Package ‘HighFreq’

November 16, 2021

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,
RcppRoll,
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 7.1.1.9001

Encoding UTF-8

R topics documented:

`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(tseries)
```

Arguments

`tseries`	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 tseries. The *high* value as the maximum of the *high* column of tseries. The *low* value as the minimum of the *low* column of tseries. The *close* value as the *close* of the last row of tseries. The *volume* value as the sum of the *volume* column of tseries.

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 tseries).

Examples

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

## End(Not run)
```

---

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(oh_lc, calc_bars = "ohlc_variance", weight_ed = TRUE, ...)
```

Arguments

...	additional parameters to the function calc_bars.
oh_lc	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.
weight_ed	*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)
vari_ance <- agg_stats_r(oh_lc=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")
```

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

Description

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

Usage

```
back_test(
  excess,
  returns,
  startp,
  endp,
  lambda,
  method = "rank_sharpe",
  eigen_thresh = 1e-05,
  eigen_max = 0L,
  conf_lev = 0.1,
  alpha = 0,
  scale = TRUE,
  vol_target = 0.01,
  coeff = 1,
  bid_offer = 0
)
```

Arguments

- `returns` A *time series* or a *matrix* of returns data (the returns in excess of the risk-free rate).
- `excess` A *time series* or a *matrix* of excess returns data (the returns in excess of the risk-free rate).
- `startp` An *integer vector* of start points.
- `endp` An *integer vector* of end points.
- `lambda` A *numeric* decay factor to multiply the past portfolio weights. (The default is `lambda = 0` - no memory.)
- `coeff` A *numeric* multiplier of the weights. (The default is `1`)
- `bid_offer` A *numeric* bid-offer spread (the default is `0`)
- `method` A *string* specifying the method for calculating the weights (see Details) (the default is `method = "rank_sharpe"`)
- `eigen_thresh` A *numeric* threshold level for discarding small singular values in order to regularize the inverse of the `returns` matrix (the default is `1e-5`).
- `eigen_max` An *integer* equal to the number of singular values used for calculating the shrinkage inverse of the `returns` matrix (the default is `0` - equivalent to `eigen_max` equal to the number of columns of `returns`).
- `conf_lev` The confidence level for calculating the quantiles (the default is `conf_lev = 0.75`).
- `alpha` The shrinkage intensity between `0` and `1`. (the default is `0`).
- `scale` A *Boolean* specifying whether the weights should be scaled (the default is `scale = TRUE`).
- `vol_target` A *numeric* volatility target for scaling the weights (the default is `1e-5`)

Details

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

It performs a loop over the end points `endp`, and subsets the *matrix* of the excess asset returns `excess` along its rows, between the corresponding *start point* and the *end point*. It passes the subset matrix of excess returns into the function `calc_weights()`, which calculates the optimal portfolio weights at each *end point*. The arguments `eigen_max`, `alpha`, `method`, and `scale` are also passed to the function `calc_weights()`.

It 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 `lambda` = λ:

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

The purpose of averaging the weights is to reduce their variance to 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 `back_test()` then calculates the out-of-sample strategy returns by multiplying the average weights times the future asset returns.

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

The function back_test() calculates the transaction costs by multiplying the bid-offer spread bid_offer 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 back_test() returns a *time series* (column *vector*) of strategy returns, of the same length as the number of rows of returns.

Value

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

Examples

```
## Not run:
# Calculate the ETF daily excess returns
re_turns <- na.omit(rutils::etf_env$re_turns[, 1:16])
# risk_free is the daily risk-free rate
risk_free <- 0.03/260
ex_cess <- re_turns - risk_free
# Define monthly end points without initial warmpup period
end_p <- rutils::calc_endpoints(re_turns, inter_val="months")
end_p <- end_p[end_p > 0]
len_gth <- NROW(end_p)
# Define 12-month look-back interval and start points over sliding window
look_back <- 12
start_p <- c(rep_len(1, look_back-1), end_p[1:(len_gth-look_back+1)])
# Define shrinkage and regularization intensities
al_pha <- 0.5
eigen_max <- 3
# Simulate a monthly rolling portfolio optimization strategy
pnl_s <- HighFreq::back_test(ex_cess, re_turns,
                            start_p-1, end_p-1,
                            eigen_max = eigen_max,
                            alpha = al_pha)
pnl_s <- xts::xts(pnl_s, index(re_turns))
colnames(pnl_s) <- "strat_rets"
# Plot dygraph of strategy
dygraphs::dygraph(cumsum(pnl_s),
  main="Cumulative Returns of Max Sharpe Portfolio Strategy")

## End(Not run)
```

---

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(tseries, method = "var", con_fi = pnorm(-2))
```

Arguments

tseries 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).

con_fi The confidence level for calculating the quantile (the default is con_fi = 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* tseries.

Examples

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

## End(Not run)
```

`calc_eigen` *Calculate the eigen decomposition of the covariance* matrix *of returns data using* `RcppArmadillo`*.*

Description

Calculate the eigen decomposition of the covariance *matrix* of returns data using `RcppArmadillo`.

Usage

```
calc_eigen(tseries)
```

Arguments

`tseries`	A *time series* or *matrix* of returns data.

Details

The function `calc_eigen()` first calculates the covariance *matrix* of `tseries`, and then calculates the eigen decomposition of the covariance *matrix*.

Value

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

Examples

```
## Not run:
# Create matrix of random data
da_ta <- matrix(rnorm(5e6), nc=5)
# Calculate eigen decomposition
ei_gen <- HighFreq::calc_eigen(scale(da_ta, scale=FALSE))
# Calculate PCA
pc_a <- prcomp(da_ta)
# Compare PCA with eigen decomposition
all.equal(pc_a$sdev^2, drop(ei_gen$values))
all.equal(abs(unname(pc_a$rotation)), abs(ei_gen$vectors))
# Compare the speed of Rcpp with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_eigen(da_ta),
  Rcode=prcomp(da_ta),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

calc_endpoints — *Calculate a vector of end points that divides a vector into equal intervals.*

Description

Calculate a vector of end points that divides a vector into equal intervals.

Usage

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

Arguments

`length`	An *integer* equal to the length of the vector to be divided into equal intervals.
`step`	The number of elements in each interval between neighboring end points.
`stub`	An *integer* value equal to the first end point for calculating the end points.

Details

The end points are a vector of integers which divide a vector of length equal to `length` into equally spaced intervals. If a whole number of intervals doesn't fit over the vector, then `calc_endpoints()` adds a stub interval at the end.

The first end point is equal to the argument `step`, unless the argument `stub` is provided, and then it becomes the first end point.

For example, consider the end points for a vector of length `20` divided into intervals of length `step=5`: `0,5,10,15,20`. In order for all the differences between neighboring end points to be equal to `5`, the first end point is set equal to `0`. But `0` doesn't correspond to any vector element, so `calc_endpoints()` doesn't include it and it only retains the non-zero end points equal to: `5,10,15,20`.

Since indexing in `C++` code starts at `0`, then `calc_endpoints()` shifts the end points by `-1` and returns the vector equal to `4,9,14,19`.

If `stub = 1` then the first end point is equal to `1` and the end points are equal to: `1,6,11,16,20`. The extra stub interval at the end is equal to `4 = 20 -16`. And `calc_endpoints()` returns `0,5,10,15,19`. The first value is equal to `0` which is the index of the first element in `C++` code.

If `stub = 2` then the first end point is equal to `2`, with an extra stub interval at the end, and the end points are equal to: `2,7,12,17,20`. And `calc_endpoints()` returns `1,6,11,16,19`.

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.

This works in `R` code because the vector element corresponding to index `0` is empty. For example, the `R` code: `(4:1)[c(0,1)]` produces `4`. So in `R` we can select vector elements using the end points starting at zero.

In `C++` the end points must be shifted by `-1` compared to `R` code, because indexing starts at `0`: `-1,4,9,14,19`. But there is no vector element corresponding to index `-1`. So in `C++` we cannot select vector elements using the end points starting at `-1`. The solution is to drop the first placeholder end point.

Value

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

Examples

```
# Calculate end points without a stub interval
HighFreq::calc_endpoints(length=20, step=5)
# Calculate end points with a final stub interval
HighFreq::calc_endpoints(length=23, step=5)
# Calculate end points with initial and final stub intervals
HighFreq::calc_endpoints(length=20, step=5, stub=2)
# Calculate end points with initial and final stub intervals
HighFreq::calc_endpoints(length=20, step=5, stub=24)
```

`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(tseries, step = 1L)
```

Arguments

`tseries`	A *time series* or a *matrix* of prices.
`step`	The number of periods in each interval between neighboring end points.

Details

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

The aggregated volatility σ_t scales (increases) with 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()` calls the function `calc_var_ag()` to calculate the aggregated volatility σ_t.

Value

The Hurst exponent calculated from the variance of aggregated returns.

Examples

```
## Not run:
# Calculate the log prices
price_s <- na.omit(rutils::etf_env$price_s[, c("XLP", "VTI")])
price_s <- log(price_s)
# Calculate the Hurst exponent from 21 day aggregations
calc_hurst(price_s, step=21)

## End(Not run)
```

---

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 = 1L,
  method = "yang_zhang",
  lag_close = 0L,
  scale = TRUE,
  in_dex = 0L
)
```

Arguments

`ohlc` A *time series* or a *matrix* of *OHLC* prices.

`step` The number of 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"`.)

`lag_close` A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `lag_close = 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`).

`in_dex` A *vector* with the time index of the *time series*. This is an optional argument (the default is `in_dex = 0`).

Details

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

The aggregated volatility σ_t scales (increases) with 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 volatility σ_t.

Value

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

Examples

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

## End(Not run)
```

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

Description

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

Usage

```
calc_inv(tseries, eigen_thresh = 0.01, eigen_max = 0L)
```

Arguments

`tseries`	A *time series* or *matrix* of data.
`eigen_thresh`	A *numeric* threshold level for discarding small singular values in order to regularize the inverse of the matrix `tseries` (the default is `0.01`).
`eigen_max`	An *integer* equal to the number of singular values used for calculating the shrinkage inverse of the matrix `tseries` (the default is `eigen_max = 0` - equivalent to `eigen_max` equal to the number of columns of `tseries`).

Details

The function `calc_inv()` calculates the shrinkage inverse of the matrix `tseries` using Singular Value Decomposition (*SVD*).

The function `calc_inv()` first performs Singular Value Decomposition (*SVD*) of the matrix `tseries`. The *SVD* of a matrix A is defined as the factorization:

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

Where U and V are the left and right *singular matrices*, and Σ is a diagonal matrix of *singular values* $\Sigma = \{\sigma_i\}$.

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

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

The *regularized inverse* of the matrix A is given by:

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

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

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

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

It then discards additional singular values so that only the largest `eigen_max` singular values remain. It calculates the shrinkage inverse from the *SVD* matrices using only the largest singular values up to `eigen_max`. For example, if `eigen_max = 3` then it only uses the `3` largest singular values. This has the effect of dimension shrinkage.

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

Value

A *matrix* equal to the shrinkage inverse of the matrix `tseries`.

Examples

```
## Not run:
# Calculate ETF returns
re_turns <- na.omit(rutils::etf_env$re_turns)
# Calculate covariance matrix
cov_mat <- cov(re_turns)
# Calculate shrinkage inverse using RcppArmadillo
in_verse <- HighFreq::calc_inv(cov_mat, eigen_max=3)
# Calculate shrinkage inverse from SVD in R
s_vd <- svd(cov_mat)
eigen_max <- 1:3
inverse_r <-  s_vd$v[, eigen_max] %*% (t(s_vd$u[, eigen_max]) / s_vd$d[eigen_max])
# Compare RcppArmadillo with R
all.equal(in_verse, inverse_r)

## End(Not run)
```

`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(tseries, method = "moment", conf_lev = 0.75)
```

Arguments

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

Details

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

If `method = "moment"` (the default) then `calc_kurtosis()` calculates the fourth moment of the data. But it doesn't de-mean the columns of `tseries` because that requires copying the matrix `tseries`, 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 `tseries` 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 `tseries`.

Examples

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

## End(Not run)
```

`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(response, predictor)
```

Arguments

`response` A single-column *time series* or a *vector* of response data.

`predictor` 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 *"z_score"*), 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
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
res_ponse <- re_turns[, 1]
# Predictor matrix equals VTI and IEF returns
predic_tor <- re_turns[, -1]
# Perform multivariate regression using lm()
reg_model <- lm(res_ponse ~ predic_tor)
sum_mary <- summary(reg_model)
# Perform multivariate regression using calc_lm()
reg_arma <- HighFreq::calc_lm(response=res_ponse, predictor=predic_tor)
# Compare the outputs of both functions
all.equal(reg_arma$coefficients[, "coeff"], unname(coef(reg_model)))
all.equal(unname(reg_arma$coefficients), unname(sum_mary$coefficients))
all.equal(unname(reg_arma$stats), c(sum_mary$r.squared, unname(sum_mary$fstatistic[1])))
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_lm(response=res_ponse, predictor=predic_tor),
  Rcode=lm(res_ponse ~ predic_tor),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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(tseries, method = "moment", conf_lev = 0.75)
```

Arguments

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

Details

The function calc_mean() calculates the mean (location) values of the columns of the *time series* tseries using RcppArmadillo C++ 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 μ as the sum of 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 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 tseries.

Examples

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

```
all.equal(drop(HighFreq::calc_mean(re_turns, method="quantile", conf_lev=0.9)),
  colSums(sapply(re_turns, 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(re_turns, method="quantile", conf_lev=0.9),
  Rcode=colSums(sapply(re_turns, 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(re_turns, method="nonparametric")
# Calculate the column medians in R
sapply(re_turns, median)
# Compare the values
all.equal(drop(HighFreq::calc_mean(re_turns, method="nonparametric")),
  sapply(re_turns, median), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_mean(re_turns, method="nonparametric"),
  Rcode=sapply(re_turns, median),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

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

Description

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

Usage

```
calc_ranks(tseries)
```

Arguments

`tseries`	A single-column *time series* or a *vector*.

Details

The function `calc_ranks()` calculates the ranks of the elements of a single-column *time series* or a *vector*. It uses the `RcppArmadillo` function `arma::sort_index()`. The function `arma::sort_index()` calculates the permutation index to sort a given vector into 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 permutation index produced by: `arma::sort_index(arma::sort_index())` is the *reverse* of the permutation index produced by: `arma::sort_index()`.

The ranks of the elements are equal to the *reverse* permutation index. The function `calc_ranks()` calculates the *reverse* permutation index.

Value

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

Examples

```
## Not run:
# Create a vector of random data
da_ta <- round(runif(7), 2)
# Calculate the ranks of the elements in two ways
all.equal(rank(da_ta), drop(HighFreq::calc_ranks(da_ta)))
# Create a time series of random data
da_ta <- xts::xts(runif(7), seq.Date(Sys.Date(), by=1, length.out=7))
# Calculate the ranks of the elements in two ways
all.equal(rank(coredata(da_ta)), drop(HighFreq::calc_ranks(da_ta)))
# Compare the speed of RcppArmadillo with R code
da_ta <- runif(7)
library(microbenchmark)
summary(microbenchmark(
  Rcpp=calc_ranks(da_ta),
  Rcode=rank(da_ta),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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(
  response,
  predictor,
  method = "least_squares",
  eigen_thresh = 1e-05,
  eigen_max = 0L,
  conf_lev = 0.1,
  alpha = 0
)
```

Arguments

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

method — A *string* specifying the type of the regression model the default is `method = "least_squares"` - see Details).

eigen_thresh — A *numeric* threshold level for discarding small singular values in order to regularize the inverse of the `predictor` matrix (the default is `1e-5`).

eigen_max — An *integer* equal to the number of singular values used for calculating the shrinkage inverse of the `predictor` matrix (the default is `0` - equivalent to `eigen_max` equal to the number of columns of `predictor`).

conf_lev — The confidence level for calculating the quantiles (the default is `conf_lev = 0.75`).

alpha — The shrinkage intensity between `0` and `1`. (the default is `0`).

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 length of the return vector depends on the number of columns of `predictor`. The number of regression coefficients is equal to the number of columns of `predictor` plus `1`. The number of t-values is equal to the number of coefficients. And there is only `1` z-score. So if the number of columns of `predictor` is equal to `n`, then the return vector will have `2n+3` elements.

For example, if the predictor matrix has `2` columns of data, then `calc_reg()` returns a vector with `7` elements: `3` regression coefficients (including the intercept coefficient), `3` corresponding t-values, and `1` z-score.

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

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

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

Value

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

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
res_ponse <- re_turns[, 1]
# Predictor matrix equals VTI and IEF returns
predic_tor <- re_turns[, -1]
# Perform multivariate regression using lm()
reg_model <- lm(res_ponse ~ predic_tor)
sum_mary <- summary(reg_model)
co_eff <- sum_mary$coefficients
# Perform multivariate regression using calc_reg()
reg_arma <- drop(HighFreq::calc_reg(response=res_ponse, predictor=predic_tor))
# Compare the outputs of both functions
all.equal(reg_arma[1:(2*(1+NCOL(predic_tor)))],
  c(co_eff[, "Estimate"], co_eff[, "t value"]), check.attributes=FALSE)
```

```
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_reg(response=res_ponse, predictor=predic_tor),
  Rcode=lm(res_ponse ~ predic_tor),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

calc_scaled — *Scale (standardize) the columns of a* matrix *of data using* `RcppArmadillo`.

Description

Scale (standardize) the columns of a *matrix* of data using `RcppArmadillo`.

Usage

```
calc_scaled(tseries, use_median = FALSE)
```

Arguments

`tseries`	A *time series* or *matrix* of data.
`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*). If `use_median = FALSE` then the centrality is calculated as the *mean* and the dispersion is calculated as the *standard deviation* (the default is `FALSE`)

Details

The function `calc_scaled()` scales (standardizes) the columns of the `tseries` argument using `RcppArmadillo`.

If the argument `use_median` is `FALSE` (the default), then it 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 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 `tseries` is less than `3` then it returns `tseries` unscaled.

The function `calc_scaled()` uses `RcppArmadillo` `C++` code and is about *5* times faster than function `scale()`, for a *matrix* with *1,000* rows and *20* columns.

Value

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

Examples

```
## Not run:
# Create a matrix of random data
re_turns <- matrix(rnorm(20000), nc=20)
scale_d <- calc_scaled(tseries=re_turns, use_median=FALSE)
scale_d2 <- scale(re_turns)
all.equal(scale_d, scale_d2, check.attributes=FALSE)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=calc_scaled(tseries=re_turns, use_median=FALSE),
  Rcode=scale(re_turns),
  times=100))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`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(tseries, method = "moment", conf_lev = 0.75)
```

Arguments

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

Details

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

Examples

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

## End(Not run)
```

`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(endp, look_back)
```

Arguments

`endp`	An *integer* vector of end points.
`look_back`	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 `endp` lagged (shifted) by an amount equal to `look_back`. 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 `endp`.

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 `look_back = 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 `endp`.

Examples

```
# Calculate end points
end_p <- HighFreq::calc_endpoints(25, 5)
# Calculate start points corresponding to the end points
start_p <- HighFreq::calc_startpoints(end_p, 2)
```

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(tseries, method = "moment", conf_lev = 0.75)
```

Arguments

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

Details

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

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

If method = "moment" (the default) then calc_var() calculates the dispersion as the second moment of the data σ^2 (the variance).

If method = "moment" 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:

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

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 tseries is less than 3 then it returns zeros.

Value

A row vector equal to the dispersion of the columns of the matrix tseries.

Examples

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

## End(Not run)
```

---

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

Description

Calculate the variance of returns aggregated over end points.

Usage

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

Arguments

tseries	A *time series* or a *matrix* of prices.
step	The number of periods in each interval between neighboring end points.

Details

The function `calc_var_ag()` calculates the variance of returns aggregated over 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 `tseries` calculated at the end points. Finally it calculates the variance of the returns.

If there are extra periods that don't fit over the length of `tseries`, then `calc_var_ag()` loops over all possible stub intervals, then it calculates all the corresponding variance values, and averages them.

For example, if the number of rows of `tseries` is equal to `20`, and `step=3` then `6` end points fit over the length of `tseries`, and there are `2` extra periods that must fit into stubs, either at the beginning or at the end (or both).

The aggregated volatility σ_t scales (increases) with 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 volatility ratio.

Value

The variance of aggregated returns.

Examples

```
## Not run:
# Calculate the log prices
price_s <- na.omit(rutils::etf_env$price_s[, c("XLP", "VTI")])
price_s <- log(price_s)
# Calculate the daily variance of percentage returns
calc_var_ag(price_s, step=1)
# Calculate the daily variance using R
sapply(rutils::diff_it(price_s), var)
# Calculate the variance of returns aggregated over 21 days
calc_var_ag(price_s, step=21)
# The variance over 21 days is approximately 21 times the daily variance
21*calc_var_ag(price_s, step=1)

## End(Not run)
```

`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",
  lag_close = 0L,
  scale = TRUE,
  in_dex = 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"`.)
`lag_close`	A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `lag_close = 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`).
`in_dex`	A *vector* with the time index of the *time series*. This is an optional argument (the default is `in_dex = 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 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.

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

The optional argument `in_dex` 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 `lag_close` 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
oh_lc <- log(HighFreq::SPY)
# Extract the time index of SPY prices
in_dex <- c(1, diff(xts::.index(oh_lc)))
# Calculate the variance of SPY returns, with scaling of the returns
HighFreq::calc_var_ohlc(oh_lc,
 method="yang_zhang", scale=TRUE, in_dex=in_dex)
# Calculate variance without accounting for overnight jumps
HighFreq::calc_var_ohlc(oh_lc,
 method="rogers_satchell", scale=TRUE, in_dex=in_dex)
# Calculate the variance without scaling the returns
HighFreq::calc_var_ohlc(oh_lc, scale=FALSE)
# Calculate the variance by passing in the lagged close prices
lag_close <- HighFreq::lag_it(oh_lc[, 4])
all.equal(HighFreq::calc_var_ohlc(oh_lc),
  HighFreq::calc_var_ohlc(oh_lc, lag_close=lag_close))
# Compare with HighFreq::calc_var_ohlc_r()
all.equal(HighFreq::calc_var_ohlc(oh_lc, in_dex=in_dex),
  HighFreq::calc_var_ohlc_r(oh_lc))
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_var_ohlc(oh_lc),
  Rcode=HighFreq::calc_var_ohlc_r(oh_lc),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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 = 1L,
  method = "yang_zhang",
  lag_close = 0L,
  scale = TRUE,
  in_dex = 0L
)
```

Arguments

ohlc	A *time series* or a *matrix* of *OHLC* prices.
step	The number of 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"`.)
lag_close	A *vector* with the lagged *close* prices of the *OHLC time series*. This is an optional argument. (The default is `lag_close = 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`).
in_dex	A *vector* with the time index of the *time series*. This is an optional argument (the default is `in_dex = 0`).

Details

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

If there are extra periods that don't fit over the length of `ohlc`, then `calc_var_ohlc_ag()` loops over all possible stub intervals, it calculates all the corresponding variance values, and it averages them.

For example, if the number of rows of `ohlc` is equal to `20`, and `step=3` then `6` end points fit over the length of `ohlc`, and there are `2` extra periods that must fit into stubs, either at the beginning or at the end (or both).

The aggregated volatility σ_t scales (increases) with 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 volatility ratio.

Value

The variance of aggregated *OHLC* prices.

Examples

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

```
## End(Not run)
```

---

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(oh_lc, method = "yang_zhang", scal_e = TRUE)
```

Arguments

`oh_lc`	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"`)
`scal_e`	*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 `scal_e` 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, scal_e=FALSE)
```

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

Description

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

Usage

```
calc_var_vec(tseries)
```

Arguments

`tseries`	A single-column *time series* or a *vector*.

Details

The function `calc_var_vec()` 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
re_turns <- rnorm(1e6)
# Compare calc_var_vec() with standard var()
all.equal(HighFreq::calc_var_vec(re_turns),
  var(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_var_vec(re_turns),
  Rcode=var(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`calc_weights` *Calculate the optimal portfolio weights for different types of objective functions.*

Description

Calculate the optimal portfolio weights for different types of objective functions.

Usage

```
calc_weights(
  returns,
  method = "rank_sharpe",
  eigen_thresh = 1e-05,
  eigen_max = 0L,
  conf_lev = 0.1,
  alpha = 0,
  scale = TRUE,
  vol_target = 0.01
)
```

Arguments

`returns`	A *time series* or a *matrix* of returns data (the returns in excess of the risk-free rate).
`method`	A *string* specifying the method for calculating the weights (see Details) (the default is `method = "rank_sharpe"`)
`eigen_thresh`	A *numeric* threshold level for discarding small singular values in order to regularize the inverse of the `returns` matrix (the default is `1e-5`).
`eigen_max`	An *integer* equal to the number of singular values used for calculating the shrinkage inverse of the `returns` matrix (the default is `0` - equivalent to `eigen_max` equal to the number of columns of `returns`).
`conf_lev`	The confidence level for calculating the quantiles (the default is `conf_lev = 0.75`).
`alpha`	The shrinkage intensity between `0` and `1`. (the default is `0`).
`scale`	A *Boolean* specifying whether the weights should be scaled (the default is `scale = TRUE`).
`vol_target`	A *numeric* volatility target for scaling the weights (the default is `0.001`)

Details

The function `calc_weights()` calculates the optimal portfolio weights for different types of methods, using `RcppArmadillo` `C++` code.

If `method = "rank_sharpe"` (the default) then it calculates the weights as the ranks (order index) of the trailing Sharpe ratios of the asset `returns`.

If `method = "rank"` then it calculates the weights as the ranks (order index) of the last row of the `returns`.

If method = "max_sharpe" then calc_weights() calculates the weights of the maximum Sharpe portfolio, by multiplying the inverse of the covariance *matrix* times the mean column returns.

If method = "min_var" then it calculates the weights of the minimum variance portfolio under linear constraints.

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

If scale = TRUE (the default) then the weights are scaled so that the resulting portfolio has a volatility equal to vol_target.

calc_weights() calculates the shrinkage inverse of the covariance *matrix* of returns from its eigen decomposition. It applies dimension regularization by selecting only the largest eigenvalues equal in number to eigen_max.

In addition, calc_weights() applies shrinkage to the columns of returns, by shrinking their means to their common mean value. The shrinkage intensity alpha determines the amount of shrinkage that is applied, with alpha = 0 representing no shrinkage (with the column means of returns unchanged), and alpha = 1 representing complete shrinkage (with the column means of returns all equal to the single mean of all the columns).

Value

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

Examples

```
## Not run:
# Calculate covariance matrix of ETF returns
re_turns <- na.omit(rutils::etf_env$re_turns[, 1:16])
ei_gen <- eigen(cov(re_turns))
# Calculate shrinkage inverse of covariance matrix
eigen_max <- 3
eigen_vec <- ei_gen$vectors[, 1:eigen_max]
eigen_val <- ei_gen$values[1:eigen_max]
in_verse <- eigen_vec %*% (t(eigen_vec) / eigen_val)
# Define shrinkage intensity and apply shrinkage to the mean returns
al_pha <- 0.5
col_means <- colMeans(re_turns)
col_means <- ((1-al_pha)*col_means + al_pha*mean(col_means))
# Calculate weights using R
weight_s <- in_verse %*% col_means
n_col <- NCOL(re_turns)
weights_r <- weights_r*sd(re_turns %*% rep(1/n_col, n_col))/sd(re_turns %*% weights_r)
# Calculate weights using RcppArmadillo
weight_s <- drop(HighFreq::calc_weights(re_turns, eigen_max, alpha=al_pha))
all.equal(weight_s, weights_r)

## End(Not run)
```

---

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

Description

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

Usage

```
diff_it(tseries, lagg = 1L, pad_zeros = TRUE)
```

Arguments

`tseries`	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 zeros, in order to return a *matrix* with the same number of rows as the input? (the default is `pad_zeros = TRUE`)

Details

The function `diff_it()` calculates the differences between the rows of the input *matrix* `tseries` and its lagged version.

The argument `lagg` specifies the number of lags applied to the rows of the lagged version of `tseries`. For positive `lagg` values, the lagged version of `tseries` has its rows shifted *forward* (down) by the number equal to `lagg` rows. For negative `lagg` values, the lagged version of `tseries` 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 the rows of the initial (warmup) period at the front, in order to return a *matrix* with the same number of rows as the input `tseries`. The default is `pad_zeros = TRUE`. The padding operation can be time-consuming, because it requires the copying of data.

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

Value

A *matrix* containing the differences between the rows of the input *matrix* `tseries`.

Examples

```
## Not run:
# Create a matrix of random data
da_ta <- matrix(sample(15), nc=3)
# Calculate differences with lagged rows
HighFreq::diff_it(da_ta, lagg=2)
# Calculate differences with advanced rows
HighFreq::diff_it(da_ta, lagg=-2)
# Compare HighFreq::diff_it() with rutils::diff_it()
all.equal(HighFreq::diff_it(da_ta, lagg=2),
  rutils::diff_it(da_ta, lagg=2),
  check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
```

```
  Rcpp=HighFreq::diff_it(da_ta, lagg=2),
  Rcode=rutils::diff_it(da_ta, lagg=2),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`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(tseries, lagg = 1L, pad_zeros = TRUE)
```

Arguments

`tseries`	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 of data.

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
re_turns <- rnorm(1e6)
# Compare diff_vec() with rutils::diff_it()
all.equal(drop(HighFreq::diff_vec(re_turns, lagg=3, pad=TRUE)),
  rutils::diff_it(re_turns, lagg=3))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::diff_vec(re_turns, lagg=3, pad=TRUE),
  Rcode=rutils::diff_it(re_turns, lagg=3),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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"])
```

---

lag_it *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

```
lag_it(tseries, lagg = 1L, pad_zeros = TRUE)
```

Arguments

tseries	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 lag_it() 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 tseries, so that it has the same dimensions as tseries. 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 tseries contains returns data, then the output *matrix* should be padded with zeros, to avoid data snooping. If tseries contains prices, then the output *matrix* should be padded with the prices.

Value

A *matrix* with the same dimensions as the input argument tseries.

Examples

```
## Not run:
# Create a matrix of random returns
re_turns <- matrix(rnorm(5e6), nc=5)
# Compare lag_it() with rutils::lag_it()
all.equal(HighFreq::lag_it(re_turns), rutils::lag_it(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::lag_it(re_turns),
```

```
  Rcode=rutils::lag_it(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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(tseries, lagg = 1L, pad_zeros = TRUE)
```

Arguments

tseries	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* tseries 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 tseries, so that it has the same number of element as tseries. 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 tseries contains returns data, then the output *matrix* should be padded with zeros, to avoid data snooping. If tseries 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
re_turns <- rnorm(1e6)
# Compare lag_vec() with rutils::lag_it()
all.equal(drop(HighFreq::lag_vec(re_turns)),
  rutils::lag_it(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
```

```
  Rcpp=HighFreq::lag_vec(re_turns),
  Rcode=rutils::lag_it(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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(omega, alpha, beta, returns, minval = 1e-06)
```

Arguments

`omega`	Parameter proportional to the long-term average level of variance.
`alpha`	The weight associated with recent realized variance updates.
`beta`	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()`:

$$\sigma_i^2 = \omega + \alpha r_i^2 + \beta \sigma_{i-1}^2$$

Where r_i is the time series of returns, and σ_i^2 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_{i-1}^2 in the previous period, as follows:

$$likelihood = -\sum_{i=1}^n (\frac{r_i^2}{\sigma_{i-1}^2} + \log(\sigma_{i-1}^2))$$

Value

The log-likelihood value.

Examples

```
## Not run:
# Define the GARCH model parameters
al_pha <- 0.79
be_ta <- 0.2
om_ega <- 1e-4*(1-al_pha-be_ta)
# Calculate historical VTI returns
re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
# Calculate the log-likelihood of VTI returns assuming GARCH(1,1)
HighFreq::lik_garch(omega=om_ega, alpha=al_pha,  beta=be_ta, returns=re_turns)

## End(Not run)
```

`mult_vec_mat` *Multiply in place (without copying) the columns or rows of a* matrix *times a* vector*, element-wise.*

Description

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

Usage

```
mult_vec_mat(vector, matrix, by_col = TRUE)
```

Arguments

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

Details

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

If the number of *vector* elements is equal to the number of matrix columns, then it multiplies the columns by the *vector*, and returns the number of columns. If the number of *vector* elements is equal to the number of rows, then it multiplies the rows, and returns the number of rows.

If the *matrix* is square and if `by_col` is `TRUE` then it multiplies the columns, otherwise it multiplies the rows.

It accepts *pointers* to the *matrix* and *vector*, and replaces the old *matrix* values with the new values. It performs the calculation in place, without copying the *matrix* in memory (which greatly increases the computation speed). It performs an implicit loop 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).

The function `mult_vec_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.

Value

A single *integer* value, equal to either the number of *matrix* columns or the number of rows.

Examples

```
## Not run:
# Multiply matrix columns using R
mat_rix <- matrix(round(runif(25e4), 2), nc=5e2)
vec_tor <- round(runif(5e2), 2)
prod_uct <- vec_tor*mat_rix
# Multiply the matrix in place
HighFreq::mult_vec_mat(vec_tor, mat_rix)
all.equal(prod_uct, mat_rix)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_vec_mat(vec_tor, mat_rix),
    Rcode=vec_tor*mat_rix,
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

# Multiply matrix rows using R
mat_rix <- matrix(round(runif(25e4), 2), nc=5e2)
vec_tor <- round(runif(5e2), 2)
prod_uct <- t(vec_tor*t(mat_rix))
# Multiply the matrix in place
HighFreq::mult_vec_mat(vec_tor, mat_rix, by_col=FALSE)
all.equal(prod_uct, mat_rix)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
    Rcpp=HighFreq::mult_vec_mat(vec_tor, mat_rix, by_col=FALSE),
    Rcode=t(vec_tor*t(mat_rix)),
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

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(x_ts, lagg = 1, col_umn = 4, scal_e = TRUE)
```

Arguments

x_ts	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)

col_umn	The column number to extract from the *OHLC* data. (default is 4, or the *Close* prices column)
scal_e	*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 scal_e 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 x_ts 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 scal_e is TRUE (the default), then the returns are expressed in the scale of the time index of the x_ts 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 x_ts 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 col_umn.

Value

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

Examples

```
# Calculate secondly returns from TAQ data
re_turns <- HighFreq::ohlc_returns(x_ts=HighFreq::SPY_TAQ)
# Calculate close to close returns
re_turns <- HighFreq::ohlc_returns(x_ts=HighFreq::SPY)
# Calculate open to open returns
re_turns <- HighFreq::ohlc_returns(x_ts=HighFreq::SPY, col_umn=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(oh_lc, method = "close")
```

Arguments

oh_lc	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 `oh_lc`.

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(oh_lc, method = "rogers_satchell")
```

Arguments

`oh_lc`	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
sk_ew <- 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(oh_lc, method = "yang_zhang", scal_e = TRUE)
```

Arguments

`oh_lc` 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"`)

`scal_e` *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 scal_e 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 scal_e 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 oh_lc 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 oh_lc.

Examples

```
# Create minutely OHLC time series of random prices
oh_lc <- HighFreq::random_ohlc()
# Calculate variance estimates for oh_lc
var_running <- HighFreq::ohlc_variance(oh_lc)
# 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")
```

---

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(
  oh_lc = NULL,
  re_duce = TRUE,
  vol_at = 6.5e-05,
  dri_ft = 0,
  in_dex = 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

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

Details

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

If `re_duce` is `TRUE` (the default), then the `oh_lc` 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 `oh_lc` time series.

Examples

```
# Create minutely synthetic OHLC time series of random prices
oh_lc <- HighFreq::random_ohlc()
# Create random time series from SPY by randomly sampling it
oh_lc <- HighFreq::random_ohlc(oh_lc=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(
  vol_at = 6.5e-05,
  dri_ft = 0,
  in_dex = seq(from = as.POSIXct(paste(Sys.Date() - 3, "09:30:00")), to =
    as.POSIXct(paste(Sys.Date() - 1, "16:00:00")), by = "1 sec"),
  bid_offer = 0.001,
  ...
)
```

Arguments

`bid_offer`	The bid-offer spread expressed as a fraction of the prices (default is 0.001=10bps).
`vol_at`	The volatility per period of the `in_dex` time index (default is `6.5e-05` per second, or about `0.01=1.0%` per day).
`dri_ft`	The drift per period of the `in_dex` time index (default is 0.0).
`in_dex`	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 `in_dex` 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 `in_dex` 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
ta_q <- HighFreq::random_taq()
# Create random TAQ time series from SPY index
ta_q <- HighFreq::random_taq(in_dex=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(oh_lc)
```

Arguments

`oh_lc`	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 `oh_lc` 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 `oh_lc` time series.

Examples

```
# Remove overnight close-to-open price jumps from SPY data
oh_lc <- 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(
  x_ts,
  agg_fun,
  look_back = 2,
  end_points = seq_along(x_ts),
  by_columns = FALSE,
  out_xts = TRUE,
  ...
)
```

Arguments

`...`	additional parameters to the function `agg_fun`.
`x_ts`	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.
`look_back`	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`)
`end_points`	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 `end_points`. If the argument `end_points` 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 `end_points` is explicitly passed to `roll_apply()`, then `roll_apply()` performs aggregations over intervals attached at the end_points. If look_back=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 `end_points`. Otherwise a list is returned, with the length equal to the length of argument `end_points`.

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 `end_points`, or a list the length of argument `end_points`.

Examples

```
# extract a single day of SPY data
oh_lc <- HighFreq::SPY["2012-02-13"]
inter_val <- 11  # number of data points between end points
look_back <- 4  # number of end points in look-back interval
```

```
# Calculate the rolling sums of oh_lc columns over a rolling look-back interval
agg_regations <- roll_apply(oh_lc, agg_fun=sum, look_back=look_back, by_columns=TRUE)
# Apply a vector-valued aggregation function over a rolling look-back interval
agg_function <- function(oh_lc)  c(max(oh_lc[, 2]), min(oh_lc[, 3]))
agg_regations <- roll_apply(oh_lc, agg_fun=agg_function, look_back=look_back)
# Define end points at 11-minute intervals (HighFreq::SPY is minutely bars)
end_points <- rutils::end_points(oh_lc, inter_val=inter_val)
# Calculate the sums of oh_lc columns over end_points using non-overlapping intervals
agg_regations <- roll_apply(oh_lc, agg_fun=sum, end_points=end_points, by_columns=TRUE)
# Apply a vector-valued aggregation function over the end_points of oh_lc
# using overlapping intervals
agg_regations <- roll_apply(oh_lc, agg_fun=agg_function,
                            look_back=5, end_points=end_points)
```

`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(
  x_ts,
  train_func,
  trade_func,
  look_back = look_forward,
  look_forward,
  end_points = rutils::calc_endpoints(x_ts, look_forward),
  ...
)
```

Arguments

`...`	additional parameters to the functions `train_func()` and `trade_func()`.
`x_ts`	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.
`look_back`	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.
`end_points`	A vector of end points along the rows of the `x_ts` 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 `x_ts` 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 lists returned by function `trade_func()`. The list names are equal to the *end_points* dates. The number of list elements is equal to the number of *end_points* 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
price_s <- cbind(rutils::etf_env$XLU, rutils::etf_env$XLP)
look_back <- 252
look_forward <- 22
# Define end points
end_points <- rutils::calc_endpoints(price_s, look_forward)
# Perform back-test
back_test <- roll_backtest(end_points=end_points,
    look_forward=look_forward,
    look_back=look_back,
    train_func = train_model,
    trade_func = trade_model,
    model_params = model_params,
    trading_params = trading_params,
    x_ts=price_s)

## End(Not run)
```

`roll_conv` *Calculate the rolling convolutions (weighted sums) of a* time series *with a* column vector *of weights.*

Description

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

Usage

```
roll_conv(tseries, weights)
```

Arguments

`tseries`	A *time series* or a *matrix* of data.
`weights`	A *column vector* of weights.

Details

The function `roll_conv()` calculates the convolutions of the *matrix* columns with a *column vector* 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 values.

The function `roll_conv()` uses the `RcppArmadillo` function `arma::conv2()`. It performs a similar calculation to the standard `R` function
`filter(x=tseries,filter=weight_s,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 `tseries`.

Examples

```
## Not run:
# First example
# Calculate a time series of returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("IEF", "VTI")])
# Create simple weights equal to a 1 value plus zeros
weight_s <- matrix(c(1, rep(0, 10)), nc=1)
# Calculate rolling weighted sums
weight_ed <- HighFreq::roll_conv(re_turns, weight_s)
# Compare with original
all.equal(coredata(re_turns), weight_ed, check.attributes=FALSE)
# Second example
# Calculate exponentially decaying weights
weight_s <- exp(-0.2*(1:11))
weight_s <- matrix(weight_s/sum(weight_s), nc=1)
# Calculate rolling weighted sums
weight_ed <- HighFreq::roll_conv(re_turns, weight_s)
# Calculate rolling weighted sums using filter()
filter_ed <- filter(x=re_turns, filter=weight_s, method="convolution", sides=1)
# Compare both methods
all.equal(filter_ed[-(1:11), ], weight_ed[-(1:11), ], check.attributes=FALSE)

## End(Not run)
```

---

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(tseries)
```

Arguments

tseries	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 `tseries`.

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)
```

---

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

Description

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

Usage

```
roll_fun(
  tseries,
  fun = "calc_var",
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "moment",
  conf_lev = 0.75
)
```

Arguments

tseries	A *time series* or a *matrix* of data.
fun	A *string* specifying the estimator function (the default is fun = "calc_var".)
startp	An *integer* vector of start points (the default is startp = 0).
endp	An *integer* vector of end points (the default is endp = 0).
step	The number of time periods between the end points (the default is step = 1).
look_back	The number of end points in the look-back interval (the default is look_back = 1).
stub	An *integer* value equal to the first end point for calculating the end points (the default is stub = 0).
method	A *string* specifying the type of the model for the estimator (the default is method = "moment".)
conf_lev	The confidence level for calculating the quantiles (the default is conf_lev = 0.75).

Details

The function roll_fun() calculates a *matrix* of estimator values, over rolling look-back intervals attached at the end points of the *time series* tseries.

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

It passes the subset time series to the function specified by the argument fun, which calculates the statistic. See the functions calc_*() for a description of the different estimators.

If the arguments endp 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 tseries 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 look_back = 3.

The function roll_fun() is implemented in RcppArmadillo C++ code, so it's many times faster than the equivalent R code.

Value

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

Examples

```
## Not run:
# Define time series of returns using package rutils
re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
# Calculate the rolling variance at 25 day end points, with a 75 day look-back
var_rollfun <- HighFreq::roll_fun(re_turns, fun="calc_var", step=25, look_back=3)
# Calculate the rolling variance using roll_var()
var_roll <- HighFreq::roll_var(re_turns, step=25, look_back=3)
# Compare the two methods
all.equal(var_rollfun, var_roll, check.attributes=FALSE)
# Define end points and start points
```

```
end_p <- HighFreq::calc_endpoints(NROW(re_turns), step=25)
start_p <- HighFreq::calc_startpoints(end_p, look_back=3)
# Calculate the rolling variance using RcppArmadillo
var_rollfun <- HighFreq::roll_fun(re_turns, fun="calc_var", startp=start_p, endp=end_p)
# Calculate the rolling variance using R code
var_roll <- sapply(1:NROW(end_p), function(it) {
  var(re_turns[start_p[it]:end_p[it]+1, ])
})  # end sapply
# 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_fun(re_turns, fun="calc_var", startp=start_p, endp=end_p),
  Rcode=sapply(1:NROW(end_p), function(it) {
    var(re_turns[start_p[it]:end_p[it]+1, ])
  }),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`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(oh_lc, look_back = 11)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`look_back`	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 `oh_lc`.

Examples

```
# Calculate rolling Hurst for SPY in March 2009
hurst_rolling <- roll_hurst(oh_lc=HighFreq::SPY["2009-03"], look_back=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(
  tseries,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "moment",
  conf_lev = 0.75
)
```

Arguments

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

The function `roll_kurtosis()` performs a loop over the end points, and at each end point it subsets the time series `tseries` over a look-back interval equal to `look_back` 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 `endp` 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 `tseries` 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 `look_back = 3`.

The function `roll_kurtosis()` is implemented in `RcppArmadillo` `C++` code, so it's many times faster than the equivalent `R` code.

Value

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

Examples

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

## End(Not run)
```

`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(
  tseries,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "moment",
  conf_lev = 0.75
)
```

Arguments

`tseries`	A *time series* or a *matrix* of data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endp`	An *integer* vector of end points (the default is `endp = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`look_back`	The number of end points in the look-back interval (the default is `look_back = 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* `tseries`.

The function `roll_mean()` performs a loop over the end points, and at each end point it subsets the time series `tseries` over a look-back interval equal to `look_back` 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 `endp` 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 `tseries` 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 `look_back = 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 `RcppArmadillo` C++ code.

The function `roll_mean()` is implemented in `RcppArmadillo` C++ code, so it's many times faster than the equivalent 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 `tseries`, and the number of rows equal to the number of end points.

Examples

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

## End(Not run)
```

`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(tseries, endp)
```

Arguments

`tseries`	A *time series* or a *matrix* with multiple columns of data.
`endp`	An *integer vector* of end points.

Details

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

Examples

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

## End(Not run)
```

`roll_reg` — *Calculate a* matrix *of regression coefficients, their t-values, and z-scores, at the end points of the predictor matrix.*

Description

Calculate a *matrix* of regression coefficients, their t-values, and z-scores, at the end points of the predictor matrix.

Usage

```
roll_reg(
  response,
  predictor,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "least_squares",
  eigen_thresh = 1e-05,
  eigen_max = 0L,
  conf_lev = 0.1,
  alpha = 0
)
```

Arguments

`response`	A single-column *time series* or a *vector* of response data.
`predictor`	A *time series* or a *matrix* of predictor data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endp`	An *integer* vector of end points (the default is `endp = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`look_back`	The number of end points in the look-back interval (the default is `look_back = 1`).
`stub`	An *integer* value equal to the first end point for calculating the end points (the default is `stub = 0`).
`method`	A *string* specifying the type of the regression model the default is `method = "least_squares"` - see Details).
`eigen_thresh`	A *numeric* threshold level for discarding small singular values in order to regularize the inverse of the `predictor` matrix (the default is `1e-5`).
`eigen_max`	An *integer* equal to the number of singular values used for calculating the shrinkage inverse of the `predictor` matrix (the default is `0` - equivalent to `eigen_max` equal to the number of columns of `predictor`).
`conf_lev`	The confidence level for calculating the quantiles (the default is `conf_lev = 0.75`).
`alpha`	The shrinkage intensity between `0` and `1`. (the default is `0`).

Details

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

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

It passes the subset time series to the function `calc_reg()`, which calculates the regression data.

If the arguments `endp` 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 `predictor` 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 `look_back = 3`.

Value

A *matrix* with the same number of rows as `predictor`, and a number of columns equal to `2n+3`, where `n` is the number of columns of `predictor`.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLP", "VTI")])
# Define monthly end points and start points
end_p <- xts::endpoints(re_turns, on="months")[-1]
look_back <- 12
start_p <- c(rep(1, look_back), end_p[1:(NROW(end_p)-look_back)])
# Calculate rolling betas using RcppArmadillo
reg_stats <- HighFreq::roll_reg(response=re_turns[, 1], predictor=re_turns[, 2], endp=(end_p-1), startp=(star
beta_s <- reg_stats[, 2]
# Calculate rolling betas in R
betas_r <- sapply(1:NROW(end_p), FUN=function(ep) {
  da_ta <- re_turns[start_p[ep]:end_p[ep], ]
  drop(cov(da_ta[, 1], da_ta[, 2])/var(da_ta[, 2]))
})  # end sapply
# Compare the outputs of both functions
all.equal(beta_s, betas_r, check.attributes=FALSE)

## End(Not run)
```

---

roll_scale — *Perform a rolling scaling (standardization) of the columns of a* matrix *of data using* `RcppArmadillo`.

Description

Perform a rolling scaling (standardization) of the columns of a *matrix* of data using `RcppArmadillo`.

Usage

```
roll_scale(matrix, look_back, use_median = FALSE)
```

Arguments

`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*). If `use_median` is `FALSE` then the centrality is calculated as the *mean* and the dispersion is calculated as the *standard deviation* (the default is `use_median = FALSE`)
`matrix`	A *matrix* of data.

`look_back`	The length of the look-back interval, equal to the number of rows of data used in the scaling.

Details

The function `roll_scale()` performs a rolling scaling (standardization) of the columns of the `matrix` argument using `RcppArmadillo`. The function `roll_scale()` performs a loop over the rows of `matrix`, subsets a number of previous (past) rows equal to `look_back`, and scales the subset matrix. It assigns the last row of the scaled subset *matrix* to the return matrix.

If the argument `use_median` is `FALSE` (the default), then it performs the same calculation as the function `roll::roll_scale()`. 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 `matrix`.

Examples

```
## Not run:
mat_rix <- matrix(rnorm(20000), nc=2)
look_back <- 11
rolled_scaled <- roll::roll_scale(data=mat_rix, width = look_back, min_obs=1)
rolled_scaled2 <- roll_scale(matrix=mat_rix, look_back = look_back, use_median=FALSE)
all.equal(rolled_scaled[-1, ], rolled_scaled2[-1, ])

## End(Not run)
```

`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(oh_lc, look_back = 11)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`look_back`	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 `oh_lc`.

Examples

```
# Calculate rolling Sharpe ratio over SPY
sharpe_rolling <- roll_sharpe(oh_lc=HighFreq::SPY, look_back=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(
  tseries,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "moment",
  conf_lev = 0.75
)
```

Arguments

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

The function `roll_skew()` performs a loop over the end points, and at each end point it subsets the time series `tseries` over a look-back interval equal to `look_back` 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 endp 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 `tseries` 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 `look_back = 3`.

The function `roll_skew()` is implemented in `RcppArmadillo` `C++` code, so it's many times faster than the equivalent `R` code.

Value

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

Examples

```
## Not run:
# Define time series of returns using package rutils
re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
# Define end points and start points
end_p <- 1 + HighFreq::calc_endpoints(NROW(re_turns), step=25)
start_p <- HighFreq::calc_startpoints(end_p, look_back=3)
# Calculate the rolling skewness at 25 day end points, with a 75 day look-back
skew_ness <- HighFreq::roll_skew(re_turns, step=25, look_back=3)
# Calculate the rolling skewness using R code
skew_r <- sapply(1:NROW(end_p), function(it) {
  HighFreq::calc_skew(re_turns[start_p[it]:end_p[it], ])
})  # end sapply
# Compare the skewness estimates
all.equal(drop(skew_ness), skew_r, check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_skew(re_turns, step=25, look_back=3),
  Rcode=sapply(1:NROW(end_p), function(it) {
    HighFreq::calc_skew(re_turns[start_p[it]:end_p[it], ])
  }),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

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(
  oh_lc,
  calc_stats = "ohlc_variance",
  look_back = 11,
  weight_ed = TRUE,
  ...
)
```

Arguments

...	additional parameters to the function `calc_stats`.
oh_lc	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.
look_back	The size of the look-back interval, equal to the number of rows of data used for calculating the rolling mean.
weight_ed	*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 `oh_lc`. 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 `oh_lc`.

Examples

```
# Calculate time series of rolling variance and skew estimates
var_rolling <- roll_stats(oh_lc=HighFreq::SPY, look_back=21)
skew_rolling <- roll_stats(oh_lc=HighFreq::SPY, calc_stats="ohlc_skew", look_back=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(tseries, look_back = 1L)
```

Arguments

tseries	A *time series* or a *matrix*.
look_back	The length of the look-back interval, equal to the number of data points included in calculating the rolling sum (the default is `look_back = 1`).

Details

The function `roll_sum()` calculates the rolling sums over the columns of the data `tseries`.

The function `roll_sum()` returns a *matrix* with the same dimensions as the input argument `tseries`.

The function `roll_sum()` uses the fast `RcppArmadillo` function `arma::cumsum()`, without explicit loops. The function `roll_sum()` is several times faster than `rutils::roll_sum()` which uses vectorized `R` code.

Value

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

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("VTI", "IEF")])
# Define parameters
look_back <- 22
# Calculate rolling sums and compare with rutils::roll_sum()
c_sum <- HighFreq::roll_sum(re_turns, look_back)
r_sum <- rutils::roll_sum(re_turns, look_back)
all.equal(c_sum, coredata(r_sum), check.attributes=FALSE)
# Calculate rolling sums using R code
r_sum <- apply(zoo::coredata(re_turns), 2, cumsum)
lag_sum <- rbind(matrix(numeric(2*look_back), nc=2), r_sum[1:(NROW(r_sum) - look_back), ])
r_sum <- (r_sum - lag_sum)
all.equal(c_sum, r_sum, check.attributes=FALSE)

## End(Not run)
```

`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(
  tseries,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L
)
```

Arguments

`tseries`	A *time series* or a *matrix*.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endp`	An *integer* vector of end points (the default is `endp = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`look_back`	The number of end points in the look-back interval (the default is `look_back = 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* `tseries`.

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 `tseries`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("VTI", "IEF")])
# Define end points at 25 day intervals
end_p <- HighFreq::calc_endpoints(NROW(re_turns), step=25)
# Define start points as 75 day lag of end points
start_p <- HighFreq::calc_startpoints(end_p, look_back=3)
# Calculate rolling sums using Rcpp
c_sum <- HighFreq::roll_sumep(re_turns, startp=start_p, endp=end_p)
```

```
# Calculate rolling sums using R code
r_sum <- sapply(1:NROW(end_p), function(ep) {
colSums(re_turns[(start_p[ep]+1):(end_p[ep]+1), ])
  })  # end sapply
r_sum <- t(r_sum)
all.equal(c_sum, r_sum, check.attributes=FALSE)

## End(Not run)
```

`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(
  tseries,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "moment",
  conf_lev = 0.75
)
```

Arguments

`tseries`	A *time series* or a *matrix* of data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endp`	An *integer* vector of end points (the default is `endp = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`look_back`	The number of end points in the look-back interval (the default is `look_back = 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* `tseries`.

The function `roll_var()` performs a loop over the end points, and at each end point it subsets the time series `tseries` over a look-back interval equal to `look_back` 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 `endp` 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 `tseries` 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 `look_back = 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` `C++` code, so it's many times faster than the equivalent `R` code.

Value

A *matrix* dispersion (variance) estimates with the same number of columns as the input time series `tseries`, and the number of rows equal to the number of end points.

Examples

```
## Not run:
# Define time series of returns using package rutils
re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
# Calculate the rolling variance at 25 day end points, with a 75 day look-back
vari_ance <- HighFreq::roll_var(re_turns, step=25, look_back=3)
# Compare the variance estimates over 11-period look-back intervals
all.equal(HighFreq::roll_var(re_turns, look_back=11)[-(1:10), ],
  drop(RcppRoll::roll_var(re_turns, n=11)), check.attributes=FALSE)
# Compare the speed of HighFreq::roll_var() with RcppRoll::roll_var()
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_var(re_turns, look_back=11),
  RcppRoll=RcppRoll::roll_var(re_turns, 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(re_turns, look_back=11, method="quantile"),
    TTR=TTR::runMAD(re_turns, n = 11),
    times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

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,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L,
  method = "yang_zhang",
  scale = TRUE,
  in_dex = 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`).
endp	An *integer* vector of end points (the default is `endp = 0`).
step	The number of time periods between the end points (the default is `step = 1`).
look_back	The number of end points in the look-back interval (the default is `look_back = 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`.)
in_dex	A *vector* with the time index of the *time series*. This is an optional argument (the default is `in_dex=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 look_back number of end points. In the initial warmup period, the variance is calculated over an expanding look-back interval.

If the arguments endp 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 look_back = 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 look_back = 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 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 optional argument in_dex 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, so it's many times faster than the equivalent 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
oh_lc <- log(HighFreq::SPY)
# Extract the time index of SPY prices
in_dex <- c(1, diff(xts::.index(oh_lc)))
# Rolling variance at minutely end points, with a 21 minute look-back
var_rolling <- HighFreq::roll_var_ohlc(oh_lc,
                              step=1, look_back=21,
                              method="yang_zhang",
```

```
                              in_dex=in_dex, scale=TRUE)
# Daily OHLC prices
oh_lc <- rutils::etf_env$VTI
in_dex <- c(1, diff(xts::.index(oh_lc)))
# Rolling variance at 5 day end points, with a 20 day look-back (20=4*5)
var_rolling <- HighFreq::roll_var_ohlc(oh_lc,
                              step=5, look_back=4,
                              method="yang_zhang",
                              in_dex=in_dex, scale=TRUE)
# Same calculation in R
n_rows <- NROW(oh_lc)
lag_close = HighFreq::lag_it(oh_lc[, 4])
end_p <- drop(HighFreq::calc_endpoints(n_rows, 3)) + 1
start_p <- drop(HighFreq::calc_startpoints(end_p, 2))
n_pts <- NROW(end_p)
var_rollingr <- sapply(2:n_pts, function(it) {
  ran_ge <- start_p[it]:end_p[it]
  sub_ohlc = oh_lc[ran_ge, ]
  sub_close = lag_close[ran_ge]
  sub_index = in_dex[ran_ge]
  HighFreq::calc_var_ohlc(sub_ohlc, lag_close=sub_close, scale=TRUE, in_dex=sub_index)
})  # end sapply
var_rollingr <- c(0, var_rollingr)
all.equal(drop(var_rolling), var_rollingr)

## End(Not run)
```

`roll_var_vec` *Calculate a* vector *of variance estimates over a rolling look-back interval for a single-column* time series *or a* column vector*, using* `RcppArmadillo`.

Description

Calculate a *vector* of variance estimates over a rolling look-back interval for a single-column *time series* or a *column vector*, using `RcppArmadillo`.

Usage

```
roll_var_vec(tseries, look_back = 1L)
```

Arguments

`tseries`	A single-column *time series* or a *column vector*.
`look_back`	The length of the look-back interval, equal to the number of *vector* elements used for calculating a single variance estimate (the default is `look_back = 1`).

Details

The function `roll_var_vec()` calculates a *vector* of variance estimates over a rolling look-back interval for a single-column *time series* or a *column vector*, using `RcppArmadillo` `C++` code.

The function `roll_var_vec()` uses an expanding look-back interval in the initial warmup period, to calculate the same number of elements as the input argument `tseries`.

The function `roll_var_vec()` 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 *column vector* with the same number of elements as the input argument `tseries`.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare the variance estimates over 11-period look-back intervals
all.equal(drop(HighFreq::roll_var_vec(re_turns, look_back=11))[-(1:10)],
  RcppRoll::roll_var(re_turns, n=11))
# Compare the speed of RcppArmadillo with RcppRoll
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_var_vec(re_turns, look_back=11),
  RcppRoll=RcppRoll::roll_var(re_turns, n=11),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`roll_vec` *Calculate the rolling sums over a single-column* time series *or a* column vector *using* Rcpp.

Description

Calculate the rolling sums over a single-column *time series* or a *column vector* using *Rcpp*.

Usage

```
roll_vec(tseries, look_back)
```

Arguments

`tseries`	A single-column *time series* or a *column vector* (a single-column matrix).
`look_back`	The length of the look-back interval, equal to the number of elements of data used for calculating the sum.

Details

The function `roll_vec()` calculates a *column vector* of rolling sums, over a *column vector* of data, using fast *Rcpp* `C++` code. The function `roll_vec()` is several times faster than `rutils::roll_sum()` which uses vectorized `R` code.

Value

A *column vector* of the same length as the argument `tseries`.

Examples

```
## Not run:
# Define a single-column matrix of returns
re_turns <- zoo::coredata(na.omit(rutils::etf_env$re_turns$VTI))
# Calculate rolling sums over 11-period look-back intervals
sum_rolling <- HighFreq::roll_vec(re_turns, look_back=11)
# Compare HighFreq::roll_vec() with rutils::roll_sum()
all.equal(HighFreq::roll_vec(re_turns, look_back=11),
         rutils::roll_sum(re_turns, look_back=11),
         check.attributes=FALSE)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_vec(re_turns, look_back=11),
  Rcode=rutils::roll_sum(re_turns, look_back=11),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`roll_vecw`	*Calculate the rolling weighted sums over a single-column* time series *or a* column vector *using* `RcppArmadillo`.

Description

Calculate the rolling weighted sums over a single-column *time series* or a *column vector* using `RcppArmadillo`.

Usage

```
roll_vecw(tseries, weights)
```

Arguments

`tseries`	A single-column *time series* or a *column vector* (a single-column matrix).
`weights`	A *column vector* of weights.

Details

The function `roll_vecw()` calculates the rolling weighted sums of a *column vector* over its past values (a convolution with the *column vector* of weights), using `RcppArmadillo`. It performs a similar calculation as the standard `R` function
`stats::filter(x=series,filter=weight_s,method="convolution",sides=1)`, but it's over `6` times faster, and it doesn't produce any `NA` values.

Value

A *column vector* of the same length as the argument `tseries`.

Examples

```
## Not run:
# First example
# Define a single-column matrix of returns
re_turns <- zoo::coredata(na.omit(rutils::etf_env$re_turns$VTI))
# Create simple weights
weight_s <- matrix(c(1, rep(0, 10)))
# Calculate rolling weighted sums
weight_ed <- HighFreq::roll_vecw(tseries=re_turns, weights=weight_s)
# Compare with original
all.equal(zoo::coredata(re_turns), weight_ed, check.attributes=FALSE)
# Second example
# Create exponentially decaying weights
weight_s <- matrix(exp(-0.2*1:11))
weight_s <- weight_s/sum(weight_s)
# Calculate rolling weighted sums
weight_ed <- HighFreq::roll_vecw(tseries=re_turns, weights=weight_s)
# Calculate rolling weighted sums using filter()
filter_ed <- stats::filter(x=re_turns, filter=weight_s, method="convolution", sides=1)
# Compare both methods
all.equal(filter_ed[-(1:11)], weight_ed[-(1:11)], check.attributes=FALSE)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::roll_vecw(tseries=re_turns, weights=weight_s),
  Rcode=stats::filter(x=re_turns, filter=weight_s, method="convolution", sides=1),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`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(oh_lc, close = oh_lc[, 4, drop = FALSE], look_back)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`close`	A time series of close prices.
`look_back`	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 oh_lc.

Examples

```
# Calculate and plot rolling volume-weighted average closing prices (VWAP)
prices_rolling <- roll_vwap(oh_lc=HighFreq::SPY["2013-11"], look_back=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(x_ts=HighFreq::SPY)
# Calculate the rolling volume-weighted average returns
roll_vwap(oh_lc=HighFreq::SPY, close=returns_running, look_back=11)
```

---

roll_wsum — *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_wsum(tseries, endp = NULL, look_back = 1L, stub = NULL, weights = NULL)
```

Arguments

tseries	A *time series* or a *matrix*.
endp	An *integer* vector of end points (the default is endp = NULL).
look_back	The length of the look-back interval, equal to the number of data points included in calculating the rolling sum (the default is look_back = 1).
stub	An *integer* value equal to the first end point for calculating the end points (the default is stub = NULL).
weights	A *column vector* of weights (the default is weights = NULL).

Details

The function roll_wsum() calculates the rolling weighted sums over the columns of the data tseries.

The function roll_wsum() calculates the rolling weighted sums as convolutions of the columns of tseries with the *column vector* of weights using the RcppArmadillo function arma::conv2(). It performs a similar calculation to the standard R function
stats::filter(x=re_turns,filter=weight_s,method="convolution",sides=1), but it can be many times faster, and it doesn't produce any leading NA values.

The function roll_wsum() returns a *matrix* with the same dimensions as the input argument tseries.

The arguments weights, endp, and stub are optional.

If the argument weights is not supplied, then simple sums are calculated, not weighted sums.

If either the stub or endp 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 look_back arguments. The first end point is equal to stub and the end points are spaced look_back periods apart.

If the arguments weights, endp, and stub are not supplied, then the sums are calculated over a number of data points equal to look_back.

The function roll_wsum() is also several times faster than rutils::roll_sum() which uses vectorized R code.

Technical note: The function roll_wsum() 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 tseries.

Examples

```
## Not run:
# First example
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("VTI", "IEF")])
# Define parameters
look_back <- 22
# Calculate rolling sums and compare with rutils::roll_sum()
c_sum <- HighFreq::roll_sum(re_turns, look_back)
r_sum <- rutils::roll_sum(re_turns, look_back)
all.equal(c_sum, coredata(r_sum), check.attributes=FALSE)
# Calculate rolling sums using R code
r_sum <- apply(zoo::coredata(re_turns), 2, cumsum)
lag_sum <- rbind(matrix(numeric(2*look_back), nc=2), r_sum[1:(NROW(r_sum) - look_back), ])
r_sum <- (r_sum - lag_sum)
all.equal(c_sum, r_sum, check.attributes=FALSE)

# Calculate rolling sums at end points
stu_b <- 21
c_sum <- HighFreq::roll_wsum(re_turns, look_back, stub=stu_b)
end_p <- (stu_b + look_back*(0:(NROW(re_turns) %/% look_back)))
end_p <- end_p[end_p < NROW(re_turns)]
r_sum <- apply(zoo::coredata(re_turns), 2, cumsum)
```

```
r_sum <- r_sum[end_p+1, ]
lag_sum <- rbind(numeric(2), r_sum[1:(NROW(r_sum) - 1), ])
r_sum <- (r_sum - lag_sum)
all.equal(c_sum, r_sum, check.attributes=FALSE)

# Calculate rolling sums at end points - pass in end_p
c_sum <- HighFreq::roll_wsum(re_turns, endp=end_p)
all.equal(c_sum, r_sum, check.attributes=FALSE)

# Create exponentially decaying weights
weight_s <- exp(-0.2*(1:11))
weight_s <- matrix(weight_s/sum(weight_s), nc=1)
# Calculate rolling weighted sum
c_sum <- HighFreq::roll_wsum(re_turns, weights=weight_s)
# Calculate rolling weighted sum using filter()
filter_ed <- filter(x=re_turns, filter=weight_s, method="convolution", sides=1)
all.equal(c_sum[-(1:11), ], filter_ed[-(1:11), ], check.attributes=FALSE)

# Calculate rolling weighted sums at end points
c_sum <- HighFreq::roll_wsum(re_turns, endp=end_p, weights=weight_s)
all.equal(c_sum, filter_ed[end_p+1, ], check.attributes=FALSE)

# Create simple weights equal to a 1 value plus zeros
weight_s <- matrix(c(1, rep(0, 10)), nc=1)
# Calculate rolling weighted sum
weight_ed <- HighFreq::roll_wsum(re_turns, weights=weight_s)
# Compare with original
all.equal(coredata(re_turns), weight_ed, check.attributes=FALSE)

## End(Not run)
```

`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(
  response,
  predictor,
  startp = 0L,
  endp = 0L,
  step = 1L,
  look_back = 1L,
  stub = 0L
)
```

Arguments

`response`	A single-column *time series* or a *vector* of response data.
`predictor`	A *time series* or a *matrix* of predictor data.
`startp`	An *integer* vector of start points (the default is `startp = 0`).
`endp`	An *integer* vector of end points (the default is `endp = 0`).
`step`	The number of time periods between the end points (the default is `step = 1`).
`look_back`	The number of end points in the look-back interval (the default is `look_back = 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* `predictor`.

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

It passes the subset time series to the function `calc_lm()`, which calculates the regression data.

If the arguments `endp` 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 `predictor` 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 `look_back = 3`.

Value

A column *vector* of the same length as the number of rows of `predictor`.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
res_ponse <- re_turns[, 1]
# Predictor matrix equals VTI and IEF returns
predic_tor <- re_turns[, -1]
# Calculate Z-scores from rolling time series regression using RcppArmadillo
look_back <- 11
z_scores <- HighFreq::roll_zscores(response=res_ponse, predictor=predic_tor, look_back)
# Calculate z-scores in R from rolling multivariate regression using lm()
z_scoresr <- sapply(1:NROW(predic_tor), function(ro_w) {
  if (ro_w == 1) return(0)
  start_point <- max(1, ro_w-look_back+1)
  sub_response <- res_ponse[start_point:ro_w]
  sub_predictor <- predic_tor[start_point:ro_w, ]
  reg_model <- lm(sub_response ~ sub_predictor)
  resid_uals <- reg_model$residuals
  resid_uals[NROW(resid_uals)]/sd(resid_uals)
})  # end sapply
```

```
# Compare the outputs of both functions
all.equal(z_scores[-(1:look_back)], z_scoresr[-(1:look_back)],
  check.attributes=FALSE)

## End(Not run)
```

---

run_covar   *Calculate the running covariance of two streaming* time series *of returns.*

Description

Calculate the running covariance of two streaming *time series* of returns.

Usage

```
run_covar(tseries, lambda)
```

Arguments

tseries	A *time series* or a *matrix* with two columns of returns data.
lambda	A *numeric* decay factor to multiply past estimates.

Details

The function `run_covar()` calculates the running covariance of two streaming *time series* of returns, by recursively weighing the products of their returns minus their means, with past covariance estimates σ^{cov}_{t-1}, using the decay factor λ:

$$\mu^1_t = (1 - \lambda) r^1_t + \lambda \mu^1_{t-1}$$

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

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

Where σ^{cov}_t is the covariance estimate at time t, r^1_t and r^2_t are the two streaming returns data, and μ^1_t and μ^2_t are the means of the returns.

The value of the decay factor λ should 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 running covariance values have a stronger dependence on past values. 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 running covariance values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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.

The function `run_covar()` returns three columns of data: the covariance and the variances of the two columns of the argument `tseries`. This allows calculating the running correlation.

Value

A *matrix* with three columns of data: the covariance and the variances of the two columns of the argument `tseries`.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- zoo::coredata(na.omit(rutils::etf_env$re_turns[, c("IEF", "VTI")]))
# Calculate the running covariance
lamb_da <- 0.9
covars <- HighFreq::run_covar(re_turns, lambda=lamb_da)
# Calculate running covariance using R code
filter_ed <- (1-lamb_da)*filter(re_turns[, 1]*re_turns[, 2],
  filter=lamb_da, init=as.numeric(re_turns[1, 1]*re_turns[1, 2])/(1-lamb_da),
  method="recursive")
all.equal(covars[, 1], unclass(filter_ed), check.attributes=FALSE)
# Calculate the running correlation
correl <- covars[, 1]/sqrt(covars[, 2]*covars[, 3])

## End(Not run)
```

`run_max` *Calculate the rolling maximum of streaming* time series *data.*

Description

Calculate the rolling maximum of streaming *time series* data.

Usage

```
run_max(tseries, lambda)
```

Arguments

`tseries`	A *time series* or a *matrix*.
`lambda`	A *numeric* decay factor to multiply past estimates.

Details

The function `run_max()` calculates the rolling maximum of streaming *time series* data by recursively weighing present and past values using the decay factor λ.

It first calculates the rolling mean of streaming data:

$$\mu_t = (1 - \lambda) p_t + \lambda \mu_{t-1}$$

Where μ_t is the mean value at time t, and p_t is the streaming data.

It then calculates the rolling maximums of streaming data, p^{max}_t:

$$p^{max}_t = max(p_t, p^{max}_{t-1}) + (1 - \lambda)(\mu_{t-1} - p^{max}_{t-1})$$

The second term pulls the maximum value down to the mean value, allowing it to gradually "forget" the maximum value from the more distant past.

The value of the decay factor λ should 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 rolling maximum values have a stronger dependence on past values. 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 rolling maximum values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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 `tseries`.

Value

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

Examples

```
## Not run:
# Calculate historical prices
price_s <- zoo::coredata(quantmod::Cl(rutils::etf_env$VTI))
# Calculate the rolling maximums
lamb_da <- 0.9
maxs <- HighFreq::run_max(price_s, lambda=lamb_da)
# Plot dygraph of VTI prices and rolling maximums
da_ta <- cbind(quantmod::Cl(rutils::etf_env$VTI), maxs)
colnames(da_ta) <- c("prices", "max")
col_names <- colnames(da_ta)
dygraphs::dygraph(da_ta, main="VTI Prices and Rolling Maximums") %>%
  dySeries(name=col_names[1], label=col_names[1], strokeWidth=2, col="blue") %>%
  dySeries(name=col_names[2], label=col_names[2], strokeWidth=2, col="red")

## End(Not run)
```

`run_mean` *Calculate the rolling mean of streaming* time series *data.*

Description

Calculate the rolling mean of streaming *time series* data.

Usage

```
run_mean(tseries, lambda)
```

Arguments

`tseries`	A *time series* or a *matrix*.
`lambda`	A *numeric* decay factor to multiply past estimates.

Details

The function run_mean() calculates the rolling mean of streaming *time series* data by recursively weighing present and past values using the decay factor λ:

$$\mu_t = (1 - \lambda) p_t + \lambda \mu_{t-1}$$

Where μ_t is the mean value at time t, and p_t is the streaming data.

The value of the decay factor λ should 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 rolling mean values have a stronger dependence on past values. 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 rolling mean values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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.

The function run_mean() performs the same calculation as the standard R function stats::filter(x=series,filter=lamb_da,method="recursive"), but it's several times faster.

The function run_mean() returns a *matrix* with the same dimensions as the input argument tseries.

Value

A *matrix* with the same dimensions as the input argument tseries.

Examples

```
## Not run:
# Calculate historical prices
price_s <- zoo::coredata(quantmod::Cl(rutils::etf_env$VTI))
# Calculate the rolling means
lamb_da <- 0.9
means <- HighFreq::run_mean(price_s, lambda=lamb_da)
# Calculate rolling means using R code
filter_ed <- (1-lamb_da)*filter(price_s,
  filter=lamb_da, init=as.numeric(price_s[1, 1])/(1-lamb_da),
  method="recursive")
all.equal(means, unclass(filter_ed), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::run_mean(price_s, lambda=lamb_da),
 Rcode=filter(price_s, filter=lamb_da, init=as.numeric(price_s[1, 1])/(1-lamb_da), method="recursive"),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

run_min — *Calculate the rolling minimum of streaming* time series *data.*

Description

Calculate the rolling minimum of streaming *time series* data.

Usage

```
run_min(tseries, lambda)
```

Arguments

tseries	A *time series* or a *matrix*.
lambda	A *numeric* decay factor to multiply past estimates.

Details

The function `run_min()` calculates the rolling minimum of streaming *time series* data by recursively weighing present and past values using the decay factor λ.

It first calculates the rolling mean of streaming data:

$$\mu_t = (1 - \lambda) p_t + \lambda \mu_{t-1}$$

Where μ_t is the mean value at time t, and p_t is the streaming data.

It then calculates the rolling minimums of streaming data, p^{min}_t:

$$p^{min}_t = min(p_t, p^{min}_{t-1}) + (1 - \lambda)(\mu_{t-1} - p^{min}_{t-1})$$

The second term pulls the minimum value up to the mean value, allowing it to gradually "forget" the minimum value from the more distant past.

The value of the decay factor λ should 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 rolling minimum values have a stronger dependence on past values. 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 rolling minimum values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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 `tseries`.

Value

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

Examples

```
## Not run:
# Calculate historical prices
price_s <- zoo::coredata(quantmod::Cl(rutils::etf_env$VTI))
# Calculate the rolling minimums
lamb_da <- 0.9
mins <- HighFreq::run_min(price_s, lambda=lamb_da)
# Plot dygraph of VTI prices and rolling minimums
da_ta <- cbind(quantmod::Cl(rutils::etf_env$VTI), mins)
colnames(da_ta) <- c("prices", "min")
col_names <- colnames(da_ta)
dygraphs::dygraph(da_ta, main="VTI Prices and Rolling Minimums") %>%
  dySeries(name=col_names[1], label=col_names[1], strokeWidth=2, col="blue") %>%
  dySeries(name=col_names[2], label=col_names[2], strokeWidth=2, col="red")

## End(Not run)
```

`run_reg`	*Perform running regressions of streaming* time series *of response and predictor data, and calculate the alphas, betas, and the residuals.*

Description

Perform running regressions of streaming *time series* of response and predictor data, and calculate the alphas, betas, and the residuals.

Usage

```
run_reg(response, predictor, lambda, method = "none")
```

Arguments

`response`	A single-column *time series* or a single-column *matrix* of response data.
`predictor`	A *time series* or a *matrix* of predictor data.
`lambda`	A *numeric* decay factor to multiply past estimates.
`method`	A *string* specifying the method for scaling the residuals (see Details) (the default is `method = "none"` - no scaling)

Details

The function `run_reg()` calculates the vectors of *alphas* α_t, *betas* β_t, and the *residuals* ϵ_t of running regressions, by recursively weighing the current estimates with past estimates, using the decay factor λ:

$$\mu_t^r = (1 - \lambda) r_t^r + \lambda \mu_{t-1}^r$$

$$\mu_t^p = (1 - \lambda) r_t^p + \lambda \mu_{t-1}^p$$

$$\sigma_t^2 = (1 - \lambda)(r_t^{p2} - \mu_t^{p2}) + \lambda \sigma_{t-1}^2$$

$$\sigma_t^{cov} = (1 - \lambda)(r_t^r - \mu_t^r)(r_t^p - \mu_t^p) + \lambda \sigma_{t-1}^{cov}$$

$$\beta_t = (1 - \lambda) \frac{\sigma_t^{cov}}{\sigma_t^2} + \lambda \beta_{t-1}$$

$$\epsilon_t = (1 - \lambda)(r_t^r - \beta_t r_t^p) + \lambda \epsilon_{t-1}$$

Where σ_t^{cov} are the covariances between the response and predictor data at time t; σ_t^2 is the vector of predictor variances, and r_t^r and r_t^p are the streaming data of the response and predictor data.

The matrices σ^2, σ^{cov}, and β have the same dimensions as the input argument `predictor`.

The value of the decay factor λ should 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 running *z-score* values have a stronger dependence on past values. 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 running *z-score* values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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 to calculate.

The *residuals* may be scaled by their volatilities. The default is method = "none" - no scaling. If the argument method = "scale" then the *residuals* ϵ_t are divided by their volatilities σ^ϵ without subtracting their means:

$$\epsilon_t = \frac{\epsilon_t}{\sigma^\epsilon}$$

If the argument method = "standardize" then the means μ_ϵ are subtracted from the *residuals*, and then they are divided by their volatilities σ^ϵ:

$$\epsilon_t = \frac{\epsilon_t - \mu_\epsilon}{\sigma^\epsilon}$$

The function run_reg() returns multiple columns of data. If the matrix predictor has n columns then run_reg() returns a matrix with n+2 columns. The first column contains the *residuals*, the second the *alphas*, and the last columns contain the *betas*.

Value

A *matrix* with the regression alphas, betas, and residuals.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
res_ponse <- re_turns[, 1]
# Predictor matrix equals VTI and IEF returns
predic_tor <- re_turns[, -1]
# Calculate the running regressions
lamb_da <- 0.9
regs <- HighFreq::run_reg(response=res_ponse, predictor=predic_tor, lambda=lamb_da)
# Plot the running alphas
da_ta <- cbind(cumsum(res_ponse), regs[, 1])
colnames(da_ta) <- c("XLF", "alphas")
col_names <- colnames(da_ta)
dygraphs::dygraph(da_ta, main="Alphas of XLF Versus VTI and IEF") %>%
  dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
  dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
 dySeries(name=col_names[1], axis="y", label=col_names[1], strokeWidth=1, col="blue") %>%
  dySeries(name=col_names[2], axis="y2", label=col_names[2], strokeWidth=1, col="red")

## End(Not run)
```

---

run_var *Calculate the running variance of streaming* time series *of returns.*

Description

Calculate the running variance of streaming *time series* of returns.

Usage

```
run_var(tseries, lambda)
```

Arguments

`tseries`	A *time series* or a *matrix* of returns.
`lambda`	A *numeric* decay factor to multiply past estimates.

Details

The function `run_var()` calculates the running variance of a streaming *time series* of returns, by recursively weighing the squared returns r_t^2 minus the squared means μ_t^2, with the past variance estimates σ_{t-1}^2, using the decay factor λ:

$$\mu_t = (1 - \lambda) r_t + \lambda \mu_{t-1}$$

$$\sigma_t^2 = (1 - \lambda)(r_t^2 - \mu_t^2) + \lambda \sigma_{t-1}^2$$

Where σ_t^2 is the variance estimate at time t, and r_t are the streaming returns data.

The value of the decay factor λ should 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 running variance values have a stronger dependence on past values. 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 running variance values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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.

The function `run_var()` performs the same calculation as the standard `R` function `stats::filter(x=series,filter=weight_s,method="recursive")`, but it's several times faster.

The function `run_var()` returns a *matrix* with the same dimensions as the input argument `tseries`.

Value

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

Examples

```
## Not run:
# Calculate historical returns
re_turns <- zoo::coredata(na.omit(rutils::etf_env$re_turns$VTI))
# Calculate the running variance
lamb_da <- 0.9
vars <- HighFreq::run_var(re_turns, lambda=lamb_da)
# Calculate running variance using R code
filter_ed <- (1-lamb_da)*filter(re_turns^2, filter=lamb_da,
  init=as.numeric(re_turns[1, 1])^2/(1-lamb_da),
  method="recursive")
all.equal(vars, unclass(filter_ed), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::run_var(re_turns, lambda=lamb_da),
 Rcode=filter(re_turns^2, filter=lamb_da, init=as.numeric(re_turns[1, 1])^2/(1-lamb_da), method="recursive")
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`run_var_ohlc` *Calculate the running variance of streaming* OHLC *price data.*

Description

Calculate the running variance of streaming *OHLC* price data.

Usage

```
run_var_ohlc(ohlc, lambda)
```

Arguments

`ohlc`	A *time series* or a *matrix* with *OHLC* price data.
`lambda`	A *numeric* decay factor to multiply 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 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, so it's many times faster than the equivalent `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
oh_lc <- log(rutils::etf_env$VTI)
# Calculate the running variance
var_running <- HighFreq::run_var_ohlc(oh_lc, lambda=0.8)
# Calculate the rolling variance
var_rolling <- HighFreq::roll_var_ohlc(oh_lc, look_back=5, method="yang_zhang", scale=FALSE)
da_ta <- cbind(var_running, var_rolling)
colnames(da_ta) <- c("running", "rolling")
col_names <- colnames(da_ta)
da_ta <- xts::xts(da_ta, index(oh_lc))
# dygraph plot of VTI running versus rolling volatility
```

```
dygraphs::dygraph(sqrt(da_ta[-(1:111), ]), main="Running and Rolling Volatility of VTI") %>%
  dyOptions(colors=c("red", "blue"), strokeWidth=1) %>%
  dyLegend(show="always", width=500)
# Compare the speed of running versus rolling volatility
library(microbenchmark)
summary(microbenchmark(
  running=HighFreq::run_var_ohlc(oh_lc, lambda=0.8),
  rolling=HighFreq::roll_var_ohlc(oh_lc, look_back=5, method="yang_zhang", scale=FALSE),
  times=10))[, c(1, 4, 5)]

## End(Not run)
```

`run_zscores` *Calculate the z-scores of running regressions of streaming* time series *of returns.*

Description

Calculate the z-scores of running regressions of streaming *time series* of returns.

Usage

```
run_zscores(response, predictor, lambda, demean = TRUE)
```

Arguments

`response`	A single-column *time series* or a single-column *matrix* of response data.
`predictor`	A *time series* or a *matrix* of predictor data.
`lambda`	A *numeric* decay factor to multiply past estimates.
`demean`	A *Boolean* specifying whether the *z-scores* should be de-meaned (the default is `demean = TRUE`).

Details

The function `run_zscores()` calculates the vectors of *betas* β_t and the residuals ϵ_t of running regressions by recursively weighing the current estimates with past estimates, using the decay factor λ:

$$\sigma^2_t = (1 - \lambda) r^{p2}_t + \lambda \sigma^2_{t-1}$$

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

$$\beta_t = (1 - \lambda) \frac{\sigma^{cov}_t}{\sigma^2_t} + \lambda \beta_{t-1}$$

$$\epsilon_t = (1 - \lambda)(r^r_t - \beta_t r^p_t) + \lambda \epsilon_{t-1}$$

Where σ^{cov}_t is the vector of covariances between the response and predictor returns, at time t; σ^2_t is the vector of predictor variances, and r^r_t and r^p_t are the streaming returns of the response and predictor data.

The above formulas for σ^2 and σ^{cov} are approximate because they don't subtract the means before squaring the returns. But they're very good approximations for daily returns.

The matrices σ^2, σ^{cov}, β have the same dimensions as the input argument `predictor`.

If the argument demean = TRUE (the default) then the *z-scores* z_t are calculated as equal to the residuals ϵ_t minus their means μ_ϵ, divided by their volatilities σ^ϵ:

$$z_t = \frac{\epsilon_t - \mu_\epsilon}{\sigma^\epsilon}$$

If the argument demean = FALSE then the *z-scores* are only divided by their volatilities without subtracting their means:

$$z_t = \frac{\epsilon_t}{\sigma^\epsilon}$$

The value of the decay factor λ should 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 running *z-score* values have a stronger dependence on past values. 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 running *z-score* values have a weaker dependence on past values. This is equivalent to a short look-back interval.

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 to calculate.

The function run_zscores() returns multiple columns of data. If the matrix predictor has n columns then run_zscores() returns a matrix with 2n+1 columns. The first column contains the *z-scores*, and the remaining columns contain the *betas* and the *variances* of the predictor data.

Value

A *matrix* with the z-scores, betas, and the variances of the predictor data.

Examples

```
## Not run:
# Calculate historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("XLF", "VTI", "IEF")])
# Response equals XLF returns
res_ponse <- re_turns[, 1]
# Predictor matrix equals VTI and IEF returns
predic_tor <- re_turns[, -1]
# Calculate the running z-scores
lamb_da <- 0.9
zscores <- HighFreq::run_zscores(response=res_ponse, predictor=predic_tor, lambda=lamb_da)
# Plot the running z-scores
da_ta <- cbind(cumsum(res_ponse), zscores[, 1])
colnames(da_ta) <- c("XLF", "zscores")
col_names <- colnames(da_ta)
dygraphs::dygraph(da_ta, main="Z-Scores of XLF Versus VTI and IEF") %>%
  dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
  dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
 dySeries(name=col_names[1], axis="y", label=col_names[1], strokeWidth=1, col="blue") %>%
  dySeries(name=col_names[2], axis="y2", label=col_names[2], strokeWidth=1, col="red")

## End(Not run)
```

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(
  sym_bol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  look_back = 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(
  sym_bol,
  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(
  sym_bol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  look_back = 51,
  vol_mult = 2,
  period = "minutes",
  tzone = "America/New_York"
)
```

Arguments

`sym_bol`	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/"
sym_bol <- "SPY"
# Aggregate SPY TAQ data to 15-min OHLC bar data, and save the data to a file
save_scrub_agg(sym_bol=sym_bol, 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(
  sym_bol,
  data_dir = "E:/mktdata/sec/",
  output_dir = "E:/output/data/",
  look_back = 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(
  ta_q,
  look_back = 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-offer 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
ta_q <- HighFreq::random_taq()
# Aggregate to ten minutes OHLC data
oh_lc <- HighFreq::scrub_agg(ta_q, period="10 min")
chart_Series(oh_lc, name="random prices")
# scrub and aggregate a single day of SPY TAQ data to OHLC
oh_lc <- HighFreq::scrub_agg(ta_q=HighFreq::SPY_TAQ)
chart_Series(oh_lc, name=sym_bol)
```

`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(ta_q, look_back = 51, vol_mult = 2, tzone = "America/New_York")
```

Arguments

`ta_q`	*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

```
ta_q <- HighFreq::scrub_taq(ta_q=HighFreq::SPY_TAQ, look_back=11, vol_mult=1)
# Create random TAQ prices and scrub them
ta_q <- HighFreq::random_taq()
ta_q <- HighFreq::scrub_taq(ta_q=ta_q)
ta_q <- HighFreq::scrub_taq(ta_q=ta_q, look_back=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(x_ts, in_dex = format(zoo::index(x_ts), "%H:%M"))
```

Arguments

x_ts	A single-column *xts* time series.
in_dex	A vector of *character* strings representing points in time, of the same length as the argument x_ts.

Details

The function season_ality() calculates the mean of values observed at the same points in time specified by the argument in_dex. 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 in_dex is passed into function tapply(), and must be the same length as the argument x_ts.

Value

An *xts* time series with mean aggregations over the seasonality interval.

Examples

```
# Calculate running variance of each minutely OHLC bar of data
x_ts <- ohlc_variance(HighFreq::SPY)
# Remove overnight variance spikes at "09:31"
in_dex <- format(index(x_ts), "%H:%M")
x_ts <- x_ts[!in_dex=="09:31", ]
# Calculate daily seasonality of variance
var_seasonal <- season_ality(x_ts=x_ts)
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=co_eff,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
co_eff <- matrix(c(0.2, 0.2))
# Calculate matrix of innovations
in_nov <- matrix(rnorm(1e4, sd=0.01))
# Calculate recursive filter using filter()
filter_ed <- filter(in_nov, filter=co_eff, method="recursive")
# Calculate recursive filter using RcppArmadillo
re_turns <- HighFreq::sim_ar(co_eff, in_nov)
# Compare the two methods
all.equal(as.numeric(re_turns), as.numeric(filter_ed))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::sim_ar(co_eff, in_nov),
  Rcode=filter(in_nov, filter=co_eff, method="recursive"),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

sim_df *Simulate a* Dickey-Fuller *process using* Rcpp.

Description

Simulate a *Dickey-Fuller* process using *Rcpp*.

Usage

```
sim_df(eq_price, volat, theta, coeff, innov)
```

Arguments

`volat`	The volatility of returns.
`eq_price`	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:

$$r_i = \theta \, (\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \sigma \, \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where r_i and p_i are the simulated returns and prices, θ, μ, and σ are the *Ornstein-Uhlenbeck* parameters, φ_i are the *autoregressive* coefficients, and ξ_i are the standard normal *innovations*. The recursion starts with: $p_1 = r_1 = \sigma \, \xi_1$.

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 returns.

Value

A single-column *matrix* of simulated returns, with the same number of rows as the argument `innov`.

Examples

```
## Not run:
# Define the Ornstein-Uhlenbeck model parameters
eq_price <- 1.0
sig_ma <- 0.01
the_ta <- 0.01
# Define AR coefficients
co_eff <- matrix(c(0.2, 0.2))
# Calculate matrix of standard normal innovations
in_nov <- matrix(rnorm(1e3))
```

```
# Simulate Dickey-Fuller process using Rcpp
re_turns <- HighFreq::sim_df(eq_price=eq_price, volat=sig_ma, theta=the_ta, co_eff, innov=in_nov)
plot(cumsum(re_turns), t="l", main="Simulated Dickey-Fuller Prices")

## End(Not run)
```

---

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(omega, alpha, beta, innov, is_random = TRUE)
```

Arguments

omega	Parameter proportional to the long-term average level of variance.
alpha	The weight associated with recent realized variance updates.
beta	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:

$$r_i = \sigma_{i-1} \xi_i$$

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

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 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
al_pha <- 0.79
be_ta <- 0.2
om_ega <- 1e-4*(1-al_pha-be_ta)
# Calculate matrix of standard normal innovations
in_nov <- matrix(rnorm(1e3))
# Simulate the GARCH process using Rcpp
garch_data <- HighFreq::sim_garch(omega=om_ega, alpha=al_pha,  beta=be_ta, innov=in_nov)
# 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
re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
# Estimate the GARCH volatility of VTI returns
garch_data <- HighFreq::sim_garch(omega=om_ega, alpha=al_pha,  beta=be_ta,
  innov=re_turns, is_random=FALSE)
# Plot dygraph of the estimated GARCH volatility
dygraphs::dygraph(xts::xts(sqrt(garch_data[, 2]), index(re_turns)),
  main="Estimated GARCH Volatility of VTI")

## End(Not run)
```

`sim_ou` *Simulate an* Ornstein-Uhlenbeck *process using* Rcpp.

Description

Simulate an *Ornstein-Uhlenbeck* process using *Rcpp*.

Usage

```
sim_ou(init_price, eq_price, volat, theta, innov)
```

Arguments

`volat`	The volatility of returns.
`init_price`	The initial price.
`eq_price`	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:

$$r_i = p_i - p_{i-1} = \theta \, (\mu - p_{i-1}) + \sigma \, \xi_i$$

$$p_i = p_{i-1} + r_i$$

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 normal *innovations*. The recursion starts with the initial price: $p_1 = init\_price$.

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 innovation proportional to the volatility σ. 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 returns.

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
eq_price <- 1.0
sig_ma <- 0.01
the_ta <- 0.01
in_nov <- matrix(rnorm(1e3))
# Simulate Ornstein-Uhlenbeck process using Rcpp
price_s <- HighFreq::sim_ou(init_price=0, eq_price=eq_price, volat=sig_ma, theta=the_ta, innov=in_nov)
plot(price_s, t="l", main="Simulated Ornstein-Uhlenbeck Prices", ylab="prices")

## End(Not run)
```

`sim_schwartz` *Simulate a* Schwartz *process using* Rcpp.

Description

Simulate a *Schwartz* process using *Rcpp*.

Usage

```
sim_schwartz(eq_price, volat, theta, innov)
```

Arguments

`volat`	The volatility of returns.
`eq_price`	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 innovation proportional to the volatility σ.

The function sim_schwartz() returns a single-column *matrix* representing the *time series* of simulated returns.

Value

A single-column *matrix* of simulated returns, with the same number of rows as the argument innov.

Examples

```
## Not run:
# Define the Schwartz model parameters
eq_price <- 2.0
sig_ma <- 0.01
the_ta <- 0.01
in_nov <- matrix(rnorm(1e3))
# Simulate Schwartz process using Rcpp
re_turns <- HighFreq::sim_schwartz(eq_price=eq_price, volat=sig_ma, theta=the_ta, innov=in_nov)
plot(exp(cumsum(re_turns)), t="l", main="Simulated Schwartz Prices", ylab="prices")

## End(Not run)
```

---

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(x_ts, look_back = 51, vol_mult = 2)
```

Arguments

x_ts	A single-column *xts* time series, or a *numeric* or *Boolean* vector.
look_back	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
ta_q <- HighFreq::SPY_TAQ
# scrub quotes with suspect bid-offer spreads
bid_offer <- ta_q[, "Ask.Price"] - ta_q[, "Bid.Price"]
sus_pect <- which_extreme(bid_offer, look_back=51, vol_mult=3)
# Remove suspect values
ta_q <- ta_q[!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(x_ts, look_back = 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
ta_q <- SPY_TAQ
# Calculate mid prices
mid_prices <- 0.5 * (ta_q[, "Bid.Price"] + ta_q[, "Ask.Price"])
# Replace whole rows containing suspect price jumps with NA, and perform locf()
ta_q[which_jumps(mid_prices, look_back=31, vol_mult=1.0), ] <- NA
ta_q <- xts:::na.locf.xts(ta_q)
```