Package 'qrmtools'

November 12, 2018

Version 0.0-10

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.

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

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 (>= 3) | file LICENCE

NeedsCompilation yes

VignetteBuilder knitr

Repository CRAN

Date/Publication 2018-11-12 05:50:03 UTC

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

See Also

fit_GARCH_11() for fast(er) and numerically more robust fitting of GARCH(1,1) processes.

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"))
```

---

edf_plot *Plot of an Empirical Distribution Function*

Description

Plotting an empirical distribution function.

Usage

```
edf_plot(x, do.points = length(x) <= 100, log = "",
         xlim = range(x, na.rm = TRUE),
         main = "", xlab = "x", ylab = "Distribution function at x", ...)
```

Arguments

x	numeric vector of data (of which the empirical distribution function is to be plotted).
do.points	logical indicating whether points are to be plotted; see ?plot.stepfun.
log	character indicating whether a logarithmic x-axis is used (available are only "" and "x").
xlim	x-axis limits; default range(x) avoids possible failure if log = "x" and data points are all positive (plot.stepfun() extends the range, possibly below 0).
main	title.
xlab	x-axis label.
ylab	y-axis label.
...	additional arguments passed to the underlying plot.stepfun().

Value

Returns the return value of the underlying plot.stepfun().

Author(s)

Marius Hofert

Examples

```
## Example 1
x <- c(5, 2, 4, 2, 3, 2, 2, 2, 1, 2) # test data
edf_plot(x, verticals = FALSE) # the 'mathematical' version
edf_plot(x, do.points = FALSE) # good for many sample points
edf_plot(x) # the default
edf_plot(x, log = "x") # logarithmic
(edf_plot(x, log = "x")) # ... with return value
## Note: flat part below first jump cannot be shown in log-scale

## Example 2
data(fire)
u <- 10 # threshold
exceed <- fire[fire > u] # exceedances
excess <- exceed - u # excesses
fit <- fit_GPD_MLE(excess) # fit GPD to excesses

## Plot empirical distribution function of excesses
## (partially with fitted GPD overlaid)
edf_plot(excess)
res <- edf_plot(excess, log = "x")
z <- tail(res$t, n = -1)
lines(z, pGPD(z, shape = fit$par["shape"], scale = fit$par["scale"]))

## Plot empirical distribution function of exceedances
## (partially with fitted GPD overlaid)
edf_plot(exceed)
res <- edf_plot(exceed, log = "x")
z <- tail(res$t, n = -1)
lines(z, pGPD(z-u, shape = fit$par["shape"], scale = fit$par["scale"]))

## Note: Q-Q plots are typically more meaningful
qf <- function(p) # quantile function of df
    qGPD(p, shape = fit$par["shape"], scale = fit$par["scale"])
qq_plot(excess, FUN = qf)
qq_plot(exceed, FUN = function(p) u + qf(p))
```

`fire` *Danish Fire Insurance Claims*

Description

Fire insurance claims in 1M DKK in Denmark from 1980-01-03 to 1990-12-31.

Usage

```
data("fire")
```

Format

xts object containing 2167 observations.

Author(s)

Marius Hofert

Source

originally Mette Rytgaard (Copenhagen Re); see **evir** and **QRM**.

Examples

```
library(xts)
data("fire")
str(fire)
stopifnot(inherits(fire, "xts"), length(fire) == 2167)
plot.zoo(fire, ylab = "Fire insurance claim")
```

---

fit_GARCH_11 *Fast(er) and Numerically More Robust Fitting of GARCH(1,1) Processes*

Description

Fast(er) and numerically more robust fitting of GARCH(1,1) processes according to Zumbach (2000).

Usage

```
fit_GARCH_11(x, init = NULL, sig2 = mean(x^2), delta = 1,
             distr = c("norm", "st"), control = list(), ...)
```

Arguments

x	vector of length n containing the data (typically log-returns) to be fitted a GARCH(1,1) to.
init	vector of length 2 giving the initial values for the likelihood fitting. Note that these are initial values for z_{corr} and z_{ema} as in Zumbach (2000).
sig2	annualized variance (third parameter of the reparameterization according to Zumbach (2000)).
delta	unit of time (defaults to 1 meaning daily data; for yearly data, use 250).
distr	character string specifying the innovation distribution ("norm" for N(0,1) or "st" for a standardized t distribution).
control	see ?optim().
...	additional arguments passed to the underlying optim().

Value

A list with components

coef estimated coefficients α_0, α_1, β_1 and, if `distr == "st"` the estimated degrees of freedom.

logLik maximized log-likelihood.

counts number of calls to the objective function; see ?optim.

convergence convergence code ('0' indicates successful completion; see ?optim.

message see ?optim.

sig.t vector of length n giving the conditional volatility.

Z.t vector of length n giving the standardized residuals.

Author(s)

Marius Hofert (based on an early version by Marcel Braeutigam)

References

Zumbach, G. (2000). The pitfalls in fitting GARCH (1,1) processes. *Advances in Quantitative Asset Management* **1**, 179–200.

See Also

fit_ARMA_GARCH() based on **rugarch**.

Examples

```
### Example 1: N(0,1) innovations ###############################################

## Generate data from a GARCH(1,1) with N(0,1) innovations
library(rugarch)
uspec <- ugarchspec(variance.model = list(model = "sGARCH",
                                          garchOrder = c(1, 1)),
                    distribution.model = "norm",
                    mean.model = list(armaOrder = c(0, 0)),
                    fixed.pars = list(mu = 0,
                                      omega = 0.1, # alpha_0
                                      alpha1 = 0.2, # alpha_1
                                      beta1 = 0.3)) # beta_1
X <- ugarchpath(uspec, n.sim = 1e4, rseed = 271) # sample (set.seed() fails!)
X.t <- as.numeric(X@path$seriesSim) # actual path (X_t)

## Fitting via ugarchfit()
uspec. <- ugarchspec(variance.model = list(model = "sGARCH",
                                           garchOrder = c(1, 1)),
                     distribution.model = "norm",
                     mean.model = list(armaOrder = c(0, 0)))
fit <- ugarchfit(uspec., data = X.t)
coef(fit) # fitted mu, alpha_0, alpha_1, beta_1
Z <- fit@fit$z # standardized residuals
```

```
stopifnot(all.equal(mean(Z), 0, tol = 1e-2),
          all.equal(var(Z),  1, tol = 1e-3))

## Fitting via fit_GARCH_11()
fit. <- fit_GARCH_11(X.t)
fit.$coef # fitted alpha_0, alpha_1, beta_1
Z. <- fit.$Z.t # standardized residuals
stopifnot(all.equal(mean(Z.), 0, tol = 5e-3),
          all.equal(var(Z.),  1, tol = 1e-3))

## Compare
stopifnot(all.equal(fit.$coef, coef(fit)[c("omega", "alpha1", "beta1")],
                    tol = 5e-3, check.attributes = FALSE)) # fitted coefficients
summary(Z. - Z) # standardized residuals


### Example 2: t_nu(0, (nu-2)/nu) innovations ##################################

## Generate data from a GARCH(1,1) with t_nu(0, (nu-2)/nu) innovations
uspec <- ugarchspec(variance.model = list(model = "sGARCH",
                                          garchOrder = c(1, 1)),
                    distribution.model = "std",
                    mean.model = list(armaOrder = c(0, 0)),
                    fixed.pars = list(mu = 0,
                                      omega = 0.1, # alpha_0
                                      alpha1 = 0.2, # alpha_1
                                      beta1 = 0.3, # beta_1
                                      shape = 4)) # nu
X <- ugarchpath(uspec, n.sim = 1e4, rseed = 271) # sample (set.seed() fails!)
X.t <- as.numeric(X@path$seriesSim) # actual path (X_t)

## Fitting via ugarchfit()
uspec. <- ugarchspec(variance.model = list(model = "sGARCH",
                                           garchOrder = c(1, 1)),
                     distribution.model = "std",
                     mean.model = list(armaOrder = c(0, 0)))
fit <- ugarchfit(uspec., data = X.t)
coef(fit) # fitted mu, alpha_0, alpha_1, beta_1, nu
Z <- fit@fit$z # standardized residuals
stopifnot(all.equal(mean(Z), 0, tol = 1e-2),
          all.equal(var(Z),  1, tol = 5e-2))

## Fitting via fit_GARCH_11()
fit. <- fit_GARCH_11(X.t, distr = "st")
c(fit.$coef, fit.$df) # fitted alpha_0, alpha_1, beta_1, nu
Z. <- fit.$Z.t # standardized residuals
stopifnot(all.equal(mean(Z.), 0, tol = 2e-2),
          all.equal(var(Z.),  1, tol = 2e-2))

## Compare
fit.coef <- coef(fit)[c("omega", "alpha1", "beta1", "shape")]
fit..coef <- c(fit.$coef, fit.$df)
stopifnot(all.equal(fit.coef, fit..coef, tol = 7e-2, check.attributes = FALSE))
```

```
summary(Z. - Z) # standardized residuals
```

---

fit_GEV *Parameter Estimators of the Generalized Extreme Value Distribution*

Description

Quantile matching estimator, probability weighted moments estimator, log-likelihood and maximum-likelihood estimator for the parameters of the generalized extreme value distribution (GEV).

Usage

```
fit_GEV_quantile(x, p = c(0.25, 0.5, 0.75), cutoff = 3)
fit_GEV_PWM(x)

logLik_GEV(param, x)
fit_GEV_MLE(x, init = c("shape0", "PWM", "quantile"),
            estimate.cov = TRUE, control = list(), ...)
```

Arguments

x	numeric vector of data. In the block maxima method, these are the block maxima.
p	numeric(3) specifying the probabilities whose quantiles are matched.
cutoff	positive z after which $\exp(-z)$ is truncated to 0.
param	numeric(3) containing the value of the shape ξ (a real), location μ (a real) and scale σ (positive real) parameters of the GEV distribution in this order.
init	character string specifying the method for computing initial values. Can also be numeric(3) for directly providing ξ, μ, σ.
estimate.cov	logical indicating whether the asymptotic covariance matrix of the parameter estimators is to be estimated (inverse of observed Fisher information (negative Hessian of log-likelihood evaluated at MLE)) and standard errors for the estimators of ξ, μ, σ returned, too.
control	list; passed to the underlying optim().
...	additional arguments passed to the underlying optim().

Details

fit_GEV_quantile() matches the empirical p-quantiles.

fit_GEV_PWM() computes the probability weighted moments (PWM) estimator of Hosking et al. (1985); see also Landwehr and Wallis (1979).

fit_GEV_MLE() uses, as default, the case $\xi = 0$ for computing initial values; this is actually a small positive value since Nelder–Mead could fail otherwise. For the other available methods for computing initial values, σ (obtained from the case $\xi = 0$) is doubled in order to guarantee a finite log-likelihood at the initial values. After several experiments (see the source code), one can safely

say that finding initial values for fitting GEVs via MLE is non-trivial; see also the block maxima method script about the Black Monday event on qrmtutorial.org.

Caution: See Coles (2001, p. 55) for how to interpret $\xi \leq -0.5$; in particular, the standard asymptotic properties of the MLE do not apply.

Value

`fit_GEV_quantile()` and `fit_GEV_PWM()` return a `numeric(3)` giving the parameter estimates for the GEV distribution.

`logLik_GEV()` computes the log-likelihood of the GEV distribution (`-Inf` if not admissible).

`fit_GEV_MLE()` returns the return object of `optim()` and, appended, the estimated asymptotic covariance matrix and standard errors of the parameter estimators, if `estimate.cov`.

Author(s)

Marius Hofert

References

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

Hosking, J. R. M., Wallis, J. R. and Wood, E. F. (1985). Estimation of the Generalized Extreme-Value Distribution by the Method of Probability-Weighted Moments. *Technometrics* **27**(3), 251–261.

Landwehr, J. M. and Wallis, J. R. (1979). Probability Weighted Moments Compared With Some Traditional Techniques in Estimating Gumbel Parameters and Quantiles. *Water Resourches Research* **15**(5), 1055–1064.

Coles, S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer-Verlag.

Examples

```
## Simulate some data
xi <- 0.5
mu <- -2
sig <- 3
n <- 1000
set.seed(271)
X <- rGEV(n, shape = xi, loc = mu, scale = sig)

## Fitting via matching quantiles
(fit.q <- fit_GEV_quantile(X))
stopifnot(all.equal(fit.q["shape"], xi,  tol = 0.12),
          all.equal(fit.q["loc"],   mu,  tol = 0.12),
          all.equal(fit.q["scale"], sig, tol = 0.005))

## Fitting via PWMs
(fit.PWM <- fit_GEV_PWM(X))
stopifnot(all.equal(fit.PWM["shape"], xi,  tol = 0.16),
          all.equal(fit.PWM["loc"],   mu,  tol = 0.15),
```

```
          all.equal(fit.PWM[["scale"]], sig, tol = 0.08))

## Fitting via MLE
(fit.MLE <- fit_GEV_MLE(X))
(est <- fit.MLE$par) # estimates of xi, mu, sigma
stopifnot(all.equal(est[["shape"]], xi,  tol = 0.07),
          all.equal(est[["loc"]],   mu,  tol = 0.12),
          all.equal(est[["scale"]], sig, tol = 0.06))
fit.MLE$SE # estimated asymp. variances of MLEs = std. errors of MLEs

## Plot the log-likelihood in the shape parameter xi for fixed
## location mu and scale sigma (fixed as generated)
xi. <- seq(-0.1, 0.8, length.out = 65)
logLik <- sapply(xi., function(xi..) logLik_GEV(c(xi.., mu, sig), x = X))
plot(xi., logLik, type = "l", xlab = expression(xi),
     ylab = expression("GEV distribution log-likelihood for fixed"~mu~"and"~sigma))
## => Numerically quite challenging (for this seed!)

## Plot the profile likelihood for these xi's
## Note: As initial values for the nuisance parameters mu, sigma, we
##       use their values in the case xi = 0 (for all fixed xi = xi.,
##       in particular those xi != 0). Furthermore, for the given data X
##       and xi = xi., we make sure the initial value for sigma is so large
##       that the density is not 0 and thus the log-likelihood is finite.
pLL <- sapply(xi., function(xi..) {
    scale.init <- sqrt(6 * var(X)) / pi
    loc.init <- mean(X) - scale.init * 0.5772157
    while(!is.finite(logLik_GEV(c(xi.., loc.init, scale.init), x = X)) &&
          is.finite(scale.init)) scale.init <- scale.init * 2
    optim(c(loc.init, scale.init), fn = function(nuis)
                logLik_GEV(c(xi.., nuis), x = X),
            control = list(fnscale = -1))$value
})
plot(xi., pLL, type = "l", xlab = expression(xi),
     ylab = "GEV distribution profile log-likelihood")
```

`fit_GPD` *Parameter Estimators of the Generalized Pareto Distribution*

Description

Method-of-moments estimator, probability weighted moments estimator, log-likelihood and maximum-likelihood estimator for the parameters of the generalized Pareto distribution (GPD).

Usage

```
fit_GPD_MOM(x)
fit_GPD_PWM(x)

logLik_GPD(param, x)
```

```
fit_GPD_MLE(x, init = c("PWM", "MOM", "shape0"),
            estimate.cov = TRUE, control = list(), ...)
```

Arguments

x	numeric vector of data. In the peaks-over-threshold method, these are the excesses (exceedances minus threshold).
param	numeric(2) containing the value of the shape ξ (a real) and scale β (positive real) parameters of the GPD in this order.
init	character string specifying the method for computing initial values. Can also be numeric(2) for directly providing ξ and β.
estimate.cov	logical indicating whether the asymptotic covariance matrix of the parameter estimators is to be estimated (inverse of observed Fisher information (negative Hessian of log-likelihood evaluated at MLE)) and standard errors for the estimators of ξ and β returned, too.
control	list; passed to the underlying optim().
...	additional arguments passed to the underlying optim().

Details

fit_GPD_MOM() computes the method-of-moments (MOM) estimator.

fit_GPD_PWM() computes the probability weighted moments (PWM) estimator of Hosking and Wallis (1987); see also Landwehr et al. (1979).

fit_GPD_MLE() uses, as default, fit_GPD_PWM() for computing initial values. The former requires the data x to be non-negative and adjusts β if ξ is negative, so that the log-likelihood at the initial value should be finite.

Value

fit_GEV_MOM() and fit_GEV_PWM() return a numeric(3) giving the parameter estimates for the GPD.

logLik_GPD() computes the log-likelihood of the GPD (-Inf if not admissible).

fit_GPD_MLE() returns the return object of optim() and, appended, the estimated asymptotic covariance matrix and standard errors of the parameter estimators, if estimate.cov.

Author(s)

Marius Hofert

References

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

Hosking, J. R. M. and Wallis, J. R. (1987). Parameter and Quantile Estimation for the Generalized Pareto Distribution. *Technometrics* **29**(3), 339–349.

Landwehr, J. M., Matalas, N. C. and Wallis, J. R. (1979). Estimation of Parameters and Quantiles of Wakeby Distributions. *Water Resourches Research* **15**(6), 1361–1379.

Examples

```
## Simulate some data
xi <- 0.5
beta <- 3
n <- 1000
set.seed(271)
X <- rGPD(n, shape = xi, scale = beta)

## Fitting via matching moments
(fit.MOM <- fit_GPD_MOM(X))
stopifnot(all.equal(fit.MOM[["shape"]], xi,   tol = 0.52),
          all.equal(fit.MOM[["scale"]], beta, tol = 0.24))

## Fitting via PWMs
(fit.PWM <- fit_GPD_PWM(X))
stopifnot(all.equal(fit.PWM[["shape"]], xi,   tol = 0.2),
          all.equal(fit.PWM[["scale"]], beta, tol = 0.12))

## Fitting via MLE
(fit.MLE <- fit_GPD_MLE(X))
(est <- fit.MLE$par) # estimates of xi, mu, sigma
stopifnot(all.equal(est[["shape"]], xi,   tol = 0.12),
          all.equal(est[["scale"]], beta, tol = 0.11))
fit.MLE$SE # estimated asymp. variances of MLEs = std. errors of MLEs

## Plot the log-likelihood in the shape parameter xi for fixed
## scale beta (fixed as generated)
xi. <- seq(-0.1, 0.8, length.out = 65)
logLik <- sapply(xi., function(xi..) logLik_GPD(c(xi.., beta), x = X))
plot(xi., logLik, type = "l", xlab = expression(xi),
     ylab = expression("GPD log-likelihood for fixed"~beta))

## Plot the profile likelihood for these xi's
## (with an initial interval for the nuisance parameter beta such that
##  logLik_GPD() is finite)
pLL <- sapply(xi., function(xi..) {
    ## Choose beta interval for optimize()
    int <- if(xi.. >= 0) {
               ## Method-of-Moment estimator
               mu.hat <- mean(X)
               sig2.hat <- var(X)
               shape.hat <- (1-mu.hat^2/sig2.hat)/2
               scale.hat <- mu.hat*(1-shape.hat)
               ## log-likelihood always fine for xi.. >= 0 for all beta
               c(1e-8, 2 * scale.hat)
           } else { # xi.. < 0
               ## Make sure logLik_GPD() is finite at endpoints of int
               mx <- max(X)
               -xi.. * mx * c(1.01, 100) # -xi * max(X) * scaling
               ## Note: for shapes xi.. closer to 0, the upper scaling factor
               ##       needs to be chosen sufficiently large in order
               ##       for optimize() to find an optimum (not just the
```

```
                ##       upper end point). Try it with '2' instead of '100'.
            }
    ## Optimization
    optimize(function(nuis) logLik_GPD(c(xi.., nuis), x = X),
             interval = int, maximum = TRUE)$maximum
})
plot(xi., pLL, type = "l", xlab = expression(xi),
     ylab = "GPD profile log-likelihood")
```

---

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 **the identity** if the data could not be retrieved (and is thus replaced by NA); **the given** FUN if FUN has been provided; **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 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, shape, loc = 0, scale = 1, log = FALSE)
pGEV(q, shape, loc = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qGEV(p, shape, loc = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rGEV(n, shape, loc = 0, scale = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations.
shape	GEV shape parameter ξ, a real.
loc	GEV location parameter μ, a real.

`scale`	GEV scale parameter σ, a positive real.
`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, shape = 0.5), shape = 0.5)) # should be U[0,1]
curve(dGEV(x, shape = 0.5), from = -3, to = 5)
```

GEV_shape_plot *Fitted GEV Shape as a Function of the Threshold*

Description

Fit GEVs to block maxima and plot the fitted GPD shape as a function of the block size.

Usage

```
GEV_shape_plot(x, blocksize = tail(pretty(seq_len(length(x)/20), n = 64), -1),
               estimate.cov = TRUE, conf.level = 0.95,
               lines.args = list(lty = 2), xlab = "Block size",  ylab = NULL,
               xlab2 = "Number of blocks", plot = TRUE, ...)
```

Arguments

x	numeric vector of data.
blocksize	numeric vector of block sizes for which to fit a GEV to the block maxima.
estimate.cov	logical indicating whether confidence intervals are to be computed.
conf.level	confidence level of the confidence intervals if estimate.cov.
lines.args	list of arguments passed to the underlying lines() for drawing the confidence intervals.
xlab	x-axis label.
ylab	y-axis label (if NULL, a default is used).
xlab2	label of the secondary x-axis.
plot	logical indicating whether a plot is produced.
...	additional arguments passed to the underlying plot().

Details

Such plots can be used in the block maxima method for determining the optimal block size (as the smallest after which the plot is (roughly) stable).

Value

Invisibly returns a list containing the block sizes considered, the corresponding block maxima and the fitted GEV distribution objects as returned by the underlying fit_GEV_MLE().

Author(s)

Marius Hofert

Examples

```
set.seed(271)
X <- rPar(5e4, shape = 4)
GEV_shape_plot(X)
abline(h = 1/4, lty = 3) # theoretical xi = 1/shape for Pareto
```

GPD *(Generalized) Pareto Distribution*

Description

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

Usage

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

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

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations.
shape	GPD shape parameter ξ (a real number) and Pareto shape parameter θ (a positive number).
scale	GPD scale parameter β (a positive number) and Pareto scale parameter κ (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

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

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
curve(dGPD(x, shape = 0.5, scale = 3), from = -1, to = 5)
plot(pGPD(rGPD(1000, shape = 0.5, scale = 3), shape = 0.5, scale = 3)) # should be U[0,1]
```

GPDtail *GPD-Based Tail Distribution (POT method)*

Description

Density, distribution function, quantile function and random variate generation for the GPD-based tail distribution in the POT method.

Usage

```
dGPDtail(x, threshold, p.exceed, shape, scale, log = FALSE)
pGPDtail(q, threshold, p.exceed, shape, scale, lower.tail = TRUE, log.p = FALSE)
qGPDtail(p, threshold, p.exceed, shape, scale, lower.tail = TRUE, log.p = FALSE)
rGPDtail(n, threshold, p.exceed, shape, scale)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations.
`threshold`	threshold u in the POT method.
`p.exceed`	probability of exceeding the threshold u; for the Smith estimator, this is `mean(x > threshold)` for `x` being the data.
`shape`	GPD shape parameter ξ (a real number).
`scale`	GPD scale parameter β (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

Let u denote the threshold (`threshold`), p_u the exceedance probability (`p.exceed`) and F_{GPD} the GPD distribution function. Then the distribution function of the GPD-based tail distribution is given by

$$F(q) = 1 - p_u(1 - F_{GPD}(q - u))$$

. The quantile function is

$$F^{-1}(p) = u + F_GPD^{-1}(1 - (1 - p)/p_u)$$

and the density is

$$f(x) = p_u f_{GPD}(x - u)$$

, where f_{GPD} denotes the GPD density.

Note that the distribution function has a jumpt of height $P(X \leq u)$ (`1-p.exceed`) at u.

Value

`dGPDtail()` computes the density, `pGPDtail()` the distribution function, `qGPDtail()` the quantile function and `rGPDtail()` random variates of the GPD-based tail distribution in the POT method.

Author(s)

Marius Hofert

References

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

Examples

```
## Data we work with
data(fire) # load the Danish fire insurance data
X <- fire

## Determine thresholds for POT method
mean_excess_plot(X[X > 0])
abline(v = 10)
u <- 10 # threshold

## Fit GPD to the excesses (per margin)
fit <- fit_GPD_MLE(X[X > u] - u)
fit$par

## Estimate threshold exceedance probabilities
p.exceed <- mean(X > u)

## Define corresponding densities, distribution function and RNG
dF <- function(x) dGPDtail(x, threshold = u, p.exceed = p.exceed,
                           shape = fit$par["shape"], scale = fit$par["scale"])
pF <- function(q) pGPDtail(q, threshold = u, p.exceed = p.exceed,
```

```
                            shape = fit$par["shape"], scale = fit$par["scale"])
rF <- function(n) rGPDtail(n, threshold = u, p.exceed = p.exceed,
                            shape = fit$par["shape"], scale = fit$par["scale"])

## Basic check of dF()
curve(dF, from = 8, to = 20)
abline(v = u, lty = 2) # threshold

## Basic check of pF()
curve(pF, from = u, to = 20, ylim = 0:1) # quite flat here
abline(v = u, h = 1-p.exceed, lty = 2) # mass at u is 1-p.exceed (see 'Details')

## Basic check of rF()
set.seed(271)
X. <- rF(1000)
plot(X.)
stopifnot(all.equal(mean(X. == u), 1-p.exceed, tol = 5e-3)) # confirms the above
## Pick out 'continuous part'
X.. <- X.[X. > u]
plot(pF(X..)) # should be U[1-p.exceed, 1]
```

GPD_shape_plot *Fitted GPD Shape as a Function of the Threshold*

Description

Fit GPDs to various thresholds and plot the fitted GPD shape as a function of the threshold.

Usage

```
GPD_shape_plot(x, thresholds = seq(quantile(x, 0.5), quantile(x, 0.99),
                                   length.out = 65),
               estimate.cov = TRUE, conf.level = 0.95,
               lines.args = list(lty = 2), xlab = "Threshold", ylab = NULL,
               xlab2 = "Excesses", plot = TRUE, ...)
```

Arguments

x	numeric vector of data.
thresholds	numeric vector of thresholds for which to fit a GPD to the excesses.
estimate.cov	logical indicating whether confidence intervals are to be computed.
conf.level	confidence level of the confidence intervals if estimate.cov.
lines.args	list of arguments passed to the underlying lines() for drawing the confidence intervals.
xlab	x-axis label.
ylab	y-axis label (if NULL, a default is used).
xlab2	label of the secondary x-axis.

plot	logical indicating whether a plot is produced.
...	additional arguments passed to the underlying plot().

Details

Such plots can be used in the peaks-over-threshold method for determining the optimal threshold (as the smallest after which the plot is (roughly) stable).

Value

Invisibly returns a list containing the thresholds considered, the corresponding excesses and the fitted GPD objects as returned by the underlying fit_GPD_MLE().

Author(s)

Marius Hofert

Examples

```
data(fire)
GPD_shape_plot(fire)
abline(v = c(10, 20), lty = 3) # possible threshold choices from mean_excess_plot()
```

---

hierarchical_matrix *Construction of Hierarchical Matrices*

Description

Constructing hierarchical matrices, used, for example, for hierarchical dependence models, clustering, etc.

Usage

```
hierarchical_matrix(x, diagonal = rep(1, d))
```

Arguments

x	list of length 2 or 3 containing the homogeneous numeric entry of the current block of the hierarchical matrix, the integer components belongning to the current block (or NULL) and, possibly, another (nested) list of the same type.
diagonal	diagonal elements of the hierarchical matrix.

Details

See the examples for how to use.

Value

A hierarchical matrix of the structure as specified in x with off-diagonal entries as specified in x and diagonal entries as specified in diagonal.

Author(s)

Marius Hofert

Examples

```
rho <- c(0.2, 0.3, 0.5, 0.8) # some entries (e.g., correlations)

## Test homogeneous case
x <- list(rho[1], 1:6)
hierarchical_matrix(x)

## Two-level case with one block of size 2
x <- list(rho[1], 1, list(rho[2], 2:3))
hierarchical_matrix(x)

## Two-level case with one block of size 2 and a larger homogeneous block
x <- list(rho[1], 1:3, list(rho[2], 4:5))
hierarchical_matrix(x)

## Test two-level case with three blocks of size 2
x <- list(rho[1], NULL, list(list(rho[2], 1:2),
                             list(rho[3], 3:4),
                             list(rho[4], 5:6)))
hierarchical_matrix(x)

## Test three-level case
x <- list(rho[1], 1:3, list(rho[2], NULL, list(list(rho[3], 4:5),
                                               list(rho[4], 6:8))))
hierarchical_matrix(x)

## Test another three-level case
x <- list(rho[1], c(3, 6, 1), list(rho[2], c(9, 2, 7, 5),
                                   list(rho[3], c(8, 4))))
hierarchical_matrix(x)
```

---

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 `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)) # random Cholesky factor
Sigma <- L
P <- cor(Sigma)

## Default
matrix_plot(P)
matrix_plot(abs(P)) # if nonnegative
L. <- L
diag(L.) <- NA
matrix_plot(L.) # Cholesky factor without diagonal

## 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 '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)
```

`mean_excess` *Mean Excess*

Description

Sample mean excess function, mean excess function of a GPD and sample mean excess plot.

Usage

```
mean_excess_np(x, omit = 3)
mean_excess_plot(x, omit = 3,
                 xlab = "Threshold", ylab = "Mean excess over threshold", ...)
mean_excess_GPD(x, shape, scale)
```

Arguments

x	`mean_excess_GPD()` `numeric` vector of evaluation points of the mean excess function of the GPD. **otherwise** `numeric` vector of data.
omit	number ≥ 1 of unique last observations to be omitted from the sorted data (as mean excess plot becomes unreliable for these observations as thresholds).
xlab	x-axis label.
ylab	y-axis label.
...	additional arguments passed to the underlying `plot()`.
shape	GPD shape parameter ξ.
scale	GPD scale parameter β.

Details

Mean excess plots can be used in the peaks-over-threshold method for choosing a threshold. To this end, one chooses the smallest threshold above which the mean excess plot is roughly linear.

Value

`mean_excess_np()` returns a two-column matrix giving the sorted data without the `omit`-largest unique values (first column) and the corresponding values of the sample mean excess function (second column). It is mainly used in `mean_excess_plot()`.

`mean_excess_plot()` returns `invisible()`.

`mean_excess_GPD()` returns the mean excess function of a generalized Pareto distribution evaluated at `x`.

Author(s)

Marius Hofert

Examples

```
## (Sample) mean excess function
data(fire)
ME <- mean_excess_np(fire)
stopifnot(dim(ME) == c(2164, 2),
          all.equal(ME[nrow(ME),], c(65.707491, 121.066231),
                    check.attributes = FALSE))

## A 'manual' (sample) mean excess plot
plot(ME, xlab = "Threshold", ylab = "Mean excess over threshold")

## (Sample) mean excess plot
mean_excess_plot(fire)
## => Any value in [10, 20] seems reasonable here as threshold choice
##    (one would probably go with 10 to benefit from a larger sample size).
```

```
## With mean excess functions of two fitted GPDs overlaid
u <- c(10, 20) # thresholds
fit <- lapply(u, function(u.) fit_GPD_MLE(fire[fire > u.] - u.))
q <- lapply(u, function(u.) seq(u., ME[nrow(ME),"x"], length.out = 129))
MEF.GPD <- lapply(1:2, function(k)
    mean_excess_GPD(q[[k]]-u[k], shape = fit[[k]]$par["shape"]],
                    scale = fit[[k]]$par["scale"]]))
mean_excess_plot(fire, ylim = range(ME, unlist(MEF.GPD)))
col <- c("royalblue3", "maroon3")
for(k in 1:2) lines(q[[k]], MEF.GPD[[k]], col = col[k])
legend("bottomright", col = rev(col), lty = rep(1, length(u)), bty = "n",
       legend = as.expression(sapply(rev(seq_along(u)),
       function(k) substitute("Threshold choice"~~u==u., list(u. = u[k])))))
```

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 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 pp_plot(): The distribution function. 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 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)
returns_qrmtools(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.

Note that the R package **timeSeries** also contains a function `returns()` (and hence the order in which **timeSeries** and **qrmtools** are loaded matters to get the right `returns()`). For this reason, `returns_qrmtools()` is an alias for `returns()` from **qrmtools**.

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, level, names = FALSE, type = 1, ...)
```

```
VaR_t(level, loc = 0, scale = 1, df = Inf)
VaR_GPD(level, shape, scale)
VaR_Par(level, shape, scale = 1)
VaR_GPDtail(level, threshold, p.exceed, shape, scale)

## Expected shortfall
ES_np(x, level, method = c(">", ">="), verbose = FALSE, ...)
ES_t(level, loc = 0, scale = 1, df = Inf)
ES_GPD(level, shape, scale)
ES_Par(level, shape, scale = 1)
ES_GPDtail(level, threshold, p.exceed, shape, scale)

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

Arguments

x	gVaR(), gEX() matrix of (rowwise) multivariate losses. **otherwise** vector of losses.
level	gVaR(), gEX() vector or matrix of (rowwise) confidence levels α (all in $[0, 1]$). **otherwise** confidence level $\alpha \in [0, 1]$.
names	see ?quantile.
type	see ?quantile.
loc	location parameter μ.
shape	VaR_GPD(), ES_GPD() GPD shape parameter ξ, a real number. VaR_Par(), ES_Par() Pareto shape parameter θ, a positive number.
scale	VaR_t(), ES_t() t scale parameter σ, a positive number. VaR_GPD(), ES_GPD() GPD scale parameter β, a positive number. VaR_Par(), ES_Par() Pareto scale parameter κ, a positive number.
df	degrees of freedom, a positive number; choose df = Inf for the normal distribution.
threshold	threhold u (used to estimate the exceedance probability based on the data x).
p.exceed	exceedance probability; typically mean(x > threshold) for x being the data modeled with the peaks-over-threshold (POT) method.
start	vector of initial values for the underlying optim().
method	ES_np() character string indicating the method for computing expected shortfall. 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(). ES_np() additional arguments passed to VaR_np(). 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_GPD(), ES_GPD() compute value-at-risk and expected shortfall for the generalized Pareto distribution (GPD).

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. (2018), 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. (2018). Multivariate geometric expectiles. *Scandinavian Actuarial Journal*, 2018(7), 629–659.

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, level = 0.99)
ES_np(L,  level = 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, scale = V*sig, df = Inf)
VaRt4 <- VaR_t(alpha, scale = V*sig*sqrt((nu-2)/nu), df = nu)
ESnorm <- ES_t(alpha, scale = V*sig, df = Inf)
ESt4 <- ES_t(alpha, scale = 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], shape = 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, level = 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, level = 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)
```

`tail_plot` *Plot of a Non-Parametric Tail Estimator*

Description

Plot a non-parametric tail distribution, possibly overlaid with the Smith estimator.

Usage

```
tail_plot(x, threshold, shape = NULL, scale = NULL,
          q = NULL, length.out = 129, lines.args = list(),
          log = "xy", xlim = NULL, ylim = NULL,
          xlab = "x", ylab = "Tail probability at x", ...)
```

Arguments

`x`	`numeric` vector of data.
`threshold`	`numeric(1)` giving the threshold u above which the tail (starts and) is to be plotted.
`shape`	`NULL` or the GPD shape parameter ξ (typically obtained via `fit_GPD_MLE()`).
`scale`	`NULL` or the GPD shape parameter β (typically obtained via `fit_GPD_MLE()`).

q	NULL, numeric(1) or numeric vector of evaluationn points of the Smith estimator (semi-parametric GPD-based tail estimator in the POT method). If NULL, the evaluation points are determined internally as an equidistant sequence of length length.out between the smallest and largest exceedance (taken equidistant in log-scale if log contains "x"). If numeric(1), then the behavior is similar to NULL with the exception that the plot is extended to the right of the largest exceedance if q is larger than the largest exceedance.
length.out	length of q.
lines.args	list of arguments passed to the underlying lines().
log	character indicating whether logarithmic axes are to be used.
xlim	x-axis limits.
ylim	y-axis limits.
xlab	x-axis label.
ylab	y-axis label.
...	additional arguments passed to the underlying plot().

Value

If both shape and scale are provided, tail_plot() overlays the non-parametric tail estimator (evaluated at the exceedances) with the corresponding GPD. In this case, tail_plot() invisibly returns a list with two two-column matrices, once containing the x-values and y-values of the non-parametric tail estimator and once containing the x-values and y-values of the Smith estimator. If shape or scale are NULL, tail_plot() invisibly returns a two-column matrix with the x-values and y-values of the non-parametric tail estimator.

Author(s)

Marius Hofert

Examples

```
data(fire)
u <- 10 # threshold choice
tail_plot(fire, threshold = u, log = "", type = "b") # => need log-scale
tail_plot(fire, threshold = u, type = "s") # as a step function
fit <- fit_GPD_MLE(fire[fire > u] - u) # fit GPD to excesses (POT method)
tail_plot(fire, threshold = u, # without log-scale
          shape = fit$par["shape"], scale = fit$par["scale"], log = "")
tail_plot(fire, threshold = u, # highlights linearity
          shape = fit$par["shape"], scale = fit$par["scale"])
```

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(level, qF, d = NULL, ...)
VaR_bounds_hom(level, 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(level, 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(level, 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(level, 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

level	confidence level α for VaR and ES (e.g., 0.99).

d — dimension (number of risk factors; ≥ 2). For `crude_VaR_bounds()`, `d` only needs to be given in the homogeneous case in which `qF` is a `function`.

qF — d-list containing the marginal quantile functions. For `crude_VaR_bounds()`, `qF` can also be function (homogeneous case, dimension `d`).

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 ϵ 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 ≥ 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 shape (the shape 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 α must be provided as argument level.

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 α for any d-dimensional model with marginal quantile functions specified by `qF`.

`VaR_bounds_hom()` returns the best and worst VaR at confidence level α 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. and Beleraj, A. (2014). An Academic Response to Basel 3.5. *Risks* **2**(1), 25–48.

Embrechts, P., Puccetti, G. and 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. and 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. and 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, shape = 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, shape = theta) # Pareto quantile function
pF <- function(q, theta = th) pPar(q, shape = 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", shape = 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, shape = xi[j], scale = 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 = 6e-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 = 2e-3),
          all.equal(ara$bound[["up"]],  abra$bound[["up"]],  tolerance = 6e-3))
```

Index