
  places,
  from = "2010-01-01",
  to = "2010-01-31",
  exclude_known = FALSE,
  quiet = FALSE
)
```

Arguments

sm	'spatialModel' object
sc	'spatialCorrelation' object
places	target places
from	character, start Date for simulations in the format 'YYYY-MM-DD'.
to	character, end Date for simulations in the format 'YYYY-MM-DD'.
exclude_known	when true the two starting points (equals for all the simulations) will be excluded from the output.
quiet	logical

spectralDistribution — *Compute the spectral distribution for a black body*

Description

Compute the spectral distribution for a black body

Usage

```
spectralDistribution(x, measure = "nanometer")
```

Arguments

`measure`	character, measure of the irradiated energy. If 'nanometer' the final energy will be in W/m2 x nanometer, otherwise if 'micrometer' the final energy will be in W/m2 x micrometer.
`lambda`	numeric, wave length in micrometers.

test_normality — *Perform normality tests*

Description

Perform normality tests

Usage

```
test_normality(x = NULL, pvalue = 0.05)
```

Arguments

`x`	numeric, a vector of observation.
`pvalue`	numeric, the desiderd level of 'p.value' at which the null hypothesis will be rejected.

Value

a tibble with the results of the normality tests.

Examples

```
set.seed(1)
x <- rnorm(1000, 0, 1) + rchisq(1000, 1)
test_normality(x)
x <- rnorm(1000, 0, 1)
test_normality(x)
```

`tnorm` *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.

Examples

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

# Density function
p <- dtnorm(x, mean = 0, sd = 1, a = -1)
plot(x, p, type = "l")

# Distribution function
p <- ptnorm(x, mean = 0, sd = 1, b = 1)
plot(x, p, type = "l")

# Quantile function
dtnorm(0.1)
ptnorm(qtnorm(0.1))

# Random Numbers
rtnorm(1000)
plot(rtnorm(100, mean = 0, sd = 1, a = 0, b = 1), type = "l")
```

Index