Package ‘HighFreq’

August 23, 2020

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 6.1.1

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(t_series)
```

Arguments

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

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 `t_series`).

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 = "run_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="run_variance")
# Calculate time series of daily skew estimates for SPY
skew_daily <- apply.daily(x=HighFreq::SPY, FUN=agg_stats_r, calc_bars="run_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(ex_cess, re_turns, start_points, end_points,
  typ_e = "max_sharpe", to_l = 0.001, max_eigen = 0L, pro_b = 0.1,
  al_pha = 0, scal_e = TRUE, vo_l = 0.01, co_eff = 1,
  bid_offer = 0)
```

Arguments

ex_cess	A *time series* or a *matrix* of excess returns data (the returns in excess of the risk-free rate).
re_turns	A *time series* or a *matrix* of returns data (the returns in excess of the risk-free rate).
start_points	An *integer vector* of start points.
end_points	An *integer vector* of end points.
typ_e	A *string* specifying the objective for calculating the weights (see Details).
to_l	A *numeric* tolerance level for discarding small eigenvalues in order to regularize the matrix inverse. (The default is 0.001)
max_eigen	An *integer* equal to the number of eigenvectors used for calculating the regularized inverse of the covariance *matrix* (the default is the number of columns of re_turns).
al_pha	The shrinkage intensity between 0 and 1. (the default is 0).
scal_e	A *Boolean* specifying whether the weights should be scaled (the default is scal_e = TRUE).
vo_l	A *numeric* volatility target for scaling the weights. (The default is 0.001)
co_eff	A *numeric* multiplier of the weights. (The default is 1)
bid_offer	A *numeric* bid-offer spread. (The default is 0)

Details

The function back_test() performs a backtest simulation of a rolling portfolio optimization strategy over a *vector* of end_points.

It performs a loop over the end_points, and subsets the *matrix* of excess returns ex_cess along its rows, between the corresponding end point and the start point. It passes the subset matrix of excess returns into the function calc_weights(), which calculates the optimal portfolio weights. The arguments max_eigen, al_pha, typ_e, and scal_e are also passed to the function calc_weights().

The function back_test() multiplies the weights by the coefficient co_eff (with default equal to 1), which allows reverting a strategy if co_eff = -1.

The function back_test() then multiplies the weights times the future portfolio returns, to calculate the out-of-sample strategy returns.

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. It then 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 re_turns.

Value

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

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_points <- rutils::calc_endpoints(re_turns, inter_val="months")
end_points <- end_points[end_points>50]
len_gth <- NROW(end_points)
# Define 12-month look_back interval and start_points over sliding window
look_back <- 12
start_points <- c(rep_len(1, look_back-1), end_points[1:(len_gth-look_back+1)])
# Define shrinkage and regularization intensities
al_pha <- 0.5
max_eigen <- 3
# Simulate a monthly rolling portfolio optimization strategy
pnl_s <- HighFreq::back_test(ex_cess, re_turns,
                            start_points-1, end_points-1,
                            max_eigen = max_eigen,
                            al_pha = 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_eigen   *Calculate the eigen decomposition of the covariance* matrix *of returns using* RcppArmadillo.

Description

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

Usage

```
calc_eigen(re_turns)
```

Arguments

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

Details

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

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 returns
re_turns <- matrix(rnorm(5e6), nc=5)
# Calculate eigen decomposition
ei_gen <- HighFreq::calc_eigen(scale(re_turns, scale=FALSE))
# Calculate PCA
pc_a <- prcomp(re_turns)
# 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(re_turns),
  Rcode=prcomp(re_turns),
  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(len_gth, ste_p = 1L, front = TRUE)
```

Arguments

len_gth	An *integer* equal to the length of the vector to be divide into equal intervals.
ste_p	The number of elements in each interval between neighboring end points.
front	*Boolean* argument: if TRUE then add a stub interval at the beginning, else add a stub interval at the end. (default is TRUE)

Details

The end points are a vector of unsigned integers which divide a vector of length equal to len_gth into equally spaced intervals. If a whole number of intervals doesn't fit over the vector, then calc_endpoints() adds a stub interval either at the beginning (the default) or at the end. The end points are shifted by -1 because indexing starts at 0 in C++ code.

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

The end points produced by calc_endpoints() don't include the first placeholder end point, which is usually equal to zero. For example, consider the end points for a vector of length 20 divided into intervals of length 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 must be equal to 0. The first end point is a placeholder and doesn't correspond to any vector element.

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 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 index values representing the end points (a vector of unsigned integers).

Examples

```
# Calculate end points without a stub interval
HighFreq::calc_endpoints(25, 5)
# Calculate end points with initial stub interval
HighFreq::calc_endpoints(23, 5)
# Calculate end points with a stub interval at the end
HighFreq::calc_endpoints(23, 5, FALSE)
```

---

calc_inv *Calculate the regularized inverse of the covariance* matrix *of returns using* RcppArmadillo.

Description

Calculate the regularized inverse of the covariance *matrix* of returns using RcppArmadillo.

Usage

```
calc_inv(re_turns, to_l = 0.001, max_eigen = 0L)
```

Arguments

re_turns	A *time series* or *matrix* of returns data.
to_l	A *numeric* tolerance level for discarding small eigenvalues in order to regularize the matrix inverse. (The default is `0.001`)
max_eigen	An *integer* equal to the regularization intensity (the number of eigenvalues and eigenvectors used for calculating the regularized inverse).

Details

The function calc_inv() calculates the regularized inverse of the *covariance matrix*, by discarding eigenvectors with small eigenvalues less than the tolerance level `to_l`. The function `calc_inv()` first calculates the covariance *matrix* of the `re_turns`, and then it calculates its regularized inverse. If `max_eigen` is not specified then it calculates the regularized inverse using the function `arma::pinv()`. If `max_eigen` is specified then it calculates the regularized inverse using eigen decomposition with only the largest `max_eigen` eigenvalues and their corresponding eigenvectors.

Value

A *matrix* equal to the regularized inverse.

Examples

```
## Not run:
# Create random matrix
re_turns <- matrix(rnorm(500), nc=5)
max_eigen <- 3
# Calculate regularized inverse using RcppArmadillo
in_verse <- HighFreq::calc_inv(re_turns, max_eigen)
# Calculate regularized inverse from eigen decomposition in R
ei_gen <- eigen(cov(re_turns))
inverse_r <- ei_gen$vectors[, 1:max_eigen] %*% (t(ei_gen$vectors[, 1:max_eigen]) / ei_gen$values[1:max_eigen]
# Compare RcppArmadillo with R
all.equal(in_verse, inverse_r)

## End(Not run)
```

`calc_lm` *Perform multivariate linear regression using* Rcpp.

Description

Perform multivariate linear regression using *Rcpp*.

Usage

```
calc_lm(res_ponse, de_sign)
```

Arguments

res_ponse	A *vector* of response data.
de_sign	A *matrix* of design (predictor i.e. explanatory) data.

Details

The function `calc_lm()` performs the same calculations as the function `lm()` from package *stats*. It uses `RcppArmadillo` C++ code and is about *10* 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"* containes the alpha and beta coefficients, and their *t-values* and *p-values*.

Examples

```
## Not run:
# Define design matrix with explanatory variables
len_gth <- 100; n_var <- 5
de_sign <- matrix(rnorm(n_var*len_gth), nc=n_var)
# Response equals linear form plus error terms
weight_s <- rnorm(n_var)
res_ponse <- -3 + de_sign %*% weight_s + rnorm(len_gth, sd=0.5)
# Perform multivariate regression using lm()
reg_model <- lm(res_ponse ~ de_sign)
sum_mary <- summary(reg_model)
# Perform multivariate regression using calc_lm()
reg_model_arma <- calc_lm(res_ponse=res_ponse, de_sign=de_sign)
reg_model_arma$coefficients
# Compare the outputs of both functions
all.equal(reg_model_arma$coefficients[, "coeff"], unname(coef(reg_model)))
all.equal(unname(reg_model_arma$coefficients), unname(sum_mary$coefficients))
all.equal(drop(reg_model_arma$residuals), unname(reg_model$residuals))
all.equal(unname(reg_model_arma$stats), c(sum_mary$r.squared, unname(sum_mary$fstatistic[1])))

## End(Not run)
```

---

calc_mad — *Calculate the Median Absolute Deviations* MAD *of the columns of a* time series *or a* matrix *using* `RcppArmadillo`.

Description

Calculate the Median Absolute Deviations *MAD* of the columns of a *time series* or a *matrix* using `RcppArmadillo`.

Usage

```
calc_mad(t_series)
```

Arguments

`t_series` A *time series* or a *matrix* of data.

Details

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

The function `calc_mad()` performs the same calculation as the function `stats::mad`, but it's much faster because it uses `RcppArmadillo` `C++` code.

Value

A row vector equal to the Median Absolute Deviations *MAD* of the columns of `t_series` matrix.

Examples

```
## Not run:
# Calculate VTI returns
re_turns <- na.omit(rutils::etf_env$re_turns[ ,"VTI", drop=FALSE])
# Compare calc_mad() with stats::mad()
all.equal(drop(HighFreq::calc_mad(re_turns)),
  mad(re_turns))
# Compare the speed of RcppArmadillo with stats::mad()
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_mad(re_turns),
  Rcode=mad(re_turns),
  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(vec_tor)
```

Arguments

`vec_tor` 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 *vector*.

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_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(mat_rix, use_median = FALSE)
```

Arguments

mat_rix	A *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 is 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 mat_rix 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*).

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 mat_rix.

Examples

```
## Not run:
# Create a matrix of random data
mat_rix <- matrix(rnorm(20000), nc=20)
scale_d <- calc_scaled(mat_rix=mat_rix, use_median=FALSE)
scale_d2 <- scale(mat_rix)
all.equal(scale_d, scale_d2, check.attributes=FALSE)
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=calc_scaled(mat_rix=mat_rix, use_median=FALSE),
  Rcode=scale(mat_rix),
  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(t_series, typ_e = "pearson", al_pha = 0.25)
```

Arguments

`t_series`	A *time series* or a *matrix* of data.
`typ_e`	A *string* specifying the objective for calculating the weights (see Details).
`al_pha`	The confidence level for calculating the quantiles. (the default is `0.25`).

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 `typ_e` == `"pearson"` (the default) then `calc_skew()` calculates the Pearson skewness using the third moment of the data.

If `typ_e` == `"quantile"` then it calculates the skewness using the differences between the quantiles of the data.

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

Value

A row vector equal to the skewness of the columns of `t_series`.

Examples

```
## Not run:
# Calculate VTI returns
re_turns <- na.omit(rutils::etf_env$re_turns[ ,"VTI", drop=FALSE])
# Calculate the Pearson skewness
HighFreq::calc_skew(re_turns)
# Compare HighFreq::calc_skew() with Pearson skewness
calc_skewr <- function(x) {
  x <- (x-mean(x)); nr <- NROW(x);
  nr*sum(x^3)/(var(x))^1.5/(nr-1)/(nr-2)
}  # end calc_skewr
all.equal(drop(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, typ_e = "quantile", al_pha = 0.1)
# Compare HighFreq::calc_skew() with quantile skewness
calc_skewq <- function(x) {
   quantile_s <- quantile(x, c(0.25, 0.5, 0.75), 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, typ_e = "quantile")),
  calc_skewq(re_turns), check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
summary(microbenchmark(
  Rcpp=HighFreq::calc_skew(re_turns, typ_e = "quantile"),
  Rcode=calc_skewq(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
# Calculate the nonparametric skewness
HighFreq::calc_skew(re_turns, typ_e = "nonparametric")
# Compare HighFreq::calc_skew() with R nonparametric skewness
all.equal(drop(HighFreq::calc_skew(re_turns, typ_e = "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, typ_e = "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 equal to the lag of a vector of end points.*

Description

Calculate a vector of start points equal to the lag of a vector of end points.

Usage

```
calc_startpoints(end_points, look_back)
```

Arguments

`end_points`	An *unsigned integer* vector of end points.
`look_back`	The length of the look-back interval, equal to the lag applied to the end points.

Details

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

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 is shifted by `-1` and starts at `0` not `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 of start points (vector of unsigned integers), associated with the vector `end_points`.

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

Description

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

Usage

```
calc_var(t_series)
```

Arguments

t_series	A *time series* or a *matrix* of data.

Details

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

The function calc_var() performs the same calculation as the function colVars() from package matrixStats, but it's much faster because it uses RcppArmadillo C++ code.

Value

A row vector equal to the variance of the columns of t_series matrix.

Examples

```
## Not run:
# Create a matrix of random returns
re_turns <- matrix(rnorm(5e6), nc=5)
# Compare calc_var() with standard var()
all.equal(drop(HighFreq::calc_var(re_turns)),
  apply(re_turns, 2, var))
# Compare calc_var() with matrixStats
all.equal(drop(HighFreq::calc_var(re_turns)),
  matrixStats::colVars(re_turns))
# 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

## End(Not run)
```

`calc_var_ohlc` *Calculate the variance of an* OHLC time series*, using different range estimators and* `RcppArmadillo`*.*

Description

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

Usage

```
calc_var_ohlc(oh_lc, calc_method = "yang_zhang", lag_close = 0L,
  scal_e = TRUE, in_dex = 0L)
```

Arguments

`oh_lc`	An *OHLC time series* or a *numeric matrix* of prices.
`calc_method`	A *character* string representing the 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.)
`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`.)
`scal_e`	*Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scal_e = 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 default `calc_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 optional argument `in_dex` is the time index of the *time series* `oh_lc`. 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 time index of SPY returns
in_dex <- c(1, diff(xts::.index(HighFreq::SPY)))
# Calculate the variance of SPY returns, with scaling of the returns
HighFreq::calc_var_ohlc(HighFreq::SPY,
 calc_method="yang_zhang", scal_e=TRUE, in_dex=in_dex)
# Calculate variance without accounting for overnight jumps
HighFreq::calc_var_ohlc(HighFreq::SPY,
 calc_method="rogers_satchell", scal_e=TRUE, in_dex=in_dex)
# Calculate the variance without scaling the returns
HighFreq::calc_var_ohlc(HighFreq::SPY, scal_e=FALSE)
# Calculate the variance by passing in the lagged close prices
lag_close <- HighFreq::lag_it(HighFreq::SPY[, 4])
all.equal(HighFreq::calc_var_ohlc(HighFreq::SPY),
  HighFreq::calc_var_ohlc(HighFreq::SPY, lag_close=lag_close))
# Compare with HighFreq::calc_var_ohlc_r()
all.equal(HighFreq::calc_var_ohlc(HighFreq::SPY, in_dex=in_dex),
  HighFreq::calc_var_ohlc_r(HighFreq::SPY))
# Compare the speed of Rcpp with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::calc_var_ohlc(HighFreq::SPY),
  Rcode=HighFreq::calc_var_ohlc_r(HighFreq::SPY),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## 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, calc_method = "yang_zhang", scal_e = TRUE)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`calc_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, calc_method="yang_zhang")
# Calculate variance without accounting for overnight jumps
HighFreq::calc_var_ohlc_r(HighFreq::SPY, calc_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(t_series)
```

Arguments

`t_series`	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 objective functions.*

Description

Calculate the optimal portfolio weights for different objective functions.

Usage

```
calc_weights(re_turns, typ_e = "max_sharpe", to_l = 0.001,
  max_eigen = 0L, pro_b = 0.1, al_pha = 0, scal_e = TRUE,
  vo_l = 0.01)
```

Arguments

re_turns	A *time series* or a *matrix* of returns data (the returns in excess of the risk-free rate).
typ_e	A *string* specifying the objective for calculating the weights (see Details).
to_l	A *numeric* tolerance level for discarding small eigenvalues in order to regularize the matrix inverse. (The default is 0.001)
max_eigen	An *integer* equal to the number of eigenvectors used for calculating the regularized inverse of the covariance *matrix* (the default is the number of columns of re_turns).
al_pha	The shrinkage intensity between 0 and 1. (the default is 0).
scal_e	A *Boolean* specifying whether the weights should be scaled (the default is scal_e = TRUE).
vo_l	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 objective functions, using RcppArmadillo C++ code.

If typ_e == "max_sharpe" (the default) 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 typ_e == "min_var" then it calculates the weights of the minimum variance portfolio under linear constraints.

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

If typ_e == "rank" then it calculates the weights as the ranks (order index) of the trailing Sharpe ratios of the portfolio assets.

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

calc_weights() applies dimensional regularization to calculate the inverse of the covariance *matrix* of returns from its eigen decomposition, using the function arma::eig_sym().

In addition, it applies shrinkage to the *vector* of mean column returns, by shrinking it to its common mean value. The shrinkage intensity al_pha determines the amount of shrinkage that is applied, with `al_pha = 0` representing no shrinkage (with the estimator of mean returns equal to the means of the columns of `re_turns`), and `al_pha = 1` representing complete shrinkage (with the estimator of mean returns equal to the single mean of all the columns of `re_turns`)

Value

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

Examples

```
## Not run:
# Calculate covariance matrix of ETF returns
re_turns <- na.omit(rutils::etf_env$re_turns[, 1:16])
ei_gen <- eigen(cov(re_turns))
# Calculate regularized inverse of covariance matrix
max_eigen <- 3
eigen_vec <- ei_gen$vectors[, 1:max_eigen]
eigen_val <- ei_gen$values[1:max_eigen]
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, max_eigen=max_eigen, al_pha=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(t_series, lagg = 1L, padd = TRUE)
```

Arguments

`t_series`	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`).
`padd`	*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 `padd = TRUE`)

Details

The function `diff_it()` calculates the differences between the rows of the input *time series* or *matrix* and its lagged version. The lagged version has its rows shifted down by the number equal to `lagg` rows.

The argument `lagg` specifies the number of lags applied to the rows of the lagged version. 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 `padd` specifies whether the output *matrix* should be padded (extended) with the rows of the initial (warmup) period at the beginning, in order to return a *matrix* with the same number of rows as the input. The default is `padd = 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 several times faster than `R` code.

Value

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

Examples

```
## Not run:
# Create a matrix of random returns
re_turns <- matrix(rnorm(5e6), nc=5)
# Compare diff_it() with rutils::diff_it()
all.equal(HighFreq::diff_it(re_turns, lagg=3, padd=TRUE),
  zoo::coredata(rutils::diff_it(re_turns, lagg=3)),
  check.attributes=FALSE)
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  Rcpp=HighFreq::diff_it(re_turns, lagg=3, padd=TRUE),
  Rcode=rutils::diff_it(re_turns, lagg=3),
  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(t_series, lagg = 1L, padd = TRUE)
```

Arguments

`t_series`	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`).
`padd`	*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 `padd = 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 `padd` specifies whether the output *vector* should be padded (extended) with zeros at the beginning, in order to return a *vector* of the same length as the input. The default is `padd = 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, padd=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, padd=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(t_series, lagg = 1L, pad_zeros = TRUE)
```

Arguments

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

Value

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

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

Arguments

`t_series`	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* `t_series` 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 `t_series`, so that it has the same number of element as `t_series`. 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 `t_series` contains returns data, then the output *matrix* should be padded with zeros, to avoid data snooping. If `t_series` 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)
```

`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(vec_tor, mat_rix, by_col = TRUE)
```

Arguments

`vec_tor`	A *vector*.
`mat_rix`	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)
```

---

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"])
```

`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 convolutions of the* matrix *columns with a* vector *of weights.*

Description

Calculate the convolutions of the *matrix* columns with a *vector* of weights.

Usage

```
roll_conv(mat_rix, weight_s)
```

Arguments

mat_rix	A *matrix* of data.
weight_s	A column *vector* of weights.

Details

The function roll_conv() calculates the convolutions of the *matrix* columns with a *vector* of weights. It performs a loop down over the *matrix* rows and multiplies the past (higher) values by the weights. It calculates the rolling weighted sum 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=mat_rix,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 mat_rix.

Examples

```
## Not run:
# First example
# Create matrix from historical prices
mat_rix <- na.omit(rutils::etf_env$re_turns[, 1:2])
# 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_conv(mat_rix, weight_s)
# Compare with original
all.equal(coredata(mat_rix), weight_ed, check.attributes=FALSE)
# Second example
# 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
weight_ed <- HighFreq::roll_conv(mat_rix, weight_s)
# Calculate rolling weighted sum using filter()
filter_ed <- filter(x=mat_rix, 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_conv_ref` *Calculate the convolutions of the* matrix *columns with a* vector *of weights.*

Description

Calculate the convolutions of the *matrix* columns with a *vector* of weights.

Usage

```
roll_conv_ref(mat_rix, weight_s)
```

Arguments

`mat_rix`	A *matrix* of data.
`weight_s`	A column *vector* of weights.

Details

The function `roll_conv_ref()` calculates the convolutions of the *matrix* columns with a *vector* of weights. It performs a loop down over the *matrix* rows and multiplies the past (higher) values by the weights. It calculates the rolling weighted sum of the past values.

The function `roll_conv_ref()` accepts a *pointer* to the argument `mat_rix`, and replaces the old *matrix* values with the weighted sums. It performs the calculation in place, without copying the *matrix* in memory (which greatly increases the computation speed).

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

Examples

```
## Not run:
# First example
# Create matrix from historical prices
mat_rix <- na.omit(rutils::etf_env$re_turns[, 1:2])
# 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_conv_ref(mat_rix, weight_s)
# Compare with original
all.equal(coredata(mat_rix), weight_ed, check.attributes=FALSE)
# Second example
# 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
weight_ed <- HighFreq::roll_conv_ref(mat_rix, weight_s)
# Calculate rolling weighted sum using filter()
filter_ed <- filter(x=mat_rix, 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(vec_tor)
```

Arguments

`vec_tor`	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 `vec_tor`.

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_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_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(t_series, end_points)
```

Arguments

t_series	A *time series* or a *matrix* with multiple columns of data.
end_points	An *integer vector* of end points.

Details

The function `roll_ohlc()` performs a loop over the *end_points*, along the rows of the `t_series` data. At each *end_point*, it selects the past rows of `t_series` data, starting at the first bar after the previous *end_point*, and then calls the function `agg_ohlc()` on the selected `t_series` data 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 *end_points* 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_points <- rutils::calc_endpoints(oh_lc, inter_val=25)
# Aggregate over end_points:
ohlc_agg <- HighFreq::roll_ohlc(t_series=oh_lc, end_points=(end_points-1))
# Compare with xts::to.period()
ohlc_agg_xts <- .Call("toPeriod", oh_lc, as.integer(end_points), TRUE, NCOL(oh_lc), FALSE, FALSE, colnames(oh_
all.equal(ohlc_agg, coredata(ohlc_agg_xts), 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(mat_rix, look_back, use_median = FALSE)
```

Arguments

mat_rix	A *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 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)
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 mat_rix argument using RcppArmadillo. The function roll_scale() performs a loop over the rows of mat_rix, 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 mat_rix.

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(mat_rix=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_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 = "run_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="run_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 weighted sum over a* time series *or a* matrix *using* Rcpp.

Description

Calculate the rolling weighted sum over a *time series* or a *matrix* using *Rcpp*.

Usage

```
roll_sum(t_series, look_back = 1L, stu_b = NULL, end_points = NULL,
  weight_s = NULL)
```

Arguments

`t_series`	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`).
`stu_b`	An *integer* value equal to the first stub interval for calculating the end points.
`end_points`	An *unsigned integer* vector of end points.
`weight_s`	A column *vector* of weights.

Details

The function roll_sum() calculates the rolling sums over the columns of the t_series data. The sums are calculated over a number of data points equal to look_back.

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

The arguments stu_b, end_points, and weight_s are optional.

If either the arguments stu_b or end_points are supplied, then the rolling sums are calculated at the end points.

If only the argument stu_b is supplied, then the end points are calculated from the stu_b and look_back arguments. The first end point is equal to stu_b and the end points are spaced look_back periods apart.

If the argument weight_s is supplied, then weighted sums are calculated. Then the function roll_sum() calculates the rolling weighted sums of the past values.

The function roll_sum() calculates the rolling weighted sums as convolutions of the t_series columns with the *vector* of weights using the RcppArmadillo function arma::conv2(). It performs a similar calculation to the standard R function stats::filter(x=t_series,filter=weight_s,method="convolutio but it's over 6 times faster, and it doesn't produce any leading NA values. using fast *RcppArmadillo* C++ code. 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 t_series.

Examples

```
## Not run:
# First example
# Create series of historical returns
re_turns <- na.omit(rutils::etf_env$re_turns[, c("VTI", "IEF")])
# Define parameters
look_back <- 22
stu_b <- 21
# Calculate rolling sums at each point
c_sum <- HighFreq::roll_sum(re_turns, look_back=look_back)
r_sum <- rutils::roll_sum(re_turns, look_back=look_back)
all.equal(c_sum, coredata(r_sum), check.attributes=FALSE)
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
c_sum <- HighFreq::roll_sum(re_turns, look_back=look_back, stu_b=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_points
c_sum <- HighFreq::roll_sum(re_turns, end_points=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_sum(re_turns, weight_s=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_sum(re_turns, end_points=end_p, weight_s=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_sum(re_turns, weight_s)
# Compare with original
all.equal(coredata(re_turns), weight_ed, check.attributes=FALSE)

## End(Not run)
```

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

Usage

```
roll_var(t_series, ste_p = 1L, look_back = 1L)
```

Arguments

`t_series`	A *time series* or a *matrix*.
`ste_p`	The number of time periods between the end points.
`look_back`	The number of end points in the look-back interval.

Details

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

The end points are calculated along the rows of `t_series` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `ste_p`.

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.

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

The function `roll_var()` with the parameter `ste_p = 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* with the same number of columns as the input time series `t_series`, 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, ste_p=25, look_back=3)
# Compare the variance estimates over 11-period lookback 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 RcppArmadillo with RcppRoll
library(microbenchmark)
summary(microbenchmark(
  RcppArmadillo=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

## 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(oh_lc, ste_p = 1L, look_back = 1L,
  calc_method = "yang_zhang", scal_e = TRUE, in_dex = 0L)
```

Arguments

`oh_lc`	A *time series* or a *matrix* with *OHLC* price data.
`ste_p`	The number of time periods between the end points.
`look_back`	The number of end points in the look-back interval.

`calc_method` A *character* string representing the 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.)

`scal_e` *Boolean* argument: Should the returns be divided by the time index, the number of seconds in each period? (The default is `scal_e = 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* `oh_lc`.

The end points are calculated along the rows of `oh_lc` using the function `calc_endpoints()`, with the number of time periods between the end points equal to `ste_p`.

The function `roll_var_ohlc()` performs a loop over the end points, subsets the previous (past) rows of `oh_lc`, 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.

For example, the rolling variance at daily end points with an `11` day look-back, can be calculated using the parameters `ste_p = 1` and `look_back = 11` (Assuming the `oh_lc` 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 `ste_p = 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 `calc_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 optional argument `in_dex` is the time index of the *time series* `oh_lc`. 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 time index of SPY returns
oh_lc <- HighFreq::SPY
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,
                              ste_p=1, look_back=21,
                              calc_method="yang_zhang",
                              in_dex=in_dex, scal_e=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,
                              ste_p=5, look_back=4,
                              calc_method="yang_zhang",
                              in_dex=in_dex, scal_e=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, scal_e=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* vector*, using* `RcppArmadillo`.

Description

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

Usage

```
roll_var_vec(t_series, look_back = 11L)
```

Arguments

`t_series`	A single-column *time series* or a *vector*.
`look_back`	The length of the look-back interval, equal to the number of *vector* elements used for calculating a single variance estimate.

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 *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 `t_series`.

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 `t_series`.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare the variance estimates over 11-period lookback 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(
  RcppArmadillo=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 sum over a single-column* time series *or a* vector *using* Rcpp.

Description

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

Usage

```
roll_vec(t_series, look_back)
```

Arguments

`t_series`	A single-column *time series* or a *vector*.
`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 *vector* of rolling sums, over a *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 `t_series`.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Calculate rolling sums over 11-period lookback 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))
# 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 sum over a single-column* time series *or a* vector *using* RcppArmadillo.

Description

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

Usage

```
roll_vecw(t_series, weight_s)
```

Arguments

`t_series`	A single-column *time series* or a *vector*.
`weight_s`	A *vector* of weights.

Details

The function `roll_vecw()` calculates the rolling weighted sum of a *vector* over its past values (a convolution with the *vector* of weights), using `RcppArmadillo`. It performs a similar calculation as the standard `R` function `stats::filter(x=t_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 `t_series`.

Examples

```
## Not run:
# First example
# Create vector from historical prices
re_turns <- as.numeric(rutils::etf_env$VTI[, 6])
# Create simple weights
weight_s <- c(1, rep(0, 10))
# Calculate rolling weighted sum
weight_ed <- HighFreq::roll_vecw(t_series=re_turns, weight_s=weight_s)
# Compare with original
all.equal(re_turns, as.numeric(weight_ed))
# Second example
# Create exponentially decaying weights
weight_s <- exp(-0.2*1:11)
weight_s <- weight_s/sum(weight_s)
# Calculate rolling weighted sum
weight_ed <- HighFreq::roll_vecw(t_series=re_turns, weight_s=weight_s)
# Calculate rolling weighted sum 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)

## 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 VWAP, but using vectorized functions, so it's a little faster.

Usage

```
roll_vwap(oh_lc, x_ts = oh_lc[, 4], look_back)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`x_ts`	A single-column *xts* time series.
`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 `x_ts` 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 <- run_returns(x_ts=HighFreq::SPY)
# Calculate the rolling volume-weighted average returns
roll_vwap(oh_lc=HighFreq::SPY, x_ts=returns_running, look_back=11)
```

`roll_zscores` *Perform rolling regressions over the rows of the design matrix, and calculate a* vector *of z-scores of the residuals.*

Description

Perform rolling regressions over the rows of the design matrix, and calculate a *vector* of z-scores of the residuals.

Usage

```
roll_zscores(res_ponse, de_sign, look_back)
```

Arguments

`res_ponse`	A *vector* of response data.
`de_sign`	A *matrix* of design (predictor i.e. explanatory) data.
`look_back`	The length of the look-back interval, equal to the number of elements of data used for calculating the regressions.

Details

The function `roll_zscores()` performs rolling regressions along the rows of the design *matrix* `de_sign`, using the function `calc_lm()`.

The function `roll_zscores()` performs a loop over the rows of `de_sign`, and it subsets `de_sign` and `res_ponse` over a number of previous (past) rows equal to `look_back`. It performs a regression on the subset data, and calculates the *z-score* of the last residual value for each regression. It returns a numeric *vector* of the *z-scores*.

Value

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

Examples

```
## Not run:
# Calculate Z-scores from rolling time series regression using RcppArmadillo
look_back <- 11
clo_se <- as.numeric(Cl(rutils::etf_env$VTI))
date_s <- xts::.index(rutils::etf_env$VTI)
z_scores <- HighFreq::roll_zscores(res_ponse=clo_se,
 de_sign=matrix(as.numeric(date_s), nc=1),
 look_back=look_back)
# Define design matrix with explanatory variables
len_gth <- 100; n_var <- 5
de_sign <- matrix(rnorm(n_var*len_gth), nc=n_var)
# response equals linear form plus error terms
weight_s <- rnorm(n_var)
res_ponse <- -3 + de_sign %*% weight_s + rnorm(len_gth, sd=0.5)
# Calculate Z-scores from rolling multivariate regression using RcppArmadillo
look_back <- 11
z_scores <- HighFreq::roll_zscores(res_ponse=res_ponse, de_sign=de_sign, look_back=look_back)
# Calculate z-scores in R from rolling multivariate regression using lm()
z_scores_r <- sapply(1:NROW(de_sign), 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_design <- de_sign[start_point:ro_w, ]
  reg_model <- lm(sub_response ~ sub_design)
  resid_uals <- reg_model$residuals
  resid_uals[NROW(resid_uals)]/sd(resid_uals)
})  # end sapply
# Compare the outputs of both functions
all.equal(unname(z_scores[-(1:look_back)]),
  unname(z_scores_r[-(1:look_back)]))

## End(Not run)
```

`run_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

```
run_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 run_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 run_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::run_returns(x_ts=HighFreq::SPY_TAQ)
# Calculate close to close returns
re_turns <- HighFreq::run_returns(x_ts=HighFreq::SPY)
# Calculate open to open returns
re_turns <- HighFreq::run_returns(x_ts=HighFreq::SPY, col_umn=1)
```

---

run_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

```
run_sharpe(oh_lc, calc_method = "close")
```

Arguments

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

calc_method A *character* string representing method for estimating the Sharpe-like exponent.

Details

The function `run_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 <- run_sharpe(HighFreq::SPY)
```

---

run_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

```
run_skew(oh_lc, calc_method = "rogers_satchell")
```

Arguments

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

Details

The function `run_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::run_skew(HighFreq::SPY)
```

`run_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

```
run_variance(oh_lc, calc_method = "yang_zhang", scal_e = TRUE)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`calc_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 `run_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 `run_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::run_variance(oh_lc)
# Calculate variance estimates for SPY
var_running <- HighFreq::run_variance(HighFreq::SPY, calc_method="yang_zhang")
# Calculate SPY variance without overnight jumps
var_running <- HighFreq::run_variance(HighFreq::SPY, calc_method="rogers_satchell")
```

---

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 `run_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 `run_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 <- run_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_arima` *Recursively filter a* vector *of innovations through a* vector *of* ARIMA *coefficients.*

Description

Recursively filter a *vector* of innovations through a *vector* of *ARIMA* coefficients.

Usage

```
sim_arima(in_nov, co_eff)
```

Arguments

`in_nov`	A *vector* of innovations (random numbers).
`co_eff`	A *vector* of *ARIMA* coefficients.

Details

The function `sim_arima()` recursively filters a *vector* of innovations through a *vector* of *ARIMA* coefficients, using `RcppArmadillo` `C++` code. It performs the same calculation as the standard `R` function `filter(x=in_nov,filter=co_eff,method="recursive")`, but it's over `6` times faster.

Value

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

Examples

```
## Not run:
# Create vector of innovations
in_nov <- rnorm(100)
# Create ARIMA coefficients
co_eff <- c(-0.8, 0.2)
# Calculate recursive filter using filter()
filter_ed <- filter(in_nov, filter=co_eff, method="recursive")
# Calculate recursive filter using RcppArmadillo
ari_ma <- HighFreq::sim_arima(in_nov, rev(co_eff))
# Compare the two methods
all.equal(as.numeric(ari_ma), as.numeric(filter_ed))

## End(Not run)
```

sim_garch *Simulate a* GARCH *process using* Rcpp.

Description

Simulate a *GARCH* process using *Rcpp*.

Usage

```
sim_garch(om_ega, al_pha, be_ta, in_nov)
```

Arguments

om_ega	Parameter proportional to the long-term average level of variance.
al_pha	The weight associated with recent realized variance updates.
be_ta	The weight associated with the past variance estimates.
in_nov	A *vector* of innovations (random numbers).

Details

The function `sim_garch()` simulates a *GARCH* process using fast *Rcpp* `C++` code.

Value

A *matrix* with two columns: the simulated returns and variance, and with the same number of rows as the length of the argument `in_nov`.

Examples

```
## Not run:
# Define the GARCH model parameters
om_ega <- 0.01
al_pha <- 0.5
be_ta <- 0.2
# Simulate the GARCH process using Rcpp
garch_rcpp <- sim_garch(om_ega=om_ega, al_pha=al_pha, be_ta=be_ta, in_nov=rnorm(10000))

## End(Not run)
```

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

Description

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

Usage

```
sim_ou(eq_price, vol_at, the_ta, in_nov)
```

Arguments

`eq_price`	The equilibrium price.
`vol_at`	The volatility of returns.
`the_ta`	The strength of mean reversion.
`in_nov`	A *vector* of innovations (random numbers).

Details

The function `sim_ou()` simulates an *Ornstein-Uhlenbeck* process using fast *Rcpp* `C++` code. It returns a column *vector* representing the *time series* of prices.

Value

A column *vector* representing the *time series* of prices, with the same length as the argument `in_nov`.

Examples

```
## Not run:
# Define the Ornstein-Uhlenbeck model parameters
eq_price <- 5.0
vol_at <- 0.01
the_ta <- 0.01
# Simulate Ornstein-Uhlenbeck process using Rcpp
price_s <- HighFreq::sim_ou_rcpp(eq_price=eq_price, vol_at=vol_at, the_ta=the_ta, in_nov=rnorm(1000))

## 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)
```

Index