Package ‘HighFreq’

November 26, 2016

Type Package

Title High Frequency Time Series Management

Version 0.1

Date 2015-05-28

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 GPL (>= 2)

Depends rutils,
quantmod,

Imports TTR,
RcppRoll,
caTools,
lubridate,

Remotes github::algoquant/rutils,

VignetteBuilder knitr

LazyData true

Repository GitHub

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

RoxygenNote 5.0.1

R topics documented:

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

Description

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

Usage

```
agg_regate(oh_lc, mo_ment = "run_variance", weight_ed = TRUE, ...)
```

Arguments

`oh_lc`	*OHLC* time series of prices and trading volumes, in *xts* format.
`mo_ment`	*character* string representing function for estimating the moment.
`weight_ed`	*Boolean* argument: should estimate be weighted by the trading volume? (default is `TRUE`)
`...`	additional parameters to the mo_ment function.

Details

The function `agg_regate()` calculates a single number representing the volume weighted average of an estimator over the *OHLC* time series of prices. By default the sum is trade volume weighted.

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_regate(oh_lc=SPY, mo_ment="run_variance")
# calculate time series of daily skew estimates for SPY
skew_daily <- apply.daily(x=SPY, FUN=agg_regate, mo_ment="run_skew")
```

---

calc_variance — *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_variance(oh_lc, calc_method = "yang_zhang", sca_le = TRUE)
```

Arguments

`oh_lc`	an *OHLC* time series of prices in *xts* format.
`calc_method`	*character* string representing 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"`)
`sca_le`	*Boolean* argument: should the returns be divided by the number of seconds in each period? (default is `TRUE`)

Details

The function `calc_variance()` calculates the variance estimate from *OHLC* prices, using several different variance estimation methods based on the range of *OHLC* prices.

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.

If `sca_le` 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 variance from minutely bar data, because dividing returns by the number of seconds decreases the effect of overnight price jumps.

If `sca_le` 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 function calc_variance() performs the same calculations as the function run_variance() and then calculates the average of the spot variance estimates.

Value

A *numeric* value equal to the variance.

Examples

```
# create minutely OHLC time series of random prices
oh_lc <- HighFreq::random_ohlc()
# calculate variance of oh_lc
vari_ance <- HighFreq::calc_variance(oh_lc)
# calculate variance of SPY
vari_ance <- HighFreq::calc_variance(SPY, calc_method="yang_zhang")
# calculate variance of SPY without accounting for overnight jumps
vari_ance <- HighFreq::calc_variance(SPY, calc_method="rogers_satchell")
```

---

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"])
```

---

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`	*OHLC* time series of prices and trading volumes, in *xts* format (default is *NULL*).
`re_duce`	*Boolean* argument: should `oh_lc` time series be transformed to reduced form? (default is `TRUE`)
`vol_at`	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`	drift per period of the `in_dex` time index (default is 0.0).
`in_dex`	time index for the *OHLC* time series.

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=SPY["2012-02-13/2012-02-15"])
```

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

Description

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

Usage

```
random_taq(vol_at = 6.5e-05, dri_ft = 0, in_dex = seq(from =
  as.POSIXct(paste(Sys.Date() - 3, "09:30:00")), to =
  as.POSIXct(paste(Sys.Date() - 1, "16:00:00")), by = "1 sec"),
  bid_offer = 0.001, ...)
```

Arguments

`vol_at`	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`	drift per period of the `in_dex` time index (default is 0.0).
`in_dex`	time index for the *TAQ* time series.
`bid_offer`	the bid-offer spread expressed as a fraction of the prices (default is 0.001=10bps).

Details

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

Value

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

Examples

```
# create secondly TAQ time series of random prices
ta_q <- HighFreq::random_taq()
# create random TAQ time series from SPY index
ta_q <- HighFreq::random_taq(in_dex=index(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`	*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 bars 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(SPY)
```

---

roll_apply *Apply an aggregation function over a rolling lookback window and the end points of an* OHLC *time series.*

Description

Apply an aggregation function over a rolling lookback window and the end points of an *OHLC* time series.

Usage

```
roll_apply(x_ts, agg_fun = "run_variance", win_dow = 11,
  end_points = (0:NROW(x_ts)), by_columns = FALSE, ...)
```

Arguments

`x_ts`	*OHLC* time series of prices and trading volumes, in *xts* format.
`agg_fun`	*character* string representing an aggregation function to be applied over a rolling lookback window.
`win_dow`	the size of the lookback window, equal to the number of bars of data used for applying the aggregation function (including the current bar).
`end_points`	an integer vector of end points.
`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`)
`...`	additional parameters to the agg_fun function.

Details

The function `roll_apply()` applies an aggregation function over a rolling lookback window and the end points of an *OHLC* time series.

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 the function `roll_apply()` is faster because it performs less type-checking and 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 windows at each point in time. If the argument `end_points` is explicitly passed to `roll_apply()`, then `roll_apply()` performs aggregations over windows spanned by the end_points. If win_dow=2 then the aggregations are performed over non-overlapping windows, otherwise they are performed over overlapping windows.

The aggregation function `agg_fun()` can return either a single value or a vector of values. If the aggregation function `agg_fun()` returns a single value, then `roll_apply()` returns an *xts* time series with a single column. If the aggregation function `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

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

Examples

```
# extract a single day of SPY data
oh_lc <- SPY["2012-02-13"]
win_dow <- 11
# calculate the rolling sums of oh_lc columns over a rolling window
agg_regations <- roll_apply(oh_lc, agg_fun=sum, win_dow=win_dow, by_columns=TRUE)
# apply a vector-valued aggregation function over a rolling window
```

```
agg_function <- function(oh_lc)  c(max(oh_lc[, 2]), min(oh_lc[, 3]))
agg_regations <- roll_apply(oh_lc, agg_fun=agg_function, win_dow=win_dow)
# define end points at 11-minute intervals (SPY is minutely bars)
end_points <- rutils::end_points(oh_lc, inter_val=win_dow)
# calculate the sums of oh_lc columns over end_points using non-overlapping windows
agg_regations <- roll_apply(oh_lc, agg_fun=sum, win_dow=2,
                            end_points=end_points, by_columns=TRUE)
# apply a vector-valued aggregation function over the end_points of oh_lc
# using overlapping windows
agg_regations <- roll_apply(oh_lc, agg_fun=agg_function,
                            win_dow=5, end_points=end_points)
```

---

roll_hurst — *Calculate a time series of* Hurst *exponents over a rolling lookback window.*

Description

Calculate a time series of *Hurst* exponents over a rolling lookback window.

Usage

```
roll_hurst(oh_lc, win_dow = 11)
```

Arguments

`oh_lc`	an *OHLC* time series of prices in *xts* format.
`win_dow`	the size of the lookback window, equal to the number of bars 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 lookback window.

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 window 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 window, instead of multiple windows 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=SPY["2009-03"], win_dow=11)
chart_Series(hurst_rolling["2009-03-10/2009-03-12"], name="SPY hurst_rolling")
```

roll_moment — *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_moment(oh_lc, mo_ment = "run_variance", win_dow = 11,
  weight_ed = TRUE, ...)
```

Arguments

`oh_lc`	*OHLC* time series of prices and trading volumes, in *xts* format.
`mo_ment`	*character* string representing a function for estimating statistics of a single bar of *OHLC* data, such as volatility, skew, and higher moments.
`win_dow`	the size of the lookback window, equal to the number of bars of data used for calculating the rolling mean.
`weight_ed`	*Boolean* argument: should statistic be weighted by trade volume? (default `TRUE`)
`...`	additional parameters to the mo_ment function.

Details

The function `roll_moment()` 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 bar 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_moment(oh_lc=SPY, win_dow=21)
skew_rolling <- roll_moment(oh_lc=SPY, mo_ment="run_skew", win_dow=21)
skew_rolling <- skew_rolling/(var_rolling)^(1.5)
skew_rolling[1, ] <- 0
skew_rolling <- rutils::na_locf(skew_rolling)
```

`roll_sharpe` *Calculate a time series of Sharpe ratios over a rolling lookback window for an* OHLC *time series.*

Description

Calculate a time series of Sharpe ratios over a rolling lookback window for an *OHLC* time series.

Usage

```
roll_sharpe(oh_lc, win_dow = 11)
```

Arguments

`oh_lc`	an *OHLC* time series of prices in *xts* format.
`win_dow`	the size of the lookback window, equal to the number of bars 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 lookback window, 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=SPY, win_dow=11)
```

`roll_variance` *Calculate a time series of variance estimates over a rolling lookback window for an* OHLC *time series of prices, using different range estimators for variance.*

Description

Calculate a time series of variance estimates over a rolling lookback window for an *OHLC* time series of prices, using different range estimators for variance.

Usage

```
roll_variance(oh_lc, win_dow = 11, calc_method = "yang_zhang",
  sca_le = TRUE)
```

Arguments

`oh_lc`	an *OHLC* time series of prices in *xts* format.
`win_dow`	the size of the lookback window, equal to the number of bars of data used for calculating the variance.
`calc_method`	*character* string representing 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"`)
`sca_le`	*Boolean* argument: should the returns be divided by the number of seconds in each period? (default is `TRUE`)

Details

The function `roll_variance()` calculates a time series of variance estimates of percentage returns, from *OHLC* prices, using several different variance estimation methods based on the range of *OHLC* prices.

If `sca_le` 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 `sca_le` 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 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 function `roll_variance()` performs the same calculations as the function `volatility()` from package TTR, but it's a little faster because it uses function RcppRoll::roll_sd(), and 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 over a 21 period window
var_rolling <- HighFreq::roll_variance(oh_lc, win_dow=21)
```

```
# calculate variance estimates for SPY
var_rolling <- HighFreq::roll_variance(SPY, calc_method="yang_zhang")
# calculate SPY variance without accounting for overnight jumps
var_rolling <- HighFreq::roll_variance(SPY, calc_method="rogers_satchell")
```

roll_vwap — *Calculate the volume-weighted average price of an* OHLC *time series over a rolling window (lookback period).*

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], win_dow)
```

Arguments

oh_lc	an *OHLC* time series of prices in *xts* format.
x_ts	single-column *xts* time series.
win_dow	the size of the lookback window, equal to the number of bars 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 lookback window, divided by the sum of trading volumes in the window. 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=SPY["2013-11"], win_dow=11)
chart_Series(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=SPY)
# calculate the rolling volume-weighted average returns
roll_vwap(oh_lc=SPY, x_ts=returns_running, win_dow=11)
```

`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, lag = 1, col_umn = 4, sca_le = TRUE)
```

Arguments

`x_ts`	*xts* time series of either *TAQ* or *OHLC* data.
`lag`	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)
`sca_le`	*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 `sca_le` 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 `sca_le` 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=SPY_TAQ)
# calculate close to close returns
re_turns <- HighFreq::run_returns(x_ts=SPY)
# calculate open to open returns
re_turns <- HighFreq::run_returns(x_ts=SPY, col_umn=1)
```

run_sharpe — *Calculate time series of Sharpe-like statistics for each bar of a* OHLC *time series.*

Description

Calculate time series of Sharpe-like statistics for each bar 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`	*character* string representing method for estimating the Sharpe-like exponent.

Details

The function `run_sharpe()` calculates Sharpe-like statistics for each bar 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 bar 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(SPY)
```

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

Description

Calculate time series of 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`	*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 bar 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 skew estimates.

Examples

```
# calculate time series of skew estimates for SPY
sk_ew <- HighFreq::run_skew(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 bars 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", sca_le = TRUE)
```

Arguments

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

`calc_method` *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"`)

`sca_le` *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 bars of data (and possibly from the neighboring bars.) All the methods are implemented assuming zero drift, since the calculations are performed only for a single bar 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 sca_le 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 sca_le 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(SPY, calc_method="yang_zhang")
# calculate SPY variance without overnight jumps
var_running <- HighFreq::run_variance(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/", win_dow = 51, vol_mult = 2,
  period = "minutes", tzone = "America/New_York")
```

Arguments

`sym_bol`	*character* string representing symbol or ticker.
`data_dir`	*character* string representing directory containing input '`*.RData`' files.
`output_dir`	*character* string representing directory containing output '`*.RData`' files.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	quantile multiplier.
`period`	aggregation period.
`tzone`	timezone to convert.

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/")
```

Arguments

`sym_bol`	*character* string representing symbol or ticker.
`data_dir`	*character* string representing directory containing input '`*.RData`' files.
`output_dir`	*character* string representing directory containing output '`*.RData`' files.

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/", win_dow = 51, vol_mult = 2,
  period = "minutes", tzone = "America/New_York")
```

Arguments

`sym_bol`	*character* string representing symbol or ticker.
`data_dir`	*character* string representing directory containing input '`*.RData`' files.
`output_dir`	*character* string representing directory containing output '`*.RData`' files.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	quantile multiplier.
`period`	aggregation period.
`tzone`	timezone to convert.

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/", win_dow = 51, vol_mult = 2,
  tzone = "America/New_York")
```

Arguments

`sym_bol`	*character* string representing symbol or ticker.
`data_dir`	*character* string representing directory containing input '`*.RData`' files.
`output_dir`	*character* string representing directory containing output '`*.RData`' files.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	quantile multiplier.
`tzone`	timezone to convert.

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, win_dow = 51, vol_mult = 2, period = "minutes",
  tzone = "America/New_York")
```

Arguments

ta_q	*TAQ* time series in *xts* format.
win_dow	number of data points for estimating rolling quantile.
vol_mult	quantile multiplier.
period	aggregation period.
tzone	timezone to convert.

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=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, win_dow = 51, vol_mult = 2, tzone = "America/New_York")
```

Arguments

`ta_q`	*TAQ* time series in *xts* format.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	quantile multiplier.
`tzone`	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=SPY_TAQ, win_dow=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, win_dow=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`	single-column *xts* time series.
`in_dex`	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(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"))
```

---

which_extreme — *Calculate a* Boolean *vector that identifies extreme tail values in a single-column* xts *time series or vector, over a rolling window.*

Description

Calculate a *Boolean* vector that identifies extreme tail values in a single-column *xts* time series or vector, over a rolling window.

Usage

```
which_extreme(x_ts, win_dow = 51, vol_mult = 2)
```

Arguments

`x_ts`	A single-column *xts* time series, or a *numeric* or *Boolean* vector.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	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 window.

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 <- 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, win_dow=win_dow, vol_mult=vol_mult)
# 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 window.*

Description

Calculate a *Boolean* vector that identifies isolated jumps (spikes) in a single-column *xts* time series or vector, over a rolling window.

Usage

```
which_jumps(x_ts, win_dow = 51, vol_mult = 2)
```

Arguments

`x_ts`	A single-column *xts* time series, or a *numeric* or *Boolean* vector.
`win_dow`	number of data points for estimating rolling quantile.
`vol_mult`	quantile multiplier.

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 window. 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, win_dow=31, vol_mult=1.0), ] <- NA
ta_q <- zoo::na.locf(ta_q)
```

Index