Package ‘HighFreq’

May 12, 2025

Type Package

Title High Frequency Time Series Management

Version 0.1

Date 2018-09-12

Author Jerzy Pawlowski (algoquant)

Maintainer Jerzy Pawlowski <jp3900@nyu.edu>

Description Functions for chaining and joining time series, scrubbing bad data, managing time zones and alligning time indices, converting TAQ data to OHLC format, aggregating data to lower frequency, estimating volatility, skew, and higher moments.

License MPL-2.0

Depends xts,
quantmod,
rutils

Imports xts,
quantmod,
rutils,
Rcpp

LinkingTo Rcpp, RcppArmadillo

SystemRequirements GNU make, C++11

Remotes github::algoquant/rutils,

VignetteBuilder knitr

LazyData true

ByteCompile true

Repository GitHub

URL https://github.com/algoquant/HighFreq

RoxygenNote 7.3.2

Encoding UTF-8

Contents

agg_ohlc *Aggregate a time series of data into a single bar of* OHLC *data.*

Description

Aggregate a time series of data into a single bar of *OHLC* data.

Usage

```
agg_ohlc(timeser)
```

Arguments

`timeser`	A *time series* or a *matrix* with multiple columns of data.

Details

The function `agg_ohlc()` aggregates a time series of data into a single bar of *OHLC* data. It can accept either a single column of data or four columns of *OHLC* data. It can also accept an additional column containing the trading volume.

The function `agg_ohlc()` calculates the *open* value as equal to the *open* value of the first row of `timeser`. The *high* value as the maximum of the *high* column of `timeser`. The *low* value as the minimum of the *low* column of `timeser`. The *close* value as the *close* of the last row of `timeser`. The *volume* value as the sum of the *volume* column of `timeser`.

For a single column of data, the *open*, *high*, *low*, and *close* values are all the same.

Value

A *matrix* containing a single row, with the *open*, *high*, *low*, and *close* values, and also the total *volume* (if provided as either the second or fifth column of `timeser`).

Examples

```
## Not run:
# Define matrix of OHLC data
ohlc <- coredata(rutils::etfenv$VTI[, 1:5])
# Aggregate to single row matrix
ohlcagg <- HighFreq::agg_ohlc(ohlc)
# Compare with calculation in R
all.equal(drop(ohlcagg),
  c(ohlc[1, 1], max(ohlc[, 2]), min(ohlc[, 3]), ohlc[NROW(ohlc), 4], sum(ohlc[, 5])),
  check.attributes=FALSE)

## End(Not run)  # end dontrun
```

agg_stats_r — *Calculate the aggregation (weighted average) of a statistical estimator over a* OHLC *time series using* R *code.*

Description

Calculate the aggregation (weighted average) of a statistical estimator over a *OHLC* time series using `R` code.

Usage

```
agg_stats_r(ohlc, calc_bars = "ohlc_variance", weighted = TRUE, ...)
```

Arguments

`...`	additional parameters to the function `calc_bars`.
`ohlc`	An *OHLC* time series of prices and trading volumes, in *xts* format.
`calc_bars`	A *character* string representing a function for calculating statistics for individual *OHLC* bars.
`weighted`	*Boolean* argument: should estimate be weighted by the trading volume? (default is `TRUE`)

Details

The function `agg_stats_r()` calculates a single number representing the volume weighted average of statistics of individual *OHLC* bars. It first calls the function `calc_bars` to calculate a vector of statistics for the *OHLC* bars. For example, the statistic may simply be the difference between the *High* minus *Low* prices. In this case the function `calc_bars` would calculate a vector of *High* minus *Low* prices. The function `agg_stats_r()` then calculates a trade volume weighted average of the vector of statistics.

The function `agg_stats_r()` is implemented in `R` code.

Value

A single *numeric* value equal to the volume weighted average of an estimator over the time series.

Examples

```
# Calculate weighted average variance for SPY (single number)
variance <- agg_stats_r(ohlc=HighFreq::SPY, calc_bars="ohlc_variance")
# Calculate time series of daily skew estimates for SPY
skew_daily <- apply.daily(x=HighFreq::SPY, FUN=agg_stats_r, calc_bars="ohlc_skew")
```

`calc_covar` *Calculate the covariance matrix of the columns of a* time series *using* `RcppArmadillo`.

Description

Calculate the covariance matrix of the columns of a *time series* using `RcppArmadillo`.

Usage

```
calc_covar(timeser, method = "moment", confl = 0.75)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`method`	A *character string* specifying the type of the covariance model (the default is `method = "moment"` - see Details).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `calc_covar()` calculates the covariance matrix of the columns of a *time series* or a *matrix* of data using `RcppArmadillo` `C++` code. The covariance is a measure of the codependency of the data.

If `method = "moment"` (the default) then `calc_covar()` calculates the covariance as the second co-moment:

$$\sigma_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Then `calc_covar()` performs the same calculation as the `R` function `stats::cov()`.

If `method = "quantile"` then it calculates the covariance as the difference between the quantiles as follows:

$$\mu = q_{\alpha} - q_{1-\alpha}$$

Where α is the confidence level for calculating the quantiles.

If `method = "nonparametric"` then it calculates the covariance as the Median Absolute Deviation (*MAD*):

$$MAD = median(abs(x - median(x)))$$

It also multiplies the *MAD* by a factor of `1.4826`, to make it comparable to the standard deviation.

If `method = "nonparametric"` then `calc_covar()` performs the same calculation as the function `stats::mad()`, but it's much faster because it uses `RcppArmadillo` `C++` code.

If the number of rows of `timeser` is less than `3` then it returns zeros.

Value

A square matrix with the covariance coefficients of the columns of the *time series* `timeser`.

Examples

```
## Not run:
# Calculate VTI and XLF returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "XLF")])
# Compare HighFreq::calc_covar() with standard var()
all.equal(drop(HighFreq::calc_covar(retp)),
  cov(retp), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with matrixStats and with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_covar(retp),
  Rcode=cov(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Compare HighFreq::calc_covar() with stats::mad()
all.equal(drop(HighFreq::calc_covar(retp, method="nonparametric")),
  sapply(retp, mad), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with stats::mad()
summary(microbenchmark(
  Rcpp=HighFreq::calc_covar(retp, method="nonparametric"),
  Rcode=sapply(retp, mad),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_cvar` *Calculate the Value at Risk (*VaR*) or the Conditional Value at Risk (*CVaR*) of an* xts time series *of returns, using* R *code.*

Description

Calculate the Value at Risk (*VaR*) or the Conditional Value at Risk (*CVaR*) of an *xts time series* of returns, using `R` code.

Usage

```
calc_cvar(timeser, method = "var", confi = pnorm(-2))
```

Arguments

`timeser`	An *xts time series* of returns with multiple columns.
`method`	A *string* specifying the type of risk measure (the default is `method = "var"` - see Details).
`confi`	The confidence level for calculating the quantile (the default is `confi = pnorm(-2) = 0.02275`).

Details

The function `calc_cvar()` calculates the Value at Risk (*VaR*) or the Conditional Value at Risk (*CVaR*) of an *xts time series* of returns, using `R`

The Value at Risk (*VaR*) and the Conditional Value at Risk (*CVaR*) are measures of the tail risk of returns.

If method = "var" then calc_cvar() calculates the Value at Risk (*VaR*) as the quantile of the returns as follows:

$$\alpha = \int_{-\infty}^{\mathrm{VaR}(\alpha)} \mathrm{f}(r) \, \mathrm{d}r$$

Where α is the confidence level for calculating the quantile, and $\mathrm{f}(r)$ is the probability density (distribution) of returns.

If method = "cvar" then calc_cvar() calculates the Value at Risk (*VaR*) as the Expected Tail Loss (*ETL*) of the returns as follows:

$$\mathrm{CVaR} = \frac{1}{\alpha} \int_{0}^{\alpha} \mathrm{VaR}(p) \, \mathrm{d}p$$

Where α is the confidence level for calculating the quantile.

Value

A vector with the risk measures of the columns of the input *time series* timeser.

Examples

```
## Not run:
# Calculate VTI and XLF returns
returns <- na.omit(rutils::etfenv$returns[, c("VTI", "XLF")])
# Calculate VaR
all.equal(HighFreq::calc_cvar(returns),
  sapply(returns, quantile, probs=pnorm(-2)), check.attributes=FALSE)
# Calculate CVaR
all.equal(HighFreq::calc_cvar(returns, method="cvar", confi=0.02),
  sapply(returns, function(x) mean(x[x < quantile(x, 0.02)])),
  check.attributes=FALSE)

## End(Not run)
```

calc_eigen — *Calculate the eigen decomposition of a square, symmetric matrix using* RcppArmadillo.

Description

Calculate the eigen decomposition of a square, symmetric matrix using RcppArmadillo.

Usage

```
calc_eigen(matrixv, eigenval, eigenvec)
```

Arguments

matrixv	A square, symmetric matrix.
eigenval	A *vector* of eigen values.
eigenvec	A *matrix* of eigen vectors.

Details

The function `calc_eigen()` calculates the eigen decomposition of a square, symmetric matrix using `RcppArmadillo`. It calls the `Armadillo` function `arma::eig_sym()` to calculate the eigen decomposition.

For small matrices, the function `calc_eigen()` is several times faster than the `R` function `eigen()`, since `calc_eigen()` has no overhead in `R` code. But for large matrices, they are about the same, since both call `C++` code.

Value

Void (no return value - passes the eigen values and eigen vectors by reference).

Examples

```
## Not run:
# Create random positive semi-definite matrix
matrixv <- matrix(runif(25), nc=5)
matrixv <- t(matrixv) %*% matrixv
# Calculate the eigen decomposition using RcppArmadillo
eigenval <- numeric(5) # Allocate eigen values
eigenvec <- matrix(numeric(25), nc=5) # Allocate eigen vectors
HighFreq::calc_eigen(matrixv, eigenval, eigenvec)
# Calculate the eigen decomposition using R
eigenr <- eigen(matrixv)
# Compare the eigen decompositions
all.equal(eigenr$values, drop(eigenval))
all.equal(abs(eigenr$vectors), abs(eigenvec))
# Compare the speed of Rcpp with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_eigen(matrixv, eigenval, eigenvec),
  Rcode=eigen(matrixv),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_eigenp` *Calculate the partial eigen decomposition of a dense symmetric matrix using* `RcppArmadillo`.

Description

Calculate the partial eigen decomposition of a dense symmetric matrix using `RcppArmadillo`.

Usage

```
calc_eigenp(matrixv, neigen)
```

Arguments

`matrixv`	A square matrix.
`neigen`	An *integer* equal to the number of eigenvalues to be calculated.

Details

The function `calc_eigenp()` calculates the partial eigen decomposition (the lowest order principal components, with the largest eigenvalues) of a dense matrix using RcppArmadillo. It calls the internal `Armadillo` eigen solver `SymEigsSolver` in the namespace `arma::newarp` to calculate the partial eigen decomposition.

The eigen solver `SymEigsSolver` uses the Implicitly Restarted Lanczos Method (IRLM) which was adapted from the ARPACK library. The eigen solver `SymEigsSolver` was implemented by Yixuan Qiu.

The function `arma::eigs_sym()` also calculates the partial eigen decomposition using the eigen solver `SymEigsSolver`, but it only works for sparse matrices which are not standard R matrices.

For matrices smaller than `100` rows, the function `calc_eigenp()` is slower than the function `calc_eigen()` which calculates the full eigen decomposition. But it's faster for very large matrices.

Value

A list with two elements: a *vector* of eigenvalues (named "values"), and a *matrix* of eigenvectors (named "vectors").

Examples

```
## Not run:
# Create random positive semi-definite matrix
matrixv <- matrix(runif(100), nc=10)
matrixv <- t(matrixv) %*% matrixv
# Calculate the partial eigen decomposition
neigen <- 5
eigenp <- HighFreq::calc_eigenp(matrixv, neigen)
# Calculate the eigen decomposition using RcppArmadillo
eigenval <- numeric(10) # Allocate eigen values
eigenvec <- matrix(numeric(100), nc=10) # Allocate eigen vectors
HighFreq::calc_eigen(matrixv, eigenval, eigenvec)
# Compare the eigen decompositions
all.equal(eigenp$values[1:neigen], eigenval[1:neigen])
all.equal(abs(eigenp$vectors), abs(eigenvec[, 1:neigen]))
# Compare the speed of partial versus full decomposition
summary(microbenchmark(
  partial=HighFreq::calc_eigenp(matrixv, neigen),
  full=HighFreq::calc_eigen(matrixv, eigenval, eigenvec),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_endpoints` *Calculate a vector of end points that divides an integer time sequence of time periods into equal time intervals.*

Description

Calculate a vector of end points that divides an integer time sequence of time periods into equal time intervals.

Usage

```
calc_endpoints(length, step = 1L, stub = 0L, stubs = TRUE)
```

Arguments

`length`	An *integer* equal to the length of the time sequence to be divided into equal intervals.
`step`	The number of time periods in each interval between neighboring end points (the default is `step = 1`).
`stub`	An *integer* equal to the first non-zero end point (the default is `stub = 0`).
`stubs`	A *Boolean* specifying whether to include stub intervals (the default is `stubs = TRUE`).

Details

The end points are a vector of integers which divide the sequence of time periods of length equal to `length` into equally spaced time intervals. The number of time periods between neighboring end points is equal to the argument `step`. If a whole number of intervals doesn't fit over the whole sequence, then `calc_endpoints()` adds a stub interval at the end. A stub interval is one where the number of periods between neighboring end points is less than the argument `step`.

If `stubs = TRUE` (the default) then the first end point is equal to `0` (since indexing in `C++` code starts at `0`). The first non-zero end point is equal to `step` or `stub` (if it's not zero). If `stub = 0` (the default) then the first end point is equal to `0` (even if `stubs = FALSE`). If `stubs = TRUE` (the default) then the last end point is always equal to `length-1`. The argument `stub` should be less than the `step`: `stub < step`.

If `step = 1` and `stub = 0` (the default), then the vector of end points is simply equal to:

$$\{0, 1, 2, ..., length - 1\}$$

If `stub = 0` (the default) and `stubs = TRUE` (the default) then the vector of end points is equal to:

$$\{0, step, 2 * step, ..., length - 1\}$$

If `stub = 0` (the default) and `stubs = FALSE` then the vector of end points is equal to:

$$\{0, step, 2 * step, ..., n * step\}$$

If `stub > 0` and `stubs = TRUE` (the default), then the vector of end points is equal to:

$$\{0, stub, stub + step, ..., length - 1\}$$

For example, the end points for `length = 20`, divided into intervals of `step = 5` are equal to: `0, 5, 10, 15, 19`.

If `stub = 1` then the first non-zero end point is equal to `1` and the end points are equal to: `0, 1, 6, 11, 16, 19`. The stub interval at the beginning is equal to `2` (including `0` and `1`). The stub interval at the end is equal to `3 = 19 - 16`.

The end points for `length = 21` divided into intervals of length `step = 5`, with `stub = 0`, are equal to: `0, 5, 10, 15, 20`. The beginning interval is equal to `5`. The end interval is equal to `5 = 20 - 15`.

If `stub = 1` then the first non-zero end point is equal to `1` and the end points are equal to: `0, 1, 6, 11, 16, 20`. The beginning stub interval is equal to `2`. The end stub interval is equal to `4 = 20 - 16`.

The function `calc_endpoints()` is similar to the function `rutils::calc_endpoints()` from package rutils.

But the end points are shifted by `-1` compared to `R` code because indexing starts at `0` in `C++` code, while it starts at `1` in `R` code. So if `calc_endpoints()` is used in `R` code then `1` should be added to it.

Value

A vector of equally spaced *integers* representing the end points.

Examples

```
# Calculate the end points without a stub interval
HighFreq::calc_endpoints(length=20, step=5)
# Calculate the end points with a final stub interval
HighFreq::calc_endpoints(length=23, step=5)
# Calculate the end points with initial and final stub intervals
HighFreq::calc_endpoints(length=20, step=5, stub=2)
```

`calc_hurst` *Calculate the Hurst exponent from the volatility ratio of aggregated returns.*

Description

Calculate the Hurst exponent from the volatility ratio of aggregated returns.

Usage

```
calc_hurst(timeser, aggv)
```

Arguments

`timeser`	A *time series* or a *matrix* of log prices.
`aggv`	A *vector* of aggregation intervals.

Details

The function `calc_hurst()` calculates the Hurst exponent from the ratios of the volatilities of aggregated returns.

An aggregation interval is equal to the number of time periods between the neighboring aggregation end points.

The aggregated volatility σ_t increases with the length of the aggregation interval Δt. The aggregated volatility increases as the length of the aggregation interval Δt raised to the power of the *Hurst exponent* H:

$$\sigma_t = \sigma \Delta t^H$$

Where σ is the daily return volatility.

For a single aggregation interval Δt, the *Hurst exponent* H is equal to the logarithm of the ratio of the volatilities divided by the logarithm of the aggregation interval Δt:

$$H = \frac{\log \sigma_t - \log \sigma}{\log \Delta t}$$

For a *vector* of aggregation intervals Δt_i, the *Hurst exponent* H is equal to the regression slope between the logarithms of the aggregated volatilities σ_i versus the logarithms of the aggregation intervals Δt_i:

$$H = \frac{\mathtt{cov}(\log \sigma_i, \log \Delta t_i)}{\mathtt{var}(\log \Delta t_i)}$$

The function `calc_hurst()` calls the function `calc_var_ag()` to calculate the variance of aggregated returns σ_t^2.

Value

The Hurst exponent calculated from the volatility ratio of aggregated returns. If `timeser` contains multiple columns, then the function `calc_hurst()` returns a single-row matrix of Hurst exponents.

Examples

```
## Not run:
# Calculate the log prices
closep <- na.omit(rutils::etfenv$prices[, c("XLP", "VTI")])
closep <- log(closep)
# Calculate the Hurst exponents for a 21 day aggregation interval
HighFreq::calc_hurst(closep, aggv=21)
# Calculate the Hurst exponents for a vector of aggregation intervals
aggv <- seq.int(from=3, to=35, length.out=9)^2
HighFreq::calc_hurst(closep, aggv=aggv)

## End(Not run)  # end dontrun
```

`calc_hurst_ohlc` *Calculate the Hurst exponent from the volatility ratio of aggregated* OHLC *prices.*

Description

Calculate the Hurst exponent from the volatility ratio of aggregated *OHLC* prices.

Usage

```
calc_hurst_ohlc(
  ohlc,
  step,
  method = "yang_zhang",
  closel = 0L,
  scale = TRUE,
  index = 0L
)
```

Arguments

`ohlc`	A *time series* or a *matrix* of *OHLC* prices.
`step`	The number of time periods in each interval between neighboring end points.
`method`	A *character string* representing the price range estimator for calculating the variance. The estimators include: • "close" close-to-close estimator, • "rogers_satchell" Rogers-Satchell estimator, • "garman_klass" Garman-Klass estimator, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "yang_zhang" Yang-Zhang estimator, (The default is the `method = "yang_zhang"`.)
`closel`	A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `closel = 0`).
`scale`	*Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scale = TRUE`).
`index`	A *vector* with the time index of the *time series*. This is an optional argument (the default is `index = 0`).

Details

The function `calc_hurst_ohlc()` calculates the Hurst exponent from the ratios of the variances of aggregated *OHLC* prices.

The aggregated volatility σ_t increases with the length of the aggregation interval Δt. The aggregated volatility increases as the length of the aggregation interval Δt raised to the power of the *Hurst exponent* H:

$$\sigma_t = \sigma \Delta t^H$$

Where σ is the daily return volatility.

The *Hurst exponent* H is equal to the logarithm of the ratio of the volatilities divided by the logarithm of the time interval Δt:

$$H = \frac{\log \sigma_t - \log \sigma}{\log \Delta t}$$

The function `calc_hurst_ohlc()` calls the function `calc_var_ohlc_ag()` to calculate the aggregated variance σ_t^2.

Value

The Hurst exponent calculated from the volatility ratio of aggregated *OHLC* prices.

Examples

```
## Not run:
# Calculate the log ohlc prices
ohlc <- log(rutils::etfenv$VTI)
# Calculate the Hurst exponent from 21 day aggregations
calc_hurst_ohlc(ohlc, step=21)

## End(Not run)  # end dontrun
```

`calc_inv` *Calculate the* reduced inverse *of a symmetric* matrix *of data using eigen decomposition.*

Description

Calculate the *reduced inverse* of a symmetric *matrix* of data using eigen decomposition.

Usage

```
calc_inv(matrixv, dimax = 0L, singmin = 0)
```

Arguments

`matrixv`	A symmetric *matrix* of data.
`dimax`	An *integer* equal to the number of *eigen values* used for calculating the *reduced inverse* of the matrix `matrixv` (the default is `dimax = 0` - standard matrix inverse using all the *eigen values*).
`singmin`	A *numeric* threshold level for discarding small *eigen values* in order to regularize the inverse of the matrix `matrixv` (the default is `0.0`).

Details

The function `calc_inv()` calculates the *reduced inverse* of the matrix `matrixv` using eigen decomposition.

The function `calc_inv()` first performs eigen decomposition of the matrix `matrixv`. The eigen decomposition of a matrix C is defined as the factorization:

$$C = O \, \Sigma \, O^T$$

Where O is the matrix of *eigen vectors* and Σ is a diagonal matrix of *eigen values*.

The inverse C^{-1} of the matrix C can be calculated from the eigen decomposition as:

$$C^{-1} = O \, \Sigma^{-1} \, O^T$$

The *reduced inverse* of the matrix C is obtained by removing *eigen vectors* with very small *eigen values*:

$$C^{-1} = O_{dimax} \, \Sigma^{-1}_{dimax} \, O^T_{dimax}$$

Where O_{dimax} is the matrix of *eigen vectors* that correspond to the largest *eigen values* Σ_{dimax}.

The function `calc_inv()` applies regularization to the matrix inverse using the arguments `dimax` and `singmin`.

The function `calc_inv()` applies regularization by discarding the smallest *eigen values* Σ_i that are less than the threshold level `singmin` times the sum of all the *eigen values*:

$$\Sigma_i < eigen\_thresh \cdot (\sum \Sigma_i)$$

It also discards additional *eigen vectors* so that only the highest order *eigen vectors* remain, up to order `dimax`. It calculates the *reduced inverse* from the eigen decomposition using only the largest *eigen values* up to `dimax`. For example, if `dimax = 3` then it only uses the `3` highest order *eigen vectors*, with the largest *eigen values*. This has the effect of dimension reduction.

If the matrix `matrixv` has a large number of small *eigen values*, then the number of remaining *eigen values* may be less than `dimax`.

Value

A *matrix* equal to the *reduced inverse* of the matrix `matrixv`.

Examples

```
## Not run:
# Calculate ETF returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "TLT", "DBC")])
# Calculate covariance matrix
covmat <- cov(retp)
# Calculate matrix inverse using RcppArmadillo
invmat <- HighFreq::calc_inv(covmat)
# Calculate matrix inverse in R
invr <- solve(covmat)
all.equal(invmat, invr, check.attributes=FALSE)
# Calculate the reduced inverse using RcppArmadillo
invmat <- HighFreq::calc_inv(covmat, dimax=3)
# Calculate the reduced inverse using eigen decomposition in R
eigend <- eigen(covmat)
dimax <- 1:3
invr <- eigend$vectors[, dimax] %*% (t(eigend$vectors[, dimax])/eigend$values[dimax])
# Compare RcppArmadillo with R
all.equal(invmat, invr)

## End(Not run)  # end dontrun
```

`calc_invrec` *Calculate the approximate inverse of a square* matrix *recursively using the Schulz formula (without copying the data in memory).*

Description

Calculate the approximate inverse of a square *matrix* recursively using the Schulz formula (without copying the data in memory).

Usage

```
calc_invrec(matrixv, invmat, niter = 1L)
```

Arguments

`matrixv`	A *matrix* of data to be inverted.
`invmat`	A *matrix* of data equal to the starting point for the recursion.
`niter`	An *integer* equal to the number of recursion iterations.

Details

The function `calc_invrec()` calculates the approximate inverse A^{-1} of a square *matrix* A recursively using the Schulz formula:

$$A_{i+1}^{-1} \leftarrow 2A_i^{-1} - A_i^{-1} A A_i^{-1}$$

The Schulz formula is repeated `niter` times. The Schulz formula is useful for updating the inverse when the matrix A changes only slightly. For example, for updating the inverse of the covariance matrix as it changes slowly over time.

The function `calc_invrec()` accepts a *pointer* to the argument `invmat` (which is the initial value of the inverse matrix for the recursion), and it overwrites the old inverse matrix values with the updated inverse values.

The function `calc_invrec()` performs the calculation in place, without copying the matrix in memory, which can significantly increase the computation speed for large matrices.

The function `calc_invrec()` doesn't return a value. The function `calc_invrec()` performs the calculations using `C++` `Armadillo` code.

Value

No return value.

Examples

```
## Not run:
# Calculate a random matrix
matrixv <- matrix(rnorm(100), nc=10)
# Define the initial value of the inverse matrix
invmat <- solve(matrixv) + matrix(rnorm(100, sd=0.1), nc=10)
# Calculate the inverse in place using RcppArmadillo
HighFreq::calc_invrec(matrixv, invmat, 3)
# Multiply the matrix times its inverse
multmat <- matrixv %*% invmat
round(multmat, 4)
# Calculate the sum of the off-diagonal elements
sum(multmat[upper.tri(multmat)])
# Compare RcppArmadillo with R
all.equal(invmat, solve(matrixv))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
   rcode=solve(matrixv),
   cppcode=HighFreq::calc_invrec(matrixv, invmat, 3),
   times=10))[, c(1, 4, 5)]

## End(Not run)  # end dontrun
```

`calc_invref` *Calculate the inverse of a square* matrix *in place, without copying the data in memory.*

Description

Calculate the inverse of a square *matrix* in place, without copying the data in memory.

Usage

```
calc_invref(matrixv)
```

Arguments

`matrixv`	A *matrix* of data to be inverted. (The argument is interpreted as a *pointer* to a *matrix*, and it is overwritten with the inverse matrix.)

Details

The function `calc_invref()` calculates the inverse of a square *matrix* in place, without copying the data in memory. It accepts a *pointer* to the argument `matrixv` (which is the `matrix` to be inverted), and it overwrites the old matrix values with the inverse matrix values. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large matrices.

The function `calc_invref()` doesn't return a value. The function `calc_invref()` calls the `C++` `Armadillo` function `arma::inv()` to calculate the matrix inverse.

Value

No return value.

Examples

```
## Not run:
# Calculate a random matrix
matrixv <- matrix(rnorm(100), nc=10)
# Copy matrixv to a matrix in a different memory location
invmat <- matrixv + 0
# Calculate the inverse in place using RcppArmadillo
HighFreq::calc_invref(invmat)
# Multiply the matrix times its inverse
multmat <- matrixv %*% invmat
round(multmat, 4)
# Calculate the sum of the off-diagonal elements
sum(multmat[upper.tri(multmat)])
# Compare RcppArmadillo with R
all.equal(invmat, solve(matrixv))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
   rcode=solve(matrixv),
   cppcode=calc_invref(matrixv),
   times=10))[, c(1, 4, 5)]

## End(Not run)  # end dontrun
```

`calc_invsvd` *Calculate the* reduced inverse *of a* matrix *of data using Singular Value Decomposition (*SVD*).*

Description

Calculate the *reduced inverse* of a *matrix* of data using Singular Value Decomposition (*SVD*).

Usage

```
calc_invsvd(matrixv, dimax = 0L, singmin = 0)
```

Arguments

`matrixv`	A *matrix* of data.
`dimax`	An *integer* equal to the number of *singular values* used for calculating the *reduced inverse* of the matrix `matrixv` (the default is `dimax = 0` - standard matrix inverse using all the *singular values*).
`singmin`	A *numeric* threshold level for discarding small *singular values* in order to regularize the inverse of the matrix `matrixv` (the default is `0.0`).

Details

The function `calc_invsvd()` calculates the *reduced inverse* of the matrix `matrixv` using Singular Value Decomposition (*SVD*).

The function `calc_invsvd()` first performs Singular Value Decomposition (*SVD*) of the matrix `matrixv`. The *SVD* of a matrix C is defined as the factorization:

$$C = U \, \Sigma \, V^T$$

Where U and V are the left and right *singular matrices*, and Σ is a diagonal matrix of *singular values*.

The inverse C^{-1} of the matrix C can be calculated from the *SVD* matrices as:

$$C^{-1} = V \, \Sigma^{-1} \, U^T$$

The *reduced inverse* of the matrix C is obtained by removing *singular vectors* with very small *singular values*:

$$C^{-1} = V_n \, \Sigma_n^{-1} \, U_n^T$$

Where U_n, V_n and Σ_n are the *SVD* matrices with the rows and columns corresponding to very small *singular values* removed.

The function `calc_invsvd()` applies regularization to the matrix inverse using the arguments `dimax` and `singmin`.

The function `calc_invsvd()` applies regularization by discarding the smallest *singular values* σ_i that are less than the threshold level `singmin` times the sum of all the *singular values*:

$$\sigma_i < eigen\_thresh \cdot (\sum \sigma_i)$$

It also discards additional *singular vectors* so that only the highest order *singular vectors* remain, up to order `dimax`. It calculates the *reduced inverse* from the *SVD* matrices using only the largest *singular values* up to order `dimax`. For example, if `dimax = 3` then it only uses the `3` highest order *singular vectors*, with the largest *singular values*. This has the effect of dimension reduction.

If the matrix `matrixv` has a large number of small *singular values*, then the number of remaining *singular values* may be less than `dimax`.

Value

A *matrix* equal to the *reduced inverse* of the matrix `matrixv`.

Examples

```
## Not run:
# Calculate ETF returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "TLT", "DBC")])
# Calculate covariance matrix
covmat <- cov(retp)
# Calculate matrix inverse using RcppArmadillo
invmat <- HighFreq::calc_invsvd(covmat)
# Calculate matrix inverse in R
invr <- solve(covmat)
all.equal(invmat, invr, check.attributes=FALSE)
# Calculate the reduced inverse using RcppArmadillo
invmat <- HighFreq::calc_invsvd(covmat, dimax=3)
# Calculate the reduced inverse from SVD in R
svdec <- svd(covmat)
dimax <- 1:3
invr <- svdec$v[, dimax] %*% (t(svdec$u[, dimax])/svdec$d[dimax])
# Compare RcppArmadillo with R
all.equal(invmat, invr)

## End(Not run)  # end dontrun
```

---

calc_kurtosis — *Calculate the kurtosis of the columns of a* time series *or a* matrix *using* `RcppArmadillo`.

Description

Calculate the kurtosis of the columns of a *time series* or a *matrix* using `RcppArmadillo`.

Usage

```
calc_kurtosis(timeser, method = "moment", confl = 0.75)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`method`	A *character string* specifying the type of the kurtosis model (the default is `method = "moment"` - see Details).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `calc_kurtosis()` calculates the kurtosis of the columns of the *matrix* `timeser` using `RcppArmadillo` C++ code.

If `method = "moment"` (the default) then `calc_kurtosis()` calculates the fourth moment of the data. But it doesn't center the columns of `timeser` because that requires copying the matrix `timeser` in memory, so it's time-consuming.

If `method = "quantile"` then it calculates the skewness κ from the differences between the quantiles of the data as follows:

$$\kappa = \frac{q_{\alpha} - q_{1-\alpha}}{q_{0.75} - q_{0.25}}$$

Where α is the confidence level for calculating the quantiles.

If `method = "nonparametric"` then it calculates the kurtosis as the difference between the mean of the data minus its median, divided by the standard deviation.

If the number of rows of `timeser` is less than `3` then it returns zeros.

The code examples below compare the function `calc_kurtosis()` with the kurtosis calculated using `R` code.

Value

A single-row matrix with the kurtosis of the columns of `timeser`.

Examples

```
## Not run:
# Define a single-column time series of returns
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the moment kurtosis
HighFreq::calc_kurtosis(retp)
# Calculate the moment kurtosis in R
calc_kurtr <- function(x) {
  x <- (x-mean(x))
  sum(x^4)/var(x)^2/NROW(x)
}  # end calc_kurtr
all.equal(HighFreq::calc_kurtosis(retp),
  calc_kurtr(retp), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_kurtosis(retp),
  Rcode=calc_kurtr(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the quantile kurtosis
HighFreq::calc_kurtosis(retp, method="quantile", confl=0.9)
# Calculate the quantile kurtosis in R
calc_kurtq <- function(x, a=0.9) {
   quantiles <- quantile(x, c(1-a, 0.25, 0.75, a), type=5)
   (quantiles[4] - quantiles[1])/(quantiles[3] - quantiles[2])
}  # end calc_kurtq
all.equal(drop(HighFreq::calc_kurtosis(retp, method="quantile", confl=0.9)),
  calc_kurtq(retp, a=0.9), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_kurtosis(retp, method="quantile"),
  Rcode=calc_kurtq(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the nonparametric kurtosis
HighFreq::calc_kurtosis(retp, method="nonparametric")
# Compare HighFreq::calc_kurtosis() with R nonparametric kurtosis
all.equal(drop(HighFreq::calc_kurtosis(retp, method="nonparametric")),
  (mean(retp)-median(retp))/sd(retp),
  check.attributes=FALSE)
```

```
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_kurtosis(retp, method="nonparametric"),
  Rcode=(mean(retp)-median(retp))/sd(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

calc_lm — *Perform multivariate linear regression using least squares and return a named list of regression coefficients, their t-values, and p-values.*

Description

Perform multivariate linear regression using least squares and return a named list of regression coefficients, their t-values, and p-values.

Usage

```
calc_lm(respv, predm)
```

Arguments

`respv`	A single-column *time series* or a *vector* of response data.
`predm`	A *time series* or a *matrix* of predictor data.

Details

The function `calc_lm()` performs the same calculations as the function `lm()` from package *stats*. It uses `RcppArmadillo` C++ code so it's several times faster than `lm()`. The code was inspired by this article (but it's not identical to it): http://gallery.rcpp.org/articles/fast-linear-model-with-armadillo/

Value

A named list with three elements: a *matrix* of coefficients (named *"coefficients"*), the *z-score* of the last residual (named *"zscore"*), and a *vector* with the R-squared and F-statistic (named *"stats"*). The numeric *matrix* of coefficients named *"coefficients"* contains the alpha and beta coefficients, and their *t-values* and *p-values*.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
respv <- retp[, 1]
# Predictor matrix equals VTI and IEF returns
predm <- retp[, -1]
# Perform multivariate regression using lm()
regmod <- lm(respv ~ predm)
regsum <- summary(regmod)
# Add unit intercept column to the predictor matrix
```

```
predm <- cbind(rep(1, NROW(predm)), predm)
# Perform multivariate regression using calc_lm()
regarma <- HighFreq::calc_lm(respv=respv, predm=predm)
# Compare the outputs of both functions
all.equal(regarma$coefficients[, "coeff"], unname(coef(regmod)))
all.equal(unname(regarma$coefficients), unname(regsum$coefficients))
all.equal(unname(regarma$stats), c(regsum$r.squared, unname(regsum$fstatistic[1])))
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_lm(respv=respv, predm=predm),
  Rcode=lm(respv ~ predm),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

calc_mean — *Calculate the mean (location) of the columns of a* time series *or a* matrix *using* RcppArmadillo.

Description

Calculate the mean (location) of the columns of a *time series* or a *matrix* using `RcppArmadillo`.

Usage

```
calc_mean(timeser, method = "moment", confl = 0.75)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`method`	A *character string* specifying the type of the mean (location) model (the default is `method = "moment"` - see Details).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `calc_mean()` calculates the mean (location) values of the columns of the *time series* `timeser` using C++ `RcppArmadillo` code.

If `method = "moment"` (the default) then `calc_mean()` calculates the location as the mean - the first moment of the data.

If `method = "quantile"` then it calculates the location $\bar{r}$ as the average of the quantiles as follows:

$$\bar{r} = \frac{q_{\alpha} + q_{1-\alpha}}{2}$$

Where α is the confidence level for calculating the quantiles (argument `confl`).

If `method = "nonparametric"` then it calculates the location as the median.

The code examples below compare the function `calc_mean()` with the mean (location) calculated using `R` code.

Value

A single-row matrix with the mean (location) of the columns of `timeser`.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLP", "VTI")])
# Calculate the column means in RcppArmadillo
HighFreq::calc_mean(retp)
# Calculate the column means in R
sapply(retp, mean)
# Compare the values
all.equal(drop(HighFreq::calc_mean(retp)),
  sapply(retp, mean), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_mean(retp),
  Rcode=sapply(retp, mean),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the quantile mean (location)
HighFreq::calc_mean(retp, method="quantile", confl=0.9)
# Calculate the quantile mean (location) in R
colSums(sapply(retp, quantile, c(0.9, 0.1), type=5))
# Compare the values
all.equal(drop(HighFreq::calc_mean(retp, method="quantile", confl=0.9)),
  colSums(sapply(retp, quantile, c(0.9, 0.1), type=5)),
  check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_mean(retp, method="quantile", confl=0.9),
  Rcode=colSums(sapply(retp, quantile, c(0.9, 0.1), type=5)),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the column medians in RcppArmadillo
HighFreq::calc_mean(retp, method="nonparametric")
# Calculate the column medians in R
sapply(retp, median)
# Compare the values
all.equal(drop(HighFreq::calc_mean(retp, method="nonparametric")),
  sapply(retp, median), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_mean(retp, method="nonparametric"),
  Rcode=sapply(retp, median),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_ranks` *Calculate the ranks of the elements of a single-column* time series, matrix, *or a* vector *using* `RcppArmadillo`.

Description

Calculate the ranks of the elements of a single-column *time series*, *matrix*, or a *vector* using RcppArmadillo.

Usage

```
calc_ranks(timeser)
```

Arguments

timeser	A single-column *time series*, *matrix*, or a *vector*.

Details

The function calc_ranks() calculates the ranks of the elements of a single-column *time series*, *matrix*, or a *vector*.

The permutation index is an integer vector which sorts a given vector into ascending order. The permutation index of the permutation index is the *reverse* permutation index, because it sorts the vector from ascending order back into its original unsorted order. The ranks of the elements are equal to the *reverse* permutation index. The function calc_ranks() calculates the *reverse* permutation index.

The ranks produced by calc_ranks() start at zero, following the C++ convention.

The Armadillo function arma::sort_index() calculates the permutation index which sorts a given vector into an ascending order. Applying the function arma::sort_index() twice:
arma::sort_index(arma::sort_index()),
calculates the *reverse* permutation index to sort the vector from ascending order back into its original unsorted order.

The function calc_ranks() calls the Armadillo function arma::sort_index() twice to calculate the *reverse* permutation index, to sort the vector from ascending order back into its original unsorted order.

Value

An *integer vector* with the ranks of the elements of the timeser.

Examples

```
## Not run:
# Create a vector of data
datav <- rnorm(1e3)
# Calculate the ranks of the elements using R code and RcppArmadillo
all.equal(rank(datav), drop(HighFreq::calc_ranks(datav))+1)
# Compare the speed of R code with RcppArmadillo
library(microbenchmark)
summary(microbenchmark(
  Rcode=rank(datav),
  Rcpp=calc_ranks(datav),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_ranks_stl` *Calculate the ranks of the elements of a single-column* time series, matrix*, or a* vector *using* `RcppArmadillo`.

Description

Calculate the ranks of the elements of a single-column *time series*, *matrix*, or a *vector* using `RcppArmadillo`.

Usage

```
calc_ranks_stl(timeser)
```

Arguments

`timeser`	A single-column *time series*, *matrix*, or a *vector*.

Details

The function `calc_ranks_stl()` calculates the ranks of the elements of a single-column *time series*, *matrix*, or a *vector*. The function `calc_ranks_stl()` is slightly faster than the function `calc_ranks()`.

The permutation index is an integer vector which sorts a given vector into ascending order. The permutation index of the permutation index is the *reverse* permutation index, because it sorts the vector from ascending order back into its original unsorted order. The ranks of the elements are equal to the *reverse* permutation index. The function `calc_ranks()` calculates the *reverse* permutation index.

The ranks produced by `calc_ranks_stl()` start at zero, following the `C++` convention.

The `STL` `C++` function `std::sort()` sorts a vector into ascending order. It can also be used to calculate the permutation index which sorts the vector into an ascending order.

The function `calc_ranks_stl()` calls the function `std::sort()` twice: First, it calculates the permutation index which sorts the vector `timeser` into ascending order. Second, it calculates the permutation index of the permutation index, which are the ranks (the *reverse* permutation index) of the vector `timeser`.

Value

An *integer vector* with the ranks of the elements of `timeser`.

Examples

```
## Not run:
# Create a vector of data
datav <- rnorm(1e3)
# Calculate the ranks of the elements using R code and RcppArmadillo
all.equal(rank(datav), drop(HighFreq::calc_ranks_stl(datav))+1)
# Compare the speed of R code with RcppArmadillo
library(microbenchmark)
summary(microbenchmark(
  Rcode=rank(datav),
  Rcpp=HighFreq::calc_ranks_stl(datav),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
## End(Not run)  # end dontrun
```

`calc_reg` *Perform multivariate regression using different methods, and return a vector of regression coefficients, their t-values, and the last residual z-score.*

Description

Perform multivariate regression using different methods, and return a vector of regression coefficients, their t-values, and the last residual z-score.

Usage

```
calc_reg(respv, predm, controll)
```

Arguments

respv	A single-column *time series* or a *vector* of response data.
predm	A *time series* or a *matrix* of predictor data.
controll	A *list* of model parameters (see Details).

Details

The function `calc_reg()` performs multivariate regression using different methods, and returns a vector of regression coefficients, their t-values, and the last residual z-score.

The function `calc_reg()` accepts a list of regression model parameters through the argument `controll`. The list of model parameters can be created using the function `param_reg()`. Below is a description of the model parameters.

If `regmod = "least_squares"` (the default) then it performs the standard least squares regression, the same as the function `calc_lm()`, and the function `lm()` from the `R` package *stats*. But it uses `RcppArmadillo` `C++` code so it's several times faster than `lm()`.

If `regmod = "regular"` then it performs shrinkage regression. It calculates the *reduced inverse* of the predictor matrix from its singular value decomposition. It performs regularization by selecting only the largest *singular values* equal in number to `dimax`.

If `regmod = "quantile"` then it performs quantile regression (not implemented yet).

The length of the return vector depends on the number of columns of the predictor matrix (including the intercept column, if it's been added in `R`). The number of regression coefficients is equal to the number of columns of the predictor matrix. The length of the return vector is equal to the number of regression coefficients, plus their t-values, plus the z-score. The number of t-values is equal to the number of coefficients.

For example, if the number of columns of the predictor matrix is equal to `n`, then `calc_reg()` returns a vector with `2n+1` elements: `n` regression coefficients, `n` corresponding t-values, and `1` z-score value.

Value

A single-row matrix with the regression coefficients, their t-values, and the last residual z-score.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
respv <- retp[, 1]
# Predictor matrix equals VTI and IEF returns
predm <- retp[, -1]
# Perform multivariate regression using lm()
regmod <- lm(respv ~ predm)
regsum <- summary(regmod)
coeff <- regsum$coefficients
# Create a default list of regression parameters
controll <- HighFreq::param_reg()
# Add unit intercept column to the predictor matrix
predm <- cbind(rep(1, NROW(predm)), predm)
# Perform multivariate regression using calc_reg()
regarma <- drop(HighFreq::calc_reg(respv=respv, predm=predm, controll=controll))
# Compare the outputs of both functions
all.equal(regarma[1:(2*NCOL(predm))],
  c(coeff[, "Estimate"], coeff[, "t value"]), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_reg(respv=respv, predm=predm, controll=controll),
  Rcode=lm(respv ~ predm),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_scale` *Standardize (center and scale) the columns of a* time series *of data in place, without copying the data in memory, using* `RcppArmadillo`.

Description

Standardize (center and scale) the columns of a *time series* of data in place, without copying the data in memory, using `RcppArmadillo`.

Usage

```
calc_scale(timeser, center = TRUE, scale = TRUE, use_median = FALSE)
```

Arguments

`timeser`	A *time series* or *matrix* of data.
`center`	A *Boolean* argument: if `TRUE` then center the columns so that they have zero mean or median (the default is `TRUE`).
`scale`	A *Boolean* argument: if `TRUE` then scale the columns so that they have unit standard deviation or MAD (the default is `TRUE`).

use_median	A *Boolean* argument: if TRUE then the centrality (central tendency) is calculated as the *median* and the dispersion is calculated as the *median absolute deviation* (*MAD*) (the default is FALSE). If use_median = FALSE then the centrality is calculated as the *mean* and the dispersion is calculated as the *standard deviation*.

Details

The function calc_scale() standardizes (centers and scales) the columns of a *time series* of data in place, without copying the data in memory, using RcppArmadillo.

If the arguments center and scale are both TRUE and use_median is FALSE (the defaults), then calc_scale() performs the same calculation as the standard R function scale(), and it calculates the centrality (central tendency) as the *mean* and the dispersion as the *standard deviation*.

If the arguments center and scale are both TRUE (the defaults), then calc_scale() standardizes the data. If the argument center is FALSE then calc_scale() only scales the data (divides it by the standard deviations). If the argument scale is FALSE then calc_scale() only demeans the data (subtracts the means).

If the argument use_median is TRUE, then it calculates the centrality as the *median* and the dispersion as the *median absolute deviation* (*MAD*).

If the number of rows of timeser is less than 3 then it does nothing and timeser is not scaled.

The function calc_scale() accepts a *pointer* to the argument timeser, and it overwrites the old data with the standardized data. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large time series.

The function calc_scale() uses RcppArmadillo C++ code, so on a typical time series it can be over *10* times faster than the function scale().

Value

Void (no return value - modifies the data in place).

Examples

```
## Not run:
# Calculate a time series of returns
retp <- zoo::coredata(na.omit(rutils::etfenv$returns[, c("IEF", "VTI")]))
# Demean the returns
demeaned <- apply(retp, 2, function(x) (x-mean(x)))
HighFreq::calc_scale(retp, scale=FALSE)
all.equal(demeaned, retp, check.attributes=FALSE)
# Calculate a time series of returns
retp <- zoo::coredata(na.omit(rutils::etfenv$returns[, c("IEF", "VTI")]))
# Standardize the returns
retss <- scale(retp)
HighFreq::calc_scale(retp)
all.equal(retss, retp, check.attributes=FALSE)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcode=scale(retp),
  Rcpp=HighFreq::calc_scale(retp),
  times=100))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

calc_skew — *Calculate the skewness of the columns of a* time series *or a* matrix *using* RcppArmadillo.

Description

Calculate the skewness of the columns of a *time series* or a *matrix* using RcppArmadillo.

Usage

```
calc_skew(timeser, method = "moment", confl = 0.75)
```

Arguments

timeser	A *time series* or a *matrix* of data.
method	A *character string* specifying the type of the skewness model (the default is method = "moment" - see Details).
confl	The confidence level for calculating the quantiles of returns (the default is confl = 0.75).

Details

The function calc_skew() calculates the skewness of the columns of a *time series* or a *matrix* of data using C++ RcppArmadillo code.

If method = "moment" (the default) then calc_skew() calculates the skewness as the third moment of the data.

If method = "quantile" then it calculates the skewness ς from the differences between the quantiles of the data as follows:

$$\varsigma = \frac{q_{\alpha} + q_{1-\alpha} - 2 q_{0.5}}{q_{\alpha} - q_{1-\alpha}}$$

Where α is the confidence level for calculating the quantiles.

If method = "nonparametric" then it calculates the skewness as the difference between the mean of the data minus its median, divided by the standard deviation.

If the number of rows of timeser is less than 3 then it returns zeros.

The code examples below compare the function calc_skew() with the skewness calculated using R code.

Value

A single-row matrix with the skewness of the columns of timeser.

Examples

```
## Not run:
# Define a single-column time series of returns
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the moment skewness
HighFreq::calc_skew(retp)
# Calculate the moment skewness in R
calc_skewr <- function(x) {
  x <- (x-mean(x))
  sum(x^3)/var(x)^1.5/NROW(x)
}  # end calc_skewr
all.equal(HighFreq::calc_skew(retp),
  calc_skewr(retp), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_skew(retp),
  Rcode=calc_skewr(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the quantile skewness
HighFreq::calc_skew(retp, method="quantile", confl=0.9)
# Calculate the quantile skewness in R
calc_skewq <- function(x, a = 0.75) {
   quantiles <- quantile(x, c(1-a, 0.5, a), type=5)
   (quantiles[3] + quantiles[1] - 2*quantiles[2])/(quantiles[3] - quantiles[1])
}  # end calc_skewq
all.equal(drop(HighFreq::calc_skew(retp, method="quantile", confl=0.9)),
  calc_skewq(retp, a=0.9), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_skew(retp, method="quantile"),
  Rcode=calc_skewq(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Calculate the nonparametric skewness
HighFreq::calc_skew(retp, method="nonparametric")
# Compare HighFreq::calc_skew() with R nonparametric skewness
all.equal(drop(HighFreq::calc_skew(retp, method="nonparametric")),
  (mean(retp)-median(retp))/sd(retp),
  check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_skew(retp, method="nonparametric"),
  Rcode=(mean(retp)-median(retp))/sd(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_startpoints` *Calculate a vector of start points by lagging (shifting) a vector of end points.*

Description

Calculate a vector of start points by lagging (shifting) a vector of end points.

Usage

```
calc_startpoints(endd, lookb)
```

Arguments

endd	An *integer* vector of end points.
lookb	The length of the look-back interval, equal to the lag (shift) applied to the end points.

Details

The start points are equal to the values of the vector `endd` lagged (shifted) by an amount equal to `lookb`. In addition, an extra value of `1` is added to them, to avoid data overlaps. The lag operation requires appending a beginning warmup interval containing zeros, so that the vector of start points has the same length as the `endd`.

For example, consider the end points for a vector of length `25` divided into equal intervals of length `5`: `4, 9, 14, 19, 24`. (In `C++` the vector indexing starts at `0` not `1`, so it's shifted by `-1`.) Then the start points for `lookb = 2` are equal to: `0, 0, 5, 10, 15`. The differences between the end points minus the corresponding start points are equal to `9`, except for the warmup interval.

Value

An *integer* vector with the same number of elements as the vector `endd`.

Examples

```
# Calculate end points
endd <- HighFreq::calc_endpoints(length=55, step=5)
# Calculate start points corresponding to the end points
startp <- HighFreq::calc_startpoints(endd, lookb=5)
```

---

calc_var — *Calculate the dispersion (variance) of the columns of a* time series *or a* matrix *using* `RcppArmadillo`.

Description

Calculate the dispersion (variance) of the columns of a *time series* or a *matrix* using `RcppArmadillo`.

Usage

```
calc_var(timeser, method = "moment", confl = 0.75)
```

Arguments

timeser	A *time series* or a *matrix* of data.
method	A *character string* specifying the type of the dispersion model (the default is `method = "moment"` - see Details).
confl	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `calc_var()` calculates the dispersion of the columns of a *time series* or a *matrix* of data using `RcppArmadillo` `C++` code.

The dispersion is a measure of the variability of the data. Examples of dispersion are the variance and the Median Absolute Deviation (*MAD*).

If `method = "moment"` (the default) then `calc_var()` calculates the dispersion as the second moment of the data (the variance). Then `calc_var()` performs the same calculation as the function `colVars()` from package matrixStats, but it's much faster because it uses `RcppArmadillo` `C++` code.

If `method = "quantile"` then it calculates the dispersion as the difference between the quantiles as follows:

$$\sigma = q_{\alpha} - q_{1-\alpha}$$

Where α is the confidence level for calculating the quantiles.

If `method = "nonparametric"` then it calculates the dispersion as the Median Absolute Deviation (*MAD*):

$$MAD = median(abs(x - median(x)))$$

It also multiplies the *MAD* by a factor of `1.4826`, to make it comparable to the standard deviation.

If `method = "nonparametric"` then `calc_var()` performs the same calculation as the function `stats::mad()`, but it's much faster because it uses `RcppArmadillo` `C++` code.

If the number of rows of `timeser` is less than `3` then it returns zeros.

Value

A row vector equal to the dispersion of the columns of the matrix `timeser`.

Examples

```
## Not run:
# Calculate VTI and XLF returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "XLF")])
# Compare HighFreq::calc_var() with standard var()
all.equal(drop(HighFreq::calc_var(retp)),
  apply(retp, 2, var), check.attributes=FALSE)
# Compare HighFreq::calc_var() with matrixStats
all.equal(drop(HighFreq::calc_var(retp)),
  matrixStats::colVars(retp), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with matrixStats and with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_var(retp),
  matrixStats=matrixStats::colVars(retp),
  Rcode=apply(retp, 2, var),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Compare HighFreq::calc_var() with stats::mad()
all.equal(drop(HighFreq::calc_var(retp, method="nonparametric")),
  sapply(retp, mad), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with stats::mad()
summary(microbenchmark(
  Rcpp=HighFreq::calc_var(retp, method="nonparametric"),
  Rcode=sapply(retp, mad),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
## End(Not run)  # end dontrun
```

calc_varvec — *Calculate the variance of a single-column* time series *or a* vector *using* RcppArmadillo.

Description

Calculate the variance of a single-column *time series* or a *vector* using RcppArmadillo.

Usage

```
calc_varvec(timeser)
```

Arguments

timeser	A single-column *time series* or a *vector*.

Details

The function calc_varvec() calculates the variance of a *vector* using RcppArmadillo C++ code, so it's significantly faster than the R function var().

Value

A *numeric* value equal to the variance of the *vector*.

Examples

```
## Not run:
# Create a vector of random returns
retp <- rnorm(1e6)
# Compare calc_varvec() with standard var()
all.equal(HighFreq::calc_varvec(retp), var(retp))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_varvec(retp),
  Rcode=var(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`calc_var_ag` *Calculate the variance of returns aggregated over the end points.*

Description

Calculate the variance of returns aggregated over the end points.

Usage

```
calc_var_ag(pricev, step = 1L)
```

Arguments

`pricev`	A *time series* or a *matrix* of prices.
`step`	The number of time periods in each interval between neighboring end points (the default is `step = 1`).

Details

The function `calc_var_ag()` calculates the variance of returns aggregated over the end points.

It first calculates the end points spaced apart by the number of periods equal to the argument `step`. Then it calculates the aggregated returns by differencing the prices `pricev` calculated at the end points. Finally it calculates the variance of the returns.

The choice of the first end point is arbitrary, so `calc_var_ag()` calculates the different end points for all the possible starting points. It then calculates the variance values for all the different end points and averages them.

The aggregated volatility σ_t increases with the length of the aggregation interval Δt. The aggregated volatility increases as the length of the aggregation interval Δt raised to the power of the *Hurst exponent* H:

$$\sigma_t = \sigma \Delta t^H$$

Where σ is the daily return volatility.

The function `calc_var_ag()` can therefore be used to calculate the *Hurst exponent* from the variance ratio.

Value

The variance of aggregated returns.

Examples

```
## Not run:
# Calculate the prices
closep <- na.omit(rutils::etfenv$prices[, c("XLP", "VTI")])
closep <- log(closep)
# Calculate the variance of daily returns
calc_var_ag(closep, step=1)
# Calculate the variance using R
sapply(rutils::diffit(closep), var)
# Calculate the variance of returns aggregated over 21 days
calc_var_ag(closep, step=21)
```

```
# The variance over 21 days is approximately 21 times the daily variance
21*calc_var_ag(closep, step=1)

## End(Not run)  # end dontrun
```

calc_var_ohlc — *Calculate the variance of returns from* OHLC *prices using different price range estimators.*

Description

Calculate the variance of returns from *OHLC* prices using different price range estimators.

Usage

```
calc_var_ohlc(
  ohlc,
  method = "yang_zhang",
  closel = 0L,
  scale = TRUE,
  index = 0L
)
```

Arguments

`ohlc`	A *time series* or a *matrix* of *OHLC* prices.
`method`	A *character string* representing the price range estimator for calculating the variance. The estimators include: • "close" close-to-close estimator, • "rogers_satchell" Rogers-Satchell estimator, • "garman_klass" Garman-Klass estimator, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "yang_zhang" Yang-Zhang estimator, (The default is the `method = "yang_zhang"`.)
`closel`	A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `closel = 0`).
`scale`	*Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scale = TRUE`).
`index`	A *vector* with the time index of the *time series*. This is an optional argument (the default is `index = 0`).

Details

The function `calc_var_ohlc()` calculates the variance from all the different intra-day and day-over-day returns (defined as the differences of *OHLC* prices), using several different variance estimation methods.

The function `calc_var_ohlc()` does not calculate the logarithm of the prices. So if the argument `ohlc` contains dollar prices then `calc_var_ohlc()` calculates the dollar variance. If the argument `ohlc` contains the log prices then `calc_var_ohlc()` calculates the percentage variance.

The default method is *"yang_zhang"*, which theoretically has the lowest standard error among unbiased estimators. The methods *"close"*, *"garman_klass_yz"*, and *"yang_zhang"* do account for *close-to-open* price jumps, while the methods *"garman_klass"* and *"rogers_satchell"* do not account for *close-to-open* price jumps.

If scale is TRUE (the default), then the returns are divided by the differences of the time index (which scales the variance to the units of variance per second squared). This is useful when calculating the variance from minutes bar data, because dividing returns by the number of seconds decreases the effect of overnight price jumps. If the time index is in days, then the variance is equal to the variance per day squared.

If the number of rows of ohlc is less than 3 then it returns zero.

The optional argument index is the time index of the *time series* ohlc. If the time index is in seconds, then the differences of the index are equal to the number of seconds in each time period. If the time index is in days, then the differences are equal to the number of days in each time period.

The optional argument closel are the lagged *close* prices of the *OHLC time series*. Passing in the lagged *close* prices speeds up the calculation, so it's useful for rolling calculations.

The function calc_var_ohlc() is implemented in RcppArmadillo C++ code, and it's over 10 times faster than calc_var_ohlc_r(), which is implemented in R code.

Value

A single *numeric* value equal to the variance of the *OHLC time series*.

Examples

```
## Not run:
# Extract the log OHLC prices of SPY
ohlc <- log(HighFreq::SPY)
# Extract the time index of SPY prices
indeks <- c(1, diff(xts::.index(ohlc)))
# Calculate the variance of SPY returns, with scaling of the returns
HighFreq::calc_var_ohlc(ohlc,
 method="yang_zhang", scale=TRUE, index=indeks)
# Calculate variance without accounting for overnight jumps
HighFreq::calc_var_ohlc(ohlc,
 method="rogers_satchell", scale=TRUE, index=indeks)
# Calculate the variance without scaling the returns
HighFreq::calc_var_ohlc(ohlc, scale=FALSE)
# Calculate the variance by passing in the lagged close prices
closel <- HighFreq::lagit(ohlc[, 4])
all.equal(HighFreq::calc_var_ohlc(ohlc),
  HighFreq::calc_var_ohlc(ohlc, closel=closel))
# Compare with HighFreq::calc_var_ohlc_r()
all.equal(HighFreq::calc_var_ohlc(ohlc, index=indeks),
  HighFreq::calc_var_ohlc_r(ohlc))
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_var_ohlc(ohlc),
  Rcode=HighFreq::calc_var_ohlc_r(ohlc),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

calc_var_ohlc_ag — *Calculate the variance of aggregated* OHLC *prices using different price range estimators.*

Description

Calculate the variance of aggregated *OHLC* prices using different price range estimators.

Usage

```
calc_var_ohlc_ag(
  ohlc,
  step,
  method = "yang_zhang",
  closel = 0L,
  scale = TRUE,
  index = 0L
)
```

Arguments

`ohlc`	A *time series* or a *matrix* of *OHLC* prices.
`step`	The number of time periods in each interval between neighboring end points.
`method`	A *character string* representing the price range estimator for calculating the variance. The estimators include: • "close" close-to-close estimator, • "rogers_satchell" Rogers-Satchell estimator, • "garman_klass" Garman-Klass estimator, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "yang_zhang" Yang-Zhang estimator, (The default is the `method = "yang_zhang"`.)
`closel`	A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `closel = 0`).
`scale`	*Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scale = TRUE`).
`index`	A *vector* with the time index of the *time series*. This is an optional argument (the default is `index = 0`).

Details

The function `calc_var_ohlc_ag()` calculates the variance of *OHLC* prices aggregated over the end points.

It first calculates the end points spaced apart by the number of periods equal to the argument `step`. Then it aggregates the *OHLC* prices to the end points. Finally it calculates the variance of the aggregated *OHLC* prices.

The choice of the first end point is arbitrary, so `calc_var_ohlc_ag()` calculates the different end points for all the possible starting points. It then calculates the variance values for all the different end points and averages them.

The aggregated volatility σ_t increases with the length of the aggregation interval Δt. The aggregated volatility increases as the length of the aggregation interval Δt raised to the power of the *Hurst exponent* H:

$$\sigma_t = \sigma \Delta t^H$$

Where σ is the daily return volatility.

The function `calc_var_ohlc_ag()` can therefore be used to calculate the *Hurst exponent* from the variance ratio.

Value

The variance of aggregated *OHLC* prices.

Examples

```
## Not run:
# Calculate the log ohlc prices
ohlc <- log(rutils::etfenv$VTI)
# Calculate the daily variance of percentage returns
calc_var_ohlc_ag(ohlc, step=1)
# Calculate the variance of returns aggregated over 21 days
calc_var_ohlc_ag(ohlc, step=21)
# The variance over 21 days is approximately 21 times the daily variance
21*calc_var_ohlc_ag(ohlc, step=1)

## End(Not run)  # end dontrun
```

`calc_var_ohlc_r` *Calculate the variance of an* OHLC *time series, using different range estimators for variance.*

Description

Calculate the variance of an *OHLC* time series, using different range estimators for variance.

Usage

```
calc_var_ohlc_r(ohlc, method = "yang_zhang", scalit = TRUE)
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`method`	A *character* string representing the method for estimating variance. The methods include: • "close" close to close, • "garman_klass" Garman-Klass, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "rogers_satchell" Rogers-Satchell, • "yang_zhang" Yang-Zhang, (default is `"yang_zhang"`)
`scalit`	*Boolean* argument: should the returns be divided by the number of seconds in each period? (default is `TRUE`)

Details

The function `calc_var_ohlc_r()` calculates the variance from all the different intra-day and day-over-day returns (defined as the differences of *OHLC* prices), using several different variance estimation methods.

The default method is `"yang_zhang"`, which theoretically has the lowest standard error among unbiased estimators. The methods `"close"`, `"garman_klass_yz"`, and `"yang_zhang"` do account for close-to-open price jumps, while the methods `"garman_klass"` and `"rogers_satchell"` do not account for close-to-open price jumps.

If `scalit` is `TRUE` (the default), then the returns are divided by the differences of the time index (which scales the variance to the units of variance per second squared.) This is useful when calculating the variance from minutely bar data, because dividing returns by the number of seconds decreases the effect of overnight price jumps. If the time index is in days, then the variance is equal to the variance per day squared.

The function `calc_var_ohlc_r()` is implemented in `R` code.

Value

A single *numeric* value equal to the variance.

Examples

```
# Calculate the variance of SPY returns
HighFreq::calc_var_ohlc_r(HighFreq::SPY, method="yang_zhang")
# Calculate variance without accounting for overnight jumps
HighFreq::calc_var_ohlc_r(HighFreq::SPY, method="rogers_satchell")
# Calculate the variance without scaling the returns
HighFreq::calc_var_ohlc_r(HighFreq::SPY, scalit=FALSE)
```

`calc_weights` *Calculate the optimal portfolio weights using a variety of different objective functions.*

Description

Calculate the optimal portfolio weights using a variety of different objective functions.

Usage

```
calc_weights(returns, controll)
```

Arguments

`returns`	A *time series* or a *matrix* of returns data (the returns in excess of the risk-free rate).
`controll`	A *list* of portfolio optimization model parameters (see Details).

Details

The function `calc_weights()` calculates the optimal portfolio weights using a variety of different objective functions.

The function `calc_weights()` accepts a list of portfolio optimization parameters through the argument `controll`. The list of portfolio optimization parameters can be created using the function `param_portf()`. Below is a description of the parameters.

If `method = "maxsharpe"` (the default) then `calc_weights()` calculates the weights of the maximum Sharpe portfolio, by multiplying the *reduced inverse* of the *covariance matrix* C^{-1} times the mean column returns $\bar{r}$:

$$w = C^{-1} \bar{r}$$

If `method = "maxsharpemed"` then `calc_weights()` uses the medians instead of the means.

If `method = "minvarlin"` then it calculates the weights of the minimum variance portfolio under linear constraint, by multiplying the *reduced inverse* of the *covariance matrix* times the unit vector:

$$w = C^{-1} \mathbb{1}$$

If `method = "minvarquad"` then it calculates the weights of the minimum variance portfolio under quadratic constraint (which is the highest order principal component).

If `method = "sharpem"` then it calculates the momentum weights equal to the Sharpe ratios (the `returns` divided by their standard deviations):

$$w = \frac{\bar{r}}{\sigma}$$

If `method = "kellym"` then it calculates the momentum weights equal to the Kelly ratios (the `returns` divided by their variance):

$$w = \frac{\bar{r}}{\sigma^2}$$

`calc_weights()` calls the function `calc_inv()` to calculate the *reduced inverse* of the *covariance matrix* of `returns`. It performs regularization by selecting only the largest eigenvalues equal in number to `dimax`.

In addition, `calc_weights()` applies shrinkage to the columns of `returns`, by shrinking their means to their common mean value:

$$r'_i = (1 - \alpha) \, \bar{r}_i + \alpha \, \mu$$

Where $\bar{r}_i$ is the mean of column i and μ is the average of all the column means. The shrinkage intensity `alpha` determines the amount of shrinkage that is applied, with `alpha = 0` representing no shrinkage (with the column means $\bar{r}_i$ unchanged), and `alpha = 1` representing complete shrinkage (with the column means all equal to the single mean of all the columns: $\bar{r}_i = \mu$).

After the weights are calculated, they are scaled, depending on several arguments.

If `rankw = TRUE` then the weights are converted into their ranks. The default is `rankw = FALSE`.

If `centerw = TRUE` then the weights are centered so that their sum is equal to `0`. The default is `centerw = FALSE`.

If `scalew = "voltarget"` (the default) then the weights are scaled (multiplied by a factor) so that the weighted portfolio has an in-sample volatility equal to `voltarget`.

If `scalew = "voleqw"` then the weights are scaled so that the weighted portfolio has the same volatility as the equal weight portfolio.

If `scalew = "sumone"` then the weights are scaled so that their sum is equal to `1`.

If `scalew = "sumsq"` then the weights are scaled so that their sum of squares is equal to `1`.

If `scalew = "none"` then the weights are not scaled.

The function `calc_weights()` is written in `C++` `RcppArmadillo` code.

Value

A column *vector* of the same length as the number of columns of `returns`.

Examples

```
## Not run:
# Calculate covariance matrix and eigen decomposition of ETF returns
retp <- na.omit(rutils::etfenv$returns[, 1:16])
ncols <- NCOL(retp)
eigend <- eigen(cov(retp))

# Calculate the reduced inverse of covariance matrix
dimax <- 3
eigenvec <- eigend$vectors[, 1:dimax]
eigenval <- eigend$values[1:dimax]
invmat <- eigenvec %*% (t(eigenvec) / eigenval)
# Define shrinkage intensity and apply shrinkage to the mean returns
alphac <- 0.5
colmeans <- colMeans(retp)
colmeans <- ((1-alphac)*colmeans + alphac*mean(colmeans))
# Calculate the weights using R
weightr <- drop(invmat %*% colmeans)
# Apply weights scaling
weightr <- weightr*sd(rowMeans(retp))/sd(retp %*% weightr)
weightr <- 0.01*weightr/sd(retp %*% weightr)
weightr <- weightr/sqrt(sum(weightr^2))
# Create a list of portfolio optimization parameters
controll <- HighFreq::param_portf(method="maxsharpe", dimax=dimax, alphac=alphac, scalew="sumsq")
# Calculate the weights using RcppArmadillo
weightcpp <- drop(HighFreq::calc_weights(retp, controll=controll))
all.equal(weightcpp, weightr)

# Calculate the max Sharpe weights and scale them to the target volatility
voltarget <- 0.015
controll <- HighFreq::param_portf(method="maxsharpe", scalew="voltarget", voltarget=voltarget)
weightcpp <- HighFreq::calc_weights(retp, controll=controll)
# The portfolio volatility matches the target volatility
all.equal(sd(retp %*% weightcpp), voltarget)


## End(Not run)  # end dontrun
```

`decode_it` *Calculate the* vector *of data from its run length encoding.*

Description

Calculate the *vector* of data from its run length encoding.

Usage

```
decode_it(encodel)
```

Arguments

`encodel`	A *list* with two *vectors*: a *numeric vector* of encoded data and an *integer vector* of data counts (repeats).

Details

The function `decode_it()` the *vector* of data from its run length encoding.

The run length encoding of a *vector* consists of two *vectors*: a *numeric vector* of encoded data (consecutive data values) and of an *integer vector* of the data counts (the number of times the same value repeats in succession).

Run length encoding (RLE) is a data compression algorithm which encodes the data in two *vectors*: the consecutive data values and their counts. If a data value occurs several times in succession then it is recorded only once and its corresponding count is equal to the number of times it occurs. Run-length encoding is different from a contingency table.

Value

A `numeric` `vector`.

Examples

```
## Not run:
# Create a vector of data
datav <- sample(5, 31, replace=TRUE)
# Calculate the run length encoding of datav
rle <- HighFreq::encode_it(datav)
# Decode the data from its run length encoding
decodev <- HighFreq::decode_it(rle)
all.equal(datav, decodev)

## End(Not run)  # end dontrun
```

`diffit` *Calculate the row differences of a* time series *or a* matrix *using* RcppArmadillo.

Description

Calculate the row differences of a *time series* or a *matrix* using *RcppArmadillo*.

Usage

```
diffit(timeser, lagg = 1L, pad_zeros = TRUE)
```

Arguments

timeser	A *time series* or a *matrix*.
lagg	An *integer* equal to the number of rows (time periods) to lag when calculating the differences (the default is lagg = 1).
pad_zeros	*Boolean* argument: Should the output *matrix* be padded (extended) with zero values, in order to return a *matrix* with the same number of rows as the input? (the default is pad_zeros = TRUE)

Details

The function diffit() calculates the differences between the rows of the input *matrix* timeser and its lagged version.

The argument lagg specifies the number of lags applied to the rows of the lagged version of timeser. For positive lagg values, the lagged version of timeser has its rows shifted *forward* (down) by the number equal to lagg rows. For negative lagg values, the lagged version of timeser has its rows shifted *backward* (up) by the number equal to -lagg rows. For example, if lagg=3 then the lagged version will have its rows shifted down by 3 rows, and the differences will be taken between each row minus the row three time periods before it (in the past). The default is lagg = 1.

The argument pad_zeros specifies whether the output *matrix* should be padded (extended) with zero values in order to return a *matrix* with the same number of rows as the input timeser. The default is pad_zeros = TRUE. If pad_zeros = FALSE then the return *matrix* has a smaller number of rows than the input timeser. The padding operation can be time-consuming, because it requires the copying the data in memory.

The function diffit() is implemented in RcppArmadillo C++ code, which makes it several times faster than R code.

Value

A *matrix* containing the differences between the rows of the input *matrix* timeser.

Examples

```
## Not run:
# Create a matrix of random data
datav <- matrix(sample(15), nc=3)
# Calculate differences with lagged rows
HighFreq::diffit(datav, lagg=2)
# Calculate differences with advanced rows
HighFreq::diffit(datav, lagg=-2)
# Compare HighFreq::diffit() with rutils::diffit()
all.equal(HighFreq::diffit(datav, lagg=2),
  rutils::diffit(datav, lagg=2),
  check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::diffit(datav, lagg=2),
  Rcode=rutils::diffit(datav, lagg=2),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`diff_vec` *Calculate the differences between the neighboring elements of a single-column* time series *or a* vector.

Description

Calculate the differences between the neighboring elements of a single-column *time series* or a *vector*.

Usage

```
diff_vec(timeser, lagg = 1L, pad_zeros = TRUE)
```

Arguments

timeser	A single-column *time series* or a *vector*.
lagg	An *integer* equal to the number of time periods to lag when calculating the differences (the default is lagg = 1).
pad_zeros	*Boolean* argument: Should the output *vector* be padded (extended) with zeros, in order to return a *vector* of the same length as the input? (the default is pad_zeros = TRUE)

Details

The function `diff_vec()` calculates the differences between the input *time series* or *vector* and its lagged version.

The argument `lagg` specifies the number of lags. For example, if `lagg=3` then the differences will be taken between each element minus the element three time periods before it (in the past). The default is `lagg = 1`.

The argument `pad_zeros` specifies whether the output *vector* should be padded (extended) with zeros at the front, in order to return a *vector* of the same length as the input. The default is `pad_zeros = TRUE`. The padding operation can be time-consuming, because it requires the copying the data in memory.

The function `diff_vec()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A column *vector* containing the differences between the elements of the input vector.

Examples

```
## Not run:
# Create a vector of random returns
retp <- rnorm(1e6)
# Compare diff_vec() with rutils::diffit()
all.equal(drop(HighFreq::diff_vec(retp, lagg=3, pad=TRUE)),
  rutils::diffit(retp, lagg=3))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::diff_vec(retp, lagg=3, pad=TRUE),
  Rcode=rutils::diffit(retp, lagg=3),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`encode_it` *Calculate the run length encoding of a single-column* time series*, ma-trix, or a* vector*.*

Description

Calculate the run length encoding of a single-column *time series*, *matrix*, or a *vector*.

Usage

```
encode_it(timeser)
```

Arguments

`timeser`	A single-column *time series*, *matrix*, or a *vector*.

Details

The function `encode_it()` calculates the run length encoding of a single-column *time series*, *matrix*, or a *vector*.

The run length encoding of a *vector* consists of two *vectors*: a *vector* of encoded data (consecutive data values) and of an *integer vector* of the data counts (the number of times the same value repeats in succession).

Run length encoding (RLE) is a data compression algorithm which encodes the data in two *vectors*: the consecutive data values and their counts. If a data value occurs several times in succession then it is recorded only once and its corresponding count is equal to the number of times it occurs. Run-length encoding is different from a contingency table.

Value

A *list* with two *vectors*: a *vector* of encoded data and an *integer vector* of data counts (repeats).

Examples

```
## Not run:
# Create a vector of data
datav <- sample(5, 31, replace=TRUE)
# Calculate the run length encoding of datav
HighFreq::encode_it(datav)

## End(Not run)  # end dontrun
```

hf_data *High frequency data sets*

Description

hf_data.RData is a file containing the datasets:

SPY an xts time series containing 1-minute OHLC bar data for the SPY etf, from 2008-01-02 to 2014-05-19. SPY contains 625,425 rows of data, each row contains a single minute bar.

TLT an xts time series containing 1-minute OHLC bar data for the TLT etf, up to 2014-05-19.

VXX an xts time series containing 1-minute OHLC bar data for the VXX etf, up to 2014-05-19.

Usage

```
data(hf_data)  # not required - data is lazy load
```

Format

Each xts time series contains OHLC data, with each row containing a single minute bar:

Open Open price in the bar

High High price in the bar

Low Low price in the bar

Close Close price in the bar

Volume trading volume in the bar

Source

https://wrds-web.wharton.upenn.edu/wrds/

References

Wharton Research Data Service (WRDS)

Examples

```
# data(hf_data)  # not required - data is lazy load
head(SPY)
chart_Series(x=SPY["2009"])
```

`lagit` *Apply a lag to the rows of a* time series *or a* matrix *using* `RcppArmadillo`.

Description

Apply a lag to the rows of a *time series* or a *matrix* using `RcppArmadillo`.

Usage

```
lagit(timeser, lagg = 1L, pad_zeros = TRUE)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`lagg`	An *integer* equal to the number of periods to lag (the default is `lagg = 1`).
`pad_zeros`	*Boolean* argument: Should the output be padded with zeros? (The default is `pad_zeros = TRUE`.)

Details

The function `lagit()` applies a lag to the input *matrix* by shifting its rows by the number equal to the argument `lagg`. For positive `lagg` values, the rows are shifted *forward* (down), and for negative `lagg` values they are shifted *backward* (up).

The output *matrix* is padded with either zeros (the default), or with rows of data from `timeser`, so that it has the same dimensions as `timeser`. If the `lagg` is positive, then the first row is copied and added upfront. If the `lagg` is negative, then the last row is copied and added to the end.

As a rule, if `timeser` contains returns data, then the output *matrix* should be padded with zeros, to avoid data snooping. If `timeser` contains prices, then the output *matrix* should be padded with the prices.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Create a matrix of random returns
retp <- matrix(rnorm(5e6), nc=5)
# Compare lagit() with rutils::lagit()
all.equal(HighFreq::lagit(retp), rutils::lagit(retp))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::lagit(retp),
  Rcode=rutils::lagit(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`lag_vec` *Apply a lag to a single-column* time series *or a* vector *using* `RcppArmadillo`.

Description

Apply a lag to a single-column *time series* or a *vector* using `RcppArmadillo`.

Usage

```
lag_vec(timeser, lagg = 1L, pad_zeros = TRUE)
```

Arguments

`timeser`	A single-column *time series* or a *vector*.
`lagg`	An *integer* equal to the number of periods to lag. (The default is `lagg = 1`.)
`pad_zeros`	*Boolean* argument: Should the output be padded with zeros? (The default is `pad_zeros = TRUE`.)

Details

The function `lag_vec()` applies a lag to the input *time series* `timeser` by shifting its elements by the number equal to the argument `lagg`. For positive `lagg` values, the elements are shifted forward in time (down), and for negative `lagg` values they are shifted backward (up).

The output *vector* is padded with either zeros (the default), or with data from `timeser`, so that it has the same number of element as `timeser`. If the `lagg` is positive, then the first element is copied and added upfront. If the `lagg` is negative, then the last element is copied and added to the end.

As a rule, if `timeser` contains returns data, then the output *matrix* should be padded with zeros, to avoid data snooping. If `timeser` contains prices, then the output *matrix* should be padded with the prices.

Value

A column *vector* with the same number of elements as the input time series.

Examples

```
## Not run:
# Create a vector of random returns
retp <- rnorm(1e6)
# Compare lag_vec() with rutils::lagit()
all.equal(drop(HighFreq::lag_vec(retp)),
  rutils::lagit(retp))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::lag_vec(retp),
  Rcode=rutils::lagit(retp),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`lik_garch` *Calculate the log-likelihood of a time series of returns assuming a* GARCH(1,1) *process.*

Description

Calculate the log-likelihood of a time series of returns assuming a *GARCH(1,1)* process.

Usage

```
lik_garch(omegac, alphac, betac, returns, minval = 1e-06)
```

Arguments

`omegac`	Parameter proportional to the long-term average level of variance.
`alphac`	The weight associated with recent realized variance updates.
`betac`	The weight associated with the past variance estimates.
`returns`	A single-column *matrix* of returns.
`minval`	The floor value applied to the variance, to avoid zero values. (The default is `minval = 0.000001`.)

Details

The function `lik_garch()` calculates the log-likelihood of a time series of returns assuming a *GARCH(1,1)* process.

It first estimates the rolling variance of the `returns` argument using function `sim_garch()`:

$$\sigma^2_i = \omega + \alpha r^2_i + \beta \sigma^2_{i-1}$$

Where r_i is the time series of returns, and σ^2_i is the estimated rolling variance. And ω, α, and β are the *GARCH* parameters. It applies the floor value `minval` to the variance, to avoid zero values. So the minimum value of the variance is equal to `minval`.

The function `lik_garch()` calculates the log-likelihood assuming a normal distribution of returns conditional on the variance σ^2_{i-1} in the previous period, as follows:

$$likelihood = - \sum_{i=1}^n (\frac{r^2_i}{\sigma^2_{i-1}} + \log(\sigma^2_{i-1}))$$

Value

The log-likelihood value.

Examples

```
## Not run:
# Define the GARCH model parameters
alphac <- 0.79
betac <- 0.2
omegac <- 1e-4*(1-alphac-betac)
# Calculate historical VTI returns
```

```
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the log-likelihood of VTI returns assuming GARCH(1,1)
HighFreq::lik_garch(omegac=omegac, alphac=alphac,  betac=betac, returns=retp)

## End(Not run)  # end dontrun
```

`mult_mat` *Multiply element-wise the rows or columns of a* matrix *times a* vector.

Description

Multiply element-wise the rows or columns of a *matrix* times a *vector*.

Usage

```
mult_mat(vectorv, matrixv, byrow = TRUE)
```

Arguments

`vector`	A *numeric vector*.
`matrix`	A *numeric matrix*.
`byrow`	A *Boolean* argument: if `TRUE` then multiply the rows of `matrix` by `vector`, otherwise multiply the columns (the default is `byrow = TRUE`.)

Details

The function `mult_mat()` multiplies element-wise the rows or columns of a *matrix* times a *vector*.

If `byrow = TRUE` (the default), then function `mult_mat()` multiplies the rows of the argument `matrix` times the argument `vector`. Otherwise it multiplies the columns of `matrix`.

In `R`, *matrix* multiplication is performed by columns. Performing multiplication by rows is often required, for example when multiplying asset returns by portfolio weights. But performing multiplication by rows requires explicit loops in `R`, or it requires *matrix* transpose. And both are slow.

The function `mult_mat()` uses `RcppArmadillo` `C++` code, so when multiplying large *matrix* columns it's several times faster than vectorized `R` code, and it's even much faster compared to `R` when multiplying the *matrix* rows.

The function `mult_mat()` performs loops over the *matrix* rows and columns using the `Armadillo` operators `each_row()` and `each_col()`, instead of performing explicit `for()` loops (both methods are equally fast).

Value

A *matrix* equal to the product of the elements of `matrix` times `vector`, with the same dimensions as the argument `matrix`.

Examples

```
## Not run:
# Create vector and matrix data
matrixv <- matrix(round(runif(25e4), 2), nc=5e2)
vectorv <- round(runif(5e2), 2)

# Multiply the matrix rows using R
matrixr <- t(vectorv*t(matrixv))
# Multiply the matrix rows using C++
matrixp <- HighFreq::mult_mat(vectorv, matrixv, byrow=TRUE)
all.equal(matrixr, matrixp)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_mat(vectorv, matrixv, byrow=TRUE),
    Rcode=t(vectorv*t(matrixv)),
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

# Multiply the matrix columns using R
matrixr <- vectorv*matrixv
# Multiply the matrix columns using C++
matrixp <- HighFreq::mult_mat(vectorv, matrixv, byrow=FALSE)
all.equal(matrixr, matrixp)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_mat(vectorv, matrixv, byrow=FALSE),
    Rcode=vectorv*matrixv,
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`mult_mat_ref` *Multiply the rows or columns of a* matrix *times a* vector*, element-wise and in place, without copying the data in memory.*

Description

Multiply the rows or columns of a *matrix* times a *vector*, element-wise and in place, without copying the data in memory.

Usage

```
mult_mat_ref(vectorv, matrixv, byrow = TRUE)
```

Arguments

vectorv	A *numeric vector.*
matrixv	A *numeric matrix.*
byrow	A *Boolean* argument: if TRUE then multiply the rows of matrixv by vectorv, otherwise multiply the columns (the default is byrow = TRUE.)

Details

The function `mult_mat_ref()` multiplies the rows or columns of a *matrix* times a *vector*, element-wise and in place, without copying the data in memory.

It accepts a *pointer* to the argument `matrixv`, and it overwrites the old `matrix` values with the new values. It performs the calculation in place, without copying the *matrix* in memory, which can significantly increase the computation speed for large matrices.

If `byrow = TRUE` (the default), then function `mult_mat_ref()` multiplies the rows of the argument `matrixv` times the argument `vectorv`. Otherwise it multiplies the columns of `matrixv`.

In `R`, *matrix* multiplication is performed by columns. Performing multiplication by rows is often required, for example when multiplying asset returns by portfolio weights. But performing multiplication by rows requires explicit loops in `R`, or it requires *matrix* transpose. And both are slow.

The function `mult_mat_ref()` uses `RcppArmadillo` `C++` code, so when multiplying large *matrix* columns it's several times faster than vectorized `R` code, and it's even much faster compared to `R` when multiplying the *matrix* rows.

The function `mult_mat_ref()` performs loops over the *matrix* rows and columns using the `Armadillo` operators `each_row()` and `each_col()`, instead of performing explicit `for()` loops (both methods are equally fast).

Value

Void (no return value - modifies the data in place).

Examples

```
## Not run:
# Create vector and matrix data
matrixv <- matrix(round(runif(25e4), 2), nc=5e2)
vectorv <- round(runif(5e2), 2)

# Multiply the matrix rows using R
matrixr <- t(vectorv*t(matrixv))
# Multiply the matrix rows using C++
HighFreq::mult_mat_ref(vectorv, matrixv, byrow=TRUE)
all.equal(matrixr, matrixv)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_mat_ref(vectorv, matrixv, byrow=TRUE),
    Rcode=t(vectorv*t(matrixv)),
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

# Multiply the matrix columns using R
matrixr <- vectorv*matrixv
# Multiply the matrix columns using C++
HighFreq::mult_mat_ref(vectorv, matrixv, byrow=FALSE)
all.equal(matrixr, matrixv)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_mat_ref(vectorv, matrixv, byrow=FALSE),
    Rcode=vectorv*matrixv,
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
## End(Not run)  # end dontrun
```

`ohlc_returns` *Calculate single period percentage returns from either* TAQ *or* OHLC *prices.*

Description

Calculate single period percentage returns from either *TAQ* or *OHLC* prices.

Usage

```
ohlc_returns(xtsv, lagg = 1, colnum = 4, scalit = TRUE)
```

Arguments

`xtsv`	An *xts* time series of either *TAQ* or *OHLC* data.
`lagg`	An integer equal to the number of time periods of lag. (default is 1)
`colnum`	The column number to extract from the *OHLC* data. (default is `4`, or the *Close* prices column)
`scalit`	*Boolean* argument: should the returns be divided by the number of seconds in each period? (default is `TRUE`)

Details

The function `ohlc_returns()` calculates the percentage returns for either *TAQ* or *OHLC* data, defined as the difference of log prices. Multi-period returns can be calculated by setting the `lag` parameter to values greater than `1` (the default).

If `scalit` is `TRUE` (the default), then the returns are divided by the differences of the time index (which scales the returns to units of returns per second.)

The time index of the `xtsv` time series is assumed to be in *POSIXct* format, so that its internal value is equal to the number of seconds that have elapsed since the *epoch*.

If `scalit` is `TRUE` (the default), then the returns are expressed in the scale of the time index of the `xtsv` time series. For example, if the time index is in seconds, then the returns are given in units of returns per second. If the time index is in days, then the returns are equal to the returns per day.

The function `ohlc_returns()` identifies the `xtsv` time series as *TAQ* data when it has six columns, otherwise assumes it's *OHLC* data. By default, for *OHLC* data, it differences the *Close* prices, but can also difference other prices depending on the value of `colnum`.

Value

A single-column *xts* time series of returns.

Examples

```
# Calculate secondly returns from TAQ data
returns <- HighFreq::ohlc_returns(xtsv=HighFreq::SPY_TAQ)
# Calculate close to close returns
returns <- HighFreq::ohlc_returns(xtsv=HighFreq::SPY)
# Calculate open to open returns
returns <- HighFreq::ohlc_returns(xtsv=HighFreq::SPY, colnum=1)
```

---

ohlc_sharpe — *Calculate time series of point Sharpe-like statistics for each row of a* OHLC *time series.*

Description

Calculate time series of point Sharpe-like statistics for each row of a *OHLC* time series.

Usage

```
ohlc_sharpe(ohlc, method = "close")
```

Arguments

ohlc	An *OHLC* time series of prices in *xts* format.
method	A *character* string representing method for estimating the Sharpe-like exponent.

Details

The function `ohlc_sharpe()` calculates Sharpe-like statistics for each row of a *OHLC* time series. The Sharpe-like statistic is defined as the ratio of the difference between *Close* minus *Open* prices divided by the difference between *High* minus *Low* prices. This statistic may also be interpreted as something like a *Hurst exponent* for a single row of data. The motivation for the Sharpe-like statistic is the notion that if prices are trending in the same direction inside a given time bar of data, then this statistic is close to either 1 or -1.

Value

An *xts* time series with the same number of rows as the argument `ohlc`.

Examples

```
# Calculate time series of running Sharpe ratios for SPY
sharpe_running <- ohlc_sharpe(HighFreq::SPY)
```

ohlc_skew

Calculate time series of point skew estimates from a OHLC *time series, assuming zero drift.*

Description

Calculate time series of point skew estimates from a *OHLC* time series, assuming zero drift.

Usage

```
ohlc_skew(ohlc, method = "rogers_satchell")
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`method`	A *character* string representing method for estimating skew.

Details

The function `ohlc_skew()` calculates a time series of skew estimates from *OHLC* prices, one for each row of *OHLC* data. The skew estimates are expressed in the time scale of the index of the *OHLC* time series. For example, if the time index is in seconds, then the skew is given in units of skew per second. If the time index is in days, then the skew is equal to the skew per day.

Currently only the `"close"` skew estimation method is correct (assuming zero drift), while the `"rogers_satchell"` method produces a skew-like indicator, proportional to the skew. The default method is `"rogers_satchell"`.

Value

A time series of point skew estimates.

Examples

```
# Calculate time series of skew estimates for SPY
skew <- HighFreq::ohlc_skew(HighFreq::SPY)
```

ohlc_variance

Calculate a time series of point estimates of variance for an OHLC *time series, using different range estimators for variance.*

Description

Calculates the point variance estimates from individual rows of *OHLC* prices (rows of data), using the squared differences of *OHLC* prices at each point in time, without averaging them over time.

Usage

```
ohlc_variance(ohlc, method = "yang_zhang", scalit = TRUE)
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`method`	A *character* string representing the method for estimating variance. The methods include: • "close" close to close, • "garman_klass" Garman-Klass, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "rogers_satchell" Rogers-Satchell, • "yang_zhang" Yang-Zhang, (default is `"yang_zhang"`)
`scalit`	*Boolean* argument: should the returns be divided by the number of seconds in each period? (default is `TRUE`)

Details

The function `ohlc_variance()` calculates a time series of point variance estimates of percentage returns, from *OHLC* prices, without averaging them over time. For example, the method `"close"` simply calculates the squares of the differences of the log *Close* prices.

The other methods calculate the squares of other possible differences of the log *OHLC* prices. This way the point variance estimates only depend on the price differences within individual rows of data (and possibly from the neighboring rows.) All the methods are implemented assuming zero drift, since the calculations are performed only for a single row of data, at a single point in time.

The user can choose from several different variance estimation methods. The methods `"close"`, `"garman_klass_yz"`, and `"yang_zhang"` do account for close-to-open price jumps, while the methods `"garman_klass"` and `"rogers_satchell"` do not account for close-to-open price jumps. The default method is `"yang_zhang"`, which theoretically has the lowest standard error among unbiased estimators.

The point variance estimates can be passed into function `roll_vwap()` to perform averaging, to calculate rolling variance estimates. This is appropriate only for the methods `"garman_klass"` and `"rogers_satchell"`, since they don't require subtracting the rolling mean from the point variance estimates.

The point variance estimates can also be considered to be technical indicators, and can be used as inputs into trading models.

If `scalit` is `TRUE` (the default), then the variance is divided by the squared differences of the time index (which scales the variance to units of variance per second squared.) This is useful for example, when calculating intra-day variance from minutely bar data, because dividing returns by the number of seconds decreases the effect of overnight price jumps.

If `scalit` is `TRUE` (the default), then the variance is expressed in the scale of the time index of the *OHLC* time series. For example, if the time index is in seconds, then the variance is given in units of variance per second squared. If the time index is in days, then the variance is equal to the variance per day squared.

The time index of the `ohlc` time series is assumed to be in *POSIXct* format, so that its internal value is equal to the number of seconds that have elapsed since the *epoch*.

The function `ohlc_variance()` performs similar calculations to the function `volatility()` from package TTR, but it assumes zero drift, and doesn't calculate a running sum using `runSum()`. It's also a little faster because it performs less data validation.

Value

An *xts* time series with a single column and the same number of rows as the argument `ohlc`.

Examples

```
# Create minutely OHLC time series of random prices
ohlc <- HighFreq::random_ohlc()
# Calculate variance estimates for ohlc
var_running <- HighFreq::ohlc_variance(ohlc)
# Calculate variance estimates for SPY
var_running <- HighFreq::ohlc_variance(HighFreq::SPY, method="yang_zhang")
# Calculate SPY variance without overnight jumps
var_running <- HighFreq::ohlc_variance(HighFreq::SPY, method="rogers_satchell")
```

`param_portf` *Create a named list of model parameters that can be passed into portfolio optimization functions.*

Description

Create a named list of model parameters that can be passed into portfolio optimization functions.

Usage

```
param_portf(
  method = "sharpem",
  singmin = 1e-05,
  dimax = 0L,
  confl = 0.1,
  alphac = 0,
  rankw = FALSE,
  centerw = FALSE,
  scalew = "voltarget",
  voltarget = 0.01
)
```

Arguments

`method`	A *character string* specifying the method for calculating the portfolio weights (the default is `method = "sharpem"`).
`singmin`	A *numeric* threshold level for discarding small *singular values* in order to regularize the inverse of the `covariance matrix` of `returns` (the default is `1e-5`).
`dimax`	An *integer* equal to the number of *singular values* used for calculating the *reduced inverse* of the `covariance matrix` of `returns` matrix (the default is `dimax = 0` - standard matrix inverse using all the *singular values*).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).
`alphac`	The shrinkage intensity of `returns` (with values between `0` and `1` - the default is `0`).

rankw	A *Boolean* specifying whether the weights should be ranked (the default is rankw = FALSE).
centerw	A *Boolean* specifying whether the weights should be centered (the default is centerw = FALSE).
scalew	A *character string* specifying the method for scaling the weights (the default is scalew = "voltarget").
voltarget	A *numeric* volatility target for scaling the weights (the default is 0.01)

Details

The function param_portf() creates a named list of model parameters that can be passed into portfolio optimization functions. For example into the functions calc_weights() and roll_portf(). See the function calc_weights() for more details.

The function param_portf() simplifies the creation of portfolio optimization parameter lists. The users can create a parameter list with the default values, or they can specify custom parameter values.

Value

A named list of model parameters that can be passed into portfolio optimization functions.

Examples

```
## Not run:
# Create a default list of portfolio optimization parameters
controll <- HighFreq::param_portf()
unlist(controll)
# Create a custom list of portfolio optimization parameters
controll <- HighFreq::param_portf(method="regular", dimax=4)
unlist(controll)

## End(Not run)  # end dontrun
```

---

param_reg *Create a named list of model parameters that can be passed into regression and machine learning functions.*

Description

Create a named list of model parameters that can be passed into regression and machine learning functions.

Usage

```
param_reg(
  regmod = "least_squares",
  intercept = TRUE,
  singmin = 1e-05,
  dimax = 0L,
  residscale = "none",
  confl = 0.1,
  alphac = 0
)
```

Arguments

`method`	A *character string* specifying the type of regression model (the default is `method = "least_squares"`).
`intercept`	A *Boolean* specifying whether an intercept term should be added to the predictor (the default is `intercept = TRUE`).
`singmin`	A *numeric* threshold level for discarding small *singular values* in order to regularize the inverse of the predictor matrix (the default is `1e-5`).
`dimax`	An *integer* equal to the number of *singular values* used for calculating the *reduced inverse* of the predictor matrix (the default is `dimax = 0` - standard matrix inverse using all the *singular values*).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).
`alphac`	The shrinkage intensity of `returns` (with values between `0` and `1` - the default is `0`).

Details

The function `param_reg()` creates a named list of model parameters that can be passed into regression and machine learning functions. For example into the functions `calc_reg()` and `roll_reg()`.

The function `param_reg()` simplifies the creation of regression parameter lists. The users can create a parameter list with the default values, or they can specify custom parameter values.

Value

A named list of model parameters that can be passed into regression and machine learning functions.

Examples

```
## Not run:
# Create a default list of regression parameters
controll <- HighFreq::param_reg()
unlist(controll)
# Create a custom list of regression parameters
controll <- HighFreq::param_reg(intercept=FALSE, method="regular", dimax=4)
unlist(controll)

## End(Not run)  # end dontrun
```

push_cov2cor *Calculate the correlation matrix from the covariance matrix.*

Description

Calculate the correlation matrix from the covariance matrix.

Usage

```
push_cov2cor(covmat)
```

Arguments

covmat	A *matrix* of covariances.

Details

The function push_cov2cor() calculates the correlation matrix from the covariance matrix, in place, without copying the data in memory.

The function push_cov2cor() accepts a *pointer* to the covariance matrix, and it overwrites it with the correlation matrix.

The function push_cov2cor() is written in RcppArmadillo C++ so it's much faster than R code.

Value

Void (no return value - modifies the covariance matrix in place).

Examples

```
## Not run:
# Calculate a time series of returns
retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI", "DBC")])
# Calculate the covariance matrix of returns
covmat <- cov(retp)
# Calculate the correlation matrix of returns
push_cov2cor(covmat)
all.equal(covmat, cor(retp))

## End(Not run)  # end dontrun
```

push_covar *Update the trailing covariance matrix of streaming asset returns, with a row of new returns using an online recursive formula.*

Description

Update the trailing covariance matrix of streaming asset returns, with a row of new returns using an online recursive formula.

Usage

```
push_covar(retsn, covmat, meanv, lambdacov)
```

Arguments

`retsn`	A *vector* of new asset returns.
`covmat`	A trailing covariance *matrix* of asset returns.
`meanv`	A *vector* of trailing means of asset returns.
`lambdacov`	A decay factor which multiplies the past covariance.

Details

The function `push_covar()` updates the trailing covariance matrix of streaming asset returns, with a row of new returns. It updates the covariance matrix in place, without copying the data in memory.

The streaming asset returns r_t contain multiple columns and the parameter `retsn` represents a single row of r_t - the asset returns at time t. The elements of the vectors `retsn` and `meanv` represent single rows of data with multiple columns.

The function `push_covar()` accepts *pointers* to the arguments `covmat` and `meanv`, and it overwrites the old values with the new values. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large matrices.

First, the function `push_covar()` updates the trailing means $\bar{r}_t$ of the streaming asset returns r_t by recursively weighting present and past values using the decay factor λ:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

This recursive formula is equivalent to the exponentially weighted moving average of the streaming asset returns r_t.

It then calculates the demeaned returns:

$$\hat{r}_t = r_t - \bar{r}_t$$

Finally, it updates the trailing covariance matrix of the returns:

$$cov_t = \lambda^2 cov_{t-1} + (1 - \lambda^2) \hat{r}^T_t \hat{r}_t$$

The decay factor λ determines the strength of the updates, with smaller λ values giving more weight to the new data. If the asset returns are not stationary, then applying more weight to the new returns reduces the bias of the trailing covariance matrix, but it also increases its variance. Simulation can be used to find the value of the λ parameter to achieve the best bias-variance tradeoff.

The function `push_covar()` is written in `RcppArmadillo` `C++` so it's much faster than `R` code.

Value

Void (no return value - modifies the trailing covariance matrix and the return means in place).

Examples

```
## Not run:
# Calculate a time series of returns
retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI", "DBC")])
# Calculate the returns without last row
nrows <- NROW(retp)
retss <- retp[-nrows]
# Calculate the covariance of returns
meanv <- colMeans(retss)
covmat <- cov(retss)
# Update the covariance of returns
HighFreq::push_covar(retsn=retp[nrows], covmat=covmat, meanv=meanv, lambdacov=0.9)

## End(Not run)  # end dontrun
```

`push_eigen` *Update the trailing eigen values and eigen vectors of streaming asset return data, with a row of new returns.*

Description

Update the trailing eigen values and eigen vectors of streaming asset return data, with a row of new returns.

Usage

```
push_eigen(retsn, covmat, eigenval, eigenvec, eigenret, meanv, lambdacov)
```

Arguments

`retsn`	A *vector* of new asset returns.
`covmat`	A trailing covariance *matrix* of asset returns.
`eigenval`	A *vector* of eigen values.
`eigenvec`	A *matrix* of eigen vectors.
`eigenret`	A *vector* of eigen portfolio returns.
`meanv`	A *vector* of trailing means of asset returns.
`lambdacov`	A decay factor which multiplies the past covariance.

Details

The function `push_eigen()` updates the trailing eigen values, eigen vectors, and the eigen portfolio returns of streaming asset returns, with a row of new data. It updates the eigenelements in place, without copying the data in memory.

The streaming asset returns r_t contain multiple columns and the parameter `retsn` represents a single row of r_t - the asset returns at time t. The elements of the vectors `retsn`, `eigenret`, and `meanv` represent single rows of data with multiple columns.

The function push_eigen() accepts *pointers* to the arguments eigenval, eigenval, eigenvec, meanv, and eigenret, and it overwrites the old values with the new values. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large matrices.

First, the function push_eigen() calls the function HighFreq::push_covar() to update the trailing covariance matrix of streaming asset returns, with a row of new returns. It updates the covariance matrix in place, without copying the data in memory.

It then calls the Armadillo function arma::eig_sym to calculate the eigen decomposition of the trailing covariance matrix.

The function push_eigen() calculates the eigen portfolio returns by multiplying the scaled asset returns times the eigen vectors v_{t-1}:

$$r^{eigen}_t = v_{t-1} \frac{r_t}{\sigma_{t-1}}$$

Where v_{t-1} is the matrix of previous eigen vectors that are passed by reference through the parameter eigenvec. The eigen returns r^{eigen}_t are the returns of the eigen portfolios, with weights equal to the eigen vectors v_{t-1}. The eigen weights are applied to the asset returns scaled by their volatilities. The eigen returns r^{eigen}_t are passed by reference through the parameter eigenret.

The decay factor λ determines the strength of the updates, with smaller λ values giving more weight to the new data. If the asset returns are not stationary, then applying more weight to the new returns reduces the bias of the trailing covariance matrix, but it also increases its variance. Simulation can be used to find the value of the λ parameter to achieve the best bias-variance tradeoff.

The function push_eigen() is written in RcppArmadillo C++ so it's much faster than R code.

Value

Void (no return value - modifies the trailing eigen values, eigen vectors, the eigen portfolio returns, and the return means in place).

Examples

```
## Not run:
# Calculate a time series of returns
retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI", "DBC")])
# Calculate the returns without the last row
nrows <- NROW(retp)
retss <- retp[-nrows]
# Calculate the previous covariance of returns
meanv <- colMeans(retss)
covmat <- cov(retss)
# Update the covariance of returns
eigenret <- numeric(NCOL(retp))
HighFreq::push_eigen(retsn=retp[nrows], covmat=covmat,
  eigenval=eigenval, eigenvec=eigenvec,
  eigenret=eigenret, meanv=meanv, lambdacov=0.9)

## End(Not run)  # end dontrun
```

`push_sga` *Update the trailing eigen values and eigen vectors of streaming asset return data, with a row of new returns, using the* SGA *algorithm.*

Description

Update the trailing eigen values and eigen vectors of streaming asset return data, with a row of new returns, using the *SGA* algorithm.

Usage

```
push_sga(retsn, eigenval, eigenvec, eigenret, meanv, varv, lambdaf, gamma)
```

Arguments

`retsn`	A *vector* of new asset returns.
`eigenval`	A *vector* of eigen values.
`eigenvec`	A *matrix* of eigen vectors.
`eigenret`	A *vector* of eigen portfolio returns.
`meanv`	A *vector* of trailing means of asset returns.
`varv`	A *vector* of the trailing asset variances.
`lambdaf`	A decay factor which multiplies the past mean and variance.
`gamma`	A *numeric* gain factor which multiplies the past eigenelements.

Details

The function `push_sga()` updates the trailing eigen values, eigen vectors, and the eigen portfolio returns of streaming asset returns, with a row of new data, using the *SGA* algorithm. It updates the eigenelements in place, without copying the data in memory.

The streaming asset returns r_t contain multiple columns and the parameter `retsn` represents a single row of r_t - the asset returns at time t. The elements of the vectors `retsn`, `meanv`, and `varv` represent single rows of data with multiple columns.

The function `push_sga()` accepts *pointers* to the arguments `eigenval`, `eigenvec`, `meanv`, and `varv`, and it overwrites the old values with the new values. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large matrices.

First, the function `push_sga()` updates the trailing means $\bar{r}_t$ and variances σ^2_t of the streaming asset returns r_t by recursively weighting present and past values using the decay factor λ:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2)(r_t - \bar{r}_t)^2$$

The past values $\bar{r}_{t-1}$ and σ^2_{t-1} are passed in by reference through the variables `meanv` and `varv`. The updated values are then passed out by reference.

These recursive formulas are equivalent to the exponentially weighted moving averages of the streaming asset returns r_t.

It then calculates a vector of the eigen portfolio returns:

$$r_t^{eigen} = v_{t-1} \frac{r_t}{\sigma_{t-1}}$$

Where v_{t-1} is the matrix of previous eigen vectors that are passed by reference through the parameter `eigenvec`. The eigen returns r_t^{eigen} are the returns of the eigen portfolios, with weights equal to the eigen vectors v_{t-1}. The eigen weights are applied to the asset returns scaled by their volatilities. The eigen returns r_t^{eigen} are passed by reference through the parameter `eigenret`.

The function `push_sga()` then standardizes the columns of the new returns:

$$\hat{r}_t = \frac{r_t - \bar{r}_t}{\sigma_t}$$

Finally, the vector of eigen values $\Lambda_{j,t}$ and the matrix of eigen vectors $v_{j,t}$ (j is the column index) are then updated using the *SGA* algorithm:

$$\Lambda_{j,t} = (1 - \gamma)\Lambda_{j,t-1} + \gamma\phi_{j,t-1}$$

$$v_{j,t} = v_{j,t-1} + \gamma\phi_{j,t-1}(\hat{r}_t - \phi_{j,t-1}v_{j,t-1} - 2\sum_{i=1}^{j-1} \phi_{i,t-1}v_{i,t-1})$$

Where $\phi_{j,t-1} = \hat{r}_t v_{j,t-1}$ are the matrix products of the new data times the previous eigen vectors.

The gain factor γ determines the strength of the updates, with larger γ values giving more weight to the new data. If the asset returns are not stationary, then applying more weight to the new returns reduces the bias of the trailing eigen vectors, but it also increases their variance. Simulation can be used to find the value of the γ parameter to achieve the best bias-variance tradeoff.

A description of the *SGA* algorithm can be found in the package onlinePCA and in the Online PCA paper.

The function `push_sga()` is written in `RcppArmadillo` `C++` code and it calls the `Armadillo` function `arma::qr_econ()` to perform the QR decomposition, to calculate the eigen vectors.

Value

Void (no return value - modifies the trailing eigen values, eigen vectors, the return means, and the return variances in place).

Examples

```
## Not run:
# Calculate a time series of returns
retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI", "DBC")])
# Calculate the covariance of returns without the last row
nrows <- NROW(retp)
retss <- retp[-nrows]
HighFreq::calc_scale(retss)
meanv <- colMeans(retss)
varv <- sapply(retss, var)
covmat <- cov(retss)
ncols <- NCOL(retss)
# Calculate the eigen decomposition using RcppArmadillo
eigenval <- numeric(ncols) # Allocate eigen values
eigenvec <- matrix(numeric(ncols^2), nc=ncols) # Allocate eigen vectors
HighFreq::calc_eigen(covmat, eigenval, eigenvec)
# Update the eigen decomposition using SGA
```

```
eigenret <- numeric(NCOL(retp))
HighFreq::push_sga(retsn=retp[nrows],
  eigenval=eigenval, eigenvec=eigenvec,
  eigenret=eigenret, meanv=meanv, varv=varv, lambdaf=0.9, gamma=0.1)

## End(Not run)  # end dontrun
```

`random_ohlc` *Calculate a random* OHLC *time series of prices and trading volumes, in* xts *format.*

Description

Calculate a random *OHLC* time series either by simulating random prices following geometric Brownian motion, or by randomly sampling from an input time series.

Usage

```
random_ohlc(
  ohlc = NULL,
  reducit = TRUE,
  volat = 6.5e-05,
  drift = 0,
  datev = seq(from = as.POSIXct(paste(Sys.Date() - 3, "09:30:00")), to =
    as.POSIXct(paste(Sys.Date() - 1, "16:00:00")), by = "1 sec"),
  ...
)
```

Arguments

`ohlc`	An *OHLC* time series of prices and trading volumes, in *xts* format (default is *NULL*).
`volat`	The volatility per period of the `datev` time index (default is `6.5e-05` per second, or about `0.01=1.0%` per day).
`drift`	The drift per period of the `datev` time index (default is 0.0).
`datev`	The time index for the *OHLC* time series.
`reducit`	*Boolean* argument: should `ohlc` time series be transformed to reduced form? (default is `TRUE`)

Details

If the input `ohlc` time series is *NULL* (the default), then the function `random_ohlc()` simulates a minutely *OHLC* time series of random prices following geometric Brownian motion, over the two previous calendar days.

If the input `ohlc` time series is not *NULL*, then the rows of `ohlc` are randomly sampled, to produce a random time series.

If `reducit` is `TRUE` (the default), then the `ohlc` time series is first transformed to reduced form, then randomly sampled, and finally converted to standard form.

Note: randomly sampling from an intraday time series over multiple days will cause the overnight price jumps to be re-arranged into intraday price jumps. This will cause moment estimates to become inflated compared to the original time series.

Value

An *xts* time series with the same dimensions and the same time index as the input `ohlc` time series.

Examples

```
# Create minutely synthetic OHLC time series of random prices
ohlc <- HighFreq::random_ohlc()
# Create random time series from SPY by randomly sampling it
ohlc <- HighFreq::random_ohlc(ohlc=HighFreq::SPY["2012-02-13/2012-02-15"])
```

`random_taq` *Calculate a random* TAQ *time series of prices and trading volumes, in* xts *format.*

Description

Calculate a *TAQ* time series of random prices following geometric Brownian motion, combined with random trading volumes.

Usage

```
random_taq(
  volat = 6.5e-05,
  drift = 0,
  datev = seq(from = as.POSIXct(paste(Sys.Date() - 3, "09:30:00")), to =
    as.POSIXct(paste(Sys.Date() - 1, "16:00:00")), by = "1 sec"),
  bidask = 0.001,
  ...
)
```

Arguments

`bidask`	The bid-ask spread expressed as a fraction of the prices (default is 0.001=10bps).
`volat`	The volatility per period of the `datev` time index (default is `6.5e-05` per second, or about `0.01=1.0%` per day).
`drift`	The drift per period of the `datev` time index (default is 0.0).
`datev`	The time index for the *TAQ* time series.

Details

The function `random_taq()` calculates an *xts* time series with four columns containing random prices following geometric Brownian motion: the bid, ask, and trade prices, combined with random trade volume data. If `datev` isn't supplied as an argument, then by default it's equal to the secondly index over the two previous calendar days.

Value

An *xts* time series, with time index equal to the input `datev` time index, and with four columns containing the bid, ask, and trade prices, and the trade volume.

Examples

```
# Create secondly TAQ time series of random prices
taq <- HighFreq::random_taq()
# Create random TAQ time series from SPY index
taq <- HighFreq::random_taq(datev=index(HighFreq::SPY["2012-02-13/2012-02-15"]))
```

`remove_jumps` *Remove overnight close-to-open price jumps from an* OHLC *time series, by adding adjustment terms to its prices.*

Description

Remove overnight close-to-open price jumps from an *OHLC* time series, by adding adjustment terms to its prices.

Usage

```
remove_jumps(ohlc)
```

Arguments

`ohlc`	An *OHLC* time series of prices and trading volumes, in *xts* format.

Details

The function `remove_jumps()` removes the overnight close-to-open price jumps from an *OHLC* time series, by adjusting its prices so that the first *Open* price of the day is equal to the last *Close* price of the previous day.

The function `remove_jumps()` adds adjustment terms to all the *OHLC* prices, so that intra-day returns and volatilities are not affected.

The function `remove_jumps()` identifies overnight periods as those that are greater than 60 seconds. This assumes that intra-day periods between neighboring rows of data are 60 seconds or less.

The time index of the `ohlc` time series is assumed to be in *POSIXct* format, so that its internal value is equal to the number of seconds that have elapsed since the *epoch*.

Value

An *OHLC* time series with the same dimensions and the same time index as the input `ohlc` time series.

Examples

```
# Remove overnight close-to-open price jumps from SPY data
ohlc <- remove_jumps(HighFreq::SPY)
```

roll_apply *Apply an aggregation function over a rolling look-back interval and the end points of an* OHLC *time series, using* R *code.*

Description

Apply an aggregation function over a rolling look-back interval and the end points of an *OHLC* time series, using R code.

Usage

```
roll_apply(
  xtsv,
  agg_fun,
  lookb = 2,
  endpoints = seq_along(xtsv),
  by_columns = FALSE,
  out_xts = TRUE,
  ...
)
```

Arguments

`...`	additional parameters to the function `agg_fun`.
`xtsv`	An *OHLC* time series of prices and trading volumes, in *xts* format.
`agg_fun`	The name of the aggregation function to be applied over a rolling look-back interval.
`lookb`	The number of end points in the look-back interval used for applying the aggregation function (including the current row).
`by_columns`	*Boolean* argument: should the function `agg_fun()` be applied column-wise (individually), or should it be applied to all the columns combined? (default is `FALSE`)
`out_xts`	*Boolean* argument: should the output be coerced into an *xts* series? (default is `TRUE`)
`endpoints`	An integer vector of end points.

Details

The function `roll_apply()` applies an aggregation function over a rolling look-back interval attached at the end points of an *OHLC* time series.

The function `roll_apply()` is implemented in R code.

`HighFreq::roll_apply()` performs similar operations to the functions `rollapply()` and `period.apply()` from package xts, and also the function `apply.rolling()` from package PerformanceAnalytics. (The function `rollapply()` isn't exported from the package xts.)

But `HighFreq::roll_apply()` is faster because it performs less type-checking and skips other overhead. Unlike the other functions, `roll_apply()` doesn't produce any leading *NA* values.

The function `roll_apply()` can be called in two different ways, depending on the argument `endpoints`. If the argument `endpoints` isn't explicitly passed to `roll_apply()`, then the default value is used, and `roll_apply()` performs aggregations over overlapping intervals at each point in time.

If the argument endpoints is explicitly passed to roll_apply(), then roll_apply() performs aggregations over intervals attached at the endpoints. If lookb=2 then the aggregations are performed over non-overlapping intervals, otherwise they are performed over overlapping intervals.

If the argument out_xts is TRUE (the default) then the output is coerced into an *xts* series, with the number of rows equal to the length of argument endpoints. Otherwise a list is returned, with the length equal to the length of argument endpoints.

If out_xts is TRUE and the aggregation function agg_fun() returns a single value, then roll_apply() returns an *xts* time series with a single column. If out_xts is TRUE and if agg_fun() returns a vector of values, then roll_apply() returns an *xts* time series with multiple columns, equal to the length of the vector returned by the aggregation function agg_fun().

Value

Either an *xts* time series with the number of rows equal to the length of argument endpoints, or a list the length of argument endpoints.

Examples

```
# extract a single day of SPY data
ohlc <- HighFreq::SPY["2012-02-13"]
interval <- 11  # number of data points between end points
lookb <- 4  # number of end points in look-back interval
# Calculate the rolling sums of ohlc columns over a rolling look-back interval
agg_regations <- roll_apply(ohlc, agg_fun=sum, lookb=lookb, by_columns=TRUE)
# Apply a vector-valued aggregation function over a rolling look-back interval
agg_function <- function(ohlc)  c(max(ohlc[, 2]), min(ohlc[, 3]))
agg_regations <- roll_apply(ohlc, agg_fun=agg_function, lookb=lookb)
# Define end points at 11-minute intervals (HighFreq::SPY is minutely bars)
endpoints <- rutils::endpoints(ohlc, interval=interval)
# Calculate the sums of ohlc columns over endpoints using non-overlapping intervals
agg_regations <- roll_apply(ohlc, agg_fun=sum, endpoints=endpoints, by_columns=TRUE)
# Apply a vector-valued aggregation function over the endpoints of ohlc
# using overlapping intervals
agg_regations <- roll_apply(ohlc, agg_fun=agg_function,
                            lookb=5, endpoints=endpoints)
```

---

roll_backtest *Perform a backtest simulation of a trading strategy (model) over a vector of end points along a time series of prices.*

Description

Perform a backtest simulation of a trading strategy (model) over a vector of end points along a time series of prices.

Usage

```
roll_backtest(
  xtsv,
  train_func,
  trade_func,
  lookb = look_forward,
  look_forward,
  endpoints = rutils::calc_endpoints(xtsv, look_forward),
  ...
)
```

Arguments

`...`	additional parameters to the functions `train_func()` and `trade_func()`.
`xtsv`	A time series of prices, asset returns, trading volumes, and other data, in *xts* format.
`train_func`	The name of the function for training (calibrating) a forecasting model, to be applied over a rolling look-back interval.
`trade_func`	The name of the trading model function, to be applied over a rolling look-forward interval.
`lookb`	The size of the look-back interval, equal to the number of rows of data used for training the forecasting model.
`look_forward`	The size of the look-forward interval, equal to the number of rows of data used for trading the strategy.
`endpoints`	A vector of end points along the rows of the `xtsv` time series, given as either integers or dates.

Details

The function `roll_backtest()` performs a rolling backtest simulation of a trading strategy over a vector of end points. At each end point, it trains (calibrates) a forecasting model using past data taken from the `xtsv` time series over the look-back interval, and applies the forecasts to the `trade_func()` trading model, using out-of-sample future data from the look-forward interval.

The function `trade_func()` should simulate the trading model, and it should return a named list with at least two elements: a named vector of performance statistics, and an *xts* time series of out-of-sample returns. The list returned by `trade_func()` can also have additional elements, like the in-sample calibrated model statistics, etc.

The function `roll_backtest()` returns a named list containing the listv returned by function `trade_func()`. The list names are equal to the *endpoints* dates. The number of list elements is equal to the number of *endpoints* minus two (because the first and last end points can't be included in the backtest).

Value

An *xts* time series with the number of rows equal to the number of end points minus two.

Examples

```
## Not run:
# Combine two time series of prices
prices <- cbind(rutils::etfenv$XLU, rutils::etfenv$XLP)
lookb <- 252
look_forward <- 22
# Define end points
endpoints <- rutils::calc_endpoints(prices, look_forward)
# Perform back-test
back_test <- roll_backtest(endpoints=endpoints,
    look_forward=look_forward,
    lookb=lookb,
    train_func = train_model,
    trade_func = trade_model,
    model_params = model_params,
    trading_params = trading_params,
    xtsv=prices)

## End(Not run)
```

roll_conv — *Calculate the rolling convolutions (weighted sums) of a* time series *with a single-column* matrix *of weights.*

Description

Calculate the rolling convolutions (weighted sums) of a *time series* with a single-column *matrix* of weights.

Usage

```
roll_conv(timeser, weightv)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`weightv`	A single-column *matrix* of weights.

Details

The function `roll_conv()` calculates the convolutions of the *matrix* columns with a single-column *matrix* of weights. It performs a loop over the *matrix* rows and multiplies the past (higher) values by the weights. It calculates the rolling weighted sums of the past data.

The function `roll_conv()` uses the `Armadillo` function `arma::conv2()`. It performs a similar calculation to the standard `R` function
`filter(x=timeser, filter=weightv, method="convolution", sides=1)`, but it's over `6` times faster, and it doesn't produce any leading `NA` values.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# First example
# Calculate a time series of returns
retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI")])
# Create simple weights equal to a 1 value plus zeros
weightv <- c(1, rep(0, 10))
# Calculate rolling weighted sums
retf <- HighFreq::roll_conv(retp, weightv)
```

```
# Compare with original
all.equal(coredata(retp), retf, check.attributes=FALSE)
# Second example
# Calculate exponentially decaying weights
weightv <- exp(-0.2*(1:11))
weightv <- weightv/sum(weightv)
# Calculate rolling weighted sums
retf <- HighFreq::roll_conv(retp, weightv)
# Calculate rolling weighted sums using filter()
retc <- filter(x=retp, filter=weightv, method="convolution", sides=1)
# Compare both methods
all.equal(retc[-(1:11), ], retf[-(1:11), ], check.attributes=FALSE)

## End(Not run)  # end dontrun
```

roll_count — *Count the number of consecutive* TRUE *elements in a Boolean vector, and reset the count to zero after every* FALSE *element.*

Description

Count the number of consecutive TRUE elements in a Boolean vector, and reset the count to zero after every FALSE element.

Usage

```
roll_count(timeser)
```

Arguments

timeser	A *Boolean vector* of data.

Details

The function roll_count() calculates the number of consecutive TRUE elements in a Boolean vector, and it resets the count to zero after every FALSE element.

For example, the Boolean vector FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, is translated into 0, 1, 2, 0, 0, 1, 2, 3, 4, 5, 0.

Value

An *integer vector* of the same length as the argument timeser.

Examples

```
## Not run:
# Calculate the number of consecutive TRUE elements
drop(HighFreq::roll_count(c(FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)))

## End(Not run)  # end dontrun
```

roll_hurst — *Calculate a time series of* Hurst *exponents over a rolling look-back interval.*

Description

Calculate a time series of *Hurst* exponents over a rolling look-back interval.

Usage

```
roll_hurst(ohlc, lookb = 11)
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`lookb`	The size of the look-back interval, equal to the number of rows of data used for aggregating the *OHLC* prices.

Details

The function `roll_hurst()` calculates a time series of *Hurst* exponents from *OHLC* prices, over a rolling look-back interval.

The *Hurst* exponent is defined as the logarithm of the ratio of the price range, divided by the standard deviation of returns, and divided by the logarithm of the interval length.

The function `roll_hurst()` doesn't use the same definition as the rescaled range definition of the *Hurst* exponent. First, because the price range is calculated using *High* and *Low* prices, which produces bigger range values, and higher *Hurst* exponent estimates. Second, because the *Hurst* exponent is estimated using a single aggregation interval, instead of multiple intervals in the rescaled range definition.

The rationale for using a different definition of the *Hurst* exponent is that it's designed to be a technical indicator for use as input into trading models, rather than an estimator for statistical analysis.

Value

An *xts* time series with a single column and the same number of rows as the argument `ohlc`.

Examples

```
# Calculate rolling Hurst for SPY in March 2009
hurst_rolling <- roll_hurst(ohlc=HighFreq::SPY["2009-03"], lookb=11)
chart_Series(hurst_rolling["2009-03-10/2009-03-12"], name="SPY hurst_rolling")
```

roll_kurtosis — *Calculate a* matrix *of kurtosis estimates over a rolling look-back interval attached at the end points of a* time series *or a* matrix.

Description

Calculate a *matrix* of kurtosis estimates over a rolling look-back interval attached at the end points of a *time series* or a *matrix*.

Usage

```
roll_kurtosis(
  timeser,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L,
  method = "moment",
  confl = 0.75
)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).
`method`	A *character string* specifying the type of the kurtosis model (the default is `method = "moment"` - see Details).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `roll_kurtosis()` calculates a *matrix* of kurtosis estimates over rolling look-back intervals attached at the end points of the *time series* `timeser`.

The function `roll_kurtosis()` performs a loop over the end points, and at each end point it subsets the time series `timeser` over a look-back interval equal to `lookb` number of end points.

It passes the subset time series to the function `calc_kurtosis()`, which calculates the kurtosis. See the function `calc_kurtosis()` for a description of the kurtosis methods.

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `timeser` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling kurtosis at 25 day end points, with a 75 day look-back, can be calculated using the parameters `step = 25` and `lookb = 3`.

The function `roll_kurtosis()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A *matrix* of kurtosis estimates with the same number of columns as the input time series `timeser`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
retp <- na.omit(rutils::etfenv$returns$VTI)
# Define end points and start points
endd <- 1 + HighFreq::calc_endpoints(NROW(retp), step=25)
startp <- HighFreq::calc_startpoints(endd, lookb=3)
# Calculate the rolling kurtosis at 25 day end points, with a 75 day look-back
kurtosisv <- HighFreq::roll_kurtosis(retp, step=25, lookb=3)
# Calculate the rolling kurtosis using R code
kurt_r <- sapply(1:NROW(endd), function(it) {
  HighFreq::calc_kurtosis(retp[startp[it]:endd[it], ])
})  # end sapply
# Compare the kurtosis estimates
all.equal(drop(kurtosisv), kurt_r, check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_kurtosis(retp, step=25, lookb=3),
  Rcode=sapply(1:NROW(endd), function(it) {
    HighFreq::calc_kurtosis(retp[startp[it]:endd[it], ])
  }),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`roll_mean` *Calculate a* matrix *of mean (location) estimates over a rolling look-back interval attached at the end points of a* time series *or a* matrix.

Description

Calculate a *matrix* of mean (location) estimates over a rolling look-back interval attached at the end points of a *time series* or a *matrix*.

Usage

```
roll_mean(
  timeser,
  lookb = 1L,
  startp = 0L,
  endd = 0L,
  step = 1L,
  stub = 0L,
  method = "moment",
  confl = 0.75
)
```

Arguments

lookb	The number of end points in the look-back interval (the default is lookb = 1).
timeser	A *time series* or a *matrix* of data.
startp	An *integer* vector of start points (the default is startp = 0).
endd	An *integer* vector of end points (the default is endd = 0).
step	The number of time periods between the end points (the default is step = 1).
stub	An *integer* value equal to the first end point for calculating the end points (the default is stub = 0).
method	A *character string* representing the type of mean measure of (the default is method = "moment").

Details

The function roll_mean() calculates a *matrix* of mean (location) estimates over rolling look-back intervals attached at the end points of the *time series* timeser.

The function roll_mean() performs a loop over the end points, and at each end point it subsets the time series timeser over a look-back interval equal to lookb number of end points.

It passes the subset time series to the function calc_mean(), which calculates the mean (location). See the function calc_mean() for a description of the mean methods.

If the arguments endd and startp are not given then it first calculates a vector of end points separated by step time periods. It calculates the end points along the rows of timeser using the function calc_endpoints(), with the number of time periods between the end points equal to step time periods.

For example, the rolling mean at 25 day end points, with a 75 day look-back, can be calculated using the parameters step = 25 and lookb = 3.

The function roll_mean() with the parameter step = 1 performs the same calculation as the function roll_mean() from package RcppRoll, but it's several times faster because it uses C++ RcppArmadillo code.

The function roll_mean() is implemented in RcppArmadillo RcppArmadillo C++ code, which makes it several times faster than R code.

If only a simple rolling mean is required (not the median) then other functions like roll_sum() or roll_vec() may be even faster.

Value

A *matrix* of mean (location) estimates with the same number of columns as the input time series timeser, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the rolling means at 25 day end points, with a 75 day look-back
meanv <- HighFreq::roll_mean(retp, lookb=3, step=25)
# Compare the mean estimates over 11-period look-back intervals
all.equal(HighFreq::roll_mean(retp, lookb=11)[-(1:10), ],
  drop(RcppRoll::roll_mean(retp, n=11)), check.attributes=FALSE)
# Define end points and start points
endd <- HighFreq::calc_endpoints(NROW(retp), step=25)
startp <- HighFreq::calc_startpoints(endd, lookb=3)
# Calculate the rolling means using RcppArmadillo
meanv <- HighFreq::roll_mean(retp, startp=startp, endd=endd)
# Calculate the rolling medians using RcppArmadillo
medianscpp <- HighFreq::roll_mean(retp, startp=startp, endd=endd, method="nonparametric")
# Calculate the rolling medians using R
medians = sapply(1:NROW(endd), function(i) {
  median(retp[startp[i]:endd[i] + 1])
})  # end sapply
all.equal(medians, drop(medianscpp))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_mean(retp, startp=startp, endd=endd, method="nonparametric"),
  Rcode=sapply(1:NROW(endd), function(i) {median(retp[startp[i]:endd[i] + 1])}),
  times=10))[, c(1, 4, 5)]

## End(Not run)  # end dontrun
```

---

roll_moment   *Calculate a* matrix *of moment values over a rolling look-back interval attached at the end points of a* time series *or a* matrix.

Description

Calculate a *matrix* of moment values over a rolling look-back interval attached at the end points of a *time series* or a *matrix*.

Usage

```
roll_moment(
  timeser,
  funame = "calc_mean",
  method = "moment",
  confl = 0.75,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L
)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`funame`	A *character string* specifying the moment function (the default is `funame = "calc_mean"`).
`method`	A *character string* specifying the type of the model for the moment (the default is `method = "moment"`).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).

Details

The function `roll_moment()` calculates a *matrix* of moment values, over rolling look-back intervals attached at the end points of the *time series* `timeser`.

The function `roll_moment()` performs a loop over the end points, and at each end point it subsets the time series `timeser` over a look-back interval equal to `lookb` number of end points.

It passes the subset time series to the function specified by the argument `funame`, which calculates the statistic. See the functions `calc_*()` for a description of the different moments. The function name must be one of the following:

- "calc_mean" for the estimator of the mean (location),
- "calc_var" for the estimator of the dispersion (variance),
- "calc_skew" for the estimator of the skewness,
- "calc_kurtosis" for the estimator of the kurtosis.

(The default is the `funame = "calc_mean"`).

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `timeser` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling variance at `25` day end points, with a `75` day look-back, can be calculated using the parameters `step = 25` and `lookb = 3`.

The function `roll_moment()` calls the function `calc_momptr()` to calculate a pointer to a moment function from the function name `funame` (string). The function pointer is used internally in the `C++` code, but the function `calc_momptr()` is not exported to `R`.

The function `roll_moment()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A *matrix* with the same number of columns as the input time series `timeser`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the rolling variance at 25 day end points, with a 75 day look-back
var_rollfun <- HighFreq::roll_moment(retp, fun="calc_var", step=25, lookb=3)
# Calculate the rolling variance using roll_var()
var_roll <- HighFreq::roll_var(retp, step=25, lookb=3)
# Compare the two methods
all.equal(var_rollfun, var_roll, check.attributes=FALSE)
# Define end points and start points
endd <- HighFreq::calc_endpoints(NROW(retp), step=25)
startp <- HighFreq::calc_startpoints(endd, lookb=3)
# Calculate the rolling variance using RcppArmadillo
var_rollfun <- HighFreq::roll_moment(retp, fun="calc_var", startp=startp, endd=endd)
# Calculate the rolling variance using R code
var_roll <- sapply(1:NROW(endd), function(it) {
  var(retp[startp[it]:endd[it]+1, ])
})  # end sapply
var_roll[1] <- 0
# Compare the two methods
all.equal(drop(var_rollfun), var_roll, check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_moment(retp, fun="calc_var", startp=startp, endd=endd),
  Rcode=sapply(1:NROW(endd), function(it) {
    var(retp[startp[it]:endd[it]+1, ])
  }),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

---

roll_ohlc   *Aggregate a time series to an* OHLC *time series with lower periodicity.*

Description

Given a time series of prices at a higher periodicity (say seconds), it calculates the *OHLC* prices at a lower periodicity (say minutes).

Usage

```
roll_ohlc(timeser, endd)
```

Arguments

timeser	A *time series* or a *matrix* with multiple columns of data.
\emph{endd}	An *integer vector* of end points.

Details

The function `roll_ohlc()` performs a loop over the end points *endd*, along the rows of the data `timeser`. At each end point, it selects the past rows of the data `timeser`, starting at the first bar after the previous end point, and then calls the function `agg_ohlc()` on the selected data `timeser` to calculate the aggregations.

The function `roll_ohlc()` can accept either a single column of data or four columns of *OHLC* data. It can also accept an additional column containing the trading volume.

The function `roll_ohlc()` performs a similar aggregation as the function `to.period()` from package xts.

Value

A *matrix* with *OHLC* data, with the number of rows equal to the number of *endd* minus one.

Examples

```
## Not run:
# Define matrix of OHLC data
ohlc <- rutils::etfenv$VTI[, 1:5]
# Define end points at 25 day intervals
endd <- HighFreq::calc_endpoints(NROW(ohlc), step=25)
# Aggregate over endd:
ohlcagg <- HighFreq::roll_ohlc(timeser=ohlc, endd=endd)
# Compare with xts::to.period()
ohlcagg_xts <- .Call("toPeriod", ohlc, as.integer(endd+1), TRUE, NCOL(ohlc), FALSE, FALSE, colnames(ohlc), PAC
all.equal(ohlcagg, coredata(ohlcagg_xts), check.attributes=FALSE)

## End(Not run)  # end dontrun
```

`roll_portf` *Simulate (backtest) a rolling portfolio optimization strategy, using* `RcppArmadillo`.

Description

Simulate (backtest) a rolling portfolio optimization strategy, using `RcppArmadillo`.

Usage

```
roll_portf(
  retx,
  retp,
  controll,
  startp,
  endd,
  lambdaf = 0,
  coeff = 1,
  bidask = 0
)
```

Arguments

`retp`	A *time series* or a *matrix* of asset returns data.
`retx`	A *time series* or a *matrix* of excess returns data (the returns in excess of the risk-free rate).
`controll`	A *list* of portfolio optimization model parameters (see Details).
`startp`	An *integer vector* of start points.
`endd`	An *integer vector* of end points.
`lambdaf`	A decay factor which multiplies the past portfolio weights. (The default is `lambdaf = 0` - no memory.)
`coeff`	A *numeric* multiplier of the weights. (The default is `1`)
`bidask`	A *numeric* bid-ask spread (the default is `0`)

Details

The function `roll_portf()` performs a backtest simulation of a rolling portfolio optimization strategy over a *vector* of the end points `endd`.

It performs a loop over the end points `endd`, and subsets the *matrix* of the excess asset returns `retx` along its rows, between the corresponding *start point* and the *end point*.

The function `roll_portf()` passes the subset matrix of excess returns into the function `calc_weights()`, which calculates the optimal portfolio weights at each *end point*. It also passes to `calc_weights()` the argument `controll`, which is the list of portfolio optimization parameters. See the function `calc_weights()` for more details. The list of portfolio optimization parameters can be created using the function `param_portf()`.

The function `roll_portf()` then recursively averages the weights w_i at the *end point = i* with the weights w_{i-1} from the previous *end point = (i-1)*, using the decay factor `lambdaf` = λ:

$$w_i = (1 - \lambda) w_i + \lambda w_{i-1}$$

The purpose of averaging the weights is to reduce their variance, and improve their out-of-sample performance. It is equivalent to extending the portfolio holding period beyond the time interval between neighboring *end points*.

The function `roll_portf()` then calculates the out-of-sample strategy returns by multiplying the average weights times the future asset returns.

The function `roll_portf()` multiplies the out-of-sample strategy returns by the coefficient `coeff` (with default equal to `1`), which allows simulating either a trending strategy (if `coeff = 1`), or a reverting strategy (if `coeff = -1`).

The function `roll_portf()` calculates the transaction costs by multiplying the bid-ask spread `bidask` times the absolute difference between the current weights minus the weights from the previous period. Then it subtracts the transaction costs from the out-of-sample strategy returns.

The function `roll_portf()` returns a *time series* (column *vector*) of strategy returns, of the same length as the number of rows of `retp`.

Value

A column *vector* of strategy returns, with the same length as the number of rows of `retp`.

Examples

```
## Not run:
# Calculate the ETF daily excess returns
retp <- na.omit(rutils::etfenv$returns[, 1:16])
# riskf is the daily risk-free rate
riskf <- 0.03/260
retx <- retp - riskf
# Define monthly end points without initial warmup period
endd <- rutils::calc_endpoints(retp, interval="months")
endd <- endd[endd > 0]
nrows <- NROW(endd)
# Define 12-month look-back interval and start points over sliding window
lookb <- 12
startp <- c(rep_len(1, lookb-1), endd[1:(nrows-lookb+1)])
# Define return shrinkage and dimension reduction
alphac <- 0.5
dimax <- 3
# Create a list of portfolio optimization parameters
controll <- HighFreq::param_portf(method="maxsharpe", dimax=dimax, alphac=alphac, scalew="sumsq")
# Simulate a monthly rolling portfolio optimization strategy
pnls <- HighFreq::roll_portf(retx, retp, controll=controll, startp=(startp-1), endd=(endd-1))
pnls <- xts::xts(pnls, index(retp))
colnames(pnls) <- "strategy"
# Plot dygraph of strategy
dygraphs::dygraph(cumsum(pnls),
  main="Cumulative Returns of Max Sharpe Portfolio Strategy")

## End(Not run)  # end dontrun
```

---

roll_reg — *Perform a rolling regression and calculate a matrix of regression coefficients, their t-values, and z-scores.*

Description

Perform a rolling regression and calculate a matrix of regression coefficients, their t-values, and z-scores.

Usage

```
roll_reg(
  respv,
  predm,
  controll,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L
)
```

Arguments

`respv`	A single-column *time series* or a *vector* of response data.
`predm`	A *time series* or a *matrix* of predictor data.
`controll`	A *list* of model parameters (see Details).
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).

Details

The function `roll_reg()` performs a rolling regression over the end points of the predictor matrix, and calculates a *matrix* of regression coefficients, their t-values, and z-scores.

The function `roll_reg()` performs a loop over the end points, and at each end point it subsets the time series `predm` over a look-back interval equal to `lookb` number of end points.

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `predm` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling regression at `25` day end points, with a `75` day look-back, can be calculated using the parameters `step = 25` and `lookb = 3`.

It passes the subset time series to the function `calc_reg()`, which calculates the regression coefficients, their t-values, and the z-score. The function `roll_reg()` accepts a list of model parameters through the argument `controll`, and passes it to the function `calc_reg()`. The list of model parameters can be created using the function `param_reg()`. See the function `param_reg()` for a description of the model parameters.

The number of columns of the return matrix depends on the number of columns of the predictor matrix (including the intercept column, if it's been added in `R`). The number of regression coefficients is equal to the number of columns of the predictor matrix. If the predictor matrix contains an intercept column then the first regression coefficient is equal to the intercept value α.

The number of columns of the return matrix is equal to the number of regression coefficients, plus their t-values, plus the z-score column. The number of t-values is equal to the number of coefficients. If the number of columns of the predictor matrix is equal to `n`, then `roll_reg()` returns a matrix with `2n+1` columns: `n` regression coefficients, `n` corresponding t-values, and `1` z-score column.

Value

A *matrix* with the regression coefficients, their t-values, and z-scores, and with the same number of rows as `predm` a number of columns equal to `2n+1`, where `n` is the number of columns of `predm`.

Examples

```
## Not run:
# Calculate historical returns
predm <- na.omit(rutils::etfenv$returns[, c("XLP", "VTI")])
# Add unit intercept column to the predictor matrix
```

```
predm <- cbind(rep(1, NROW(predm)), predm)
# Define monthly end points and start points
endd <- xts::endpoints(predm, on="months")[-1]
lookb <- 12
startp <- c(rep(1, lookb), endd[1:(NROW(endd)-lookb)])
# Create a default list of regression parameters
controll <- HighFreq::param_reg()
# Calculate rolling betas using RcppArmadillo
regroll <- HighFreq::roll_reg(respv=predm[, 2], predm=predm[, -2], endd=(endd-1), startp=(startp-1), controll
betas <- regroll[, 2]
# Calculate rolling betas in R
betar <- sapply(1:NROW(endd), FUN=function(ep) {
  datav <- predm[startp[ep]:endd[ep], ]
  # HighFreq::calc_reg(datav[, 2], datav[, -2], controll)
  drop(cov(datav[, 2], datav[, 3])/var(datav[, 3]))
})  # end sapply
# Compare the outputs of both functions
all.equal(betas, betar, check.attributes=FALSE)

## End(Not run)  # end dontrun
```

roll_scale — *Perform a rolling standardization (centering and scaling) of the columns of a* time series *of data using* RcppArmadillo.

Description

Perform a rolling standardization (centering and scaling) of the columns of a *time series* of data using RcppArmadillo.

Usage

```
roll_scale(matrix, lookb, center = TRUE, scale = TRUE, use_median = FALSE)
```

Arguments

timeser	A *time series* or *matrix* of data.
lookb	The length of the look-back interval, equal to the number of rows of data used in the scaling.
center	A *Boolean* argument: if TRUE then center the columns so that they have zero mean or median (the default is TRUE).
scale	A *Boolean* argument: if TRUE then scale the columns so that they have unit standard deviation or MAD (the default is TRUE).
use_median	A *Boolean* argument: if TRUE then the centrality (central tendency) is calculated as the *median* and the dispersion is calculated as the *median absolute deviation* (*MAD*) (the default is FALSE). If use_median = FALSE then the centrality is calculated as the *mean* and the dispersion is calculated as the *standard deviation*.

Details

The function `roll_scale()` performs a rolling standardization (centering and scaling) of the columns of the `timeser` argument using `RcppArmadillo`. The function `roll_scale()` performs a loop over the rows of `timeser`, subsets a number of previous (past) rows equal to `lookb`, and standardizes the subset matrix by calling the function `calc_scale()`. It assigns the last row of the standardized subset *matrix* to the return matrix.

If the arguments `center` and `scale` are both `TRUE` and `use_median` is `FALSE` (the defaults), then `calc_scale()` performs the same calculation as the function `roll::roll_scale()`.

If the arguments `center` and `scale` are both `TRUE` (the defaults), then `calc_scale()` standardizes the data. If the argument `center` is `FALSE` then `calc_scale()` only scales the data (divides it by the standard deviations). If the argument `scale` is `FALSE` then `calc_scale()` only demeans the data (subtracts the means).

If the argument `use_median` is `TRUE`, then it calculates the centrality as the *median* and the dispersion as the *median absolute deviation* (*MAD*).

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Calculate a time series of returns
retp <- zoo::coredata(na.omit(rutils::etfenv$returns[, c("IEF", "VTI")]))
lookb <- 11
rolled_scaled <- roll::roll_scale(retp, width=lookb, min_obs=1)
rolled_scaled2 <- HighFreq::roll_scale(retp, lookb=lookb)
all.equal(rolled_scaled[-(1:2), ], rolled_scaled2[-(1:2), ],
  check.attributes=FALSE)

## End(Not run)  # end dontrun
```

---

roll_sharpe — *Calculate a time series of Sharpe ratios over a rolling look-back interval for an* OHLC *time series.*

Description

Calculate a time series of Sharpe ratios over a rolling look-back interval for an *OHLC* time series.

Usage

```
roll_sharpe(ohlc, lookb = 11)
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`lookb`	The size of the look-back interval, equal to the number of rows of data used for aggregating the *OHLC* prices.

Details

The function `roll_sharpe()` calculates the rolling Sharpe ratio defined as the ratio of percentage returns over the look-back interval, divided by the average volatility of percentage returns.

Value

An *xts* time series with a single column and the same number of rows as the argument `ohlc`.

Examples

```
# Calculate rolling Sharpe ratio over SPY
sharpe_rolling <- roll_sharpe(ohlc=HighFreq::SPY, lookb=11)
```

`roll_skew` *Calculate a* matrix *of skewness estimates over a rolling look-back interval attached at the end points of a* time series *or a* matrix.

Description

Calculate a *matrix* of skewness estimates over a rolling look-back interval attached at the end points of a *time series* or a *matrix*.

Usage

```
roll_skew(
  timeser,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L,
  method = "moment",
  confl = 0.75
)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).
`method`	A *character string* specifying the type of the skewness model (the default is `method = "moment"` - see Details).
`confl`	The confidence level for calculating the quantiles of returns (the default is `confl = 0.75`).

Details

The function `roll_skew()` calculates a *matrix* of skewness estimates over rolling look-back intervals attached at the end points of the *time series* `timeser`.

The function `roll_skew()` performs a loop over the end points, and at each end point it subsets the time series `timeser` over a look-back interval equal to `lookb` number of end points.

It passes the subset time series to the function `calc_skew()`, which calculates the skewness. See the function `calc_skew()` for a description of the skewness methods.

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `timeser` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling skewness at `25` day end points, with a `75` day look-back, can be calculated using the parameters `step = 25` and `lookb = 3`.

The function `roll_skew()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A *matrix* of skewness estimates with the same number of columns as the input time series `timeser`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
retp <- na.omit(rutils::etfenv$returns$VTI)
# Define end points and start points
endd <- 1 + HighFreq::calc_endpoints(NROW(retp), step=25)
startp <- HighFreq::calc_startpoints(endd, lookb=3)
# Calculate the rolling skewness at 25 day end points, with a 75 day look-back
skewv <- HighFreq::roll_skew(retp, step=25, lookb=3)
# Calculate the rolling skewness using R code
skewr <- sapply(1:NROW(endd), function(it) {
  HighFreq::calc_skew(retp[startp[it]:endd[it], ])
})  # end sapply
# Compare the skewness estimates
all.equal(drop(skewv), skewr, check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_skew(retp, step=25, lookb=3),
  Rcode=sapply(1:NROW(endd), function(it) {
    HighFreq::calc_skew(retp[startp[it]:endd[it], ])
  }),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`roll_stats`	*Calculate a vector of statistics over an* OHLC *time series, and calculate a rolling mean over the statistics.*

Description

Calculate a vector of statistics over an *OHLC* time series, and calculate a rolling mean over the statistics.

Usage

```
roll_stats(
  ohlc,
  calc_stats = "ohlc_variance",
  lookb = 11,
  weighted = TRUE,
  ...
)
```

Arguments

`...`	additional parameters to the function `calc_stats`.
`ohlc`	An *OHLC* time series of prices and trading volumes, in *xts* format.
`calc_stats`	The name of the function for estimating statistics of a single row of *OHLC* data, such as volatility, skew, and higher moments.
`lookb`	The size of the look-back interval, equal to the number of rows of data used for calculating the rolling mean.
`weighted`	*Boolean* argument: should statistic be weighted by trade volume? (default `TRUE`)

Details

The function `roll_stats()` calculates a vector of statistics over an *OHLC* time series, such as volatility, skew, and higher moments. The statistics could also be any other aggregation of a single row of *OHLC* data, for example the *High* price minus the *Low* price squared. The length of the vector of statistics is equal to the number of rows of the argument `ohlc`. Then it calculates a trade volume weighted rolling mean over the vector of statistics over and calculate statistics.

Value

An *xts* time series with a single column and the same number of rows as the argument `ohlc`.

Examples

```
# Calculate time series of rolling variance and skew estimates
var_rolling <- roll_stats(ohlc=HighFreq::SPY, lookb=21)
skew_rolling <- roll_stats(ohlc=HighFreq::SPY, calc_stats="ohlc_skew", lookb=21)
skew_rolling <- skew_rolling/(var_rolling)^(1.5)
skew_rolling[1, ] <- 0
skew_rolling <- rutils::na_locf(skew_rolling)
```

roll_sum *Calculate the rolling sums over a* time series *or a* matrix *using* Rcpp.

Description

Calculate the rolling sums over a *time series* or a *matrix* using *Rcpp*.

Usage

```
roll_sum(timeser, lookb = 1L, weightv = 0L)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`lookb`	The length of the look-back interval, equal to the number of data points included in calculating the rolling sum (the default is `lookb = 1`).
`weightv`	A single-column *matrix* of weights (the default is `weightv = 0`).

Details

The function `roll_sum()` calculates the rolling *weighted* sums over the columns of the data `timeser`.

If the argument `weightv` is equal to zero (the default), then the function `roll_sum()` calculates the simple rolling sums of the *time series* data p_t over the look-back interval Δ:

$$\bar{p}_t = \sum_{j=(t-\Delta+1)}^{t} p_j$$

If the `weightv` argument has the same number of rows as the argument `timeser`, then the function `roll_sum()` calculates rolling *weighted* sums of the *time series* data p_t in two steps.

It first calculates the rolling sums of the products of the weights w_t times the *time series* data p_t over the look-back interval Δ:

$$\bar{w}_t = \sum_{j=(t-\Delta+1)}^{t} w_j$$

$$\bar{p}^w_t = \sum_{j=(t-\Delta+1)}^{t} w_j p_j$$

It then calculates the rolling *weighted* sums $\bar{p}_t$ as the ratio of the sum products of the weights and the data, divided by the sums of the weights:

$$\bar{p}_t = \frac{\bar{p}^w_t}{\bar{w}_t}$$

The function `roll_sum()` returns a *matrix* with the same dimensions as the input argument `timeser`.

The function `roll_sum()` is written in `C++` `Armadillo` code, so it's much faster than equivalent `R` code.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "IEF")])
# Define parameters
lookb <- 11
# Calculate rolling sums and compare with rutils::roll_sum()
sumc <- HighFreq::roll_sum(retp, lookb)
sumr <- rutils::roll_sum(retp, lookb)
all.equal(sumc, coredata(sumr), check.attributes=FALSE)
# Calculate rolling sums using R code
sumr <- apply(zoo::coredata(retp), 2, cumsum)
sumlag <- rbind(matrix(numeric(2*lookb), nc=2), sumr[1:(NROW(sumr) - lookb), ])
sumr <- (sumr - sumlag)
all.equal(sumc, sumr, check.attributes=FALSE)
# Calculate weights equal to the trading volumes
weightv <- quantmod::Vo(rutils::etfenv$VTI)
weightv <- weightv[zoo::index(retp)]
# Calculate rolling weighted sums
sumc <- HighFreq::roll_sum(retp, lookb, 1/weightv)
# Plot dygraph of the weighted sums
datav <- cbind(retp$VTI, sumc[, 1])
colnames(datav) <- c("VTI", "Weighted")
endd <- rutils::calc_endpoints(datav, interval="weeks")
dygraphs::dygraph(cumsum(datav)[endd], main=colnames(datav)) %>%
  dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
  dyLegend(width=300)

## End(Not run)  # end dontrun
```

`roll_sumep` *Calculate the rolling sums at the end points of a* time series *or a* matrix.

Description

Calculate the rolling sums at the end points of a *time series* or a *matrix*.

Usage

```
roll_sumep(timeser, startp = 0L, endd = 0L, step = 1L, lookb = 1L, stub = 0L)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points.

Details

The function `roll_sumep()` calculates the rolling sums at the end points of the *time series* `timeser`.

The function `roll_sumep()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A *matrix* with the same number of columns as the input time series `timeser`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "IEF")])
# Define end points at 25 day intervals
endd <- HighFreq::calc_endpoints(NROW(retp), step=25)
# Define start points as 75 day lag of end points
startp <- HighFreq::calc_startpoints(endd, lookb=3)
# Calculate rolling sums using Rcpp
sumc <- HighFreq::roll_sumep(retp, startp=startp, endd=endd)
# Calculate rolling sums using R code
sumr <- sapply(1:NROW(endd), function(ep) {
colSums(retp[(startp[ep]+1):(endd[ep]+1), ])
  })  # end sapply
sumr <- t(sumr)
all.equal(sumc, sumr, check.attributes=FALSE)

## End(Not run)  # end dontrun
```

`roll_sumw` *Calculate the rolling weighted sums over a* time series *or a* matrix *using* Rcpp.

Description

Calculate the rolling weighted sums over a *time series* or a *matrix* using *Rcpp*.

Usage

```
roll_sumw(timeser, endd = NULL, lookb = 1L, stub = NULL, weightv = NULL)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`endd`	An *integer* vector of end points (the default is `endd = NULL`).
`lookb`	The length of the look-back interval, equal to the number of data points included in calculating the rolling sum (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = NULL`).
`weightv`	A single-column *matrix* of weights (the default is `weightv = NULL`).

Details

The function `roll_sumw()` calculates the rolling weighted sums over the columns of the data `timeser`.

The function `roll_sumw()` calculates the rolling weighted sums as convolutions of the columns of `timeser` with the *column vector* of weights using the `Armadillo` function `arma::conv2()`. It performs a similar calculation to the standard `R` function
`stats::filter(x=retp, filter=weightv, method="convolution", sides=1)`, but it can be many times faster, and it doesn't produce any leading `NA` values.

The function `roll_sumw()` returns a *matrix* with the same dimensions as the input argument `timeser`.

The arguments `weightv`, `endd`, and `stub` are optional.

If the argument `weightv` is not supplied, then simple sums are calculated, not weighted sums.

If either the `stub` or `endd` arguments are supplied, then the rolling sums are calculated at the end points.

If only the argument `stub` is supplied, then the end points are calculated from the `stub` and `lookb` arguments. The first end point is equal to `stub` and the end points are spaced `lookb` periods apart.

If the arguments `weightv`, `endd`, and `stub` are not supplied, then the sums are calculated over a number of data points equal to `lookb`.

The function `roll_sumw()` is also several times faster than `rutils::roll_sum()` which uses vectorized `R` code.

Technical note: The function `roll_sumw()` has arguments with default values equal to `NULL`, which are implemented in `Rcpp` code.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# First example
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("VTI", "IEF")])
# Define parameters
lookb <- 11
# Calculate rolling sums and compare with rutils::roll_sum()
sumc <- HighFreq::roll_sum(retp, lookb)
sumr <- rutils::roll_sum(retp, lookb)
all.equal(sumc, coredata(sumr), check.attributes=FALSE)
# Calculate rolling sums using R code
sumr <- apply(zoo::coredata(retp), 2, cumsum)
sumlag <- rbind(matrix(numeric(2*lookb), nc=2), sumr[1:(NROW(sumr) - lookb), ])
sumr <- (sumr - sumlag)
all.equal(sumc, sumr, check.attributes=FALSE)

# Calculate rolling sums at end points
stubv <- 21
sumc <- HighFreq::roll_sumw(retp, lookb, stub=stubv)
endd <- (stubv + lookb*(0:(NROW(retp) %/% lookb)))
endd <- endd[endd < NROW(retp)]
sumr <- apply(zoo::coredata(retp), 2, cumsum)
sumr <- sumr[endd+1, ]
```

```
sumlag <- rbind(numeric(2), sumr[1:(NROW(sumr) - 1), ])
sumr <- (sumr - sumlag)
all.equal(sumc, sumr, check.attributes=FALSE)

# Calculate rolling sums at end points - pass in endd
sumc <- HighFreq::roll_sumw(retp, endd=endd)
all.equal(sumc, sumr, check.attributes=FALSE)

# Create exponentially decaying weights
weightv <- exp(-0.2*(1:11))
weightv <- matrix(weightv/sum(weightv), nc=1)
# Calculate rolling weighted sum
sumc <- HighFreq::roll_sumw(retp, weightv=weightv)
# Calculate rolling weighted sum using filter()
retc <- filter(x=retp, filter=weightv, method="convolution", sides=1)
all.equal(sumc[-(1:11), ], retc[-(1:11), ], check.attributes=FALSE)

# Calculate rolling weighted sums at end points
sumc <- HighFreq::roll_sumw(retp, endd=endd, weightv=weightv)
all.equal(sumc, retc[endd+1, ], check.attributes=FALSE)

# Create simple weights equal to a 1 value plus zeros
weightv <- matrix(c(1, rep(0, 10)), nc=1)
# Calculate rolling weighted sum
weighted <- HighFreq::roll_sumw(retp, weightv=weightv)
# Compare with original
all.equal(coredata(retp), weighted, check.attributes=FALSE)

## End(Not run)  # end dontrun
```

roll_var

Calculate a matrix *of dispersion (variance) estimates over a rolling look-back interval attached at the end points of a* time series *or a* matrix.

Description

Calculate a *matrix* of dispersion (variance) estimates over a rolling look-back interval attached at the end points of a *time series* or a *matrix*.

Usage

```
roll_var(
  timeser,
  lookb = 1L,
  startp = 0L,
  endd = 0L,
  step = 1L,
  stub = 0L,
  method = "moment",
  confl = 0.75
)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).
`method`	A *character string* representing the type of the measure of dispersion (the default is `method = "moment"`).

Details

The function `roll_var()` calculates a *matrix* of dispersion (variance) estimates over rolling look-back intervals attached at the end points of the *time series* `timeser`.

The function `roll_var()` performs a loop over the end points, and at each end point it subsets the time series `timeser` over a look-back interval equal to `lookb` number of end points.

It passes the subset time series to the function `calc_var()`, which calculates the dispersion. See the function `calc_var()` for a description of the dispersion methods.

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `timeser` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling variance at `25` day end points, with a `75` day look-back, can be calculated using the parameters `step = 25` and `lookb = 3`.

The function `roll_var()` with the parameter `step = 1` performs the same calculation as the function `roll_var()` from package RcppRoll, but it's several times faster because it uses `RcppArmadillo C++` code.

The function `roll_var()` is implemented in `RcppArmadillo RcppArmadillo C++` code, which makes it several times faster than `R` code.

Value

A *matrix* dispersion (variance) estimates with the same number of columns as the input time series `timeser`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
retp <- na.omit(rutils::etfenv$returns$VTI)
# Calculate the rolling variance at 25 day end points, with a 75 day look-back
varv <- HighFreq::roll_var(retp, lookb=3, step=25)
# Compare the variance estimates over 11-period look-back intervals
all.equal(HighFreq::roll_var(retp, lookb=11)[-(1:10), ],
  drop(RcppRoll::roll_var(retp, n=11)), check.attributes=FALSE)
# Compare the speed of HighFreq::roll_var() with RcppRoll::roll_var()
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_var(retp, lookb=11),
  RcppRoll=RcppRoll::roll_var(retp, n=11),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
# Compare the speed of HighFreq::roll_var() with TTR::runMAD()
summary(microbenchmark(
    Rcpp=HighFreq::roll_var(retp, lookb=11, method="quantile"),
    TTR=TTR::runMAD(retp, n = 11),
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

roll_varvec — *Calculate a* vector *of variance estimates over a rolling look-back interval for a single-column* time series *or a single-column* matrix*, using* RcppArmadillo.

Description

Calculate a *vector* of variance estimates over a rolling look-back interval for a single-column *time series* or a single-column *matrix*, using RcppArmadillo.

Usage

```
roll_varvec(timeser, lookb = 1L)
```

Arguments

timeser	A single-column *time series* or a single-column *matrix*.
lookb	The length of the look-back interval, equal to the number of *vector* elements used for calculating a single variance estimate (the default is lookb = 1).

Details

The function roll_varvec() calculates a *vector* of variance estimates over a rolling look-back interval for a single-column *time series* or a single-column *matrix*, using RcppArmadillo C++ code.

The function roll_varvec() uses an expanding look-back interval in the initial warmup period, to calculate the same number of elements as the input argument timeser.

The function roll_varvec() performs the same calculation as the function roll_var() from package RcppRoll, but it's several times faster because it uses RcppArmadillo C++ code.

Value

A single-column *matrix* with the same number of elements as the input argument timeser.

Examples

```
## Not run:
# Create a vector of random returns
retp <- rnorm(1e6)
# Compare the variance estimates over 11-period look-back intervals
all.equal(drop(HighFreq::roll_varvec(retp, lookb=11))[-(1:10)],
  RcppRoll::roll_var(retp, n=11))
# Compare the speed of RcppArmadillo with RcppRoll
```

```
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_varvec(retp, lookb=11),
  RcppRoll=RcppRoll::roll_var(retp, n=11),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

`roll_var_ohlc` *Calculate a* vector *of variance estimates over a rolling look-back interval attached at the end points of a* time series *or a* matrix *with* OHLC *price data.*

Description

Calculate a *vector* of variance estimates over a rolling look-back interval attached at the end points of a *time series* or a *matrix* with *OHLC* price data.

Usage

```
roll_var_ohlc(
  ohlc,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L,
  method = "yang_zhang",
  scale = TRUE,
  index = 0L
)
```

Arguments

`ohlc`	A *time series* or a *matrix* with *OHLC* price data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endd`	An *integer* vector of end points (the default is `endd = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`lookb`	The number of end points in the look-back interval (the default is `lookb = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).
`method`	A *character string* representing the price range estimator for calculating the variance. The estimators include: • "close" close-to-close estimator, • "rogers_satchell" Rogers-Satchell estimator, • "garman_klass" Garman-Klass estimator, • "garman_klass_yz" Garman-Klass with account for close-to-open price jumps, • "yang_zhang" Yang-Zhang estimator, (The default is the *"yang_zhang"* estimator.)

`scale`	*Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scale = TRUE`.)
`index`	A *vector* with the time index of the *time series*. This is an optional argument (the default is `index=0`).

Details

The function `roll_var_ohlc()` calculates a *vector* of variance estimates over a rolling look-back interval attached at the end points of the *time series* `ohlc`.

The input *OHLC time series* `ohlc` is assumed to contain the log prices.

The function `roll_var_ohlc()` performs a loop over the end points, subsets the previous (past) rows of `ohlc`, and passes them into the function `calc_var_ohlc()`.

At each end point, the variance is calculated over a look-back interval equal to `lookb` number of end points. In the initial warmup period, the variance is calculated over an expanding look-back interval.

If the arguments `endd` and `startp` are not given then it first calculates a vector of end points separated by `step` time periods. It calculates the end points along the rows of `ohlc` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `step` time periods.

For example, the rolling variance at daily end points with an `11` day look-back, can be calculated using the parameters `step = 1` and `lookb = 1` (Assuming the `ohlc` data has daily frequency.)

Similarly, the rolling variance at `25` day end points with a `75` day look-back, can be calculated using the parameters `step = 25` and `lookb = 3` (because `3*25 = 75`).

The function `roll_var_ohlc()` calculates the variance from all the different intra-day and day-over-day returns (defined as the differences between *OHLC* prices), using several different variance estimation methods.

The default `method` is *"yang_zhang"*, which theoretically has the lowest standard error among unbiased estimators. The methods *"close"*, *"garman_klass_yz"*, and *"yang_zhang"* do account for *close-to-open* price jumps, while the methods *"garman_klass"* and *"rogers_satchell"* do not account for *close-to-open* price jumps.

If `scale` is `TRUE` (the default), then the returns are divided by the differences of the time index (which scales the variance to the units of variance per second squared.) This is useful when calculating the variance from minutes bar data, because dividing returns by the number of seconds decreases the effect of overnight price jumps. If the time index is in days, then the variance is equal to the variance per day squared.

The optional argument `index` is the time index of the *time series* `ohlc`. If the time index is in seconds, then the differences of the index are equal to the number of seconds in each time period. If the time index is in days, then the differences are equal to the number of days in each time period.

The function `roll_var_ohlc()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A column *vector* of variance estimates, with the number of rows equal to the number of end points.

Examples

```
## Not run:
# Extract the log OHLC prices of SPY
ohlc <- log(HighFreq::SPY)
```

```
# Extract the time index of SPY prices
indeks <- c(1, diff(xts::.index(ohlc)))
# Rolling variance at minutes end points, with a 21 minute look-back
varoll <- HighFreq::roll_var_ohlc(ohlc,
                              step=1, lookb=21,
                              method="yang_zhang",
                              index=indeks, scale=TRUE)
# Daily OHLC prices
ohlc <- rutils::etfenv$VTI
indeks <- c(1, diff(xts::.index(ohlc)))
# Rolling variance at 5 day end points, with a 20 day look-back (20=4*5)
varoll <- HighFreq::roll_var_ohlc(ohlc,
                              step=5, lookb=4,
                              method="yang_zhang",
                              index=indeks, scale=TRUE)
# Same calculation in R
nrows <- NROW(ohlc)
closel = HighFreq::lagit(ohlc[, 4])
endd <- drop(HighFreq::calc_endpoints(nrows, 3)) + 1
startp <- drop(HighFreq::calc_startpoints(endd, 2))
npts <- NROW(endd)
varollr <- sapply(2:npts, function(it) {
  rangev <- startp[it]:endd[it]
  sub_ohlc = ohlc[rangev, ]
  sub_close = closel[rangev]
  sub_index = indeks[rangev]
  HighFreq::calc_var_ohlc(sub_ohlc, closel=sub_close, scale=TRUE, index=sub_index)
})  # end sapply
varollr <- c(0, varollr)
all.equal(drop(var_rolling), varollr)

## End(Not run)  # end dontrun
```

`roll_vwap` *Calculate the volume-weighted average price of an* OHLC *time series over a rolling look-back interval.*

Description

Performs the same operation as function `VWAP()` from package TTR, but using vectorized functions, so it's a little faster.

Usage

```
roll_vwap(ohlc, close = ohlc[, 4, drop = FALSE], lookb)
```

Arguments

`ohlc`	An *OHLC* time series of prices in *xts* format.
`close`	A time series of close prices.
`lookb`	The size of the look-back interval, equal to the number of rows of data used for calculating the average price.

Details

The function `roll_vwap()` calculates the volume-weighted average closing price, defined as the sum of the prices multiplied by trading volumes in the look-back interval, divided by the sum of trading volumes in the interval. If the argument `close` is passed in explicitly, then its volume-weighted average value over time is calculated.

Value

An *xts* time series with a single column and the same number of rows as the argument `ohlc`.

Examples

```
# Calculate and plot rolling volume-weighted average closing prices (VWAP)
prices_rolling <- roll_vwap(ohlc=HighFreq::SPY["2013-11"], lookb=11)
chart_Series(HighFreq::SPY["2013-11-12"], name="SPY prices")
add_TA(prices_rolling["2013-11-12"], on=1, col="red", lwd=2)
legend("top", legend=c("SPY prices", "VWAP prices"),
bg="white", lty=c(1, 1), lwd=c(2, 2),
col=c("black", "red"), bty="n")
# Calculate running returns
returns_running <- ohlc_returns(xtsv=HighFreq::SPY)
# Calculate the rolling volume-weighted average returns
roll_vwap(ohlc=HighFreq::SPY, close=returns_running, lookb=11)
```

`roll_zscores` *Calculate a* vector *of z-scores of the residuals of rolling regressions at the end points of the predictor matrix.*

Description

Calculate a *vector* of z-scores of the residuals of rolling regressions at the end points of the predictor matrix.

Usage

```
roll_zscores(
  respv,
  predm,
  startp = 0L,
  endd = 0L,
  step = 1L,
  lookb = 1L,
  stub = 0L
)
```

Arguments

`respv`	A single-column *time series* or a *vector* of response data.
`predm`	A *time series* or a *matrix* of predictor data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).

endd	An *integer* vector of end points (the default is endd = 0).
step	The number of time periods between the end points (the default is step = 1).
lookb	The number of end points in the look-back interval (the default is lookb = 1).
stub	An *integer* value equal to the first end point for calculating the end points (the default is stub = 0).

Details

The function roll_zscores() calculates a *vector* of z-scores of the residuals of rolling regressions at the end points of the *time series* predm.

The function roll_zscores() performs a loop over the end points, and at each end point it subsets the time series predm over a look-back interval equal to lookb number of end points.

It passes the subset time series to the function calc_lm(), which calculates the regression data.

If the arguments endd and startp are not given then it first calculates a vector of end points separated by step time periods. It calculates the end points along the rows of predm using the function calc_endpoints(), with the number of time periods between the end points equal to step time periods.

For example, the rolling variance at 25 day end points, with a 75 day look-back, can be calculated using the parameters step = 25 and lookb = 3.

Value

A column *vector* of the same length as the number of rows of predm.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
respv <- retp[, 1]
# Predictor matrix equals VTI and IEF returns
predm <- retp[, -1]
# Calculate Z-scores from rolling time series regression using RcppArmadillo
lookb <- 11
zscores <- HighFreq::roll_zscores(respv=respv, predm=predm, lookb)
# Calculate z-scores in R from rolling multivariate regression using lm()
zscoresr <- sapply(1:NROW(predm), function(ro_w) {
  if (ro_w == 1) return(0)
  startpoint <- max(1, ro_w-lookb+1)
  responsi <- response[startpoint:ro_w]
  predicti <- predictor[startpoint:ro_w, ]
  regmod <- lm(responsi ~ predicti)
  residuals <- regmod$residuals
  residuals[NROW(residuals)]/sd(residuals)
})  # end sapply
# Compare the outputs of both functions
all.equal(zscores[-(1:lookb)], zscoresr[-(1:lookb)],
  check.attributes=FALSE)

## End(Not run)  # end dontrun
```

`run_autocovar` *Calculate the trailing autocovariance of a* time series *of returns using an online recursive formula.*

Description

Calculate the trailing autocovariance of a *time series* of returns using an online recursive formula.

Usage

```
run_autocovar(timeser, lambdaf, lagg = 1L)
```

Arguments

`timeser`	A *time series* or a *matrix* with a single column of returns data.
`lambdaf`	A decay factor which multiplies past estimates.
`lagg`	An *integer* equal to the number of periods to lag. (The default is `lagg = 1`.)

Details

The function `run_autocovar()` calculates the trailing autocovariance of a streaming *time series* of returns, by recursively weighting the past autocovariance estimates cov_{t-1}, with the products of their returns minus their means, using the decay factor λ:

$$\bar{x}_t = \lambda \bar{x}_{t-1} + (1 - \lambda) x_t$$

$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2)(x_t - \bar{x}_t)^2$$

$$cov_t = \lambda^2 cov_{t-1} + (1 - \lambda^2)(x_t - \bar{x}_t)(x_{t-l} - \bar{x}_{t-l})$$

Where cov_t is the trailing autocovariance estimate at time t, with `lagg=l`. And σ^2_t and $\bar{x}_t$ are the trailing variances and means of the streaming data.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

Note that the variance and covariance decay as the square of λ, while the mean returns decay as λ. This is because the variance is proportional to the square of the returns.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, and the trailing covariance values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, and the trailing covariance values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_autocovar()` returns three columns of data: the trailing autocovariances, the variances, and the mean values of the argument `timeser`. This allows calculating the trailing autocorrelations.

Value

A *matrix* with three columns of data: the trailing autocovariances, the variances, and the mean values of the argument `timeser`.

Examples

```
## Not run:
# Calculate historical returns
retp <- zoo::coredata(na.omit(rutils::etfenv$returns$VTI))
# Calculate the trailing autocovariance
lambdaf <- 0.9 # Decay factor
lagg <- 3
covars <- HighFreq::run_autocovar(retp, lambdaf=lambdaf, lagg=lagg)
# Calculate the trailing autocorrelation
correl <- covars[, 1]/covars[, 2]
# Calculate the trailing autocovariance using R code
nrows <- NROW(retp)
retm <- numeric(nrows)
retm[1] <- retp[1, ]
retd <- numeric(nrows)
covarr <- numeric(nrows)
covarr[1] <- retp[1, ]^2
for (it in 2:nrows) {
  retm[it] <- lambdaf*retm[it-1] + (1-lambdaf)*(retp[it])
  retd[it] <- retp[it] - retm[it]
  covarr[it] <- lambdaf*covarr[it-1] + (1-lambdaf)*retd[it]*retd[max(it-lagg, 1)]
} # end for
all.equal(covarr, covars[, 1])

## End(Not run)  # end dontrun
```

`run_covar` *Calculate the trailing covariances of two streaming* time series *of returns using an online recursive formula.*

Description

Calculate the trailing covariances of two streaming *time series* of returns using an online recursive formula.

Usage

```
run_covar(timeser, lambdaf)
```

Arguments

timeser	A *time series* or a *matrix* with two columns of returns data.
lambdaf	A decay factor which multiplies past estimates.

Details

The function `run_covar()` calculates the trailing covariances of two streaming *time series* of returns, by recursively weighting the past covariance estimates cov_{t-1}, with the products of their returns minus their means, using the decay factor λ:

$$\bar{x}_t = \lambda \bar{x}_{t-1} + (1 - \lambda) x_t$$

$$\begin{aligned}\bar{y}_t &= \lambda \bar{y}_{t-1} + (1-\lambda) y_t \\ \sigma^2_{xt} &= \lambda^2 \sigma^2_{xt-1} + (1-\lambda^2)(x_t - \bar{x}_t)^2 \\ \sigma^2_{yt} &= \lambda^2 \sigma^2_{yt-1} + (1-\lambda^2)(y_t - \bar{y}_t)^2 \\ cov_t &= \lambda^2 cov_{t-1} + (1-\lambda^2)(x_t - \bar{x}_t)(y_t - \bar{y}_t)\end{aligned}$$

Where cov_t is the trailing covariance estimate at time t, σ^2_{xt}, σ^2_{yt}, $\bar{x}_t$ and $\bar{x}_t$ are the trailing variances and means of the returns, and x_t and y_t are the two streaming returns data.

Note that the variance and covariance decay as the square of λ, while the mean returns decay as λ. This is because the variance is proportional to the square of the returns.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, and the trailing covariance values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, and the trailing covariance values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_covar()` returns five columns of data: the trailing covariances, the variances, and the mean values of the two columns of the argument `timeser`. This allows calculating the trailing correlations, betas, and alphas.

Value

A *matrix* with five columns of data: the trailing covariances, the variances, and the mean values of the two columns of the argument `timeser`.

Examples

```
## Not run:
# Calculate historical returns
retp <- zoo::coredata(na.omit(rutils::etfenv$returns[, c("IEF", "VTI")]))
# Calculate the trailing covariance
lambdaf <- 0.9 # Decay factor
covars <- HighFreq::run_covar(retp, lambdaf=lambdaf)
# Calculate the trailing correlation
correl <- covars[, 1]/sqrt(covars[, 2]*covars[, 3])
# Calculate the trailing covariance using R code
nrows <- NROW(retp)
retm <- matrix(numeric(2*nrows), nc=2)
retm[1, ] <- retp[1, ]
retd <- matrix(numeric(2*nrows), nc=2)
covarr <- numeric(nrows)
covarr[1] <- retp[1, 1]*retp[1, 2]
for (it in 2:nrows) {
  retm[it, ] <- lambdaf*retm[it-1, ] + (1-lambdaf)*(retp[it, ])
  retd[it, ] <- retp[it, ] - retm[it, ]
  covarr[it] <- lambdaf*covarr[it-1] + (1-lambdaf)*retd[it, 1]*retd[it, 2]
} # end for
all.equal(covars[, 1], covarr, check.attributes=FALSE)

## End(Not run)  # end dontrun
```

`run_max` *Calculate the trailing maximum values of streaming* time series *data using an online recursive formula.*

Description

Calculate the trailing maximum values of streaming *time series* data using an online recursive formula.

Usage

```
run_max(timeser, lambdaf)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`lambdaf`	A decay factor which multiplies past estimates.

Details

The function `run_max()` calculates the trailing maximum values of streaming *time series* data by recursively weighting present and past values using the decay factor λ.

It calculates the trailing maximum values p^{max}_t of the streaming data p_t as follows:

$$p^{max}_t = max(p_t, \lambda p^{max}_{t-1} + (1 - \lambda) p_t)$$

The first term in the sum is the maximum value multiplied by the decay factor λ, so that the past maximum value is gradually "forgotten". The second term pulls the maximum value to the current value p_t.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the past maximum values persist for longer. This is equivalent to a long look-back interval. If λ is much less than `1` then the past maximum values decay quickly, and the trailing maximum depends on the more recent streaming data. This is equivalent to a short look-back interval.

The above formula can also be expressed as:

$$p^{max}_t = \lambda max(p_t, p^{max}_{t-1}) + (1 - \lambda) p_t$$

The first term is the maximum value multiplied by the decay factor λ, so that the past maximum value is gradually "forgotten". The second term pulls the maximum value to the current value p_t.

The above recursive formula is convenient for processing live streaming data because it doesn't require maintaining a buffer of past data.

The function `run_max()` returns a *matrix* with the same dimensions as the input argument `timeser`.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Calculate historical prices
closep <- zoo::coredata(quantmod::Cl(rutils::etfenv$VTI))
# Calculate the trailing maximums
lambdaf <- 0.9 # Decay factor
pricmax <- HighFreq::run_max(closep, lambdaf=lambdaf)
# Plot dygraph of VTI prices and trailing maximums
datav <- cbind(quantmod::Cl(rutils::etfenv$VTI), pricmax)
colnames(datav) <- c("prices", "max")
colnamev <- colnames(datav)
dygraphs::dygraph(datav, main="VTI Prices and Trailing Maximums") %>%
  dySeries(label=colnamev[1], strokeWidth=2, col="blue") %>%
  dySeries(label=colnamev[2], strokeWidth=2, col="red")

## End(Not run)  # end dontrun
```

`run_mean` *Calculate the exponential moving average (EMA) of streaming* time series *data using an online recursive formula.*

Description

Calculate the exponential moving average (EMA) of streaming *time series* data using an online recursive formula.

Usage

```
run_mean(timeser, lambdaf, weightv = 0L)
```

Arguments

`timeser`	A *time series* or a *matrix*.
`lambdaf`	A decay factor which multiplies past estimates.
`weightv`	A single-column *matrix* of weights.

Details

The function `run_mean()` calculates the exponential moving average (EMA) of the streaming *time series* data p_t by recursively weighting present and past values using the decay factor λ. If the `weightv` argument is equal to zero, then the function `run_mean()` simply calculates the exponentially weighted moving average value of the streaming *time series* data p_t:

$$\bar{p}_t = \lambda \bar{p}_{t-1} + (1 - \lambda) p_t = (1 - \lambda) \sum_{j=0}^{n} \lambda^j p_{t-j}$$

Some applications require applying additional weight factors, like for example the volume-weighted average price indicator (VWAP). Then the streaming prices can be multiplied by the streaming trading volumes.

If the argument `weightv` has the same number of rows as the argument `timeser`, then the function `run_mean()` calculates the exponential moving average (EMA) in two steps.

First it calculates the trailing mean weights $\bar{w}_t$:

$$\bar{w}_t = \lambda \bar{w}_{t-1} + (1 - \lambda) w_t$$

Second it calculates the trailing mean products $\bar{w p}_t$ of the weights w_t and the data p_t:

$$\bar{w p}_t = \lambda \bar{w p}_{t-1} + (1 - \lambda) w_t p_t$$

Where p_t is the streaming data, w_t are the streaming weights, $\bar{w}_t$ are the trailing mean weights, and $\bar{w p}_t$ are the trailing mean products of the data and the weights.

The trailing mean weighted value $\bar{p}_t$ is equal to the ratio of the data and weights products, divided by the mean weights:

$$\bar{p}_t = \frac{\bar{w p}_t}{\bar{w}_t}$$

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, and the trailing mean values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, and the trailing mean values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_mean()` performs the same calculation as the standard `R` function `stats::filter(x=series, filter=lambdaf, method="recursive")`, but it's several times faster.

The function `run_mean()` returns a *matrix* with the same dimensions as the input argument `timeser`.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Calculate historical prices
ohlc <- rutils::etfenv$VTI
closep <- quantmod::Cl(ohlc)
# Calculate the trailing means
lambdaf <- 0.9 # Decay factor
meanv <- HighFreq::run_mean(closep, lambdaf=lambdaf)
# Calculate the trailing means using R code
pricef <- (1-lambdaf)*filter(closep,
  filter=lambdaf, init=as.numeric(closep[1, 1])/(1-lambdaf),
  method="recursive")
all.equal(drop(meanv), unclass(pricef), check.attributes=FALSE)

# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::run_mean(closep, lambdaf=lambdaf),
  Rcode=filter(closep, filter=lambdaf, init=as.numeric(closep[1, 1])/(1-lambdaf), method="recursive"),
   times=10))[, c(1, 4, 5)]  # end microbenchmark summary

# Calculate weights equal to the trading volumes
weightv <- quantmod::Vo(ohlc)
# Calculate the exponential moving average (EMA)
meanw <- HighFreq::run_mean(closep, lambdaf=lambdaf, weightv=weightv)
# Plot dygraph of the EMA
datav <- xts(cbind(meanv, meanw), zoo::index(ohlc))
colnames(datav) <- c("means trailing", "means weighted")
dygraphs::dygraph(datav, main="Trailing Means") %>%
  dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
  dyLegend(show="always", width=300)

## End(Not run)  # end dontrun
```

run_min   *Calculate the trailing minimum values of streaming* time series *data using an online recursive formula.*

Description

Calculate the trailing minimum values of streaming *time series* data using an online recursive formula.

Usage

```
run_min(timeser, lambdaf)
```

Arguments

`timeser`	A *time series* or a *matrix.*
`lambdaf`	A decay factor which multiplies past estimates.

Details

The function `run_min()` calculates the trailing minimum values of streaming *time series* data by recursively weighting present and past values using the decay factor λ.

It calculates the trailing minimum values p^{min}_t of the streaming data p_t as follows:

$$p^{min}_t = min(p_t, \lambda p^{min}_{t-1} + (1 - \lambda) p_t)$$

The first term in the sum is the minimum value multiplied by the decay factor λ, so that the past minimum value is gradually "forgotten". The second term pulls the minimum value to the current value p_t.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the past minimum values persist for longer. This is equivalent to a long look-back interval. If λ is much less than `1` then the past minimum values decay quickly, and the trailing minimum depends on the more recent streaming data. This is equivalent to a short look-back interval.

The above formula can also be expressed as:

$$p^{min}_t = \lambda min(p_t, p^{min}_{t-1}) + (1 - \lambda) p_t$$

The first term is the minimum value multiplied by the decay factor λ, so that the past minimum value is gradually "forgotten". The second term pulls the minimum value to the current value p_t.

The above recursive formula is convenient for processing live streaming data because it doesn't require maintaining a buffer of past data.

The function `run_min()` returns a *matrix* with the same dimensions as the input argument `timeser`.

Value

A *matrix* with the same dimensions as the input argument `timeser`.

Examples

```
## Not run:
# Calculate historical prices
closep <- zoo::coredata(quantmod::Cl(rutils::etfenv$VTI))
# Calculate the trailing minimums
lambdaf <- 0.9 # Decay factor
pricmin <- HighFreq::run_min(closep, lambdaf=lambdaf)
# Plot dygraph of VTI prices and trailing minimums
datav <- cbind(quantmod::Cl(rutils::etfenv$VTI), pricmin)
colnames(datav) <- c("prices", "min")
colnamev <- colnames(datav)
dygraphs::dygraph(datav, main="VTI Prices and Trailing Minimums") %>%
  dySeries(label=colnamev[1], strokeWidth=1, col="blue") %>%
  dySeries(label=colnamev[2], strokeWidth=1, col="red")

## End(Not run)  # end dontrun
```

`run_reg` *Perform regressions on the streaming* time series *of response and predictor data, and calculate the regression coefficients, the residuals, and the forecasts, using online recursive formulas.*

Description

Perform regressions on the streaming *time series* of response and predictor data, and calculate the regression coefficients, the residuals, and the forecasts, using online recursive formulas.

Usage

```
run_reg(respv, predm, lambdaf, controll)
```

Arguments

`respv`	A single-column *time series* or a single-column *matrix* of response data.
`predm`	A *time series* or a *matrix* of predictor data.
`lambdaf`	A decay factor which multiplies past estimates.
`controll`	A *list* of model parameters (see Details).

Details

The function `run_reg()` performs regressions on the streaming *time series* of response r_t and predictor p_t data:

$$r_t = \beta_t p_t + \epsilon_t$$

Where β_t are the trailing regression coefficients and ϵ_t are the residuals.

It recursively updates the covariance matrix cov_t between the response and the predictor data, and the covariance matrix cov_{pt} between the predictors, using the decay factor λ:

$$cov_t = \lambda cov_{t-1} + (1 - \lambda) r_t^T p_t$$

$$cov_{pt} = \lambda cov_{p(t-1)} + (1 - \lambda) p_t^T p_t$$

It calculates the regression coefficients β_t as equal to the covariance matrix between the response and the predictor data cov_t, divided by the covariance matrix between the predictors cov_{pt}:

$$\beta_t = cov_t \, cov_{pt}^{-1}$$

It calculates the residuals ϵ_t as the difference between the response r_t minus the fitted values $\beta_t p_t$:

$$\epsilon_t = r_t - \beta_t p_t$$

And the residual variance σ_t^2 as:

$$\bar{\epsilon}_t = \lambda \bar{\epsilon}_{t-1} + (1 - \lambda) \epsilon_t$$

$$\sigma_t^2 = \lambda^2 \sigma_{t-1}^2 + (1 - \lambda^2)(\epsilon_t - \bar{\epsilon}_t)^2$$

Note that the variance decays as the square of λ, while the mean residuals decay as λ. This is because the variance is proportional to the square of the residuals.

It then calculates the regression forecasts f_t, as equal to the past regression coefficients β_{t-1} times the current predictor data p_t:

$$f_t = \beta_{t-1} p_t$$

It finally calculates the forecast errors as the difference between the response minus the regression forecasts: $r_t - f_t$.

The coefficient matrix β and the residuals ϵ have the same number of rows as the predictor argument `predm`.

The function `run_reg()` accepts a list of regression model parameters through the argument `controll`. The argument `controll` contains the parameters `regmod` and `residscale`. Below is a description of how these parameters work. The list of model parameters can be created using the function `param_reg()`.

The number of regression coefficients is equal to the number of columns of the predictor matrix `n`. If the predictor matrix contains a unit intercept column then the first regression coefficient is equal to the alpha value α.

If `regmod = "least_squares"` (the default) then it performs the standard least squares regression. This is currently the only option.

The *residuals* and the the *forecast errors* may be scaled by their volatilities to obtain the *z-scores*. The default is `residscale = "none"` - no scaling. If the argument `residscale = "scale"` then the *residuals* ϵ_t are divided by their volatilities σ_t without subtracting their means:

$$\epsilon_t = \frac{\epsilon_t}{\sigma_t}$$

If the argument `residscale = "standardize"` then the residual means $\bar{\epsilon}$ are subtracted from the *residuals*, and then they are divided by their volatilities σ_t:

$$\epsilon_t = \frac{\epsilon_t - \bar{\epsilon}}{\sigma_t}$$

Which are equal to the *z-scores*.

The*forecast errors* are also scaled in the same way as the *residuals*, according to the argument`residscale`.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The above recursive formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, so the trailing values have a greater dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, so the trailing values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_reg()` returns multiple columns of data, with the same number of rows as the predictor matrix `predm`. If the predictor matrix `predm` has `n` columns then `run_reg()` returns a matrix with `n+2` columns. The first `n` columns contain the regression coefficients (with the first column equal to the alpha value α). The last `2` columns are the regression residuals and the forecast errors.

Value

A *matrix* with the regression coefficients, the residuals, and the forecasts (in that order - see details), with the same number of rows as the predictor argument `predm`.

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
respv <- retp[, 1]
# Predictor matrix equals VTI and IEF returns
predm <- retp[, -1]
# Add unit intercept column to the predictor matrix
predm <- cbind(rep(1, NROW(predm)), predm)
# Calculate the trailing regressions
lambdaf <- 0.9 # Decay factor
# Create a list of regression parameters
controll <- HighFreq::param_reg(residscale="standardize")
regs <- HighFreq::run_reg(respv=respv, predm=predm, lambdaf=lambdaf, controll=controll)
# Plot the trailing residuals
datav <- cbind(cumsum(respv), regs[, NCOL(regs)])
colnames(datav) <- c("XLF", "residuals")
colnamev <- colnames(datav)
dygraphs::dygraph(datav["2008/2009"], main="Residuals of XLF Versus VTI and IEF") %>%
  dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
  dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
  dySeries(axis="y", strokeWidth=2, col="blue") %>%
  dySeries(axis="y2", strokeWidth=2, col="red") %>%
  dyLegend(show="always", width=300)

# Calculate the trailing regressions using R code
lambda1 <- (1-lambdaf)
respv <- zoo::coredata(respv)
predm <- zoo::coredata(predm)
nrows <- NROW(predm)
ncols <- NCOL(predm)
covrespred <- respv[1, ]*predm[1, ]
covpred <- outer(predm[1, ], predm[1, ])
betas <- matrix(numeric(nrows*ncols), nc=ncols)
betas[1, ] <- covrespred %*% MASS::ginv(covpred)
resids <- numeric(nrows)
residm <- 0
residv <- 0
for (it in 2:nrows) {
 covrespred <- lambdaf*covrespred + lambda1*respv[it, ]*predm[it, ]
 covpred <- lambdaf*covpred + lambda1*outer(predm[it, ], predm[it, ])
 betas[it, ] <- covrespred %*% MASS::ginv(covpred)
 resids[it] <- respv[it, ] - (betas[it, ] %*% predm[it, ])
 residm <- lambdaf*residm + lambda1*resids[it]
 residv <- lambdaf*residv + lambda1*(resids[it] - residm)^2
 resids[it] <- (resids[it] - residm)/sqrt(residv)
} # end for
# Compare values, excluding warmup period
all.equal(regs[-(1:1e3), ], cbind(betas, resids)[-(1:1e3), ], check.attributes=FALSE)


## End(Not run)  # end dontrun
```

run_scale

Standardize (center and scale) the columns of a time series *of data over time and in place, without copying the data in memory, using* `RcppArmadillo`.

Description

Standardize (center and scale) the columns of a *time series* of data over time and in place, without copying the data in memory, using `RcppArmadillo`.

Usage

```
run_scale(timeser, lambdaf, center = TRUE, scale = TRUE)
```

Arguments

`timeser`	A *time series* or *matrix* of data.
`lambdaf`	A decay factor which multiplies past estimates.
`center`	A *Boolean* argument: if `TRUE` then center the columns so that they have zero mean or median (the default is `TRUE`).
`scale`	A *Boolean* argument: if `TRUE` then scale the columns so that they have unit standard deviation or MAD (the default is `TRUE`).

Details

The function run_scale() performs a trailing standardization (centering and scaling) of the columns of the timeser argument using RcppArmadillo.

The function run_scale() accepts a *pointer* to the argument timeser, and it overwrites the old data with the standardized data. It performs the calculation in place, without copying the data in memory, which can significantly increase the computation speed for large time series.

The function run_scale() performs a loop over the rows of timeser, and standardizes the data using its trailing means and standard deviations.

The function run_scale() calculates the trailing mean and variance of streaming *time series* data r_t, by recursively weighting the past estimates with the new data, using the decay factor λ:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2) (r_t - \bar{r}_t)^2$$

Where $\bar{r}_t$ is the trailing mean and σ^2_t is the trailing variance.

It then calculates the standardized data as follows:

$$r^{\prime}_t = \frac{r_t - \bar{r}_t}{\sigma_t}$$

If the arguments center and scale are both TRUE (the defaults), then calc_scale() standardizes the data. If the argument center is FALSE then calc_scale() only scales the data (divides it by the standard deviations). If the argument scale is FALSE then calc_scale() only demeans the data (subtracts the means).

The value of the decay factor λ must be in the range between 0 and 1. If λ is close to 1 then the decay is weak and past values have a greater weight, and the trailing variance values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than 1 then the decay is strong and past values have a smaller weight, and the trailing variance values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The function run_scale() uses RcppArmadillo C++ code, so it can be over 100 times faster than the equivalent R code.

Value

Void (no return value - modifies the data in place).

Examples

```
## Not run:
# Calculate historical returns
retp <- na.omit(rutils::etfenv$returns[, c("XLF", "VTI")])
# Calculate the trailing standardized returns using R code
lambdaf <- 0.9 # Decay factor
lambda1 <- 1 - lambdaf
scaled <- zoo::coredata(retp)
meanm <- scaled[1, ];
```

```
vars <- scaled[1, ]^2;
for (it in 2:NROW(retp)) {
  meanm <- lambdaf*meanm + lambda1*scaled[it, ];
  vars <- lambdaf*vars + lambda1*(scaled[it, ] - meanm)^2;
  scaled[it, ] <- (scaled[it, ] - meanm)/sqrt(vars)
}  # end for
# Calculate the trailing standardized returns using C++ code
HighFreq::run_scale(retp, lambdaf=lambdaf)
all.equal(zoo::coredata(retp), scaled, check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::run_scale(retp, lambdaf=lambdaf),
  Rcode={for (it in 2:NROW(retp)) {
   meanm <- lambdaf*meanm + lambda1*scaled[it, ];
   vars <- lambdaf*vars + lambda1*(scaled[it, ] - meanm)^2;
   scaled[it, ] <- (scaled[it, ] - meanm)/sqrt(vars)
  }},  # end for
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

run_var — *Calculate the trailing mean and variance of streaming* time series *of data using an online recursive formula.*

Description

Calculate the trailing mean and variance of streaming *time series* of data using an online recursive formula.

Usage

```
run_var(timeser, lambdaf)
```

Arguments

`timeser`	A *time series* or a *matrix* of data.
`lambdaf`	A decay factor which multiplies past estimates.

Details

The function `run_var()` calculates the trailing mean and variance of streaming *time series* of data r_t, by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the data minus its trailing means $(r_t - \bar{r}_t)^2$, using the decay factor λ^2:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2)(r_t - \bar{r}_t)^2$$

Where r_t are the streaming data, $\bar{r}_t$ are the trailing means, and σ^2_t are the trailing variance estimates.

Note that the variance decays as the square of λ, while the mean returns decay as λ. This is because the variance is proportional to the square of the returns.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, and the trailing variance values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, and the trailing variance values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_var()` performs the same calculation as the standard `R` function `stats::filter(x=series, filter=weightv, method="recursive")`, but it's several times faster.

The function `run_var()` returns a *matrix* with two columns and the same number of rows as the input argument `timeser`. The first column contains the trailing means and the second contains the variance.

Value

A *matrix* with two columns and the same number of rows as the input argument `timeser`. The first column contains the trailing means and the second contains the variance.

`run_var_ohlc` *Calculate the trailing variance of streaming* OHLC *price data using an online recursive formula.*

Description

Calculate the trailing variance of streaming *OHLC* price data using an online recursive formula.

Usage

```
run_var_ohlc(ohlc, lambdaf)
```

Arguments

`ohlc`	A *time series* or a *matrix* with *OHLC* price data.
`lambdaf`	A decay factor which multiplies past estimates.

Details

The function `run_var_ohlc()` calculates a single-column *matrix* of variance estimates of streaming *OHLC* price data.

The function `run_var_ohlc()` calculates the variance from the differences between the *Open*, *High*, *Low*, and *Close* prices, using the *Yang-Zhang* range volatility estimator:

$$\sigma^2_t = (1-\lambda)((O_t - C_{t-1})^2 + 0.134(C_t - O_t)^2 + 0.866((H_i - O_i)(H_i - C_i) + (L_i - O_i)(L_i - C_i))) + \lambda \sigma^2_{t-1}$$

It recursively weighs the current variance estimate with the past estimates σ^2_{t-1}, using the decay factor λ.

The above recursive formula is convenient for processing live streaming data because it doesn't require maintaining a buffer of past data. The formula is equivalent to a convolution with exponentially decaying weights, but it's faster. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The function `run_var_ohlc()` does not calculate the logarithm of the prices. So if the argument `ohlc` contains dollar prices then `run_var_ohlc()` calculates the dollar variance. If the argument `ohlc` contains the log prices then `run_var_ohlc()` calculates the percentage variance.

The function `run_var_ohlc()` is implemented in `RcppArmadillo` `C++` code, which makes it several times faster than `R` code.

Value

A single-column *matrix* of variance estimates, with the same number of rows as the input `ohlc` price data.

Examples

```
## Not run:
# Extract the log OHLC prices of VTI
ohlc <- log(rutils::etfenv$VTI)
# Calculate the trailing variance
vart <- HighFreq::run_var_ohlc(ohlc, lambdaf=0.8)
# Calculate the rolling variance
varoll <- HighFreq::roll_var_ohlc(ohlc, lookb=5, method="yang_zhang", scale=FALSE)
datav <- cbind(vart, varoll)
colnames(datav) <- c("trailing", "rolling")
colnamev <- colnames(datav)
datav <- xts::xts(datav, index(ohlc))
# dygraph plot of VTI trailing versus rolling volatility
dygraphs::dygraph(sqrt(datav[-(1:111), ]), main="Trailing and Rolling Volatility of VTI") %>%
  dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
  dyLegend(show="always", width=300)
# Compare the speed of trailing versus rolling volatility
library(microbenchmark)
summary(microbenchmark(
  trailing=HighFreq::run_var_ohlc(ohlc, lambdaf=0.8),
  rolling=HighFreq::roll_var_ohlc(ohlc, lookb=5, method="yang_zhang", scale=FALSE),
  times=10))[, c(1, 4, 5)]

## End(Not run)  # end dontrun
```

---

run_zscores — *Calculate the trailing means, volatilities, and z-scores of a streaming* time series *of data using an online recursive formula.*

Description

Calculate the trailing means, volatilities, and z-scores of a streaming *time series* of data using an online recursive formula.

Usage

```
run_zscores(timeser, lambdaf)
```

Arguments

`timeser`	A single *time series* or a single column *matrix* of data.
`lambdaf`	A decay factor which multiplies past estimates.

Details

The function `run_zscores()` calculates the trailing means, volatilities, and z-scores of a single streaming *time series* of data r_t, by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the data minus its trailing means $(r_t - \bar{r}_t)^2$, using the decay factor λ:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2)(r_t - \bar{r}_t)^2$$

$$z_t = \frac{r_t - \bar{r}_t}{\sigma_t}$$

Where r_t are the streaming data, $\bar{r}_t$ are the trailing means, σ^2_t are the trailing variance estimates, and z_t are the z-scores.

The above online recursive formulas are convenient for processing live streaming data because they don't require maintaining a buffer of past data. The formulas are equivalent to a convolution with exponentially decaying weights, but they're much faster to calculate. Using exponentially decaying weights is more natural than using a sliding look-back interval, because it gradually "forgets" about the past data.

The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, and the trailing variance values have a stronger dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, and the trailing variance values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `run_zscores()` returns a *matrix* with three columns (means, volatilities, and z-scores) and the same number of rows as the input argument `timeser`.

Value

A *matrix* with three columns (means, volatilities, and z-scores) and the same number of rows as the input argument `timeser`.

Examples

```
## Not run:
# Calculate historical VTI log prices
pricev <- log(na.omit(rutils::etfenv$prices$VTI))
# Calculate the trailing variance and z-scores of prices
lambdaf <- 0.9 # Decay factor
zscores <- HighFreq::run_zscores(pricev, lambdaf=lambdaf)
datav <- cbind(pricev, zscores[, 3])
colnames(datav) <- c("VTI", "Z-Scores")
colnamev <- colnames(datav)
dygraphs::dygraph(datav, main="VTI Prices and Z-scores") %>%
   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
   dySeries(axis="y", label=colnamev[1], strokeWidth=2, col="blue") %>%
   dySeries(axis="y2", label=colnamev[2], strokeWidth=2, col="red") %>%
   dyLegend(show="always", width=300)
```

```
## End(Not run)  # end dontrun
```

---

save_rets — *Load, scrub, aggregate, and rbind multiple days of* TAQ *data for a single symbol. Calculate returns and save them to a single '`*.RData`' file.*

Description

Load, scrub, aggregate, and rbind multiple days of *TAQ* data for a single symbol. Calculate returns and save them to a single '`*.RData`' file.

Usage

```
save_rets(
  symbol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  lookb = 51,
  vol_mult = 2,
  period = "minutes",
  tzone = "America/New_York"
)
```

Details

The function `save_rets` loads multiple days of *TAQ* data, then scrubs, aggregates, and rbinds them into a *OHLC* time series. It then calculates returns using function `ohlc_returns()`, and stores them in a variable named '`symbol.rets`', and saves them to a file called '`symbol.rets.RData`'. The *TAQ* data files are assumed to be stored in separate directories for each '`symbol`'. Each '`symbol`' has its own directory (named '`symbol`') in the '`data_dir`' directory. Each '`symbol`' directory contains multiple daily '`*.RData`' files, each file containing one day of *TAQ* data.

Value

A time series of returns and volume in *xts* format.

Examples

```
## Not run:
save_rets("SPY")

## End(Not run)
```

save_rets_ohlc — *Load* OHLC *time series data for a single symbol, calculate its returns, and save them to a single '*.RData' file, without aggregation.*

Description

Load *OHLC* time series data for a single symbol, calculate its returns, and save them to a single '*.RData' file, without aggregation.

Usage

```
save_rets_ohlc(
  symbol,
  data_dir = "E:/output/data/",
  output_dir = "E:/output/data/"
)
```

Details

The function `save_rets_ohlc()` loads *OHLC* time series data from a single file. It then calculates returns using function `ohlc_returns()`, and stores them in a variable named '`symbol.rets`', and saves them to a file called '`symbol.rets.RData`'.

Value

A time series of returns and volume in *xts* format.

Examples

```
## Not run:
save_rets_ohlc("SPY")

## End(Not run)
```

save_scrub_agg — *Load, scrub, aggregate, and rbind multiple days of* TAQ *data for a single symbol, and save the* OHLC *time series to a single '*.RData' file.*

Description

Load, scrub, aggregate, and rbind multiple days of *TAQ* data for a single symbol, and save the *OHLC* time series to a single '*.RData' file.

Usage

```
save_scrub_agg(
  symbol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  lookb = 51,
  vol_mult = 2,
  period = "minutes",
  tzone = "America/New_York"
)
```

Arguments

`symbol`	A *character* string representing symbol or ticker.
`data_dir`	A *character* string representing directory containing input '`*.RData`' files.
`output_dir`	A *character* string representing directory containing output '`*.RData`' files.

Details

The function `save_scrub_agg()` loads multiple days of *TAQ* data, then scrubs, aggregates, and rbinds them into a *OHLC* time series, and finally saves it to a single '`*.RData`' file. The *OHLC* time series is stored in a variable named '`symbol`', and then it's saved to a file named '`symbol.RData`' in the '`output_dir`' directory. The *TAQ* data files are assumed to be stored in separate directories for each '`symbol`'. Each '`symbol`' has its own directory (named '`symbol`') in the '`data_dir`' directory. Each '`symbol`' directory contains multiple daily '`*.RData`' files, each file containing one day of *TAQ* data.

Value

An *OHLC* time series in *xts* format.

Examples

```
## Not run:
# set data directories
data_dir <- "C:/Develop/data/hfreq/src/"
output_dir <- "C:/Develop/data/hfreq/scrub/"
symbol <- "SPY"
# Aggregate SPY TAQ data to 15-min OHLC bar data, and save the data to a file
save_scrub_agg(symbol=symbol, data_dir=data_dir, output_dir=output_dir, period="15 min")

## End(Not run)
```

`save_taq` *Load and scrub multiple days of* TAQ *data for a single symbol, and save it to multiple '*`*.RData`*' files.*

Description

Load and scrub multiple days of *TAQ* data for a single symbol, and save it to multiple '`*.RData`' files.

Usage

```
save_taq(
  symbol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  lookb = 51,
  vol_mult = 2,
  tzone = "America/New_York"
)
```

Details

The function `save_taq()` loads multiple days of *TAQ* data, scrubs it, and saves the scrubbed TAQ data to individual '`*.RData`' files. It uses the same file names for output as the input file names. The *TAQ* data files are assumed to be stored in separate directories for each '`symbol`'. Each '`symbol`' has its own directory (named '`symbol`') in the '`data_dir`' directory. Each '`symbol`' directory contains multiple daily '`*.RData`' files, each file containing one day of *TAQ* data.

Value

a *TAQ* time series in *xts* format.

Examples

```
## Not run:
save_taq("SPY")

## End(Not run)
```

`scrub_agg` *Scrub a single day of* TAQ *data, aggregate it, and convert to* OHLC *format.*

Description

Scrub a single day of *TAQ* data, aggregate it, and convert to *OHLC* format.

Usage

```
scrub_agg(
  taq,
  lookb = 51,
  vol_mult = 2,
  period = "minutes",
  tzone = "America/New_York"
)
```

Arguments

`period`	The aggregation period.

Details

The function `scrub_agg()` performs:

- index timezone conversion,
- data subset to trading hours,
- removal of duplicate time stamps,
- scrubbing of quotes with suspect bid-ask spreads,
- scrubbing of quotes with suspect price jumps,
- cbinding of mid prices with volume data,
- aggregation to OHLC using function `to.period()` from package *xts*,

Valid 'period' character strings include: "minutes", "3 min", "5 min", "10 min", "15 min", "30 min", and "hours". The time index of the output time series is rounded up to the next integer multiple of 'period'.

Value

A *OHLC* time series in *xts* format.

Examples

```
# Create random TAQ prices
taq <- HighFreq::random_taq()
# Aggregate to ten minutes OHLC data
ohlc <- HighFreq::scrub_agg(taq, period="10 min")
chart_Series(ohlc, name="random prices")
# scrub and aggregate a single day of SPY TAQ data to OHLC
ohlc <- HighFreq::scrub_agg(taq=HighFreq::SPY_TAQ)
chart_Series(ohlc, name=symbol)
```

`scrub_taq` *Scrub a single day of* TAQ *data in* xts *format, without aggregation.*

Description

Scrub a single day of *TAQ* data in *xts* format, without aggregation.

Usage

```
scrub_taq(taq, lookb = 51, vol_mult = 2, tzone = "America/New_York")
```

Arguments

`taq`	*TAQ* A time series in *xts* format.
`tzone`	The timezone to convert.

Details

The function `scrub_taq()` performs the same scrubbing operations as `scrub_agg`, except it doesn't aggregate, and returns the *TAQ* data in *xts* format.

Value

A *TAQ* time series in *xts* format.

Examples

```
taq <- HighFreq::scrub_taq(taq=HighFreq::SPY_TAQ, lookb=11, vol_mult=1)
# Create random TAQ prices and scrub them
taq <- HighFreq::random_taq()
taq <- HighFreq::scrub_taq(taq=taq)
taq <- HighFreq::scrub_taq(taq=taq, lookb=11, vol_mult=1)
```

---

season_ality — *Perform seasonality aggregations over a single-column* xts *time series.*

Description

Perform seasonality aggregations over a single-column *xts* time series.

Usage

```
season_ality(xtsv, endp = format(zoo::index(xtsv), "%H:%M"))
```

Arguments

`xtsv`	A single-column *xts* time series.
`endp`	A vector of *character* strings representing points in time, of the same length as the argument `xtsv`.

Details

The function `season_ality()` calculates the mean of values observed at the same points in time specified by the argument `endp`. An example of a daily seasonality aggregation is the average price of a stock between 9:30AM and 10:00AM every day, over many days. The argument `endp` is passed into function `tapply()`, and must be the same length as the argument `xtsv`.

Value

An *xts* time series with mean aggregations over the seasonality interval.

Examples

```
# Calculate running variance of each minutely OHLC bar of data
xtsv <- ohlc_variance(HighFreq::SPY)
# Remove overnight variance spikes at "09:31"
endp <- format(index(xtsv), "%H:%M")
xtsv <- xtsv[!endp=="09:31", ]
# Calculate daily seasonality of variance
var_seasonal <- season_ality(xtsv=xtsv)
chart_Series(x=var_seasonal, name=paste(colnames(var_seasonal),
  "daily seasonality of variance"))
```

`sim_ar` *Simulate* autoregressive *returns by recursively filtering a* matrix *of innovations through a* matrix *of* autoregressive *coefficients.*

Description

Simulate *autoregressive* returns by recursively filtering a *matrix* of innovations through a *matrix* of *autoregressive* coefficients.

Usage

```
sim_ar(coeff, innov)
```

Arguments

`innov`	A single-column *matrix* of innovations.
`coeff`	A single-column *matrix* of *autoregressive* coefficients.

Details

The function `sim_ar()` recursively filters the *matrix* of innovations `innov` through the *matrix* of *autoregressive* coefficients `coeff`, using fast `RcppArmadillo` `C++` code.

The function `sim_ar()` simulates an *autoregressive* process $AR(n)$ of order n:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

Where r_i is the simulated output time series, φ_i are the *autoregressive* coefficients, and ξ_i are the standard normal *innovations*.

The order n of the *autoregressive* process $AR(n)$, is equal to the number of rows of the *autoregressive* coefficients `coeff`.

The function `sim_ar()` performs the same calculation as the standard `R` function `filter(x=innov, filter=coeff, method="recursive")`, but it's several times faster.

Value

A single-column *matrix* of simulated returns, with the same number of rows as the argument `innov`.

Examples

```
## Not run:
# Define AR coefficients
coeff <- matrix(c(0.1, 0.3, 0.5))
# Calculate matrix of innovations
innov <- matrix(rnorm(1e4, sd=0.01))
# Calculate recursive filter using filter()
innof <- filter(innov, filter=coeff, method="recursive")
# Calculate recursive filter using RcppArmadillo
retp <- HighFreq::sim_ar(coeff, innov)
# Compare the two methods
all.equal(as.numeric(retp), as.numeric(innof))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
```

```
summary(microbenchmark(
  Rcpp=HighFreq::sim_ar(coeff, innov),
  Rcode=filter(innov, filter=coeff, method="recursive"),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)  # end dontrun
```

sim_df — *Simulate a* Dickey-Fuller *process using* Rcpp.

Description

Simulate a *Dickey-Fuller* process using *Rcpp*.

Usage

```
sim_df(prici, priceq, theta, coeff, innov)
```

Arguments

prici	The initial price.
priceq	The equilibrium price.
theta	The strength of mean reversion.
coeff	A single-column *matrix* of *autoregressive* coefficients.
innov	A single-column *matrix* of innovations (random numbers).

Details

The function `sim_df()` simulates the following *Dickey-Fuller* process:

$$r_i = \theta \, (\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where r_i and p_i are the simulated returns and prices, θ and μ are the *Ornstein-Uhlenbeck* parameters, φ_i are the *autoregressive* coefficients, and ξ_i are the normal *innovations*. The recursion starts with: $r_1 = \xi_1$ and $p_1 = init\_price$.

The *Dickey-Fuller* process is a combination of an *Ornstein-Uhlenbeck* process and an *autoregressive* process. The order n of the *autoregressive* process $AR(n)$, is equal to the number of rows of the *autoregressive* coefficients `coeff`.

The function `sim_df()` simulates the *Dickey-Fuller* process using fast *Rcpp* `C++` code.

The function `sim_df()` returns a single-column *matrix* representing the *time series* of prices.

Value

A single-column *matrix* of simulated prices, with the same number of rows as the argument `innov`.

Examples

```
## Not run:
# Define the Ornstein-Uhlenbeck model parameters
prici <- 1.0
priceq <- 2.0
thetav <- 0.01
# Define AR coefficients
coeff <- matrix(c(0.1, 0.3, 0.5))
# Calculate matrix of standard normal innovations
innov <- matrix(rnorm(1e3, sd=0.01))
# Simulate Dickey-Fuller process using Rcpp
pricev <- HighFreq::sim_df(prici=prici, priceq=priceq, theta=thetav, coeff=coeff, innov=innov)
plot(pricev, t="l", main="Simulated Dickey-Fuller Prices")

## End(Not run)  # end dontrun
```

`sim_garch` *Simulate or estimate the rolling variance under a* GARCH(1,1) *process using* Rcpp.

Description

Simulate or estimate the rolling variance under a *GARCH(1,1)* process using *Rcpp*.

Usage

```
sim_garch(omegac, alphac, betac, innov, is_random = TRUE)
```

Arguments

`omegac`	Parameter proportional to the long-term average level of variance.
`alphac`	The weight associated with recent realized variance updates.
`betac`	The weight associated with the past variance estimates.
`innov`	A single-column *matrix* of innovations.
`is_random`	*Boolean* argument: Are the innovations random numbers or historical returns? (The default is `is_random = TRUE`.)

Details

The function `sim_garch()` simulates or estimates the rolling variance under a *GARCH(1,1)* process using *Rcpp*.

If `is_random = TRUE` (the default) then the innovations `innov` are treated as random numbers ξ_i and the *GARCH(1,1)* process is given by:

$$r_i = \sigma_{i-1} \xi_i$$

$$\sigma^2_i = \omega + \alpha r^2_i + \beta \sigma_{i-1}^2$$

Where r_i and σ^2_i are the simulated returns and variance, and ω, α, and β are the *GARCH* parameters, and ξ_i are standard normal *innovations*.

The long-term equilibrium level of the simulated variance is proportional to the parameter ω:

$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

So the sum of α plus β should be less than 1, otherwise the volatility becomes explosive.

If `is_random = FALSE` then the function `sim_garch()` *estimates* the rolling variance from the historical returns. The innovations `innov` are equal to the historical returns r_i and the *GARCH(1,1)* process is simply:

$$\sigma^2_i = \omega + \alpha r^2_i + \beta \sigma^2_{i-1}$$

Where σ^2_i is the rolling variance.

The above should be viewed as a formula for *estimating* the rolling variance from the historical returns, rather than simulating them. It represents exponential smoothing of the squared returns with a decay factor equal to β.

The function `sim_garch()` simulates the *GARCH* process using fast *Rcpp* `C++` code.

Value

A *matrix* with two columns and with the same number of rows as the argument `innov`. The first column are the simulated returns and the second column is the variance.

Examples

```
## Not run:
# Define the GARCH model parameters
alphac <- 0.79
betac <- 0.2
omegac <- 1e-4*(1-alphac-betac)
# Calculate matrix of standard normal innovations
innov <- matrix(rnorm(1e3))
# Simulate the GARCH process using Rcpp
garch_data <- HighFreq::sim_garch(omegac=omegac, alphac=alphac,  betac=betac, innov=innov)
# Plot the GARCH rolling volatility and cumulative returns
plot(sqrt(garch_data[, 2]), t="l", main="Simulated GARCH Volatility", ylab="volatility")
plot(cumsum(garch_data[, 1]), t="l", main="Simulated GARCH Cumulative Returns", ylab="cumulative returns")
# Calculate historical VTI returns
retp <- na.omit(rutils::etfenv$returns$VTI)
# Estimate the GARCH volatility of VTI returns
garch_data <- HighFreq::sim_garch(omegac=omegac, alphac=alphac,  betac=betac,
  innov=retp, is_random=FALSE)
# Plot dygraph of the estimated GARCH volatility
dygraphs::dygraph(xts::xts(sqrt(garch_data[, 2]), index(retp)),
  main="Estimated GARCH Volatility of VTI")

## End(Not run)  # end dontrun
```

sim_ou   *Simulate an* Ornstein-Uhlenbeck *process using* Rcpp.

Description

Simulate an *Ornstein-Uhlenbeck* process using *Rcpp*.

Usage

```
sim_ou(prici, priceq, theta, innov)
```

Arguments

`prici`	The initial price.
`priceq`	The equilibrium price.
`theta`	The strength of mean reversion.
`innov`	A single-column *matrix* of innovations (random numbers).

Details

The function `sim_ou()` simulates the following *Ornstein-Uhlenbeck* process:

$$r_i = p_i - p_{i-1} = \theta \left(\mu - p_{i-1}\right) + \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where r_i and p_i are the simulated returns and prices, θ, μ, and σ are the *Ornstein-Uhlenbeck* parameters, and ξ_i are the standard *innovations*. The recursion starts with: $r_1 = \xi_1$ and $p_1 = init\_price$.

The function `sim_ou()` simulates the percentage returns as equal to the difference between the equilibrium price μ minus the latest price p_{i-1}, times the mean reversion parameter θ, plus a random normal innovation. The log prices are calculated as the sum of returns (not compounded), so they can become negative.

The function `sim_ou()` simulates the *Ornstein-Uhlenbeck* process using fast *Rcpp* `C++` code.

The function `sim_ou()` returns a single-column *matrix* representing the *time series* of simulated prices.

Value

A single-column *matrix* of simulated prices, with the same number of rows as the argument `innov`.

Examples

```
## Not run:
# Define the Ornstein-Uhlenbeck model parameters
prici <- 0.0
priceq <- 1.0
sigmav <- 0.01
thetav <- 0.01
innov <- matrix(rnorm(1e3))
# Simulate Ornstein-Uhlenbeck process using Rcpp
pricev <- HighFreq::sim_ou(prici=prici, priceq=priceq, volat=sigmav, theta=thetav, innov=innov)
plot(pricev, t="l", main="Simulated Ornstein-Uhlenbeck Prices", ylab="prices")

## End(Not run)  # end dontrun
```

`sim_portfoptim` *Simulate a portfolio optimization strategy using online (recursive) updating of the covariance matrix.*

Description

Simulate a portfolio optimization strategy using online (recursive) updating of the covariance matrix.

Usage

```
sim_portfoptim(rets, dimax, lambdaf, lambdacov, lambdaw)
```

Arguments

`rets`	A *time series* or *matrix* of asset returns.
`dimax`	An *integer* equal to the number of *eigen values* used for calculating the reduced inverse of the covariance matrix (the default is `dimax = 0` - standard matrix inverse using all the *eigen values*).
`lambdaf`	A decay factor which multiplies the past asset returns.
`lambdacov`	A decay factor which multiplies the past covariance.
`lambdaw`	A decay factor which multiplies the past portfolio weights.

Details

The function `sim_portfoptim()` simulates a portfolio optimization strategy. The strategy calculates the maximum Sharpe portfolio weights *in-sample* at every point in time, and applies them in the *out-of-sample* time interval. It updates the trailing covariance matrix recursively, instead of using past batches of data. The function `sim_portfoptim()` uses three different decay factors for averaging past values, to reduce the variance of its forecasts.

The function `sim_portfoptim()` first scales the returns by their trailing volatilities:

$$r^s_t = \frac{r_t}{\sigma_{t-1}}$$

Returns scaled by their volatility are more stationary so they're easier to model.

Then at every point in time, the function `sim_portfoptim()` calls the function `HighFreq::push_covar()` to update the trailing covariance matrix of the returns:

$$\bar{r}_t = \lambda_c \bar{r}_{t-1} + (1 - \lambda_c) r^s_t$$

$$\hat{r}_t = r^s_t - \bar{r}_t$$

$$cov_t = \lambda_c cov_{t-1} + (1 - \lambda^2_c) \hat{r}^T_t \hat{r}_t$$

Where λ_c is the decay factor which multiplies the past mean and covariance.

It then calls the function `HighFreq::calc_inv()` to calculate the *reduced inverse* of the covariance matrix using its eigen decomposition:

$$C^{-1} = O_{dimax}\, \Sigma^{-1}_{dimax}\, O^T_{dimax}$$

See the function `HighFreq::calc_inv()` for details.

It then calculates the *in-sample* weights of the maximum Sharpe portfolio, by multiplying the inverse covariance matrix times the trailing means of the asset returns:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r^s_t$$

$$w_t = C^{-1} \bar{r}_t$$

Note that the decay factor λ is different from the decay factor λ_c used for updating the trailing covariance matrix.

It then scales the weights so their sum of squares is equal to one:

$$w_t = \frac{w_t}{\sqrt{\sum w^2_t}}$$

It then calculates the trailing mean of the weights:

$$\bar{w}_t = \lambda_w \bar{w}_{t-1} + (1 - \lambda_w) w_t$$

Note that the decay factor λ_w is different from the decay factor λ used for updating the trailing means.

It finally calculates the *out-of-sample* portfolio returns by multiplying the trailing mean weights times the scaled asset returns:

$$r^p_t = \bar{w}_{t-1} r^s_t$$

Applying weights to scaled returns means trading stock amounts with unit dollar volatility. So if the weight is equal to `2` then we should purchase an amount of stock with dollar volatility equal to `2` dollars. Trading stock amounts with unit dollar volatility improves portfolio diversification.

The function `sim_portfoptim()` uses three different decay factors for averaging past values, to reduce the variance of its forecasts. The value of the decay factor λ must be in the range between `0` and `1`. If λ is close to `1` then the decay is weak and past values have a greater weight, so the trailing values have a greater dependence on past data. This is equivalent to a long look-back interval. If λ is much less than `1` then the decay is strong and past values have a smaller weight, so the trailing values have a weaker dependence on past data. This is equivalent to a short look-back interval.

The function `sim_portfoptim()` returns multiple columns of data, with the same number of rows as the input argument `rets`. The first column contains the strategy returns and the remaining columns contain the portfolio weights.

Value

A *matrix* of strategy returns and the portfolio weights, with the same number of rows as the argument `rets`.

Examples

```
## Not run:
# Load ETF returns
retp <- rutils::etfenv$returns[, c("VTI", "TLT", "DBC", "USO", "XLF", "XLK")]
retp <- na.omit(retp)
datev <- zoo::index(retp) # dates
# Simulate a portfolio optimization strategy
dimax <- 6
lambdaf <- 0.978 # Decay factor
lambdacov <- 0.995
lambdaw <- 0.9
pnls <- HighFreq::sim_portfoptim(retp, dimax, lambdaf, lambdacov, lambdaw)
```

```
colnames(pnls) <- c("pnls", "VTI", "TLT", "DBC", "USO", "XLF", "XLK")
pnls <- xts::xts(pnls, order.by=datev)
# Plot dygraph of strategy
wealthv <- cbind(retp$VTI, pnls$pnls*sd(retp$VTI)/sd(pnls$pnls))
colnames(wealthv) <- c("VTI", "Strategy")
endd <- rutils::calc_endpoints(wealthv, interval="weeks")
dygraphs::dygraph(cumsum(wealthv)[endd], main="Portfolio Optimization Strategy Returns") %>%
 dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
 dyLegend(width=300)
# Plot dygraph of weights
symbolv <- "VTI"
stockweights <- cbind(cumsum(get(symbolv, retp)), get(symbolv, pnls))
colnames(stockweights)[2] <- "Weight"
colnamev <- colnames(stockweights)
endd <- rutils::calc_endpoints(pnls, interval="weeks")
dygraphs::dygraph(stockweights[endd], main="Returns and Weight") %>%
  dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
  dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
  dySeries(axis="y", label=colnamev[1], strokeWidth=2, col="blue") %>%
  dySeries(axis="y2", label=colnamev[2], strokeWidth=2, col="red")

## End(Not run)  # end dontrun
```

`sim_schwartz` *Simulate a* Schwartz *process using* Rcpp.

Description

Simulate a *Schwartz* process using *Rcpp*.

Usage

```
sim_schwartz(prici, priceq, theta, innov)
```

Arguments

`prici`	The initial price.
`priceq`	The equilibrium price.
`theta`	The strength of mean reversion.
`innov`	A single-column *matrix* of innovations (random numbers).

Details

The function `sim_schwartz()` simulates a *Schwartz* process using fast *Rcpp* `C++` code.

The *Schwartz* process is the exponential of the *Ornstein-Uhlenbeck* process, and similar comments apply to it. The prices are calculated as the exponentially compounded returns, so they are never negative. The log prices can be obtained by taking the logarithm of the prices.

The function `sim_schwartz()` simulates the percentage returns as equal to the difference between the equilibrium price μ minus the latest price p_{i-1}, times the mean reversion parameter θ, plus a random normal innovation.

The function `sim_schwartz()` returns a single-column *matrix* representing the *time series* of simulated prices.

Value

A single-column *matrix* of simulated prices, with the same number of rows as the argument `innov`.

Examples

```
## Not run:
# Define the Schwartz model parameters
prici <- 1.0
priceq <- 2.0
thetav <- 0.01
innov <- matrix(rnorm(1e3, sd=0.01))
# Simulate Schwartz process using Rcpp
pricev <- HighFreq::sim_schwartz(prici=prici, priceq=priceq, theta=thetav, innov=innov)
plot(pricev, t="l", main="Simulated Schwartz Prices", ylab="prices")

## End(Not run)  # end dontrun
```

`which_extreme` *Calculate a* Boolean *vector that identifies extreme tail values in a single-column* xts *time series or vector, over a rolling look-back interval.*

Description

Calculate a *Boolean* vector that identifies extreme tail values in a single-column *xts* time series or vector, over a rolling look-back interval.

Usage

```
which_extreme(xtsv, lookb = 51, vol_mult = 2)
```

Arguments

`xtsv`	A single-column *xts* time series, or a *numeric* or *Boolean* vector.
`lookb`	The number of data points in rolling look-back interval for estimating rolling quantile.
`vol_mult`	The quantile multiplier.

Details

The function `which_extreme()` calculates a *Boolean* vector, with `TRUE` for values that belong to the extreme tails of the distribution of values.

The function `which_extreme()` applies a version of the Hampel median filter to identify extreme values, but instead of using the median absolute deviation (MAD), it uses the `0.9` quantile values calculated over a rolling look-back interval.

Extreme values are defined as those that exceed the product of the multiplier times the rolling quantile. Extreme values belong to the fat tails of the recent (trailing) distribution of values, so they are present only when the trailing distribution of values has fat tails. If the trailing distribution of values is closer to normal (without fat tails), then there are no extreme values.

The quantile multiplier `vol_mult` controls the threshold at which values are identified as extreme. Smaller quantile multiplier values will cause more values to be identified as extreme.

Value

A *Boolean* vector with the same number of rows as the input time series or vector.

Examples

```
# Create local copy of SPY TAQ data
taq <- HighFreq::SPY_TAQ
# scrub quotes with suspect bid-ask spreads
bidask <- taq[, "Ask.Price"] - taq[, "Bid.Price"]
sus_pect <- which_extreme(bidask, lookb=51, vol_mult=3)
# Remove suspect values
taq <- taq[!sus_pect]
```

---

which_jumps

Calculate a Boolean *vector that identifies isolated jumps (spikes) in a single-column* xts *time series or vector, over a rolling interval.*

Description

Calculate a *Boolean* vector that identifies isolated jumps (spikes) in a single-column *xts* time series or vector, over a rolling interval.

Usage

```
which_jumps(xtsv, lookb = 51, vol_mult = 2)
```

Details

The function `which_jumps()` calculates a *Boolean* vector, with `TRUE` for values that are isolated jumps (spikes).

The function `which_jumps()` applies a version of the Hampel median filter to identify jumps, but instead of using the median absolute deviation (MAD), it uses the `0.9` quantile of returns calculated over a rolling interval. This is in contrast to function `which_extreme()`, which applies a Hampel filter to the values themselves, instead of the returns. Returns are defined as simple differences between neighboring values.

Jumps (or spikes), are defined as isolated values that are very different from the neighboring values, either before or after. Jumps create pairs of large neighboring returns of opposite sign.

Jumps (spikes) must satisfy two conditions:

1. Neighboring returns both exceed a multiple of the rolling quantile,
2. The sum of neighboring returns doesn't exceed that multiple.

The quantile multiplier `vol_mult` controls the threshold at which values are identified as jumps. Smaller quantile multiplier values will cause more values to be identified as jumps.

Value

A *Boolean* vector with the same number of rows as the input time series or vector.

Examples

```
# Create local copy of SPY TAQ data
taq <- SPY_TAQ
# Calculate mid prices
mid_prices <- 0.5 * (taq[, "Bid.Price"] + taq[, "Ask.Price"])
# Replace whole rows containing suspect price jumps with NA, and perform locf()
taq[which_jumps(mid_prices, lookb=31, vol_mult=1.0), ] <- NA
taq <- xts:::na.locf.xts(taq)
```