Risk Analysis and Model Construction

FRE6871 & FRE7241, Fall 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 19, 2024

NYU | TANDON SCHOOL OF ENGINEERING

Tests for Market Timing Skill

Market timing skill is the ability to forecast the direction and magnitude of market returns.

The *market timing* skill can be measured by performing a *linear regression* of a strategy's returns against a strategy with perfect *market timing* skill.

The *Merton-Henriksson* market timing test uses a linear *market timing* term:

$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma \max(R_m - R_f, 0) + \varepsilon$$

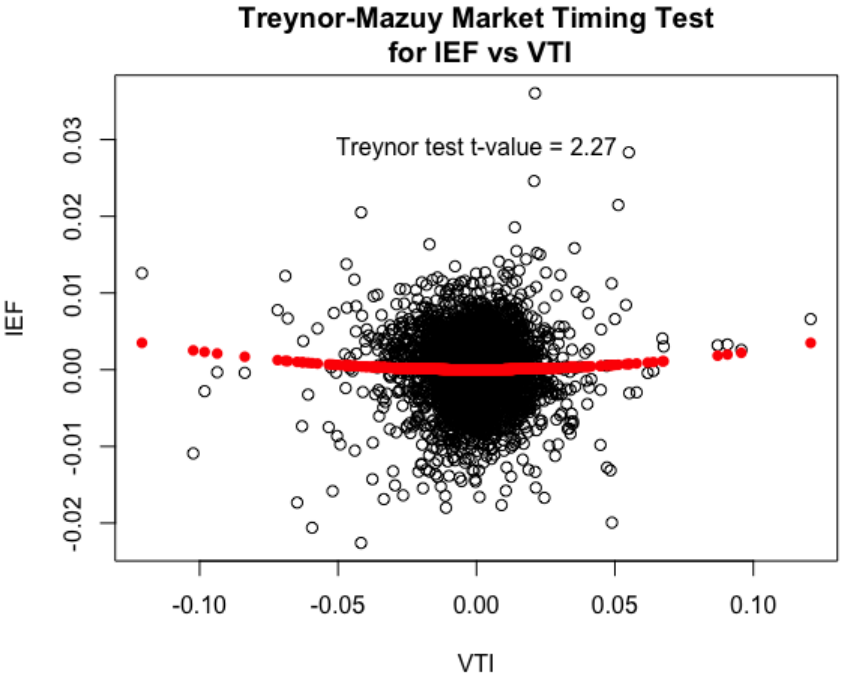
Where R are the strategy returns, R_m are the market returns, and R_f are the risk-free rates.

If the coefficient γ is statistically significant, then it's very likely due to *market timing* skill.

The *market timing* regression is a generalization of the *Capital Asset Pricing Model*.

The *Treynor-Mazuy* test uses a quadratic term, which makes it more sensitive to the magnitude of returns:

$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma(R_m - R_f)^2 + \varepsilon$$




```
> # Test if IEF can time VTI
> retp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI")])
> retvti <- retp$VTI
> desm <- cbind(retp, 0.5*(retvti+abs(retvti)), retvti^2)
> colnames(desm)[3:4] <- c("merton", "treynor")
> # Merton-Henriksson test
> regmod <- lm(IEF ~ VTI + merton, data=desm); summary(regmod)
```

```
> # Treynor-Mazuy test
> regmod <- lm(IEF ~ VTI + treynor, data=desm); summary(regmod)
> # Plot residual scatterplot
> x11(width=6, height=5)
> resids <- (desm$IEF - regmod$coeff["VTI"]*retvti)
> plot.default(x=retvti, y=resids, xlab="VTI", ylab="IEF")
> title(main="Treynor-Mazuy Market Timing Test\n for IEF vs VTI", li
> # Plot fitted (predicted) response values
> coefreg <- summary(regmod)$coeff
> fitv <- regmod$fitted.values - coefreg["VTI", "Estimate"]*retvti
> tvalue <- round(coefreg["treynor", "t value"], 2)
> points.default(x=retvti, y=fitv, pch=16, col="red")
> text(x=0.0, y=0.8*max(resids), paste("Treynor test t-value =", tva
```

draft: Identifying Managers With Skill

Consider a binary investment (gamble) with the probability of winning equal to p, the winning amount (gain) equal to a, and the loss equal to b.

The investor makes no up-front payments, and either wins an amount a, or loses an amount b.

Assuming that an investor makes decisions exclusively on the basis of the expected value of future wealth, then they would choose to invest all their wealth on the gamble if its expected value is positive, and choose not to invest at all if its expected value is negative.

	win	lose
probability	p	q = 1 - p
payout	a	-b

The expected value of the gamble is equal to: $m = p\,a - q\,b$.

The variance of the gamble is equal to: $var = p\,q\,(a+b)^2$.

Without loss of generality we can assume that $p = q = \frac{1}{2}$,
$m = 0.5\,(b-a)$,
$var = 0.25\,(a+b)^2$.

The *Sharpe ratio* of the gamble is then equal to:

$$S_r = \frac{m}{sqrt(var)} = \frac{(b-a)}{(a+b)}$$

Single Period Binary Gamble

Consider a single investment (gamble) with a binary outcome:
The investor makes no up-front payments, and either wins an amount a (with probability p), or loses an amount b (with probability $q = 1 - p$).

	win	lose
probability	p	q = 1 - p
payout	a	-b
terminal wealth	1 + a	1 - b

The initial wealth is equal to 1 dollar, and the terminal wealth after the gamble is either $1 + a$ (with probability p), or $1 - b$ (with probability $q = 1 - p$).

The amounts a and b are expressed as percentages of the wealth risked in the gamble, and the ratio a/b is called the *betting odds*.

The expected return on the gamble is called the *edge* and is equal to: $\mu = p\,a - q\,b$, and the variance of returns is equal to: $\sigma^2 = p\,q\,(a+b)^2$.

If the investor chooses to risk only a fraction k_f of wealth, then the return on the gamble is either $k_f\,a$ (with probability p), or $-k_f\,b$ (with probability $q = 1 - p$).

The fraction k_f can be greater than 1 (leveraged investing), or it can be negative (shorting).

And the expected return on the gamble is equal to: $p\,k_f\,a - q\,k_f\,b = k_f\,\mu$.

If an investor makes decisions exclusively based on the expected return μ, then they would either invest all their wealth ($k_f = 1$) on the gamble if $\mu > 0$, or choose not to invest at all ($k_f = 0$) if $\mu < 0$.

Without loss of generality we can assume that $p = q = \frac{1}{2}$.

And then $\mu = 0.5\,(a - b)$, and $\sigma^2 = 0.25\,(a+b)^2$.

The *Sharpe ratio* of the gamble is then equal to:

$$S_r = \frac{\mu}{\sigma} = \frac{(a-b)}{(a+b)}$$

Investor Utility and Fractional Betting

The *expected utility* hypothesis states that investors try to maximize the expected *utility* of wealth, not the expected wealth.

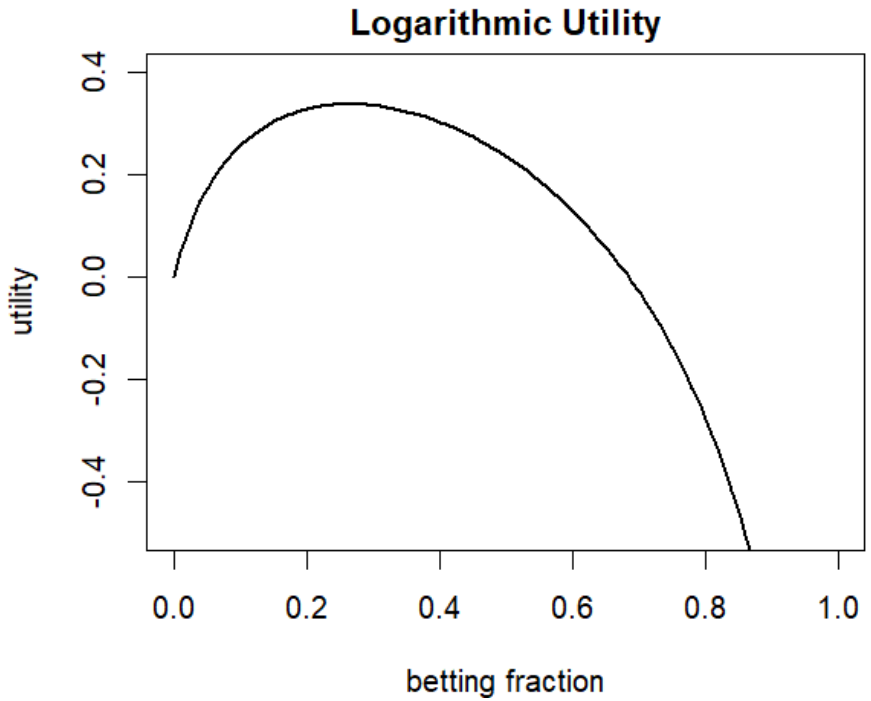
In 1738 Daniel Bernoulli introduced the concept of *logarithmic utility* in his work *"Specimen Theoriae Novae de Mensura Sortis"* (New Theory of the Measurement of Risk).

The *logarithmic utility* function is defined as the logarithm of wealth: $u(w) = \log(w)$.

Under *logarithmic utility* investor preferences depend on the percentage change of wealth, instead of the absolute change of wealth: $\mathrm{d}u(w) = \frac{\mathrm{d}w}{w}$.

An investor with *logarithmic utility* invests only a fraction k_f of their wealth in a gamble, depending on the risk-return of the gamble.

If the initial wealth is equal to 1, then the expected value of *logarithmic utility* for the binary gamble is equal to: $u(k_f) = p\ \log(1 + k_f a) + q\ \log(1 - k_f b)$.




```
> # Define logarithmic utility
> utilfun <- function(frac, p=0.3, a=20, b=1) {
+   p*log(1+frac*a) + (1-p)*log(1-frac*b)
+ }  # end utilfun
> # Plot utility
> curve(expr=utilfun, xlim=c(0, 1),
+ ylim=c(-0.5, 0.4), xlab="betting fraction",
+ ylab="utility", main="", lwd=2)
> title(main="Logarithmic Utility", line=0.5)
```

Optimal Fractional Betting Under Logarithmic Utility

The betting fraction that maximizes the *utility* can be found by equating the derivative of *utility* to zero:

$$\frac{\mathrm{d}u(k_f)}{\mathrm{d}k_f} = \frac{p\,a}{1+k_f\,a} - \frac{q\,b}{1-k_f\,b} = 0$$

$$k_f = \frac{p}{b} - \frac{q}{a} = \frac{p\,a - q\,b}{b\,a} = \frac{\mu}{b\,a}$$

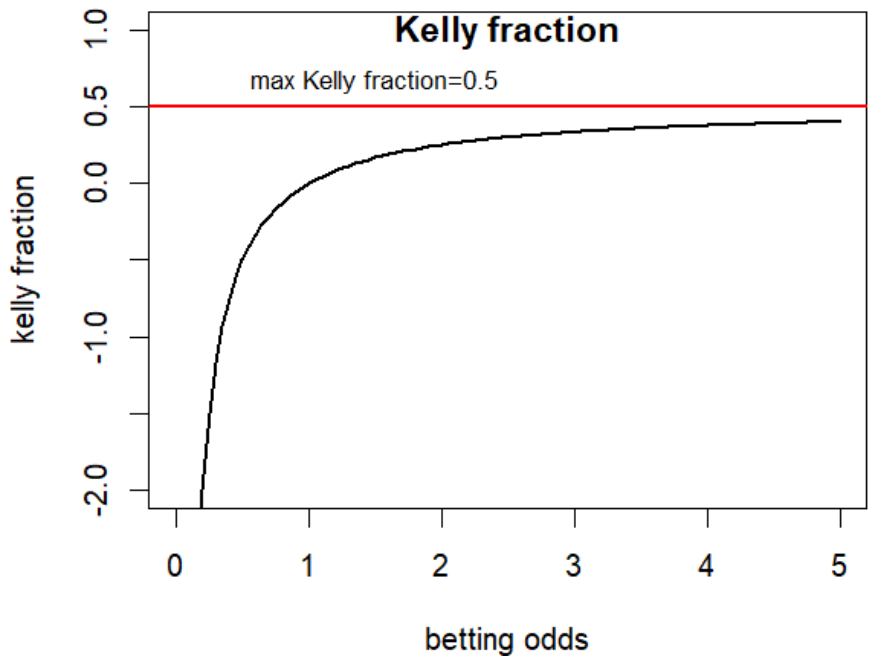
The optimal k_f is called the *Kelly fraction*, and it depends on the parameters of the gamble.

The *Kelly fraction* can be greater than 1 (leveraged investing), or it can be negative (shorting).

If we assume that $b = 1$, then the betting odds are equal to a and the *Kelly fraction* is: $k_f = \frac{p(a+1)-1}{a}$

The *Kelly fraction* is then equal to the expected payout divided by the betting odds.

If the expected payout of the gamble is not positive, then an investor with logarithmic utility should not allocate any capital to the gamble.




```
> # Define and plot Kelly fraction
> kelly_frac <- function(a, p=0.5, b=1) {
+   p/b - (1-p)/a
+ }  # end kelly_frac
> curve(expr=kelly_frac, xlim=c(0, 5),
+ ylim=c(-2, 1), xlab="betting odds",
+ ylab="kelly fraction", main="", lwd=2)
> abline(h=0.5, lwd=2, col="red")
> text(x=1.5, y=0.5, pos=3, cex=0.8, labels="max Kelly fraction=0.5"
> title(main="Kelly fraction", line=-0.8)
```

The Kelly Criterion

The *Kelly criterion* states that investors should bet the optimal *Kelly fraction* of their capital in a gamble.

Investors with concave utility functions (for example logarithmic utility) are sensitive to the risk of ruin (losing all their capital).

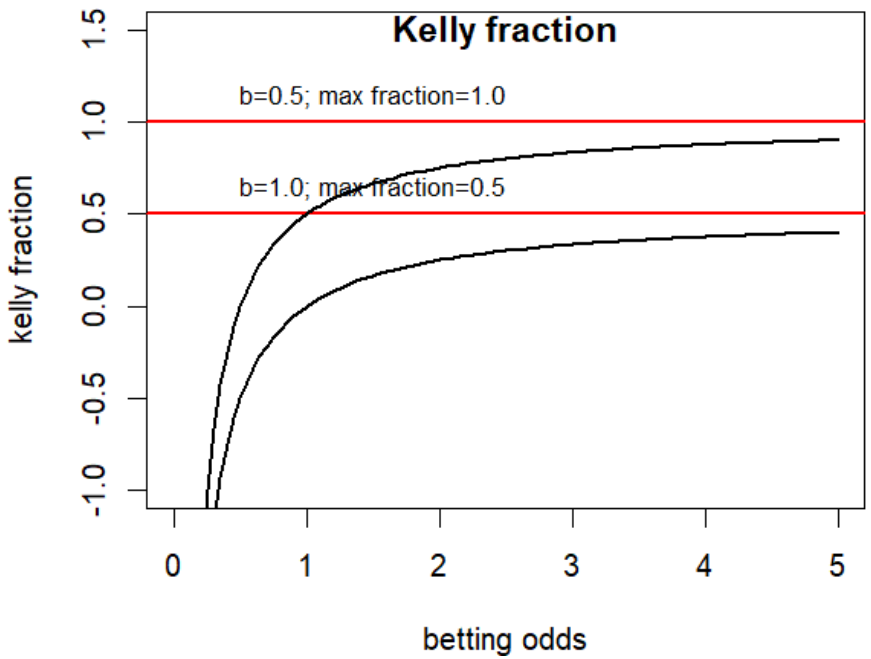
Applying the *Kelly criterion* and betting only a fraction of their capital reduces the risk of ruin (but it doesn't eliminate the risk if prices drop suddenly).

The loss amount b determines the risk of ruin, with larger values of b increasing the risk of ruin.

Therefore investors will choose a smaller betting fraction k_f for larger values of b.

This means that even for huge odds in their favor, investors may not choose to invest all their capital, because of the risk of ruin.

For example, if the betting odds are very large $a \to \infty$, then the *Kelly fraction*: $k_f = \frac{p}{b}$.




```
> # Plot several Kelly curves
> curve(expr=kelly_frac(x, b=1), xlim=c(0, 5),
+ ylim=c(-1, 1.5), xlab="betting odds",
+ ylab="kelly fraction", main="", lwd=2)
> abline(h=0.5, lwd=2, col="red")
> text(x=1.5, y=0.5, pos=3, cex=0.8, labels="b=1.0; max fraction=0.5
> curve(expr=kelly_frac(x, b=0.5), add=TRUE, main="", lwd=2)
> abline(h=1.0, lwd=2, col="red")
> text(x=1.5, y=1.0, pos=3, cex=0.8, labels="b=0.5; max fraction=1.0
> title(main="Kelly fraction", line=-0.8)
```

Utility of Multiperiod Betting

Let r_t be the random return on the gamble in period i, and let $w_i = (1 + k_f r_t)$ be the random wealth increment.

Then the terminal wealth after n rounds is equal to the compounded wealth increments:
$w_n = \prod_{i=1}^{n} w_i = \prod_{i=1}^{n}(1 + k_f r_t)$.

And the utility is equal to the sum of the individual utilities:

$$u_n = \log(w_n) = \sum_{i=1}^{n} \log(w_i) = \sum_{i=1}^{n} \log(1+k_f r_t) = \sum_{i=1}^{n} u_i$$

The individual utilities are all maximized by the same *Kelly fraction* k_f, so the *Kelly fraction* for multiperiod betting is the same as for single period betting:

$$k_f = \frac{p}{b} - \frac{q}{a}$$

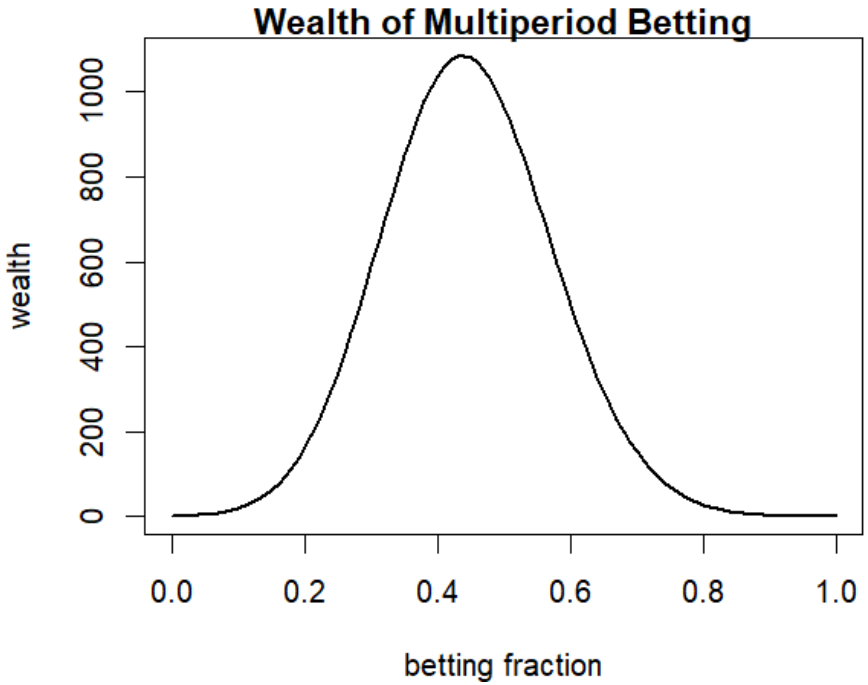
Wealth of Multiperiod Betting

In multiperiod betting the investor participates in n rounds of gambles, and in each round they risk a fixed fraction k_f of their current outstanding wealth.

In each round the wealth is multiplied by either $(1 + k_f a)$ (win) or $(1 - k_f b)$ (loss), so that the current outstanding wealth changes over time.

The terminal wealth after n rounds with m wins is equal to: $w(k_f) = (1 + k_f a)^m (1 - k_f b)^{n-m}$.

If the number of rounds n is very large, then the number of wins is almost always equal to $m = n\,p$, and the terminal wealth is equal to: $w(k_f) = (1 + k_f a)^{np} (1 - k_f b)^{nq}$.




```
> # Wealth of multiperiod binary betting
> wealthv <- function(f, a=0.8, b=0.1, n=1e3, i=150) {
+   (1+f*a)^i * (1-f*b)^(n-i)
+ }  # end wealth
> curve(expr=wealthv, xlim=c(0, 1),
+ xlab="betting fraction",
+ ylab="wealth", main="", lwd=2)
> title(main="Wealth of Multiperiod Betting", line=0.1)
```

Optimal Multiperiod Betting

The betting fraction k_f that maximizes the terminal wealth is found by setting the derivative of $w(k_f)$ to zero:

$$\frac{\mathrm{d}w(k_f)}{\mathrm{d}k_f} = npa(1+k_f a)^{np-1}(1-k_f b)^{nq} - nqb(1+k_f a)^{np}(1-k_f b)^{nq-1}$$

$$= \left(\frac{npa}{1+k_f a} - \frac{nqb}{1-k_f b}\right)(1+k_f a)^{np}(1-k_f b)^{nq} = 0$$

We can then solve for the optimal betting fraction k_f:

$$\frac{pa}{1+k_f a} - \frac{qb}{1-k_f b} = 0$$

$$pa(1-k_f b) - qb(1+k_f a) = 0$$

$$pa - qb - k_f ab = 0$$

$$k_f = \frac{pa-qb}{ab} = \frac{p}{b} - \frac{q}{a}$$

The above is just the *Kelly fraction* k_f that maximizes the utility.

So the *Kelly fraction* k_f that maximizes the utility also maximizes the terminal wealth.

depr: Multiperiod Binary Gambles

The terminal wealth after n repeated gambles with m wins is equal to: $(1 + k_f a)^m (1 - k_f b)^{n-m}$.

And the expected value of the wealth is equal to:

$$w(k_f) = \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} (1 + k_f a)^m (1 - k_f b)^{n-m}$$

We can then find the fraction k_f which maximizes the expected wealth $w(k_f)$:

$$\frac{\mathrm{d}w(k_f)}{\mathrm{d}k_f} = \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} (1 + k_f a)^m (1 - k_f b)^{n-m} \left(\frac{am}{1 + k_f a} - \frac{b(n-m)}{1 - k_f b}\right) =$$

$$\frac{a}{1 + k_f a} \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} m -$$

$$\sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} (1 + k_f a)^m (1 - k_f b)^{n-m} \left(\frac{am}{1 + k_f a} - \frac{b(n-m)}{1 - k_f b}\right)$$

If the investor chooses to risk only a fraction k_f of wealth, then the wealth after the gamble is either $1 + k_f a$ (with probability p), or $1 - k_f b$ (with probability $q = 1 - p$).

(with probability p), or $1 - b$ (with probability $q = 1 - p$). initial wealth is equal to 1, and the The *Kelly fraction* for multiperiod betting can be found by maximizing the expected *utility* of the final wealth distribution:

$$u(k_f) = \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} \log((1 + k_f a)^m (1 - k_f b)^{n-m})$$

depr: Utility of Multiperiod Binary Gambles

The *Kelly fraction* for multiperiod betting can be found by maximizing the expected *utility* of the final wealth distribution:

$$u(k_f) = \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} \log((1 + k_f a)^m (1 - k_f b)^{n-m})$$

$$= \log(1 + k_f a) \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} m +$$

$$\log(1 - k_f b) \sum_{m=0}^{n} \binom{n}{m} p^m q^{n-m} (n - m)$$

$$= n\,p\, \log(1 + k_f a) + n\,q\, \log(1 - k_f b)$$

The above is just the single period *utility* multiplied by the number of rounds of betting n.

The *Kelly fraction* k_f for multiperiod betting is the same as for single period betting:

$$k_f = \frac{p}{b} - \frac{q}{a}$$

Investing With Fixed Margin

Let r_t be the percentage returns on a *risky asset*, so that the asset price p_t at time t is given by:

$$p_t = p_0 \prod_{i=1}^{t} (1 + r_t)$$

The initial investor wealth at time $t = 0$ is equal to 1 dollar, and they also borrow on margin m dollars to invest in the *risky asset*.

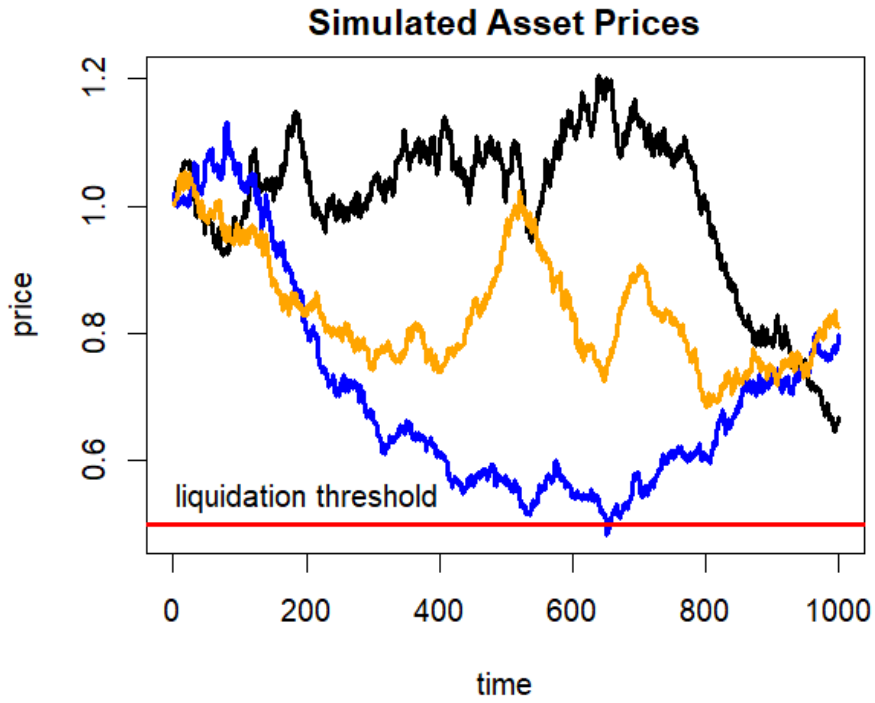
The investor's *wealth* at time t is equal to (the margin borrowing rate is assumed to be zero):

$$w_t = 1 + m \frac{p_t - p_0}{p_0}$$

The *leverage* k_f is equal to the *margin debt* m divided by the total wealth w_t: $k_f = m/w_t$.

If the asset price drops then the *leverage* increases, because the *margin debt* is fixed while the wealth drops.

If the asset price drops enough so that the wealth reaches zero, then the investment is liquidated and the investor is ruined.




```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Simulate asset prices
> calc_pricev <- function(x) cumprod(1 + rnorm(1e3, sd=0.01))
> price_paths <- sapply(1:3, calc_pricev)
> plot(price_paths[, 1], type="l", lwd=3,
+      main="Simulated Asset Prices",
+      ylim=range(price_paths),
+      lty="solid", xlab="time", ylab="price")
> lines(price_paths[, 2], col="blue", lwd=3)
> lines(price_paths[, 3], col="orange", lwd=3)
> abline(h=0.5, col="red", lwd=3)
> text(x=200, y=0.5, pos=3, labels="liquidation threshold")
```

Investing With Fixed Leverage

In order to avoid ruin, the investor may choose to maintain a fixed *leverage ratio* equal to k_f, so that the amount invested in the *risky asset* is proportional to the *wealth*: $k_f\, w_t$.

This requires buying the *risky asset* when its price increases, and selling it when it drops.

The return on the *risky asset* in a single period is equal to: $k_f\, w_t\, r_t$, so the *terminal wealth* at time t is equal to the compounded returns:

$$w_t = (1 + k_f\, r_1) \ldots (1 + k_f\, r_t) = \prod_{i=1}^{t} (1 + k_f\, r_t)$$

The utility of the *terminal wealth* is equal to the sum of the utilities of single periods:

$$\mathbb{E}[\log w_t] = \mathbb{E}[\log((1 + k_f\, r_1) \ldots (1 + k_f\, r_t))]$$

$$= \sum_{i=1}^{t} \mathbb{E}[\log\,(1 + k_f\, r_t)] = t\, \mathbb{E}[\log\,(1 + k_f\, r)]$$

The last equality holds because all the utilities of single periods are the same.

Let the returns over a short time period be equal to r, with probability distribution $p(r)$.

The mean return $\bar{r}$, and variance σ^2 are:

$$\bar{r} = \int r\, p(r)\, \mathrm{d}r\ ; \quad \sigma^2 = \int (r - \bar{r})^2\, p(r)\, \mathrm{d}r$$

Since the returns are over a short time period, we have: $r \ll 1$ and $\bar{r} \ll \sigma$, so that we can replace $r - \bar{r}$ with r as follows:

$$\int (r - \bar{r})^2\, p(r)\, \mathrm{d}r \approx \int r^2\, p(r)\, \mathrm{d}r$$

Utility of Leveraged Asset Returns

So the utility of the *terminal wealth* u_t is equal to the utility of a single period times the number of periods:

$$u_t = \mathbb{E}[\log w_t] = t\, \mathbb{E}[\log (1 + k_f\, r)] = t\, u_r$$

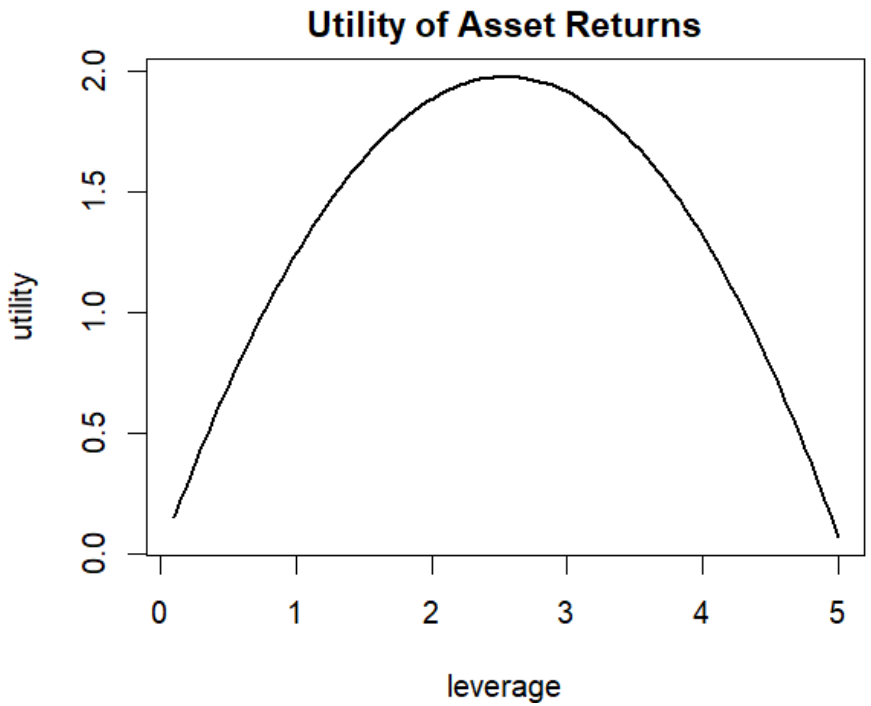
The utility of the asset returns u_r is equal to:

$$u_r = \mathbb{E}[\log (1 + k_f\, r)] = \int \log(1 + k_f\, r)\, p(r)\, \mathrm{d}r$$

The leverage k_f is limited so that $(1 + k_f\, r) > 0$ for all return values r.

If the mean returns are positive, then at first the utility increases with leverage, but only up to a point.

With higher leverage, the negative utility of the time periods with negative returns becomes significant, forcing the aggregate utility to drop.




```
> # Calculate the VTI returns
> retp <- rutils::etfenv$returns$VTI
> retp <- na.omit(retp)
> c(mean=mean(retp), stdev=sd(retp))
> range(retp)
```

```
> # Define vectorized logarithmic utility function
> utilfun <- function(kellyfrac, retp) {
+   sapply(kellyfrac, function(x) sum(log(1 + x*retp)))
+ }  # end utilfun
> utilfun(1, retp)
> utilfun(c(1, 4), retp)
> # Plot the logarithmic utility
> curve(expr=utilfun(x, retp=retp),
+ xlim=c(0.1, 5), xlab="leverage", ylab="utility",
+ main="Utility of Asset Returns", lwd=2)
```

Kelly Criterion for Optimal Leverage of Asset Returns

The *logarithmic utility* u_r can be expanded in the moments of the return distribution:

$$u_r = \mathbb{E}[\log(1 + k_f\, r)] = \int \log(1 + k_f\, r)\, p(r)\, \mathrm{d}r$$

$$= \int (k_f\, r - \frac{(k_f\, r)^2}{2} + \frac{(k_f\, r)^3}{3} - \frac{(k_f\, r)^4}{4})\, p(r)\, \mathrm{d}r$$

$$= k_f \bar{r} - \frac{k_f^2 \sigma^2}{2} + \frac{k_f^3 \sigma^3 \varsigma}{3} - \frac{k_f^4 \sigma^4 \kappa}{4}$$

Where $\varsigma = \int \frac{r^3}{\sigma^3}\, p(r)\, \mathrm{d}r$ is the *skewness*, and $\kappa = \int \frac{r^4}{\sigma^4}\, p(r)\, \mathrm{d}r$ is the *kurtosis*.

The *Kelly leverage* which maximizes the *utility* is found by equating the derivative of *utility* to zero:

$$\frac{\mathrm{d}u_r}{\mathrm{d}k_f} = \bar{r} - k_f \sigma^2 + k_f^2 \sigma^3 \varsigma - k_f^3 \sigma^4 \kappa = 0$$

This shows that the logarithmic utility has positive odd derivatives and negative even derivatives.

Assuming that the third and fourth moments $\sigma^4 \varsigma$ and $\sigma^4 \kappa$ are small and can be neglected, we get:

$$k_f = \frac{\bar{r}}{\sigma^2} = \frac{S_r}{\sigma}\ ; \quad u_r = \frac{1}{2}\frac{\bar{r}^2}{\sigma^2} = \frac{1}{2} S_r^2$$

The *Kelly leverage* is *approximately* equal to the *Sharpe ratio* divided by the *standard deviation*.

The optimal utility u_r is *approximately* equal to half the *Sharpe ratio* S_r squared.

The *standard deviation* and *Sharpe ratio* are calculated over the same time interval as the returns (not annualized).

```
> # Approximate Kelly leverage
> mean(retp)/var(retp)
> PerformanceAnalytics::KellyRatio(R=retp, method="full")
> # Kelly leverage
> unlist(optimize(
+   f=function(x) -utilfun(x, retp),
+   interval=c(1, 4)))
```

Kelly Strategy Wealth Path

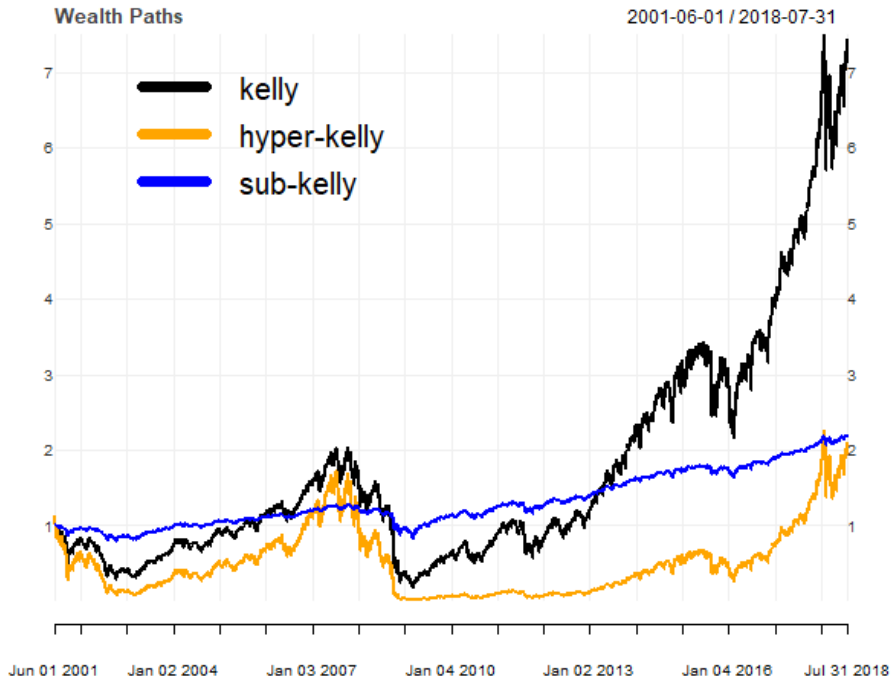
The wealth of a Kelly Strategy with a fixed leverage ratio k_f is equal to:

$$w_t = \prod_{i=1}^{t} (1 + k_f \, r_t)$$

The *Kelly fraction* k_f provides the optimal leverage to maximize the utility of wealth, by balancing the benefit of leveraging higher positive returns, with the risk of ruin due to excessive leverage.

If the mean asset returns are positive, then a higher leverage ratio provides higher returns.

But if the leverage is too high, then the losses in periods with negative returns wipe out most of the wealth, so then it's slow to recover.




```
> retp <- na.omit(retp)
> # Calculate the wealth paths
> kelly_ratio <- drop(mean(retp)/var(retp))
> kelly_wealthv <- cumprod(1 + kelly_ratio*retp)
> hyper_kelly <- cumprod(1 + (kelly_ratio+2)*retp)
> sub_kelly <- cumprod(1 + (kelly_ratio-2)*retp)
> kelly_paths <- cbind(kelly_wealth, hyper_kelly, sub_kelly)
> colnames(kelly_paths) <- c("kelly", "hyper-kelly", "sub-kelly")
```

```
> # Plot wealth paths
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "orange", "blue")
> quantmod::chart_Series(kelly_paths, theme=plot_theme, name="Wealth
> legend("topleft", legend=colnames(kelly_paths),
+  inset=0.1, bg="white", lty=1, lwd=6, y.intersp=0.5,
+  col=plot_theme$col$line.col, bty="n")
```

Kelly Strategy With Margin Account

The *margin debt* m_t is equal to the dollar amount borrowed to purchase the *risky asset*.

The wealth w_t at time t is equal to the initial wealth $w_0 = 1$ plus the dollar amount of the *risky asset* a_t, minus the *margin debt* m_t: $w_t = 1 + a_t - m_t$.

The dollar amount of the *risky asset* a_t is equal to the wealth w_t times the *leverage* k_f: $a_t = k_f w_t$.

So the *margin debt* m_t is proportional to the wealth w_t: $m_t = (k_f - 1)w_t + 1$.

The wealth changes from w_{t-1} to: $w_t = w_{t-1}(1 + k_f\, r_t)$, while the dollar amount of the *risky asset* changes from $a_{t-1} = k_f w_{t-1}$ to: $a_t = k_f w_{t-1}(1 + r_t)$, so that the leverage changes from k_f to:

$$\frac{k_f w_{t-1}(1 + r_t)}{w_{t-1}(1 + k_f\, r_t)} = \frac{k_f(1 + r_t)}{1 + k_f\, r_t}$$

In order to maintain a fixed *leverage ratio* equal to k_f, the investor must actively trade the *risky asset*, and the *margin debt* m_t changes over time.

The change in margin in a single time period is equal to:

$$\Delta m_t = (k_f - 1)\Delta w_t = k_f(k_f - 1)w_{t-1} r_t$$

The dollar amount of the *risky asset* traded is equal to the change in *margin*.

Therefore the investor must borrow on margin and buy the *risky asset* when its price increases, and sell it when it drops.

Kelly Strategy With Transaction Costs of Trading

The *bid-ask spread* is the percentage difference between the *offer* minus the *bid* price, divided by the *mid* price.

The *bid-ask spread* for liquid stocks can be assumed to be about 10 basis points (bps).

The *transaction costs* c^r due to the *bid-ask spread* are equal to half the *bid-ask spread* δ times the absolute value of the traded dollar amount of the *risky asset*:

$$c^r = \frac{\delta}{2} |\Delta m_t|$$

If the transaction costs are much less than the change in wealth $c^r \ll |\Delta w_t|$, then we can write approximately:

$$c^r = \frac{\delta}{2} k_f (k_f - 1) w_{t-1} |r_t|$$

The wealth of the Kelly Strategy after accounting for the *bid-ask spread* is then equal to:

$$w_t = \prod_{i=1}^{t} (1 + k_f \, r_t - \frac{\delta}{2} k_f (k_f - 1) |r_t|)$$

The effect of the *bid-ask spread* is to reduce the effective asset returns by an amount proportional to the *bid-ask spread*.

```
> # bidask equal to 1 bp for liquid ETFs
> bidask <- 0.001
> # Calculate the wealth paths
> kelly_ratio <- drop(mean(retp)/var(retp))
> wealthv <- cumprod(1 + kelly_ratio*retp)
> wealth_trans <- cumprod(1 + kelly_ratio*retp -
+   0.5*bidask*kelly_ratio*(kelly_ratio-1)*abs(retp))
> # Calculate the compounded wealth from returns
> wealthv <- cbind(wealthv, wealth_trans)
> colnames(wealthv) <- c("Kelly", "Including bid-ask")
> # Plot compounded wealth
> dygraphs::dygraph(wealthv, main="Kelly Strategy With Transaction C
+   dyOptions(colors=c("green", "blue"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

draft: The Half-Kelly Criterion

In reality investors don't know the probability of winning or the odds of the gamble, so they can't accurately Calculate the optimal *Kelly fraction*.

The *Kelly fraction*: $k_f = \frac{\bar{r}}{\sigma^2}$ is especially sensitive to the uncertainty of the expected returns $\bar{r}$.

If the expected returns are over-estimated, then it can produce an inflated value of the *Kelly fraction*, leading to ruin.

The risk of applying too much leverage (over-betting) is much greater than the risk of applying too little leverage (under-betting).

Too much leverage (over-betting) not only reduces returns, but it increases the risk of ruin.

So in practice many investors apply only half the theoretical *Kelly fraction* (the Half-Kelly), to reduce the risk of ruin.

Perform bootstrap simulation to obtain the standard error of the *Kelly fraction*.

```
> # Plot several Kelly curves
> curve(expr=kelly_frac(x, b=1), xlim=c(0, 5),
+ ylim=c(-1, 1.5), xlab="betting odds",
+ ylab="kelly fraction", main="", lwd=2)
> abline(h=0.5, lwd=2, col="red")
> text(x=1.5, y=0.5, pos=3, cex=0.8, labels="b=1.0; max fraction=0.5
> curve(expr=kelly_frac(x, b=0.5), add=TRUE, main="", lwd=2)
> abline(h=1.0, lwd=2, col="red")
> text(x=1.5, y=1.0, pos=3, cex=0.8, labels="b=0.5; max fraction=1.0
> title(main="Kelly fraction", line=-0.8)
```

Risk Aversion

Risk aversion is the investor preference to avoid losses more than to seek similar percentage gains in wealth.

For example, for a risk averse investor, a 10% loss of wealth is more important than a 10% gain.

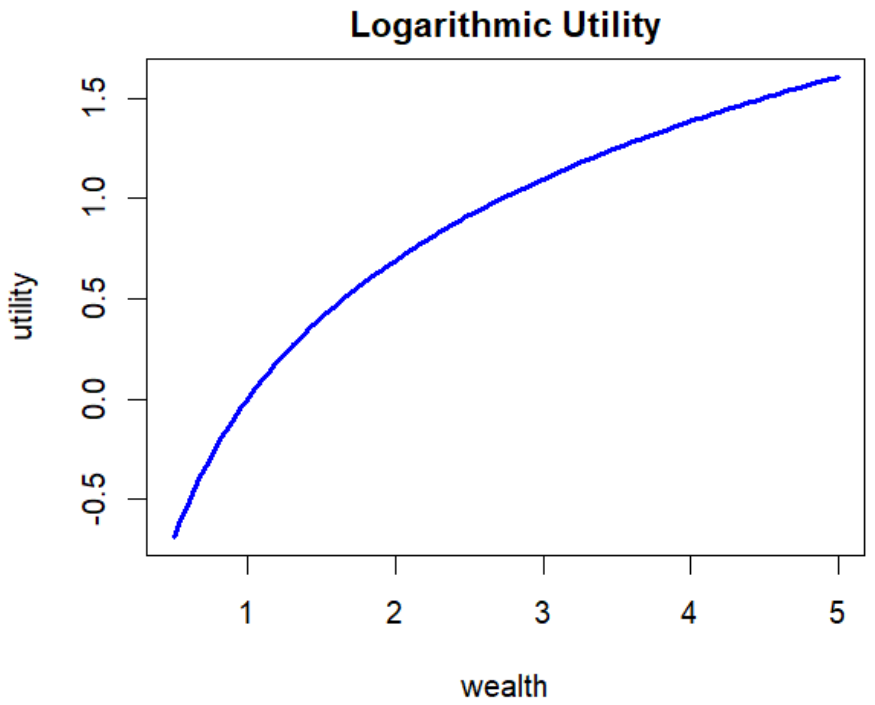
Risk aversion is associated with the *diminishing marginal utility* of the percentage change in wealth Δw.

This manifests itself as a concave utility function, with a negative second derivative $u''(w) < 0$.

For example, the *logarithmic utility* function is concave.

The Arrow-Pratt coefficient of relative risk aversion is proportional to the convexity $u''(w)$ of the utility, and is defined as: $\eta = -\frac{w\, u''(w)}{u'(w)}$.

The relative risk aversion of *logarithmic utility* is equal to one: $\eta = 1$.




```
> # Plot logarithmic utility function
> curve(expr=log, lwd=3, col="blue", xlim=c(0.5, 5),
+ xlab="wealth", ylab="utility",
+ main="Logarithmic Utility")
```

Constant Relative Risk Aversion

It's not a given that all investors have a risk aversion coefficient equal to 1, and other *utility functions* are possible.

The Constant Relative Risk Aversion (*CRRA*) utility function is a generalization of logarithmic utility:

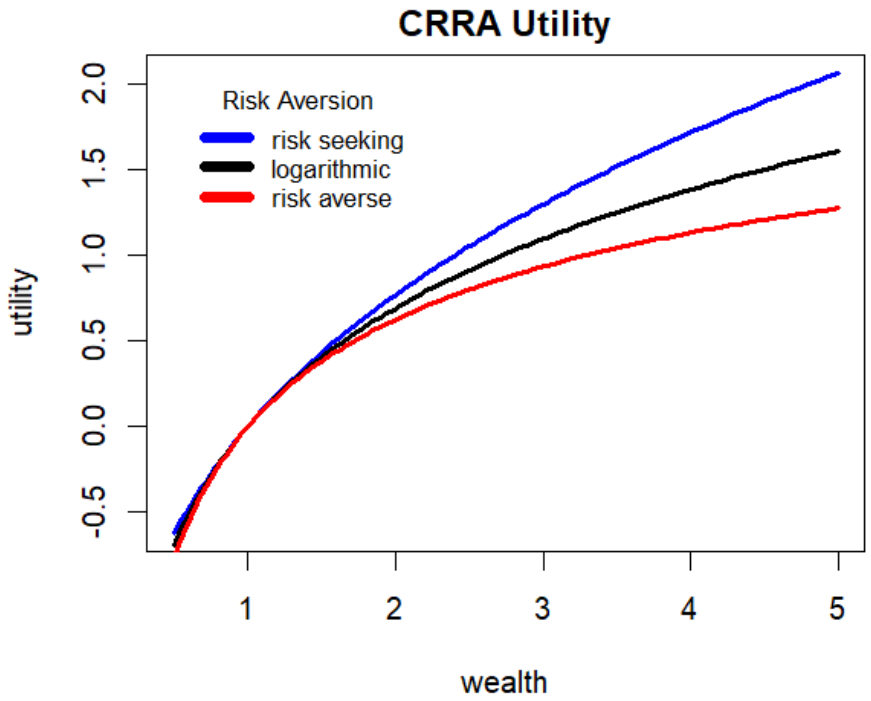
$$u(w) = \frac{w^{1-\eta} - 1}{1 - \eta}$$

Where η is the risk aversion parameter.

The relative risk aversion of the *CRRA* utility function is constant and equal to η.

When the risk aversion parameter is equal to one $\eta = 1$, then the *CRRA* utility function is equal to the logarithmic utility.

In practice, the risk aversion parameter η is not known, and must be estimated through empirical studies.




```
> # Define CRRA utility
> cr_ra <- function(w, ra) {
+   (w^(1-ra) - 1)/(1-ra)
+ }  # end cr_ra
> # Plot utility functions
> curve(expr=cr_ra(x, ra=0.7), xlim=c(0.5, 5), lwd=3,
+ xlab="wealth", ylab="utility", main="", col="blue")
> curve(expr=log, add=TRUE, lwd=3)
> curve(expr=cr_ra(x, ra=1.3), add=TRUE, lwd=3, col="red")
> # Add title and legend
> title(main="CRRA Utility", line=0.5)
> legend(x="topleft", legend=c("risk seeking", "logarithmic", "risk
+  title="Risk Aversion", inset=0.05, cex=0.8, bg="white", y.intersp
+  lwd=6, lty=1, bty="n", col=c("blue", "black", "red"))
```

draft: CRRA Optimal Leverage

It's not a given that all investors have a risk aversion coefficient equal to 1, and other *utility functions* are possible.

The Constant Relative Risk Aversion (*CRRA*) utility function is a generalization of logarithmic utility:

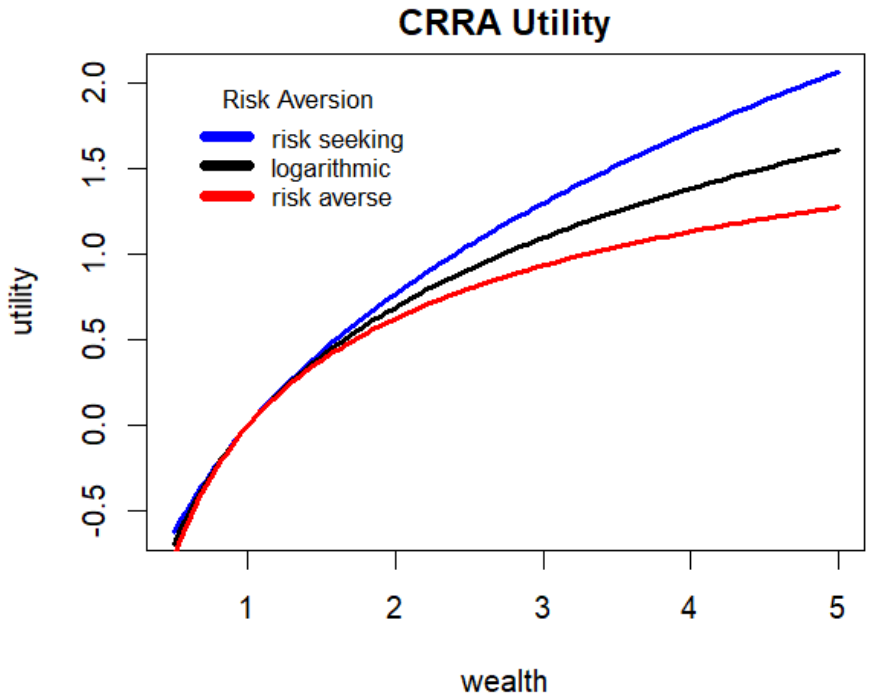
$$u(w) = \frac{w^{1-\eta} - 1}{1 - \eta}$$

Where η is the risk aversion parameter.

The relative risk aversion of the *CRRA* utility function is constant and equal to η.

When the risk aversion parameter is equal to one $\eta = 1$, then the *CRRA* utility function is equal to the logarithmic utility.

In practice, the risk aversion parameter η is not known, and must be estimated through empirical studies.




```
> # Define CRRA utility
> cr_ra <- function(w, ra) {
+   (w^(1-ra) - 1)/(1-ra)
+ }  # end cr_ra
> # Plot utility functions
> curve(expr=cr_ra(x, ra=0.7), xlim=c(0.5, 5), lwd=3,
+ xlab="wealth", ylab="utility", main="", col="blue")
> curve(expr=log, add=TRUE, lwd=3)
> curve(expr=cr_ra(x, ra=1.3), add=TRUE, lwd=3, col="red")
> # Add title and legend
> title(main="CRRA Utility", line=0.5)
> legend(x="topleft", legend=c("risk seeking", "logarithmic", "risk
+  title="Risk Aversion", inset=0.05, cex=0.8, bg="white", y.intersp
+  lwd=6, lty=1, bty="n", col=c("blue", "black", "red"))
```

draft: CRRA Strategy Wealth Path

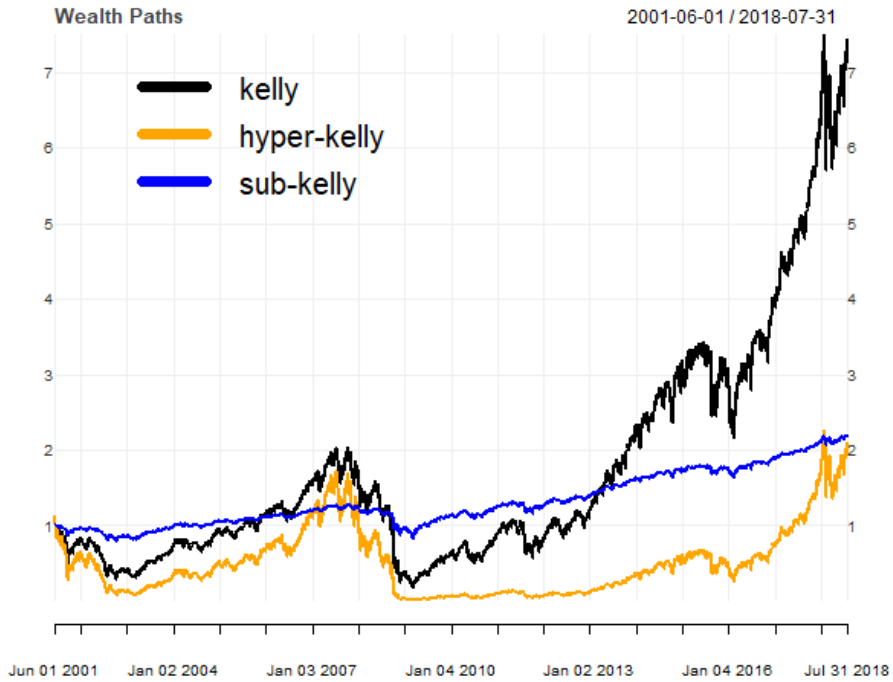
The wealth of a Kelly Strategy with a fixed leverage ratio k_f is equal to:

$$w_t = \prod_{i=1}^{t} (1 + k_f \, r_t)$$

The *Kelly fraction* k_f provides the optimal leverage to maximize the utility of wealth, by balancing the benefit of leveraging higher positive returns, with the risk of ruin due to excessive leverage.

If the mean asset returns are positive, then a higher leverage ratio provides higher returns.

But if the leverage is too high, then the losses in periods with negative returns wipe out most of the wealth, so then it's slow to recover.




```
> retp <- na.omit(retp)
> # Calculate the wealth paths
> kelly_ratio <- drop(mean(retp)/var(retp))
> kelly_wealthv <- cumprod(1 + kelly_ratio*retp)
> hyper_kelly <- cumprod(1 + (kelly_ratio+2)*retp)
> sub_kelly <- cumprod(1 + (kelly_ratio-2)*retp)
> kelly_paths <- cbind(kelly_wealth, hyper_kelly, sub_kelly)
> colnames(kelly_paths) <- c("kelly", "hyper-kelly", "sub-kelly")
```

```
> # Plot wealth paths
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "orange", "blue")
> quantmod::chart_Series(kelly_paths, theme=plot_theme,
+         name="Wealth Paths")
> legend("topleft", legend=colnames(kelly_paths),
+  inset=0.1, bg="white", lty=1, lwd=6, y.intersp=0.5,
+  col=plot_theme$col$line.col, bty="n")
```

The Utility of Lottery Tickets

Lottery tickets are equivalent to binary gambles with a very small probability of winning p, but a very large winning amount a, and a small loss amount b equal to the ticket price.

The expected payout $\mu = p\,a - q\,b$ of most lottery tickets is negative.

So under *logarithmic utility*, the Kelly fraction k_f for most lottery tickets is also negative, meaning that investors should not be expected to buy these lottery tickets.

But in reality many people do buy lottery tickets with negative expected payouts, which means that their utility functions are not logarithmic.

The demand for lottery tickets can be explained by assuming a strong demand for positive *skewness*, which exceeds the demand for a positive payout.

People buy lottery tickets because they want a small chance of a very large payout, even if the average payout is negative.

Without loss of generality we can assume that the lottery ticket price is one dollar $b = 1$, that it pays out a dollars, and that the expected payout is equal to zero: $\mu = p\,a - q\,b = 0$.

Then the probabilities of winning and losing are equal to: $p = \frac{1}{a+1}$ and $q = \frac{a}{a+1}$.

The variance is equal to: $\sigma^2 = p\,q\,(a+1)^2 = a$.

And the *skewness* is equal to:
$\varsigma = \frac{1}{\sigma^3}\left(\frac{a^3}{a+1} - \frac{a}{a+1}\right) = \frac{a-1}{\sqrt{a}}$.

So the positive *skewness* of a lottery ticket increases as the square root of the *betting odds* a, and it can become very large for large *betting odds*.

Investor *Risk Aversion*, *Prudence* and *Temperance*

Investor risk and return preferences depend on the signs of the derivatives of their *utility* function.

Investors with *logarithmic utility* have positive *odd* derivatives ($u'(w) > 0$ and $u'''(w) > 0$) and negative *even* derivatives ($u''(w) < 0$ and $u''''(w) < 0$), which is typical for most other investors as well.

Risk averse investors have a negative second derivative of utility $u''(w) < 0$.

The demand for lottery tickets shows that investors' utility typically has a positive third derivative $u'''(w) > 0$.

Positive *odd* derivatives imply a preference for larger *odd moments* of the change in the wealth distribution (mean, skewness).

Negative *even* derivatives imply a preference for smaller *even moments* (variance, kurtosis).

The preference for smaller *variance* is called *risk aversion*, for larger *skewness* is called *prudence*, and for smaller *kurtosis* is called *temperance*.

The expected change of the *utility* of wealth $\mathbb{E}[\Delta u(w)]$ can be expanded in the moments of the wealth distribution Δw:

$$\mathbb{E}[\Delta u(w)] = u'(w)\mathbb{E}[\Delta w] + \frac{u''(w)}{2}\sigma^2 + \frac{u'''(w)}{3!}\mu 3 + \frac{u''''(w)}{4!}\mu 3$$

Where $\mathbb{E}[\Delta w]$ is the expected change of wealth, $\sigma^2 = \int \Delta w^2\, p(w)\, \mathrm{d}w$ is the *variance* of The change in wealth, and $\mu 3 = \int \Delta w^3\, p(w)\, \mathrm{d}w = \sigma^3 \varsigma$ and $\mu 4 = \int \Delta w^4\, p(w)\, \mathrm{d}w = \sigma^4 \kappa$ are the third and fourth moments, proportional to the *skewness* ς and the *kurtosis* κ.

Investor Preferences and Empirical Return Distributions

The investor preference for higher *returns* and for lower *volatility* is expressed by maximizing the *Sharpe ratio*.

The third and fourth moments of asset returns are usually much smaller than the *variance*, so they typically have a smaller effect on the investor risk and return preferences.

Nevertheless, there is evidence that investors also have significant preferences for positive *skewness* and lower *kurtosis*.

But stock returns typically have negative *skewness* and excess *kurtosis*, the opposite of what investors prefer.

Many investors may prefer positive *skewness*, even at the expense of lower *returns*, similar to the buyers of lottery tickets.

A paper by Amaya asks if the *Realized Skewness Predicts the Cross-Section of Equity Returns?*

But higher moments are hard to estimate accurately from low frequency (daily) returns, which makes empirical investigations more difficult.

```
> # Calculate the VTI returns
> retp <- rutils::etfenv$returns$VTI
> retp <- na.omit(retp)
> # Calculate the higher moments of VTI returns
> c(mean=sum(retp),
+   variance=sum(retp^2),
+   mom3=sum(retp^3),
+   mom4=sum(retp^4))/NROW(retp)
> # Calculate the higher moments of minutely SPY returns
> spy <- HighFreq::SPY[, 4]
> spy <- na.omit(spy)
> spy <- rutils::diffit(log(spy))
> c(mean=sum(spy),
+   variance=sum(spy^2),
+   mom3=sum(spy^3),
+   mom4=sum(spy^4))/NROW(spy)
```

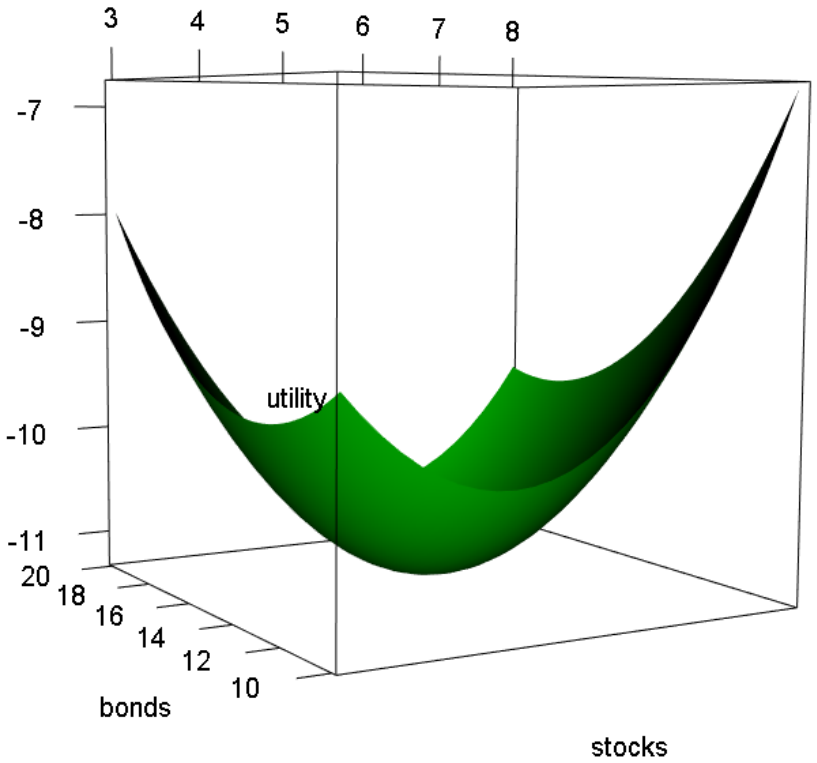
Utility of Stock and Bond Portfolio

The utility u of the stock and bond portfolio with weights *stocku*, *bondu* is equal to:

$$u = \sum_{i=1}^{n} \log(1 + stocku\, r_i^s + bondu\, r_i^b)$$

Where r_i^s, r_i^b are the stock and bond returns.

```
> retp <- na.omit(rutils::etfenv$returns[, c("VTI", "IEF")])
> # Logarithmic utility of stock and bond portfolio
> utilfun <- function(stocku, bondu) {
+   -sum(log(1 + stocku*retp$VTI + bondu*retp$IEF))
+ }  # end utilfun
> # Create matrix of utility values
> stocku <- seq(from=3, to=7, by=0.2)
> bondu <- seq(from=12, to=20, by=0.2)
> utilm <- sapply(bondu, function(y) sapply(stocku,
+   function(x) utilfun(x, y)))
> # Set rgl options and load package rgl
> options(rgl.useNULL=TRUE)
> library(rgl)
> # Draw 3d surface plot of utility
> rgl::persp3d(stocku, bondu, utilm, col="green",
+   xlab="stocks", ylab="bonds", zlab="utility")
> # Render the surface plot
> rgl::rglwidget(elementId="plot3drgl")
> # Save the surface plot to png file
> rgl::rgl.snapshot("utility_surface.png")
```



Kelly Optimal Weights

The Kelly optimal stock and bond portfolio weights *stocku*, *bondu* can be calculated by maximizing the utility *u*.

```
> # Approximate Kelly weights
> weightv <- sapply(retp, function(x) mean(x)/var(x))
> # Kelly weight for stocks
> unlist(optimize(f=function(x) utilfun(x, bondu=0), interval=c(1,
> # Kelly weight for bonds
> unlist(optimize(f=function(x) utilfun(x, stocku=0), interval=c(1
> # Vectorized utility of stock and bond portfolio
> utility_vec <- function(weightv) {
+   utilfun(weightv[1], weightv[2])
+ }  # end utility_vec
> # Optimize with respect to vector argument
> optiml <- optim(fn=utility_vec, par=c(3, 10),
+           method="L-BFGS-B",
+           upper=c(8, 20), lower=c(2, 5))
> # Exact Kelly weights
> optiml$par
```

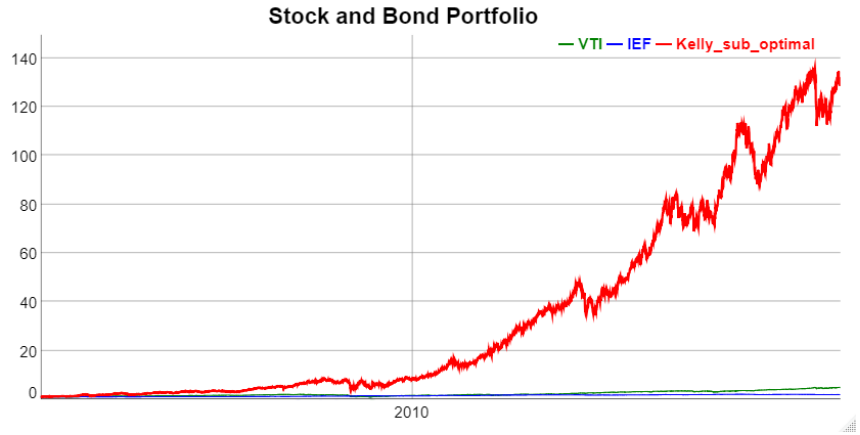
The Kelly optimal weights can be calculated approximately by first calculating the individual stock and bond weights, and then multiplying them by the Kelly weight of the combined portfolio.

```
> # Approximate Kelly weights
> retsport <- (retp %*% weightv)
> drop(mean(retsport)/var(retsport))*weightv
> # Exact Kelly weights
> optiml$par
```

Kelly Optimal Stock and Bond Portfolio

In practice, the Kelly optimal weights under logarithmic utility are too aggressive and they require very active trading, so half-Kelly or even quarter-Kelly weights are used instead.

```
> # Quarter-Kelly sub-optimal weights
> weightv <- optiml$par/4
> # Plot Kelly optimal portfolio
> retp <- cbind(retp, weightv[1]*retp$VTI + weightv[2]*retp$IEF)
> colnames(retp)[3] <- "Kelly_sub_optimal"
> # Calculate the compounded wealth from returns
> wealthv <- cumprod(1 + retp)
> # Plot compounded wealth
> dygraphs::dygraph(wealthv, main="Stock and Bond Portfolio") %>%
+   dyOptions(colors=c("green", "blue", "green")) %>%
+   dySeries("Kelly_sub_optimal", color="red", strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```



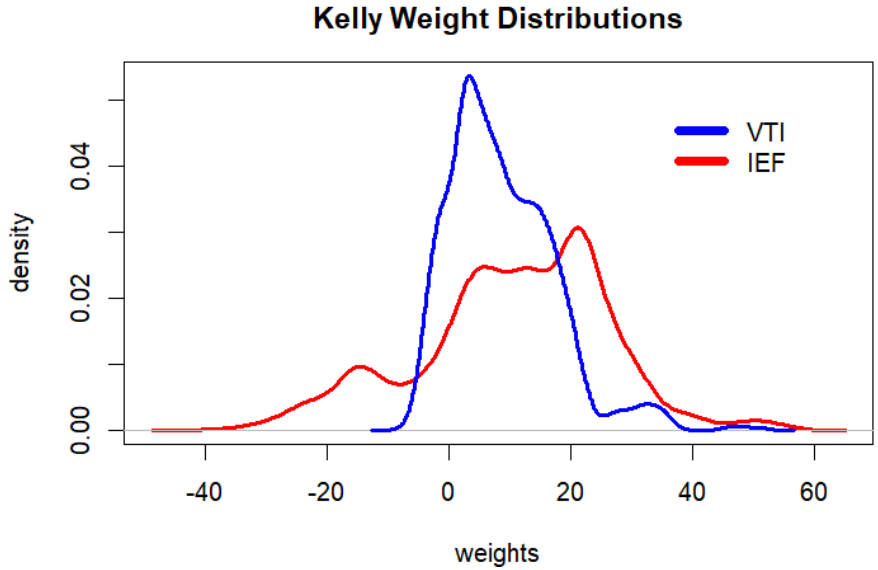
Rolling Kelly Weights

The Kelly weights k_f are calculated daily over a rolling look-back interval:

$$k_f = \frac{\bar{r}_t}{\sigma_t^2}$$

The distribution of the Kelly weights depends on the rolling returns $\bar{r}_t$ and variance σ_t^2.

```
> retp <- na.omit(rutils::etfenv$returns[, c("VTI", "IEF")])
> # Calculate the rolling returns and variance
> look_back <- 200
> var_rolling <- HighFreq::roll_var(retp, look_back)
> weightv <- HighFreq::roll_sum(retp, look_back)/look_back
> weightv <- weightv/var_rolling
> weightv[1, ] <- 1/NCOL(weightv)
> weightv <- zoo::na.locf(weightv)
> sum(is.na(weightv))
> range(weightv)
```

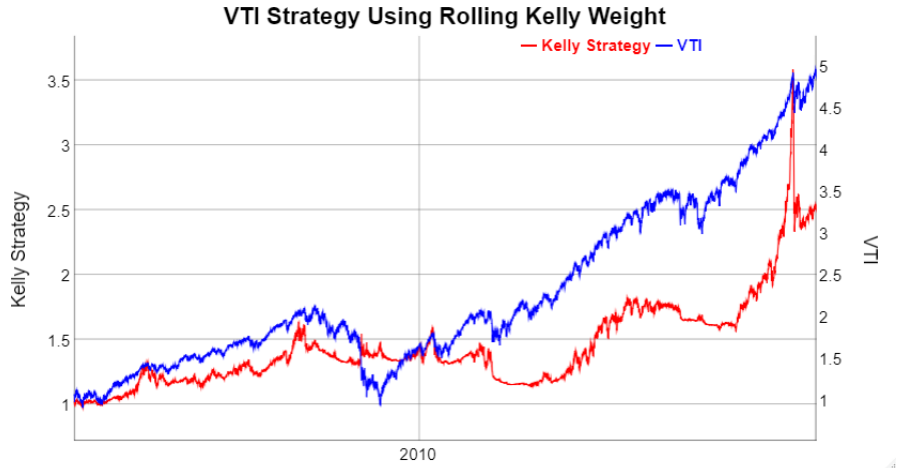



```
> # Plot the weights
> x11(width=6, height=5)
> par(mar=c(4, 4, 3, 1), oma=c(0, 0, 0, 0))
> plot(density(retp$IEF), t="l", lwd=3, col="red",
+      xlab="weights", ylab="density",
+      ylim=c(0, max(density(retp$VTI)$y)),
+      main="Kelly Weight Distributions")
> lines(density(retp$VTI), t="l", col="blue", lwd=3)
> legend("topright", legend=c("VTI", "IEF"),
+  inset=0.1, bg="white", lty=1, lwd=6, y.intersp=0.5,
+  col=c("blue", "red"), bty="n")
```

Rolling Kelly Strategy For Stocks

In the rolling Kelly strategy, the leverage of the risky asset k_f changes over time.

The leverage is equal to the updated weight from the previous period.




```
> # Scale and lag the Kelly weights
> weightv <- lapply(weightv, function(x) 10*x/sum(abs(range(x))))
> weightv <- do.call(cbind, weightv)
> weightv <- rutils::lagit(weightv)
> # Calculate the compounded Kelly wealth and VTI
> wealthv <- cbind(cumprod(1 + weightv$VTI*retp$VTI), cumprod(1 + re
> colnames(wealthv) <- c("Kelly Strategy", "VTI")
> dygraphs::dygraph(wealthv, main="VTI Strategy Using Rolling Kelly Weight") %>%
+   dyAxis("y", label="Kelly Strategy", independentTicks=TRUE) %>%
+   dyAxis("y2", label="VTI", independentTicks=TRUE) %>%
+   dySeries(name="Kelly Strategy", axis="y", label="Kelly Strategy", strokeWidth=1, col="red") %>%
+   dySeries(name="VTI", axis="y2", label="VTI", strokeWidth=1, col="blue")
```

Rolling Kelly Strategy With Transaction Costs

The *margin debt* m_t is proportional to the wealth w_t: $m_t = (k_f - 1)w_t + 1$.

The dollar amount of the *risky asset* traded is equal to the change in *margin*, equal to: $\Delta m_t = \Delta[(k_f - 1)w_t]$.

If the transaction costs are large, then they will reduce the wealth and reduce the dollar amount of the *risky asset* held by the investor.

The transaction costs depend on the change in wealth, and the wealth is decreased by the transaction costs.

So the transaction costs in each time period must be calculated recursively in a loop from the wealth in the past period.

If the transaction costs are much less than the change in wealth $c^r \ll |\Delta w_t|$, then they can be calculated approximately as the absolute value of the change in *margin* m_t^{nc} for a wealth path with no transaction costs:

$$c^r = \frac{\delta}{2} \left| \Delta m_t^{nc} \right|$$

The transaction costs as a percentage of wealth are equal to: c_t / w_t^{nc}, where w_t^{nc} is the wealth assuming no transaction costs.

The wealth of the Kelly Strategy after accounting for the *bid-ask spread* is then equal to:

$$w_t = \prod_{i=1}^{t} (1 + k_f \, r_t - \frac{\delta}{2} \frac{|\Delta m_i^{nc}|}{w_i^{nc}})$$

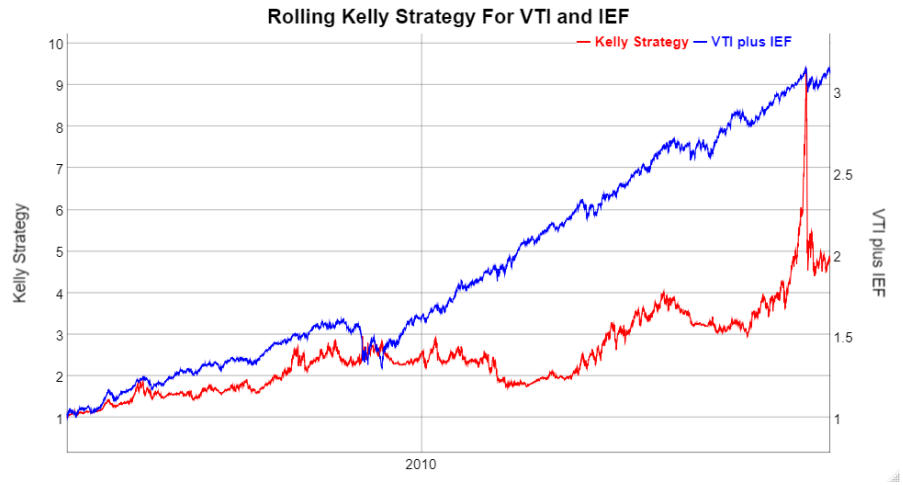
The effect of the *bid-ask spread* is to reduce the effective asset returns by an amount proportional to the *bid-ask spread*.

```
> # bidask equal to 1 bp for liquid ETFs
> bidask <- 0.001
> # Calculate the compounded Kelly wealth and margin
> wealthv <- cumprod(1 + weightv$VTI*retp$VTI)
> marginv <- (retp$VTI - 1)*wealthv + 1
> # Calculate the transaction costs
> costs <- bidask*drop(rutils::diffit(marginv))/2
> wealth_diff <- drop(rutils::diffit(wealthv))
> costs_rel <- ifelse(wealth_diff>0, costs/wealth_diff, 0)
> range(costs_rel)
> hist(costs_rel, breaks=10000, xlim=c(-0.02, 0.02))
> # Scale and lag the transaction costs
> costs <- rutils::lagit(abs(costs)/wealthv)
> # ReCalculate the compounded Kelly wealth
> wealth_trans <- cumprod(1 + retp$VTI*retp$VTI - costs)
> # Plot compounded wealth
> wealthv <- cbind(wealthv, wealth_trans)
> colnames(wealthv) <- c("Kelly", "Including bid-ask")
> dygraphs::dygraph(wealthv, main="Kelly Strategy With Transaction C
+   dyOptions(colors=c("green", "blue"), strokeWidth=2) %>%
```

Rolling Kelly Strategy For Stocks and Bonds

In the rolling Kelly strategy, the leverage of the risky asset k_f changes over time.

The leverage is equal to the updated weight from the previous period.




```
> # Calculate the compounded wealth from returns
> wealthv <- cumprod(1 + rowSums(weightv*retp))
> wealthv <- xts::xts(wealthv, zoo::index(retp))
> quantmod::chart_Series(wealthv, name="Rolling Kelly Strategy For V
> # Calculate the compounded Kelly wealth and VTI
> wealthv <- cbind(wealthv, cumprod(1 + 0.6*retp$IEF + 0.4*retp$VTI)
> colnames(wealthv) <- c("Kelly Strategy", "VTI plus IEF")
> dygraphs::dygraph(wealthv, main="Rolling Kelly Strategy For VTI and IEF") %>%
+   dyAxis("y", label="Kelly Strategy", independentTicks=TRUE) %>%
+   dyAxis("y2", label="VTI plus IEF", independentTicks=TRUE) %>%
+   dySeries(name="Kelly Strategy", axis="y", label="Kelly Strategy", strokeWidth=1, col="red") %>%
+   dySeries(name="VTI plus IEF", axis="y2", label="VTI plus IEF", strokeWidth=1, col="blue")
```

Package *PerformanceAnalytics* for Risk and Performance Analysis

The package *PerformanceAnalytics* contains functions for calculating risk and performance statistics, such as the variance, skewness, kurtosis, beta, alpha, etc.

The function `data()` loads external data or listv data sets in a package.

`managers` is an *xts* time series containing monthly percentage returns of six asset managers (HAM1 through HAM6), the EDHEC Long-Short Equity hedge fund index, the `S&P 500`, and US Treasury 10-year bond and 3-month bill total returns.

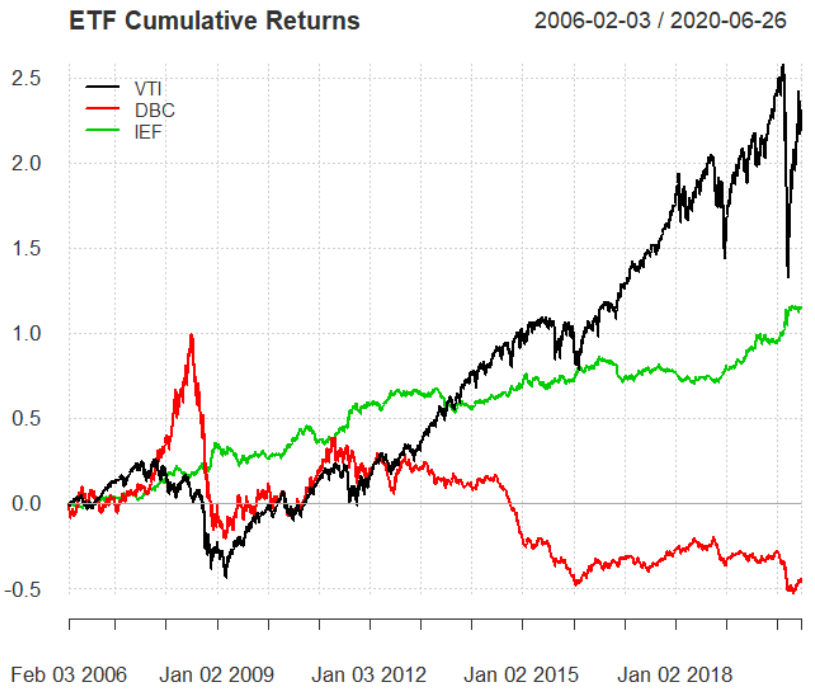
```
> # Load package PerformanceAnalytics
> library(PerformanceAnalytics)
> # Get documentation for package PerformanceAnalytics
> # Get short description
> packageDescription("PerformanceAnalytics")
> # Load help page
> help(package="PerformanceAnalytics")
> # List all objects in PerformanceAnalytics
> ls("package:PerformanceAnalytics")
> # List all datasets in PerformanceAnalytics
> data(package="PerformanceAnalytics")
> # Remove PerformanceAnalytics from search path
> detach("package:PerformanceAnalytics")
```

```
> perfstats <- unclass(data(
+     package="PerformanceAnalytics"))$results[, -(1:2)]
> apply(perfstats, 1, paste, collapse=" - ")
> # Load "managers" data set
> data(managers)
> class(managers)
> dim(managers)
> head(managers, 3)
```

Plots of Cumulative Returns

The function `chart.CumReturns()` from package *PerformanceAnalytics* plots the cumulative returns of a time series of returns.

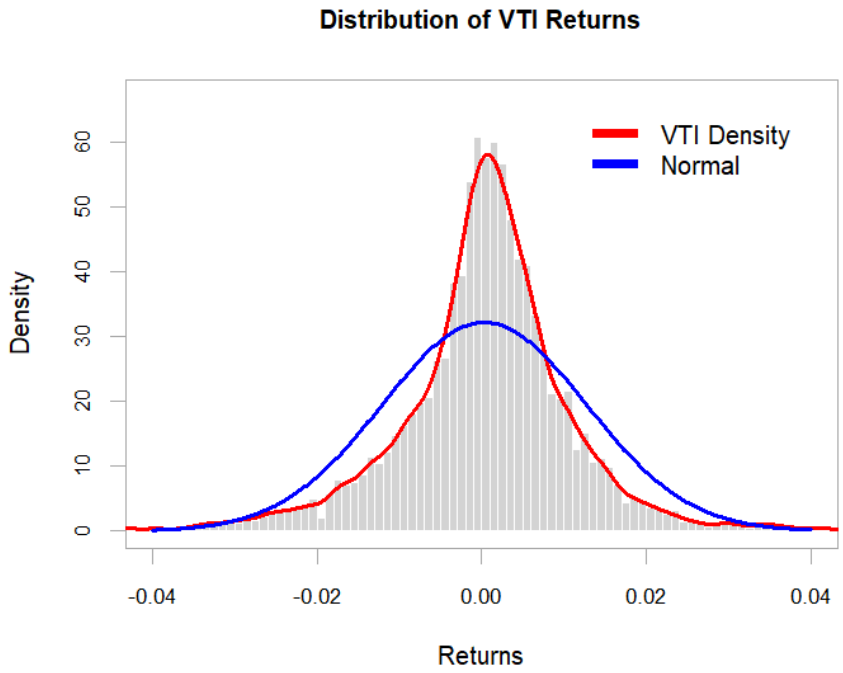
```
> # Load package "PerformanceAnalytics"
> library(PerformanceAnalytics)
> # Calculate ETF returns
> retp <- rutils::etfenv$returns[, c("VTI", "DBC", "IEF")]
> retp <- na.omit(retp)
> # Plot cumulative ETF returns
> x11(width=6, height=5)
> chart.CumReturns(retp, lwd=2, ylab="",
+   legend.loc="topleft", main="ETF Cumulative Returns")
```



The Distribution of Asset Returns

The function `chart.Histogram()` from package *PerformanceAnalytics* plots the histogram (frequency distribution) and the density of returns.

```
> retp <- na.omit(rutils::etfenv$returns$VTI)
> chart.Histogram(retp, xlim=c(-0.04, 0.04),
+   colorset = c("lightgray", "red", "blue"), lwd=3,
+   main=paste("Distribution of", colnames(retp), "Returns"),
+   methods = c("add.density", "add.normal"))
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+  leg=c("VTI Density", "Normal"),
+  lwd=6, lty=1, col=c("red", "blue"))
```



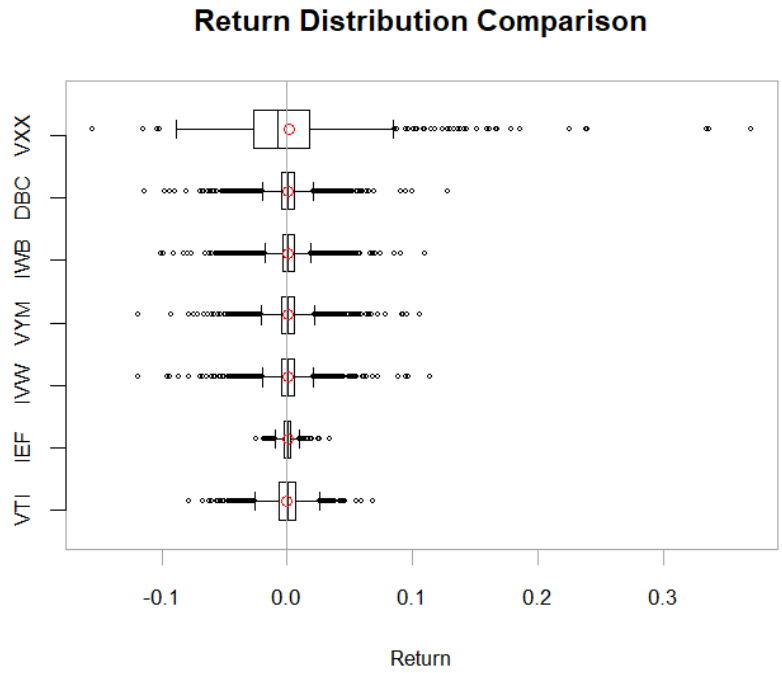
Boxplots of Returns

The function `chart.Boxplot()` from package *PerformanceAnalytics* plots a box-and-whisker plot for a distribution of returns.

The function `chart.Boxplot()` is a wrapper and calls the function `graphics::boxplot()` to plot the box plots.

A *box plot* (box-and-whisker plot) is a graphical display of a distribution of data:
The *box* represents the upper and lower quartiles,
The vertical lines (whiskers) represent values beyond the quartiles,
Open circles represent values beyond the nominal range (outliers).

```
> retp <- rutils::etfenv$returns[,
+   c("VTI", "IEF", "IVW", "VYM", "IWB", "DBC", "VXX")]
> x11(width=6, height=5)
> chart.Boxplot(names=FALSE, retp)
> par(cex.lab=0.8, cex.axis=0.8)
> axis(side=2, at=(1:NCOL(retp))/7.5-0.05,labels=colnames(retp))
```



The Median Absolute Deviation Estimator of Dispersion

The *Median Absolute Deviation* (*MAD*) is a nonparametric measure of dispersion (variability), defined using the median instead of the mean:

$$\text{MAD} = \text{median}(\text{abs}(x_i - \text{median}(\mathbf{x})))$$

The advantage of *MAD* is that it's always well defined, even for data that has infinite variance.

The *MAD* for normally distributed data is equal to $\Phi^{-1}(0.75) \cdot \hat{\sigma} = 0.6745 \cdot \hat{\sigma}$.

The function `mad()` calculates the *MAD* and divides it by $\Phi^{-1}(0.75)$ to make it comparable to the standard deviation.

For normally distributed data the *MAD* has a larger standard error than the standard deviation.

```
> # Simulate normally distributed data
> nrows <- 1000
> datav <- rnorm(nrows)
> sd(datav)
> mad(datav)
> median(abs(datav - median(datav)))
> median(abs(datav - median(datav)))/qnorm(0.75)
> # Bootstrap of sd and mad estimators
> bootd <- sapply(1:10000, function(x) {
+   samplev <- datav[sample.int(nrows, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ })  # end sapply
> bootd <- t(bootd)
> # Analyze bootstrapped variance
> head(bootd)
> sum(is.na(bootd))
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster
> bootd <- parLapply(cluster, 1:10000,
+   function(x, datav) {
+     samplev <- datav[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, datav=datav)  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:10000, function(x) {
+     samplev <- datav[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster
> bootd <- rutils::do_call(rbind, bootd)
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
```

The Median Absolute Deviation of Asset Returns

For normally distributed data the *MAD* has a larger standard error than the standard deviation.

But for distributions with fat tails (like asset returns), the standard deviation has a larger standard error than the *MAD*.

The *bootstrap* procedure performs a loop, which naturally lends itself to parallel computing.

The function `makeCluster()` starts running `R` processes on several CPU cores under *Windows*.

The function `parLapply()` is similar to `lapply()`, and performs loops under *Windows* using parallel computing on several CPU cores.

The `R` processes started by `makeCluster()` don't inherit any data from the parent `R` process.

Therefore the required data must be either passed into `parLapply()` via the dots `"..."` argument, or by calling the function `clusterExport()`.

The function `mclapply()` performs loops using parallel computing on several CPU cores under *Mac-OSX* or *Linux*.

The function `stopCluster()` stops the `R` processes running on several CPU cores.

```
> # Calculate VTI returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> sd(retp)
> mad(retp)
> # Bootstrap of sd and mad estimators
> bootd <- sapply(1:10000, function(x) {
+   samplev <- retp[sample.int(nrows, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ })  # end sapply
> bootd <- t(bootd)
> # Means and standard errors from bootstrap
> 100*apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster
> clusterExport(cluster, c("nrows", "returns"))
> bootd <- parLapply(cluster, 1:10000,
+   function(x) {
+     samplev <- retp[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   })  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:10000, function(x) {
+     samplev <- retp[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster
> bootd <- rutils::do_call(rbind, bootd)
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
```

The Downside Deviation of Asset Returns

Some investors argue that positive returns don't represent risk, only those returns less than the target rate of return r_t.

The *Downside Deviation* (semi-deviation) σ_d is equal to the standard deviation of returns less than the target rate of return r_t:

$$\sigma_d = \sqrt{\frac{1}{n}\sum_{i=1}^{n}([r_i - r_t]_-)^2}$$

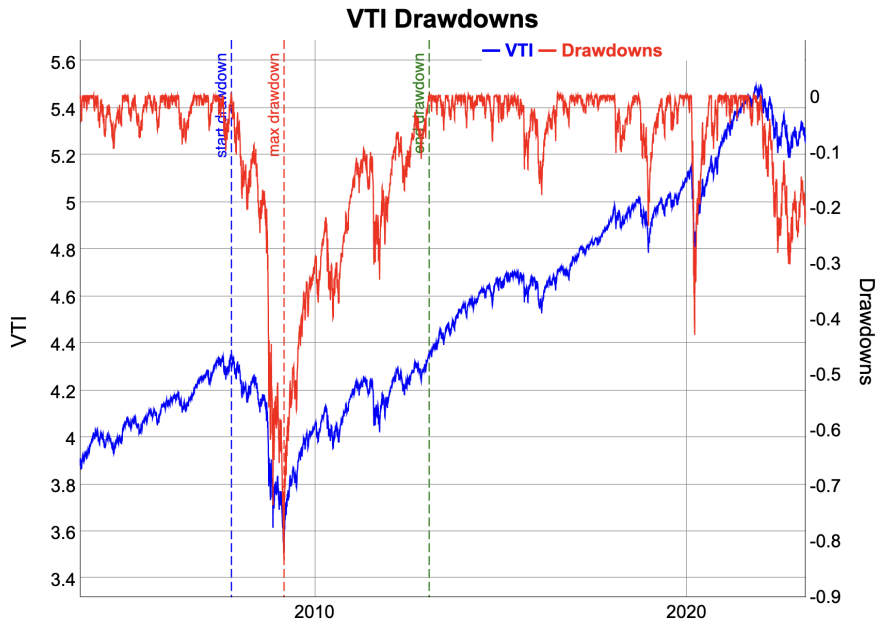
The function `DownsideDeviation()` from package *PerformanceAnalytics* calculates the downside deviation, for either the full time series (`method="full"`) or only for the subseries less than the target rate of return r_t (`method="subset"`).

```
> library(PerformanceAnalytics)
> # Define target rate of return of 50 bps
> targetr <- 0.005
> # Calculate the full downside returns
> retsub <- (retp - targetr)
> retsub <- ifelse(retsub < 0, retsub, 0)
> nrows <- NROW(retsub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(retsub^2)/nrows),
+   drop(DownsideDeviation(retp, MAR=targetr, method="full")))
> # Calculate the subset downside returns
> retsub <- (retp - targetr)
> retsub <- retsub[retsub < 0]
> nrows <- NROW(retsub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(retsub^2)/nrows),
+   drop(DownsideDeviation(retp, MAR=targetr, method="subset")))
```

Drawdown Risk

A *drawdown* is the drop in prices from their historical peak, and is equal to the difference between the prices minus the cumulative maximum of the prices.

Drawdown risk determines the risk of liquidation due to stop loss limits.




```
> # Calculate time series of VTI drawdowns
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> drawdns <- (closep - cummax(closep))
> # Extract the date index from the time series closep
> datev <- zoo::index(closep)
> # Calculate the maximum drawdown date and depth
> indexmin <- which.min(drawdns)
> datemin <- datev[indexmin]
> maxdd <- drawdns[datemin]
> # Calculate the drawdown start and end dates
> startd <- max(datev[(datev < datemin) & (drawdns == 0)])
> endd <- min(datev[(datev > datemin) & (drawdns == 0)])
> # dygraph plot of VTI drawdowns
> datav <- cbind(closep, drawdns)
> colnamev <- c("VTI", "Drawdowns")
> colnames(datav) <- colnamev
> dygraphs::dygraph(datav, main="VTI Drawdowns") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2],
+    valueRange=(1.2*range(drawdns)+0.1), independentTicks=TRUE) %>
+   dySeries(name=colnamev[1], axis="y", col="blue") %>%
+   dySeries(name=colnamev[2], axis="y2", col="red") %>%
+   dyEvent(startd, "start drawdown", col="blue") %>%
+   dyEvent(datemin, "max drawdown", col="red") %>%
+   dyEvent(endd, "end drawdown", col="green")
```

```
> # Plot VTI drawdowns using package quantmod
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> x11(width=6, height=5)
> quantmod::chart_Series(x=closep, name="VTI Drawdowns", theme=plot_
> xval <- match(startd, datev)
> yval <- max(closep)
> abline(v=xval, col="blue")
> text(x=xval, y=0.95*yval, "start drawdown", col="blue", cex=0.9)
> xval <- match(datemin, datev)
> abline(v=xval, col="red")
> text(x=xval, y=0.9*yval, "max drawdown", col="red", cex=0.9)
> xval <- match(endd, datev)
> abline(v=xval, col="green")
> text(x=xval, y=0.85*yval, "end drawdown", col="green", cex=0.9)
```

Drawdown Risk Using PerformanceAnalytics::table.Drawdowns()

The function table.Drawdowns() from package *PerformanceAnalytics* calculates a data frame of drawdowns.

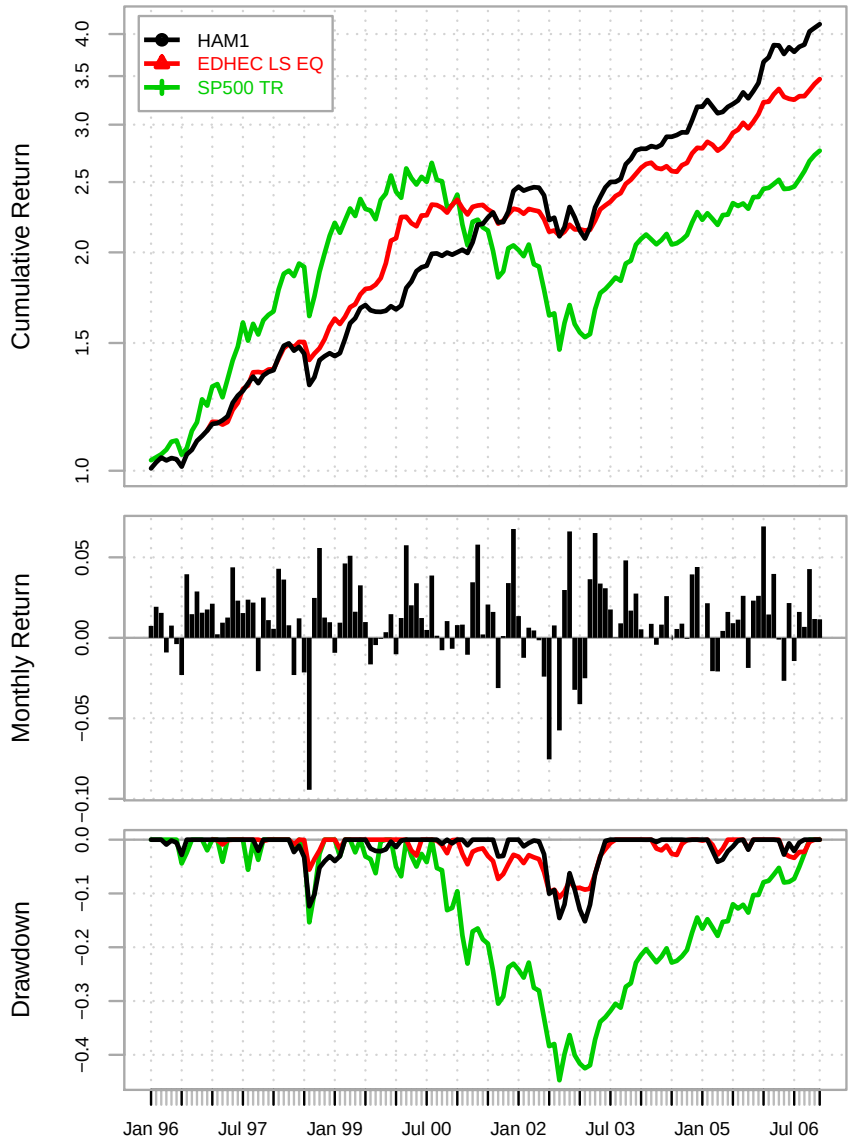
```
> library(xtable)
> library(PerformanceAnalytics)
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> retp <- rutils::diffit(closep)
> # Calculate table of VTI drawdowns
> tablev <- PerformanceAnalytics::table.Drawdowns(retp, geometric=FALSE)
> # Convert dates to strings
> tablev <- cbind(sapply(tablev[, 1:3], as.character), tablev[, 4:7])
> # Print table of VTI drawdowns
> print(xtable(tablev), comment=FALSE, size="tiny", include.rownames=FALSE)
```

From	Trough	To	Depth	Length	To Trough	Recovery
2007-10-10	2009-03-09	2012-03-13	-0.57	1115.00	355.00	760.00
2001-06-06	2002-10-09	2004-11-04	-0.45	858.00	336.00	522.00
2020-02-20	2020-03-23	2020-08-12	-0.18	122.00	23.00	99.00
2022-01-04	2022-10-12		-0.10	473.00	195.00	
2018-09-21	2018-12-24	2019-04-23	-0.10	146.00	65.00	81.00

PerformanceSummary Plots

The function `charts.PerformanceSummary()` from package *PerformanceAnalytics* plots three charts: cumulative returns, return bars, and drawdowns, for time series of returns.

```
> data(managers)
> charts.PerformanceSummary(ham1,
+   main="", lwd=2, ylog=TRUE)
```



The Loss Distribution of Asset Returns

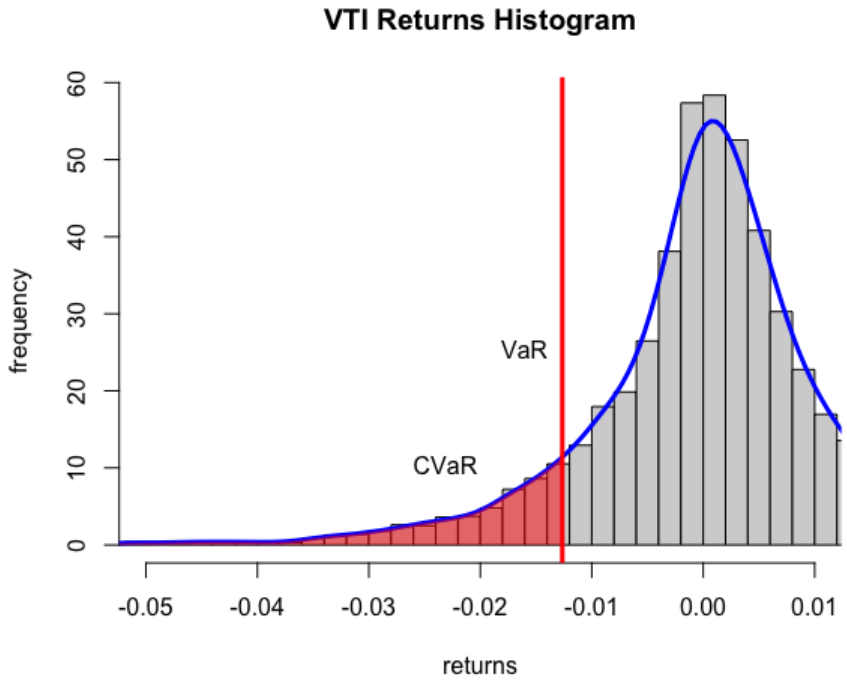
The distribution of returns has a long left tail of negative returns representing the risk of loss.

The *Value at Risk* (VaR) is equal to the quantile of returns corresponding to a given confidence level α.

The *Conditional Value at Risk* (CVaR) is equal to the average of negative returns less than the VaR.

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

The function `density()` calculates a kernel estimate of the probability density for a sample of data.




```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> confl <- 0.1
> varisk <- quantile(retp, confl)
> cvar <- mean(retp[retp <= varisk])
> # Plot histogram of VTI returns
> x11(width=6, height=5)
> par(mar=c(3, 2, 1, 0), oma=c(0, 0, 0, 0))
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", ylab="frequency", breaks=100,
+   xlim=c(-0.05, 0.01), freq=FALSE, main="VTI Returns Histogram")
> # Calculate density
> densv <- density(retp, adjust=1.5)
```

```
> # Plot density
> lines(densv, lwd=3, col="blue")
> # Plot line for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk, y=25, labels="VaR", lwd=2, pos=2)
> # Plot polygon shading for CVaR
> text(x=1.5*varisk, y=10, labels="CVaR", lwd=2, pos=2)
> varmax <- -0.06
> rangev <- (densv$x < varisk) &  (densv$x > varmax)
> polygon(c(varmax, densv$x[rangev], varisk),
+   c(0, densv$y[rangev], 0), col=rgb(1, 0, 0,0.5), border=NA)
```

Value at Risk (*VaR*)

The *Value at Risk* (VaR) is equal to the quantile of returns corresponding to a given confidence level α:

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

Where $\mathrm{f}(r)$ is the probability density (distribution) of returns.

At a high confidence level, the value of VaR is subject to estimation error, and various numerical methods are used to approximate it.

The function `quantile()` calculates the sample quantiles. It uses interpolation to improve the accuracy. Information about the different interpolation methods can be found by typing `?quantile`.

A simpler but less accurate way of calculating the quantile is by sorting and selecting the data closest to the quantile.

The function `VaR()` from package *PerformanceAnalytics* calculates the *Value at Risk* using several different methods.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> confl <- 0.05
> # Calculate VaR approximately by sorting
> sortv <- sort(as.numeric(retp))
> cutoff <- round(confl*nrows)
> varisk <- sortv[cutoff]
> # Calculate VaR as quantile
> varisk <- quantile(retp, probs=confl)
> # PerformanceAnalytics VaR
> PerformanceAnalytics::VaR(retp, p=(1-confl), method="historical")
> all.equal(unname(varisk),
+   as.numeric(PerformanceAnalytics::VaR(retp,
+   p=(1-confl), method="historical")))
```

Conditional Value at Risk (*CVaR*)

The *Conditional Value at Risk* (CVaR) is equal to the average of negative returns less than the VaR:

$$\mathrm{CVaR} = \frac{1}{\alpha} \int_0^{\alpha} \mathrm{VaR}(p)\, \mathrm{d}p$$

The *Conditional Value at Risk* is also called the *Expected Shortfall* (*ES*), or the Expected Tail Loss (*ETL*).

The function `ETL()` from package *PerformanceAnalytics* calculates the *Conditional Value at Risk* using several different methods.

```
> # Calculate VaR as quantile
> varisk <- quantile(retp, confl)
> # Calculate CVaR as expected loss
> cvar <- mean(retp[retp <= varisk])
> # PerformanceAnalytics VaR
> PerformanceAnalytics::ETL(retp, p=(1-confl), method="historical")
> all.equal(unname(cvar),
+   as.numeric(PerformanceAnalytics::ETL(retp,
+     p=(1-confl), method="historical")))
```

Risk and Return Statistics

The function `table.Stats()` from package *PerformanceAnalytics* calculates a data frame of risk and return statistics of the return distributions.

```
> # Calculate the risk-return statistics
> riskstats <-
+   PerformanceAnalytics::table.Stats(rutils::etfenv$returns)
> class(riskstats)
> # Transpose the data frame
> riskstats <- as.data.frame(t(riskstats))
> # Add Name column
> riskstats$Name <- rownames(riskstats)
> # Add Sharpe ratio column
> riskstats$"Arithmetic Mean" <-
+   sapply(rutils::etfenv$returns, mean, na.rm=TRUE)
> riskstats$Sharpe <-
+   sqrt(252)*riskstats$"Arithmetic Mean"/riskstats$Stdev
> # Sort on Sharpe ratio
> riskstats <- riskstats[order(riskstats$Sharpe, decreasing=TRUE),
```

	Sharpe	Skewness	Kurtosis
USMV	0.779	-0.857	21.21
QUAL	0.650	-0.509	12.74
MTUM	0.593	-0.675	11.85
IEF	0.472	0.056	2.59
VLUE	0.444	-0.950	17.06
XLV	0.435	0.071	10.06
GLD	0.425	-0.306	6.16
VTV	0.407	-0.659	13.69
VTI	0.406	-0.375	10.65
XLP	0.392	-0.120	8.67
VYM	0.386	-0.672	14.48
XLY	0.382	-0.356	6.56
XLI	0.366	-0.375	7.48
IWB	0.354	-0.385	9.91
IWD	0.336	-0.483	12.54
IVW	0.335	-0.296	8.33
XLU	0.330	0.001	11.82
IVE	0.325	-0.475	10.01
QQQ	0.321	-0.025	6.38
XLB	0.321	-0.366	5.28
XLK	0.313	0.074	6.60
IWF	0.304	-0.650	30.46
EEM	0.283	0.025	15.51
XLE	0.263	-0.529	12.41
TLT	0.260	-0.012	3.59
AIEQ	0.231	-0.701	6.95
VNQ	0.227	-0.531	17.90
SVXY	0.166	-18.142	656.67
XLF	0.153	-0.121	14.04
VEU	0.134	-0.501	11.60
DBC	0.026	-0.493	3.28
USO	-0.308	-1.139	14.12
VXX	-1.191	1.109	5.08

Investor Risk and Return Preferences

Investors typically prefer larger *odd moments* of the return distribution (mean, skewness), and smaller *even moments* (variance, kurtosis).

But positive skewness is often associated with lower returns, which can be observed in the *VIX* volatility ETFs, *VXX* and *SVXY*.

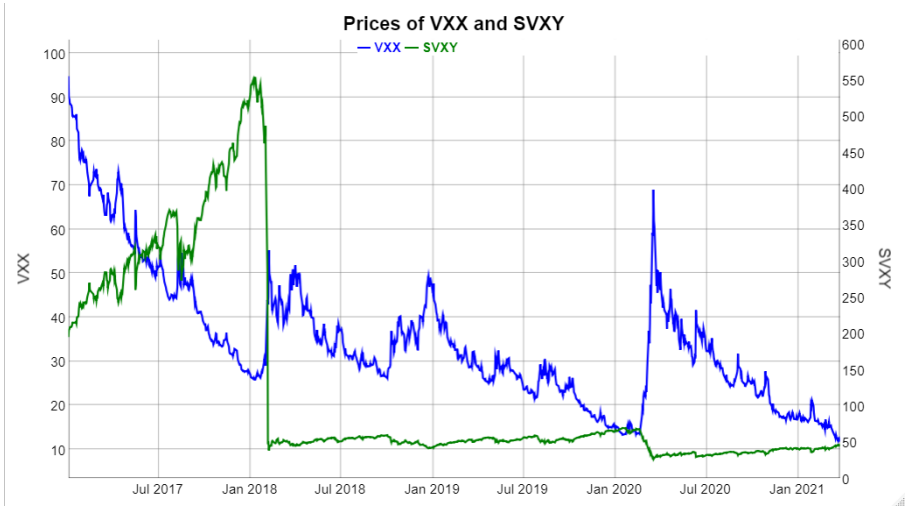
The *VXX* ETF is long the *VIX* index (effectively long an option), so it has positive skewness and small kurtosis, but negative returns (it's short market risk).

Since the *VXX* is effectively long an option, it pays option premiums so it has negative returns most of the time, with isolated periods of positive returns when markets drop.

The *SVXY* ETF is short the *VIX* index, so it has negative skewness and large kurtosis, but positive returns (it's long market risk).

Since the *SVXY* is effectively short an option, it earns option premiums so it has positive returns most of the time, but it suffers sharp losses when markets drop.

	Sharpe	Skewness	Kurtosis
VXX	-1.191	1.11	5.08
SVXY	0.166	-18.14	656.67




```
> # dygraph plot of VXX versus SVXY
> pricev <- na.omit(rutils::etfenv$prices[, c("VXX", "SVXY")])
> pricev <- pricev["2017/"]
> colnamev <- c("VXX", "SVXY")
> colnames(pricev) <- colnamev
> dygraphs::dygraph(pricev, main="Prices of VXX and SVXY") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", strokeWidth=2, col="blue")
+   dySeries(name=colnamev[2], axis="y2", strokeWidth=2, col="green"
+   dyLegend(show="always", width=300) %>% dyLegend(show="always",
+   dyLegend(show="always", width=300)
```

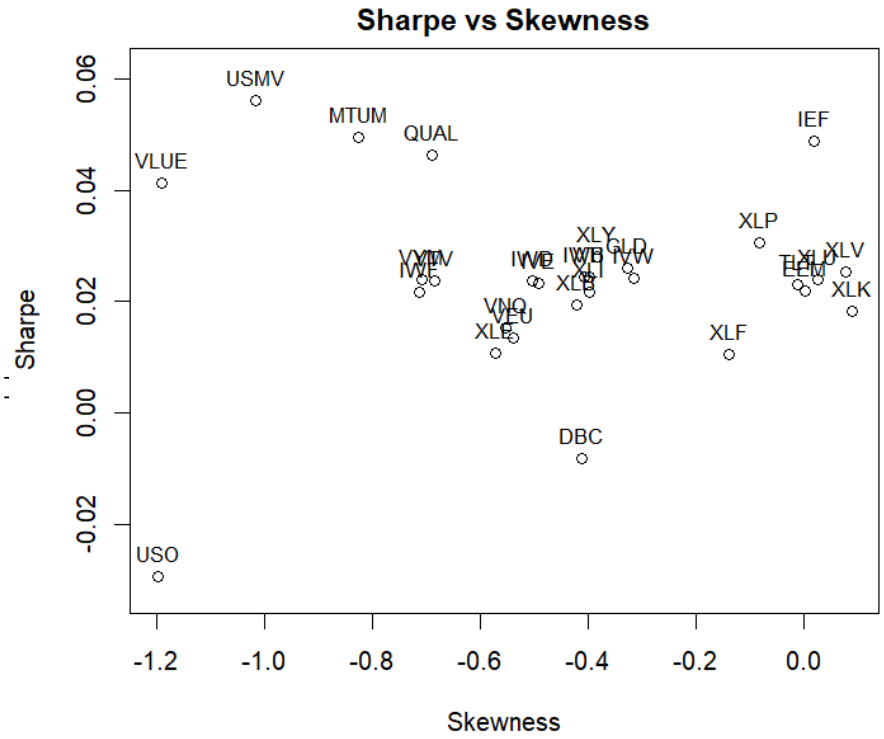
Skewness and Return Tradeoff

Similarly to the *VXX* and *SVXY*, for most other ETFs positive skewness is often associated with lower returns.

Some of the exceptions are bond ETFs (like *IEF*), which have both non-negative skewness and positive returns.

Another exception are commodity ETFs (like *USO* oil), which have both negative skewness and negative returns.

```
> # Remove VIX volatility ETF data
> riskstats <- riskstats[-match(c("VXX", "SVXY"), riskstats$Name),
> # Plot scatterplot of Sharpe vs Skewness
> plot(Sharpe ~ Skewness, data=riskstats,
+      ylim=1.1*range(riskstats$Sharpe),
+      main="Sharpe vs Skewness")
> # Add labels
> text(x=riskstats$Skewness, y=riskstats$Sharpe,
+     labels=riskstats$Name, pos=3, cex=0.8)
> # Plot scatterplot of Kurtosis vs Skewness
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 1), oma=c(0, 0, 0, 0))
> plot(Kurtosis ~ Skewness, data=riskstats,
+      ylim=c(1, max(riskstats$Kurtosis)),
+      main="Kurtosis vs Skewness")
> # Add labels
> text(x=riskstats$Skewness, y=riskstats$Kurtosis,
+     labels=riskstats$Name, pos=1, cex=0.5)
```



draft: Skewness and Return Tradeoff for ETFs and Stocks

The ETFs or stocks can be sorted on their skewness to create high_skew and low_skew cohorts.

But the high_skew cohort has better returns than the low_skew cohort - contrary to the thesis that assets with positive skewness produce lower returns than those with a negative skewness.

The low and high volatility cohorts have very similar returns, contrary to expectations. So do the low and high kurtosis cohorts.

```
> ### Below is for ETFs
> # Sort on Sharpe ratio
> riskstats <- riskstats[order(riskstats$Skewness, decreasing=TRUE),
> # Select high skew and low skew ETFs
> cutoff <- (NROW(riskstats) %/% 2)
> high_skew <- riskstats$Name[1:cutoff]
> low_skew <- riskstats$Name[(cutoff+1):NROW(riskstats)]
> # Calculate returns and log prices
> retp <- rutils::etfenv$returns
> retp <- zoo::na.locf(retp, na.rm=FALSE)
> retp[is.na(retp)] <- 0
> sum(is.na(retp))
> high_skew <- rowMeans(retp[, high_skew])
> low_skew <- rowMeans(retp[, low_skew])
> wealthv <- cbind(high_skew, low_skew)
> wealthv <- xts::xts(wealthv, zoo::index(retp))
> wealthv <- cumsum(wealthv)
> # dygraph plot of high skew and low skew ETFs
> colnamev <- colnames(wealthv)
> dygraphs::dygraph(wealthv, main="Log Wealth of Low and High Skew E
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", strokeWidth=2, col="blue")
+   dySeries(name=colnamev[2], axis="y2", strokeWidth=2, col="green"
+   dyLegend(show="always", width=300)
>
> ### Below is for S&P500 constituent stocks
> # calc_mom() calculates the moments of returns
> calc_mom <- function(retp, moment=3) {
+   retp <- na.omit(retp)
+   sum(((retp - mean(retp))/sd(retp))^moment)/NROW(retp)
+ }  # end calc_mom
> # Calculate skew and kurtosis of VTI returns
> calc_mom(retp, moment=3)
> calc_mom(retp, moment=4)
> # Load the S&P500 constituent stock returns
```

Risk-adjusted Return Measures

The *Sharpe ratio* $\mathrm{S_r}$ is equal to the excess returns (in excess of the risk-free rate r_f) divided by the standard deviation σ of the returns:

$$\mathrm{S_r} = \frac{E[r - r_f]}{\sigma}$$

The *Sortino ratio* $\mathrm{So_r}$ is equal to the excess returns divided by the *downside deviation* σ_d (standard deviation of returns that are less than a target rate of return r_t):

$$\mathrm{So_r} = \frac{E[r - r_t]}{\sigma_d}$$

The *Calmar ratio* $\mathrm{C_r}$ is equal to the excess returns divided by the *maximum drawdown* DD of the returns:

$$\mathrm{C_r} = \frac{E[r - r_f]}{\mathrm{DD}}$$

The *Dowd ratio* $\mathrm{D_r}$ is equal to the excess returns divided by the *Value at Risk* (VaR) of the returns:

$$\mathrm{D_r} = \frac{E[r - r_f]}{\mathrm{VaR}}$$

The *Conditional Dowd ratio* $\mathrm{Dc_r}$ is equal to the excess returns divided by the *Conditional Value at Risk* (CVaR) of the returns:

$$\mathrm{Dc_r} = \frac{E[r - r_f]}{\mathrm{CVaR}}$$

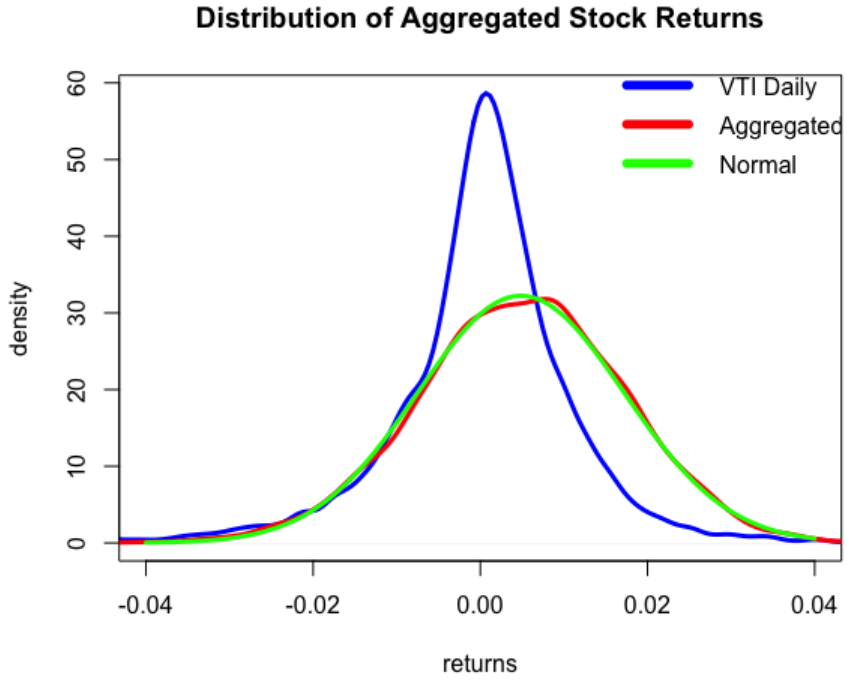
```
> library(PerformanceAnalytics)
> retp <- rutils::etfenv$returns[, c("VTI", "IEF")]
> retp <- na.omit(retp)
> # Calculate the Sharpe ratio
> confl <- 0.05
> PerformanceAnalytics::SharpeRatio(retp, p=(1-confl),
+   method="historical")
> # Calculate the Sortino ratio
> PerformanceAnalytics::SortinoRatio(retp)
> # Calculate the Calmar ratio
> PerformanceAnalytics::CalmarRatio(retp)
> # Calculate the Dowd ratio
> PerformanceAnalytics::SharpeRatio(retp, FUN="VaR",
+   p=(1-confl), method="historical")
> # Calculate the Dowd ratio from scratch
> varisk <- sapply(retp, quantile, probs=confl)
> -sapply(retp, mean)/varisk
> # Calculate the Conditional Dowd ratio
> PerformanceAnalytics::SharpeRatio(retp, FUN="ES",
+   p=(1-confl), method="historical")
> # Calculate the Conditional Dowd ratio from scratch
> cvar <- sapply(retp, function(x) {
+   mean(x[x < quantile(x, confl)])
+ })
> -sapply(retp, mean)/cvar
```

Risk of Aggregated Stock Returns

Stock returns aggregated over longer holding periods are closer to normally distributed, and their skewness, kurtosis, and tail risks are significantly lower than for daily returns.

Stocks become less risky over longer holding periods, so investors may choose to own a higher percentage of stocks, provided they hold them for a longer period of time.

```
> # Calculate VTI daily percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> # Bootstrap aggregated monthly VTI returns
> holdp <- 22
> reta <- sqrt(holdp)*sapply(1:nrows, function(x) {
+     mean(retp[sample.int(nrows, size=holdp, replace=TRUE)])
+ })  # end sapply
> # Calculate mean, standard deviation, skewness, and kurtosis
> datav <- cbind(retp, reta)
> colnames(datav) <- c("VTI", "Agg")
> sapply(datav, function(x) {
+   # Standardize the returns
+   meanv <- mean(x); stdev <- sd(x); x <- (x - meanv)/stdev
+   c(mean=meanv, stdev=stdev, skew=mean(x^3), kurt=mean(x^4))
+ })  # end sapply
> # Calculate the Sharpe and Dowd ratios
> confl <- 0.02
> ratiom <- sapply(datav, function(x) {
+   stdev <- sd(x)
+   varisk <- unname(quantile(x, probs=confl))
+   cvar <- mean(x[x < varisk])
+   mean(x)/c(Sharpe=stdev, Dowd=-varisk, DowdC=-cvar)
+ })  # end sapply
> # Annualize the daily risk
> ratiom[, 1] <- sqrt(22)*ratiom[, 1]
```




```
> # Plot the densities of returns
> plot(density(retp), t="l", lwd=3, col="blue",
+      xlab="returns", ylab="density", xlim=c(-0.04, 0.04),
+      main="Distribution of Aggregated Stock Returns")
> lines(density(reta), t="l", col="red", lwd=3)
> curve(expr=dnorm(x, mean=mean(reta), sd=sd(reta)), col="green", lw
> legend("topright", legend=c("VTI Daily", "Aggregated", "Normal"),
+  inset=-0.1, bg="white", lty=1, lwd=6, col=c("blue", "red", "green
```

draft: Feature Engineering

Feature engineering derives predictive data elements (features) from a large input data set.

Feature engineering reduces the size of the input data set to a smaller set of features with the highest predictive power.

The predictive features are then used as inputs into machine learning models.

Out-of-sample features only depend on past data, while *in-sample* features depend both on past and future data.

A *trailing* data filter is an example of an *out-of-sample* feature.

A *centered* data filter is an example of an *in-sample* feature.

Out-of-sample features are used in forecasting and scrubbing real-time (live) data.

In-sample features are used in data labeling and scrubbing historical data.

Principal Component Analysis (*PCA*) is a *dimension reduction* technique used in multivariate feature engineering.

Feature engineering can be developed using *domain knowledge* and analytical techniques.

Some features indicate trend, for example the moving

```
> # Number of flights from each airport
> dtable[, .N, by=origin]
> # Same, but add names to output
> dtable[, .(flights=.N), by=.(airport=origin)]
> # Number of AA flights from each airport
> dtable[carrier=="AA", .(flights=.N),
+       by=.(airport=origin)]
> # Number of flights from each airport and airline
> dtable[, .(flights=.N),
+       by=.(airport=origin, airline=carrier)]
> # Average aircraft_delay
> dtable[, mean(aircraft_delay)]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Average aircraft_delay from each airport
> dtable[, .(delay=mean(aircraft_delay)),
+       by=.(airport=origin)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+       by=.(airport=origin, month=month)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+       keyby=.(airport=origin, month=month)]
```

Convolution Filtering of Time Series

The function `filter()` applies a trailing linear filter to time series, vectors, and matrices, and returns a time series of class `"ts"`.

The function `filter()` with the argument `method="convolution"` calculates the *convolution* of the vector r_t with the filter φ_i:

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p}$$

Where f_t is the filtered output vector, and φ_i are the filter coefficients.

`filter()` is very fast because it calculates the filter by calling compiled C++ functions.

`filter()` with `method="convolution"` calls the function `stats:::C_cfilter()` to calculate the *convolution*.

Convolution filtering can be performed even faster by directly calling the compiled function `stats:::C_cfilter()`.

The function `HighFreq::roll_conv()` calculates the *weighted* trailing sum (convolution) even faster than `stats:::C_cfilter()`.

```
> # Extract log VTI prices
> ohlc <- log(rutils::etfenv$VTI)
> closep <- quantmod::Cl(ohlc)
> colnames(closep) <- "VTI"
> nrows <- NROW(closep)
> # Inspect the R code of the function filter()
> filter
> # Calculate EMA weights
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate convolution using filter()
> pricef <- filter(closep, filter=weightv, method="convolution", sid
> # filter() returns time series of class "ts"
> class(pricef)
> # Get information about C_cfilter()
> getAnywhere(C_cfilter)
> # Filter using C_cfilter() over past values (sides=1).
> priceff <- .Call(stats:::C_cfilter, closep, filter=weightv,
+                  sides=1, circular=FALSE)
> all.equal(as.numeric(pricef), priceff, check.attributes=FALSE)
> # Calculate EMA prices using HighFreq::roll_conv()
> pricecpp <- HighFreq::roll_conv(closep, weightv=weightv)
> all.equal(priceff[-(1:look_back)], as.numeric(pricecpp)[-(1:look_b
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   priceff=.Call(stats:::C_cfilter, closep, filter=weightv, sides=1
+   rcpp=HighFreq::roll_conv(closep, weightv=weightv)
+   ), times=10)[, c(1, 4, 5)]
```

Recursive Filtering of Time Series

The function `filter()` with `method="recursive"` calls the function `stats:::C_rfilter()` to calculate the *recursive filter* as follows:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p} + \xi_t$$

Where r_t is the filtered output vector, φ_i are the filter coefficients, and ξ_t are standard normal *innovations*.

The *recursive* filter describes an *AR(p)* process, which is a special case of an *ARIMA* process.

The function `HighFreq::sim_arima()` is very fast because it's written using the `C++` *Armadillo* numerical library.

```
> # Simulate AR process using filter()
> nrows <- NROW(closep)
> # Calculate AR coefficients and innovations
> coeff <- matrix(weightv)/4
> ncoeff <- NROW(coeff)
> innov <- matrix(rnorm(nrows))
> arimav <- filter(x=innov, filter=coeff, method="recursive")
> # Get information about C_rfilter()
> getAnywhere(C_rfilter)
> # Filter using C_rfilter() compiled C++ function directly
> arimafast <- .Call(stats:::C_rfilter, innov, coeff,
+                double(ncoeff + nrows))
> all.equal(as.numeric(arimav), arimafast[-(1:ncoeff)],
+     check.attributes=FALSE)
> # Filter using C++ code
> arimacpp <- HighFreq::sim_ar(coeff, innov)
> all.equal(arimafast[-(1:ncoeff)], drop(arimacpp))
> # Benchmark speed of the three methods
> summary(microbenchmark(
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   priceff=.Call(stats:::C_rfilter, innov, coeff, double(ncoeff + r
+   Rcpp=HighFreq::sim_ar(coeff, innov)
+   ), times=10)[, c(1, 4, 5)]
```

Data Smoothing and The Bias-Variance Tradeoff

Filtering through an averaging filter produces data *smoothing*.

Smoothing real-time data with a trailing filter reduces its *variance* but it increases its *bias* because it introduces a time lag.

Smoothing historical data with a centered filter reduces its *variance* but it introduces *data snooping*.

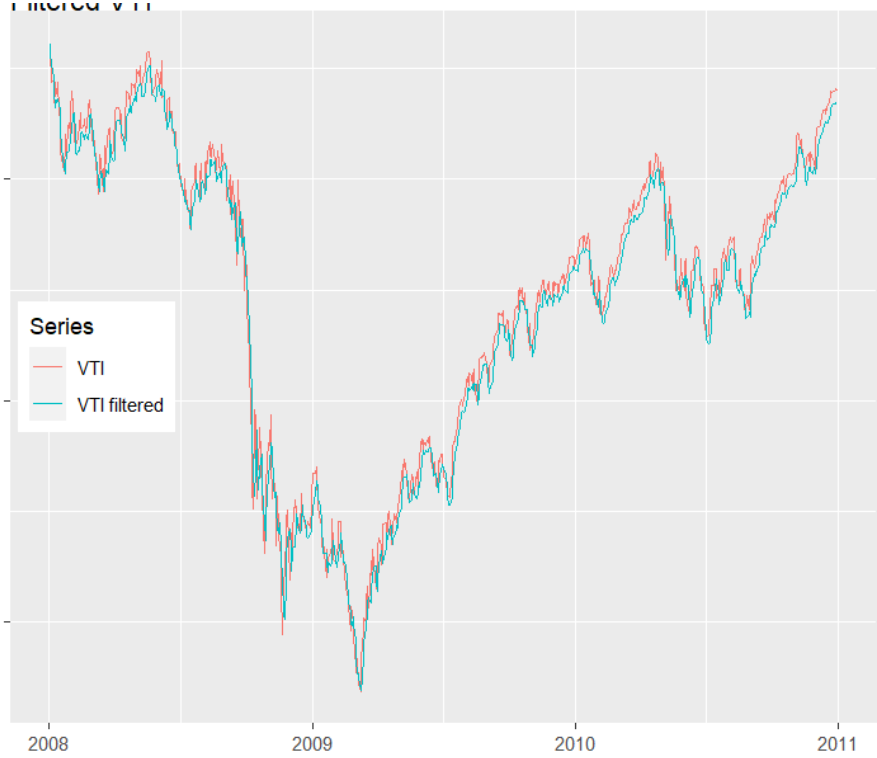
In engineering, smoothing is called a *low-pass filter*, since it eliminates high frequency signals, and it passes through low frequency signals.


```
> # Calculate trailing EMA prices using HighFreq::roll_conv()
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> pricef <- HighFreq::roll_conv(closep, weightv=weightv)
> # Combine prices with smoothed prices
> pricev <- cbind(closep, pricef)
> colnames(pricev)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diffit(pricev), sd)
> # Plot dygraph
> dygraphs::dygraph(pricev["2009"], main="VTI Prices and Trailing S
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

```
> # Center the smoothed prices
> pricef <- rutils::lagit(pricef, -(look_back %/% 2), pad_zeros=FALS
> # Combine prices with smoothed prices
> pricev <- cbind(closep, pricef)
> colnames(pricev)[2] <- "VTI Smooth"
> # Plot dygraph
> dygraphs::dygraph(pricev["2009"], main="VTI Prices and Centered Sm
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

depr: Plotting Filtered Time Series

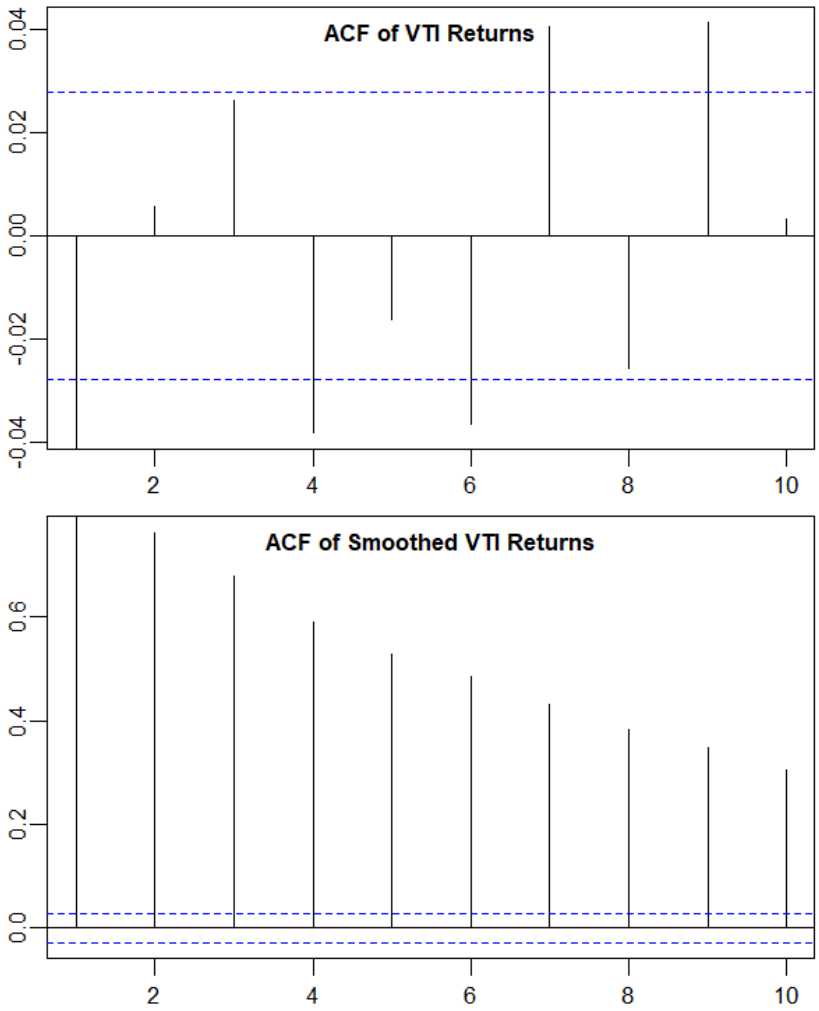
```
> library(rutils)  # Load package rutils
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> # Coerce to zoo and merge the time series
> pricef <- cbind(closep, pricef)
> colnames(pricef) <- c("VTI", "VTI filtered")
> # Plot ggplot2
> autoplot(pricef["2008/2010"],
+     main="Filtered VTI", facets=NULL) +  # end autoplot
+ xlab("") + ylab("") +
+ theme(  # Modify plot theme
+     legend.position=c(0.1, 0.5),
+     plot.title=element_text(vjust=-2.0),
+     plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+     plot.background=element_blank(),
+     axis.text.y=element_blank()
+     )  # end theme
> # end ggplot2
```



Autocorrelations of Smoothed Time Series

Smoothing a time series of prices produces autocorrelations of their returns.

```
> # Calculate VTI log returns
> retp <- rutils::diffit(closef)
> # Open plot window
> x11(width=6, height=7)
> # Set plot parameters
> par(oma=c(1, 1, 0, 1), mar=c(1, 1, 1, 1), mgp=c(0, 0.5, 0),
+     cex.lab=0.8, cex.axis=0.8, cex.main=0.8, cex.sub=0.5)
> # Set two plot panels
> par(mfrow=c(2,1))
> # Plot ACF of VTI returns
> rutils::plot_acf(retp[, 1], lag=10, xlab="")
> title(main="ACF of VTI Returns", line=-1)
> # Plot ACF of smoothed VTI returns
> rutils::plot_acf(retp[, 2], lag=10, xlab="")
> title(main="ACF of Smoothed VTI Returns", line=-1)
```



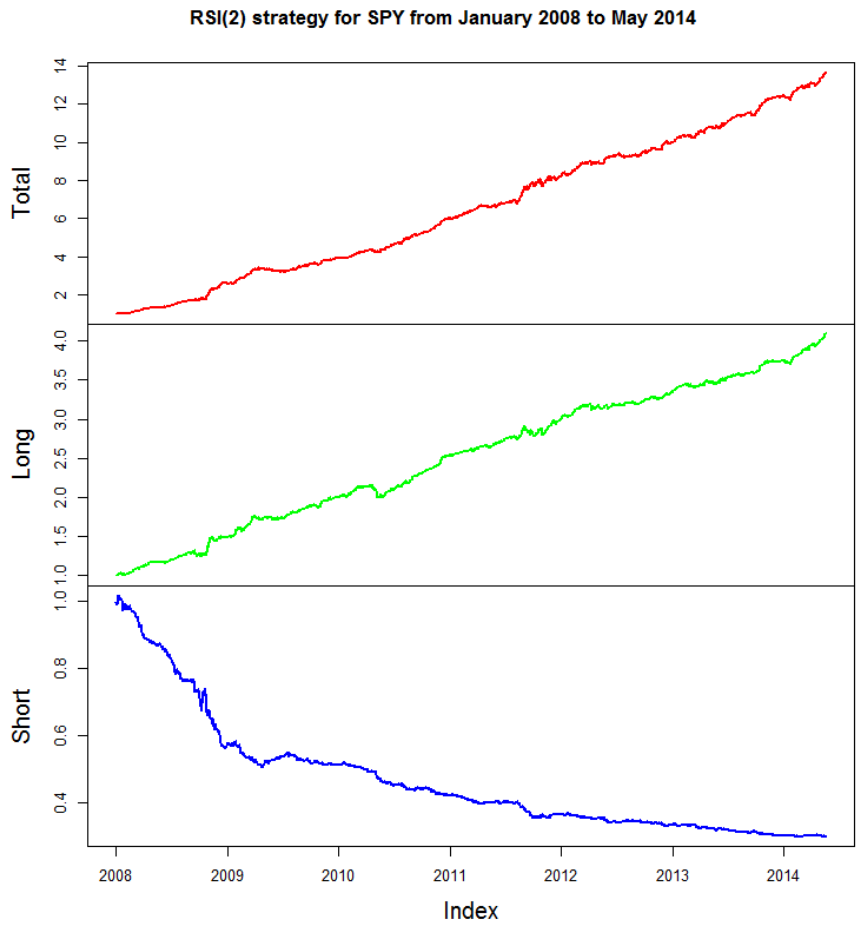
draft: *RSI* Price Technical Indicator

The *Relative Strength Index* (*RSI*) is defined as the weighted average of prices over a trailing interval:

$$p_t^{RSI} = (1 - \exp(-\lambda)) \sum_{j=0}^{\infty} \exp(-\lambda j) p_{t-j}$$

The decay parameter λ determines the rate of decay of the *RSI* weights, with larger values of λ producing faster decay, giving more weight to recent prices, and vice versa.

```
> # Get close prices and calculate close-to-close returns
> # closep <- quantmod::Cl(rutils::etfenv$VTI)
> closep <- quantmod::Cl(HighFreq::SPY)
> colnames(closep) <- rutils::get_name(colnames(closep))
> retspy <- TTR::ROC(closep)
> retspy[1] <- 0
> # Calculate the RSI indicator
> r_si <- TTR::RSI(closep, 2)
> # Calculate the long (up) and short (dn) signals
> sig_up <- ifelse(r_si < 10, 1, 0)
> sig_dn <- ifelse(r_si > 90, -1, 0)
> # Lag signals by one period
> sig_up <- rutils::lagit(sig_up, 1)
> sig_dn <- rutils::lagit(sig_dn, 1)
> # Replace NA signals with zero position
> sig_up[is.na(sig_up)] <- 0
> sig_dn[is.na(sig_dn)] <- 0
> # Combine up and down signals into one
> sig_nals <- sig_up + sig_dn
> # Calculate cumulative returns
> eq_up <- exp(cumsum(sig_up*retspy))
> eq_dn <- exp(cumsum(-1*sig_dn*retspy))
> eq_all <- exp(cumsum(sig_nals*retspy))
```




```
> # Plot daily cumulative returns in panels
> endd <- endpoints(retspy, on="days")
> plot.zoo(cbind(eq_all, eq_up, eq_dn)[endd], lwd=c(2, 2, 2),
+   ylab=c("Total","Long","Short"), col=c("red","green","blue"),
+   main=paste("RSI(2) strategy for", colnames(closep), "from",
+        format(start(retspy), "%B %Y"), "to",
+        format(end(retspy), "%B %Y")))
```

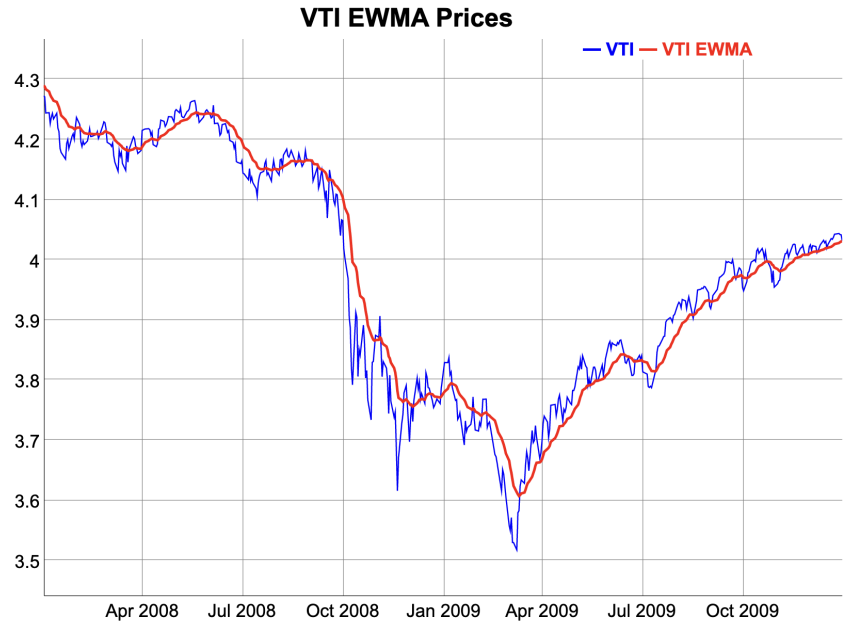
EMA Price Technical Indicator

The *Exponentially Weighted Moving Average Price* (*EMA*) is defined as the weighted average of prices over a trailing interval:

$$p_t^{EMA} = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j p_{t-j}$$

The decay parameter λ determines the rate of decay of the *EMA* weights, with smaller values of λ producing faster decay, giving more weight to recent prices, and vice versa.

The function `HighFreq::roll_wsum()` calculates the convolution of a time series with a vector of weights.




```
> # Extract log VTI prices
> ohlc <- rutils::etfenv$VTI
> datev <- zoo::index(ohlc)
> closep <- log(quantmod::Cl(ohlc))
> colnames(closep) <- "VTI"
> nrows <- NROW(closep)
> # Calculate EMA weights
> look_back <- 111
> lambdaf <- 0.9
> weightv <- lambdaf^(1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate EMA prices as the convolution
> emacpp <- HighFreq::roll_sumw(closep, weightv=weightv)
> pricev <- cbind(closep, emacpp)
> colnames(pricev) <- c("VTI", "VTI EMA")
```

```
> # Dygraphs plot with custom line colors
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main="VTI EMA Prices") %>%
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dyLegend(show="always", width=300)
> # Standard plot of  EMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> colorv <- c("blue", "red")
> plot_theme$col$line.col <- colors
> quantmod::chart_Series(pricev["2008/2009"], theme=plot_theme,
+        lwd=2, name="VTI EMA Prices")
> legend("topleft", legend=colnames(pricev), y.intersp=0.4,
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Recursive EMA Price Indicator

The *EMA* prices can be calculated recursively as follows:

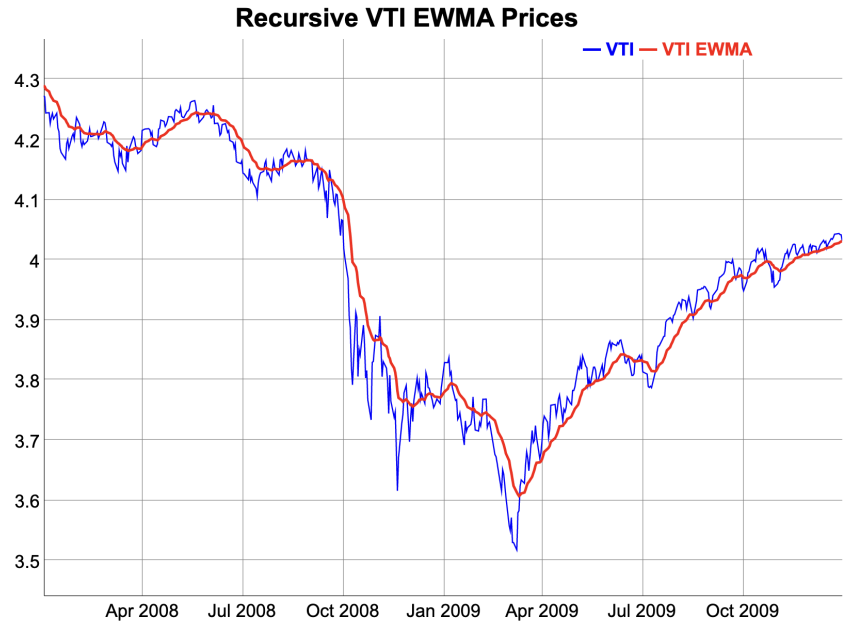
$$p_t^{EMA} = \lambda p_{t-1}^{EMA} + (1-\lambda)p_t$$

The decay parameter λ determines the rate of decay of the *EMA* weights, with smaller values of λ producing faster decay, giving more weight to recent prices, and vice versa.

The recursive *EMA* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

The compiled `C++` function `stats:::C_rfilter()` calculates the EMA prices recursively.

The function `HighFreq::run_mean()` calculates the EMA prices recursively using the `C++` *Armadillo* numerical library.




```
> # Calculate EMA prices recursively using C++ code
> emar <- .Call(stats:::C_rfilter, closep, lambdaf, c(as.numeric(cl
> # Or R code
> # emar <- filter(closep, filter=lambdaf, init=as.numeric(closep[1
> emar <- (1-lambdaf)*emar
> # Calculate EMA prices recursively using RcppArmadillo C++
> emacpp <- HighFreq::run_mean(closep, lambda=lambdaf)
> all.equal(drop(emacpp), emar)
> # Compare the speed of C++ code with RcppArmadillo C++
> library(microbenchmark)
> summary(microbenchmark(
+   filtercpp=HighFreq::run_mean(closep, lambda=lambdaf),
+   rfilter=.Call(stats:::C_rfilter, closep, lambdaf, c(as.numeric
+   times=10))[, c(1, 4, 5)]
```

```
> # Dygraphs plot with custom line colors
> pricev <- cbind(closep, emacpp)
> colnames(pricev) <- c("VTI", "VTI EMA")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main="Recursive VTI EMA Pri
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dyLegend(show="always", width=300)
> # Standard plot of  EMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> colorv <- c("blue", "red")
> plot_theme$col$line.col <- colors
> quantmod::chart_Series(pricev["2008/2009"], theme=plot_theme,
+        lwd=2, name="VTI EMA Prices")
> legend("topleft", legend=colnames(pricev),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Volume-Weighted Average Price Indicator

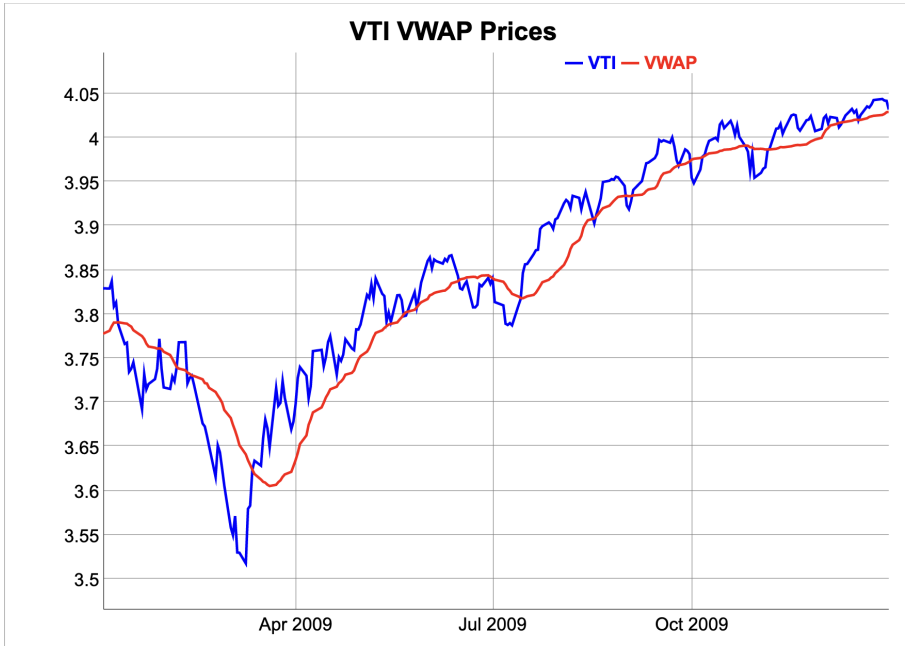
The Volume-Weighted Average Price (*VWAP*) is defined as the sum of prices multiplied by trading volumes, divided by the sum of volumes:

$$p_t^{VWAP} = \frac{\sum_{j=0}^{n} v_{t-j} p_{t-j}}{\sum_{j=0}^{n} v_{t-j}}$$

The *VWAP* applies more weight to prices with higher trading volumes, which allows it to react more quickly to recent market volatility.

The drawback of the *VWAP* indicator is that it applies large weights to prices far in the past.

The *VWAP* is often used as a technical indicator in trend following strategies.




```
> # Calculate log OHLC prices and volumes
> volumv <- quantmod::Vo(ohlc)
> colnames(volumv) <- "Volume"
> nrows <- NROW(closep)
> # Calculate the VWAP prices
> look_back <- 21
> vwap <- HighFreq::roll_sum(closep, look_back=look_back, weightv=v
> colnames(vwap) <- "VWAP"
> pricev <- cbind(closep, vwap)
```

```
> # Dygraphs plot with custom line colors
> colorv <- c("blue", "red")
> dygraphs::dygraph(pricev["2008/2009"], main="VTI VWAP Prices") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
> # Plot VWAP prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colors
> quantmod::chart_Series(pricev["2008/2009"], theme=plot_theme,
+        lwd=2, name="VTI VWAP Prices")
> legend("bottomright", legend=colnames(pricev),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Recursive VWAP Price Indicator

The *VWAP* prices p^{VWAP} can also be calculated recursively as the ratio of the mean volume weighted prices $\bar{vp}$ divided by the mean trading volumes $\bar{v}$:

$$\bar{v}_t = \lambda \bar{v}_{t-1} + (1-\lambda) v_t$$
$$\bar{vp}_t = \lambda \bar{vp}_{t-1} + (1-\lambda) v_t p_t$$
$$p^{VWAP} = \frac{\bar{vp}}{\bar{v}}$$

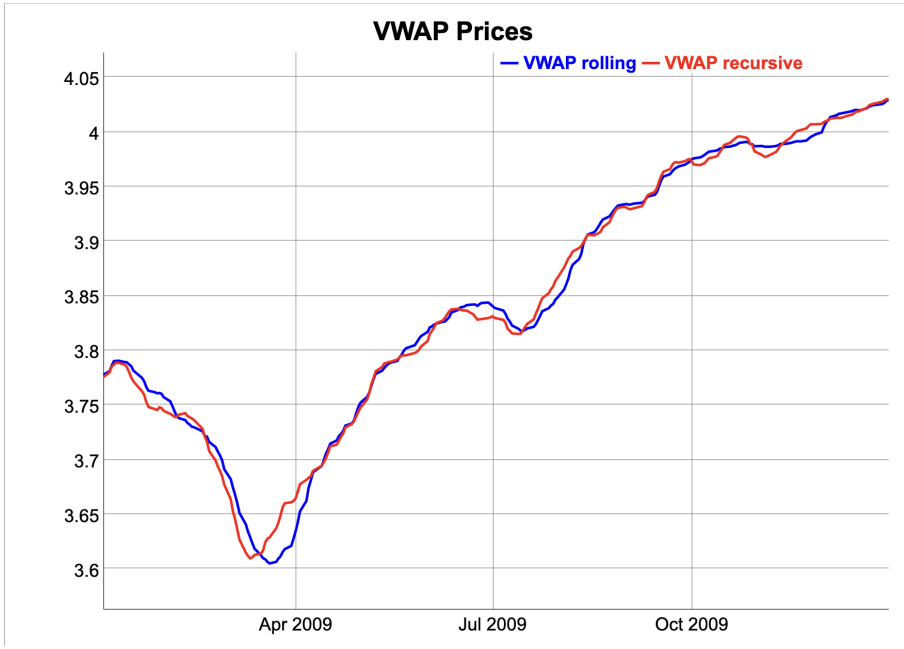
The recursive *VWAP* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

The advantage of the recursive *VWAP* indicator is that it gradually "forgets" about large trading volumes far in the past.

The recursive formula is also much faster to calculate because it doesn't require a buffer of past data.

The compiled C++ function `stats:::C_rfilter()` calculates the trailing weighted values recursively.

The function `HighFreq::run_mean()` also calculates the trailing weighted values recursively.




```
> # Calculate VWAP prices recursively using C++ code
> lambdaf <- 0.9
> volumer <- .Call(stats:::C_rfilter, volumv, lambdaf, c(as.numeric(
> pricer <- .Call(stats:::C_rfilter, volumv*closep, lambdaf, c(as.nu
> vwapr <- pricer/volumer
> # Calculate VWAP prices recursively using RcppArmadillo C++
> vwapcpp <- HighFreq::run_mean(closep, lambda=lambdaf, weightv=volu
> all.equal(vwapr, drop(vwapcpp))
> # Dygraphs plot the VWAP prices
> pricev <- xts(cbind(vwap, vwapr), zoo::index(ohlc))
> colnames(pricev) <- c("VWAP trailing", "VWAP recursive")
> dygraphs::dygraph(pricev["2008/2009"], main="VWAP Prices") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Smooth Asset Returns

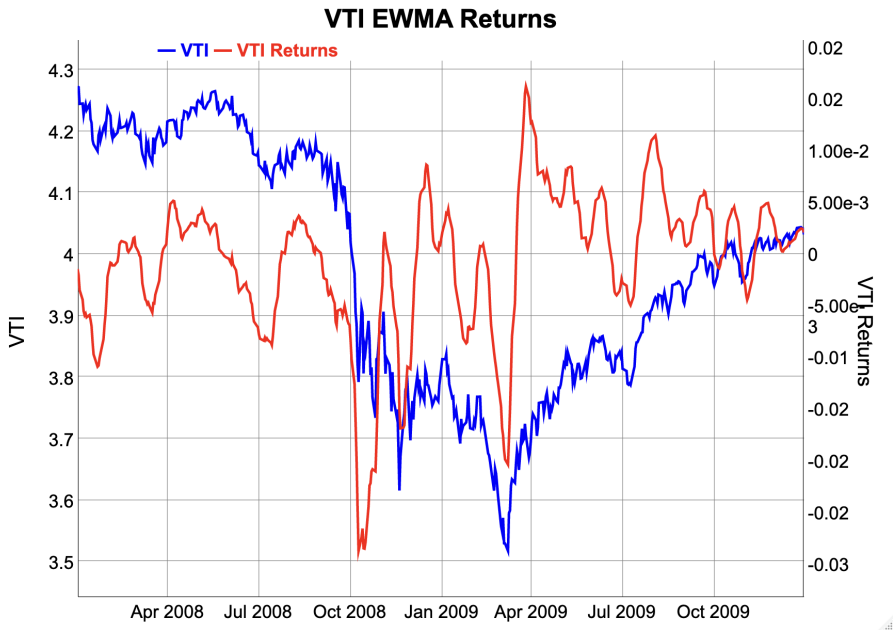
Asset returns are calculated by filtering prices through a *differencing* filter.

The simplest *differencing* filter is the filter with coefficients $(1, -1)$: $r_t = p_t - p_{t-1}$.

Differencing is a *high-pass filter*, since it eliminates low frequency signals, and it passes through high frequency signals.

An alternative measure of returns is the difference between two moving averages of prices: $r_t = p_t^{fast} - p_t^{slow}$

The difference between moving averages is a *mid-pass filter*, since it eliminates both low and high frequency signals, and it passes through medium frequency signals.




```
> # Calculate two EMA prices
> look_back <- 21
> lambdaf <- 0.1
> weightv <- exp(-lambdaf*1:look_back)
> weightv <- weightv/sum(weightv)
> emaf <- HighFreq::roll_conv(closep, weightv=weightv)
> lambdas <- 0.05
> weightv <- exp(-lambdas*1:look_back)
> weightv <- weightv/sum(weightv)
> emas <- HighFreq::roll_conv(closep, weightv=weightv)
```

```
> # Calculate VTI prices
> emad <- (emaf - emas)
> pricev <- cbind(closep, emad)
> symbol <- "VTI"
> colnames(pricev) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main=paste(symbol, "EMA Ret
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Fractional Asset Returns

The lag operator L applies a lag (time shift) to a time series: $L(p_t) = p_{t-1}$.

The simple returns can then be expressed as equal to the returns operator $(1 - L)$ applied to the prices: $r_t = (1 - L)p_t$.

The simple returns can be generalized to the fractional returns by raising the returns operator to some power $\delta < 1$:

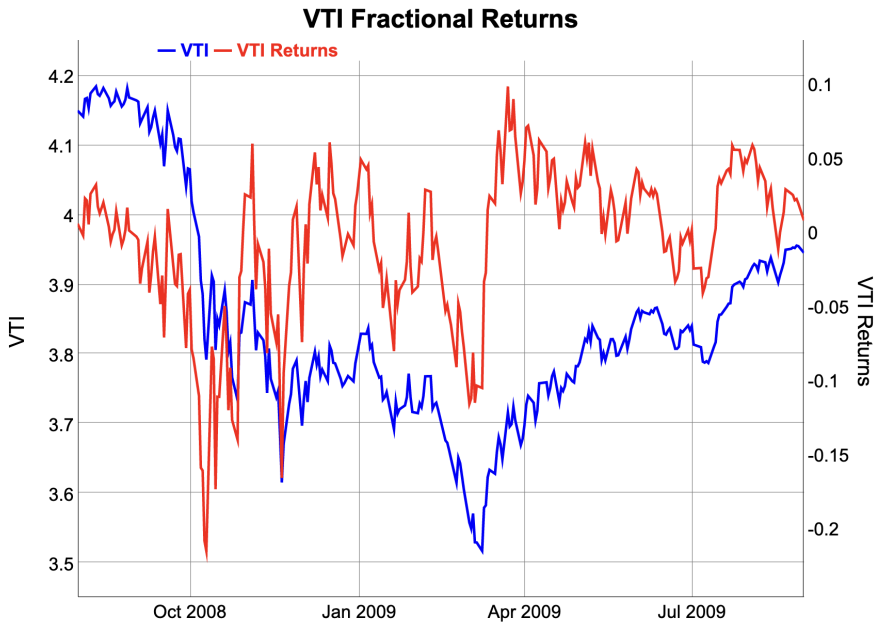
$$r_t = (1 - L)^{\delta} p_t =$$

$$p_t - \delta L p_t + \frac{\delta(\delta - 1)}{2!} L^2 p_t - \frac{\delta(\delta - 1)(\delta - 2)}{3!} L^3 p_t + \cdots =$$

$$p_t - \delta p_{t-1} + \frac{\delta(\delta - 1)}{2!} p_{t-2} - \frac{\delta(\delta - 1)(\delta - 2)}{3!} p_{t-3} + \cdots$$

The fractional returns provide a tradeoff between simple returns (which are range-bound but with no memory) and prices (which have memory but are not range-bound).

```
> # Calculate the fractional weights
> deltav <- 0.1
> weightv <- (deltav - 0:(look_back-2)) / 1:(look_back-1)
> weightv <- (-1)^(1:(look_back-1))*cumprod(weightv)
> weightv <- c(1, weightv)
> weightv <- (weightv - mean(weightv))
```




```
> # Calculate the fractional VTI returns
> retf <- HighFreq::roll_conv(closep, weightv=weightv)
> pricev <- cbind(closep, retf)
> symbol <- "VTI"
> colnames(pricev) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008-08/2009-08"], main=paste(symbol, "
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Augmented Dickey-Fuller Test for Asset Returns

The cumulative sum of a given process is called its *integrated* process.

For example, asset prices follow an *integrated* process with respect to asset returns: $p_t = \sum_{i=1}^{t} r_i$.

Integrated processes typically have a *unit root* (they have unlimited range), even if their underlying difference process does not have a *unit root* (has limited range).

Asset returns don't have a *unit root* (they have limited range) while prices have a *unit root* (they have unlimited range).

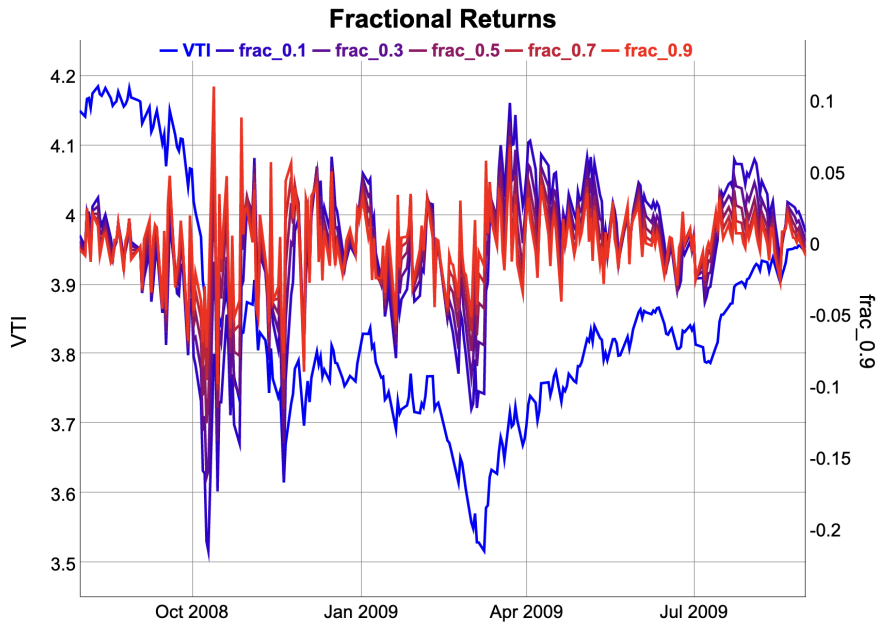
The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

```
> # Perform ADF test for prices
> tseries::adf.test(closep)
> # Perform ADF test for returns
> tseries::adf.test(retp)
```

Augmented Dickey-Fuller Test for Fractional Returns

The fractional returns for exponent values close to zero $\delta \approx 0$ resemble the asset price, while for values close to one $\delta \approx 1$ they resemble the standard returns.

```
> # Calculate fractional VTI returns
> deltav <- 0.1*c(1, 3, 5, 7, 9)
> retfrac <- lapply(deltav, function(deltav) {
+   weightv <- (deltav - 0:(look_back-2)) / 1:(look_back-1)
+   weightv <- c(1, (-1)^(1:(look_back-1))*cumprod(weightv))
+   weightv <- (weightv - mean(weightv))
+   HighFreq::roll_conv(closep, weightv=weightv)
+ })  # end lapply
> retfrac <- do.call(cbind, retfrac)
> retfrac <- cbind(closep, retfrac)
> colnames(retfrac) <- c("VTI", paste0("frac_", deltav))
> # Calculate ADF test statistics
> adfstats <- sapply(retfrac, function(x)
+   suppressWarnings(tseries::adf.test(x)$statistic)
+ )  # end sapply
> names(adfstats) <- colnames(retfrac)
```




```
> # Plot dygraph of fractional VTI returns
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(retfrac))
> colnamev <- colnames(retfrac)
> dyplot <- dygraphs::dygraph(retfrac["2008-08/2009-08"], main="Frac
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
> for (i in 2:NROW(colnamev))
+   dyplot <- dyplot %>%
+   dyAxis("y2", label=colnamev[i], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[i], axis="y2", label=colnamev[i], strokeW
> dyplot <- dyplot %>% dyLegend(width=300)
> dyplot
```

Trading Volume Z-Scores

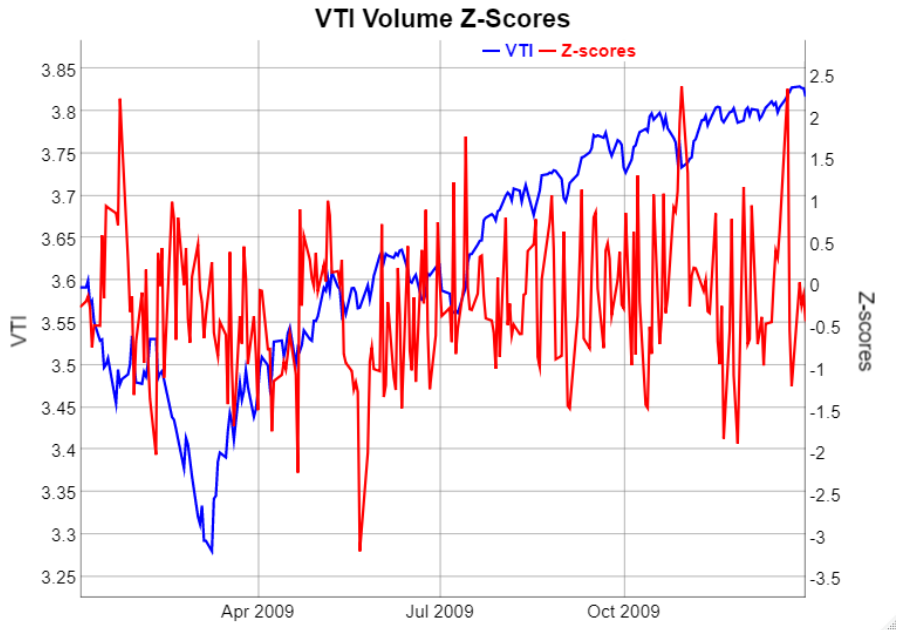
The trailing *volume z-score* is equal to the volume v_t minus the trailing average volumes $\bar{v}_t$ divided by the volatility of the volumes σ_t:

$$z_t = \frac{v_t - \bar{v}_t}{\sigma_t}$$

Trading volumes are typically higher when prices drop and they are also positively correlated with the return volatility.

The volume z-scores represent the first derivative (slope) of the volumes, since the volume level is subtracted.

The volume z-scores are positively skewed because returns are negatively skewed.




```
> # Calculate volume z-scores
> volumv <- quantmod::Vo(ohlc)
> look_back <- 21
> volumean <- HighFreq::roll_mean(volumv, look_back=look_back)
> volumsd <- sqrt(HighFreq::roll_var(rutils::diffit(volumv), look_
> volumsd[1] <- 0
> volumz <- ifelse(volumsd > 0, (volumv - volumean)/volumsd, 0)
> # Plot histogram of volume z-scores
> hist(volumz, breaks=1e2)
```

```
> # Plot dygraph of volume z-scores of VTI prices
> pricev <- cbind(closep, volumz)
> colnames(pricev) <- c("VTI", "Z-Scores")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main="VTI Volume Z-Scores")
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Volatility Z-Scores

The *true range* is the difference between low and high prices is a proxy for the spot volatility in a bar of data.

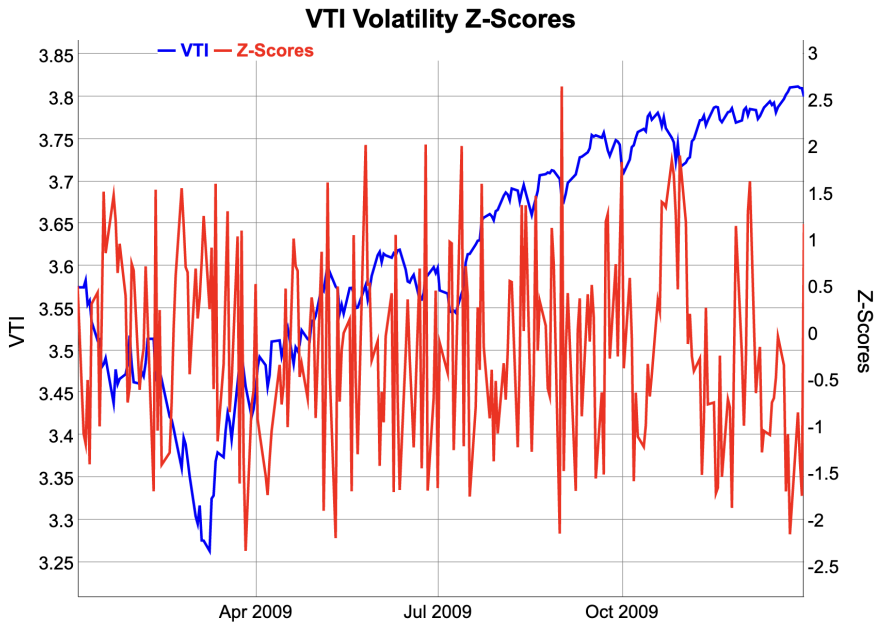
The *volatility z-score* is equal to the spot volatility v_t minus the trailing average volatility $\bar{v}_t$ divided by the standard deviation of the volatility σ_t:

$$z_t = \frac{v_t - \bar{v}_t}{\sigma_t}$$

Volatility is typically higher when prices drop and it's also positively correlated with the trading volumes.

The volatility z-scores represent the first derivative (slope) of the volatilities, since the volatility level is subtracted.

The volatility z-scores are positively skewed because returns are negatively skewed.




```
> # Calculate volatility (true range) z-scores
> volat <- log(quantmod::Hi(ohlc) - quantmod::Lo(ohlc))
> look_back <- 21
> volatm <- HighFreq::roll_mean(volat, look_back=look_back)
> volat <- (volat - volatm)
> volatsd <- sqrt(HighFreq::roll_var(rutils::diffit(volat), look_ba
> volatsd[1] <- 0
> volatz <- ifelse(volatsd > 0, volat/volatsd, 0)
> # Plot histogram of the volatility z-scores
> hist(volatz, breaks=1e2)
```

```
> # Plot scatterplot of volume and volatility z-scores
> plot(as.numeric(volatz), as.numeric(volumz),
+      xlab="volatility z-score", ylab="volume z-score")
> regmod <- lm(volatz ~ volumz)
> abline(regmod, col="red", lwd=3)
> # Plot dygraph of VTI volatility z-scores
> pricev <- cbind(closep, volatz)
> colnames(pricev) <- c("VTI", "Z-Scores")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main="VTI Volatility Z-Scor
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeW
+   dyLegend(show="always", width=300)
```

Trailing Volatility Z-Scores

The *volatility z-score* can also be defined as the difference between the fast v_t^f minus the slow v_t^s trailing volatilities, divided by the standard deviation of the volatility σ_t:

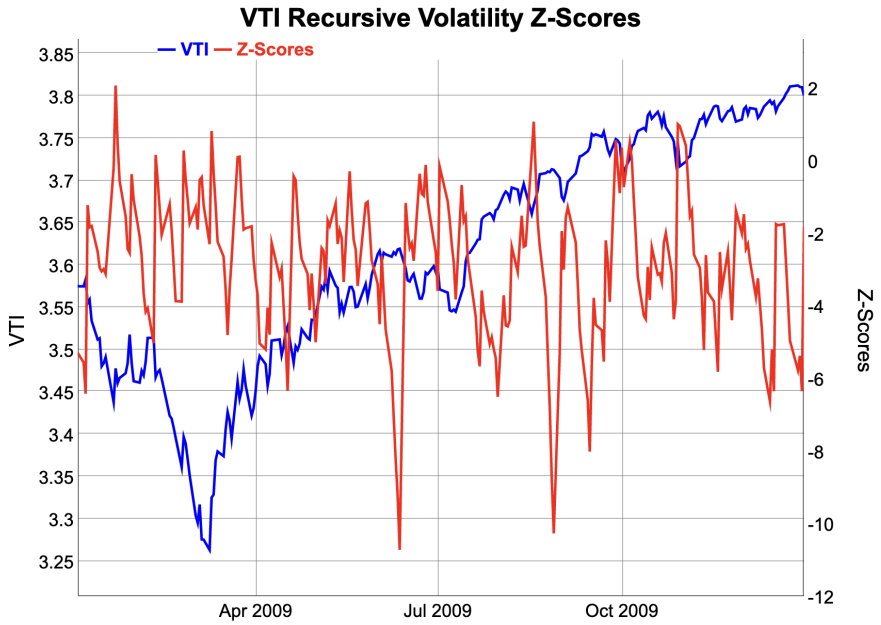
$$z_t = \frac{v_t^f - v_t^s}{\sigma_t}$$

The function `HighFreq::run_var()` calculates the trailing variance of a *time series* of returns, by recursively weighting the past variance estimates σ_{t-1}^2, with the squared differences of the returns minus the trailing means $(r_t - \bar{r}_t)^2$, using the weight decay factor λ:

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

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

The parameter λ determines the rate of decay of the weight of past values.




```
> # Calculate the recursive trailing VTI volatility
> lambdaf <- 0.8
> lambdas <- 0.81
> volatf <- sqrt(HighFreq::run_var(retp, lambda=lambdaf))
> volats <- sqrt(HighFreq::run_var(retp, lambda=lambdas))
> # Calculate the recursive trailing z-scores of VTI volatility
> volatd <- volatf - volats
> volatsd <- sqrt(HighFreq::run_var(rutils::diffit(volatd), lambda=
> volatsd[1] <- 0
```

```
> # Plot histogram of the volatility z-scores
> hist(volatz, breaks=1e2)
> # Plot scatterplot of volume and volatility z-scores
> plot(as.numeric(volatz), as.numeric(volumz),
+      xlab="volatility z-score", ylab="volume z-score")
> regmod <- lm(volatz ~ volumz)
> abline(regmod, col="red", lwd=3)
> # Plot dygraph of VTI volatility z-scores
> pricev <- cbind(closep, volatz)
> colnames(pricev) <- c("VTI", "Z-Scores")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2008/2009"], main="VTI Online Volatility
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Centered Price Z-Scores

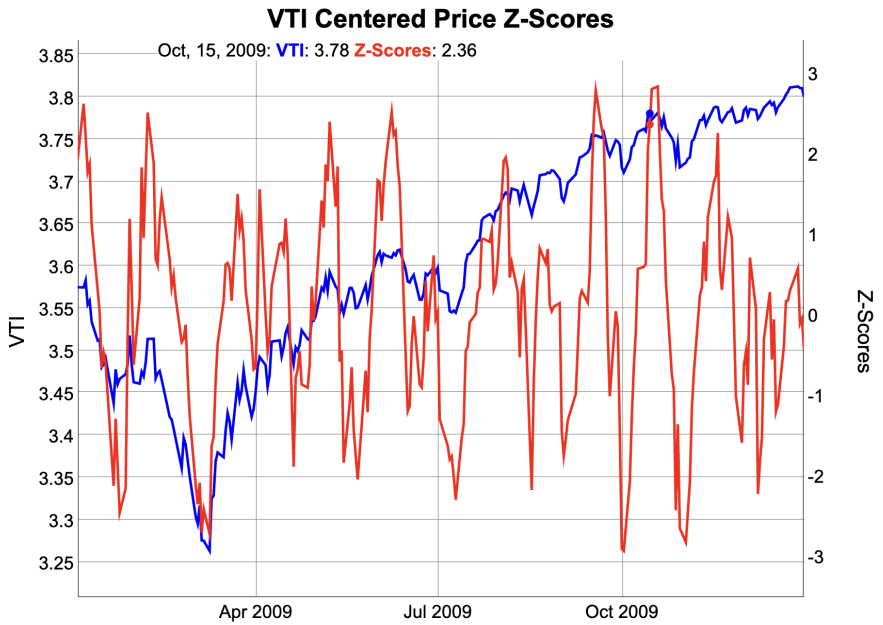
An extreme local price is a price which differs significantly from neighboring prices.

Extreme prices can be identified in-sample using the centered *price z-score* equal to the price difference with neighboring prices divided by the volatility of returns σ_t:

$$z_t = \frac{p_t - 0.5(p_{t-k} - p_{t+k})}{\sigma_t}$$

Where p_{t-k} and p_{t+k} are the lagged and advanced prices.

The lag parameter k is the interval for calculating the volatility of returns σ_t.




```
> # Calculate the centered volatility
> look_back <- 21
> half_back <- look_back %/% 2
> volat <- HighFreq::roll_var(closep, look_back=look_back)
> volat <- sqrt(volat)
> volat <- rutils::lagit(volat, lagg=(-half_back))
> # Calculate the z-scores of prices
> pricez <- (closep -
+   0.5*(rutils::lagit(closep, half_back, pad_zeros=FALSE) +
+   rutils::lagit(closep, -half_back, pad_zeros=FALSE)))
> pricez <- ifelse(volat > 0, pricez/volat, 0)
```

```
> # Plot dygraph of z-scores of VTI prices
> pricev <- cbind(closep, pricez)
> colnames(pricev) <- c("VTI", "Z-Scores")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2009"], main="VTI Centered Price Z-Score
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Labeling the Tops and Bottoms of Prices

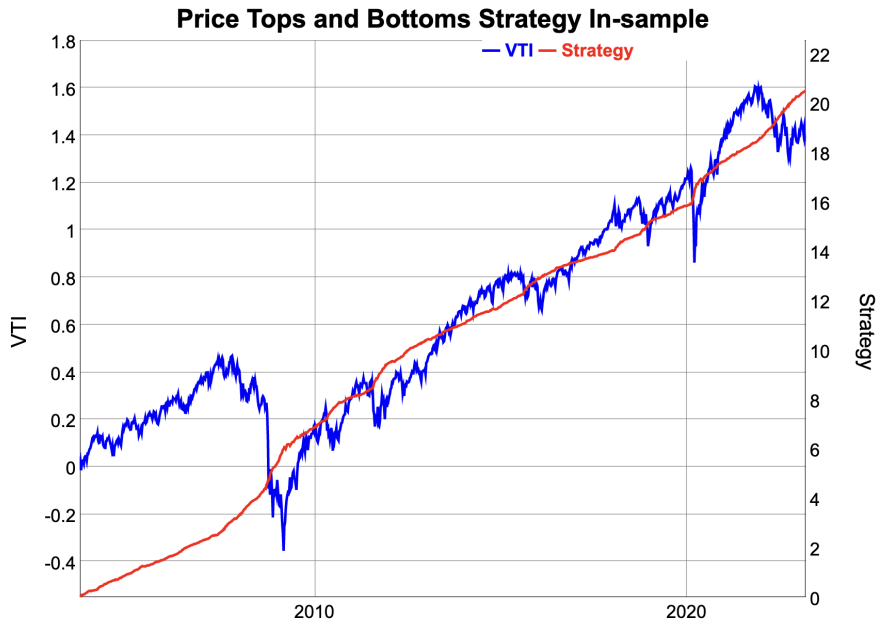
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

The local tops of prices represent *overbought* conditions, while the bottoms represent *oversold* conditions.

The labeled data can be used as a response or target variable in machine learning classifier models.

But it's not feasible to classify the prices out-of-sample exactly according to their in-sample labels.

```
> # Calculate the thresholds for labeling tops and bottoms
> confl <- c(0.2, 0.8)
> threshv <- quantile(pricez, confl)
> # Calculate the vectors of tops and bottoms
> tops <- zoo::coredata(pricez > threshv[2])
> bottoms <- zoo::coredata(pricez < threshv[1])
> # Simulate in-sample VTI strategy
> posv <- rep(NA_integer_, nrows)
> posv[1] <- 0
> posv[tops] <- (-1)
> posv[bottoms] <- 1
> posv <- zoo::na.locf(posv)
> posv <- rutils::lagit(posv)
> pnls <- retp*posv
```




```
> # Plot dygraph of in-sample VTI strategy
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Price Tops and Bottoms Strategy In-sample") %>%
+   dyAxis("y", label="VTI", independentTicks=TRUE) %>%
+   dyAxis("y2", label="Strategy", independentTicks=TRUE) %>%
+   dySeries(name="VTI", axis="y", label="VTI", strokeWidth=2, col="
+   dySeries(name="Strategy", axis="y2", label="Strategy", strokeWi
```

Trailing Price Z-Scores

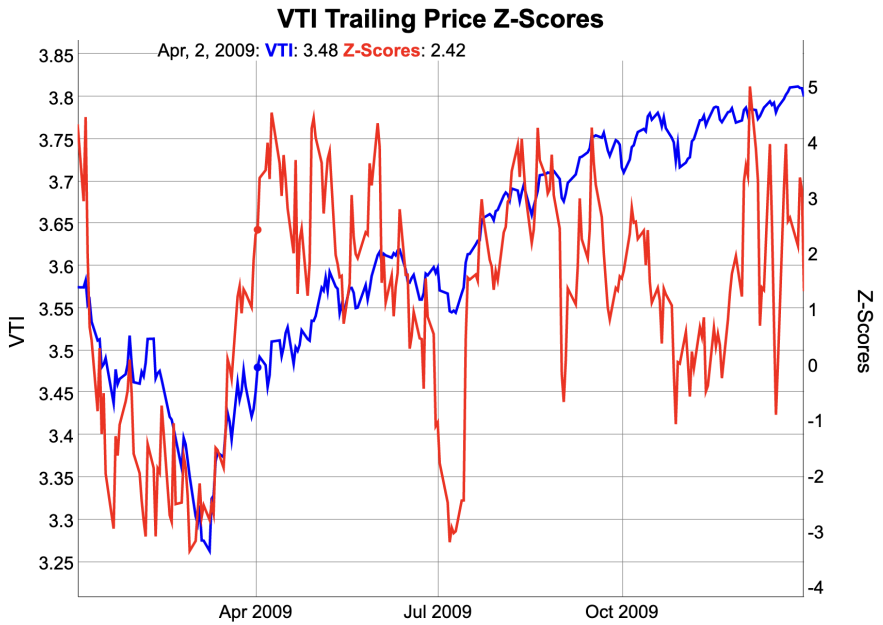
The trailing price z-score is equal to the difference between the current price p_t minus the trailing average price $\bar{p}_{t-k}$, divided by the volatility of the price σ_t:

$$z_t = \frac{p_t - \bar{p}_{t-k}}{\sigma_t}$$

The lag parameter k is the look-back interval for calculating the volatility of returns σ_t.

The trailing price z-scores represent the first derivative (slope) of the prices, since the price level is subtracted.

```
> # Calculate the trailing VTI volatility
> volat <- HighFreq::roll_var(closep, look_back=look_back)
> volat <- sqrt(volat)
> # Calculate the trailing z-scores of VTI prices
> pricez <- (closep - rutils::lagit(closep, look_back, pad_zeros=F
> pricez <- ifelse(volat > 0, pricez/volat, 0)
```




```
> # Plot dygraph of the trailing z-scores of VTI prices
> pricev <- cbind(closep, pricez)
> colnames(pricev) <- c("VTI", "Z-Scores")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2009"],
+   main="VTI Trailing Price Z-Scores") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(axis="y", label=colnamev[1], strokeWidth=2, col="blue")
+   dySeries(axis="y2", label=colnamev[2], strokeWidth=2, col="red")
+   dyLegend(show="always", width=300)
```

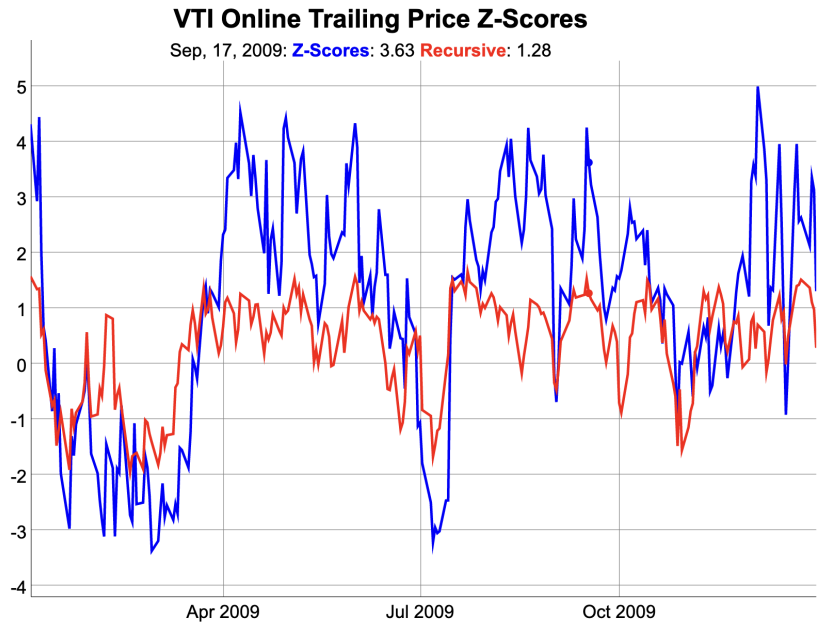
Recursive Trailing Price Z-Scores

The recursive trailing price z-score is equal to the difference between the current price p_t minus the trailing average price $\bar{p}$, divided by the volatility of the price σ_t:

$$\bar{p}_t = \lambda \bar{p}_{t-1} + (1-\lambda)p_t$$
$$\sigma_t^2 = \lambda \sigma_{t-1}^2 + (1-\lambda)(p_t - \bar{p}_t)^2$$
$$z_t = \frac{p_t - \bar{p}_t}{\sigma_t}$$

The parameter λ determines the rate of decay of the weight of past prices. If λ is close to 1 then the decay is weak and past prices have a greater weight, and the trailing mean values have a stronger dependence on past prices. This is equivalent to a long look-back interval. And vice versa if λ is close to 0.

The functions `HighFreq::run_mean()` and `HighFreq::run_var()` calculate the trailing mean and variance by recursively updating the past estimates with the new values, using the weight decay factor λ.




```
> # Calculate the recursive trailing VTI volatility
> lambdaf <- 0.9
> volat <- HighFreq::run_var(closep, lambda=lambdaf)
> volat <- sqrt(volat)
> # Calculate the recursive trailing z-scores of VTI prices
> pricer <- (closep - HighFreq::run_mean(closep, lambda=lambdaf))
> pricer <- ifelse(volat > 0, pricer/volat, 0)
> # Plot dygraph of the trailing z-scores of VTI prices
> pricev <- xts::xts(cbind(pricez, pricer), datev)
> colnames(pricev) <- c("Z-Scores", "Recursive")
> colnamev <- colnames(pricev)
> dygraphs::dygraph(pricev["2009"], main="VTI Online Trailing Price
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Trailing Regression Z-Scores

We can define the trailing *z-score* z_t of the stock price p_t as the *standardized residual* of the linear regression with respect to a predictor variable (for example the time t_i):

$$z_t = \frac{p_t - p_t^{fit}}{\sigma_t}$$

$$p_t^{fit} = \alpha_t + \beta_t t_i$$

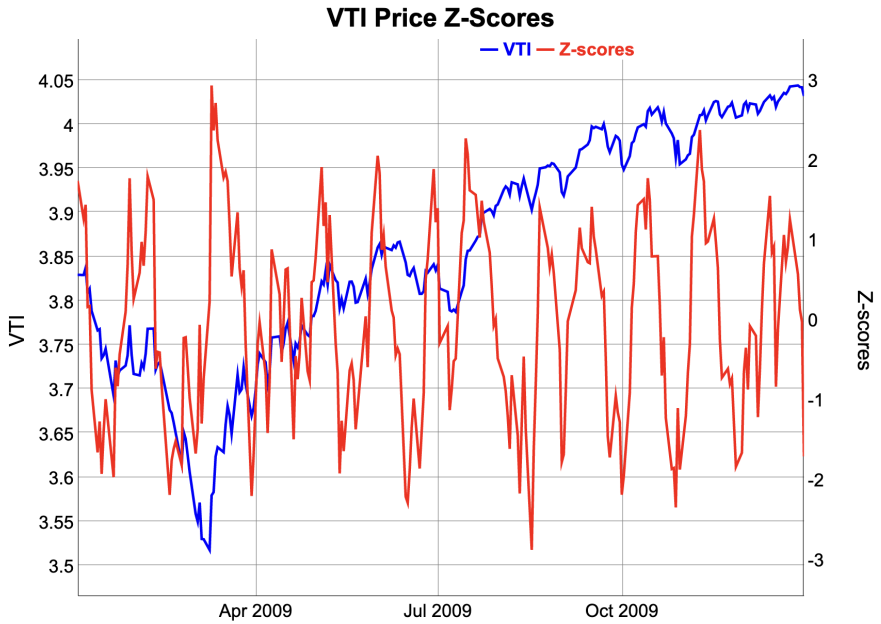
Where p_t^{fit} are the fitted values, α_t and β_t are the *regression coefficients*, and σ_t is the standard deviation of the residuals.

The regression z-scores represent the second derivative (curvature) of the stock prices, since the price level and slope are subtracted.

The regression z-scores can be used as a rich or cheap indicator, either relative to past prices, or relative to prices in a stock pair.

The regression residuals must be calculated in a loop, so it's much faster to calculate them using functions written in C++ code.

The function `HighFreq::roll_reg()` calculates trailing regressions and their residuals.




```
> # Calculate trailing price regression z-scores
> datev <- matrix(zoo::index(closep))
> look_back <- 21
> # Create a default list of regression parameters
> controlv <- HighFreq::param_reg()
> regs <- HighFreq::roll_reg(respv=closep, predm=datev,
+    look_back=look_back, controlv=controlv)
> regs[1:look_back, ] <- 0
> # Plot dygraph of z-scores of VTI prices
> datav <- cbind(closep, regs[, NCOL(regs)])
> colnames(datav) <- c("VTI", "Z-Scores")
> colnamev <- colnames(datav)
> dygraphs::dygraph(datav["2009"], main="VTI Regression Z-Scores") %
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Recursive Trailing Regression

The trailing regressions of the stock price p_t with respect to the predictor (explanatory) variables X_t are defined by:

$$p_t = \beta_t X_t + \epsilon_t$$

The trailing regression coefficients β_t and the residuals ϵ_t can be calculated as:

$$\beta_t = \text{cov}_{Xt}^{-1} \, \text{cov}_t$$

$$\epsilon_t = r_t - \beta_t p_t$$

Where cov_t is the covariance matrix between the response p_t and the predictor X_t variables, and cov_{Xt} is the covariance matrix between the predictors.

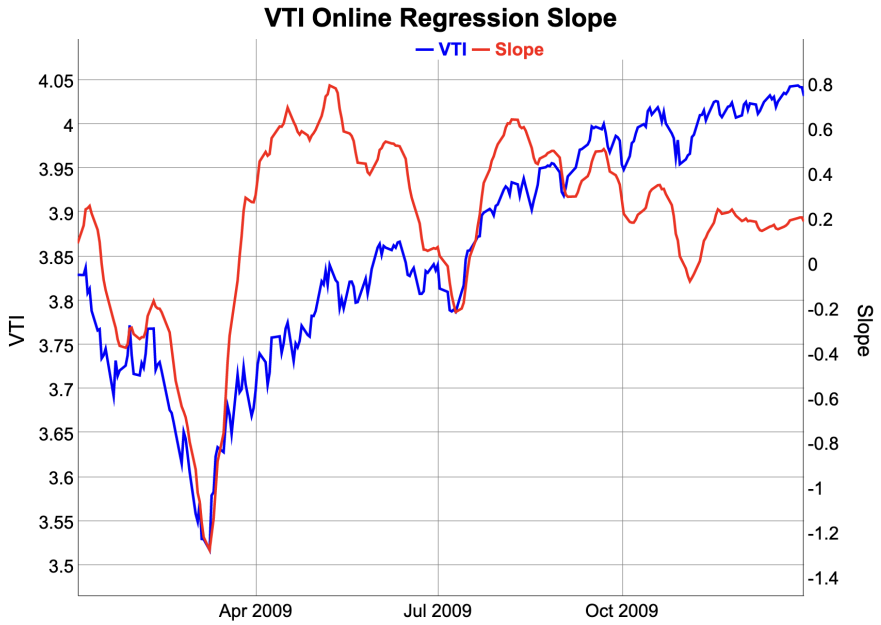
The covariance matrices are updated using the following recursive (online) formulas:

$$\text{cov}_t = \lambda \, \text{cov}_{t-1} + (1 - \lambda) p_t^T X_t$$

$$\text{cov}_{Xt} = \lambda \, \text{cov}_{X(t-1)} + (1 - \lambda) X_t^T X_t$$

The function `HighFreq::run_reg()` recursively calculates trailing regressions and their residuals.

```
> # Calculate recursive trailing price regression versus time
> lambdaf <- 0.9
> # Create a list of regression parameters
> controlv <- HighFreq::param_reg(residscale="standardize")
> regs <- HighFreq::run_reg(closep, matrix(datev), lambda=lambdaf, controlv=controlv)
> colnames(regs) <- c("alpha", "beta", "zscores")
> tail(regs)
```



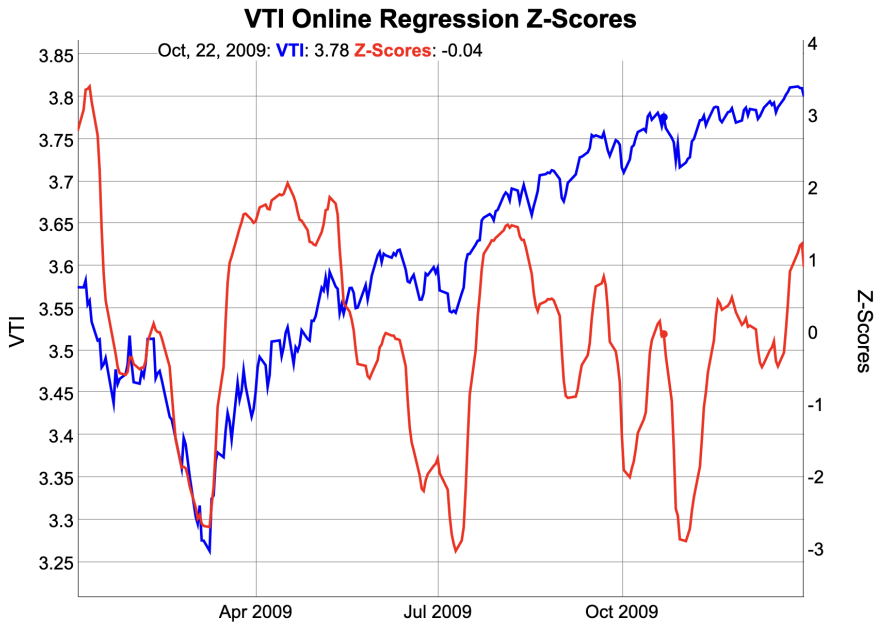

```
> # Plot dygraph of regression betas
> datav <- cbind(closep, 252*regs[, "beta"])
> colnames(datav) <- c("VTI", "Slope")
> colnamev <- colnames(datav)
> dygraphs::dygraph(datav["2009"], main="VTI Online Regression Slope
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(axis="y", label=colnamev[1], strokeWidth=2, col="blue")
+   dySeries(axis="y2", label=colnamev[2], strokeWidth=2, col="red")
+   dyLegend(show="always", width=300)
```

Recursive Trailing Regression Z-Scores

The recursive trailing *z-score* z_t of the stock price p_t is equal to the standardized residual ϵ_t:

$$\epsilon_t = \lambda\epsilon_{t-1} + (1-\lambda)(p_t - \beta_t p_t)$$
$$\bar{\epsilon}_t = \lambda\bar{\epsilon}_{t-1} + (1-\lambda)\epsilon_t$$
$$\varsigma_t^2 = \lambda\varsigma_{t-1}^2 + (1-\lambda)(\epsilon_t - \bar{\epsilon}_t)^2$$
$$z_t = \frac{\epsilon_t}{\varsigma_t}$$

Where ς_t^2 is the variance of the residuals ϵ_t.




```
> # Plot dygraph of z-scores of VTI prices
> datav <- cbind(closep, regs[, "zscores"])
> colnames(datav) <- c("VTI", "Z-Scores")
> colnamev <- colnames(datav)
> dygraphs::dygraph(datav["2009"], main="VTI Online Regression Z-Sc
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

The Hampel Filter

The *Median Absolute Deviation* (*MAD*) is a nonparametric measure of dispersion (variability):

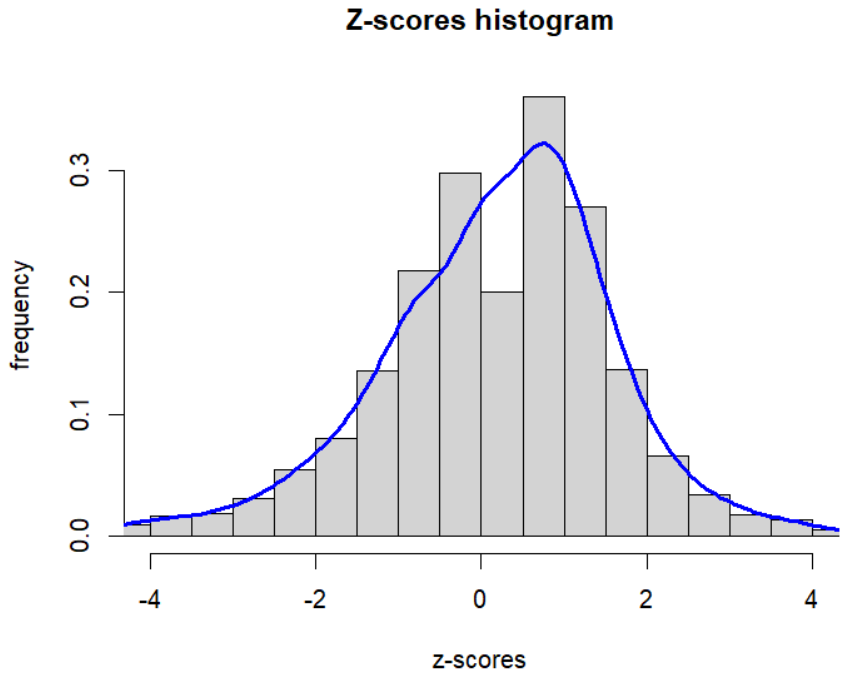
$$\text{MAD} = \text{median}(\text{abs}(p_t - \text{median}(\mathbf{p})))$$

The *Hampel filter* is effective in detecting outliers in the data because it uses the nonparametric *MAD* dispersion measure.

The *Hampel z-score* is equal to the deviation from the median divided by the *MAD*:

$$z_i = \frac{p_t - \text{median}(\mathbf{p})}{\text{MAD}}$$

A time series of *z-scores* over past data can be calculated using a trailing look-back window.




```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Define look-back window
> look_back <- 11
> # Calculate time series of trailing medians
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> # Calculate time series of MAD
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="
> # madv <- TTR::runMAD(closep, n=look_back)
> # Calculate time series of z-scores
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
```

```
> # Plot the prices and medians
> dygraphs::dygraph(cbind(closep, medianv), main="VTI median") %>%
+   dyOptions(colors=c("black", "red")) %>%
+   dyLegend(show="always", width=300)
> # Plot histogram of z-scores
> histp <- hist(zscores, col="lightgrey",
+   xlab="z-scores", breaks=50, xlim=c(-4, 4),
+   ylab="frequency", freq=FALSE, main="Hampel Z-Scores histogram")
> lines(density(zscores, adjust=1.5), lwd=3, col="blue")
```

One-sided and Two-sided Data Filters

Filters calculated over past data are referred to as *one-sided* filters, and they are appropriate for filtering real-time data.

Filters calculated over both past and future data are called *two-sided* (centered) filters, and they are appropriate for filtering historical data.

The function `HighFreq::roll_var()` with parameter `method="nonparametric"` calculates the trailing *MAD* using a look-back interval over past data.

The functions `TTR::runMedian()` and `TTR::runMAD()` calculate the trailing medians and *MAD* using a trailing look-back interval over past data.

If the trailing medians and *MAD* are advanced (shifted backward) in time, then they are calculated over both past and future data (centered).

The function `rutils::lag_it()` with a negative `lagg` parameter value advances (shifts back) future data points to the present.

```
> # Calculate one-sided Hampel z-scores
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="no
> # madv <- TTR::runMAD(closep, n=look_back)
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
> # Calculate two-sided Hampel z-scores
> half_back <- look_back %/% 2
> medianv <- rutils::lagit(medianv, lagg=(-half_back))
> madv <- rutils::lagit(madv, lagg=(-half_back))
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
```

Calculating the Trailing Variance of Asset Returns

The variance of asset returns exhibits *heteroskedasticity*, i.e. it changes over time.

The trailing variance of returns is given by:

$$\sigma_t^2 = \frac{1}{k-1}\sum_{j=0}^{k-1}(r_{t-j} - \bar{r}_t)^2$$

$$\bar{r}_t = \frac{1}{k}\sum_{j=0}^{k-1} r_{t-j}$$

Where `k` is the *look-back interval* equal to the number of data points for performing aggregations over the past.

It's also possible to calculate the trailing variance in `R` using vectorized functions, without using an `apply()` loop.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> # Define end points
> endd <- 1:NROW(retp)
> # Start points are multi-period lag of endd
> look_back <- 11
> startp <- c(rep_len(0, look_back-1), endd[1:(nrows-look_back+1)])
> # Calculate trailing variance in sapply() loop - takes long
> varv <- sapply(1:nrows, function(it) {
+   retp <- retp[startp[it]:endd[it]]
+   sum((retp - mean(retp))^2)/look_back
+ })  # end sapply
> # Use only vectorized functions
> retc <- cumsum(retp)
> retc <- (retc - c(rep_len(0, look_back), retc[1:(nrows-look_back)]
> retc2 <- cumsum(retp^2)
> retc2 <- (retc2 - c(rep_len(0, look_back), retc2[1:(nrows-look_bac
> var2 <- (retc2 - retc^2/look_back)/look_back
> all.equal(varv[-(1:look_back)], as.numeric(var2)[-(1:look_back)])
> # Or using package rutils
> retc <- rutils::roll_sum(retp, look_back=look_back)
> retc2 <- rutils::roll_sum(retp^2, look_back=look_back)
> var2 <- (retc2 - retc^2/look_back)/look_back
> # Coerce variance into xts
> tail(varv)
> class(varv)
> varv <- xts(varv, order.by=zoo::index(retp))
> colnames(varv) <- "VTI.variance"
> head(varv)
```

Calculating the Trailing Variance Using Package *roll*

The package *roll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()` for the *weighted* trailing sum,
- `roll_var()` for the *weighted* trailing variance,
- `roll_scale()` for the trailing scaling and centering of time series,
- `roll_pcr()` for the trailing principal component regressions of time series.

The *roll* functions are about `1,000` times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled `C++` code, using packages *Rcpp*, *RcppArmadillo*, and *RcppParallel*.

The *roll* functions accept *xts* time series, and they return *xts*.

```
> # Calculate trailing VTI variance using package HighFreq
> varv <- roll::roll_var(retp, width=look_back)
> colnames(varv) <- "Variance"
> head(varv)
> sum(is.na(varv))
> varv[1:(look_back-1)] <- 0
> # Benchmark calculation of trailing variance
> library(microbenchmark)
> summary(microbenchmark(
+   sapply=sapply(1:nrows, function(it) {
+     var(retp[startp[it]:endd[it]])
+   }),
+   roll=roll::roll_var(retp, width=look_back),
+   times=10))[, c(1, 4, 5)]
```

Trailing *EMA* Realized Volatility Estimator

Time-varying volatility can be more accurately estimated using an *Exponentially Weighted Moving Average* (*EMA*) variance estimator.

If the *time series* has zero *expected* mean, then the *EMA realized* variance estimator can be written approxiamtely as:

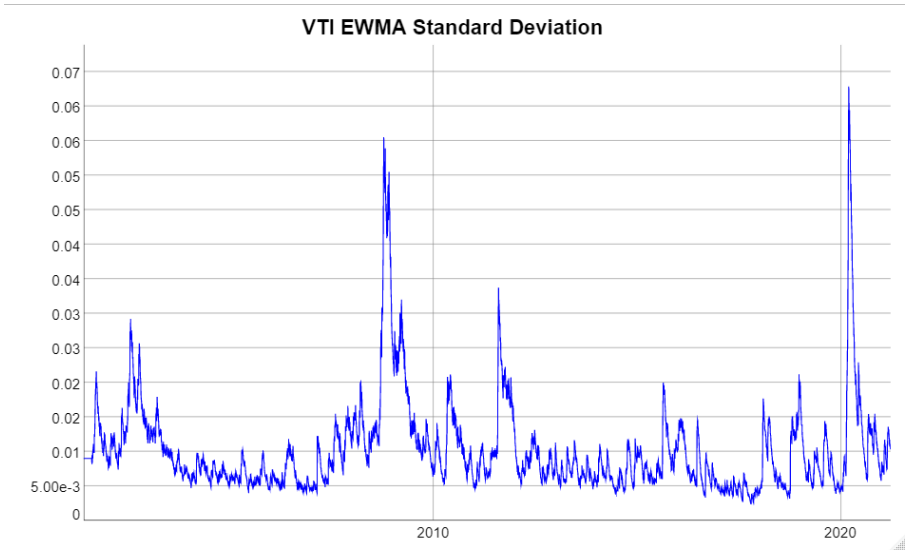
$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2 = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j r_{t-j}^2$$

σ_t^2 is the weighted *realized* variance, equal to the weighted average of the point realized variance for period `i` and the past *realized* variance.

The parameter λ determines the rate of decay of the *EMA* weights, with smaller values of λ producing faster decay, giving more weight to recent realized variance, and vice versa.

The function `stats:::C_cfilter()` calculates the convolution of a vector or a time series with a filter of coefficients (weights).

The function `stats:::C_cfilter()` is very fast because it's compiled `C++` code.




```
> # Calculate EMA VTI variance using compiled C++ function
> look_back <- 51
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> varv <- .Call(stats:::C_cfilter, retp^2, filter=weightv, sides=1,
> varv[1:(look_back-1)] <- varv[look_back]
> # Plot EMA volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(retp))
> dygraphs::dygraph(varv, main="VTI EMA Volatility") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
> quantmod::chart_Series(xtsv, name="VTI EMA Volatility")
```

Estimating *EMA* Variance Using Package *roll*

If the *time series* has non-zero *expected* mean, then the trailing *EMA* variance is a vector given by the estimator:

$$\sigma_t^2 = \frac{1}{k-1}\sum_{j=0}^{k-1} w_j (r_{t-j} - \bar{r}_t)^2$$

$$\bar{r}_t = \frac{1}{k}\sum_{j=0}^{k-1} w_j r_{t-j}$$

Where w_j is the vector of exponentially decaying weights:

$$w_j = \frac{\lambda^j}{\sum_{j=0}^{k-1} \lambda^j}$$

The function `roll_var()` from package *roll* calculates the trailing *EMA* variance.

```
> # Calculate trailing VTI variance using package roll
> library(roll)  # Load roll
> varv <- roll::roll_var(retp, weights=rev(weightv), width=look_back
> colnames(varv) <- "VTI.variance"
> class(varv)
> head(varv)
> sum(is.na(varv))
> varv[1:(look_back-1)] <- 0
```

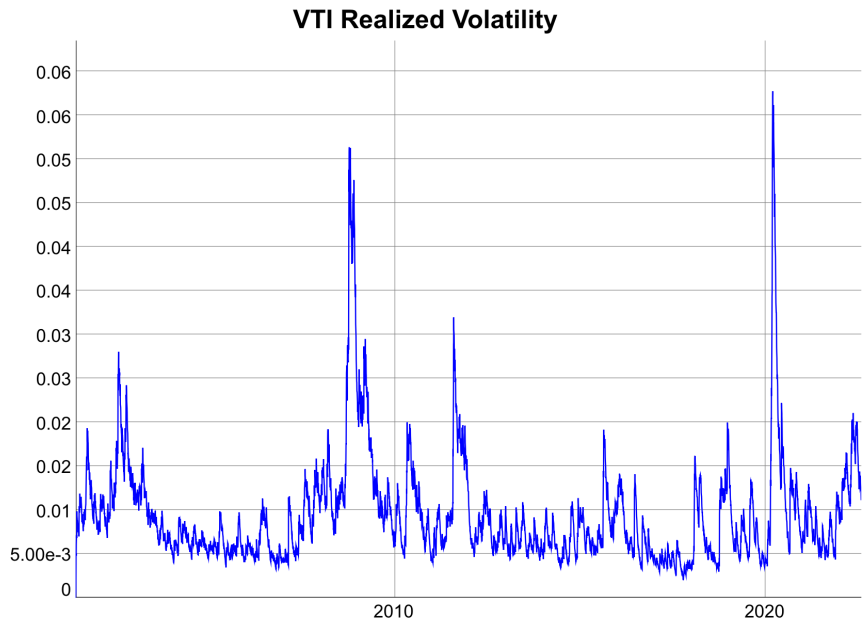
Trailing Realized Volatility Estimator

The function `HighFreq::run_var()` calculates the trailing variance of a *time series* of returns, by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the returns minus the trailing means $(r_t - \bar{r}_t)^2$, using the weight decay factor λ:

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

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

Where σ^2_t is the trailing variance at time t, and r_t are the returns.




```
> # Calculate realized variance recursively
> lambdaf <- 0.9
> volat <- HighFreq::run_var(retp, lambda=lambdaf)
> volat <- sqrt(volat)
> # Plot EMA volatility
> volat <- xts:::xts(volat, order.by=datev)
> dygraphs::dygraph(volat, main="VTI Realized Volatility") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
```

Estimating Daily Volatility From Intraday Returns

The standard *close-to-close* volatility σ depends on the *Close* prices C_i from *OHLC* data:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2$$

$$\bar{r} = \frac{1}{n}\sum_{i=0}^{n} r_i \quad r_i = \log(\frac{C_i}{C_{i-1}})$$

But intraday time series of prices (for example `HighFreq::SPY` prices), can have large overnight jumps which inflate the volatility estimates.

So the overnight returns must be divided by the overnight time interval (in seconds), which produces per second returns.

The per second returns can be multiplied by 60 to scale them back up to per minute returns.

The function `zoo::index()` extracts the time index of a time series.

The function `xts::.index()` extracts the time index expressed in the number of seconds.

```
> library(HighFreq)  # Load HighFreq
> # Minutely SPY returns (unit per minute) single day
> # Minutely SPY volatility (unit per minute)
> retspy <- rutils::diffit(log(SPY["2012-02-13", 4]))
> sd(retspy)
> # SPY returns multiple days (includes overnight jumps)
> retspy <- rutils::diffit(log(SPY[, 4]))
> sd(retspy)
> # Table of time intervals - 60 second is most frequent
> indeks <- rutils::diffit(xts:::.index(SPY))
> table(indeks)
> # SPY returns divided by the overnight time intervals (unit per se
> retspy <- retspy/indeks
> retspy[1] <- 0
> # Minutely SPY volatility scaled to unit per minute
> 60*sd(retspy)
```

Range Volatility Estimators of *OHLC* Time Series

Range estimators of return volatility utilize the `high` and `low` prices, and therefore have lower standard errors than the standard *close-to-close* estimator.

The *Garman-Klass* estimator uses the *low-to-high* price range, but it underestimates volatility because it doesn't account for *close-to-open* price jumps:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(0.5\log(\frac{H_i}{L_i})^2 - (2\log 2 - 1)\log(\frac{C_i}{O_i})^2)$$

The *Yang-Zhang* estimator accounts for *close-to-open* price jumps and has the lowest standard error among unbiased estimators:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\log(\frac{O_i}{C_{i-1}}) - \bar{r}_{co})^2 + 0.134(\log(\frac{C_i}{O_i}) - \bar{r}_{oc})^2 + \frac{0.866}{n}\sum_{i=1}^{n}(\log(\frac{H_i}{O_i})\log(\frac{H_i}{C_i}) + \log(\frac{L_i}{O_i})\log(\frac{L_i}{C_i}))$$

The *Yang-Zhang* (*YZ*) and *Garman-Klass-Yang-Zhang* (*GKYZ*) estimators are unbiased and have up to seven times smaller standard errors than the standard close-to-close estimator.

But in practice, prices are not observed continuously, so the price range is underestimated, and so is the variance when using the *YZ* and *GKYZ* range estimators.

Therefore in practice the *YZ* and *GKYZ* range estimators underestimate the volatility, and their standard errors are reduced less than by the theoretical amount, for the same reason.

The *Garman-Klass-Yang-Zhang* estimator is another very efficient and unbiased estimator, and also accounts for *close-to-open* price jumps:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}((\log(\frac{O_i}{C_{i-1}}) - \bar{r})^2 + 0.5\log(\frac{H_i}{L_i})^2 - (2\log 2 - 1)(\log(\frac{C_i}{O_i})^2))$$

Calculating the Trailing Range Variance Using *HighFreq*

The function `HighFreq::calc_var_ohlc()` calculates the *variance* of returns using several different range volatility estimators.

If the logarithms of the *OHLC* prices are passed into `HighFreq::calc_var_ohlc()` then it calculates the variance of percentage returns, and if simple *OHLC* prices are passed then it calculates the variance of dollar returns.

The function `HighFreq::roll_var_ohlc()` calculates the *trailing* variance of returns using several different range volatility estimators.

The functions `HighFreq::calc_var_ohlc()` and `HighFreq::roll_var_ohlc()` are very fast because they are written in C++ code.

The function `TTR::volatility()` calculates the range volatility, but it's significantly slower than `HighFreq::calc_var_ohlc()`.

```
> library(HighFreq)  # Load HighFreq
> spy <- HighFreq::SPY["2008/2009"]
> # Calculate daily SPY volatility using package HighFreq
> sqrt(6.5*60*HighFreq::calcvar_ohlc(log(spy),
+   method="yang_zhang"))
> # Calculate daily SPY volatility from minutely prices using packag
> sqrt((6.5*60)*mean(na.omit(
+   TTR::volatility(spy, N=1, calc="yang.zhang"))^2))
> # Calculate trailing SPY variance using package HighFreq
> varv <- HighFreq::roll_var_ohlc(log(spy), method="yang_zhang",
+   look_back=look_back)
> # Plot range volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(spy))
> dygraphs::dygraph(varv["2009-02"], main="SPY Trailing Range Volati
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
> # Benchmark the speed of HighFreq vs TTR
> library(microbenchmark)
> summary(microbenchmark(
+   ttr=TTR::volatility(rutils::etfenv$VTI, N=1, calc="yang.zhang"),
+   highfreq=HighFreq::calcvar_ohlc(log(rutils::etfenv$VTI), method=
+   times=2))[, c(1, 4, 5)]
```

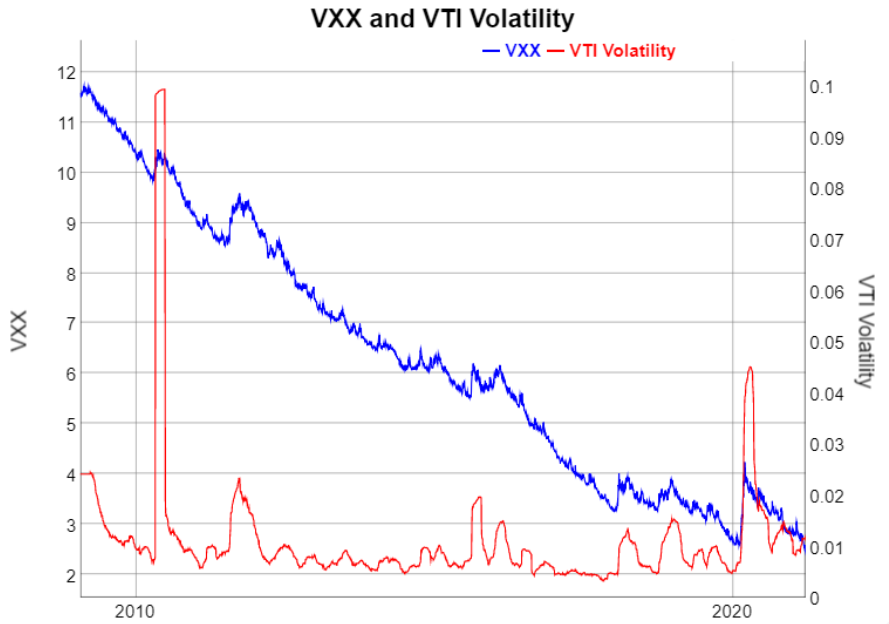
VXX Prices and the Trailing Volatility

The *VXX* ETF invests in *VIX* futures, so its price is tied to the level of the *VIX* index, with higher *VXX* prices corresponding to higher levels of the *VIX* index.

The trailing volatility of past returns moves in sympathy with the implied volatility and *VXX* prices, but with a lag.

But *VXX* prices exhibit a very strong downward trend which makes them hard to compare with the trailing volatility.

```
> # Calculate VXX log prices
> vxx <- na.omit(rutils::etfenv$prices$VXX)
> datev <- zoo::index(vxx)
> look_back <- 41
> vxx <- log(vxx)
> # Calculate trailing VTI volatility
> closep <- get("VTI", rutils::etfenv)[datev]
> closep <- log(closep)
> volat <- sqrt(HighFreq::roll_var_ohlc(ohlc=closep, look_back=look
> volat[1:look_back] <- volat[look_back+1]
```



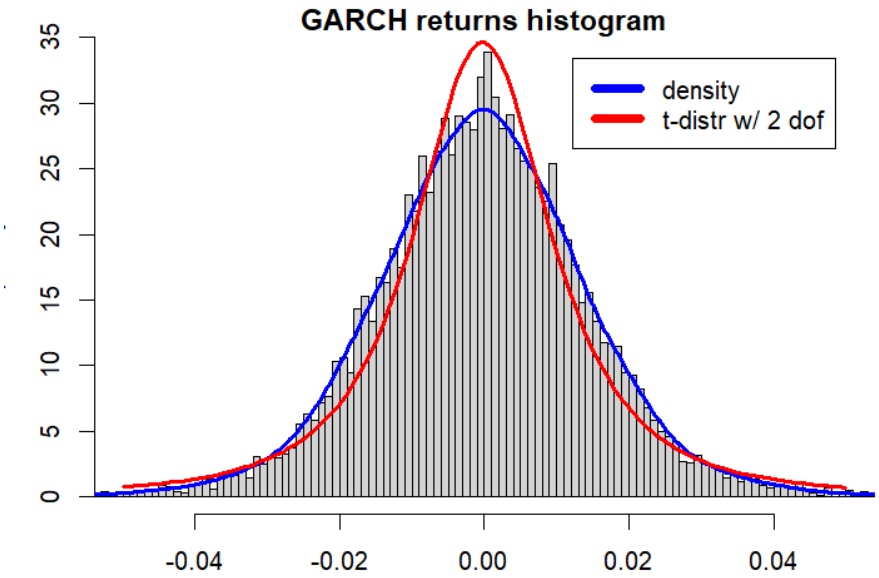

```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volat)
> colnames(datav)[2] <- "VTI Volatility"
> colnamev <- colnames(datav)
> captiont <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=captiont) %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

draft: Cointegration of VXX Prices and the trailing Volatility

The trailing volatility of past returns moves in sympathy with the implied volatility and *VXX* prices, but with a lag.

The parameter α is the weight of the squared realized returns in the variance.

Greater values of α produce a stronger feedback between the realized returns and variance, causing larger variance spikes and higher kurtosis.



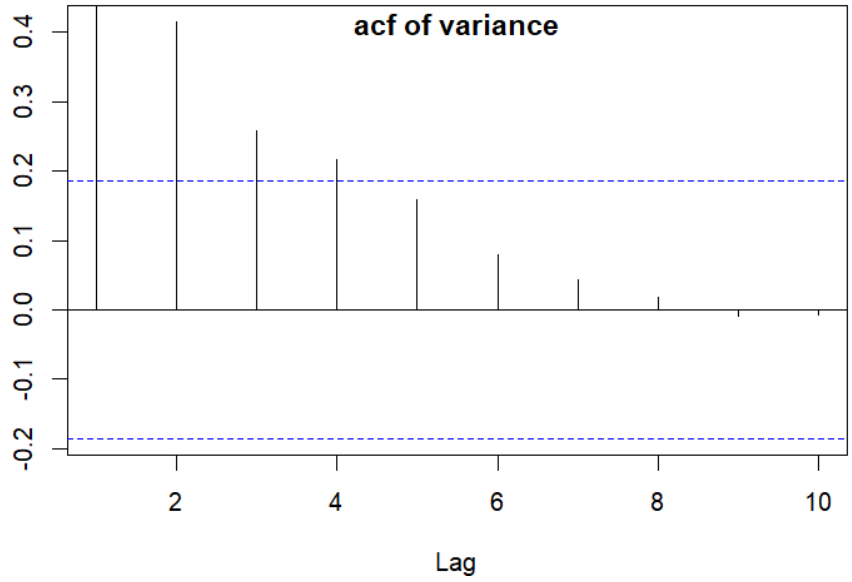

```
> # Calculate VXX log prices
> vxx <- na.omit(rutils::etfenv$prices$VXX)
> datev <- zoo::index(vxx)
> look_back <- 41
> vxx <- log(vxx)
> vxx <- (vxx - HighFreq::roll_mean(vxx, look_back=look_back))
> vxx[1:look_back] <- vxx[look_back+1]
> # Calculate trailing VTI volatility
> closep <- get("VTI", rutils::etfenv)[datev]
> closep <- log(closep)
> volat <- sqrt(HighFreq::roll_var_ohlc(ohlc=closep, look_back=look
> volat[1:look_back] <- volat[look_back+1]
> # Calculate regression coefficients of XLB ~ XLE
> betac <- drop(cov(vxx, volat)/var(volat))
> alphac <- drop(mean(vxx) - betac*mean(volat))
> # Calculate regression residuals
> fitv <- (alphac + betac*volat)
> residuals <- (vxx - fitv)
> # Perform ADF test on residuals
> tseries::adf.test(residuals, k=1)
```

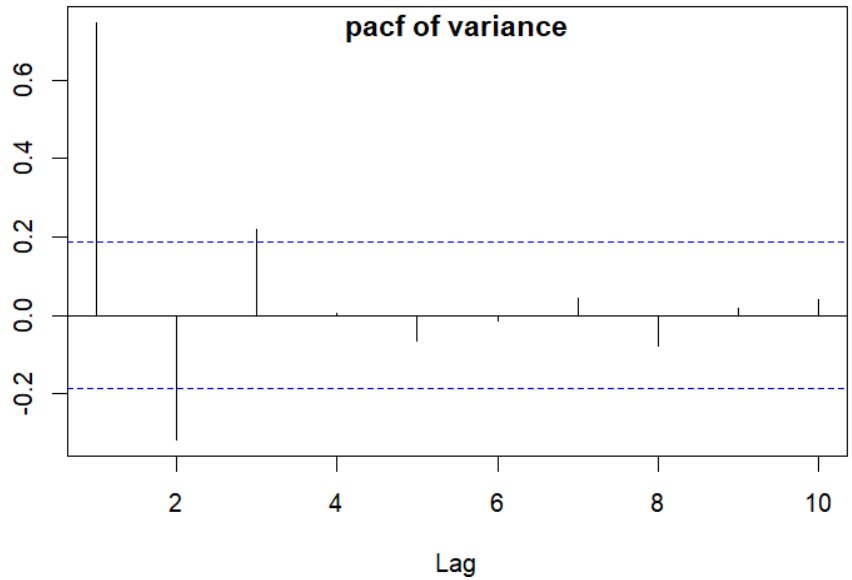
```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volat)
> colnamev <- colnames(datav)
> captiont <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=captiont) %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyLegend(show="always", width=300)
```

Autocorrelation of Volatility

Variance calculated over non-overlapping intervals has very statistically significant autocorrelations.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> # Calculate trailing VTI variance using package roll
> look_back <- 21
> varv <- HighFreq::roll_var(retp, look_back=look_back)
> colnames(varv) <- "Variance"
> # Number of look_backs that fit over returns
> nrows <- NROW(retp)
> nagg <- nrows %/% look_back
> # Define end points with beginning stub
> endd <- c(0, nrows-look_back*nagg + (0:nagg)*look_back)
> nrows <- NROW(endd)
> # Subset variance to end points
> varv <- varv[endd]
> # Plot autocorrelation function
> rutils::plot_acf(varv, lag=10, main="ACF of Variance")
> # Plot partial autocorrelation
> pacf(varv, lag=10, main="PACF of Variance", ylab=NA)
```





draft: The *ARCH* Volatility Model

The *ARCH(1,1)* is a volatility model defined by two coupled equations:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \xi_t^2$$

Where σ_t^2 is the time-dependent variance, equal to the weighted average of the point *realized* variance $(r_t - \bar{r}_t)^2$ and the past variance σ_{t-1}^2, and ξ_t are standard normal *innovations*.

The return process r_t follows a normal distribution with a time-dependent variance σ_t^2.

The parameter α is the weight associated with recent realized variance updates, and β is the weight associated with the past variance.

The long-term expected value of the variance is proportional to the parameter ω:

$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

So the sum of α plus β should be less than 1, otherwise the volatility is explosive.

```
> # Define GARCH parameters
> alphac <- 0.3; betac <- 0.5;
> omega <- 1e-4*(1 - alphac - betac)
> nrows <- 1000
> # Calculate matrix of standard normal innovations
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Res
> innov <- rnorm(nrows)
> retp <- numeric(nrows)
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> retp[1] <- sqrt(varv[1])*innov[1]
> # Simulate GARCH model
> for (i in 2:nrows) {
+   retp[i] <- sqrt(varv[i-1])*innov[i]
+   varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+ }  # end for
> # Simulate the GARCH process using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> all.equal(garch_data, cbind(retp, varv), check.attributes=FALSE)
```

The *GARCH* process must be simulated using an explicit loop, so it's better to perform it in `C++` instead of `R`.

The *GARCH* Volatility Model

The *GARCH(1,1)* is a volatility model defined by two coupled equations:

$$r_t = \sigma_{t-1}\xi_t$$
$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

Where σ_t^2 is the time-dependent variance, equal to the weighted average of the point *realized* variance r_t^2 and the past variance σ_{t-1}^2, and ξ_t are standard normal *innovations*.

The parameter α is the weight associated with recent realized variance updates, and β is the weight associated with the past variance.

The return process r_t follows a normal distribution, *conditional* on the variance in the previous period σ_{t-1}^2.

But the *unconditional* distribution of returns is *not* normal, since their standard deviation is time-dependent, so they are *leptokurtic* (fat tailed).

The long-term expected value of the variance is proportional to the parameter ω:

$$\sigma^2 = \frac{\omega}{1-\alpha-\beta}$$

So the sum of α plus β should be less than 1, otherwise the volatility is explosive.

```
> # Define GARCH parameters
> alphac <- 0.3; betac <- 0.5;
> omega <- 1e-4*(1 - alphac - betac)
> nrows <- 1000
> # Calculate matrix of standard normal innovations
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Res
> innov <- rnorm(nrows)
> retp <- numeric(nrows)
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> retp[1] <- sqrt(varv[1])*innov[1]
> # Simulate GARCH model
> for (i in 2:nrows) {
+   retp[i] <- sqrt(varv[i-1])*innov[i]
+   varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+ }  # end for
> # Simulate the GARCH process using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> all.equal(garch_data, cbind(retp, varv), check.attributes=FALSE)
```

The *GARCH* process must be simulated using an explicit loop, so it's better to perform it in `C++` instead of `R`.

GARCH Volatility Time Series

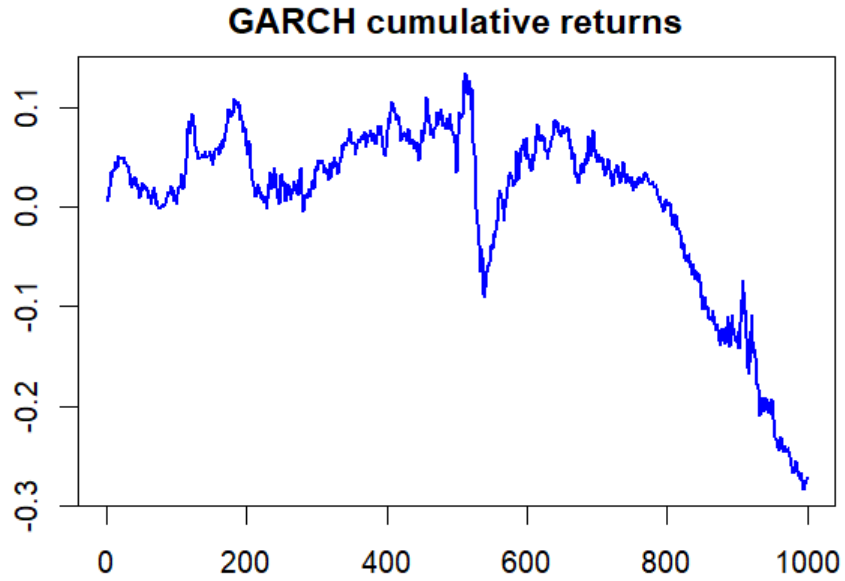
The *GARCH* volatility model produces volatility clustering - periods of high volatility followed by a quick decay.

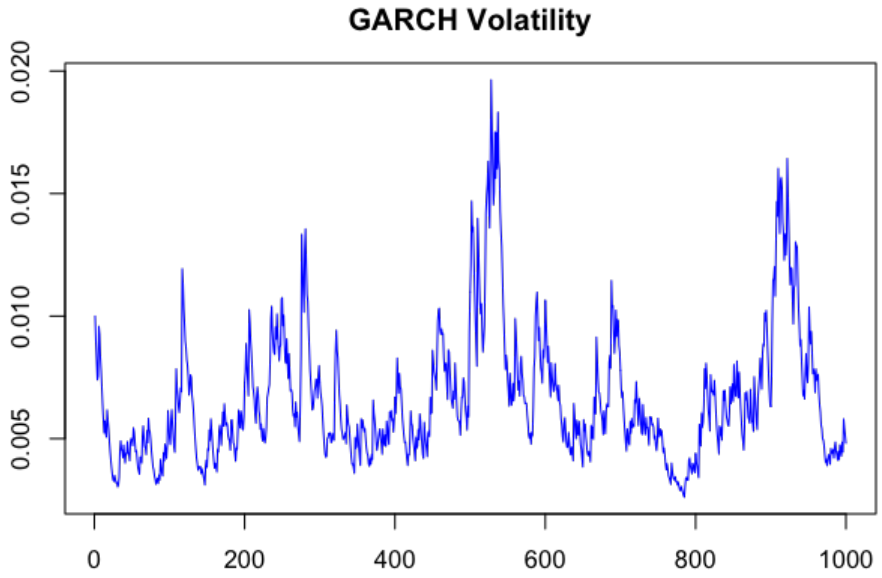
But the decay of the volatility in the *GARCH* model is faster than what is observed in practice.

The parameter α is the weight of the squared realized returns in the variance.

Larger values of α produce a stronger feedback between the realized returns and variance, which produce larger variance spikes, which produce larger kurtosis.

```
> # Open plot window on Mac
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot GARCH cumulative returns
> plot(cumsum(retp), t="l", col="blue", xlab="", ylab="",
+   main="GARCH Cumulative Returns")
> quartz.save("figure/garch_returns.png", type="png",
+   width=6, height=5)
> # Plot GARCH volatility
> plot(sqrt(varv), t="l", col="blue", xlab="", ylab="",
+   main="GARCH Volatility")
> quartz.save("figure/garch_volat.png", type="png",
+   width=6, height=5)
```





GARCH Returns Distribution

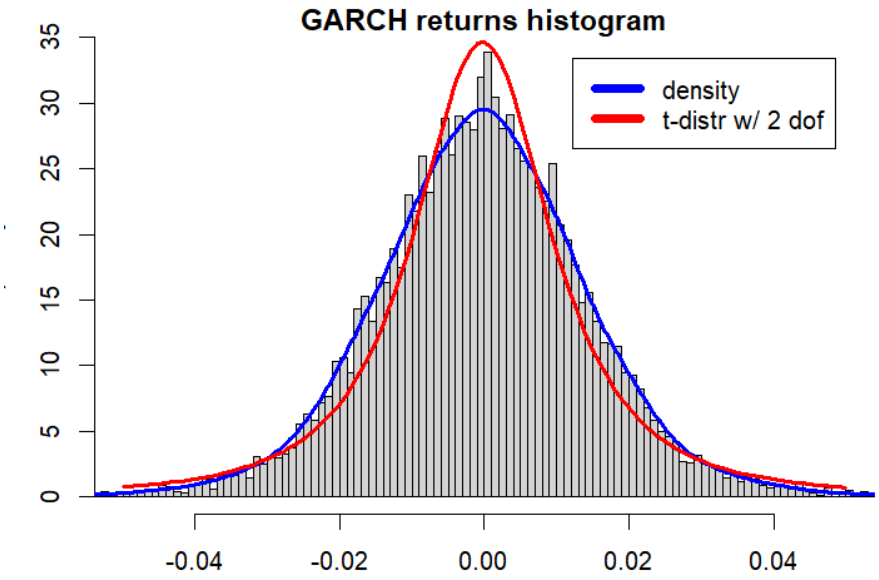
The *GARCH* volatility model produces *leptokurtic* returns with fat tails in their the distribution.

Student's *t-distribution* has fat tails, so it fits asset returns much better than the normal distribution.

Student's *t-distribution* with 3 degrees of freedom is often used to represent asset returns.

The function `fitdistr()` from package *MASS* fits a univariate distribution into a sample of data, by performing *maximum likelihood* optimization.

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.




```
> # Calculate kurtosis of GARCH returns
> mean(((retp-mean(retp))/sd(retp))^4)
> # Perform Jarque-Bera test of normality
> tseries::jarque.bera.test(retp)
> # Fit t-distribution into GARCH returns
> fitobj <- MASS::fitdistr(retp, densfun="t", df=2)
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
```

```
> # Plot histogram of GARCH returns
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.03, 0.03),
+   ylab="frequency", freq=FALSE, main="GARCH Returns Histogram")
> lines(density(retp, adjust=1.5), lwd=2, col="blue")
> curve(expr=dt((x-locv)/scalev, df=2)/scalev,
+   type="l", xlab="", ylab="", lwd=2,
+   col="red", add=TRUE)
> legend("topright", inset=-0, bty="n", y.intersp=0.4,
+  leg=c("density", "t-distr w/ 2 dof"),
+  lwd=6, lty=1, col=c("blue", "red"))
> quartz.save("figure/garch_hist.png", type="png", width=6, height=5
```

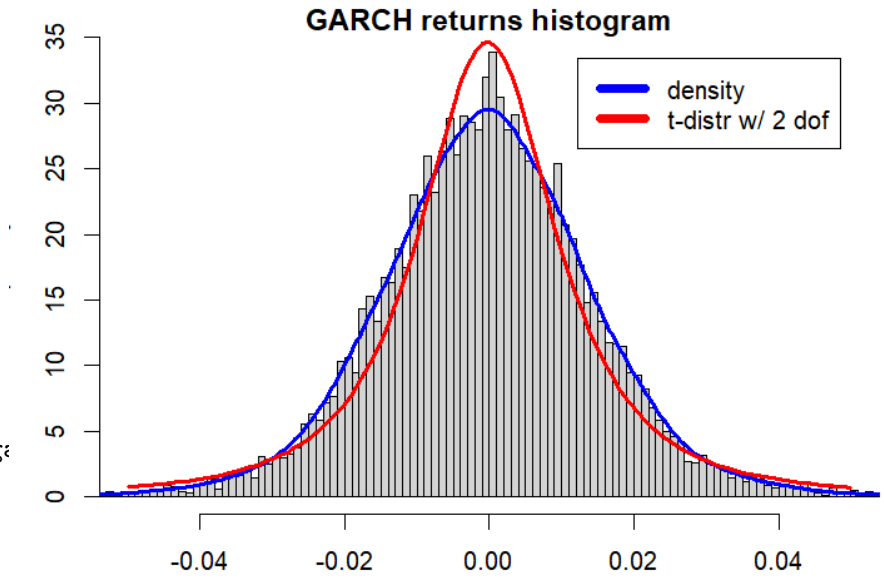
GARCH Model Simulation

The package *fGarch* contains functions for applying *GARCH* models.

The function `fGarch::garchSpec()` specifies a *GARCH* model.

The function `fGarch::garchSim()` simulates a *GARCH* model, but it uses its own random innovations, so its output is not reproducible.

```
> # Specify GARCH model
> garch_spec <- fGarch::garchSpec(model=list(ar=c(0, 0), omega=omega
+   alpha=alphac, beta=betac))
> # Simulate GARCH model
> garch_sim <- fGarch::garchSim(spec=garch_spec, n=nrows)
> retp <- as.numeric(garch_sim)
> # Calculate kurtosis of GARCH returns
> moments::moment(retp, order=4) /
+   moments::moment(retp, order=2)^2
> # Perform Jarque-Bera test of normality
> tseries::jarque.bera.test(retp)
> # Plot histogram of GARCH returns
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.05, 0.05),
+   ylab="frequency", freq=FALSE,
+   main="GARCH Returns Histogram")
> lines(density(retp, adjust=1.5), lwd=3, col="blue")
```




```
> # Fit t-distribution into GARCH returns
> fitobj <- MASS::fitdistr(retp, densfun="t", df=2, lower=c(-1, 1e-7
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
> curve(expr=dt((x-locv)/scalev, df=2)/scalev,
+   type="l", xlab="", ylab="", lwd=3,
+   col="red", add=TRUE)
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+  leg=c("density", "t-distr w/ 2 dof"),
+  lwd=6, lty=1, col=c("blue", "red"))
```

GARCH Returns Kurtosis

The expected value of the variance σ^2 of *GARCH* returns is proportional to the parameter ω:

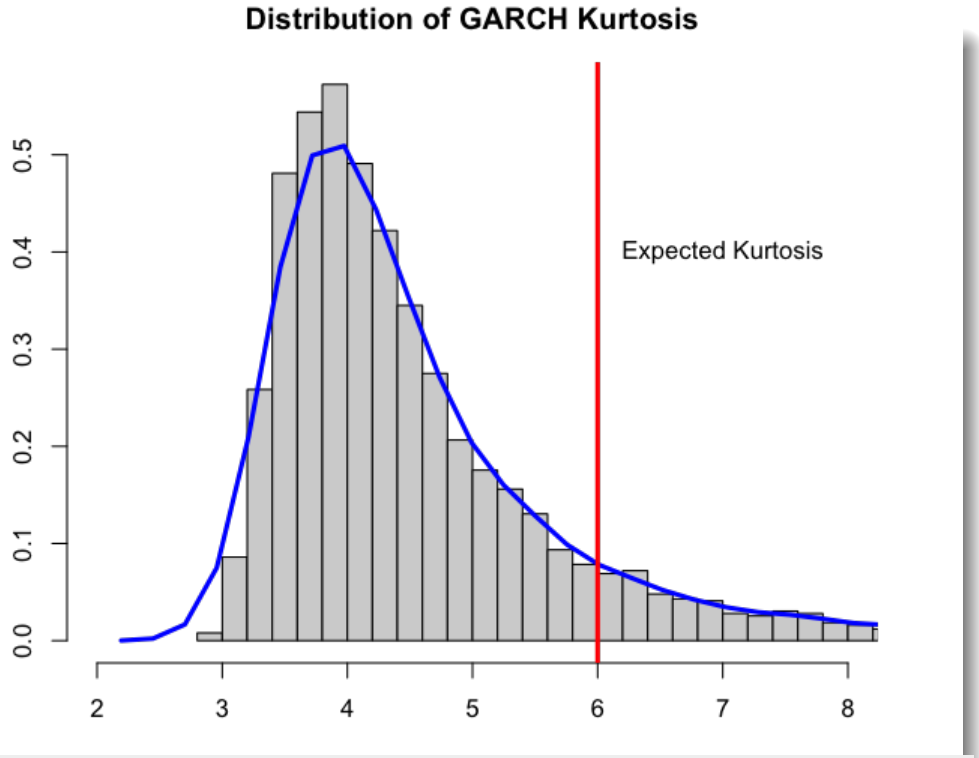
$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

The expected value of the kurtosis κ of *GARCH* returns is equal to:

$$\kappa = 3 + \frac{6\alpha^2}{1 - 2\alpha^2 - (\alpha + \beta)^2}$$

The excess kurtosis $\kappa - 3$ is proportional to α^2 because larger values of the parameter α produce larger variance spikes which produce larger kurtosis.

The distribution of kurtosis is highly positively skewed, especially for short returns samples, so most kurtosis values will be significantly below their expected value.




```
> # Calculate variance of GARCH returns
> var(retp)
> # Calculate expected value of variance
> omega/(1 - alphac - betac)
> # Calculate kurtosis of GARCH returns
> mean(((retp-mean(retp))/sd(retp))^4)
> # Calculate expected value of kurtosis
> 3 + 6*alpha^2/(1-2*alpha^2-(alphac+betac)^2)
```

```
> # Calculate the distribution of GARCH kurtosis
> kurt <- sapply(1:1e4, function(x) {
+   garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+     beta=betac, innov=matrix(rnorm(nrows)))
+   retp <- garch_data[, 1]
+   c(var(retp), mean(((retp-mean(retp))/sd(retp))^4))
+ })  # end sapply
> kurt <- t(kurt)
> apply(kurt, 2, mean)
> # Plot the distribution of GARCH kurtosis
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> histp <- hist(kurt[, 2], breaks=500, col="lightgrey",
+   xlim=c(2, 8), xlab="returns", ylab="frequency", freq=FALSE,
+   main="Distribution of GARCH Kurtosis")
```

GARCH Variance Estimation

The *GARCH* model can be used to estimate the trailing variance of empirical (historical) returns.

If the time series of returns r_t is given, then it can be used in the *GARCH(1,1)* formula to estimate the trailing variance σ_t^2:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

If the returns from the *GARCH(1,1)* simulation are used in the above formula, then it produces the simulated *GARCH(1,1)* variance.

But to estimate the trailing variance of historical returns, the parameters ω, α, and β must be estimated through model calibration.

```
> # Simulate the GARCH process using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> # Extract the returns
> retp <- garch_data[, 1]
> # Estimate the trailing variance from the returns
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> for (i in 2:nrows) {
+   varv[i] <- omega + alphac*retp[i]^2 +
+     betac*varv[i-1]
+ }  # end for
> all.equal(garch_data[, 2], varv, check.attributes=FALSE)
```

GARCH Model Calibration

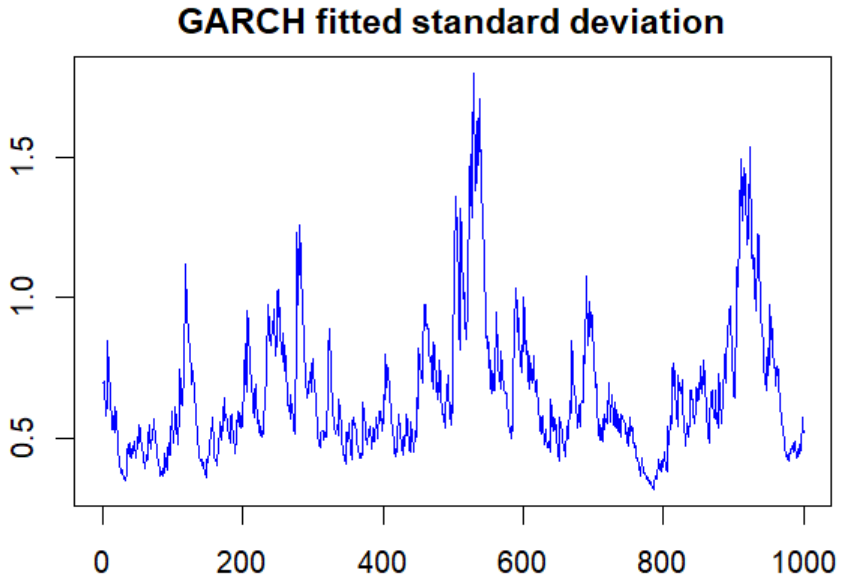
GARCH models can be calibrated from the returns using the *maximum-likelihood* method.

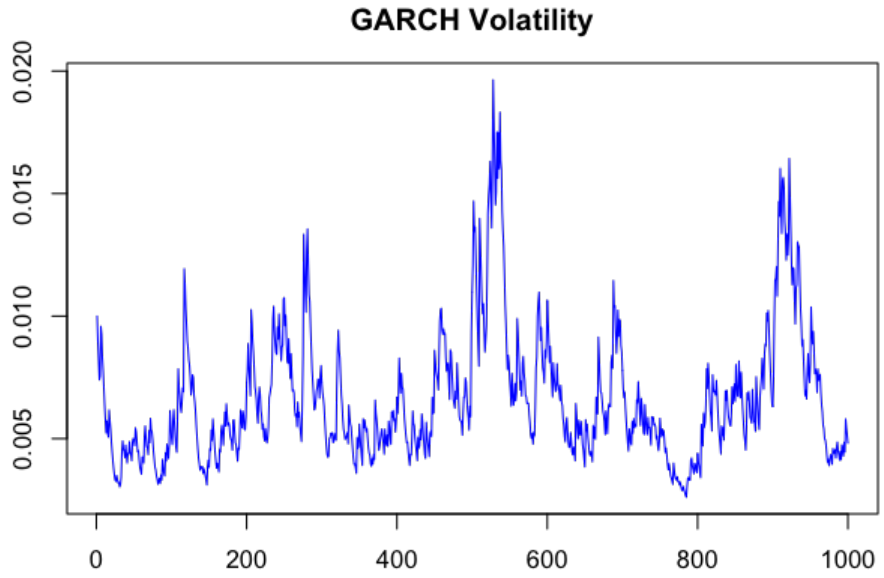
But it's a complex optimization procedure which requires a large amount of data for accurate results.

The function `fGarch::garchFit()` calibrates a *GARCH* model on a time series of returns.

The function `garchFit()` returns an `S4` object of class *fGARCH*, with multiple slots containing the *GARCH* model outputs and diagnostic information.

```
> library(fGarch)
> # Fit returns into GARCH
> garch_fit <- fGarch::garchFit(data=retp)
> # Fitted GARCH parameters
> garch_fit@fit$coef
> # Actual GARCH parameters
> c(mu=mean(retp), omega=omega,alpha=alphac, beta=betac)
> # Plot GARCH fitted volatility
> plot(sqrt(garch_fit@fit$series$h), t="l",
+   col="blue", xlab="", ylab="",
+   main="GARCH Fitted Volatility")
> quartz.save("figure/garch_fGarch_fitted.png",
+   type="png", width=6, height=5)
```





GARCH Likelihood Function

Under the *GARCH(1,1)* volatility model, the returns follow the process: $r_t = \sigma_{t-1}\xi_t$. (We can assume that the returns have been centered.)

So the *conditional* distribution of returns is normal with standard deviation equal to σ_{t-1}:

$$\phi(r_t, \sigma_{t-1}) = \frac{e^{-r_t^2/2\sigma_{t-1}^2}}{\sqrt{2\pi}\sigma_{t-1}}$$

The *log-likelihood* function $\mathcal{L}(\omega, \alpha, \beta|r_t)$ for the normally distributed returns is therefore equal to:

$$\mathcal{L}(\omega, \alpha, \beta|r_t) = -\sum_{t=1}^{n}(\frac{r_t^2}{\sigma_{t-1}^2} + \log(\sigma_{t-1}^2))$$

The *log-likelihood* depends on the *GARCH(1,1)* parameters ω, α, and β because the trailing variance σ_t^2 depends on the *GARCH(1,1)* parameters:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

The *GARCH* process must be simulated using an explicit loop, so it's better to perform it in `C++` instead of `R`.

```
> # Define likelihood function
> likefun <- function(omega, alphac, betac) {
+   # Estimate the trailing variance from the returns
+   varv <- numeric(nrows)
+   varv[1] <- omega/(1 - alphac - betac)
+   for (i in 2:nrows) {
+     varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+   }  # end for
+   varv <- ifelse(varv > 0, varv, 0.000001)
+   # Lag the variance
+   varv <- rutils::lagit(varv, pad_zeros=FALSE)
+   # Calculate the likelihood
+   -sum(retp^2/varv + log(varv))
+ }  # end likefun
> # Calculate the likelihood in R
> likefun(omega, alphac, betac)
> # Calculate the likelihood in Rcpp
> HighFreq::lik_garch(omega=omega, alpha=alphac,
+   beta=betac, returns=matrix(retp))
> # Benchmark speed of likelihood calculations
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode=likefun(omega, alphac, betac),
+   Rcpp=HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac,
+   ), times=10)[, c(1, 4, 5)]
```

GARCH Likelihood Function Matrix

The *GARCH(1,1) log-likelihood* function depends on three parameters $\mathcal{L}(\omega, \alpha, \beta | r_t)$.

The more parameters the harder it is to find their optimal values using optimization.

We can simplify the optimization task by assuming that the expected variance is equal to the realized variance:

$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta} = \frac{1}{n-1} \sum_{t=1}^{n} (r_t - \bar{r})^2$$

This way the *log-likelihood* becomes a function of only two parameters, say α and β.

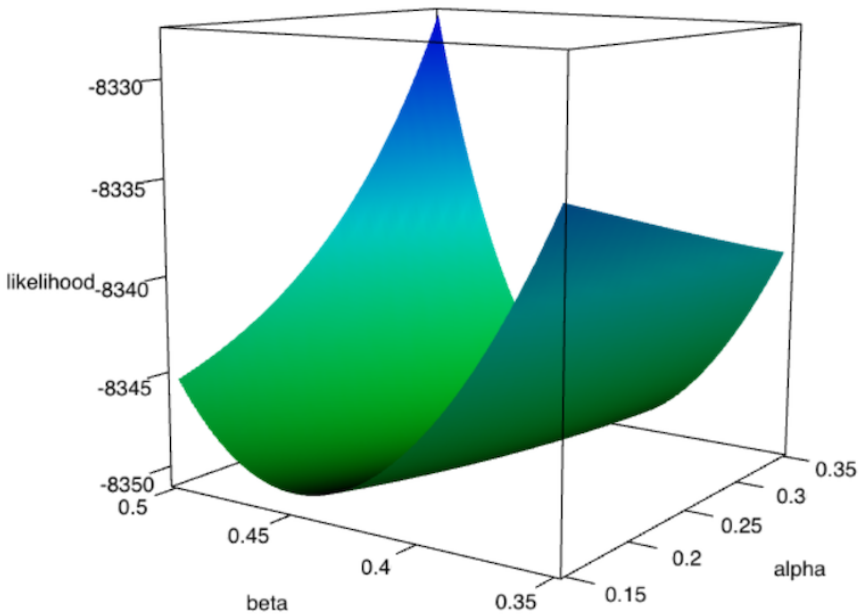
```
> # Calculate the variance of returns
> retp <- garch_data[, 1, drop=FALSE]
> varv <- var(retp)
> retp <- (retp - mean(retp))
> # Calculate likelihood as function of alpha and betac parameters
> likefun <- function(alphac, betac) {
+   omega <- variance*(1 - alpha - betac)
+   -HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac, retu
+ }  # end likefun
> # Calculate matrix of likelihood values
> alphas <- seq(from=0.15, to=0.35, len=50)
> betac <- seq(from=0.35, to=0.5, len=50)
> likmat <- sapply(alphacs, function(alphac) sapply(betac,
+   function(betac) likefun(alphac, betac)))
```

GARCH Likelihood Perspective Plot

The perspective plot shows that the *log-likelihood* is much more sensitive to the β parameter than to α.

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a *vectorized* function or a matrix.

The optimal values of α and β can be found approximately using a grid search on the *log-likelihood* matrix.




```
> # Set rgl options and load package rgl
> options(rgl.useNULL=TRUE); library(rgl)
> # Draw and render 3d surface plot of likelihood function
> ncols <- 100
> color <- rainbow(ncols, start=2/6, end=4/6)
> zcols <- cut(likmat, ncols)
> rgl::persp3d(alphacs, betac, likmat, col=color[zcols],
+   xlab="alpha", ylab="beta", zlab="likelihood")
> rgl::rglwidget(elementId="plot3drgl", width=700, height=700)
> # Perform grid search
> coord <- which(likmat == min(likmat), arr.ind=TRUE)
> c(alphacs[coord[2]], betac[coord[1]])
> likmat[coord]
> likefun(alphacs[coord[2]], betac[coord[1]])
> # Optimal and actual parameters
> options(scipen=2)  # Use fixed not scientific notation
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+   optimal=c(alphacs[coord[2]], betac[coord[1]], variance*(1 - sum(alphacs[coord[2]], betac[coord[1]]))))
```

GARCH Likelihood Function Optimization

The flat shape of the *GARCH* likelihood function makes it difficult for steepest descent optimizers to find the best parameters.

The function `DEoptim()` from package *DEoptim* performs *global* optimization using the *Differential Evolution* algorithm.

Differential Evolution is a genetic algorithm which evolves a population of solutions over several generations:

https://link.springer.com/content/pdf/10.1023/A:1008202821328.pdf

The first generation of solutions is selected randomly.

Each new generation is obtained by combining the best solutions from the previous generation.

The *Differential Evolution* algorithm is well suited for very large multi-dimensional optimization problems, such as portfolio optimization.

Gradient optimization methods are more efficient than *Differential Evolution* for smooth objective functions with no local minima.

```
> # Define vectorized likelihood function
> likefun <- function(x, retp) {
+   alphac <- x[1]; betac <- x[2]; omega <- x[3]
+   -HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac, retu
+ }  # end likefun
> # Initial parameters
> initp <- c(alphac=0.2, beta=0.4, omega=varv/0.2)
> # Find max likelihood parameters using steepest descent optimizer
> fitobj <- optim(par=initp,
+   fn=likefun, # Log-likelihood function
+   method="L-BFGS-B", # Quasi-Newton method
+   returns=retp,
+   upper=c(0.35, 0.55, varv), # Upper constraint
+   lower=c(0.15, 0.35, varv/100)) # Lower constraint
> # Optimal and actual parameters
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+ optimal=c(fitobj$par["alpha"], fitobj$par["beta"], fitobj$par["ome
> # Find max likelihood parameters using DEoptim
> optiml <- DEoptim::DEoptim(fn=likefun,
+   upper=c(0.35, 0.55, varv), # Upper constraint
+   lower=c(0.15, 0.35, varv/100), # Lower constraint
+   returns=retp,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal and actual parameters
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+ optimal=c(optiml$optim$bestmem[1], optiml$optim$bestmem[2], optiml
```

GARCH Variance of Stock Returns

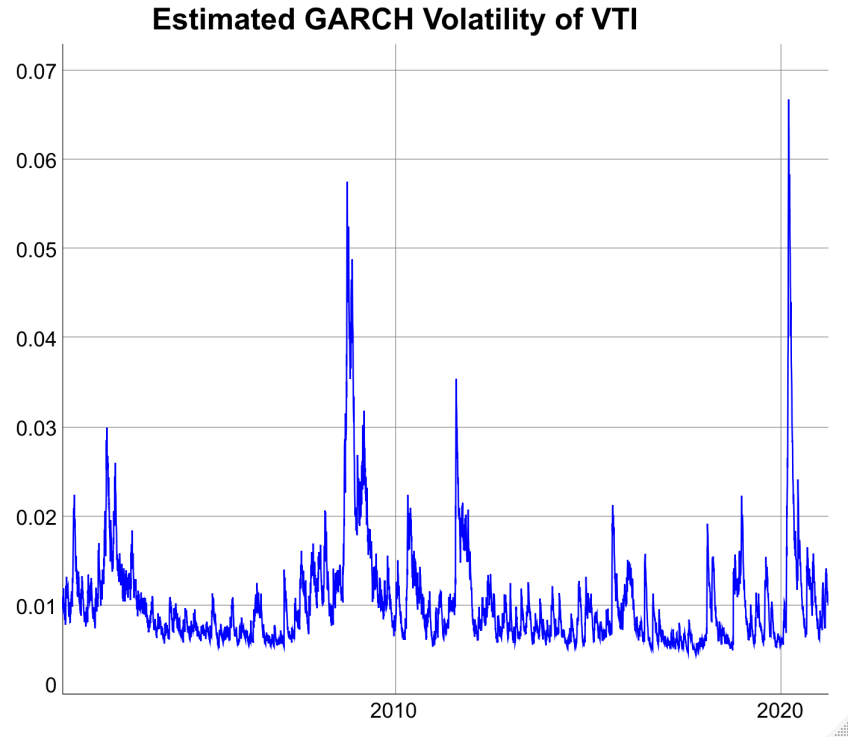
The *GARCH* model can be used to estimate the trailing variance of empirical (historical) returns.

If the time series of returns r_t is given, then it can be used in the *GARCH(1,1)* formula to estimate the trailing variance σ_t^2:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

The *GARCH* formula can be viewed as a generalization of the *EMA* trailing variance.

```
> # Calculate VTI returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> # Find max likelihood parameters using DEoptim
> optiml <- DEoptim::DEoptim(fn=likefun,
+   upper=c(0.4, 0.9, varv), # Upper constraint
+   lower=c(0.1, 0.5, varv/100), # Lower constraint
+   returns=retp,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal parameters
> par_am <- unname(optiml$optim$bestmem)
> alphac <- par_am[1]; betac <- par_am[2]; omega <- par_am[3]
> c(alphac, betac, omega)
> # Equilibrium GARCH variance
> omega/(1 - alphac - betac)
> drop(var(retp))
```




```
> # Estimate the GARCH volatility of VTI returns
> nrows <- NROW(retp)
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> for (i in 2:nrows) {
+   varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+ }  # end for
> # Estimate the GARCH volatility using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=retp, is_random=FALSE)
> all.equal(garch_data[, 2], varv, check.attributes=FALSE)
> # Plot dygraph of the estimated GARCH volatility
> dygraphs::dygraph(xts::xts(sqrt(varv), zoo::index(retp)),
+   main="Estimated GARCH Volatility of VTI") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
```

GARCH Variance Forecasts

The one-step-ahead forecast of the squared returns is equal to their expected value: $r_{t+1}^2 = \mathbb{E}[(\sigma_t \xi_t)^2] = \sigma_t^2$, since $\mathbb{E}[\xi_t^2] = 1$.

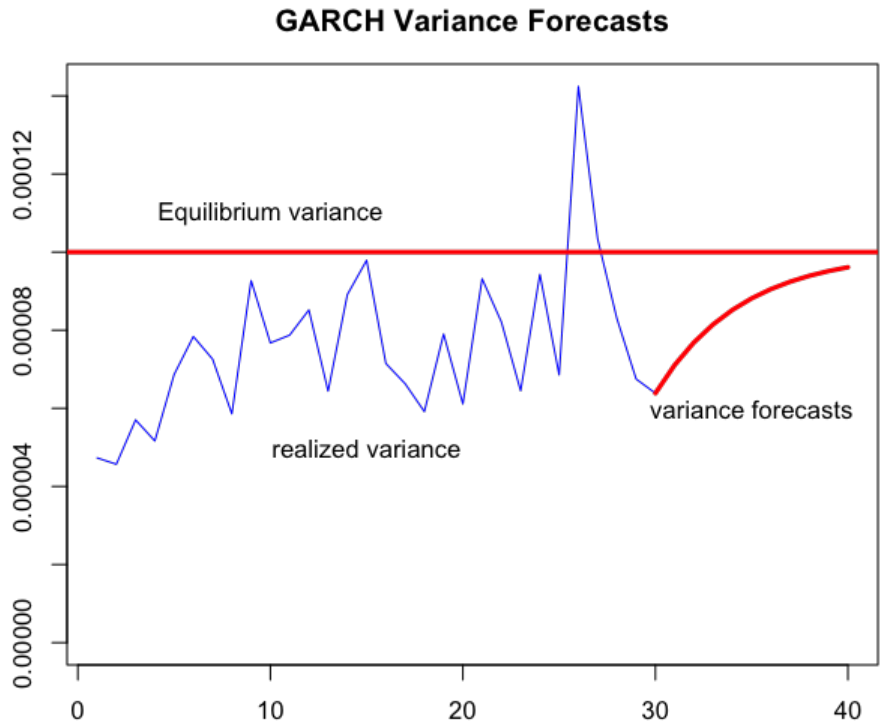
So the variance forecasts depend on the variance in the previous period:
$\sigma_{t+1}^2 = \mathbb{E}[\omega + \alpha r_{t+1}^2 + \beta \sigma_t^2] = \omega + (\alpha + \beta)\sigma_t^2$

The variance forecasts gradually settles to the equilibrium value σ^2, such that the forecast is equal to itself: $\sigma^2 = \omega + (\alpha + \beta)\sigma^2$.

This gives: $\sigma^2 = \frac{\omega}{1-\alpha-\beta}$, which is the long-term expected value of the variance.

So the variance forecasts decay exponentially to their equilibrium value σ^2 at the decay rate equal to $(\alpha + \beta)$:

$$\sigma_{t+1}^2 - \sigma^2 = (\alpha + \beta)(\sigma_t^2 - \sigma^2)$$




```
> # Simulate GARCH model
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> varv <- garch_data[, 2]
> # Calculate the equilibrium variance
> vareq <- omega/(1 - alphac - betac)
> # Calculate the variance forecasts
> varf <- numeric(10)
> varf[1] <- vareq + (alphac + betac)*(xts::last(varv) - vareq)
> for (i in 2:10) {
+   varf[i] <- vareq + (alphac + betac)*(varf[i-1] - vareq)
+ }  # end for
```

```
> # Open plot window on Mac
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot GARCH variance forecasts
> plot(tail(varv, 30), t="l", col="blue", xlab="", ylab="",
+   xlim=c(1, 40), ylim=c(0, max(tail(varv, 30))),
+   main="GARCH Variance Forecasts")
> text(x=15, y=0.5*vareq, "realized variance")
> lines(x=30:40, y=c(xts::last(varv), varf), col="red", lwd=3)
> text(x=35, y=0.6*vareq, "variance forecasts")
> abline(h=vareq, lwd=3, col="red")
> text(x=10, y=1.1*vareq, "Equilibrium variance")
> quartz.save("figure/garch_forecast.png", type="png",
+   width=6, height=5)
```

depr: old stuff about Estimating Volatility of Intraday Time Series

The *close-to-close* estimator depends on *Close* prices specified over the aggregation intervals:

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\log(\frac{C_i}{C_{i-1}}) - \bar{r})^2$$
$$\bar{r} = \frac{1}{n}\sum_{i=1}^{n}\log(\frac{C_i}{C_{i-1}})$$

Volatility estimates for intraday time series depend both on the units of returns (per second, minute, day, etc.), and on the aggregation interval (secondly, minutely, daily, etc.)

A minutely time interval is equal to 60 seconds, a daily time interval is equal to 24*60*60 = 86,400 seconds.

For example, it's possible to measure returns in minutely intervals in units per second.

The estimated volatility is directly proportional to the measurement units.

For example, the volatility estimated from per minute returns is 60 times the volatility estimated from per second returns.

```
> library(HighFreq)  # Load HighFreq
> # Minutely SPY returns (unit per minute) single day
> retspy <- rutils::diffit(log(SPY["2012-02-13", 4]))
> # Minutely SPY volatility (unit per minute)
> sd(retspy)
> # Divide minutely SPY returns by time intervals (unit per second)
> retspy <- retspy/rutils::diffit(xts::.index(SPY["2012-02-13"]))
> retspy[1] <- 0
> # Minutely SPY volatility scaled to unit per minute
> 60*sd(retspy)
> # SPY returns multiple days
> retspy <- rutils::diffit(log(SPY[, 4]))
> # Minutely SPY volatility (includes overnight jumps)
> sd(retspy)
> # Table of intervals - 60 second is most frequent
> indeks <- rutils::diffit(xts::.index(SPY))
> table(indeks)
> # hist(indeks)
> # SPY returns with overnight scaling (unit per second)
> retspy <- retspy/indeks
> retspy[1] <- 0
> # Minutely SPY volatility scaled to unit per minute
> 60*sd(retspy)
```

draft: Comparing Range Volatility

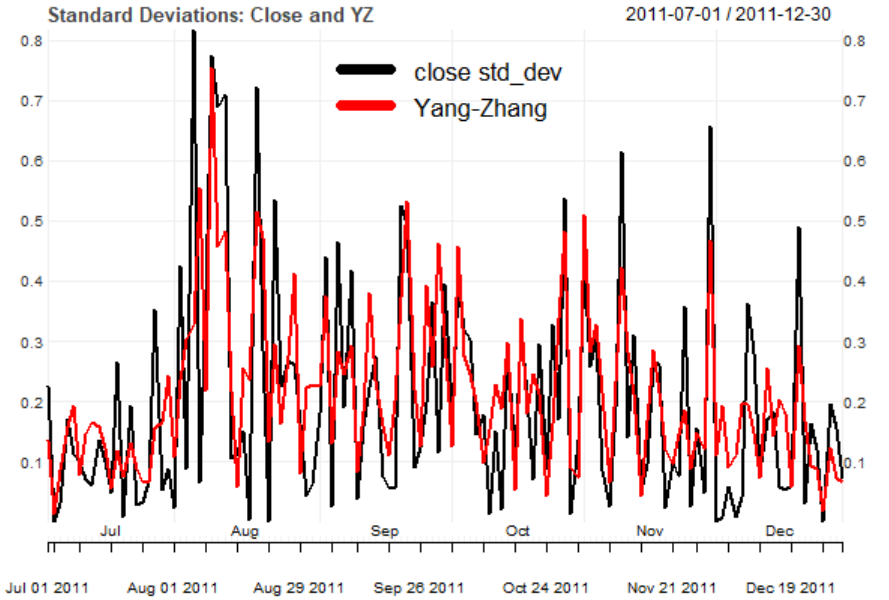
The range volatility estimators have much lower variability (standard errors) than the standard *Close-to-Close* estimator.

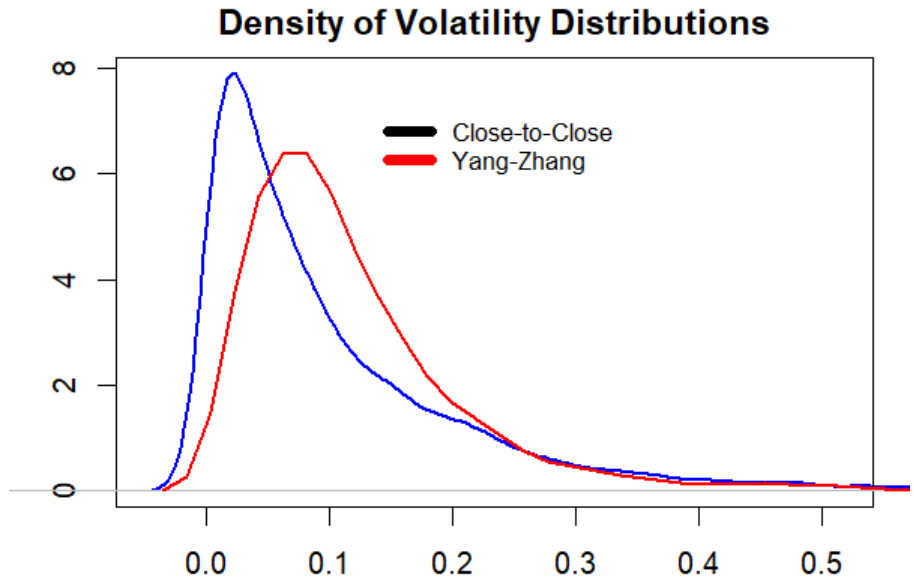
Is the above correct? Because the plot shows otherwise.

The range volatility estimators follow the standard *Close-to-Close* estimator, except in intervals of high intra-period volatility.

During the May 6, 2010 *flash crash*, range volatility spiked more than the *Close-to-Close* volatility.

```
> library(HighFreq)  # Load HighFreq
> ohlc <- log(rutils::etfenv$VTI)
> # Calculate variance
> varcl <- HighFreq::run_variance(ohlc=ohlc,
+   method="close")
> var_yang_zhang <- HighFreq::run_variance(ohlc=ohlc)
> stdev <- 24*60*60*sqrt(252*cbind(varcl, var_yang_zhang))
> colnames(stdev) <- c("close stdev", "Yang-Zhang")
> # Plot the time series of volatility
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> quantmod::chart_Series(stdev["2011-07/2011-12"],
+   theme=plot_theme, name="Standard Deviations: Close and YZ")
> legend("top", legend=colnames(stdev), y.intersp=0.4,
+  bg="white", lty=1, lwd=6, inset=0.1, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
> # Plot volatility around 2010 flash crash
> quantmod::chart_Series(stdev["2010-04/2010-06"],
+   theme=plot_theme, name="Volatility Around 2010 Flash Crash")
> legend("top", legend=colnames(stdev), y.intersp=0.4,
+  bg="white", lty=1, lwd=6, inset=0.1, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
> # Plot density of volatility distributions
```





draft: *Log-range* Volatility Proxies

To-do: plot time series of *intra-day range* volatility estimator and standard close-to-close volatility estimator. Emphasize flash-crash of 2010.

An alternative range volatility estimator can be created by calculating the logarithm of the range, (as opposed to the range percentage, or the logarithm of the price ratios).

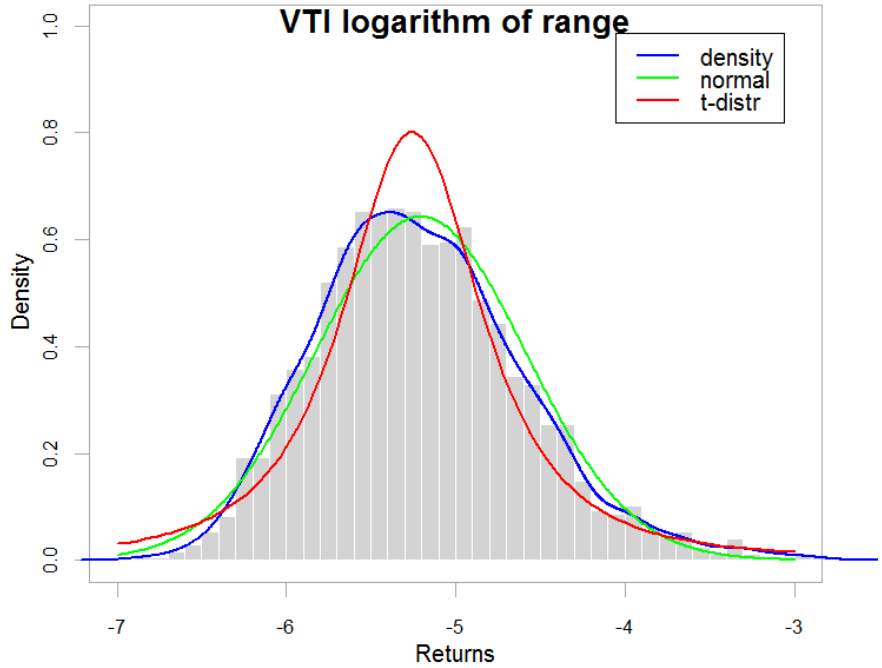
To-do: plot scatterplot of *intra-day range* volatility estimator and standard close-to-close volatility estimator.

Emphasize the two are different: the intra-day range volatility estimator captures volatility events which aren't captured by close-to-close volatility estimator, and vice versa.

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} \log(\frac{H_i - L_i}{H_i + L_i})^2$$

The range logarithm fits better into the normal distribution than the range percentage.

```
> ohlc <- rutils::etfenv$VTI
> retp <- log((ohlc[, 2] - ohlc[, 3]) / (ohlc[, 2] + ohlc[, 3]))
> foo <- rutils::diffit(log(ohlc[, 4]))
> plot(as.numeric(foo)^2, as.numeric(retp)^2)
> bar <- lm(retp ~ foo)
> summary(bar)
>
>
> # Perform normality tests
```

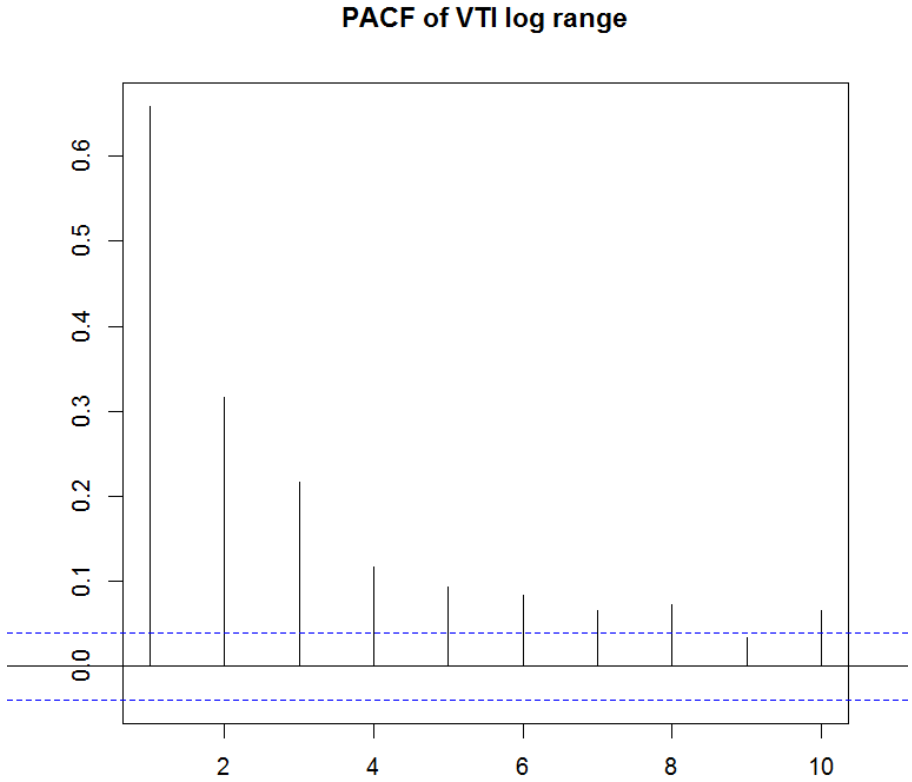



```
> # Plot histogram of VTI returns
> colorv <- c("lightgray", "blue", "green", "red")
> PerformanceAnalytics::chart.Histogram(retp,
+   main="", xlim=c(-7, -3), col=colorv[1:3],
+   methods = c("add.density", "add.normal"))
> curve(expr=dt((x-fitobj$estimate[1])/
+   fitobj$estimate[2], df=2)/fitobj$estimate[2],
+ type="l", xlab="", ylab="", lwd=2,
+ col=colorv[4], add=TRUE)
> # Add title and legend
> title(main="VTI logarithm of range",
+ cex.main=1.3, line=-1)
> legend("topright", inset=0.05, y.intersp=0.4,
+   legend=c("density", "normal", "t-distr"),
+   lwd=6, lty=1, col=colorv[2:4], bty="n")
```

draft: Autocorrelations of Alternative *Range* Estimators

The logarithm of the range exhibits very significant autocorrelations, unlike the range percentage.

```
> # Calculate VTI range variance partial autocorrelations
> pacf(retp^2, lag=10, xlab=NA, ylab=NA,
+      main="PACF of VTI log range")
> quantmod::chart_Series(retp^2, name="VTI log of range squared")
```



depr: Standard Errors of Volatility Estimators Using Bootstrap

The standard errors of estimators can be calculated using a *bootstrap* simulation.

The *bootstrap* procedure generates new data by randomly sampling with replacement from the observed data set.

The *bootstrapped* data is then used to recalculate the estimator many times, producing a vector of values.

The *bootstrapped* estimator values can then be used to calculate the probability distribution of the estimator and its standard error.

Bootstrapping doesn't provide accurate estimates for estimators that are sensitive to the ordering and correlations in the data.

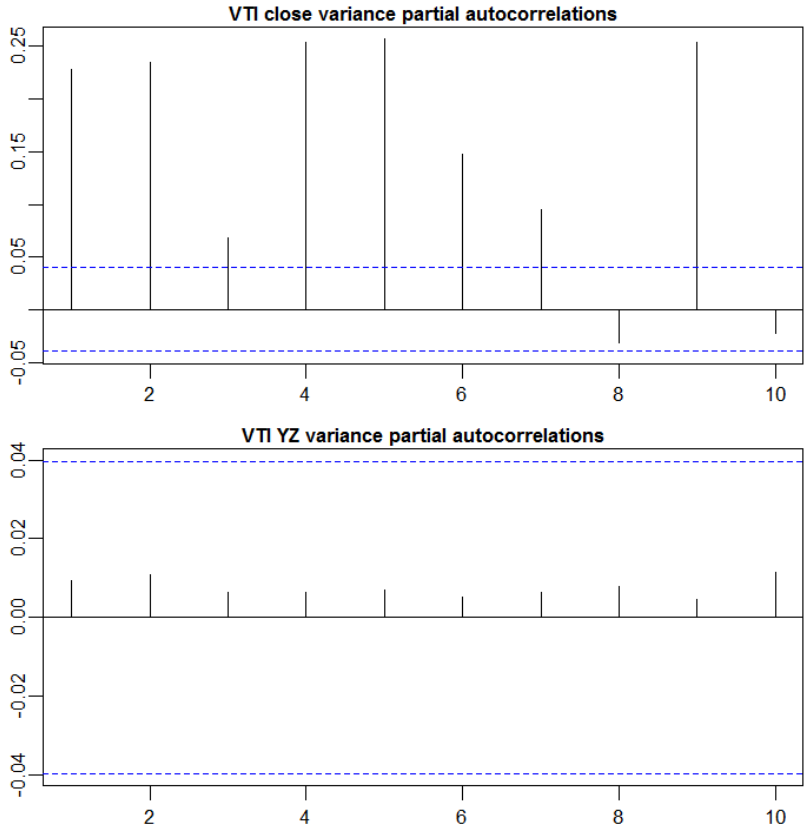
```
> # Standard errors of variance estimators using bootstrap
> bootd <- sapply(1:1e2, function(x) {
+   # Create random OHLC
+   ohlc <- HighFreq::random_ohlc()
+   # Calculate variance estimate
+   c(var=var(ohlc[, 4]),
+     yang_zhang=HighFreq::calcvariance(
+ ohlc, method="yang_zhang", scalev=FALSE))
+ })  # end sapply
> # Analyze bootstrapped variance
> bootd <- t(bootd)
> head(bootd)
> colMeans(bootd)
> apply(bootd, MARGIN=2, sd) /
+   colMeans(bootd)
```

draft: Autocorrelations of *Close-to-Close* and *Range* Variances

The standard *Close-to-Close* estimator exhibits very significant autocorrelations, but the *range* estimators are not autocorrelated.

That is because the time series of squared intra-period ranges is not autocorrelated.

```
> # Close variance estimator partial autocorrelations
> pacf(varcl, lag=10, xlab=NA, ylab=NA)
> title(main="VTI close variance partial autocorrelations")
>
> # Range variance estimator partial autocorrelations
> pacf(var_yang_zhang, lag=10, xlab=NA, ylab=NA)
> title(main="VTI YZ variance partial autocorrelations")
>
> # Squared range partial autocorrelations
> retp <- log(rutils::etfenv$VTI[,2] /
+             rutils::etfenv$VTI[,3])
> pacf(retp^2, lag=10, xlab=NA, ylab=NA)
> title(main="VTI squared range partial autocorrelations")
```



Defining Look-back Time Intervals

A time *period* is the time between two neighboring points in time.

A time *interval* is the time spanned by one or more time *periods*.

A *look-back interval* is a time *interval* for performing aggregations over the past, starting from a *start point* and ending at an *end point*.

The *start points* are the *end points* lagged by the *look-back interval*.

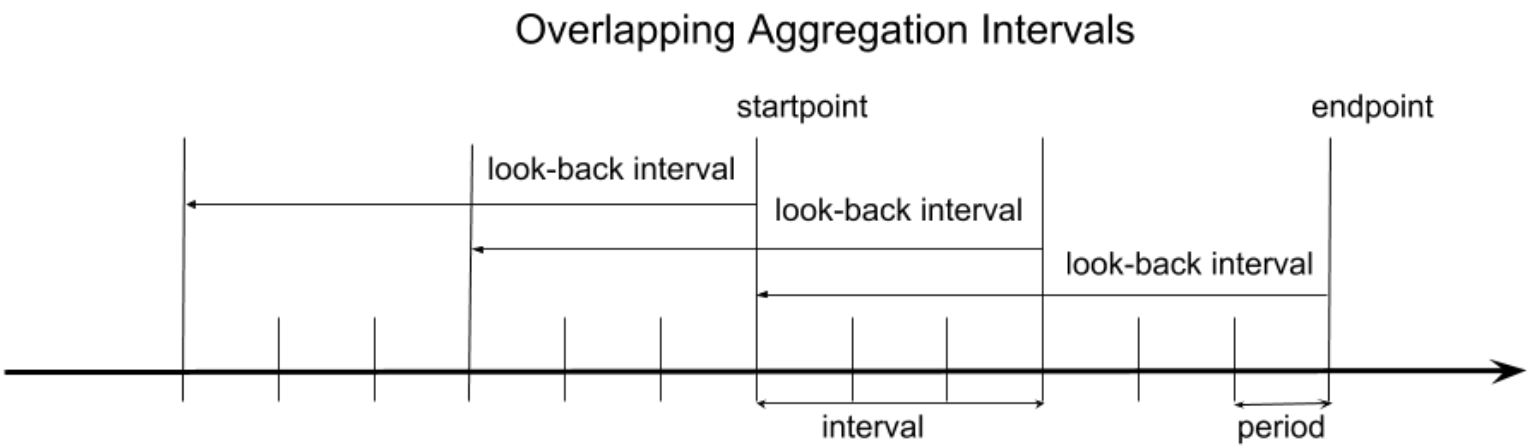
The look-back *intervals* may or may not *overlap* with their neighboring intervals.

A *trailing aggregation* is performed over a vector of *end points* in time.

An example of a trailing aggregation are moving average prices.

An *interval aggregation* is specified by *end points* separated by many time *periods*.

Examples of interval aggregations are monthly asset returns, or trailing 12-month asset returns calculated every month.



Defining *Trailing* Look-back Time Intervals

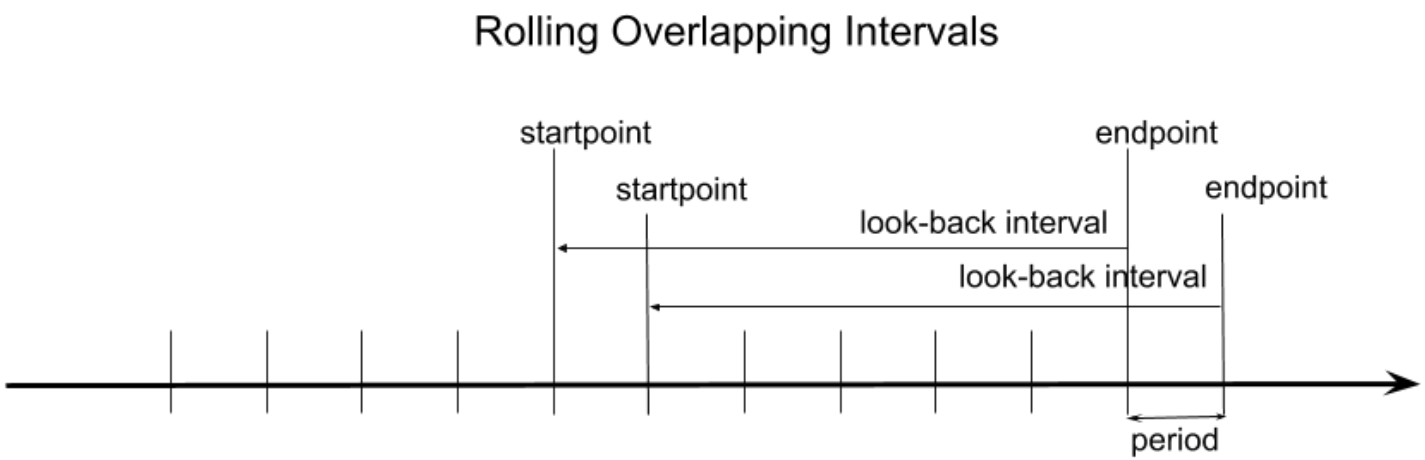
A *trailing aggregation* is performed over a vector of *end points* in time.

The first *end point* is equal to zero 0.

The *start points* are the *end points* lagged by the *look-back interval*.

An example of a trailing aggregation are moving average prices.

```
> ohlc <- rutils::etfenv$VTI
> # Number of data points
> nrows <- NROW(ohlc["2018-06/"])
> # Define endd at each point in time
> endd <- 0:nrows
> # Number of data points in look_back interval
> look_back <- 22
> # startp are endd lagged by look_back
> startp <- c(rep_len(0, look_back), endd[1:(NROW(endd)-look_back)])
> head(startp, 33)
```



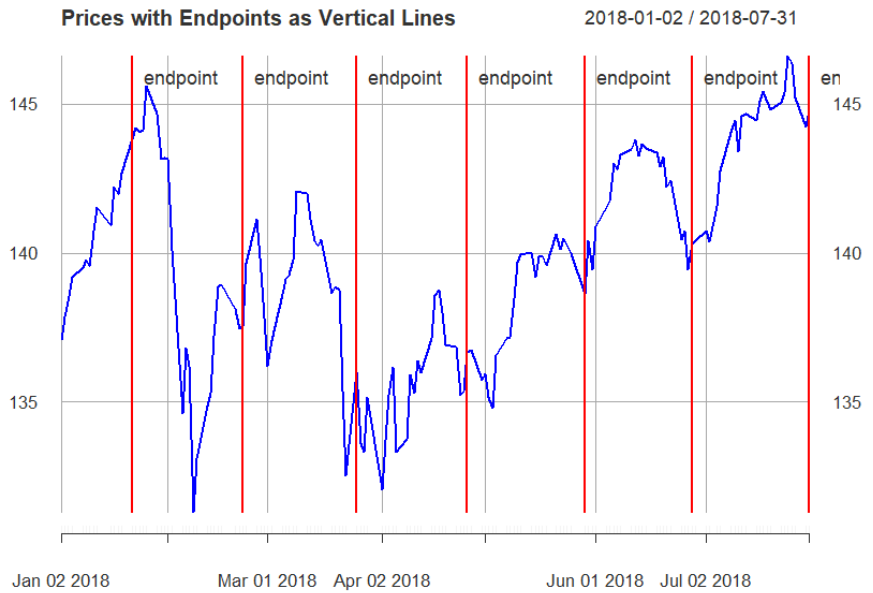
Defining Equally Spaced *end points* of a Time Series

The neighboring *end points* may be separated by a fixed number of periods, equal to `npoints`.

If the total number of data points is not an integer multiple of `npoints`, then a stub interval must be added either at the beginning or at the end of the *end points*.

The function `xts::endpoints()` extracts the indices of the last observations in each calendar period of an *xts* series.

```
> # Number of data points
> closep <- quantmod::Cl(ohlc["2018/"])
> nrows <- NROW(closep)
> # Number of periods between endpoints
> npoints <- 21
> # Number of npoints that fit over nrows
> nagg <- nrows %/% npoints
> # If(nrows==npoints*nagg then whole number
> endd <- (0:nagg)*npoints
> # Stub interval at beginning
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Else stub interval at end
> endd <- c((0:nagg)*npoints, nrows)
> # Or use xts::endpoints()
> endd <- xts::endpoints(closep, on="weeks")
```




```
> # Plot data and endpoints as vertical lines
> plot.xts(closep, col="blue", lwd=2, xlab="", ylab="",
+    main="Prices with Endpoints as Vertical Lines")
> addEventLines(xts(rep("endpoint", NROW(endd)-1), zoo::index(closep
+         col="red", lwd=2, pos=4)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- "blue"
> quantmod::chart_Series(closep, theme=plot_theme,
+   name="prices with endpoints as vertical lines")
> abline(v=endd, col="red", lwd=2)
```

Defining *Overlapping* Look-back Time Intervals

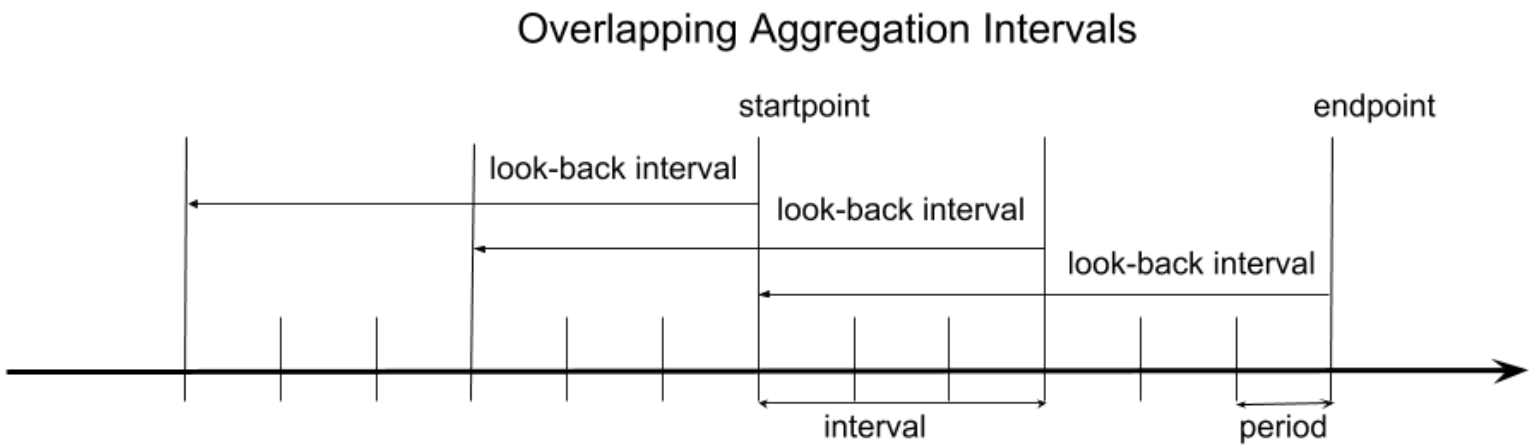
Overlapping time intervals can be defined if the *start points* are equal to the *end points* lagged by the *look-back interval*.

An example of an overlapping interval aggregation are trailing 12-month asset returns calculated every month.

```
> # Number of data points
> nrows <- NROW(rutils::etfenv$VTI["2019/"])
> # Number of npoints that fit over nrows
> npoints <- 21
> nagg <- nrows %/% npoints
> # Stub interval at beginning
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
```

The length of the *look-back interval* can be defined either as the number of data points, or as the number of *end points* to look back over.

```
> # look_back defined as number of data points
> look_back <- 252
> # startp are endd lagged by look_back
> startp <- (endd - look_back + 1)
> startp <- ifelse(startp < 0, 0, startp)
> # look_back defined as number of endd
> look_back <- 12
> startp <- c(rep_len(0, look_back), endd[1:(NROW(endd)- look_back)]
> # Bind startp with endd
> cbind(startp, endd)
```



Defining *Non-overlapping* Look-back Time Intervals

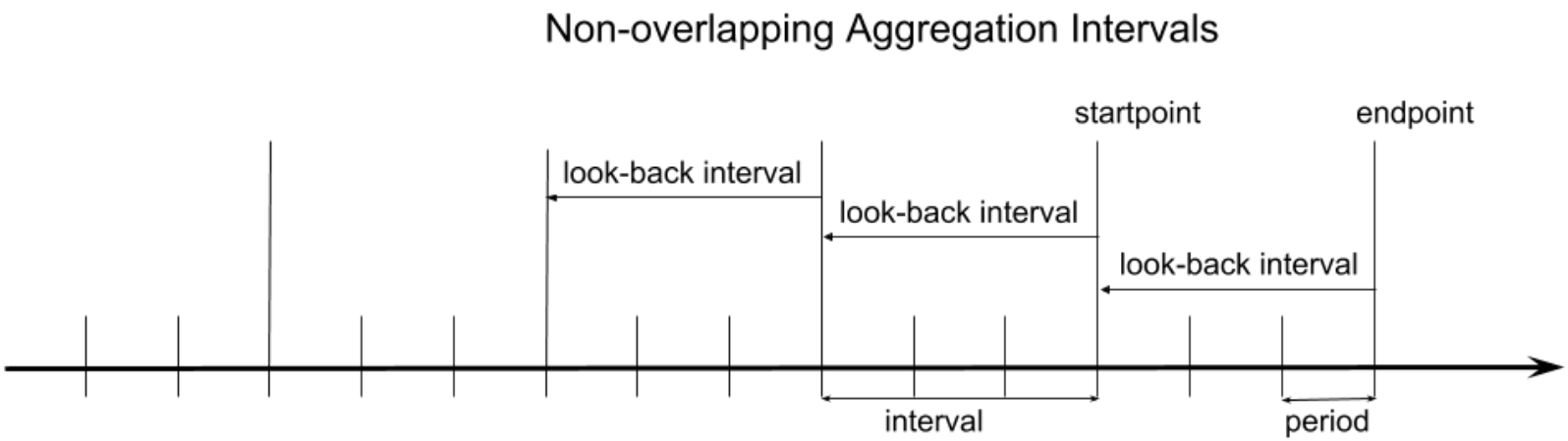
Non-overlapping time intervals can be defined if *start points* are equal to the previous *end points*.

In that case the look-back *intervals* are non-overlapping and *contiguous* (each *start point* is the *end point* of the previous interval).

If the *start points* are defined as the previous *end points* plus 1, then the *intervals* are *exclusive*.

Exclusive intervals are used for calculating *out-of-sample* aggregations over future intervals.

```
> # Number of data points
> nrows <- NROW(rutils::etfenv$VTI["2019/"])
> # Number of data points per interval
> npoints <- 21
> # Number of npointss that fit over nrows
> nagg <- nrows %/% npoints
> # Define endd with beginning stub
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Define contiguous startp
> startp <- c(0, endd[1:(NROW(endd)-1)])
> # Define exclusive startp
> startp <- c(0, endd[1:(NROW(endd)-1)]+1)
```



Performing Trailing Aggregations Using sapply()

Aggregations performed over time series can be extremely slow if done improperly, therefore it's very important to find the fastest methods of performing aggregations.

The `sapply()` functional allows performing aggregations over the look-back *intervals*.

The `sapply()` functional by default returns a vector or matrix, not an *xts* series.

The vector or matrix returned by `sapply()` therefore needs to be coerced into an *xts* series.

The variable `look_back` is the size of the look-back interval, equal to the number of data points used for applying the aggregation function (including the current point).

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> endd <- 0:NROW(closep)  # End points at each point
> npts <- NROW(endd)
> look_back <- 22  # Number of data points per look-back interval
> # startp are multi-period lag of endd
> startp <- c(rep_len(0, look_back), endd[1:(npts - look_back)])
> # Define list of look-back intervals for aggregations over past
> look_backs <- lapply(2:npts, function(it) {
+     startp[it]:endd[it]
+ })  # end lapply
> # Define aggregation function
> aggfun <- function(xtsv) c(max=max(xtsv), min=min(xtsv))
> # Perform aggregations over look_backs list
> aggs <- sapply(look_backs,
+     function(look_back) aggfun(closep[look_back])
+ )  # end sapply
> # Coerce aggs into matrix and transpose it
> if (is.vector(aggs))
+   aggs <- t(aggs)
> aggs <- t(aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
```

Performing Trailing Aggregations Using lapply()

The lapply() functional allows performing aggregations over the look-back *intervals*.

The lapply() functional by default returns a list, not an *xts* series.

If lapply() returns a list of *xts* series, then this list can be collapsed into a single *xts* series using the function do_call_rbind() from package *rutils*.

The function chart_Series() from package *quantmod* can produce a variety of time series plots.

chart_Series() plots can be modified by modifying *plot objects* or *theme objects*.

A plot *theme object* is a list containing parameters that determine the plot appearance (colors, size, fonts).

The function chart_theme() returns the theme object.

```
> # Perform aggregations over look_backs list
> aggs <- lapply(look_backs,
+     function(look_back) aggfun(closep[look_back])
+ )  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Convert into xts
> aggs <- xts::xts(aggs, order.by=zoo::index(closep))
> aggs <- cbind(aggs, closep)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green")
> x11(width=6, height=5)
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6, y.intersp=0.4,
+   col=plot_theme$col$line.col, bty="n")
```

Defining Functionals for Trailing Aggregations

The functional `roll_agg()` performs trailing aggregations of its function argument `FUN`, over an *xts* series (`x_ts`), and a look-back interval (`look_back`).

The argument `FUN` is an aggregation function over a subset of `x_ts` series.

The dots `"..."` argument is passed into `FUN` as additional arguments.

The argument `look_back` is equal to the number of periods of `x_ts` series which are passed to the aggregation function `FUN`.

The functional `roll_agg()` calls `lapply()`, which loops over the length of series `x_ts`.

Note that two different intervals may be used with `roll_agg()`.

The first interval is the argument `look_back`.

A second interval may be one of the variables bound to the dots `"..."` argument, and passed to the aggregation function `FUN` (for example, an *EMA* window).

```
> # Define functional for trailing aggregations
> roll_agg <- function(xtsv, look_back, FUN, ...) {
+ # Define end points at every period
+   endd <- 0:NROW(xtsv)
+   npts <- NROW(endd)
+ # Define starting points as lag of endd
+   startp <- c(rep_len(0, look_back), endd[1:(npts- look_back)])
+ # Perform aggregations over look_backs list
+   aggs <- lapply(2:npts, function(it)
+     FUN(xtsv[startp[it]:endd[it]], ...)
+   )  # end lapply
+ # rbind list into single xts or matrix
+   aggs <- rutils::do_call(rbind, aggs)
+ # Coerce aggs into xts series
+   if (!is.xts(aggs))
+     aggs <- xts(aggs, order.by=zoo::index(xtsv))
+   aggs
+ }  # end roll_agg
> # Define aggregation function
> aggfun <- function(xtsv)
+   c(max=max(xtsv), min=min(xtsv))
> # Perform aggregations over trailing interval
> aggs <- roll_agg(closep, look_back=look_back, FUN=aggfun)
> class(aggs)
> dim(aggs)
```

Benchmarking Speed of Trailing Aggregations

The speed of trailing aggregations using `apply()` loops can be greatly increased by simplifying the aggregation function

For example, an aggregation function that returns a vector is over 13 times faster than a function that returns an *xts* object.

```
> # Define aggregation function that returns a vector
> agg_vector <- function(xtsv)
+   c(max=max(xtsv), min=min(xtsv))
> # Define aggregation function that returns an xts
> agg_xts <- function(xtsv)
+   xts(t(c(max=max(xtsv), min=min(xtsv))), order.by=end(xtsv))
> # Benchmark the speed of aggregation functions
> library(microbenchmark)
> summary(microbenchmark(
+   agg_vector=roll_agg(closep, look_back=look_back, FUN=agg_vector),
+   agg_xts=roll_agg(closep, look_back=look_back, FUN=agg_xts),
+   times=10))[, c(1, 4, 5)]
```

Benchmarking Functionals for Trailing Aggregations

Several packages contain functionals designed for performing trailing aggregations:

- `rollapply.zoo()` from package *zoo*,
- `rollapply.xts()` from package *xts*,
- `apply.rolling()` from package *PerformanceAnalytics*,

These functionals don't require specifying the *end points*, and instead calculate the *end points* from the trailing interval width.

These functionals can only apply functions that return a single value, not a vector.

These functionals return an *xts* series with leading `NA` values at points before the trailing interval can fit over the data.

The argument `align="right"` of `rollapply()` determines that aggregations are taken from the past.

The functional `rollapply.xts` is the fastest, about as fast as performing an `lapply()` loop directly.

```
> # Define aggregation function that returns a single value
> aggfun <- function(xtsv)  max(xtsv)
> # Perform aggregations over a trailing interval
> aggs <- xts:::rollapply.xts(closep, width=look_back,
+               FUN=aggfun, align="right")
> # Perform aggregations over a trailing interval
> library(PerformanceAnalytics)  # Load package PerformanceAnalytics
> aggs <- apply.rolling(closep, width=look_back, FUN=aggfun)
> # Benchmark the speed of the functionals
> library(microbenchmark)
> summary(microbenchmark(
+   roll_agg=roll_agg(closep, look_back=look_back, FUN=max),
+   roll_xts=xts:::rollapply.xts(closep, width=look_back, FUN=max, align="
+   apply_rolling=apply.rolling(closep, width=look_back, FUN=max),
+   times=10))[, c(1, 4, 5)]
```

Trailing Aggregations Using *Vectorized* Functions

The generic functions `cumsum()`, `cummax()`, and `cummin()` return the cumulative sums, minima, and maxima of *vectors* and *time series* objects.

The methods for these functions are implemented as *vectorized compiled* functions, and are therefore much faster than `apply()` loops.

The `cumsum()` function can be used to efficiently calculate the trailing sum of an an *xts* series.

Using the function `cumsum()` is over 25 times faster than using `apply()` loops.

But trailing volatilities and higher moments can't be easily calculated using `cumsum()`.

```
> # Trailing sum using cumsum()
> roll_sum <- function(xtsv, look_back) {
+   cumsumv <- cumsum(na.omit(xtsv))
+   output <- cumsumv - rutils::lagit(x=cumsumv, lagg=look_back)
+   output[1:look_back, ] <- cumsumv[1:look_back, ]
+   colnames(output) <- paste0(colnames(xtsv), "_stdev")
+   output
+ }  # end roll_sum
> aggs <- roll_sum(closep, look_back=look_back)
> # Perform trailing aggregations using lapply loop
> aggs <- lapply(2:npts, function(it)
+     sum(closep[startp[it]:endd[it]])
+ )  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> head(aggs)
> tail(aggs)
> # Benchmark the speed of both methods
> library(microbenchmark)
> summary(microbenchmark(
+   roll_sum=roll_sum(closep, look_back=look_back),
+   s_apply=sapply(look_backs,
+     function(look_back) sum(closep[look_back])),
+   times=10))[, c(1, 4, 5)]
```

Filtering Time Series Using Function filter()

The function filter() applies a linear filter to time series, vectors, and matrices, and returns a time series of class "ts".

The function filter() with the argument method="convolution" calculates the *convolution* of the vector r_t with the filter φ_i:

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p}$$

Where f_t is the filtered output vector, and φ_i are the filter coefficients.

filter() with method="recursive" calculates a *recursive* filter over the vector of random *innovations* ξ_t as follows:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p} + \xi_t$$

Where r_t is the filtered output vector, and φ_i are the filter coefficients.

The *recursive* filter describes an *AR(p)* process, which is a special case of an *ARIMA* process.

filter() is very fast because it calculates the filter by calling compiled C++ functions.

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Calculate EMA prices using filter()
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> pricef <- stats::filter(closep, filter=weightv,
+                         method="convolution", sides=1)
> pricef <- as.numeric(pricef)
> # filter() returns time series of class "ts"
> class(pricef)
> # Filter using compiled C++ function directly
> getAnywhere(C_cfilter)
> str(stats:::C_cfilter)
> priceff <- .Call(stats:::C_cfilter, closep,
+                  filter=weightv, sides=1, circular=FALSE)
> all.equal(as.numeric(pricef), priceff, check.attributes=FALSE)
> # Calculate EMA prices using HighFreq::roll_conv()
> pricecpp <- HighFreq::roll_conv(closep, weightv=weightv)
> all.equal(pricef[-(1:look_back)],
+     as.numeric(pricecpp)[-(1:look_back)],
+     check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   priceff=.Call(stats:::C_cfilter, closep, filter=weightv, sides=1
+   cumsumv=cumsum(closep),
+   rcpp=HighFreq::roll_conv(closep, weightv=weightv)
+   ), times=10)[, c(1, 4, 5)]
```

Performing Trailing Aggregations Using Package *TTR*

The package *TTR* contains functions for calculating trailing aggregations over *vectors* and *time series* objects:

- `runSum()` for trailing sums,
- `runMin()` and `runMax()` for trailing minima and maxima,
- `runSD()` for trailing standard deviations,
- `runMedian()` and `runMAD()` for trailing medians and Median Absolute Deviations (*MAD*),
- `runCor()` for trailing correlations,

The trailing *TTR* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` or `Fortran` code).

But the trailing *TTR* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> # Calculate the trailing maximum and minimum over a vector of data
> roll_maxminr <- function(vecv, look_back) {
+   nrows <- NROW(vecv)
+   max_min <- matrix(numeric(2:nrows), nc=2)
+   # Loop over periods
+   for (it in 1:nrows) {
+     sub_vec <- vecv[max(1, it-look_back+1):it]
+     max_min[it, 1] <- max(sub_vec)
+     max_min[it, 2] <- min(sub_vec)
+   }  # end for
+   return(max_min)
+ }  # end roll_maxminr
> max_minr <- roll_maxminr(closep, look_back)
> max_minr <- xts::xts(max_minr, zoo::index(closep))
> library(TTR)  # Load package TTR
> max_min <- cbind(TTR::runMax(x=closep, n=look_back),
+            TTR::runMin(x=closep, n=look_back))
> all.equal(max_min[-(1:look_back), ], max_minr[-(1:look_back), ],
> # Benchmark the speed of TTR::runMax
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=roll_maxminr(closep, look_back),
+   ttr=TTR::runMax(closep, n=look_back),
+   times=10))[, c(1, 4, 5)]
> # Benchmark the speed of TTR::runSum
> summary(microbenchmark(
+   vector_r=cumsum(coredata(closep)),
+   rutils=rutils::roll_sum(closep, look_back=look_back),
+   ttr=TTR::runSum(closep, n=look_back),
+   times=10))[, c(1, 4, 5)]
```

Trailing *Weighted* Aggregations Using Package *roll*

The package *roll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()`, `roll_max()`, `roll_mean()`, and `roll_median()` for *weighted* trailing sums, maximums, means, and medians,
- `roll_var()` for *weighted* trailing variance,
- `roll_scale()` for trailing scaling and centering of time series,
- `roll_lm()` for trailing regression,
- `roll_pcr()` for trailing principal component regressions of time series,

The *roll* functions are about `1,000` times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled `C++` code, using packages *Rcpp* and *RcppArmadillo*.

The *roll* functions accept *xts* time series, and they return *xts*.

```
> # Calculate trailing VTI variance using package roll
> library(roll)  # Load roll
> retp <- na.omit(rutils::etfenv$returns$VTI)
> look_back <- 22
> # Calculate trailing sum using roll::roll_sum
> sum_roll <- roll::roll_sum(retp, width=look_back, min_obs=1)
> # Calculate trailing sum using rutils
> sum_rutils <- rutils::roll_sum(retp, look_back=look_back)
> all.equal(sum_roll[-(1:look_back), ],
+     sum_rutils[-(1:look_back), ], check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retp),
+   roll=roll::roll_sum(retp, width=look_back),
+   RcppRoll=RcppRoll::roll_sum(retp, n=look_back),
+   rutils=rutils::roll_sum(retp, look_back=look_back),
+   times=10))[, c(1, 4, 5)]
```

Trailing *Weighted* Aggregations Using Package *RcppRoll*

The package *RcppRoll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()` for *weighted* trailing sums,
- `roll_min()` and `roll_max()` for *weighted* trailing minima and maxima,
- `roll_sd()` for *weighted* trailing standard deviations,
- `roll_median()` for *weighted* trailing medians,

The *RcppRoll* functions accept *xts* objects, but they return matrices, not *xts* objects.

The trailing *RcppRoll* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` code).

But the trailing *RcppRoll* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> library(RcppRoll)  # Load package RcppRoll
> # Calculate trailing sum using RcppRoll
> sum_roll <- RcppRoll::roll_sum(retp, align="right", n=look_back)
> # Calculate trailing sum using rutils
> sum_rutils <- rutils::roll_sum(retp, look_back=look_back)
> all.equal(sum_roll, coredata(sum_rutils[-(1:(look_back-1))]),
+     check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retp),
+   RcppRoll=RcppRoll::roll_sum(retp, n=look_back),
+   rutils=rutils::roll_sum(retp, look_back=look_back),
+   times=10))[, c(1, 4, 5)]
> # Calculate EMA prices using RcppRoll
> closep <- quantmod::Cl(rutils::etfenv$VTI)
> weightv <- exp(0.1*1:look_back)
> pricema <- RcppRoll::roll_mean(closep,
+ align="right", n=look_back, weights=weightv)
> pricema <- cbind(closep,
+   rbind(coredata(closep[1:(look_back-1), ]), pricema))
> colnames(pricema) <- c("VTI", "VTI EMA")
> # Plot an interactive dygraph plot
> dygraphs::dygraph(pricema)
> # Or static plot of EMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> quantmod::chart_Series(pricema, theme=plot_theme, name="EMA prices
> legend("top", legend=colnames(pricema),
+  bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Performing Trailing Aggregations Using Package *caTools*

The package *caTools* contains functions for calculating trailing interval aggregations over a `vector` of data:

- `runmin()` and `runmax()` for trailing minima and maxima,
- `runsd()` for trailing standard deviations,
- `runmad()` for trailing Median Absolute Deviations (*MAD*),
- `runquantile()` for trailing quantiles,

Time series need to be coerced to *vectors* before they are passed to *caTools* functions.

The trailing *caTools* functions are very fast because they are *compiled* functions (compiled from `C++` code).

The argument `"endrule"` determines how the end values of the data are treated.

The argument `"align"` determines whether the interval is centered (default), left-aligned or right-aligned, with `align="center"` the fastest option.

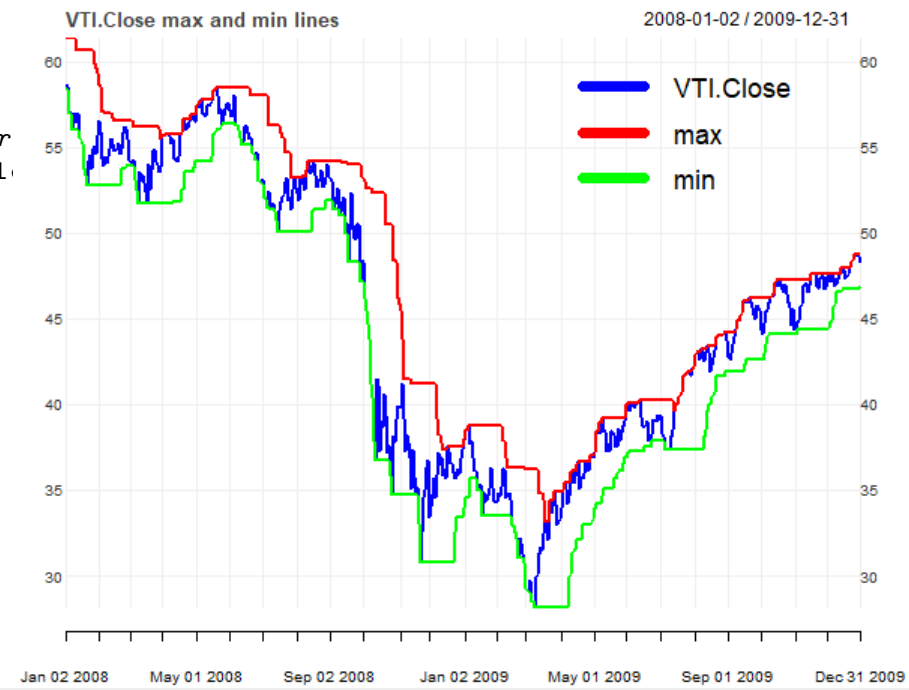
```
> library(caTools)  # Load package "caTools"
> # Get documentation for package "caTools"
> packageDescription("caTools")  # Get short description
> help(package="caTools")  # Load help page
> data(package="caTools")  # List all datasets in "caTools"
> ls("package:caTools")  # List all objects in "caTools"
> detach("package:caTools")  # Remove caTools from search path
> # Median filter
> look_back <- 2
> closep <- quantmod::Cl(HighFreq::SPY["2012-02-01/2012-04-01"])
> med_ian <- runmed(x=closep, k=look_back)
> # Vector of trailing volatilities
> sigmav <- runsd(x=closep, k=look_back,
+           endrule="constant", align="center")
> # Vector of trailing quantiles
> quantvs <- runquantile(x=closep, k=look_back,
+   probs=0.9, endrule="constant", align="center")
```

Performing Trailing Aggregations Using *RcppArmadillo*

RcppArmadillo functions for calculating trailing aggregations are often the fastest.

```
// Rcpp header with information for C++ compiler
#include <RcppArmadillo.h> // include C++ header file fr
using namespace arma; // use C++ namespace from Armadill
// declare dependency on RcppArmadillo
// [[Rcpp::depends(RcppArmadillo)]]

// export the function roll_maxmin() to R
// [[Rcpp::export]]
arma::mat roll_maxmin(const arma::vec& vecv,
                      const arma::uword& look_back) {
  arma::uword n_rows = vecv.size();
  arma::mat max_min[nrows, 2);
  arma::vec sub_vec;
  // startup period
  max_min(0, 0) = vecv[0];
  max_min(0, 1) = vecv[0];
  for (uword it = 1; it < look_back; it++) {
    sub_vec = vecv.subvec(0, it);
    max_min(it, 0) = sub_vec.max();
    max_min(it, 1) = sub_vec.min();
  }  // end for
  // remaining periods
  for (uword it = look_back; it <.n_rows; it++) {
    sub_vec = vecv.subvec(it- look_back + 1, it);
    max_min(it, 0) = sub_vec.max();
    max_min(it, 1) = sub_vec.min();
  }  // end for
  return max_min;
}  // end roll_maxmin
```




```
> # Compile Rcpp functions
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/R/Rcpp/roll_maxmin.cpp"
> max_minarma <- roll_maxmin(closep, look_back)
> max_minarma <- xts::xts(max_minr, zoo::index(closep))
> max_min <- cbind(TTR::runMax(x=closep, n=look_back),
+            TTR::runMin(x=closep, n=look_back))
> all.equal(max_min[-(1:look_back), ], max_minarma[-(1:look_back),
> # Benchmark the speed of TTR::runMax
> library(microbenchmark)
> summary(microbenchmark(
+   arma=roll_maxmin(closep, look_back),
+   ttr=TTR::runMax(closep, n=look_back),
+   times=10))[, c(1, 4, 5)]
> # Dygraphs plot with max_min lines
> datav <- cbind(closep, max_minarma)
> colnames(datav)[2:3] <- c("max", "min")
```

Determining Calendar *end points* of *xts* Time Series

The function `xts::endpoints()` extracts the indices of the last observations in each calendar period of an *xts* series.

For example:
`endpoints(x, on="hours")`
extracts the indices of the last observations in each hour.

The *end points* calculated by `endpoints()` aren't always equally spaced, and aren't the same as those calculated from fixed intervals.

For example, the last observations in each day aren't equally spaced due to weekends and holidays.

```
> # Indices of last observations in each hour
> endd <- xts::endpoints(closep, on="hours")
> head(endd)
> # extract the last observations in each hour
> head(closep[endd, ])
```

Performing Non-overlapping Aggregations Using sapply()

The apply() functionals allow for applying a function over intervals of an *xts* series defined by a vector of *end points*.

The sapply() functional by default returns a vector or matrix, not an *xts* series.

The vector or matrix returned by sapply() therefore needs to be coerced into an *xts* series.

The function chart_Series() from package *quantmod* can produce a variety of time series plots.

chart_Series() plots can be modified by modifying *plot objects* or *theme objects*.

A plot *theme object* is a list containing parameters that determine the plot appearance (colors, size, fonts).

The function chart_theme() returns the theme object.

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Number of data points
> nrows <- NROW(closep)
> # Number of data points per interval
> look_back <- 22
> # Number of look_backs that fit over nrows
> nagg <- nrows %/% look_back
> # Define endd with beginning stub
> endd <- c(0, nrows-look_back*nagg + (0:nagg)*look_back)
> # Define contiguous startp
> startp <- c(0, endd[1:(NROW(endd)-1)])
> # Define list of look-back intervals for aggregations over past
> look_backs <- lapply(2:NROW(endd), function(it) {
+     startp[it]:endd[it]
+ })  # end lapply
> look_backs[[1]]
> look_backs[[2]]
> # Perform sapply() loop over look_backs list
> aggs <- sapply(look_backs, function(look_back) {
+   xtsv <- closep[look_back]
+   c(max=max(xtsv), min=min(xtsv))
+ })  # end sapply
> # Coerce aggs into matrix and transpose it
> if (is.vector(aggs))
+   aggs <- t(aggs)
> aggs <- t(aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> head(aggs)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Performing Non-overlapping Aggregations Using lapply()

The apply() functionals allow for applying a function over intervals of an *xts* series defined by a vector of *end points*.

The lapply() functional by default returns a list, not an *xts* series.

If lapply() returns a list of *xts* series, then this list can be collapsed into a single *xts* series using the function do_call_rbind() from package *rutils*.

```
> # Perform lapply() loop over look_backs list
> aggs <- lapply(look_backs, function(look_back) {
+   xtsv <- closep[look_back]
+   c(max=max(xtsv), min=min(xtsv))
+ })  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> head(aggs)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme, name="price aggrega
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Performing Interval Aggregations Using period.apply()

The functional period.apply() from package *xts* performs *aggregations* over non-overlapping intervals of an *xts* series defined by a vector of *end points*.

Internally period.apply() performs an sapply() loop, and is therefore about as fast as an sapply() loop.

The package *xts* also has several specialized functionals for aggregating data over *end points*:

- period.sum() calculate the sum for each period,
- period.max() calculate the maximum for each period,
- period.min() calculate the minimum for each period,
- period.prod() calculate the product for each period,

```
> # Define functional for trailing aggregations over endd
> roll_agg <- function(xtsv, endd, FUN, ...) {
+   nrows <- NROW(endd)
+ # startp are single-period lag of endd
+   startp <- c(1, endd[1:(nrows-1)])
+ # Perform aggregations over look_backs list
+   aggs <- lapply(look_backs,
+     function(look_back) FUN(xtsv[look_back], ...))  # end lapply
+ # rbind list into single xts or matrix
+   aggs <- rutils::do_call(rbind, aggs)
+   if (!is.xts(aggs))
+     aggs <-  # Coerce aggs into xts series
+     xts(aggs, order.by=zoo::index(xtsv[endd]))
+   aggs
+ }  # end roll_agg
> # Apply sum() over endd
> aggs <- roll_agg(closep, endd=endd, FUN=sum)
> aggs <- period.apply(closep, INDEX=endd, FUN=sum)
> # Benchmark the speed of aggregation functions
> summary(microbenchmark(
+   roll_agg=roll_agg(closep, endd=endd, FUN=sum),
+   period_apply=period.apply(closep, INDEX=endd, FUN=sum),
+   times=10))[, c(1, 4, 5)]
> aggs <- period.sum(closep, INDEX=endd)
> head(aggs)
```

Performing Aggregations of *xts* Over Calendar Periods

The package *xts* has convenience wrapper functionals for `period.apply()`, that apply functions over calendar periods:

- `apply.daily()` applies functions over daily periods,
- `apply.weekly()` applies functions over weekly periods,
- `apply.monthly()` applies functions over monthly periods,
- `apply.quarterly()` applies functions over quarterly periods,
- `apply.yearly()` applies functions over yearly periods,

These functionals don't require specifying a vector of *end points*, because they determine the *end points* from the calendar periods.

```
> # Load package HighFreq
> library(HighFreq)
> # Extract closing minutely prices
> closep <- quantmod::Cl(rutils::etfenv$VTI["2019"])
> # Apply "mean" over daily periods
> aggs <- apply.daily(closep, FUN=sum)
> head(aggs)
```

Performing Aggregations Over Overlapping Intervals

The functional `period.apply()` performs aggregations over *non-overlapping* intervals.

But it's often necessary to perform aggregations over *overlapping* intervals, defined by a vector of *end points* and a *look-back interval*.

The *start points* are defined as the *end points* lagged by the interval width (number of periods in the *look-back interval*).

Each point in time has an associated *look-back interval*, which starts at a certain number of periods in the past (*start_point*) and ends at that point (*end_point*).

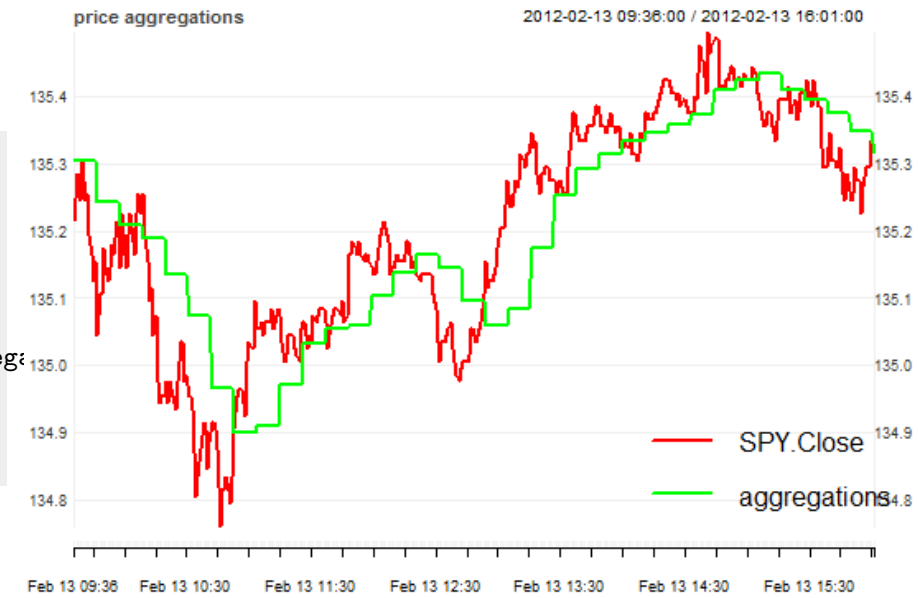
The variable `look_back` is equal to the number of end points in the *look-back interval*, while (`look_back - 1`) is equal to the number of intervals in the look-back.

```
> # Define endd with beginning stub
> npoints <- 5
> nrows <- NROW(closep)
> nagg <- nrows %/% npoints
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Number of data points in look_back interval
> look_back <- 22
> # startp are endd lagged by look_back
> startp <- (endd - look_back + 1)
> startp <- ifelse(startp < 0, 0, startp)
> # Perform lapply() loop over look_backs list
> aggs <- lapply(2:NROW(endd), function(it) {
+ xtsv <- closep[startp[it]:endd[it]]
+ c(max=max(xtsv), min=min(xtsv))
+ })  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Extending Interval Aggregations

Interval aggregations produce values only at the *end points*, but they can be carried forward in time using the function `na.locf.xts()` from package *xts*.

```
> aggs <- cbind(closep, aggs)
> tail(aggs)
> aggs <- na.omit(xts:::na.locf.xts(aggs))
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme, name="price aggrega
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

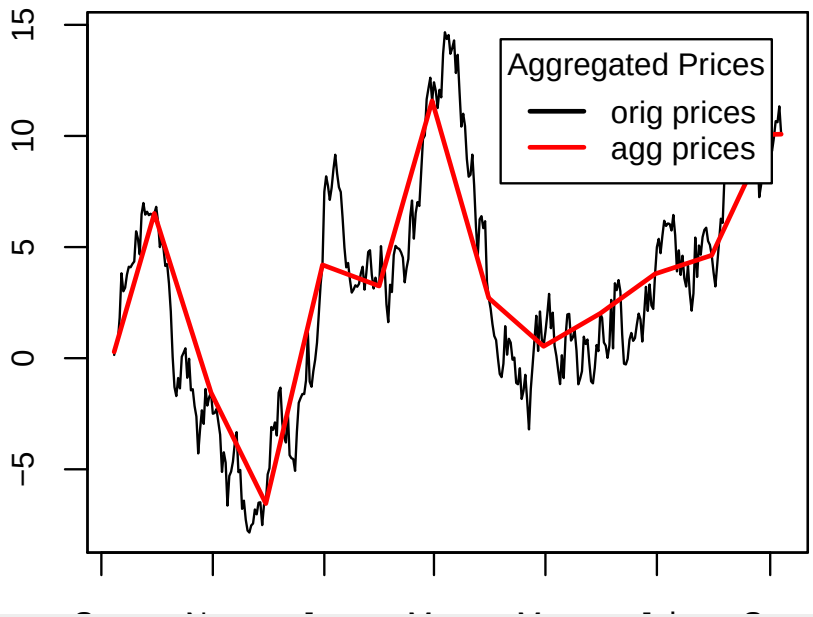


Performing Interval Aggregations of *zoo* Time Series

The method `aggregate.zoo()` performs aggregations of *zoo* series over non-overlapping intervals defined by a vector of aggregation groups (minutes, hours, days, etc.).

For example, `aggregate.zoo()` can calculate the average monthly returns.

```
> library(zoo)  # Load package zoo
> # Create zoo time series of random returns
> datev <- Sys.Date() + 0:365
> zoo_series <- zoo(rnorm(NROW(datev)), order.by=datev)
> # Create monthly dates
> dates_agg <- as.Date(as.yearmon(zoo::index(zoo_series)))
> # Perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=datev_agg, FUN=mean)
> # Merge with original zoo - union of dates
> zoo_agg <- cbind(zoo_series, zoo_agg)
> # Replace NA's using locf
> zoo_agg <- na.locf(zoo_agg, na.rm=FALSE)
> # Extract aggregated zoo
> zoo_agg <- zoo_agg[zoo::index(zoo_series), 2]
```



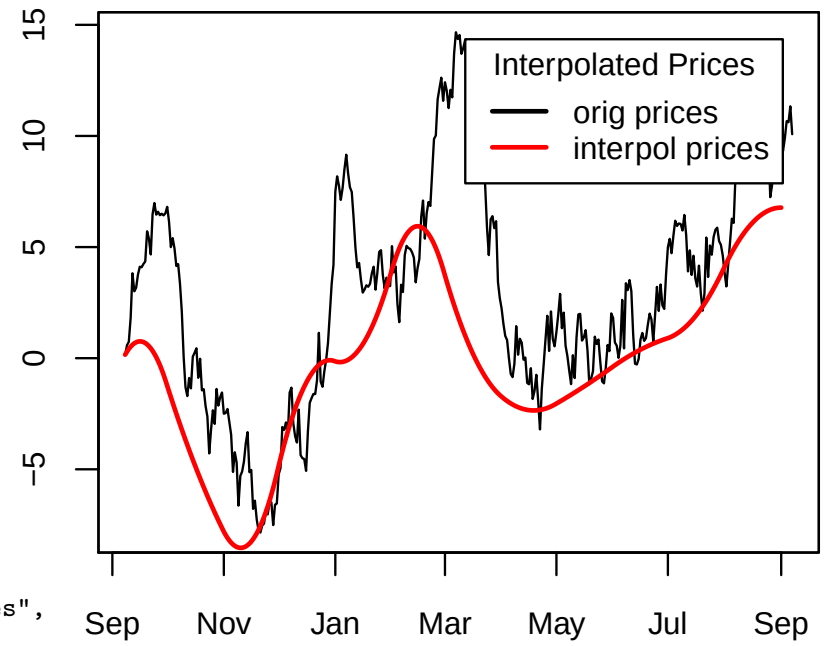

```
> # Plot original and aggregated cumulative returns
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, bty="n",
+   title="Aggregated Prices", y.intersp=0.4,
+   leg=c("orig prices", "agg prices"),
+   lwd=2, bg="white", col=c("black", "red"))
```

Interpolating *zoo* Time Series

The package *zoo* has two functions for replacing `NA` values using interpolation:

- `na.approx()` performs linear interpolation,
- `na.spline()` performs spline interpolation,

```
> # Perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=datev_agg, FUN=mean)
> # Merge with original zoo - union of dates
> zoo_agg <- cbind(zoo_series, zoo_agg)
> # Replace NA's using linear interpolation
> zoo_agg <- na.approx(zoo_agg)
> # Extract interpolated zoo
> zoo_agg <- zoo_agg[zoo::index(zoo_series), 2]
> # Plot original and interpolated zoo
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title="Interpolated Prices",
+  leg=c("orig prices", "interpol prices"), lwd=2, bg="white",
+  col=c("black", "red"), bty="n", y.intersp=0.4)
```

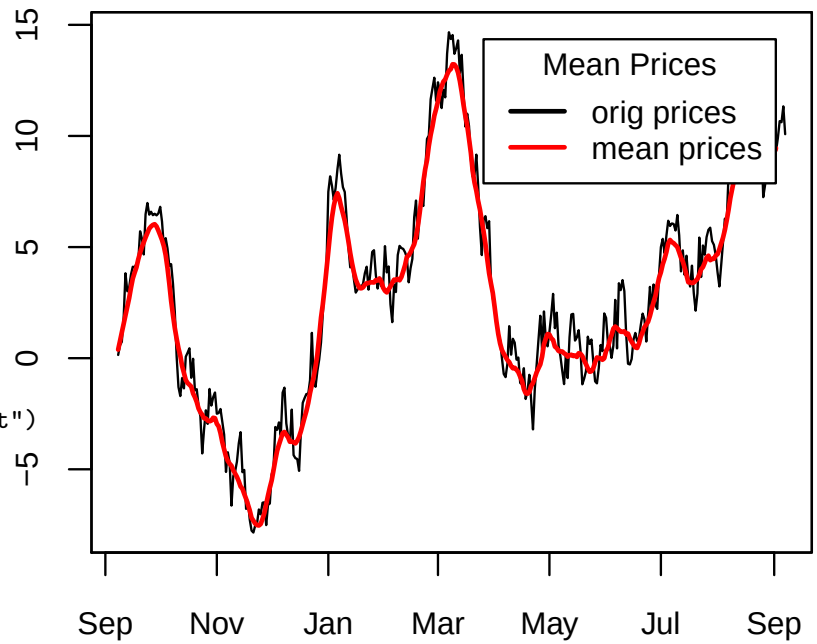


Performing Trailing Aggregations Over *zoo* Time Series

The package *zoo* has several functions for trailing calculations:

- `rollapply()` performing aggregations over a trailing (sliding) interval,
- `rollmean()` calculating trailing means,
- `rollmedian()` calculating trailing median,
- `rollmax()` calculating trailing max,

```
> # "mean" aggregation over interval with width=11
> zoo_mean <- rollapply(zoo_series, width=11, FUN=mean, align="right")
> # Merge with original zoo - union of dates
> zoo_mean <- cbind(zoo_series, zoo_mean)
> # Replace NA's using na.locf
> zoo_mean <- na.locf(zoo_mean, na.rm=FALSE, fromLast=TRUE)
> # Extract mean zoo
> zoo_mean <- zoo_mean[zoo::index(zoo_series), 2]
> # Plot original and interpolated zoo
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_mean), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title="Mean Prices",
+  leg=c("orig prices", "mean prices"), lwd=2, bg="white",
+  col=c("black", "red"), bty="n", y.intersp=0.4)
```



aggregations are taken from the past,

The argument `align="right"` determines that