# Package 'qrmtools'

June 16, 2017

**Version** 0.0-7

**Encoding** UTF-8

**Title** Tools for Quantitative Risk Management

**Description** Functions and data sets for reproducing selected results from the book ``Quantitative Risk Management: Concepts, Techniques and Tools". Furthermore, new developments and auxiliary functions for Quantitative Risk Management practice.

**Maintainer** Marius Hofert <marius.hofert@uwaterloo.ca>

**Depends** R (>= 3.2.0)

**Imports** graphics, lattice, quantmod, Quandl, zoo, xts, methods, grDevices, stats, rugarch, utils

**Suggests** combinat, copula, knitr, mvtnorm, QRM, sfsmisc, RColorBrewer, sn

**Enhances**

**License** GPL-2 | GPL-3

**NeedsCompilation** yes

**VignetteBuilder** knitr

**Repository** CRAN

**Date/Publication** 2017-06-16 21:28:20 UTC

**Author** Marius Hofert [aut, cre],
Kurt Hornik [aut],
Alexander J. McNeil [aut]

## R topics documented:

---

ARMA_GARCH *Fitting ARMA-GARCH Processes*

---

## Description

Fail-safe componentwise fitting of univariate ARMA-GARCH processes.

## Usage

```
fit_ARMA_GARCH(x, ugarchspec.list = ugarchspec(), solver = "hybrid",
               verbose = TRUE, ...)
```

## Arguments

| | |
|---|---|
| x | matrix-like data structure, possibly an xts object. |
| ugarchspec.list | object of class uGARCHspec (as returned by ugarchspec()) or a list of such. In case of a list, its length has to be equal to the number of columns of x. ugarchspec.list provides the ARMA-GARCH specifications for each of the time series (columns of x). |
| solver | string indicating the solver used; see ?ugarchfit. |
| verbose | logical indicating whether verbose output is given. |
| ... | additional arguments passed to the underlying ugarchfit(). |

## Value

If x consists of one column only (e.g. a vector), ARMA_GARCH() returns the fitted object; otherwise it returns a list of such.

## Author(s)

Marius Hofert

## Examples

```
library(rugarch)
library(copula)

## Read the data, build -log-returns
data(SMI.12) # Swiss Market Index data
stocks <- c("CSGN", "BAER", "UBSN", "SREN", "ZURN") # components we work with
x <- SMI.12[, stocks]
X <- -returns(x)
n <- nrow(X)
d <- ncol(X)

## Fit ARMA-GARCH models to the -log-returns
## Note: - Our choice here is purely for demonstration purposes.
##         The models are not necessarily adequate
##       - The sample size n is *too* small here for properly capturing GARCH effects.
##         Again, this is only for demonstration purposes here.
uspec <- c(rep(list(ugarchspec(distribution.model = "std")), d-2), # ARMA(1,1)-GARCH(1,1)
           list(ugarchspec(variance.model = list(model = "sGARCH", garchOrder = c(2,2)),
                           distribution.model = "std")),
           list(ugarchspec(variance.model = list(model = "sGARCH", garchOrder = c(2,1)),
                           mean.model = list(armaOrder = c(1,2), include.mean = TRUE),
                           distribution.model = "std")))
system.time(fitAG <- fit_ARMA_GARCH(X, ugarchspec.list = uspec))
str(fitAG, max.level = 1) # list with components fit, warning, error
stopifnot(sapply(fitAG$error, is.null)) # NULL = no error
stopifnot(sapply(fitAG$warning, is.null)) # NULL = no warning

## Not run:
## Pick out the standardized residuals, plot them and fit a t copula to them
## Note: ugarchsim() needs the residuals to be standardized; working with
##       standardize = FALSE still requires to simulate them from the
##       respective standardized marginal distribution functions.
Z <- sapply(fitAG$fit, residuals, standardize = TRUE)
U <- pobs(Z)
pairs(U, gap = 0)
system.time(fitC <- fitCopula(tCopula(dim = d, dispstr = "un"), data = U,
                              method = "mpl"))

## Simulate (standardized) Z
set.seed(271)
U. <- rCopula(n, fitC@copula) # simulate from the fitted copula
nu <- sapply(1:d, function(j) fitAG$fit[[j]]@fit$coef["shape"]) # extract (fitted) d.o.f. nu
Z. <- sapply(1:d, function(j) sqrt((nu[j]-2)/nu[j]) * qt(U.[,j], df = nu[j])) # Z

## Simulate from fitted model
X. <- sapply(1:d, function(j)
    fitted(ugarchsim(fitAG$fit[[j]], n.sim = n, m.sim = 1, startMethod = "sample",
                     rseed = 271, custom.dist = list(name = "sample",
                                                     distfit = Z.[,j, drop = FALSE]))))

## Plots original vs simulated -log-returns
```

```
opar <- par(no.readonly = TRUE)
layout(matrix(1:(2*d), ncol = d)) # layout
ran <- range(X, X.)
for(j in 1:d) {
    plot(X[,j],  type = "l", ylim = ran, ylab = paste(stocks[j], "-log-returns"))
    plot(X.[,j], type = "l", ylim = ran, ylab = "Simulated -log-returns")
}
    par(opar)

## End(Not run)
```

---

`Black_Scholes` *Black–Scholes formula and the Greeks*

---

## Description

Compute the Black–Scholes formula and the Greeks.

## Usage

```
Black_Scholes(t, S, r, sigma, K, T, type = c("call", "put"))
Black_Scholes_Greeks(t, S, r, sigma, K, T)
```

## Arguments

| | |
|---|---|
| `t` | initial or current time $t$ (in years). |
| `S` | stock price at time $t$. |
| `r` | risk-free annual interest rate. |
| `sigma` | annual volatility (standard deviation). |
| `K` | strike. |
| `T` | maturity (in years). |
| `type` | `character` string indicating whether the price of a call (the default) or of put option is to be computed. |

## Value

`Black_Scholes()` returns the value of a European-style call or put option (depending on the chosen `type`) on a non-dividend paying stock.

`Black_Scholes_Greeks()` returns the first-order derivatives delta, theta, rho, vega and the second-order derivatives gamma, vanna and vomma (in this order).

## Author(s)

Marius Hofert

## References

McNeil, A. J., Frey, R., and Embrechts, P. (2015). *Quantitative Risk Management: Concepts, Techniques, Tools*. Princeton University Press.

---

`catch` *Catching Results, Warnings and Errors Simultaneously*

---

## Description

Catches results, warnings and errors.

## Usage

```
catch(expr)
```

## Arguments

| | |
|---|---|
| `expr` | expression to be evaluated, typically a function call. |

## Details

This function is particularly useful for large(r) simulation studies to not fail until finished.

## Value

`list` with components:

| | |
|---|---|
| `value` | value of `expr` or `NULL` in case of an error. |
| `warning` | warning message (see `simpleWarning` or `warning()`) or `NULL` in case of no warning. |
| `error` | error message (see `simpleError` or `stop()`) or `NULL` in case of no error. |

## Author(s)

Marius Hofert (based on `doCallWE()` and `tryCatch.W.E()` in the R package **simsalapar**).

## Examples

```
catch(log(2))
catch(log(-1))
catch(log("a"))
```

get_data *Tools for Getting and Working with Data*

## Description

Download (and possibly) merge data from freely available databases.

## Usage

```
get_data(x, from = NULL, to = NULL,
         src = c("yahoo", "quandl", "oanda", "FRED", "google"),
         FUN = NULL, verbose = TRUE, warn = TRUE, ...)
```

## Arguments

| | |
|---|---|
| x | vector of ticker symbols (e.g. "^GSPC" if src = "yahoo" or "EUR/USD" if src = "oanda"). |
| from | start date as a Date object or character string (in international date format "yyyy-mm-dd"); if NULL, the earliest date with available data is picked. |
| to | end date as a Date object or character string (in international date format "yyyy-mm-dd"); if NULL, the last date with available data is picked. |
| src | character string specifying the data source (e.g. "yahoo" for stocks or "oanda" for FX data); see getSymbols() and Quandl(). |
| FUN | function to be applied to the data before being returned. This can be<br>**the identity** if the data could not be retrieved (and is thus replaced by NA);<br>**the given** FUN if FUN has been provided;<br>**a useful default** if FUN = NULL; the default uses the adjusted close price Ad() if src = "yahoo", the close price Cl() if src = "google" and the identity otherwise. |
| verbose | logical indicating whether progress monitoring should be done. |
| warn | logical indicating whether a warning is given showing the error message when fetching x fails. |
| ... | additional arguments passed to the underlying function getSymbols() from **quantmod** or Quandl() from **Quandl** (if src = "quandl"). |

## Details

FUN is typically one of **quantmod**'s Op, Hi, Lo, Cl, Vo, Ad or one of the combined functions OpCl, ClCl, HiCl, LoCl, LoHi, OpHi, OpLo, OpOp.

## Value

xts object containing the data with column name(s) adjusted to be the ticker symbol (in case lengths match; otherwise the column names are not adjusted); NA if data is not available.

### Author(s)

Marius Hofert

### Examples

```
## Not run:
    ## Note: This needs a working internet connection
    ## Get stock and volatility data (for all available trading days)
    dat <- get_data(c("^GSPC", "^VIX")) # note: this needs a working internet connection
    ## Plot them (Alternative: plot.xts() from xtsExtra)
    library(zoo)
    plot.zoo(dat, screens = 1, main = "", xlab = "Trading day", ylab = "Value")

## End(Not run)
```

---

## GEV *Generalized Extreme Value Distribution*

---

### Description

Density, distribution function, quantile function and random variate generation for the generalized extreme value distribution (GEV).

### Usage

```
dGEV(x, xi, mu = 0, sigma = 1, log = FALSE)
pGEV(q, xi, mu = 0, sigma = 1, lower.tail = TRUE, log.p = FALSE)
qGEV(p, xi, mu = 0, sigma = 1, lower.tail = TRUE, log.p = FALSE)
rGEV(n, xi, mu = 0, sigma = 1)
```

### Arguments

| | |
|---|---|
| `x, q` | vector of quantiles. |
| `p` | vector of probabilities. |
| `n` | number of observations. |
| `xi` | GEV shape parameter, a real number. |
| `mu` | GEV location parameter, a real number. |
| `sigma` | GEV scale parameter, a positive number. |
| `lower.tail` | logical; if TRUE (default) probabilities are $P(X \leq x)$ otherwise, $P(X > x)$. |
| `log, log.p` | logical; if TRUE, probabilities p are given as log(p). |

### Details

The distribution function of the generalized extreme value distribution is given by

$$F(x) = \begin{cases} \exp(-(1-\xi(x-\mu)/\sigma)^{-1/\xi}), & \text{if } \xi \neq 0,\ 1+\xi(x-\mu)/\sigma > 0, \\ \exp(-e^{-(x-\mu)/\sigma}), & \text{if } \xi = 0, \end{cases}$$

where $\sigma > 0$.

## Value

dGEV() computes the density, pGEV() the distribution function, qGEV() the quantile function and rGEV() random variates of the generalized extreme value distribution.

## Author(s)

Marius Hofert

## References

McNeil, A. J., Frey, R., and Embrechts, P. (2015). *Quantitative Risk Management: Concepts, Techniques, Tools*. Princeton University Press.

## Examples

```
## Basic sanity checks
plot(pGEV(rGEV(1000, xi = 0.5), xi = 0.5)) # should be U[0,1]
curve(dGEV(x, xi = 0.5), from = -3, to = 5)
```

---

# GPD *(Generalized) Pareto Distribution*

---

## Description

Density, distribution function, quantile function and random variate generation for the (generalized) Pareto distribution (GPD).

## Usage

```
dGPD(x, xi, beta, log = FALSE)
pGPD(q, xi, beta, lower.tail = TRUE, log.p = FALSE)
qGPD(p, xi, beta, lower.tail = TRUE, log.p = FALSE)
rGPD(n, xi, beta)

dPar(x, theta, kappa = 1, log = FALSE)
pPar(q, theta, kappa = 1, lower.tail = TRUE, log.p = FALSE)
qPar(p, theta, kappa = 1, lower.tail = TRUE, log.p = FALSE)
rPar(n, theta, kappa = 1)
```

## Arguments

| | |
|---|---|
| x, q | vector of quantiles. |
| p | vector of probabilities. |
| n | number of observations. |
| xi | GPD shape parameter, a real number. |
| beta | GPD scale parameter, a positive number. |

| | |
|---|---|
| theta | Pareto shape parameter, a positive number. |
| kappa | Pareto scale parameter, a positive number. |
| lower.tail | logical; if TRUE (default) probabilities are $P(X \leq x)$ otherwise, $P(X > x)$. |
| log, log.p | logical; if TRUE, probabilities p are given as log(p). |

## Details

The distribution function of the generalized Pareto distribution is given by

$$F(x) = \begin{cases} 1 - (1 + \xi x/\beta)^{-1/\xi}, & \text{if } \xi \neq 0, \\ 1 - \exp(-x/\beta), & \text{if } \xi = 0, \end{cases}$$

where $\beta > 0$ and $x \geq 0$ if $\xi \geq 0$ and $x \in [0, -\beta/\xi]$ if $\xi < 0$.

The distribution function of the Pareto distribution is given by

$$F(x) = 1 - (\kappa/(\kappa + x))^{\theta}, \ x \geq 0,$$

where $\theta > 0$, $\kappa > 0$.

In contrast to `dGPD()`, `pGPD()`, `qGPD()` and `rGPD()`, the functions `dPar()`, `pPar()`, `qPar()` and `rPar()` are vectorized in their main argument and the parameters.

## Value

`dGPD()` computes the density, `pGPD()` the distribution function, `qGPD()` the quantile function and `rGPD()` random variates of the generalized Pareto distribution.

Similary for `dPar()`, `pPar()`, `qPar()` and `rPar()` for the Pareto distribution.

## Author(s)

Marius Hofert

## References

McNeil, A. J., Frey, R., and Embrechts, P. (2015). *Quantitative Risk Management: Concepts, Techniques, Tools*. Princeton University Press.

## Examples

```
## Basic sanity checks
plot(pGPD(rGPD(1000, xi = 0.5, beta = 3), xi = 0.5, beta = 3)) # should be U[0,1]
curve(dGPD(x, xi = 0.5, beta = 3), from = -1, to = 5)
```

`matrix_density_plota` *Density Plot of the Values from a Lower Triangular Matrix*

## Description

Density plot of all values in the lower triangular part of a matrix.

## Usage

```
matrix_density_plot(x, xlab = "Entries in the lower triangular matrix",
                    main = "", text = NULL, side = 4, line = 1, adj = 0, ...)
```

## Arguments

| | |
|---|---|
| `x` | `matrix`-like object. |
| `xlab` | x-axis label. |
| `main` | title. |
| `text` | see `mtext()`. The `text = ""`, it is omitted. |
| `side` | see `mtext()`. |
| `line` | see `mtext()`. |
| `adj` | see `mtext()`. |
| `...` | additional arguments passed to the underlying `plot()`. |

## Details

`matrix_density_plot()` is typically used for symmetric matrices (like correlation matrices, matrices of pairwise Kendall's tau or tail dependence parameters) to check the distribution of their off-diagonal entries.

## Value

`invisible()`.

## Author(s)

Marius Hofert

## Examples

```
## Generate a random correlation matrix
d <- 50
L <- diag(1:d)
set.seed(271)
L[lower.tri(L)] <- runif(choose(d,2))
Sigma <- L
P <- cor(Sigma)
## Density of its lower triangular entries
matrix_density_plot(P)
```

matrix_plot *Graphical Tool for Visualizing Matrices*

## Description

Plot of a matrix.

## Usage

```
matrix_plot(x, ylim = rev(c(0.5, nrow(x) + 0.5)),
            xlab = "Column", ylab = "Row",
            scales = list(alternating = c(1,1), tck = c(1,0),
                          x = list(at = pretty(1:ncol(x)), rot = 90),
                          y = list(at = pretty(1:nrow(x)))),
          at = NULL, colorkey = NULL, col = c("royalblue3", "white", "maroon3"),
            col.regions = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | matrix-like object. |
| ylim | y-axis limits in reverse order (for the rows to appear 'top down'). |
| xlab | x-axis label. |
| ylab | y-axis label. |
| scales | see levelplot(); if NULL, labels and ticks are omitted. |
| at | see levelplot(). If NULL, a useful default is computed based on the given values in x. |
| colorkey | see levelplot(). If NULL, a useful default is computed based on at. |
| col | vector of length two (if all values of x are non-positive or all are non-negative; note that also a vector of length three is allowed in this case) or three (if x contains negative and positive values) providing the color key's default colors. |
| col.regions | see levelplot(). If NULL, a useful default is computed based on at. |
| ... | additional arguments passed to the underlying function levelplot(). |

## Details

Plot of a matrix.

## Value

The plot, a Trellis object.

## Author(s)

Marius Hofert

## Examples

```
## Generate a random correlation matrix
d <- 50
L <- diag(1:d)
set.seed(271)
L[lower.tri(L)] <- runif(choose(d,2))
Sigma <- L
P <- cor(Sigma)

## Default
matrix_plot(P)

## Default if nonnegative
matrix_plot(abs(P))

## Without diagonal
P. <- abs(P)
diag(P.) <- NA
matrix_plot(P.)

## Default if nonpositive
matrix_plot(-abs(P))

## Extending the color key to [-1,1] with darker color for |rho| >> 0
## Note: When specifying 'at', one most likely also wants to provide 'col.regions'
matrix_plot(P, at = seq(-1, 1, length.out = 200),
          col.regions = grey(c(seq(0, 1, length.out = 100), seq(1, 0, length.out = 100))))

## An example with overlaid lines
library(lattice)
my_panel <- function(...) {
    panel.levelplot(...)
    panel.abline(h = c(10, 20), v = c(10, 20), lty = 2)
}
matrix_plot(P, panel = my_panel)
```

---

# NA_plot *Graphical Tool for Visualizing NAs in a Data Set*

---

## Description

Plot NAs in a data set.

## Usage

```
NA_plot(x, col = c("black", "white"), xlab = "Time", ylab = "Component",
        text = "Black: NA; White: Available data",
        side = 4, line = 1, adj = 0, ...)
```

## Arguments

| | |
|---|---|
| x | matrix (ideally an xts object). |
| col | bivariate vector containing the colors for missing and available data, respectively. |
| xlab | x-axis label. |
| ylab | y-axis label. |
| text | see mtext(). The text = "", it is omitted. |
| side | see mtext(). |
| line | see mtext(). |
| adj | see mtext(). |
| ... | additional arguments passed to the underlying function image(). |

## Details

Indicate NAs in a data set.

## Value

invisible().

## Author(s)

Marius Hofert

## Examples

```
## Generate data
n <- 1000 # sample size
d <- 100 # dimension
set.seed(271) # set seed
x <- matrix(runif(n*d), ncol = d) # generate data

## Assign missing data
k <- ceiling(d/4) # fraction of columns with some NAs
j <- sample(1:d, size = k) # columns j with NAs
i <- sample(1:n, size = k) # 1:i will be NA in each column j
X <- x
for(k. in seq_len(k)) X[1:i[k.], j[k.]] <- NA # put in NAs

## Plot NAs
NA_plot(X) # indicate NAs
```

pp_qq_plot *P-P and Q-Q Plots*

## Description

Probability-probability plots and quantile-quantile plots.

## Usage

```
pp_plot(x, FUN, xlab = "Theoretical probabilities",
        ylab = "Sample probabilities", ...)
qq_plot(x, FUN = qnorm, xlab = "Theoretical quantiles", ylab = "Sample quantiles",
        do.qqline = TRUE, method = c("theoretical", "empirical"),
        qqline.args = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | data vector. |
| FUN | function. For<br>pp_plot(): The distribution function.<br>qq_plot(): The quantile function. |
| xlab | x-axis label. |
| ylab | y-axis label. |
| do.qqline | logical indicating whether a Q-Q line is plotted. |
| method | method used to construct the Q-Q line. If "theoretical", the theoretically true line with intercept 0 and slope 1 is displayed; if "empirical", the intercept and slope are determined with qqline(). The former helps deciding whether x comes from the distribution specified by FUN exactly, the latter whether x comes from a location-scale transformed distribution specified by FUN. |
| qqline.args | list containing additional arguments passed to the underlying abline() functions. Defaults to list(a = 0, b = 1) if method = "theoretical" and list() if method = "empirical". |
| ... | additional arguments passed to the underlying function plot(). |

## Details

Note that Q-Q plots are more widely used (as they make deviations in the tails more visible).

## Value

invisible().

## Author(s)

Marius Hofert

## Examples

```
## Generate data
n <- 1000
mu <- 1
sig <- 3
nu <- 3.5
set.seed(271) # set seed
x <- mu + sig * sqrt((nu-2)/nu) * rt(n, df = nu) # sample from t_nu(mu, sig^2)

## P-P plot
pF <- function(q) pt((q - mu) / (sig * sqrt((nu-2)/nu)), df = nu)
pp_plot(x, FUN = pF)

## Q-Q plot
qF <- function(p) mu + sig * sqrt((nu-2)/nu) * qt(p, df = nu)
qq_plot(x, FUN = qF)

## A comparison with R's qqplot() and qqline()
qqplot(qF(ppoints(length(x))), x) # the same (except labels)
qqline(x, distribution = qF) # slightly different (since *estimated*)

## Difference of the two methods
set.seed(271)
z <- rnorm(1000)
## Standardized data
qq_plot(z, FUN = qnorm) # fine
qq_plot(z, FUN = qnorm, method = "empirical") # fine
## Location-scale transformed data
mu <- 3
sig <- 2
z. <- mu+sig*z
qq_plot(z., FUN = qnorm) # not fine (z. comes from N(mu, sig^2), not N(0,1))
qq_plot(z., FUN = qnorm, method = "empirical") # fine (as intercept and slope are estimated)
```

---

`returns` *Computing Returns and Inverse Transformation*

---

## Description

Compute log-returns, simple returns and basic differences (or the inverse operations) from given data.

## Usage

```
returns(x, method = c("logarithmic", "simple", "diff"), inverse = FALSE,
        start, start.date)
```

## Arguments

| | |
|---|---|
| x | matrix or vector (possibly a xts object) to be turned into returns (if inverse = FALSE) or returns to be turned into the original data (if inverse = TRUE). |
| method | character string indicating the method to be used (log-returns (logarithmic changes), simple returns (relative changes), or basic differences). |
| inverse | logical indicating whether the inverse transformation (data from given returns) shall be computed (if TRUE, this requires start to be specified). |
| start | if inverse = TRUE, the last available value of the time series to be constructed from the given returns x. |
| start.date | character or Date object to be used as the date corresponding to the value start; currently only used for xts objects. |

## Details

If inverse = FALSE and x is an xts object, the returned object is an xts, too.

## Value

vector or matrix with the same number of columns as x just one row less if inverse = FALSE or one row more if inverse = TRUE.

## Author(s)

Marius Hofert

## Examples

```
## Generate two paths of a geometric Brownian motion
S0 <- 10 # current stock price S_0
r <- 0.01 # risk-free annual interest rate
sig <- 0.2 # (constant) annual volatility
T <- 2 # maturity in years
N <- 250 # business days per year
t <- 1:(N*T) # time points to be sampled
npath <- 2 # number of paths
set.seed(271) # for reproducibility
S <- replicate(npath, S0 * exp(cumsum(rnorm(N*T, # sample paths of S_t
                                            mean = (r-sig^2/2)/N,
                                            sd = sqrt((sig^2)/N))))) # (N*T, npath)

## Turn into xts objects
library(xts)
sdate <- as.Date("2000-05-02") # start date
S.  <- as.xts(S, order.by = seq(sdate, length.out = N*T, by = "1 week"))
plot(S.[,1], main = "Stock 1")
plot(S.[,2], main = "Stock 2")


### Log-returns ################################################################
```

```
## Based on S[,1]
X <- returns(S[,1]) # build log-returns (one element less than S)
Y <- returns(X, inverse = TRUE, start = S[1,1]) # transform back
stopifnot(all.equal(Y, S[,1]))

## Based on S
X <- returns(S) # build log-returns (one element less than S)
Y <- returns(X, inverse = TRUE, start = S[1,]) # transform back
stopifnot(all.equal(Y, S))

## Based on S.[,1]
X <- returns(S.[,1])
Y <- returns(X, inverse = TRUE, start = S.[1,1], start.date = sdate)
stopifnot(all.equal(Y, S.[,1], check.attributes = FALSE))

## Based on S.
X <- returns(S.)
Y <- returns(X, inverse = TRUE, start = S.[1], start.date = sdate)
stopifnot(all.equal(Y, S., check.attributes = FALSE))

## Sign-adjusted (negative) log-returns
X <- -returns(S) # build -log-returns
Y <- returns(-X, inverse = TRUE, start = S[1,]) # transform back
stopifnot(all.equal(Y, S))


### Simple returns ##############################################################

## Simple returns based on S
X <- returns(S, method = "simple")
Y <- returns(X, method = "simple", inverse = TRUE, start = S[1,])
stopifnot(all.equal(Y, S))

## Simple returns based on S.
X <- returns(S., method = "simple")
Y <- returns(X, method = "simple", inverse = TRUE, start = S.[1,],
             start.date = sdate)
stopifnot(all.equal(Y, S., check.attributes = FALSE))

## Sign-adjusted (negative) simple returns
X <- -returns(S, method = "simple")
Y <- returns(-X, method = "simple", inverse = TRUE, start = S[1,])
stopifnot(all.equal(Y, S))


### Basic differences ###########################################################

## Basic differences based on S
X <- returns(S, method = "diff")
Y <- returns(X, method = "diff", inverse = TRUE, start = S[1,])
stopifnot(all.equal(Y, S))
```

```
## Basic differences based on S.
X <- returns(S., method = "diff")
Y <- returns(X, method = "diff", inverse = TRUE, start = S.[1,],
             start.date = sdate)
stopifnot(all.equal(Y, S., check.attributes = FALSE))

## Sign-adjusted (negative) basic differences
X <- -returns(S, method = "diff")
Y <- returns(-X, method = "diff", inverse = TRUE, start = S[1,])
stopifnot(all.equal(Y, S))
```

---

`risk_measures` *Risk Measures*

---

## Description

Computing risk measures.

## Usage

```
## value-at-risk
VaR_np(x, alpha, names = FALSE, type = 1, ...)
VaR_t(alpha, mu = 0, sigma = 1, df = Inf)
VaR_Par(alpha, theta, kappa = 1)

## expected shortfall
ES_np(x, alpha, method = c(">", ">="), verbose = FALSE, ...)
ES_t(alpha, mu = 0, sigma = 1, df = Inf)
ES_Par(alpha, theta, kappa = 1)

## multivariate geometric value-at-risk and expectiles
gVaR(x, alpha, start = colMeans(x),
     method = if(length(alpha) == 1) "Brent" else "Nelder-Mead", ...)
gEX(x, alpha, start = colMeans(x),
    method = if(length(alpha) == 1) "Brent" else "Nelder-Mead", ...)
```

## Arguments

| | |
|---|---|
| `x` | `gVaR()`, `gEX()` `matrix` of (rowwise) multivariate losses.<br>**otherwise** `vector` of losses. |
| `alpha` | `gVaR()`, `gEX()` `vector` or `matrix` of (rowwise) confidence levels (all in $[0, 1]$).<br>**otherwise** confidence level $\alpha \in [0, 1]$. |
| `names` | see `?quantile`. |
| `type` | see `?quantile`. |
| `mu` | location parameter. |
| `sigma` | scale parameter, a positive number. |

| | |
|---|---|
| df | degrees of freedom, a positive number; choose df = Inf for the normal distribution. |
| theta | Pareto shape parameter, a positive number. |
| kappa | Pareto scale parameter, a positive number. |
| start | vector of initial values for the underlying optim(). |
| method | ES_np() character string indicating the method for computing expected shortfall.<br>gVaR(), gEX() the optimization method passed to the underlying optim(). |
| verbose | logical indicating whether verbose output is given (in case the mean is computed over (too) few observations). |
| ... | VaR_np() additional arguments passed to the underlying quantile().<br>ES_np() additional arguments passed to VaR_np().<br>gVaR(), gEX() additional arguments passed to the underlying optim(). |

## Details

The distribution function of the Pareto distribution is given by

$$F(x) = 1 - (\kappa/(\kappa + x))^{\theta},\ x \geq 0,$$

where $\theta > 0$, $\kappa > 0$.

## Value

VaR_np(), ES_np() estimate value-at-risk and expected shortfall non-parametrically. For the latter, the mean over all losses (strictly) beyond value-at-risk is computed. If method = ">=", there is always at least one such loss, whereas if method = ">", there might be no such loss, in which case NaN is returned.

VaR_t(), ES_t() compute value-at-risk and expected shortfall for the $t$ (or normal) distribution.

VaR_Par(), ES_Par() compute value-at-risk and expected shortfall for the Pareto distribution.

gVaR(), gEX() compute the multivariate geometric value-at-risk and expectiles suggested by Chaudhuri (1996) and Herrmann et al. (2017), respectively.

## Author(s)

Marius Hofert

## References

McNeil, A. J., Frey, R. and Embrechts, P. (2015). *Quantitative Risk Management: Concepts, Techniques, Tools*. Princeton University Press.

Chaudhuri, P. (1996). On a geometric notion of quantiles for multivariate data. *Journal of the American Statistical Assosiation* 91(434), 862–872.

Herrmann, K., Hofert, M. and Mailhot, M. (2017). Multivariate geometric expectiles.

## Examples

```
### 1 Univariate measures ##################################################

## Generate some losses and (non-parametrically) estimate VaR_alpha and ES_alpha
set.seed(271)
L <- rlnorm(1000, meanlog = -1, sdlog = 2) # L ~ LN(mu, sig^2)
## Note: - meanlog = mean(log(L)) = mu, sdlog = sd(log(L)) = sig
##       - E(L) = exp(mu + (sig^2)/2), var(L) = (exp(sig^2)-1)*exp(2*mu + sig^2)
##         To obtain a sample with E(L) = a and var(L) = b, use:
##         mu = log(a)-log(1+b/a^2)/2 and sig = sqrt(log(1+b/a^2))
VaR_np(L, alpha = 0.99)
ES_np(L,  alpha = 0.99)

## Example 2.16 in McNeil, Frey, Embrechts (2015)
V <- 10000 # value of the portfolio today
sig <- 0.2/sqrt(250) # daily volatility (annualized volatility of 20%)
nu <- 4 # degrees of freedom for the t distribution
alpha <- seq(0.001, 0.999, length.out = 256) # confidence levels
VaRnorm <- VaR_t(alpha, sigma = V*sig, df = Inf)
VaRt4 <- VaR_t(alpha, sigma = V*sig*sqrt((nu-2)/nu), df = nu)
ESnorm <- ES_t(alpha, sigma = V*sig, df = Inf)
ESt4 <- ES_t(alpha, sigma = V*sig*sqrt((nu-2)/nu), df = nu)
ran <- range(VaRnorm, VaRt4, ESnorm, ESt4)
plot(alpha, VaRnorm, type = "l", ylim = ran, xlab = expression(alpha), ylab = "")
lines(alpha, VaRt4, col = "royalblue3")
lines(alpha, ESnorm, col = "darkorange2")
lines(alpha, ESt4, col = "maroon3")
legend("bottomright", bty = "n", lty = rep(1,4), col = c("black",
       "royalblue3", "darkorange3", "maroon3"),
       legend = c(expression(VaR[alpha]~~"for normal model"),
                  expression(VaR[alpha]~~"for "*t[4]*" model"),
                  expression(ES[alpha]~~"for normal model"),
                  expression(ES[alpha]~~"for "*t[4]*" model")))


### 2 Multivariate measures ################################################

## Setup
library(copula)
n <- 1e4 # MC sample size
nu <- 3 # degrees of freedom
th <- iTau(tCopula(df = nu), tau = 0.5) # correlation parameter
cop <- tCopula(param = th, df = nu) # t copula
set.seed(271) # for reproducibility
U <- rCopula(n, cop = cop) # copula sample
theta <- c(2.5, 4) # marginal Pareto parameters
stopifnot(theta > 2) # need finite 2nd moments
X <- sapply(1:2, function(j) qPar(U[,j], theta = theta[j])) # generate X
N <- 17 # number of angles (rather small here because of run time)
phi <- seq(0, 2*pi, length.out = N) # angles
r <- 0.98 # radius
alpha <- r * cbind(alpha1 = cos(phi), alpha2 = sin(phi)) # vector of confidence levels
```

```
## Compute geometric value-at-risk
system.time(res <- gVaR(X, alpha = alpha))
gvar <- t(sapply(seq_len(nrow(alpha)), function(i) {
    x <- res[[i]]
    if(x[["convergence"]] != 0) # 0 = 'converged'
        warning("No convergence for alpha = (", alpha[i,1], ", ", alpha[i,2],
                ") (row ", i, ")")
    x[["par"]]
})) # (N, 2)-matrix

## Compute geometric expectiles
system.time(res <- gEX(X, alpha = alpha))
gex <- t(sapply(seq_len(nrow(alpha)), function(i) {
    x <- res[[i]]
    if(x[["convergence"]] != 0) # 0 = 'converged'
        warning("No convergence for alpha = (", alpha[i,1], ", ", alpha[i,2],
                ") (row ", i, ")")
    x[["par"]]
})) # (N, 2)-matrix

## Plot geometric VaR and geometric expectiles
plot(gvar, type = "b", xlab = "Component 1 of geometric VaRs and expectiles",
     ylab = "Component 2 of geometric VaRs and expectiles",
     main = "Multivariate geometric VaRs and expectiles")
lines(gex, type = "b", col = "royalblue3")
legend("bottomleft", lty = 1, bty = "n", col = c("black", "royalblue3"),
       legend = c("geom. VaR", "geom. expectile"))
lab <- substitute("MC sample size n ="~n.*","~t[nu.]~"copula with Par("*th1*
                  ") and Par("*th2*") margins",
                  list(n. = n, nu. = nu, th1 = theta[1], th2 = theta[2]))
mtext(lab, side = 4, line = 1, adj = 0)
```

---

VaR_ES_bounds *Worst and Best Value-at-Risk and Best Expected Shortfall for Given Marginals*

---

## Description

Compute the worst and best Value-at-Risk (VaR) and the best expected shortfall (ES) for given marginal distributions.

## Usage

```
## Homogeneous case
crude_VaR_bounds(alpha, qF, d = NULL, ...)
VaR_bounds_hom(alpha, d, method = c("Wang", "Wang.Par", "dual"),
               interval = NULL, tol = NULL, ...)
dual_bound(s, d, pF, tol = .Machine$double.eps^0.25, ...)
```

```
## Inomogeneous case

## Workhorses
rearrange(X, tol = 0, tol.type = c("relative", "absolute"),
          n.lookback = ncol(X), max.ra = Inf,
          method = c("worst.VaR", "best.VaR", "best.ES"),
  sample = TRUE, is.sorted = FALSE, trace = FALSE, ...)
block_rearrange(X, tol = 0, tol.type = c("absolute", "relative"),
                n.lookback = ncol(X), max.ra = Inf,
                method = c("worst.VaR", "best.VaR", "best.ES"),
                sample = TRUE, trace = FALSE, ...)

## User interfaces
## Rearrangement Algorithm
RA(alpha, qF, N, abstol = 0, n.lookback = length(qF), max.ra = Inf,
   method = c("worst.VaR", "best.VaR", "best.ES"), sample = TRUE)
## Adaptive Rearrangement Algorithm
ARA(alpha, qF, N.exp = seq(8, 19, by = 1), reltol = c(0, 0.01),
    n.lookback = length(qF), max.ra = 10*length(qF),
    method = c("worst.VaR", "best.VaR", "best.ES"),
    sample = TRUE)
## Adaptive Block Rearrangement Algorithm
ABRA(alpha, qF, N.exp = seq(8, 19, by = 1), absreltol = c(0, 0.01),
     n.lookback = NULL, max.ra = Inf,
     method = c("worst.VaR", "best.VaR", "best.ES"),
     sample = TRUE)
```

## Arguments

| | |
|---|---|
| alpha | confidence level for VaR and ES (e.g., 0.99). |
| d | dimension (number of risk factors; $\geq 2$). For `crude_VaR_bounds()`, `d` only needs to be given in the homogeneous case in which `qF()` is a `function`. |
| qF | marginal quantile function (in the homogeneous case) or a `d`-list containing the marginal quantile functions (in the general case). |
| method | `character` string. For |

`VaR_bounds_hom()`: `method = "Wang"` and `method = "Wang.Par"` apply the approach of McNeil et al. (2015, Proposition 8.32) for computing best (i.e., smallest) and worst (i.e., largest) VaR. The latter method assumes Pareto margins and thus does not require numerical integration. `method = "dual"` applies the dual bound approach as in Embrechts et al. (2013, Proposition 4) for computing worst VaR (no value for the best VaR can be obtained with this approach and thus `NA` is returned for the best VaR).

`rearrange()`, `block_rearrange()`, `RA()`, `ARA()`: `method` indicates whether bounds for the worst/best VaR or the best ES should be computed. These bounds are termed $\underline{s}_N$ and $\overline{s}_N$ in the literature (and below) and are theoretically not guaranteed bounds of worst/best VaR or best ES; however, they are treated

as such in practice and are typically in line with results from `VaR_bounds_hom()` in the homogeneous case, for example.

`interval` initial interval (a `numeric(2)`) for computing worst VaR. If not provided, these are the defaults chosen:

`method = "Wang"`: initial interval is $[0, (1-\alpha)/d]$.

`method = "Wang.Par"`: initial interval is $[c_l, c_u]$, where $c_l$ and $c_u$ are chosen as in Hofert et al. (2015).

`method = "dual"`: in this case, no good defaults are known. Note that the lower endpoint of the initial interval has to be sufficiently large in order for the the inner root-finding algorithm to find a root; see Details.

`tol` `VaR_bounds_hom()`: tolerance for `uniroot()` for computing worst VaR. This defaults (for `tol = NULL`) to $2.2204 * 10^{-16}$ for `method = "Wang"` or `method = "Wang.Par"` (where a smaller tolerance is crucial) and to `uniroot()`'s default `.Machine$double.eps^0.25` otherwise. Note that for `method = "dual"`, `tol` is used for both the outer and the inner root-finding procedure.

`rearrange()`, `block_rearrange()`: (absolute or relative) tolerance to determine (the individual) convergence. This should normally be a number greater than or equal to 0, but `rearrange()` also allows for `tol = NULL` which means that columns are rearranged until each column is oppositely ordered to the sum of all other columns.

`tol.type` `character` string indicating the type of convergence tolerance function to be used (`"relative"` for relative tolerance and `"absolute"` for absolute tolerance).

`n.lookback` number of rearrangements to look back for deciding about numerical convergence. Use this option with care.

`s` dual bound evaluation point.

`pF` marginal loss distribution function.

`X` (`N`, `d`)-matrix of quantiles (to be rearranged). If `is.sorted` it is assumed that the columns of `X` are sorted in *increasing* order.

`max.ra` maximal number of (considered) column rearrangements of the underlying matrix of quantiles (can be set to `Inf`).

`N` number of discretization points.

`N.exp` exponents of the number of discretization points (a `vector`) over which the algorithm iterates to find the smallest number of discretization points for which the desired accuracy (specified by `abstol` and `reltol`) is attained; for each number of discretization points, at most `max.ra`-many column rearrangements are of the underlying matrix of quantiles are considered.

`abstol` absolute convergence tolerance $\epsilon$ to determine the individual convergence, i.e., the change in the computed minimal row sums (for `method = "worst.VaR"`) or maximal row sums (for `method = "best.VaR"`) or expected shortfalls (for `method = "best.ES"`) for the lower bound $\underline{s}_N$ and the upper bound $\overline{s}_N$. `abstol` is typically $\geq 0$; it can also be `NULL`, see `tol` above.

reltol — vector of length two containing the individual (first component; used to determine convergence of the minimal row sums (for method = "worst.VaR") or maximal row sums (for method = "best.VaR") or expected shortfalls (for method = "best.ES") for $\underline{s}_N$ and $\overline{s}_N$) and the joint (second component; relative tolerance between the computed $\underline{s}_N$ and $\overline{s}_N$ with respect to $\overline{s}_N$) relative convergence tolerances. reltol can also be of length one in which case it denotes the joint relative tolerance; the individual relative tolerance is taken as NULL (see tol above) in this case.

absreltol — vector of length two containing the individual (first component; used to determine convergence of the minimal row sums (for method = "worst.VaR") or maximal row sums (for method = "best.VaR") or expected shortfalls (for method = "best.ES") for $\underline{s}_N$ and $\overline{s}_N$) absolute and the joint (second component; relative tolerance between the computed $\underline{s}_N$ and $\overline{s}_N$ with respect to $\overline{s}_N$) relative convergence tolerances. absreltol can also be of length one in which case it denotes the joint relative tolerance; the individual absolute tolerance is taken as 0 in this case.

sample — logical indicating whether each column of the two underlying matrices of quantiles (see Step 3 of the Rearrangement Algorithm in Embrechts et al. (2013)) are randomly permuted before the rearrangements begin. This typically has quite a positive effect on run time (as most of the time is spent (oppositely) ordering columns (for rearrange()) or blocks (for block_rearrange())).

is.sorted — logical indicating whether the columns of X are sorted in increasing order.

trace — logical indicating whether the underlying matrix is printed after each rearrangement step. See vignette("VaR_bounds", package = "qrmtools") for how to interpret the output.

... — crude_VaR_bounds(): ellipsis argument passed to (all provided) quantile functions.

VaR_bounds_hom(): case method = "Wang" requires the quantile function qF() to be provided and additional arguments passed via the ellipsis argument are passed on to the underlying integrate(). For method = "Wang.Par" the ellipsis argument must contain the parameter $\theta > 0$ of the Pareto distribution. For method = "dual", the ellipsis argument must contain the distribution function pF() and the initial interval interval for the outer root finding procedure (not for d = 2); additional arguments are passed on to the underlying integrate() for computing the dual bound $D(s)$.

dual_bound(): ellipsis argument is passed to the underlying integrate().

rearrange(): additional arguments passed to the underlying optimization function. Currently, this is only used if method = "best.ES" in which case the required confidence level $\alpha$ must be provided as argument alpha.

## Details

For d = 2, VaR_bounds_hom() uses the method of Embrechts et al. (2013, Proposition 2). For method = "Wang" and method = "Wang.Par" the method presented in McNeil et al. (2015, Prop. 8.32) is implemented; this goes back to Embrechts et al. (2014, Prop. 3.1; note that the published version of this paper contains typos for both bounds). This requires one uniroot() and, for the

generic method = "Wang", one integrate(). The critical part for the generic method = "Wang" is the lower endpoint of the initial interval for uniroot(). If the (marginal) distribution function has finite first moment, this can be taken as 0. However, if it has infinite first moment, the lower endpoint has to be positive (but must lie below the unknown root). Note that the upper endpoint $(1-\alpha)/d$ also happens to be a root and thus one needs a proper initional interval containing the root and being stricticly contained in $(0, (1-\alpha)/d$. In the case of Pareto margins, Hofert et al. (2015) have derived such an initial (which is used by method = "Wang.Par"). Also note that the chosen smaller default tolerances for uniroot() in case of method = "Wang" and method = "Wang.Par" are crucial for obtaining reliable VaR values; see Hofert et al. (2015).

For method = "dual" for computing worst VaR, the method presented of Embrechts et al. (2013, Proposition 4) is implemented. This requires two (nested) uniroot(), and an integrate(). For the inner root-finding procedure to find a root, the lower endpoint of the provided initial interval has to be "sufficiently large".

Note that these approaches for computing the VaR bounds in the homogeneous case are numerically non-trivial; see the source code and vignette("VaR_bounds", package = "qrmtools") for more details. As a rule of thumb, use method = "Wang" if you have to (i.e., if the margins are not Pareto) and method = "Wang.Par" if you can (i.e., if the margins are Pareto). It is not recommended to use (the numerically even more challenging) method = "dual".

Concerning the inhomogeneous case, rearrange() is an auxiliary function (workhorse). It is called by RA() and ARA(). After a column rearrangement of X, the tolerance between the minimal row sum (for the worst VaR) or maximal row sum (for the best VaR) or expected shortfall (obtained from the row sums; for the best ES) after this rearrangement and the one of $n.lookback$ rearrangement steps before is computed and convergence determined. For performance reasons, no input checking is done for rearrange() and it can change in future versions to (futher) improve run time. Overall it should only be used by experts.

block_rearrange(), the workhorse underlying ABRA(), is similar to rearrange() in that it checks whether convergence has occurred after every rearrangement by comparing the change to the row sum variance from n.lookback rearrangement steps back. block_rearrange() differs from rearrange in the following ways. First, instead of single columns, whole (randomly chosen) blocks (two at a time) are chosen and oppositely ordered. Since some of the ideas for improving the speed of rearrange() do not carry over to block_rearrange(), the latter should in general not be as fast as the former. Second, instead of using minimal or maximal row sums or expected shortfall to determine numerical convergence, block_rearrange() uses the variance of the vector of row sums to determine numerical convergence. By default, it targets a variance of 0 (which is also why the default tol.type is "absolute").

For the Rearrangement Algorithm RA(), convergence of $\underline{s}_N$ and $\overline{s}_N$ is determined if the minimal row sum (for the worst VaR) or maximal row sum (for the best VaR) or expected shortfall (obtained from the row sums; for the best ES) satisfies the specified abstol (so $\leq \epsilon$) after at most max.ra-many column rearrangements. This is different from Embrechts et al. (2013) who use $< \epsilon$ and only check for convergence after an iteration through all columns of the underlying matrix of quantiles has been completed.

For the Adaptive Rearrangement Algorithm ARA() and the Adaptive Block Rearrangement Algorithm ABRA(), convergence of $\underline{s}_N$ and $\overline{s}_N$ is determined if, after at most max.ra-many column rearrangements, the (the individual relative tolerance) reltol[1] is satisfied *and* the relative (joint) tolerance between both bounds is at most reltol[2].

Note that RA(), ARA() and ABRA() need to evalute the 0-quantile (for the lower bound for the best VaR) and the 1-quantile (for the upper bound for the worst VaR). As the algorithms, due to

performance reasons, can only handle finite values, the 0-quantile and the 1-quantile need to be adjusted if infinite. Instead of the 0-quantile, the $\alpha/(2N)$-quantile is computed and instead of the 1-quantile the $\alpha + (1-\alpha)(1-1/(2N))$-quantile is computed for such margins (if the 0-quantile or the 1-quantile is finite, no adjustment is made).

`rearrange()`, `block_rearrange()`, `RA()`, `ARA()` and `ABRA()` compute $\underline{s}_N$ and $\overline{s}_N$ which are, from a practical point of view, treated as bounds for the worst (i.e., largest) or the best (i.e., smallest) VaR or the best (i.e., smallest ES), but which are not known to be such bounds from a theoretical point of view; see also above. Calling them "bounds" for worst/best VaR or best ES is thus theoretically not correct (unless proven) but "practical". The literature thus speaks of $(\underline{s}_N, \overline{s}_N)$ as the rearrangement gap.

More details not provided here can be found in the references listed below.

## Value

`crude_VaR_bounds()` returns crude lower and upper bounds for VaR at confidence level $\alpha$ for any $d$-dimensional model with marginal quantile functions specified by `qF`.

`VaR_bounds_hom()` returns the best and worst VaR at confidence level $\alpha$ for $d$ risks with equal distribution function specified by the ellipsis ....

`dual_bound()` returns the value of the dual bound $D(s)$ as given in Embrechts, Puccetti, Rüschendorf (2013, Eq. (12)).

`rearrange()` and `block_rearrange()` return a `list` containing

`bound`: computed $\underline{s}_N$ or $\overline{s}_N$.

`tol`: reached tolerance (i.e., the (absolute or relative) change of the minimal row sum (for `method = "worst.VaR"`) or maximal row sum (for `method = "best.VaR"`) or expected shortfall (for `method = "best.ES"`) after the last rearrangement).

`converged`: `logical` indicating whether the desired (absolute or relative) tolerance `tol` has been reached.

`opt.row.sums`: `vector` containing the computed optima (minima for `method = "worst.VaR"`; maxima for `method = "best.VaR"`; expected shortfalls for `method = "best.ES"`) for the row sums after each (considered) rearrangement.

`X.rearranged`: (`N`, `d`)-`matrix` containing the rearranged `X`.

`RA()` returns a `list` containing

`bounds`: bivariate vector containing the computed $\underline{s}_N$ and $\overline{s}_N$ (the so-called rearrangement range) which are typically treated as bounds for worst/best VaR or best ES; see also above.

`rel.ra.gap`: reached relative tolerance (also known as relative rearrangement gap) between $\underline{s}_N$ and $\overline{s}_N$ computed with respect to $\overline{s}_N$.

`ind.abs.tol`: bivariate `vector` containing the reached individual absolute tolerances (i.e., the absolute change of the minimal row sums (for `method = "worst.VaR"`) or maximal row sums (for `method = "best.VaR"`) or expected shortfalls (for `mehtod = "best.ES"`) for computing $\underline{s}_N$ and $\overline{s}_N$; see also `tol` returned by `rearrange()` above).

`converged`: bivariate `logical` vector indicating convergence of the computed $\underline{s}_N$ and $\overline{s}_N$ (i.e., whether the desired tolerances were reached).

`num.ra`: bivariate vector containing the number of column rearrangments of the underlying matrices of quantiles for $\underline{s}_N$ and $\overline{s}_N$.

`opt.row.sums`: `list` of length two containing the computed optima (minima for `method = "worst.VaR"`; maxima for `method = "best.VaR"`; expected shortfalls for `method = "best.ES"`) for the row sums after each (considered) column rearrangement for the computed $\underline{s}_N$ and $\overline{s}_N$; see also `rearrange()`.

`X`: initially constructed (`N`, `d`)-matrices of quantiles for computing $\underline{s}_N$ and $\overline{s}_N$.

`X.rearranged`: rearranged matrices `X` (for $\underline{s}_N$ and $\overline{s}_N$).

`ARA()` and `ABRA()` return a `list` containing

`bounds`: see `RA()`.

`rel.ra.gap`: see `RA()`.

`tol`: trivariate `vector` containing the reached individual (relative for `ARA()`; absolute for `ABRA()`) tolerances and the reached joint relative tolerance (computed with respect to $\overline{s}_N$).

`converged`: trivariate `logical vector` indicating individual convergence of the computed $\underline{s}_N$ (first entry) and $\overline{s}_N$ (second entry) and indicating joint convergence of the two bounds according to the attained joint relative tolerance (third entry).

`N.used`: actual `N` used for computing the (final) $\underline{s}_N$ and $\overline{s}_N$.

`num.ra`: see `RA()`; computed for `N.used`.

`opt.row.sums`: see `RA()`; computed for `N.used`.

`X`: see `RA()`; computed for `N.used`.

`X.rearranged`: see `RA()`; computed for `N.used`.

## Author(s)

Marius Hofert

## References

Embrechts, P., Puccetti, G., Rüschendorf, L., Wang, R., Beleraj, A. (2014). An Academic Response to Basel 3.5. *Risks* **2**(1), 25–48.

Embrechts, P., Puccetti, G., Rüschendorf, L. (2013). Model uncertainty and VaR aggregation. *Journal of Banking \& Finance* **37**, 2750–2764.

McNeil, A. J., Frey, R., and Embrechts, P. (2015). *Quantitative Risk Management: Concepts, Techniques, Tools*. Princeton University Press.

Hofert, M., Memartoluie, A., Saunders, D., Wirjanto, T. (2017). Improved Algorithms for Computing Worst Value-at-Risk. *Statistics \& Risk Modeling* or, for an earlier version, http://arxiv.org/abs/1505.02281.

Bernard, C., Rüschendorf, L., Vanduffel, S. (2013). Value-at-Risk bounds with variance constraints. See http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2342068.

Bernard, C. and McLeish, D. (2014). Algorithms for Finding Copulas Minimizing Convex Functions of Sums. See http://arxiv.org/abs/1502.02130v3.

## See Also

vignette("VaR_bounds", package = "qrmtools") for more example calls, numerical challenges encoutered and a comparison of the different methods for computing the worst (i.e., largest) Value-at-Risk.

## Examples

```
### 1 Reproducing selected examples of McNeil et al. (2015; Table 8.1) #########

## Setup
alpha <- 0.95
d <- 8
theta <- 3
qF <- rep(list(function(p) qPar(p, theta = theta)), d)

## Worst VaR
N <- 5e4
set.seed(271)
system.time(RA.worst.VaR <- RA(alpha, qF = qF, N = N, method = "worst.VaR"))
RA.worst.VaR$bounds
stopifnot(RA.worst.VaR$converged,
          all.equal(RA.worst.VaR$bounds[["low"]],
                    RA.worst.VaR$bounds[["up"]], tol = 1e-4))

## Best VaR
N <- 5e4
set.seed(271)
system.time(RA.best.VaR <- RA(alpha, qF = qF, N = N, method = "best.VaR"))
RA.best.VaR$bounds
stopifnot(RA.best.VaR$converged,
          all.equal(RA.best.VaR$bounds[["low"]],
                    RA.best.VaR$bounds[["up"]], tol = 1e-4))

## Best ES
N <- 5e4 # actually, we need a (much larger) N here (but that's time consuming)
set.seed(271)
system.time(RA.best.ES <- RA(alpha, qF = qF, N = N, method = "best.ES"))
RA.best.ES$bounds
stopifnot(RA.best.ES$converged,
          all.equal(RA.best.ES$bounds[["low"]],
                    RA.best.ES$bounds[["up"]], tol = 5e-1))


### 2 More Pareto examples (d = 2, d = 8; hom./inhom. case; explicit/RA/ARA) ###

alpha <- 0.99 # VaR confidence level
th <- 2 # Pareto parameter theta
qF <- function(p, theta = th) qPar(p, theta = theta) # Pareto quantile function
pF <- function(q, theta = th) pPar(q, theta = theta) # Pareto distribution function


### 2.1 The case d = 2 ############################################################
```

```
d <- 2 # dimension

## Explicit
VaRbounds <- VaR_bounds_hom(alpha, d = d, qF = qF) # (best VaR, worst VaR)

## Adaptive Rearrangement Algorithm (ARA)
set.seed(271) # set seed (for reproducibility)
ARAbest  <- ARA(alpha, qF = rep(list(qF), d), method = "best.VaR")
ARAworst <- ARA(alpha, qF = rep(list(qF), d))

## Rearrangement Algorithm (RA) with N as in ARA()
RAbest  <- RA(alpha, qF = rep(list(qF), d), N = ARAbest$N.used, method = "best.VaR")
RAworst <- RA(alpha, qF = rep(list(qF), d), N = ARAworst$N.used)

## Compare
stopifnot(all.equal(c(ARAbest$bounds[1], ARAbest$bounds[2],
                      RAbest$bounds[1],  RAbest$bounds[2]),
                    rep(VaRbounds[1], 4), tolerance = 0.004, check.names = FALSE))
stopifnot(all.equal(c(ARAworst$bounds[1], ARAworst$bounds[2],
                      RAworst$bounds[1],  RAworst$bounds[2]),
                    rep(VaRbounds[2], 4), tolerance = 0.003, check.names = FALSE))


### 2.2 The case d = 8 ##########################################################

d <- 8 # dimension

## Compute VaR bounds with various methods
I <- crude_VaR_bounds(alpha, qF = qF, d = d) # crude bound
VaR.W     <- VaR_bounds_hom(alpha, d = d, method = "Wang", qF = qF)
VaR.W.Par <- VaR_bounds_hom(alpha, d = d, method = "Wang.Par", theta = th)
VaR.dual  <- VaR_bounds_hom(alpha, d = d, method = "dual", interval = I, pF = pF)

## Adaptive Rearrangement Algorithm (ARA) (with different relative tolerances)
set.seed(271) # set seed (for reproducibility)
ARAbest  <- ARA(alpha, qF = rep(list(qF), d), reltol = c(0.001, 0.01), method = "best.VaR")
ARAworst <- ARA(alpha, qF = rep(list(qF), d), reltol = c(0.001, 0.01))

## Rearrangement Algorithm (RA) with N as in ARA and abstol (roughly) chosen as in ARA
RAbest  <- RA(alpha, qF = rep(list(qF), d), N = ARAbest$N.used,
              abstol = mean(tail(abs(diff(ARAbest$opt.row.sums$low)), n = 1),
                            tail(abs(diff(ARAbest$opt.row.sums$up)), n = 1)),
              method = "best.VaR")
RAworst <- RA(alpha, qF = rep(list(qF), d), N = ARAworst$N.used,
              abstol = mean(tail(abs(diff(ARAworst$opt.row.sums$low)), n = 1),
                            tail(abs(diff(ARAworst$opt.row.sums$up)), n = 1)))

## Compare
stopifnot(all.equal(c(VaR.W[1], ARAbest$bounds, RAbest$bounds),
                    rep(VaR.W.Par[1],5), tolerance = 0.004, check.names = FALSE))
stopifnot(all.equal(c(VaR.W[2], VaR.dual[2], ARAworst$bounds, RAworst$bounds),
                    rep(VaR.W.Par[2],6), tolerance = 0.003, check.names = FALSE))
```

```
## Using (some of) the additional results computed by (A)RA()
xlim <- c(1, max(sapply(RAworst$opt.row.sums, length)))
ylim <- range(RAworst$opt.row.sums)
plot(RAworst$opt.row.sums[[2]], type = "l", xlim = xlim, ylim = ylim,
     xlab = "Number or rearranged columns",
     ylab = paste0("Minimal row sum per rearranged column"),
     main = substitute("Worst VaR minimal row sums ("*alpha==a.*","~d==d.*" and Par("*
                       th.*"))", list(a. = alpha, d. = d, th. = th)))
lines(1:length(RAworst$opt.row.sums[[1]]), RAworst$opt.row.sums[[1]], col = "royalblue3")
legend("bottomright", bty = "n", lty = rep(1,2),
       col = c("black", "royalblue3"), legend = c("upper bound", "lower bound"))
## => One should use ARA() instead of RA()


### 3 "Reproducing" examples from Embrechts et al. (2013) ######################

### 3.1 "Reproducing" Table 1 (but seed and eps are unknown) ###################

## Left-hand side of Table 1
N <- 50
d <- 3
qPar <- rep(list(qF), d)
p <- alpha + (1-alpha)*(0:(N-1))/N # for 'worst' (= largest) VaR
X <- sapply(qPar, function(qF) qF(p))
cbind(X, rowSums(X))

## Right-hand side of Table 1
set.seed(271)
res <- RA(alpha, qF = qPar, N = N)
row.sum <- rowSums(res$X.rearranged$low)
cbind(res$X.rearranged$low, row.sum)[order(row.sum),]


### 3.2 "Reproducing" Table 3 for alpha = 0.99 #################################

## Note: The seed for obtaining the exact results as in Table 3 is unknown
N <- 2e4 # we use a smaller N here to save run time
eps <- 0.1 # absolute tolerance
xi <- c(1.19, 1.17, 1.01, 1.39, 1.23, 1.22, 0.85, 0.98)
beta <- c(774, 254, 233, 412, 107, 243, 314, 124)
qF.lst <- lapply(1:8, function(j){ function(p) qGPD(p, xi = xi[j], beta = beta[j])})
set.seed(271)
res.best <- RA(0.99, qF = qF.lst, N = N, abstol = eps, method = "best.VaR")
print(format(res.best$bounds, scientific = TRUE), quote = FALSE) # close to first value of 1st row
res.worst <- RA(0.99, qF = qF.lst, N = N, abstol = eps)
print(format(res.worst$bounds, scientific = TRUE), quote = FALSE) # close to last value of 1st row


### 4 Further checks ###########################################################

## Calling the workhorses directly
set.seed(271)
```

```
ra <- rearrange(X)
bra <- block_rearrange(X)
stopifnot(ra$converged, bra$converged,
          all.equal(ra$bound, bra$bound, tolerance = 5e-3))

## Checking ABRA against ARA
set.seed(271)
ara  <- ARA (alpha, qF = qPar)
abra <- ABRA(alpha, qF = qPar)
stopifnot(ara$converged, abra$converged,
          all.equal(ara$bound[["low"]], abra$bound[["low"]], tolerance = 1e-3),
          all.equal(ara$bound[["up"]],  abra$bound[["up"]],  tolerance = 5e-3))
```

# Index