Risk Analysis and Model Construction

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

November 25, 2022

NYU | TANDON SCHOOL OF ENGINEERING

Kernel Density of Asset Returns

The kernel density is proportional to the number of data points close to a given point.

The kernel density is analogous to a histogram, but it provides more detailed information about the distribution of the data.

The smoothing kernel $K(x)$ is a symmetric function which decreases with the distance x.

The kernel density d_r at a point r is equal to the sum over the kernel function $K(x)$:

$$d_r = \sum_{j=1}^{n} K(r - r_j)$$

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

The parameter *smoothing bandwidth* is the standard deviation of the smoothing kernel $K(x)$.

The function `density()` returns a vector of densities at equally spaced points, not for the original data points.

The function `approx()` interpolates a vector of data into another vector.

```
> library(rutils)  # Load package rutils
> # Calculate VTI percentage returns
> retvti <- rutils::etfenv$returns$VTI
> retvti <- drop(coredata(na.omit(retvti)))
> nrows <- NROW(retvti)
> # Mean and standard deviation of returns
> c(mean(retvti), sd(retvti))
> # Calculate the smoothing bandwidth as the MAD of returns 10 point
> retvti <- sort(retvti)
> bwidth <- 10*mad(rutils::diffit(retvti, lagg=10))
> # Calculate the kernel density
> densityv <- sapply(1:nrows, function(it) {
+   sum(dnorm(retvti-retvti[it], sd=bwidth))
+ })  # end sapply
> madv <- mad(retvti)
> plot(retvti, densityv, xlim=c(-5*madv, 5*madv),
+      t="l", col="blue", lwd=3,
+      xlab="returns", ylab="density",
+      main="Density of VTI Returns")
> # Calculate the kernel density using density()
> densityv <- density(retvti, bw=bwidth)
> NROW(densityv$y)
> x11(width=6, height=5)
> plot(densityv, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      col="blue", lwd=3, main="Density of VTI Returns")
> # Interpolate the densityv vector into returns
> densityv <- approx(densityv$x, densityv$y, xout=retvti)
> all.equal(densityv$x, retvti)
> plot(densityv, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      t="l", col="blue", lwd=3,
+      main="Density of VTI Returns")
```

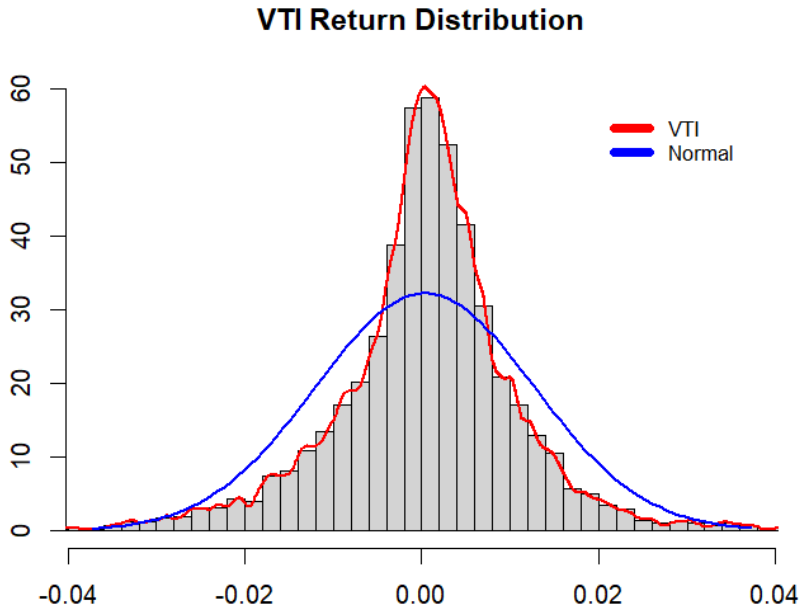
Distribution of Asset Returns

Asset returns are usually not normally distributed and they exhibit *leptokurtosis* (large kurtosis, or fat tails).

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

The parameter `breaks` is the number of cells of the histogram.

The function `lines()` draws a line through specified points.




```
> # Plot histogram
> histp <- hist(retvti, breaks=100, freq=FALSE,
+   xlim=c(-5*madv, 5*madv), xlab="", ylab="",
+   main="VTI Return Distribution")
> # Draw kernel density of histogram
> lines(densityv, col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(retvti), sd=sd(retvti)),
+ add=TRUE, lwd=2, col="blue")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("VTI", "Normal"), bty="n",
+  lwd=6, bg="white", col=c("red", "blue"))
```

depr: Distribution of Asset Returns

Asset returns are usually not normally distributed and they exhibit *leptokurtosis* (large kurtosis, or fat tails).

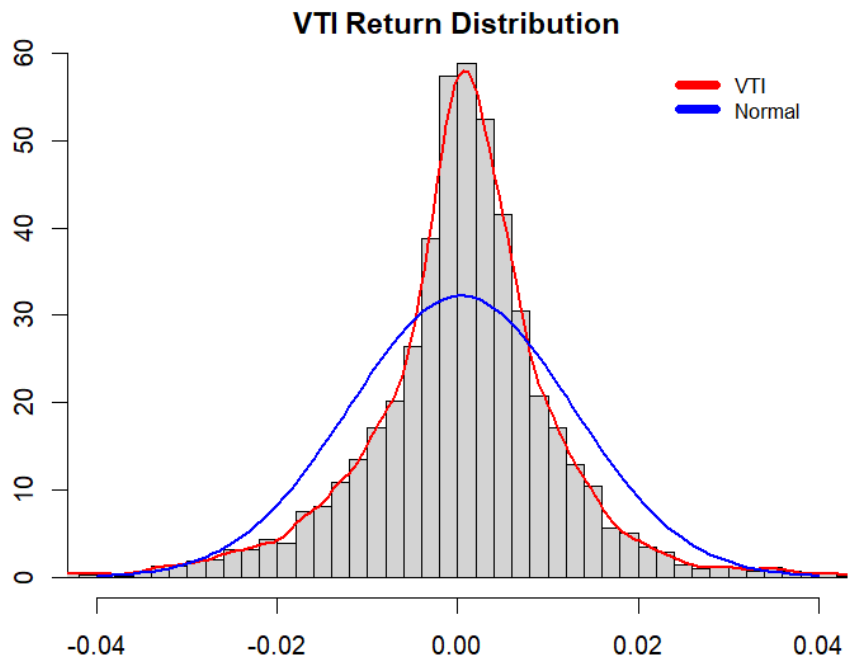
The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

The parameter `breaks` is the number of cells of the histogram.

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

The function `lines()` draws a line through specified points.

```
> library(rutils)  # Load package rutils
> # Calculate VTI percentage returns
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> # Mean and standard deviation of returns
> c(mean(retvti), sd(retvti))
```




```
> # Plot histogram
> x11(width=6, height=5)
> par(mar=c(1, 1, 1, 1), oma=c(2, 2, 2, 0))
> madv <- mad(retvti)
> histp <- hist(retvti, breaks=100,
+   main="", xlim=c(-5*madv, 5*madv),
+   xlab="", ylab="", freq=FALSE)
> # Draw kernel density of histogram
> lines(density(retvti), col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(retvti), sd=sd(retvti)),
+ add=TRUE, type="l", lwd=2, col="blue")
> title(main="VTI Return Distribution", line=0)  # Add title
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("VTI", "Normal"), bty="n",
```

The Quantile-Quantile Plot

A *Quantile-Quantile* (*Q-Q*) plot is a plot of points with the same *quantiles*, from two probability distributions.

If the two distributions are similar then all the points in the *Q-Q* plot lie along the diagonal.

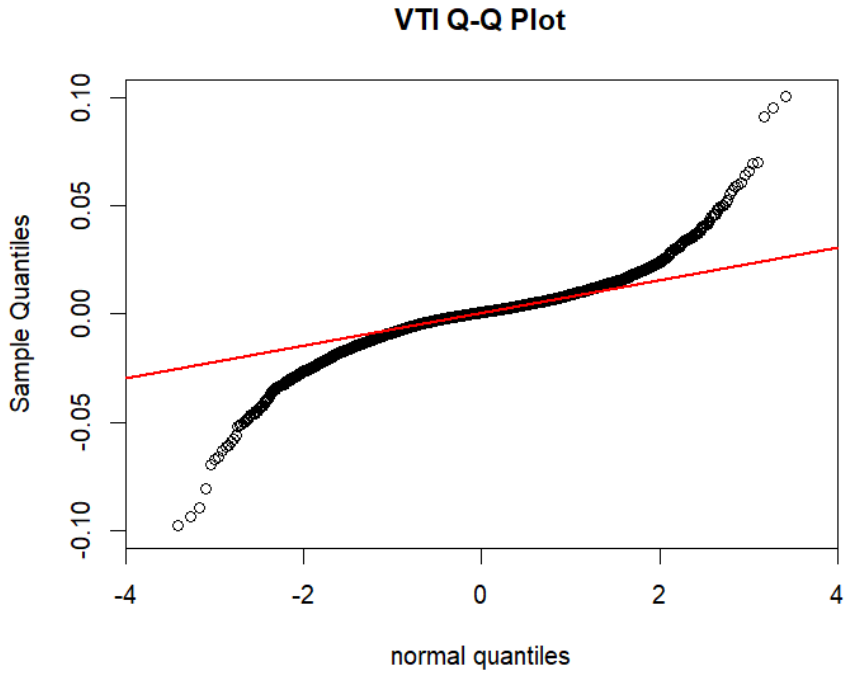
The *VTI Q-Q* plot shows that the *VTI* return distribution has fat tails.

The *p*-value of the *Shapiro-Wilk* test is very close to zero, which shows that the *VTI* returns are very unlikely to be normal.

The function `shapiro.test()` performs the *Shapiro-Wilk* test of normality.

The function `qqnorm()` produces a normal *Q-Q* plot.

The function `qqline()` fits a line to the normal quantiles.




```
> # Create normal Q-Q plot
> qqnorm(retvti, ylim=c(-0.1, 0.1), main="VTI Q-Q Plot",
+  xlab="Normal Quantiles")
> # Fit a line to the normal quantiles
> qqline(retvti, col="red", lwd=2)
> # Perform Shapiro-Wilk test
> shapiro.test(retvti[1:4999])
```

Boxplots of Distributions of Values

Box-and-whisker plots (*boxplots*) are graphical representations of a distribution of values.

The bottom and top box edges (*hinges*) are equal to the first and third quartiles, and the *box* width is equal to the interquartile range (*IQR*).

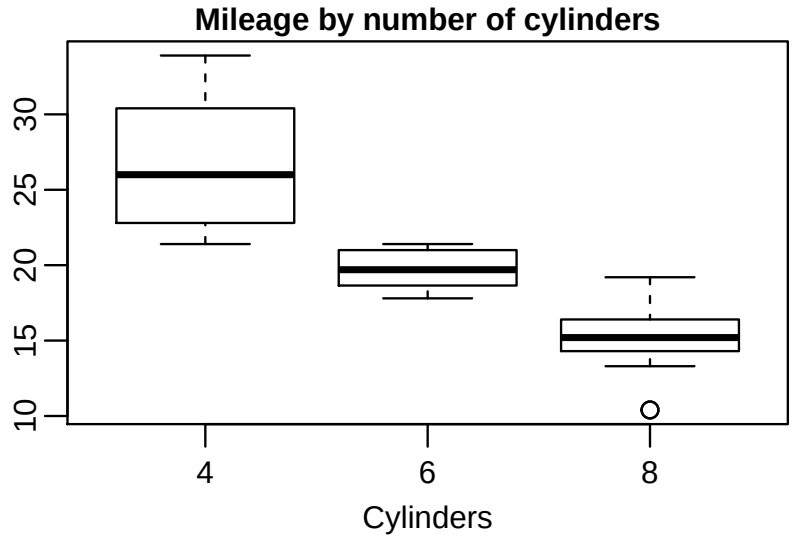
The nominal range is equal to 1.5 times the *IQR* above and below the box *hinges*.

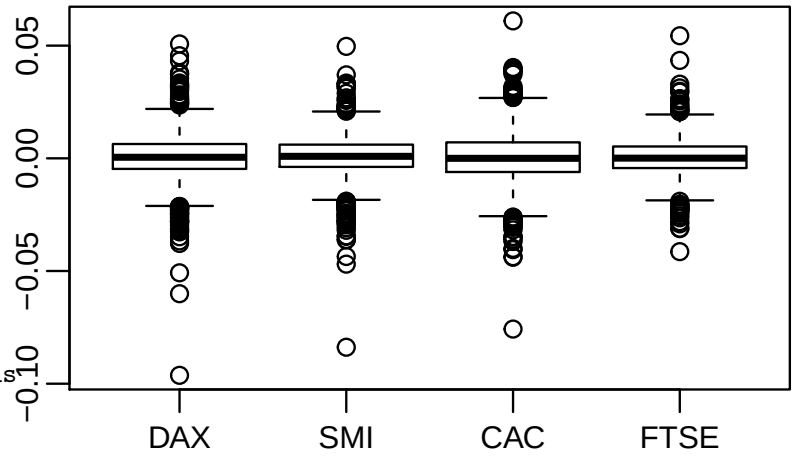
The *whiskers* are dashed vertical lines representing values beyond the first and third quartiles, but within the nominal range.

The *whiskers* end at the last values within the nominal range, while the open circles represent outlier values beyond the nominal range.

The function `boxplot()` has two `methods`: one for `formula` objects (for categorical variables), and another for `data frames`.

```
> # Boxplot method for formula
> boxplot(formula=mpg ~ cyl, data=mtcars,
+   main="Mileage by number of cylinders",
+   xlab="Cylinders", ylab="Miles per gallon")
> # Boxplot method for data frame of EuStockMarkets percentage returns
> boxplot(x=diff(log(EuStockMarkets)))
```





Higher Moments of Asset Returns

The estimators of moments of a probability distribution are given by:

Sample mean: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

Sample variance: $\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$

With their expected values equal to the population mean and standard deviation:
$\mathbb{E}[\bar{x}] = \mu$ and $\mathbb{E}[\hat{\sigma}] = \sigma$

The sample skewness (third moment):

$$\varsigma = \frac{n}{(n-1)(n-2)}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\hat{\sigma}}\right)^3$$

The sample kurtosis (fourth moment):

$$\kappa = \frac{n(n+1)}{(n-1)(n-2)(n-3)}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\hat{\sigma}}\right)^4$$

The normal distribution has skewness equal to 0 and kurtosis equal to 3.

Stock returns typically have negative skewness and kurtosis much greater than 3.

```
> # Calculate VTI percentage returns
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> # Number of observations
> nrows <- NROW(retvti)
> # Mean of VTI returns
> retsm <- mean(retvti)
> # Standard deviation of VTI returns
> sdrets <- sd(retvti)
> # Skewness of VTI returns
> nrows/((nrows-1)*(nrows-2))*sum(((retvti - retsm)/sdrets)^3)
> # Kurtosis of VTI returns
> nrows*(nrows+1)/((nrows-1)^3)*sum(((retvti - retsm)/sdrets)^4)
> # Random normal returns
> retvti <- rnorm(nrows, sd=sdrets)
> # Mean and standard deviation of random normal returns
> retsm <- mean(retvti)
> sdrets <- sd(retvti)
> # Skewness of random normal returns
> nrows/((nrows-1)*(nrows-2))*sum(((retvti - retsm)/sdrets)^3)
> # Kurtosis of random normal returns
> nrows*(nrows+1)/((nrows-1)^3)*sum(((retvti - retsm)/sdrets)^4)
```

Functions for Calculating Skew and Kurtosis

R provides an easy way for users to write functions.

The function `calc_skew()` calculates the skew of returns, and `calc_kurt()` calculates the kurtosis.

Functions return the value of the last expression that is evaluated.

```
> # calc_skew() calculates skew of returns
> calc_skew <- function(retsp) {
+   retsp <- na.omit(retsp)
+   sum(((retsp - mean(retsp))/sd(retsp))^3)/NROW(retsp)
+ }  # end calc_skew
> # calc_kurt() calculates kurtosis of returns
> calc_kurt <- function(retsp) {
+   retsp <- na.omit(retsp)
+   sum(((retsp - mean(retsp))/sd(retsp))^4)/NROW(retsp)
+ }  # end calc_kurt
> # Calculate skew and kurtosis of VTI returns
> calc_skew(retvti)
> calc_kurt(retvti)
> # calcmom() calculates the moments of returns
> calcmom <- function(retsp, moment=3) {
+   retsp <- na.omit(retsp)
+   sum(((retsp - mean(retsp))/sd(retsp))^moment)/NROW(retsp)
+ }  # end calcmom
> # Calculate skew and kurtosis of VTI returns
> calcmom(retvti, moment=3)
> calcmom(retvti, moment=4)
```

Standard Errors of Estimators

Statistical estimators are functions of samples (which are random variables), and therefore are themselves *random variables*.

The *standard error* (SE) of an estimator is defined as its *standard deviation* (not to be confused with the *population standard deviation* of the underlying random variable).

For example, the *standard error* of the estimator of the mean is equal to:

$$\sigma_\mu = \frac{\sigma}{\sqrt{n}}$$

Where σ is the *population standard deviation* (which is usually unkown).

The *estimator* of this *standard error* is equal to:

$$SE_\mu = \frac{\hat{\sigma}}{\sqrt{n}}$$

where: $\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ is the sample standard deviation (the estimator of the population standard deviation).

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Sample mean
> mean(datav)
> # Sample standard deviation
> sd(datav)
> # Standard error of sample mean
> sd(datav)/sqrt(nrows)
```

Normal (Gaussian) Probability Distribution

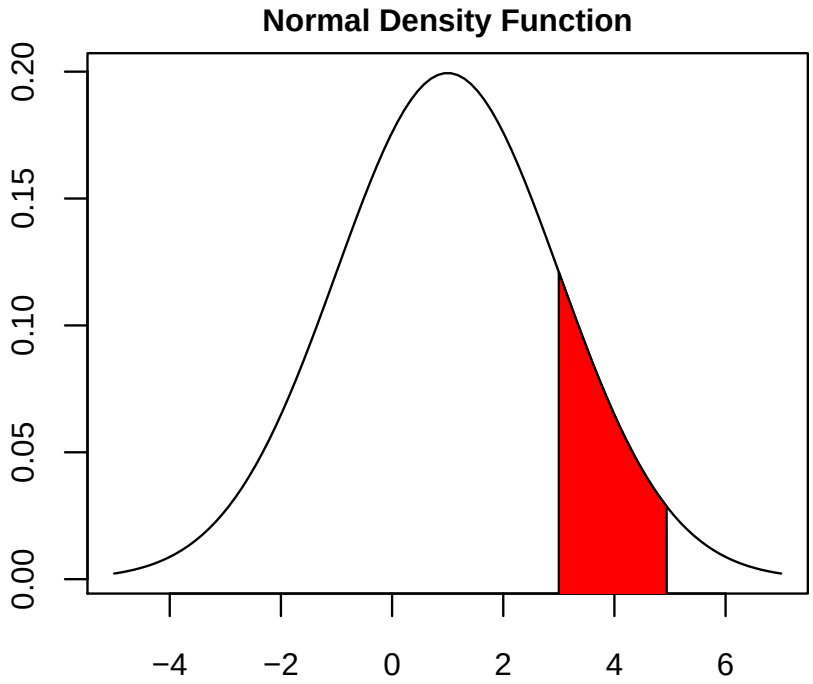
The *Normal (Gaussian)* probability density function is given by:

$$\phi(x, \mu, \sigma) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

The *Standard Normal* distribution $\phi(0, 1)$ is a special case of the *Normal* $\phi(\mu, \sigma)$ with $\mu = 0$ and $\sigma = 1$.

The function `dnorm()` calculates the *Normal* probability density.

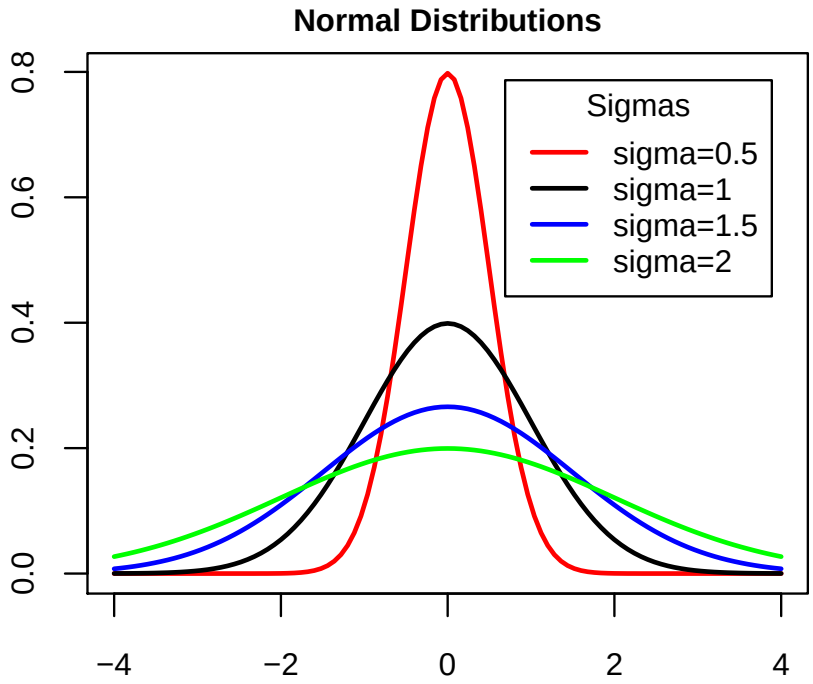
```
> xvar <- seq(-5, 7, length=100)
> yvar <- dnorm(xvar, mean=1.0, sd=2.0)
> plot(xvar, yvar, type="l", lty="solid", xlab="", ylab="")
> title(main="Normal Density Function", line=0.5)
> startp <- 3; endd <- 5  # Set lower and upper bounds
> # Set polygon base
> subv <- ((xvar >= startp) & (xvar <= endd))
> polygon(c(startp, xvar[subv], endd),  # Draw polygon
+   c(-1, yvar[subv], -1), col="red")
```



Normal (Gaussian) Probability Distributions

Plots of several *Normal* distributions with different values of σ, using the function `curve()` for plotting functions given by their name.

```
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colorv <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> for (it in 1:4) {  # Plot four curves
+   curve(expr=dnorm(x, sd=sigmavs[it]),
+   xlim=c(-4, 4), xlab="", ylab="", lwd=2,
+   col=colorv[it], add=as.logical(it-1))
+ }  # end for
> # Add title
> title(main="Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, title="Sigmas",
+  labelv, cex=0.8, lwd=2, lty=1, bty="n", col=colorv)
```



Student's *t-distribution*

Let $z_1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu}\sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu-1}\sum_{i=1}^{\nu}(z_i - \bar{z})^2$

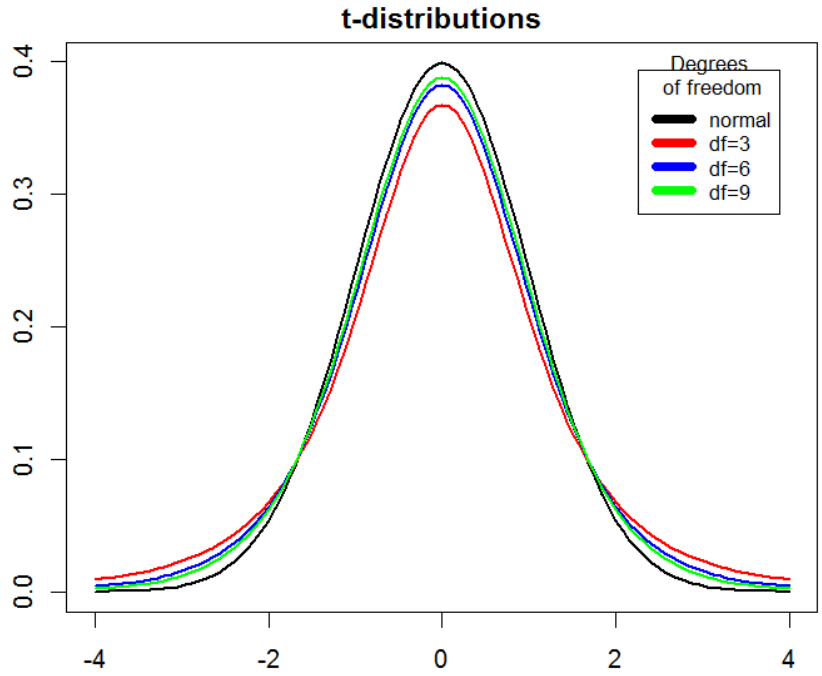
Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma}/\sqrt{\nu}}$$

Follows the *t-distribution* with ν degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}\left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

```
> degf <- c(3, 6, 9)  # Df values
> colorv <- c("black", "red", "blue", "green")
> labelv <- c("normal", paste("df", degf, sep="="))
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-4, 4), xlab="", ylab="", lwd=2)
> for (it in 1:3) {  # Plot three t-distributions
+   curve(expr=dt(x, df=degf[it]), xlab="", ylab="",
+ lwd=2, col=colorv[it+1], add=TRUE)
+ }  # end for
```




```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

Mixture Models of Returns

Mixture models are produced by randomly sampling data from different distributions.

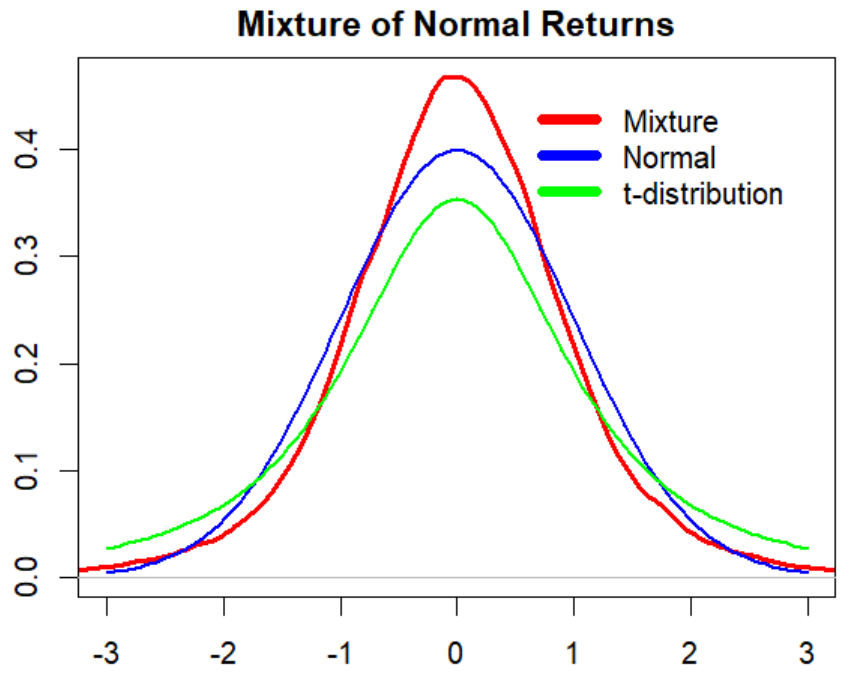
The mixture of two normal distributions with different variances produces a distribution with *leptokurtosis* (large kurtosis, or fat tails).

Student's *t-distribution* has fat tails because the sample variance in the denominator of the *t-ratio* is variable.

The time-dependent volatility of asset returns is referred to as *heteroskedasticity*.

Random processes with *heteroskedasticity* can be considered a type of mixture model.

The *heteroskedasticity* produces *leptokurtosis* (large kurtosis, or fat tails).




```
> # Mixture of two normal distributions with sd=1 and sd=2
> nrows <- 1e5
> retsp <- c(rnorm(nrows/2), 2*rnorm(nrows/2))
> retsp <- (retsp-mean(retsp))/sd(retsp)
> # Kurtosis of normal
> calc_kurt(rnorm(nrows))
> # Kurtosis of mixture
> calc_kurt(retsp)
> # Or
> nrows*sum(retsp^4)/(nrows-1)^2
```

```
> # Plot the distributions
> plot(density(retsp), xlab="", ylab="",
+   main="Mixture of Normal Returns",
+   xlim=c(-3, 3), type="l", lwd=3, col="red")
> curve(expr=dnorm, lwd=2, col="blue", add=TRUE)
> curve(expr=dt(x, df=3), lwd=2, col="green", add=TRUE)
> # Add legend
> legend("topright", inset=0.05, lty=1, lwd=6, bty="n",
+   legend=c("Mixture", "Normal", "t-distribution"),
+   col=c("red", "blue", "green"))
```

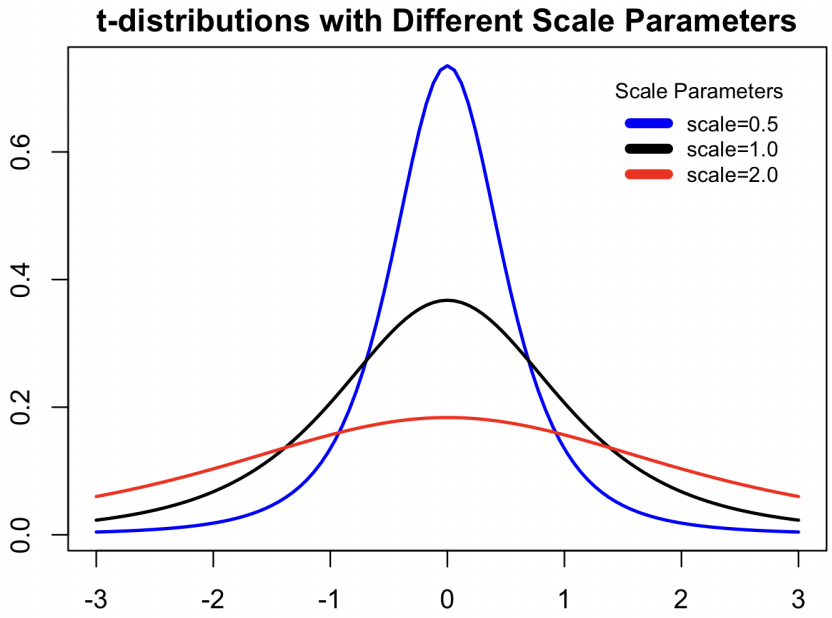
Non-standard Student's *t-distribution*

The non-standard Student's *t-distribution* has the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\sigma\,\Gamma(\nu/2)}\,(1+(\frac{t-\mu}{\sigma})^2/\nu)^{-(\nu+1)/2}$$

It has non-zero mean equal to the location parameter μ, and a standard deviation proportional to the scale parameter σ.

```
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # x11(width=6, height=5)
> # Define density of non-standard t-distribution
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   dt((x-locv)/scalev, df=dfree)/scalev
+ }  # end tdistr
> # Or
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   gamma((dfree+1)/2)/(sqrt(pi*dfree)*gamma(dfree/2)*scalev)*
+     (1+((x-locv)/scalev)^2/dfree)^(-(dfree+1)/2)
+ }  # end tdistr
> # Calculate vector of scale values
> scalev <- c(0.5, 1.0, 2.0)
> colorv <- c("blue", "black", "red")
> labelv <- paste("scale", format(scalev, digits=2), sep="=")
> # Plot three t-distributions
> for (it in 1:3) {
+   curve(expr=tdistr(x, dfree=3, scalev=scalev[it]), xlim=c(-3, 3),
+ xlab="", ylab="", lwd=2, col=colorv[it], add=(it>1))
+ }  # end for
```




```
> # Add title
> title(main="t-distributions with Different Scale Parameters", line
> # Add legend
> legend("topright", inset=0.05, bty="n", title="Scale Parameters",
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

The *Shapiro-Wilk* Test of Normality

The *Shapiro-Wilk* test is designed to test the *null hypothesis* that a sample: $\{x_1, \ldots, x_n\}$ is from a normally distributed population.

The test statistic is equal to:

$$W = \frac{(\sum_{i=1}^{n} a_i x_{(i)})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Where the: $\{a_1, \ldots, a_n\}$ are proportional to the *order statistics* of random variables from the normal distribution.

$x_{(k)}$ is the k-th *order statistic*, and is equal to the k-th smallest value in the sample: $\{x_1, \ldots, x_n\}$.

The *Shapiro-Wilk* statistic follows its own distribution, and is less than or equal to 1.

The *Shapiro-Wilk* statistic is close to 1 for samples from normal distributions.

The p-value for *VTI* returns is extremely small, and we conclude that the *null hypothesis* is FALSE, and the *VTI* returns are not from a normally distributed population.

The *Shapiro-Wilk* test is not reliable for large sample sizes, so it's limited to less than 5000 sample size.

```
> # Calculate VTI percentage returns
> library(rutils)
> retvti <- as.numeric(na.omit(rutils::etfenv$returns$VTI))[1:4999]
> # Reduce number of output digits
> ndigits <- options(digits=5)
> # Shapiro-Wilk test for normal distribution
> nrows <- NROW(retvti)
> shapiro.test(rnorm(nrows))

Shapiro-Wilk normality test

data:  rnorm(nrows)
W = 1, p-value = 0.8
> # Shapiro-Wilk test for VTI returns
> shapiro.test(retvti)

Shapiro-Wilk normality test

data:  retvti
W = 0.886, p-value <2e-16
> # Shapiro-Wilk test for uniform distribution
> shapiro.test(runif(nrows))

Shapiro-Wilk normality test

data:  runif(nrows)
W = 0.954, p-value <2e-16
> # Restore output digits
> options(digits=ndigits$digits)
```

The *Jarque-Bera* Test of Normality

The *Jarque-Bera* test is designed to test the *null hypothesis* that a sample: $\{x_1, \ldots, x_n\}$ is from a normally distributed population.

The test statistic is equal to:

$$JB = \frac{n}{6}(\varsigma^2 + \frac{1}{4}(\kappa - 3)^2)$$

Where the *skewness* and *kurtosis* are defined as:

$$\varsigma = \frac{1}{n}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\hat{\sigma}})^3 \qquad \kappa = \frac{1}{n}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\hat{\sigma}})^4$$

The *Jarque-Bera* statistic asymptotically follows the *chi-squared* distribution with 2 degrees of freedom.

The *Jarque-Bera* statistic is small for samples from normal distributions.

The *p*-value for *VTI* returns is extremely small, and we conclude that the *null hypothesis* is `FALSE`, and the *VTI* returns are not from a normally distributed population.

```
> library(tseries)  # Load package tseries
> # Jarque-Bera test for normal distribution
> jarque.bera.test(rnorm(nrows))

Jarque Bera Test

data:  rnorm(nrows)
X-squared = 8, df = 2, p-value = 0.02
> # Jarque-Bera test for VTI returns
> jarque.bera.test(retvti)

Jarque Bera Test

data:  retvti
X-squared = 28386, df = 2, p-value <2e-16
> # Jarque-Bera test for uniform distribution
> jarque.bera.test(runif(NROW(retvti)))

Jarque Bera Test

data:  runif(NROW(retvti))
X-squared = 311, df = 2, p-value <2e-16
```

The *Kolmogorov-Smirnov* Test for Probability Distributions

The *Kolmogorov-Smirnov* test *null hypothesis* is that two samples: $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$ were obtained from the same probability distribution.

The *Kolmogorov-Smirnov* statistic depends on the maximum difference between two empirical cumulative distribution functions (cumulative frequencies):

$$D = \sup_i |P(x_i) - P(y_i)|$$

The function `ks.test()` performs the *Kolmogorov-Smirnov* test and returns the statistic and its *p*-value *invisibly*.

The second argument to `ks.test()` can be either a `numeric` vector of data values, or a name of a cumulative distribution function.

The *Kolmogorov-Smirnov* test can be used as a *goodness of fit* test, to test if a set of observations fits a probability distribution.

```
> # KS test for normal distribution
> ks_test <- ks.test(rnorm(100), pnorm)
> ks_test$p.value
> # KS test for uniform distribution
> ks.test(runif(100), pnorm)
> # KS test for two shifted normal distributions
> ks.test(rnorm(100), rnorm(100, mean=0.1))
> ks.test(rnorm(100), rnorm(100, mean=1.0))
> # KS test for two different normal distributions
> ks.test(rnorm(100), rnorm(100, sd=2.0))
> # KS test for VTI returns vs normal distribution
> retvti <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> retvti <- (retvti - mean(retvti))/sd(retvti)
> ks.test(retvti, pnorm)
```

Chi-squared Distribution

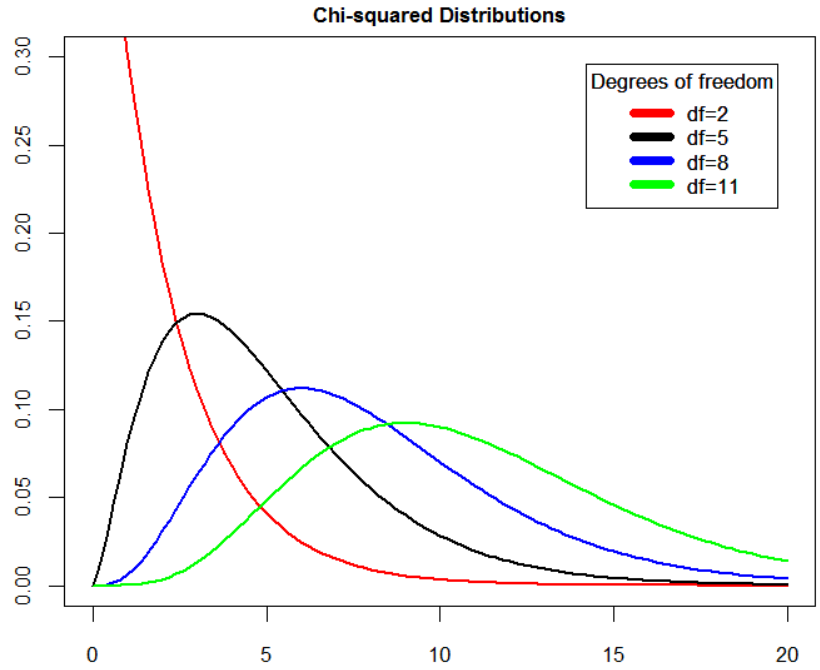
Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with k degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1}\, e^{-x/2}}{2^{k/2}\, \Gamma(k/2)}$$

The *Chi-squared* distribution with k degrees of freedom has mean equal to k and variance equal to $2k$.

```
> # Degrees of freedom
> degf <- c(2, 5, 8, 11)
> # Plot four curves in loop
> colorv <- c("red", "black", "blue", "green")
> for (it in 1:4) {
+   curve(expr=dchisq(x, df=degf[it]),
+   xlim=c(0, 20), ylim=c(0, 0.3),
+   xlab="", ylab="", col=colorv[it],
+   lwd=2, add=as.logical(it-1))
+ }  # end for
```




```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, bty="n",
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

The *Chi-squared* Test for the Goodness of Fit

Goodness of Fit tests are designed to test if a set of observations fits an assumed theoretical probability distribution.

The *Chi-squared* test tests if a frequency of counts fits the specified distribution.

The *Chi-squared* statistic is the sum of squared differences between the observed frequencies o_i and the theoretical frequencies p_i:

$$\chi^2 = N \sum_{i=1}^{n} \frac{(o_i - p_i)^2}{p_i}$$

Where N is the total number of observations.

The *null hypothesis* is that the observed frequencies are consistent with the theoretical distribution.

The function `chisq.test()` performs the *Chi-squared* test and returns the statistic and its p-value *invisibly*.

The parameter `breaks` in the function `hist()` should be chosen large enough to capture the shape of the frequency distribution.

```
> # Observed frequencies from random normal data
> histp <- hist(rnorm(1e3, mean=0), breaks=100, plot=FALSE)
> countsn <- histp$counts
> # Theoretical frequencies
> countst <- rutils::diffit(pnorm(histp$breaks))
> # Perform Chi-squared test for normal data
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
> # Return p-value
> chisq_test <- chisq.test(x=countsn, p=countst, rescale.p=TRUE, sim
> chisq_test$p.value
> # Observed frequencies from shifted normal data
> histp <- hist(rnorm(1e3, mean=2), breaks=100, plot=FALSE)
> countsn <- histp$counts/sum(histp$counts)
> # Theoretical frequencies
> countst <- rutils::diffit(pnorm(histp$breaks))
> # Perform Chi-squared test for shifted normal data
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
> # Calculate histogram of VTI returns
> histp <- hist(retvti, breaks=100, plot=FALSE)
> countsn <- histp$counts
> # Calculate cumulative probabilities and then difference them
> countst <- pt((histp$breaks-locv)/scalev, df=2)
> countst <- rutils::diffit(countst)
> # Perform Chi-squared test for VTI returns
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
```

The Likelihood Function of Student's *t-distribution*

The non-standard Student's *t-distribution* is:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\sigma\,\Gamma(\nu/2)}\,(1+(\frac{t-\mu}{\sigma})^2/\nu)^{-(\nu+1)/2}$$

It has non-zero mean equal to the location parameter μ, and a standard deviation proportional to the scale parameter σ.

The negative logarithm of the probability density is equal to:

$$-\log(f(t)) = -\log(\frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}) + \log(\sigma) + \frac{\nu+1}{2}\,\log(1+(\frac{t-\mu}{\sigma})^2/\nu)$$

The *likelihood* function $\mathcal{L}(\theta|\bar{x})$ is a function of the model parameters θ, given the observed values $\bar{x}$, under the model's probability distribution $f(x|\theta)$:

$$\mathcal{L}(\theta|x) = \prod_{i=1}^{n} f(x_i|\theta)$$

```
> # Objective function from function dt()
> likefun <- function(par, dfree, data) {
+   -sum(log(dt(x=(data-par[1])/par[2], df=dfree)/par[2]))
+ }  # end likefun
> # Demonstrate equivalence with log(dt())
> likefun(c(1, 0.5), 2, 2:5)
> -sum(log(dt(x=(2:5-1)/0.5, df=2)/0.5))
> # Objective function is negative log-likelihood
> likefun <- function(par, dfree, data) {
+   sum(-log(gamma((dfree+1)/2)/(sqrt(pi*dfree)*gamma(dfree/2))) +
+     log(par[2]) + (dfree+1)/2*log(1+((data-par[1])/par[2])^2/dfree
+ }  # end likefun
```

The *likelihood* function measures how *likely* are the parameters, given the observed values $\bar{x}$.

The *maximum-likelihood* estimate (*MLE*) of the parameters are those that maximize the *likelihood* function:

$$\theta_{MLE} = \arg\max_{\theta} \mathcal{L}(\theta|x)$$

In practice the logarithm of the *likelihood* $\log(\mathcal{L})$ is maximized, instead of the *likelihood* itself.

Fitting Asset Returns into Student's *t-distribution*

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

The function `fitdistr()` performs a *maximum likelihood* optimization to find the non-standardized Student's *t-distribution* location and scale parameters.

```
> # Calculate VTI percentage returns
> retvti <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> # Fit VTI returns using MASS::fitdistr()
> fitobj <- MASS::fitdistr(retvti, densfun="t", df=3)
> summary(fitobj)
> # Fitted parameters
> fitobj$estimate
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
> locv; scalev
> # Standard errors of parameters
> fitobj$sd
> # Log-likelihood value
> fitobj$value
> # Fit distribution using optim()
> initp <- c(mean=0, scale=0.01)  # Initial parameters
> fitobj <- optim(par=initp,
+   fn=likefun, # Log-likelihood function
+   data=retvti,
+   dfree=3, # Degrees of freedom
+   method="L-BFGS-B", # Quasi-Newton method
+   upper=c(1, 0.1), # Upper constraint
+   lower=c(-1, 1e-7)) # Lower constraint
> # Optimal parameters
> locv <- fitobj$par["mean"]
> scalev <- fitobj$par["scale"]
> locv; scalev
```

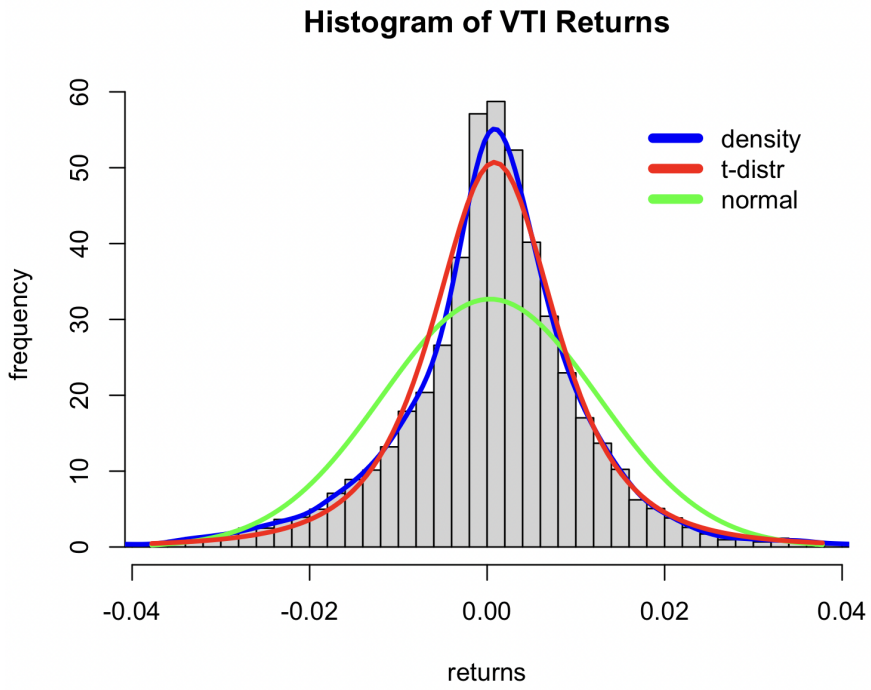
The Student's *t-distribution* Fitted to Asset Returns

Asset returns typically exhibit *negative skewness* and *large kurtosis* (leptokurtosis), or fat tails.

Stock returns fit the non-standard *t-distribution* with 3 degrees of freedom quite well.

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

The parameter `breaks` is the number of cells of the histogram.




```
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # x11(width=6, height=5)
> # Plot histogram of VTI returns
> madv <- mad(retvti)
> histp <- hist(retvti, col="lightgrey",
+   xlab="returns", breaks=100, xlim=c(-5*madv, 5*madv),
+   ylab="frequency", freq=FALSE, main="Histogram of VTI Returns")
> lines(density(retvti, adjust=1.5), lwd=3, col="blue")
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(retvti),
+   sd=sd(retvti)), add=TRUE, lwd=3, col="green")
> # Define non-standard t-distribution
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   dt((x-locv)/scalev, df=dfree)/scalev
+ }  # end tdistr
> # Plot t-distribution function
> curve(expr=tdistr(x, dfree=3, locv=locv, scalev=scalev), col="red", lwd=3, add=TRUE)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("density", "t-distr", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Goodness of Fit of Student's *t-distribution* Fitted to Asset Returns

The *Q-Q* plot illustrates the relative distributions of two samples of data.

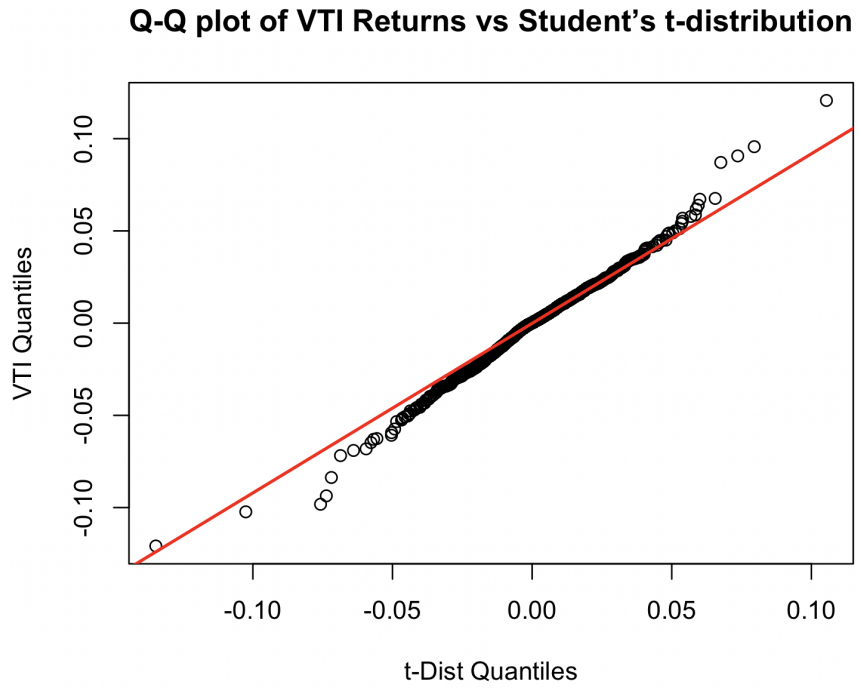
The *Q-Q* plot shows that stock returns fit the non-standard *t-distribution* with 3 degrees of freedom quite well.

The function `qqplot()` produces a *Q-Q* plot for two samples of data.

The function `ks.test()` performs the *Kolmogorov-Smirnov* test for the similarity of two distributions.

The *null hypothesis* of the *Kolmogorov-Smirnov* test is that the two samples were obtained from the same probability distribution.

The *Kolmogorov-Smirnov* test rejects the *null hypothesis* that stock returns follow closely the non-standard *t-distribution* with 3 degrees of freedom.




```
> # Calculate sample from non-standard t-distribution with df=3
> tdata <- scalev*rt(NROW(retvti), df=3) + locv
> # Q-Q plot of VTI Returns vs non-standard t-distribution
> qqplot(tdata, retvti, xlab="t-Dist Quantiles", ylab="VTI Quantile
+        main="Q-Q plot of VTI Returns vs Student's t-distribution"
> # Calculate quartiles of the distributions
> probs <- c(0.25, 0.75)
> qrets <- quantile(retvti, probs)
> qtdata <- quantile(tdata, probs)
> # Calculate slope and plot line connecting quartiles
> slope <- diff(qrets)/diff(qtdata)
> intercept <- qrets[1]-slope*qtdata[1]
> abline(intercept, slope, lwd=2, col="red")
```

```
> # KS test for VTI returns vs t-distribution data
> ks.test(retvti, tdata)
> # Define cumulative distribution of non-standard t-distribution
> ptdistr <- function(x, dfree, locv=0, scalev=1) {
+   pt((x-locv)/scalev, df=dfree)
+ }  # end ptdistr
> # KS test for VTI returns vs cumulative t-distribution
> ks.test(sample(retvti, replace=TRUE), ptdistr, dfree=3, locv=locv
```

Leptokurtosis Fat Tails of Asset Returns

The probability under the *normal* distribution decreases exponentially for large values of x:

$$\phi(x) \propto e^{-x^2/2\sigma^2} \qquad (as\ |x| \to \infty)$$

This is because a normal variable can be thought of as the sum of a large number of independent binomial variables of equal size.

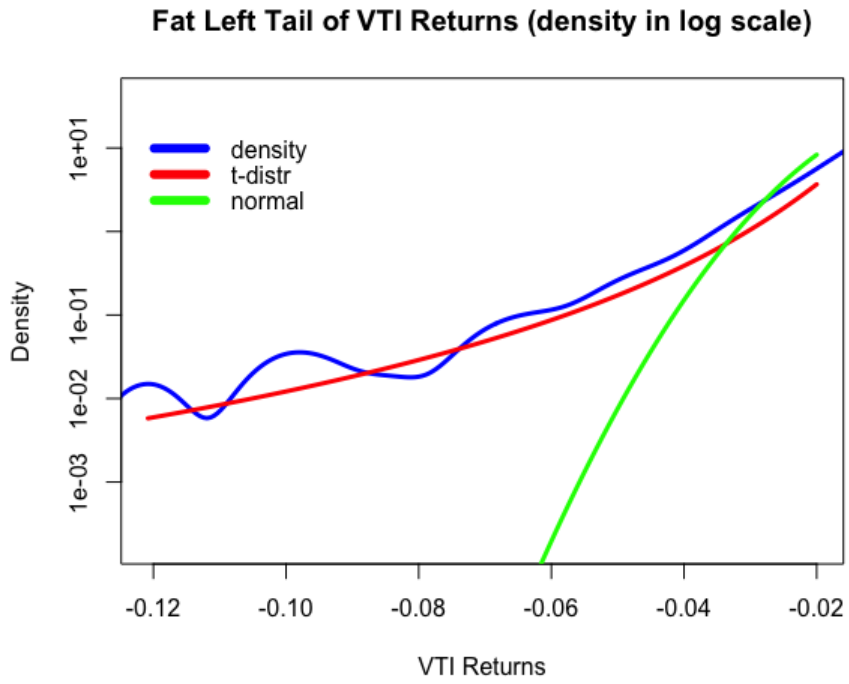
So large values are produced only when all the contributing binomial variables are of the same sign, which is very improbable, so it produces extremely low tail probabilities (thin tails),

But in reality, the probability of large negative asset returns decreases much slower, as the negative power of the returns (fat tails).

The probability under Student's *t-distribution* decreases as a power for large values of x:

$$f(x) \propto |x|^{-(\nu+1)} \qquad (as\ |x| \to \infty)$$

This is because a *t-variable* can be thought of as the sum of normal variables with different volatilities (different sizes).



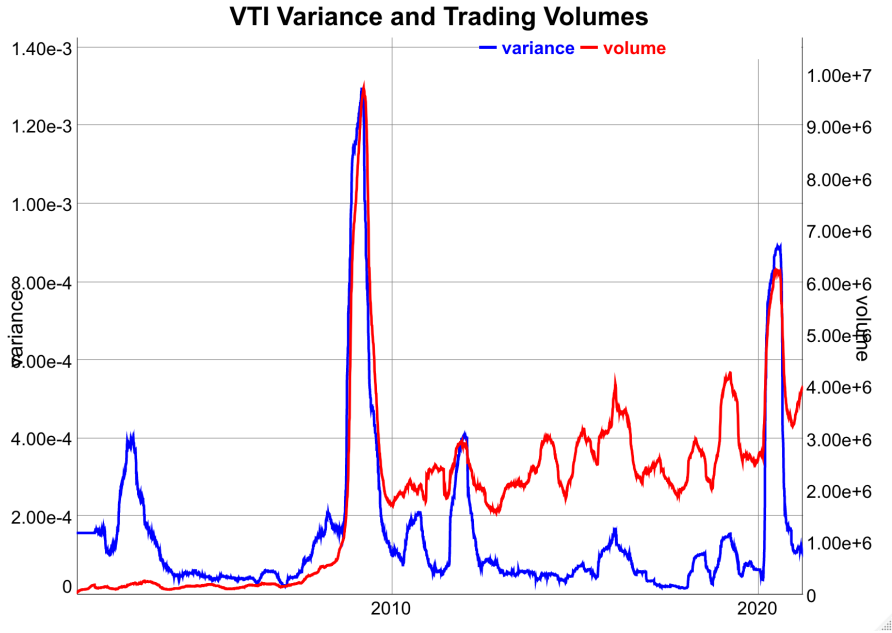

```
> # Plot log density of VTI returns
> plot(density(retvti, adjust=4), xlab="VTI Returns", ylab="Density"
+      main="Fat Left Tail of VTI Returns (density in log scale)",
+      type="l", lwd=3, col="blue", xlim=c(min(retvti), -0.02), log=
> # Plot t-distribution function
> curve(expr=dt((x-locv)/scalev, df=3)/scalev, lwd=3, col="red", add
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(retvti), sd=sd(retvti)), lwd=3, col=
> # Add legend
> legend("topleft", inset=0.01, bty="n", y.intersp=c(0.25, 0.25, 0.2
+   leg=c("density", "t-distr", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Trading Volumes

The rolling average trading volumes have increased significantly since the 2008 crisis, mostly because of high frequency trading (HFT).

Higher levels of volatility coincide with higher *trading volumes*.

The time-dependent volatility of asset returns (*heteroskedasticity*) produces their fat tails (*leptokurtosis*).




```
> # Calculate VTI returns and trading volumes
> ohlc <- rutils::etfenv$VTI
> closep <- drop(coredata(quantmod::Cl(ohlc)))
> retvti <- rutils::diffit(log(closep))
> volumes <- coredata(quantmod::Vo(ohlc))
> # Calculate rolling variance
> look_back <- 121
> varv <- HighFreq::roll_var_ohlc(log(ohlc), method="close", look_back=look_back, scale=FALSE)
> varv[1:look_back, ] <- varv[look_back+1, ]
> # Calculate rolling average volume
> volume_roll <- HighFreq::roll_vec(volumes, look_back=look_back)/look_back
> # dygraph plot of VTI variance and trading volumes
> datav <- xts::xts(cbind(varv, volume_roll), zoo::index(ohlc))
> colnamev <- c("variance", "volume")
> colnames(datav) <- colnamev
> dygraphs::dygraph(datav, main="VTI Variance and Trading Volumes") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], strokeWidth=2, axis="y", col="blue") %>%
+   dySeries(name=colnamev[2], strokeWidth=2, axis="y2", col="red")
```

Asset Returns in Trading Time

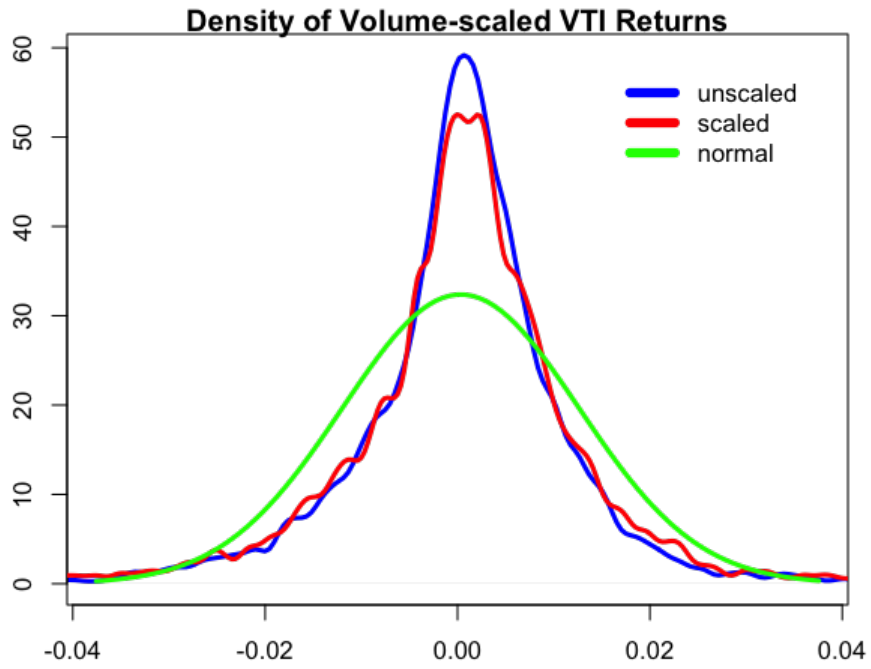
The time-dependent volatility of asset returns (*heteroskedasticity*) produces their fat tails (*leptokurtosis*).

If asset returns were measured at fixed intervals of *trading volumes* (*trading time* instead of clock time), then the volatility would be lower and less time-dependent.

The asset returns can be adjusted to *trading time* by dividing them by the *square root of the trading volumes*, to obtain scaled returns over equal trading volumes.

The scaled returns have a more positive *skewness* and a smaller *kurtosis* than unscaled returns.

```
> # Scale returns using volume (volume clock)
> rets_scaled <- ifelse(volumes > 0,
+   sqrt(volume_roll)*retvti/sqrt(volumes), 0)
> rets_scaled <- sd(retvti)*rets_scaled/sd(rets_scaled)
> # rets_scaled <- ifelse(volumes > 1e4, retvti/volumes, 0)
> # Calculate moments of scaled returns
> nrows <- NROW(retvti)
> sapply(list(retvti=retvti, rets_scaled=rets_scaled),
+   function(rets) {sapply(c(skew=3, kurt=4),
+     function(x) sum((rets/sd(rets))^x)/nrows)
+ })  # end sapply
```




```
> # x11(width=6, height=5)
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> # Plot densities of SPY returns
> madv <- mad(retvti)
> # bwidth <- mad(rutils::diffit(retvti))
> plot(density(retvti, bw=madv/10), xlim=c(-5*madv, 5*madv),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of Volume-scaled VTI Returns")
> lines(density(rets_scaled, bw=madv/10), lwd=3, col="red")
> curve(expr=dnorm(x, mean=mean(retvti), sd=sd(retvti)),
+ add=TRUE, lwd=3, col="green")
> # Add legend
> legend("topright", inset=0.05, bty="n",
```

draft: Central Limit Theorem

Let $x_1, \ldots, x_n$ be independent and identically distributed (i.i.d.) random variables with expected value μ and variance σ^2, and let $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ be their mean.

The random variables x_i don't have to be normally distributed, they only need a finite second moment σ.

The *Central Limit Theorem* states that as $n \to \infty$, then in the limit, the random variable z:

$$z = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}}$$

Follows the *standard normal* distribution $\phi(0, 1)$.

The *normal* distribution is the limiting distribution of sums of random variables which have a finite second moment.

For example, the sums of random variables with fat tails, which decrease as a power for large values of x:

$$f(x) \propto |x|^{-(\nu+1)} \qquad (\textit{with } \nu > 1)$$

Tend to the *standard normal* distribution $\phi(0, 1)$.

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.

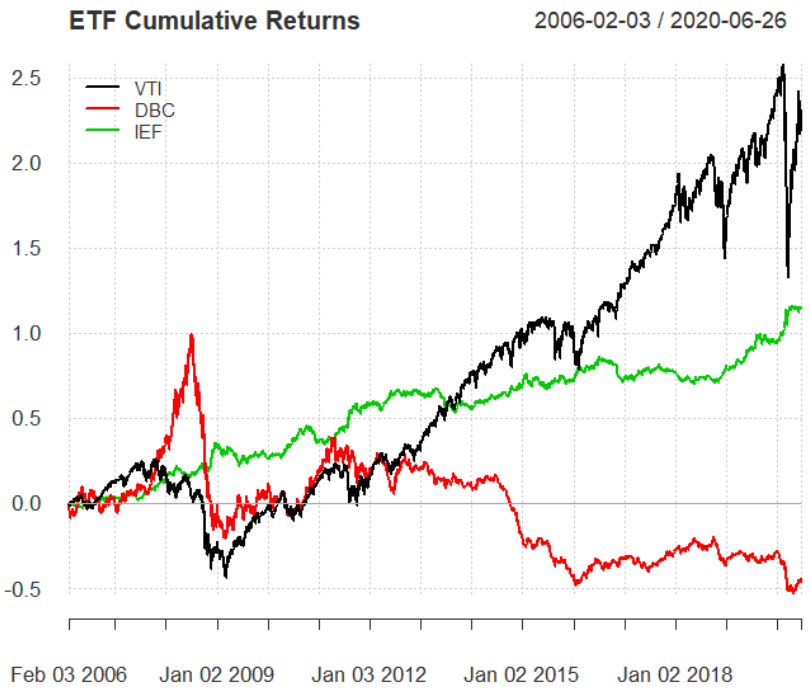
```
> # Load package PerformanceAnalytics
> library(PerformanceAnalytics)
> # Get documentation for package PerformanceAnalytics
> # Get short description
> packageDescription("PerformanceAnalytics")
> # Load help page
> help(package="PerformanceAnalytics")
> # List all objects in PerformanceAnalytics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

```
> perf_data <- unclass(data(
+     package="PerformanceAnalytics"))$results[, -(1:2)]
> apply(perf_data, 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.

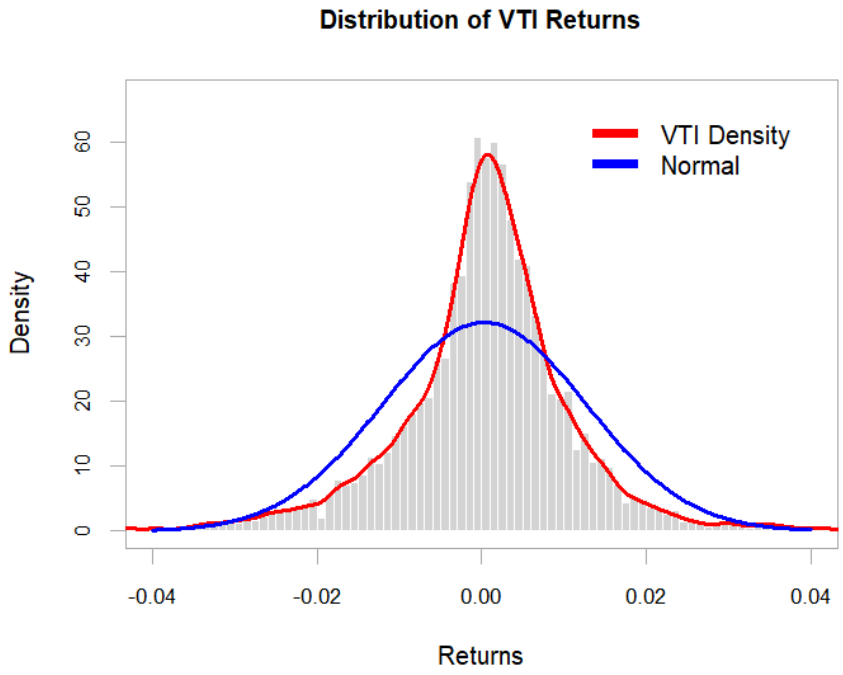
```
> # Load package "PerformanceAnalytics"
> library(PerformanceAnalytics)
> # Calculate ETF returns
> retsp <- rutils::etfenv$returns[, c("VTI", "DBC", "IEF")]
> retsp <- na.omit(retsp)
> # Plot cumulative ETF returns
> x11(width=6, height=5)
> chart.CumReturns(retsp, lwd=2, ylab="",
+   legend.loc="topleft",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

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



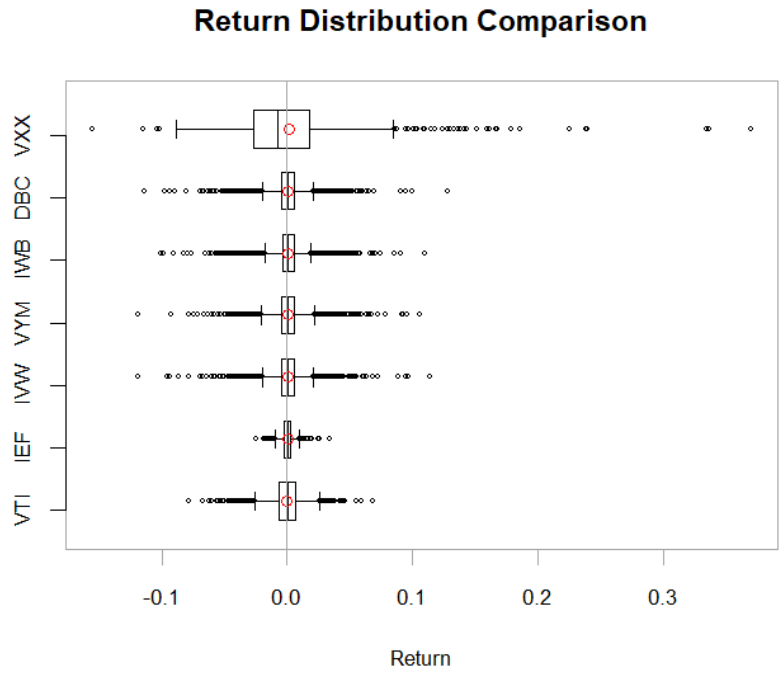
Boxplots of Returns

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).

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



The Median Absolute Deviation Estimator of Dispersion

The *Median Absolute Deviation* (*MAD*) is a robust 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
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retvti)
> sd(retvti)
> mad(retvti)
> # Bootstrap of sd and mad estimators
> bootd <- sapply(1:10000, function(x) {
+   samplev <- retvti[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 <- retvti[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 <- retvti[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
> returns_sub <- (retvti - targetr)
> returns_sub <- ifelse(returns_sub < 0, returns_sub, 0)
> nrows <- NROW(returns_sub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(returns_sub^2)/nrows),
+   drop(DownsideDeviation(retvti, MAR=targetr, method="full")))
> # Calculate the subset downside returns
> returns_sub <- (retvti - targetr)
> returns_sub <- returns_sub[returns_sub < 0]
> nrows <- NROW(returns_sub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(returns_sub^2)/nrows),
+   drop(DownsideDeviation(retvti, MAR=targetr, method="subset")))
```

Drawdown Risk

The *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
> dates <- zoo::index(closep)
> # Calculate the maximum drawdown date and depth
> indexmin <- which.min(drawdns)
> datemin <- dates[indexmin]
> maxdd <- drawdns[datemin]
> # Calculate the drawdown start and end dates
> startd <- max(dates[(dates < datemin) & (drawdns == 0)])
> endd <- min(dates[(dates > 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, dates)
> 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, dates)
> abline(v=xval, col="red")
> text(x=xval, y=0.9*yval, "max drawdown", col="red", cex=0.9)
> xval <- match(endd, dates)
> 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.

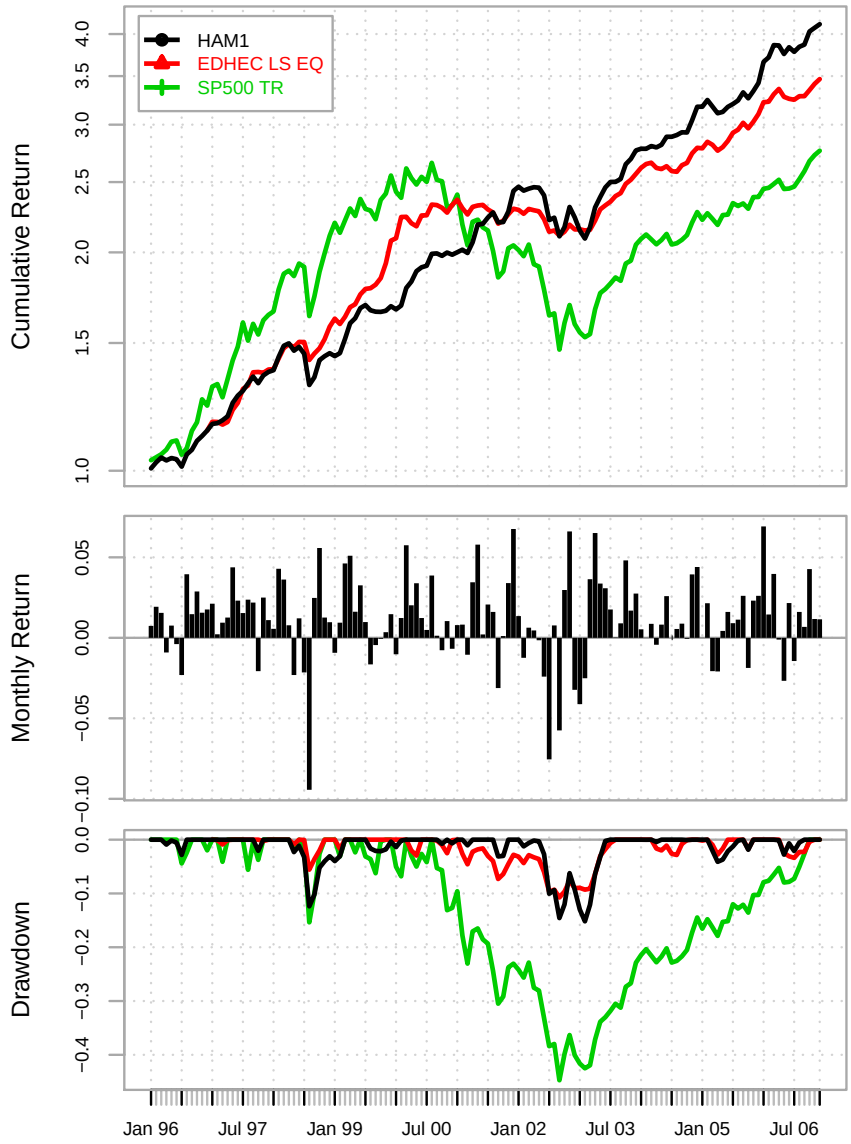
```
> library(xtable)
> library(PerformanceAnalytics)
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> retvti <- rutils::diffit(closep)
> # Calculate table of VTI drawdowns
> tablev <- PerformanceAnalytics::table.Drawdowns(retvti, geometric=FALSE)
> # Convert dates to strings
> tablev <- cbind(sapply(tablev[, 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	2022-06-16		-0.10	149.00	114.00	
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

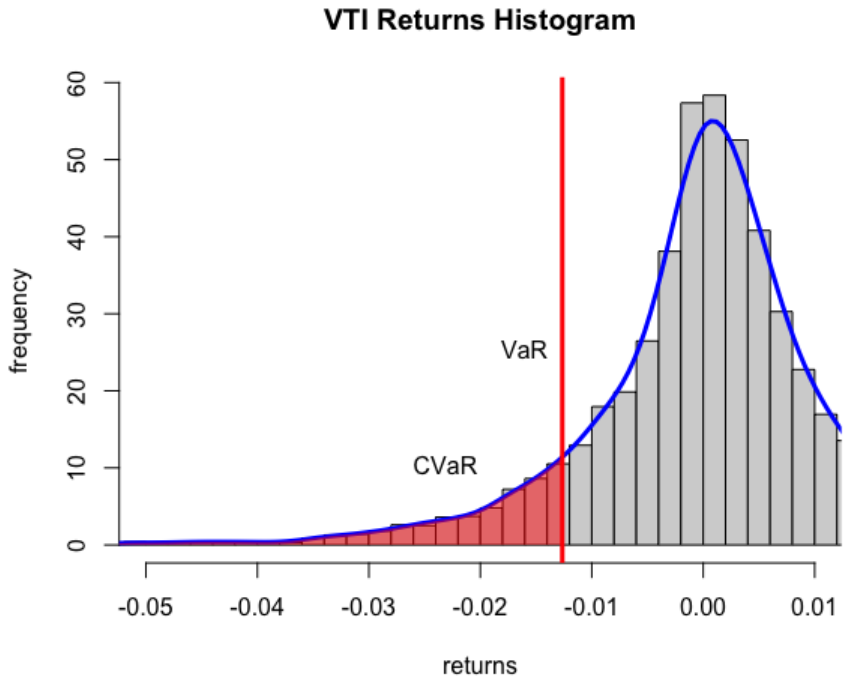
The distribution of returns has a long left tail of negative returns representing the risk of loss.

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
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> confl <- 0.1
> varisk <- quantile(retvti, confl)
> cvar <- mean(retvti[retvti <= 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(retvti, col="lightgrey",
+   xlab="returns", ylab="frequency", breaks=100,
+   xlim=c(-0.05, 0.01), freq=FALSE, main="VTI Returns Histogram")
> # Calculate density
> densv <- density(retvti, 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
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retvti)
> confl <- 0.05
> # Calculate VaR approximately by sorting
> sortv <- sort(as.numeric(retvti))
> cutoff <- round(confl*nrows)
> varisk <- sortv[cutoff]
> # Calculate VaR as quantile
> varisk <- quantile(retvti, probs=confl)
> # PerformanceAnalytics VaR
> PerformanceAnalytics::VaR(retvti, p=(1-confl), method="historical"
> all.equal(unname(varisk),
+   as.numeric(PerformanceAnalytics::VaR(retvti,
+   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(retvti, confl)
> # Calculate CVaR as expected loss
> cvar <- mean(retvti[retvti <= varisk])
> # PerformanceAnalytics VaR
> PerformanceAnalytics::ETL(retvti, p=(1-confl), method="historical"
> all.equal(unname(cvar),
+   as.numeric(PerformanceAnalytics::ETL(retvti,
+     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
> risk_ret <-
+   PerformanceAnalytics::table.Stats(rutils::etfenv$returns)
> class(risk_ret)
> # Transpose the data frame
> risk_ret <- as.data.frame(t(risk_ret))
> # Add Name column
> risk_ret$Name <- rownames(risk_ret)
> # Add Sharpe ratio column
> risk_ret$Sharpe <- risk_ret$"Arithmetic Mean"/risk_ret$Stdev
> # Sort on Sharpe ratio
> risk_ret <- risk_ret[order(risk_ret$Sharpe, decreasing=TRUE), ]
```

	Sharpe	Skewness	Kurtosis
USMV	0.056	-0.962	23.16
IEF	0.048	-0.013	2.81
QUAL	0.045	-0.642	14.56
MTUM	0.040	-0.713	12.13
VLUE	0.033	-1.076	18.83
XLP	0.030	-0.120	8.82
XLY	0.028	-0.385	6.92
GLD	0.027	-0.332	6.20
XLV	0.026	0.070	10.10
VTI	0.024	-0.401	11.01
IWB	0.024	-0.411	10.21
VYM	0.024	-0.700	14.65
VTV	0.024	-0.680	13.75
XLU	0.024	0.006	12.45
IWD	0.024	-0.502	12.74
IVW	0.024	-0.329	8.58
TLT	0.022	-0.034	4.13
XLI	0.022	-0.386	7.59
IWF	0.022	-0.691	31.73
XLB	0.020	-0.387	5.39
XLK	0.018	0.064	6.77
EEM	0.017	0.019	15.26
XLE	0.016	-0.542	12.82
VNQ	0.016	-0.561	17.98
IVE	0.016	-0.491	10.14
XLF	0.011	-0.122	13.87
VEU	0.007	-0.524	11.54
SVXY	0.006	-17.520	603.90
DBC	0.000	-0.513	3.44
USO	-0.021	-1.182	14.69
VXX	-0.070	1.126	5.14

Investor Risk and Return Preferences

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

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

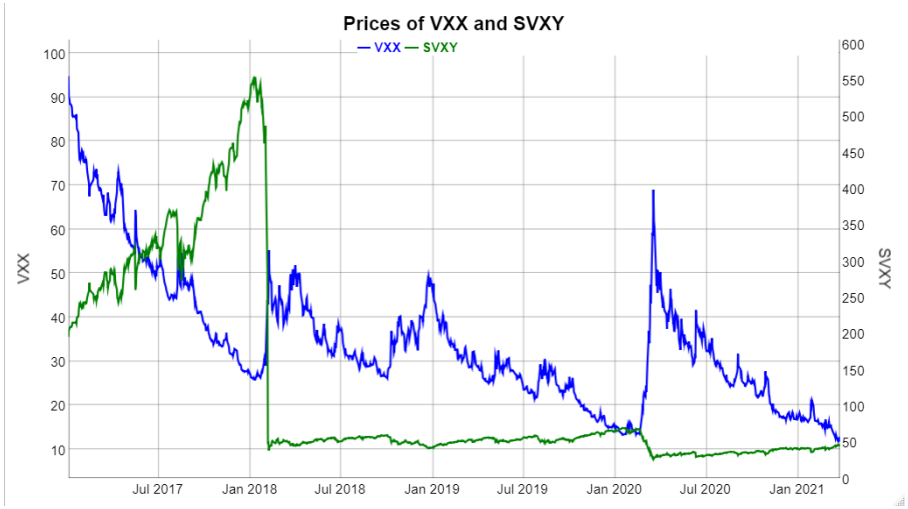
The *VXX* ETF is long the *VIX* index (effectively long an option), so it has positive skewness and small kurtosis, but negative returns (it's short market risk).

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	-0.070	1.13	5.14
SVXY	0.006	-17.52	603.90




```
> # dygraph plot of VXX versus SVXY
> prices <- na.omit(rutils::etfenv$prices[, c("VXX", "SVXY")])
> prices <- prices["2017/"]
> colnamev <- c("VXX", "SVXY")
> colnames(prices) <- colnamev
> dygraphs::dygraph(prices, 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=500)
```

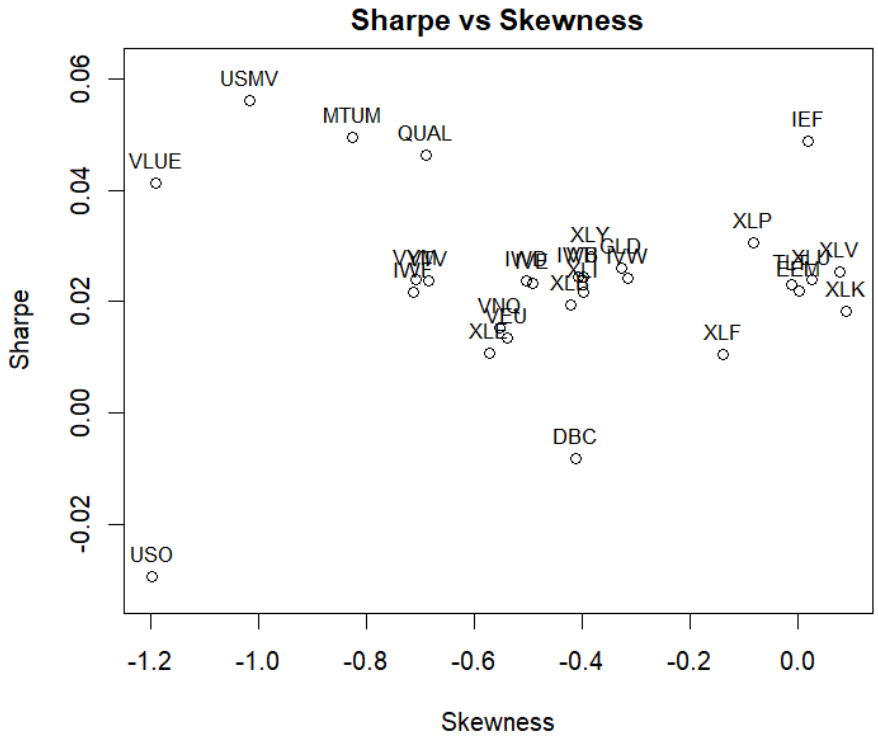
Skewness and Return Tradeoff

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



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

```
> ### Below is for ETFs
> # Sort on Sharpe ratio
> risk_ret <- risk_ret[order(risk_ret$Skewness, decreasing=TRUE), ]
> # Select high skew and low skew ETFs
> cutoff <- (NROW(risk_ret) %/% 2)
> high_skew <- risk_ret$Name[1:cutoff]
> low_skew <- risk_ret$Name[(cutoff+1):NROW(risk_ret)]
> # Calculate returns and log prices
> retsp <- rutils::etfenv$returns
> retsp <- zoo::na.locf(retsp, na.rm=FALSE)
> retsp[is.na(retsp)] <- 0
> sum(is.na(retsp))
> high_skew <- rowMeans(retsp[, high_skew])
> low_skew <- rowMeans(retsp[, low_skew])
> wealthv <- cbind(high_skew, low_skew)
> wealthv <- xts::xts(wealthv, zoo::index(retsp))
> wealthv <- cumsum(wealthv)
> # dygraph plot of high skew and low skew ETFs
> colnamev <- colnames(wealthv)
> dygraphs::dygraph(wealthv, main="Log Wealth of High and Low 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=500)
>
> ### Below is for S&P500 constituent stocks
> # calcmom() calculates the moments of returns
> calcmom <- function(retsp, moment=3) {
+   retsp <- na.omit(retsp)
+   sum(((retsp - mean(retsp))/sd(retsp))^moment)/NROW(retsp)
+ }  # end calcmom
> # Calculate skew and kurtosis of VTI returns
> calcmom(retsp, moment=3)
> calcmom(retsp, 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 return 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:

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

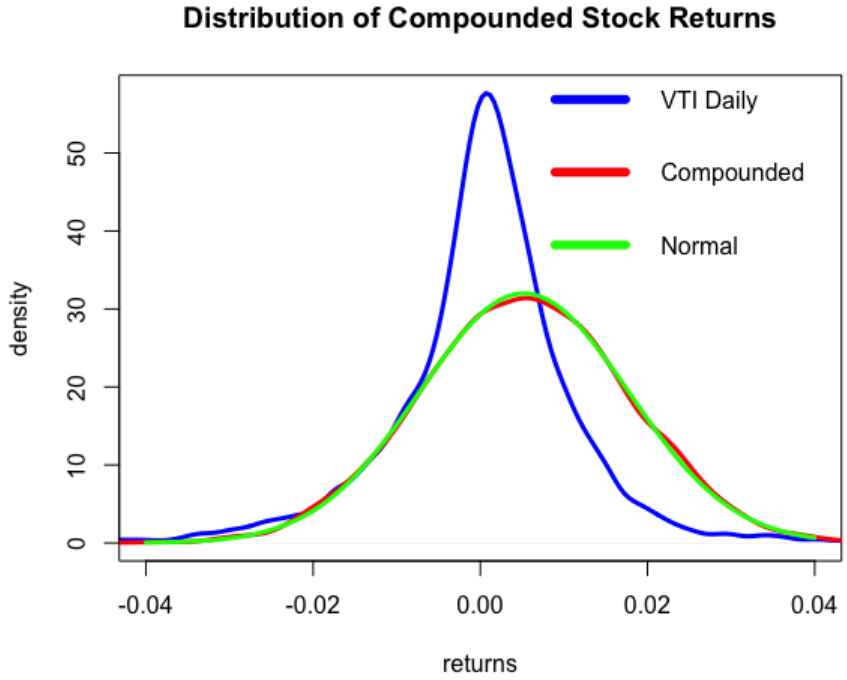
```
> library(PerformanceAnalytics)
> retsp <- rutils::etfenv$returns[, c("VTI", "IEF")]
> retsp <- na.omit(retsp)
> # Calculate the Sharpe ratio
> confl <- 0.05
> PerformanceAnalytics::SharpeRatio(retsp, p=(1-confl),
+   method="historical")
> # Calculate the Sortino ratio
> PerformanceAnalytics::SortinoRatio(retsp)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

Risk and Return of Compounded Stock Returns

Compounded stock returns become closer to normally distributed, and their skewness, kurtosis, and tail risks decrease significantly compared to daily returns.

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

```
> # Calculate VTI percentage returns
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> retvti <- drop(zoo::coredata(retvti))
> nrows <- NROW(retvti)
> # Calculate compounded VTI returns
> holdp <- 252
> retc <- sqrt(holdp)*sapply(1:nrows, function(x) {
+     mean(retvti[sample.int(nrows, size=holdp, replace=TRUE)])
+ })  # end sapply
> # Calculate mean, standard deviation, skewness, and kurtosis
> datav <- cbind(retvti, retc)
> colnames(datav) <- c("VTI", "Agg")
> apply(datav, MARGIN=2, function(x) {
+   # Standardize the returns
+   meanval <- mean(x); stddev <- sd(x); x <- (x - meanval)/stddev
+   c(mean=meanval, stddev=stddev, skew=mean(x^3), kurt=mean(x^4))
+ })  # end sapply
> # Calculate the Sharpe and Dowd ratios
> confl <- 0.05
> sapply(colnames(datav), function(name) {
+   x <- datav[, name]; stddev <- sd(x)
+   varisk <- unname(quantile(x, probs=confl))
+   cvar <- mean(x[x < varisk])
+   ratio <- 1
+   if (name == colnames(datav)[2]) {ratio <- holdp}
+   sqrt(252/ratio)*mean(x)/c(Sharpe=stddev, Dowd=-varisk, DowdC=-cvar)
+ })  # end sapply
```




```
> # Plot the densities of returns
> x11(width=6, height=5)
> par(mar=c(4, 4, 3, 1), oma=c(0, 0, 0, 0))
> plot(density(retvti), t="l", lwd=3, col="blue",
+      xlab="returns", ylab="density", xlim=c(-0.04, 0.04),
+      main="Distribution of Compounded Stock Returns")
> lines(density(retc), t="l", col="red", lwd=3)
> curve(expr=dnorm(x, mean=mean(retc), sd=sd(retc)), col="green", lw
> legend("topright", legend=c("VTI Daily", "Compounded", "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 `roll::roll_sum()` calculates the *weighted* rolling sum (convolution) even faster than `stats:::C_cfilter()`.

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Inspect the R code of the function filter()
> filter
> # Calculate EWMA weights
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate convolution using filter()
> filtered <- filter(closep, filter=weightv,
+                    method="convolution", sides=1)
> # filter() returns time series of class "ts"
> class(filtered)
> # Get information about C_cfilter()
> getAnywhere(C_cfilter)
> # Filter using C_cfilter() over past values (sides=1).
> filterfast <- .Call(stats:::C_cfilter, closep, filter=weightv,
+                     sides=1, circular=FALSE)
> all.equal(as.numeric(filtered), filterfast, check.attributes=FALSE
> # Calculate EWMA prices using roll::roll_sum()
> weightrev <- rev(weightv)
> filtercpp <- roll::roll_sum(closep, width=look_back, weights=weigh
> all.equal(filterfast[-(1:look_back)], as.numeric(filtercpp)[-(1:lo
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   filterfast=.Call(stats:::C_cfilter, closep, filter=weightv, side
+   roll=roll::roll_sum(closep, width=look_back, weights=weightrev)
+   ), 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 ARIMA 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"),
+   filterfast=.Call(stats:::C_rfilter, innov, coeff, double(ncoeff
+   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*.

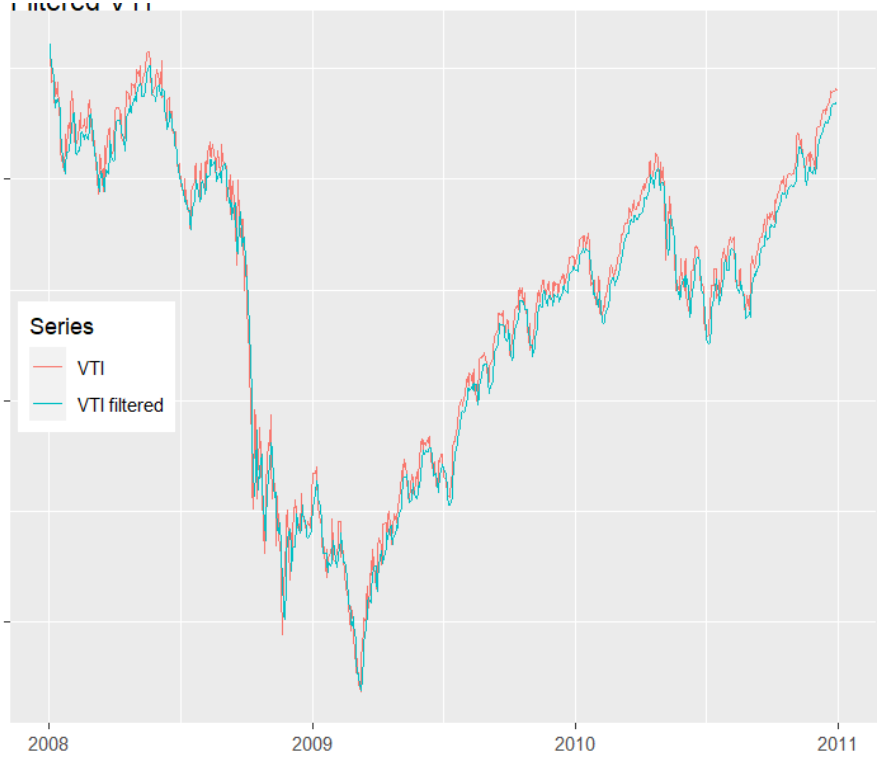
In engineering, smoothing is called a *low-pass filter*, since it eliminates high frequency signals, and it passes through low frequency signals.


```
> # Calculate trailing EWMA prices using roll::roll_sum()
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> weightrev <- rev(weightv)
> filtered <- roll::roll_sum(closep, width=NROW(weightv), weights=
> # Copy warmup period
> filtered[1:look_back] <- closep[1:look_back]
> # Combine prices with smoothed prices
> prices <- cbind(closep, filtered)
> colnames(prices)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diffit(prices), sd)
> # Plot dygraph
> dygraphs::dygraph(prices["2009"], main="VTI Prices and Trailing S
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

```
> # Calculate centered EWMA prices using roll::roll_sum()
> weightv <- c(weightrev, weightv[-1])
> weightv <- weightv/sum(weightv)
> filtered <- roll::roll_sum(closep, width=NROW(weightv), weights=we
> # Copy warmup period
> filtered[1:(2*look_back)] <- closep[1:(2*look_back)]
> # Center the data
> filtered <- rutils::lagit(filtered, -(look_back-1), pad_zeros=FALS
> # Combine prices with smoothed prices
> closef <- cbind(closep, filtered)
> colnames(closef)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diffit(closef), sd)
> # Plot dygraph
> dygraphs::dygraph(closef["2009"], main="VTI Prices and Centered Sm
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

depr: Plotting Filtered Time Series

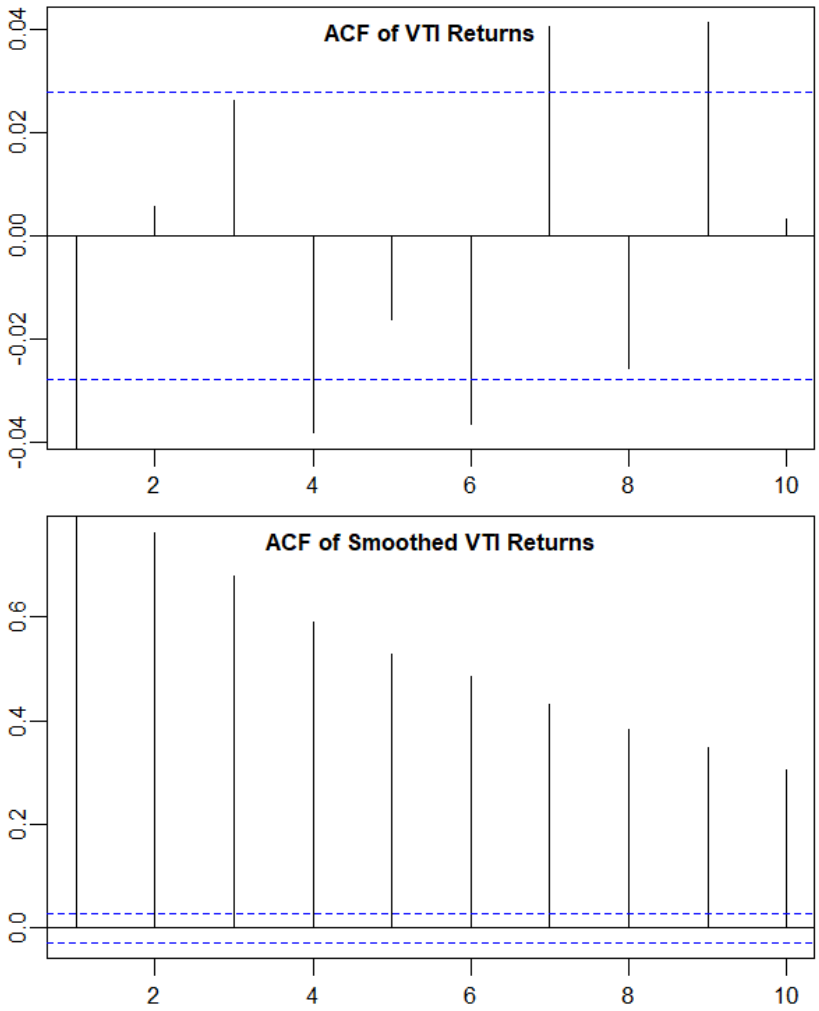
```
> library(rutils)  # Load package rutils
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> # Coerce to zoo and merge the time series
> filtered <- cbind(closep, filtered)
> colnames(filtered) <- c("VTI", "VTI filtered")
> # Plot ggplot2
> autoplot(filtered["2008/2010"],
+     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
> retsp <- 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(retsp[, 1], lag=10, xlab="")
> title(main="ACF of VTI Returns", line=-1)
> # Plot ACF of smoothed VTI returns
> rutils::plot_acf(retsp[, 2], lag=10, xlab="")
> title(main="ACF of Smoothed VTI Returns", line=-1)
```



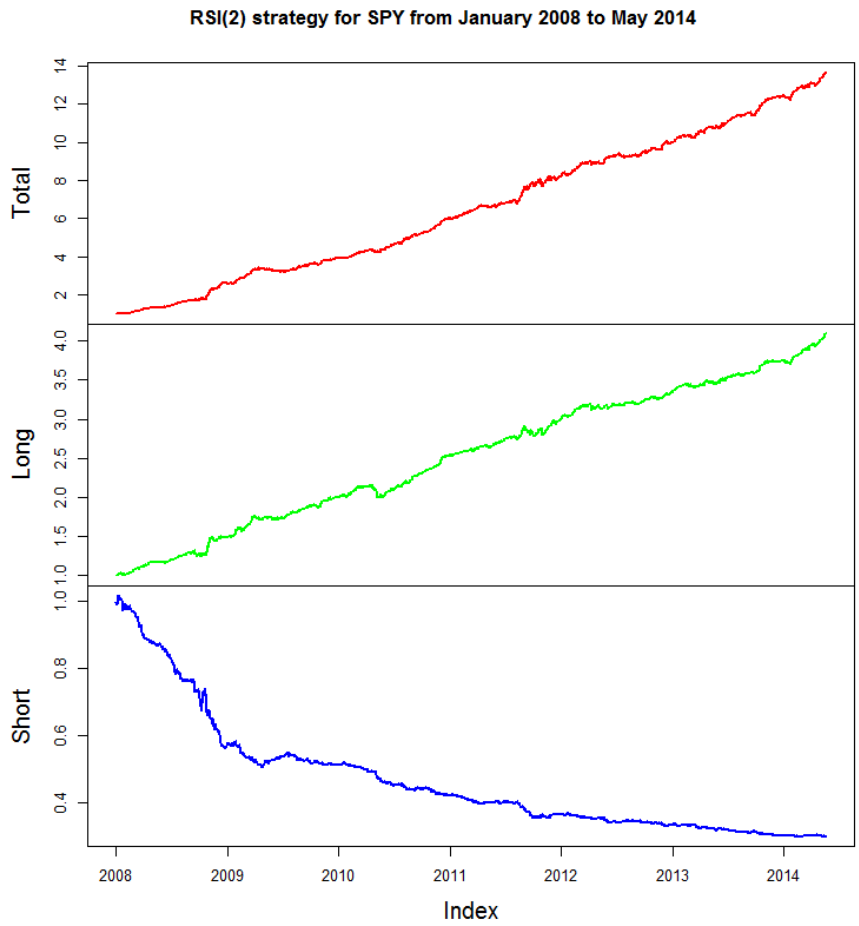
draft: *RSI* Price Technical Indicator

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

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

Where the decay parameter λ determines the rate of decay of the *RSI* weightv, 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")))
```

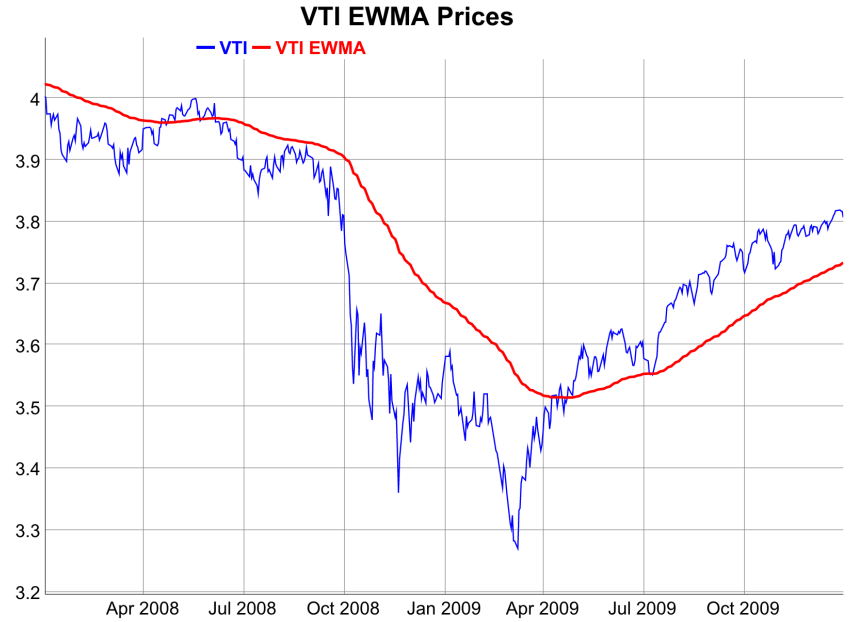
EWMA Price Technical Indicator

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

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

Where the decay parameter λ determines the rate of decay of the *EWMA* weightv, 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 <- log(rutils::etfenv$VTI)
> closep <- quantmod::Cl(ohlc)
> colnames(closep) <- "VTI"
> nrows <- NROW(closep)
> # Calculate EWMA weights
> look_back <- 333
> lambda <- 0.9
> weightv <- lambda^(1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate EWMA prices as the convolution
> ewmacpp <- HighFreq::roll_wsum(closep, weights=weightv)
> prices <- cbind(closep, ewmacpp)
> colnames(prices) <- c("VTI", "VTI EWMA")
```

```
> # Dygraphs plot with custom line colors
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="VTI EWMA 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=500)
> # Standard plot of  EWMA 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(prices["2009"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(prices),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Recursive EWMA Price Indicator

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

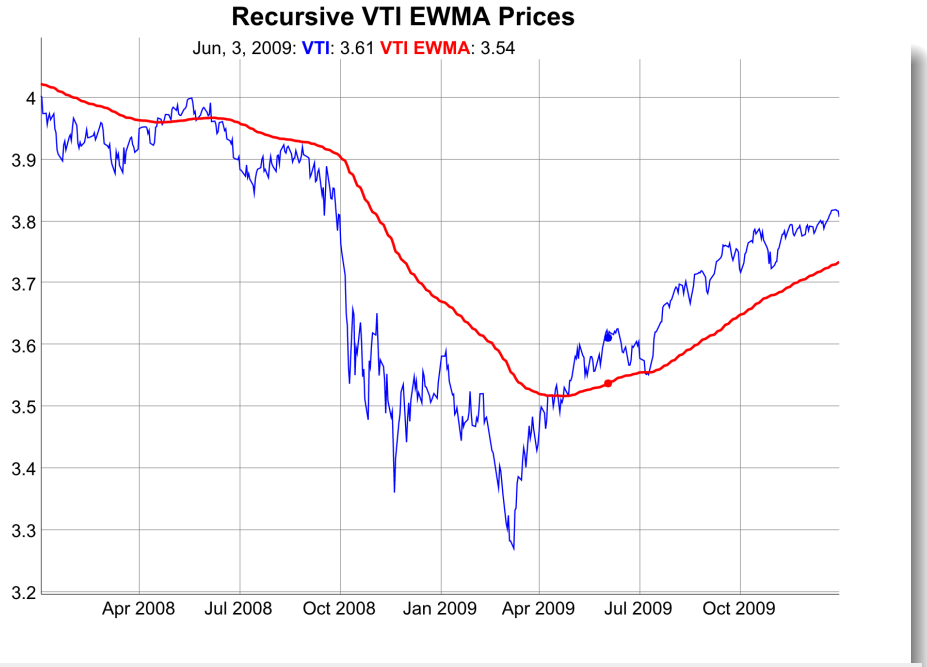
$$p_t^{EWMA} = \lambda p_{t-1}^{EWMA} + (1 - \lambda) p_t$$

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

The recursive *EWMA* 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 exponentially weighted moving average prices recursively.

The function `HighFreq::run_mean()` calculates the exponentially weighted moving average prices recursively.




```
> # Calculate EWMA prices recursively using C++ code
> ewmar <- .Call(stats:::C_rfilter, closep, lambda, c(as.numeric(c
> # Or R code
> # ewmar <- filter(closep, filter=lambda, init=as.numeric(closep[
> ewmar <- (1-lambda)*ewmar
> # Calculate EWMA prices recursively using RcppArmadillo
> ewmacpp <- HighFreq::run_mean(closep, lambda=lambda, weights=0)
> all.equal(drop(ewmacpp), ewmar)
> # Compare the speed of C++ code with RcppArmadillo
> library(microbenchmark)
> summary(microbenchmark(
+   filtercpp=HighFreq::run_mean(closep, lambda=lambda, weights=0)
+   rfilter=.Call(stats:::C_rfilter, closep, lambda, c(as.numeric(
```

```
> # Dygraphs plot with custom line colors
> prices <- cbind(closep, ewmacpp)
> colnames(prices) <- c("VTI", "VTI EWMA")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="Recursive VTI EWMA 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=500)
> # Standard plot of  EWMA 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(prices["2009"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(prices),
+  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

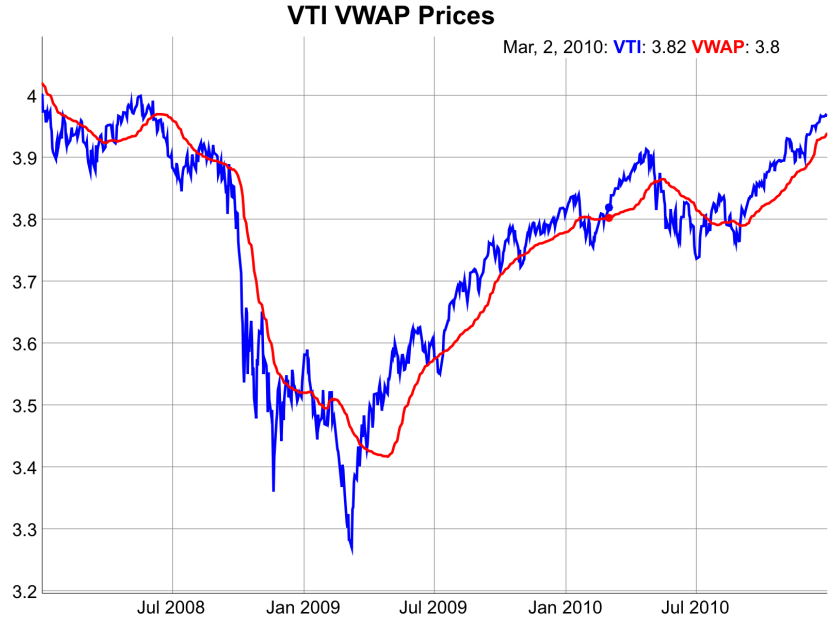
The Volume-Weighted Average Price (*VWAP*) is defined as the sum of prices multiplied by trading volumes, divided by the sum of volumes:

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

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
> ohlc <- rutils::etfenv$VTI
> closep <- log(quantmod::Cl(ohlc))
> colnames(closep) <- "VTI"
> volumes <- quantmod::Vo(ohlc)
> colnames(volumes) <- "Volume"
> nrows <- NROW(closep)
> # Calculate the VWAP prices
> look_back <- 21
> vwap <- roll::roll_sum(closep*volumes, width=look_back, min_obs=
> volume_roll <- roll::roll_sum(volumes, width=look_back, min_obs=
> vwap <- vwap/volume_roll
> colnames(vwap) <- "VWAP"
> prices <- cbind(closep, vwap)
```

```
> # Dygraphs plot with custom line colors
> colorv <- c("blue", "red")
> dygraphs::dygraph(prices["2009"], main="VTI VWAP Prices") %>%
+   dyOptions(colors=colorv, strokeWidth=2)
> # 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(prices["2009"], theme=plot_theme,
+        lwd=2, name="VTI VWAP Prices")
> legend("bottomright", legend=colnames(prices),
+  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 as the ratio of the volume weighted prices μ^{pv} divided by the mean trading volumes μ^{v}:

$$p^{VWAP} = \frac{\mu^{pv}}{\mu^{v}}$$

The volume weighted prices μ^{pv} and the mean trading volumes μ^{v} are both calculated recursively:

$$\mu^{v}_{t} = \lambda \mu^{v}_{t-1} + (1 - \lambda) v_t$$

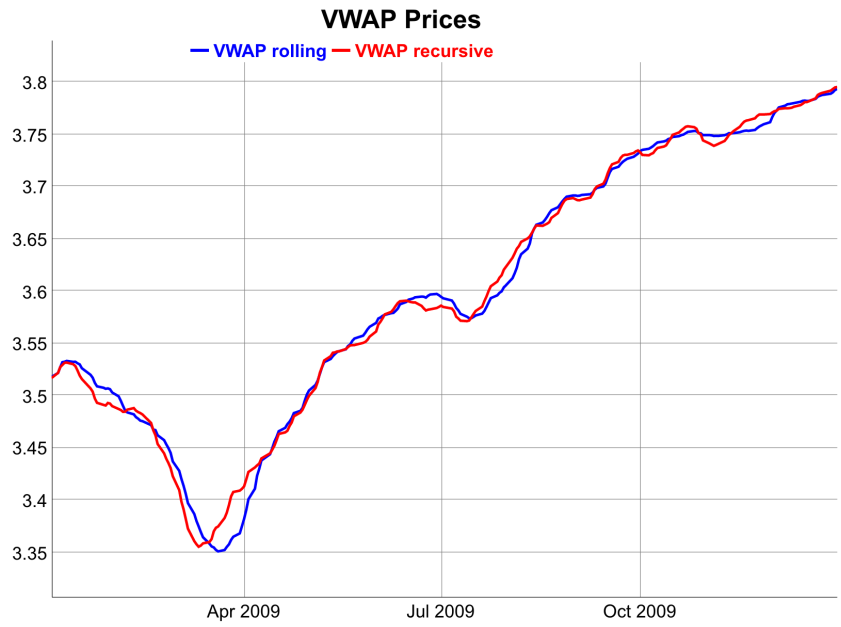
$$\mu^{pv}_{t} = \lambda \mu^{pv}_{t-1} + (1 - \lambda) v_t p_t$$

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 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
> lambda <- 0.9
> volumer <- .Call(stats:::C_rfilter, volumes, lambda, c(as.numeric(
> pricer <- .Call(stats:::C_rfilter, volumes*closep, lambda, c(as.nu
> vwapr <- pricer/volumer
> # Calculate VWAP prices recursively using RcppArmadillo
> vwapcpp <- HighFreq::run_mean(closep, lambda=lambda, weights=volum
> all.equal(vwapr, drop(vwapcpp))
> # Dygraphs plot the VWAP prices
> prices <- xts(cbind(vwap, vwapr), zoo::index(ohlc))
> colnames(prices) <- c("VWAP rolling", "VWAP recursive")
> dygraphs::dygraph(prices["2009"], main="VWAP Prices") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Smooth Asset Returns

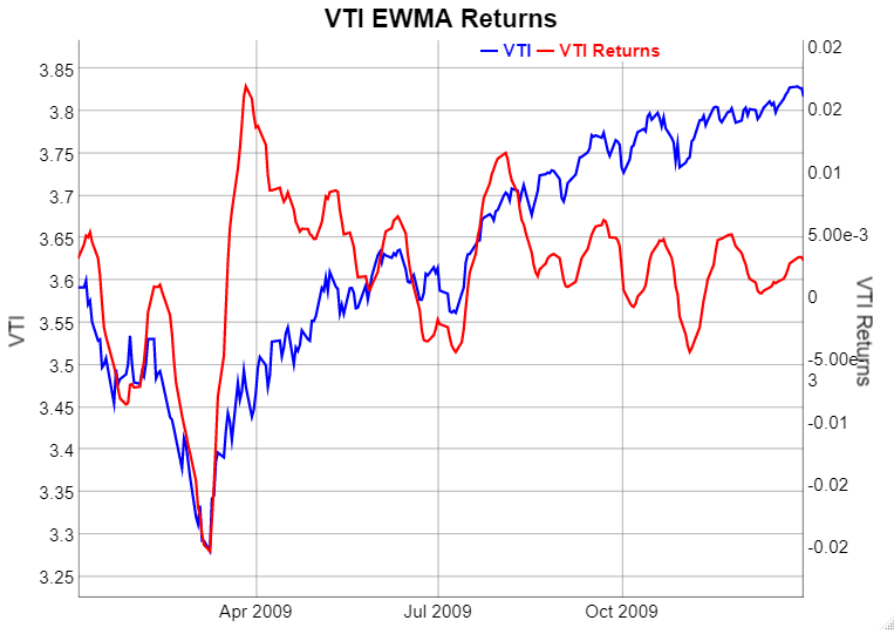
Asset returns are calculated by filtering prices through a *differencing* filter.

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 high and low frequency signals, and it passes through medium frequency signals.




```
> # Calculate two EWMA prices
> look_back <- 21
> lambda <- 0.1
> weightv <- exp(lambda*1:look_back)
> weightv <- weightv/sum(weightv)
> ewma_fast <- roll::roll_sum(closep, width=look_back, weights=wei
> lambda <- 0.05
> weightv <- exp(lambda*1:look_back)
> weightv <- weightv/sum(weightv)
> ewma_slow <- roll::roll_sum(closep, width=look_back, weights=wei
```

```
> # Calculate VTI prices
> ewmadiff <- (ewma_fast - ewma_slow)
> prices <- cbind(closep, ewmadiff)
> symbol <- "VTI"
> colnames(prices) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main=paste(symbol, "EWMA Returns
+   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
```

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: $r_t = (1 - L)p_t$.

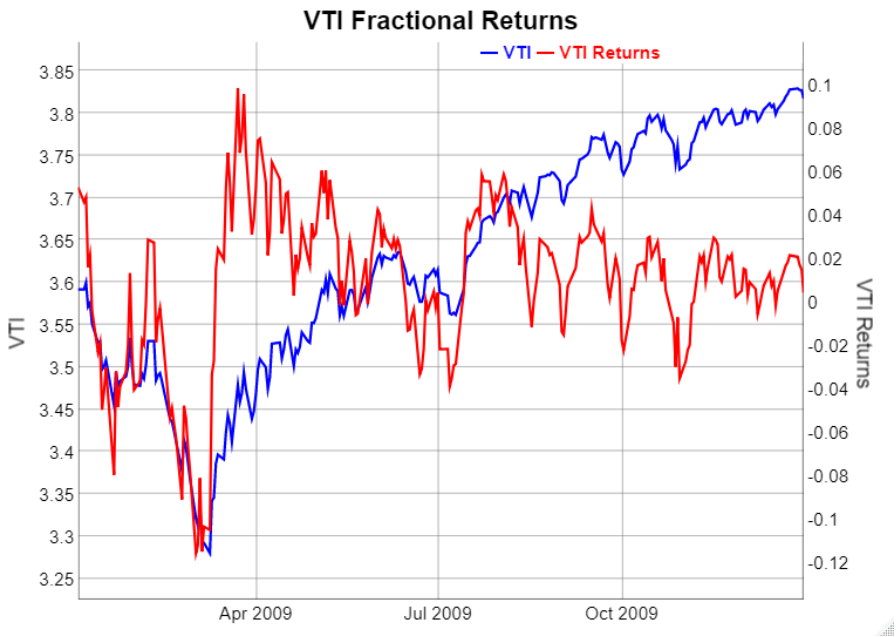
The simple returns can be generalized to the fractional returns by raising the returns operator to some power $\delta < 1$:

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

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

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

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 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))
> weightv <- rev(weightv)
> # Calculate fractional VTI returns
> retvti <- roll::roll_sum(closep, width=look_back, weights=weightv,
> prices <- cbind(closep, retvti)
> symbol <- "VTI"
> colnames(prices) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main=paste(symbol, "Fractional R
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
```

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).

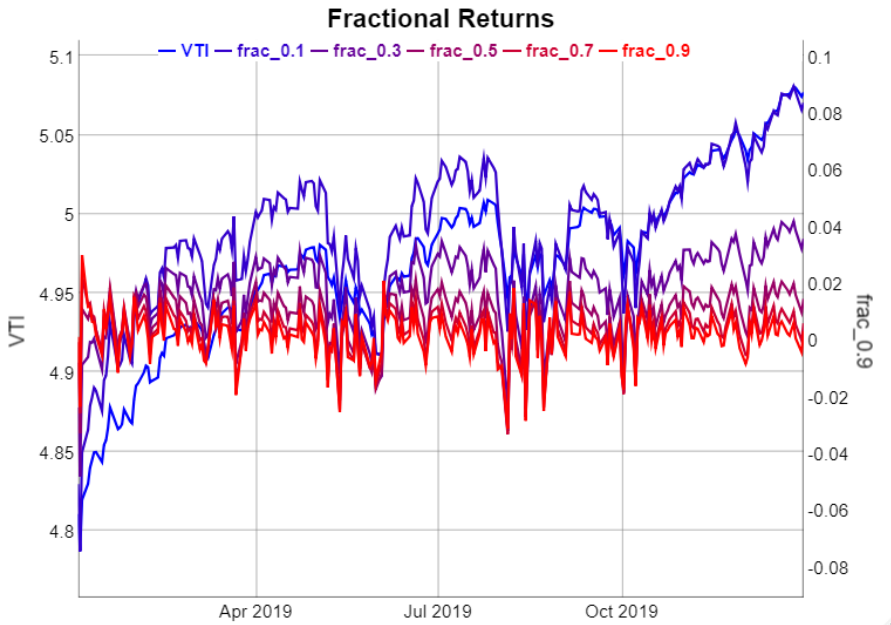
The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

```
> # Calculate VTI log returns
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> retvti <- rutils::diffit(closep)
> # Perform ADF test for prices
> tseries::adf.test(closep)
> # Perform ADF test for returns
> tseries::adf.test(retvti)
```

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)
> retvti <- 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 <- rev(weightv - mean(weightv))
+   roll::roll_sum(closep, width=look_back, weights=weightv, min_obs
+ })  # end lapply
> retvti <- do.call(cbind, retvti)
> retvti <- cbind(closep, retvti)
> colnames(retvti) <- c("VTI", paste0("frac_", deltav))
> # Calculate ADF test statistics
> adfstats <- sapply(retvti, function(x)
+   suppressWarnings(tseries::adf.test(x)$statistic)
+ )  # end sapply
> names(adfstats) <- colnames(retvti)
```




```
> # Plot dygraph of fractional VTI returns
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(retvti))
> colnamev <- colnames(retvti)
> dyplot <- dygraphs::dygraph(retvti["2019"], main="Fractional Retu
+   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], stroke
> dyplot <- dyplot %>% dyLegend(width=500)
> dyplot
```

Trading Volume Z-Scores

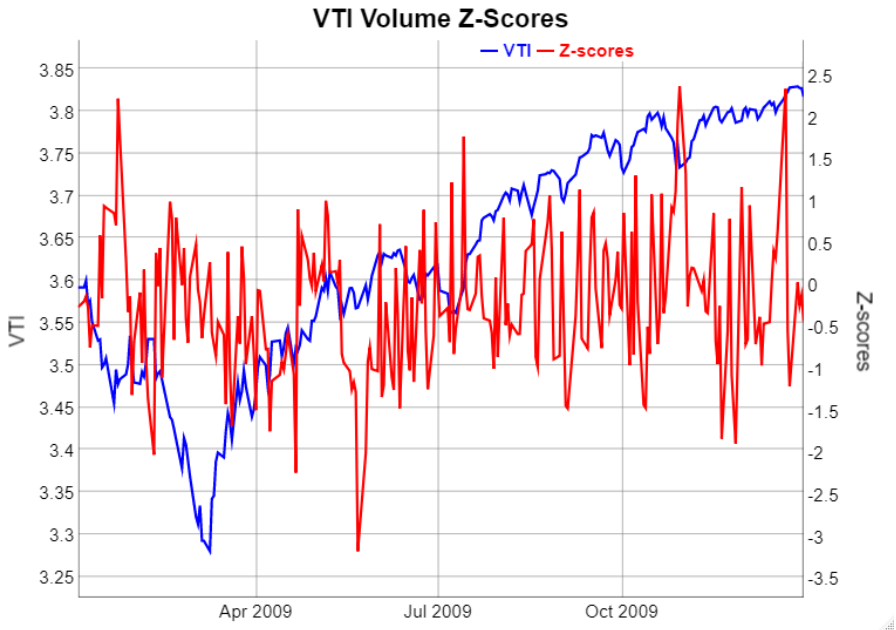
The trailing *volume z-score* is equal to the volume v_t minus the trailing average volumes $\bar{v}_t$ divided by the volatility of the volumes σ_t:

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

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

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

```
> # Calculate volume z-scores
> volumes <- quantmod::Vo(rutils::etfenv$VTI)
> look_back <- 21
> volumem <- roll::roll_mean(volumes, width=look_back, min_obs=1)
> volumesd <- roll::roll_sd(rutils::diffit(volumes), width=look_ba
> volumesd[1] <- 0
> volumez <- ifelse(volumesd > 0, (volumes - volumem)/volumesd, 0)
> # Plot histogram of volume z-scores
> x11(width=6, height=5)
> hist(volumez, breaks=1e2)
```




```
> # Plot dygraph of volume z-scores of VTI prices
> prices <- cbind(closep, volumez)
> colnames(prices) <- c("VTI", "Z-scores")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["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
```

Volatility Z-Scores

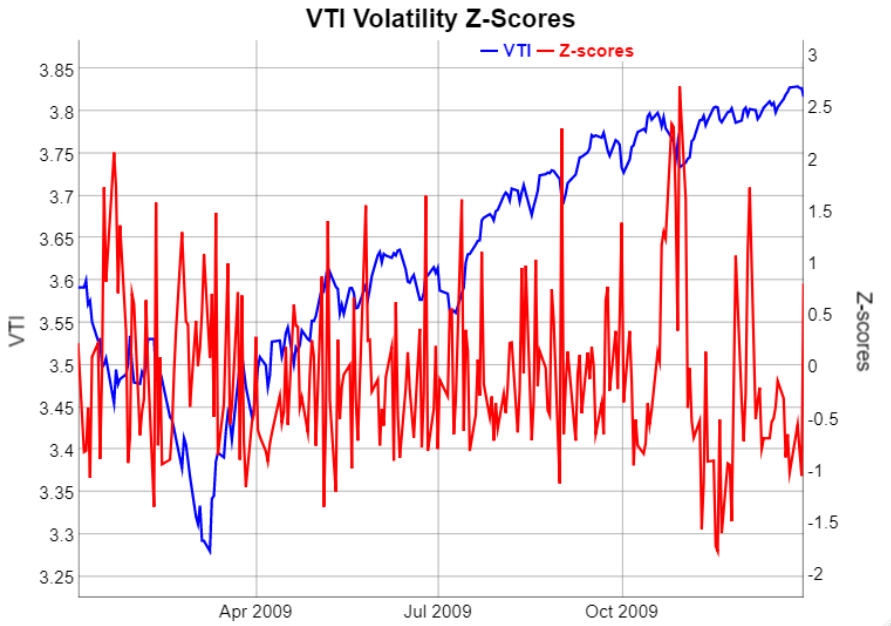
The difference between high and low prices is a proxy for the spot volatility in a bar of data.

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:

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

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

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




```
> # Extract VTI log OHLC prices
> ohlc <- log(rutils::etfenv$VTI)
> # Calculate volatility z-scores
> volat <- quantmod::Hi(ohlc)-quantmod::Lo(ohlc)
> look_back <- 21
> volatm <- roll::roll_mean(volat, width=look_back, min_obs=1)
> volatsd <- roll::roll_sd(rutils::diffit(volat), width=look_back,
> volatsd[1] <- 0
> volatz <- ifelse(volatsd > 0, (volat - volatm)/volatsd, 0)
> # Plot histogram of volatility z-scores
> x11(width=6, height=5)
> hist(volatz, breaks=1e2)
> # Plot scatterplot of volume and volatility z-scores
> plot(as.numeric(volatz), as.numeric(volumez),
+      xlab="volatility z-score", ylab="volume z-score")
> regmod <- lm(volatz ~ volumez)
> abline(regmod, col="red", lwd=3)
```

```
> # Plot dygraph of VTI volatility z-scores
> closep <- quantmod::Cl(ohlc)
> prices <- cbind(closep, volatz)
> colnames(prices) <- c("VTI", "Z-scores")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="VTI Volatility 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
```

Centered Price Z-scores

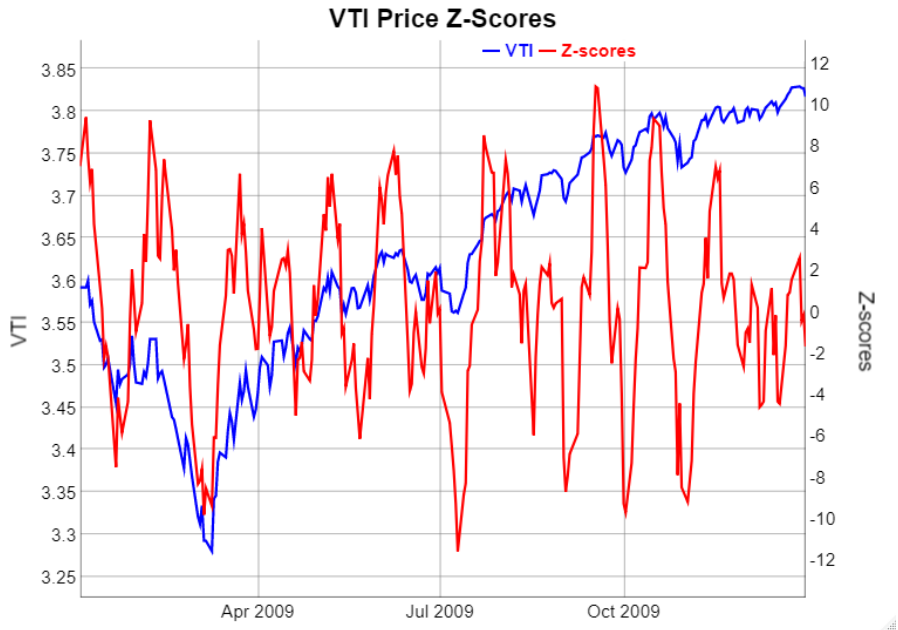
An extreme local price is a price which differs significantly from neighboring prices.

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:

$$z_i = \frac{2p_t - p_{t-k} - p_{t+k}}{\sigma_t}$$

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

The lag parameter k determines the scale of the extreme local prices, with smaller k producing larger z-scores for more local price extremes.




```
> # Calculate the centered volatility
> look_back <- 21
> half_back <- look_back %/% 2
> retvti <- rutils::diffit(closep)
> volat <- roll::roll_sd(retvti, width=look_back, min_obs=1)
> volat <- rutils::lagit(volat, lagg=(-half_back))
> # Calculate the z-scores of prices
> pricez <- (2*closep -
+   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
> prices <- cbind(closep, pricez)
> colnames(prices) <- c("VTI", "Z-scores")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="VTI Price 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
```

Labeling the Tops and Bottoms of Prices

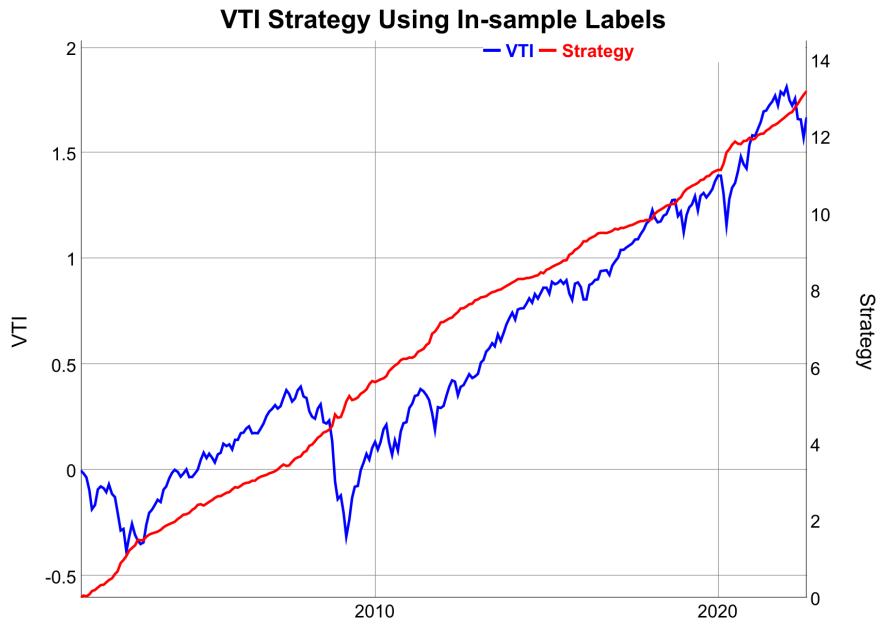
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

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 thresholds for labeling tops and bottoms
> threshv <- quantile(pricez, c(0.1, 0.9))
> # Calculate the vectors of tops and bottoms
> tops <- (pricez > threshv[2])
> colnames(tops) <- "tops"
> bottoms <- (pricez < threshv[1])
> colnames(bottoms) <- "bottoms"
> # Backtest in-sample VTI strategy
> posit <- rep(NA_integer_, NROW(retvti))
> posit[1] <- 0
> posit[tops] <- (-1)
> posit[bottoms] <- 1
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- cumsum(retvti*posit)
```




```
> # Plot dygraph of in-sample VTI strategy
> prices <- cbind(closep, pnls)
> colnames(prices) <- c("VTI", "Strategy")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices, main="VTI Strategy Using In-sample Label
+   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
```

Regression Z-Scores

The trailing *z-score* z_i of a price p_t can be defined as the *standardized residual* of the linear regression with respect to time t_i or some other variable:

$$z_i = \frac{p_t - (\alpha + \beta t_i)}{\sigma_t}$$

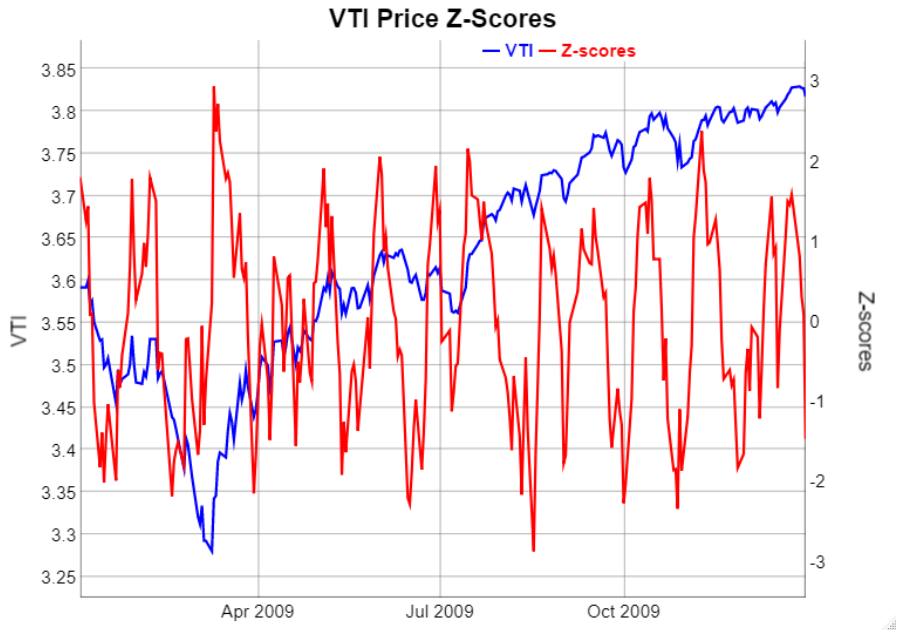
Where α and β are the *regression coefficients*, and σ_t is the standard deviation of the residuals.

The regression *z-scores* can be used as rich or cheap indicators, 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_zscores()` calculates the residuals of a rolling regression.

```
> # Calculate trailing price regression z-scores
> dates <- matrix(zoo::index(closep))
> look_back <- 21
> regz <- drop(HighFreq::roll_zscores(response=closep, predictor=da
> regz[1:look_back] <- 0
```




```
> # Plot dygraph of z-scores of VTI prices
> prices <- cbind(closep, regz)
> colnames(prices) <- c("VTI", "Z-scores")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="VTI Price 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
```

Hampel Filter for Outlier Detection

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

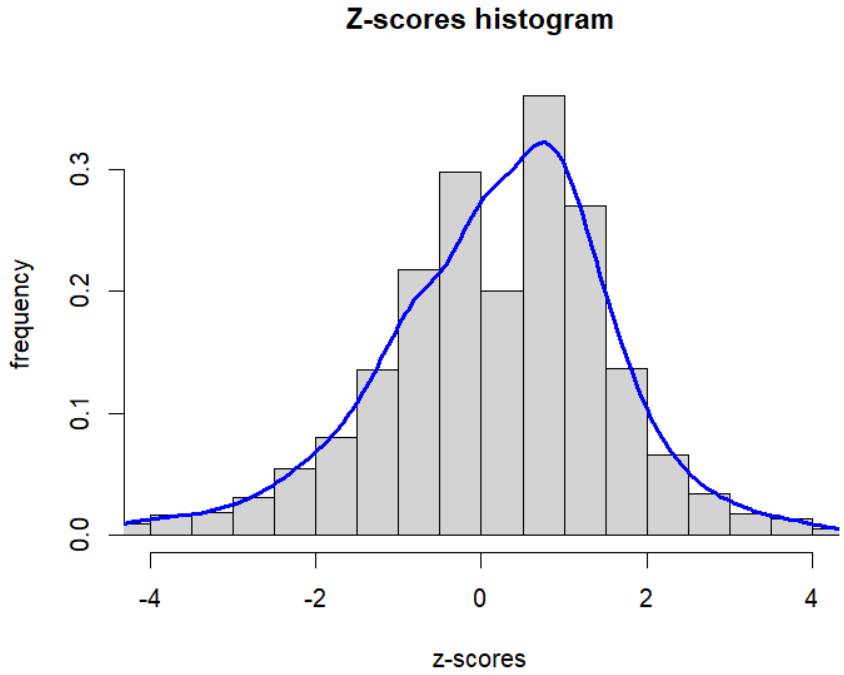
$$\text{MAD} = \text{median}(\text{abs}(p_t - \text{median}(\mathbf{p})))$$

The *Hampel filter* uses the *MAD* dispersion measure to detect outliers in data.

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 rolling 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 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 prices and medians
> dygraphs::dygraph(cbind(closep, medianv), main="VTI median") %>%
+   dyOptions(colors=c("black", "red"))
> # 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 rolling *MAD* using a trailing look-back interval over past data.

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

If the rolling 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)
```

draft: State Space Models

A *state space model* is a stochastic process for a *state variable* θ, which is subject to *measurement error*.

The *state variable* θ is latent (not directly observable), and its value is only measured by observing the *measurement variable* y_t.

A simple *state space model* can be written as a *transition equation* and a *measurement equation*:

$$\theta_t = g_t \theta_{t-1} + w_t$$
$$y_t = f_t \theta_t + v_t$$

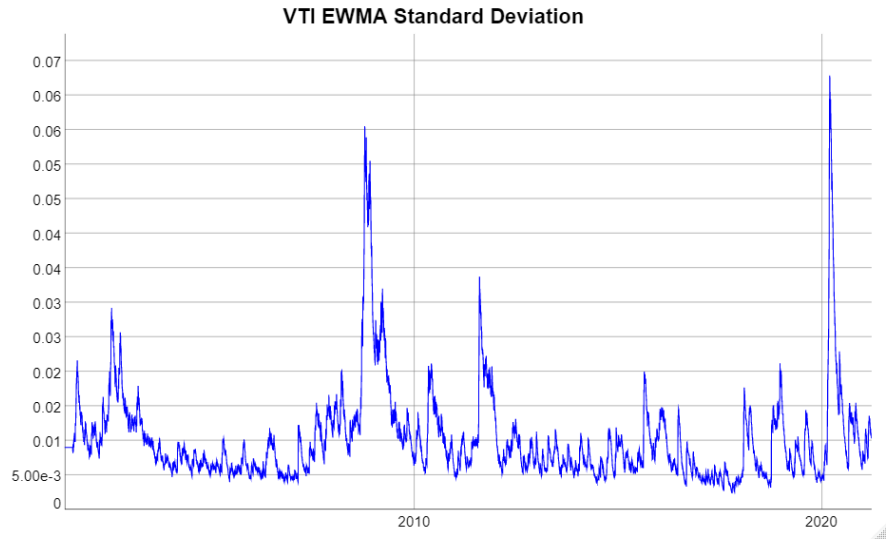
Where w_t and v_t follow the normal distributions $\phi(0, \sigma_t^w)$ and $\phi(0, \sigma_t^v)$.

The system variables (matrices) g_t and f_t are deterministic functions of time.

If the *time series* has zero *expected* mean, then the *EWMA realized* variance estimator can be written approxiamtely as: 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 *EWMA* weightv, 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 (weightv).




```
> # Calculate EWMA 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, retvti^2, filter=weightv, sides=1
> varv[1:(look_back-1)] <- varv[look_back]
> # Plot EWMA volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(retvti))
> dygraphs::dygraph(varv, main="VTI EWMA Volatility")
> quantmod::chart_Series(xtsv, name="VTI EWMA Volatility")
```

Calculating the Rolling Variance of Asset Returns

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

The rolling 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 rolling variance in `R` using vectorized functions, without using an `apply()` loop.

```
> # Calculate VTI percentage returns
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retvti)
> # Define end points
> endd <- 1:NROW(retvti)
> # 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 rolling variance in sapply() loop - takes long
> varv <- sapply(1:nrows, function(it) {
+   retsp <- retvti[startp[it]:endd[it]]
+   sum((retsp - mean(retsp))^2)/look_back
+ })  # end sapply
> # Use only vectorized functions
> retc <- cumsum(retvti)
> retc <- (retc - c(rep_len(0, look_back), retc[1:(nrows-look_back)]
> retc2 <- cumsum(retvti^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(retvti, look_back=look_back)
> retc2 <- rutils::roll_sum(retvti^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(retvti))
> colnames(varv) <- "VTI.variance"
> head(varv)
```

Calculating the Rolling Variance Using Package *roll*

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

- `roll_sum()` for the *weighted* rolling sum,
- `roll_var()` for the *weighted* rolling variance,
- `roll_scale()` for the rolling scaling and centering of time series,
- `roll_pcr()` for the rolling 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 rolling VTI variance using package roll
> library(roll)  # Load roll
> varv <- roll::roll_var(retvti, width=look_back)
> colnames(varv) <- "VTI.variance"
> head(varv)
> sum(is.na(varv))
> varv[1:(look_back-1)] <- 0
> # Benchmark calculation of rolling variance
> library(microbenchmark)
> summary(microbenchmark(
+   sapply=sapply(1:nrows, function(it) {
+     var(retvti[startp[it]:endd[it]])
+   }),
+   roll=roll::roll_var(retvti, width=look_back),
+   times=10))[, c(1, 4, 5)]
```

Rolling *EWMA* Realized Volatility Estimator

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

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

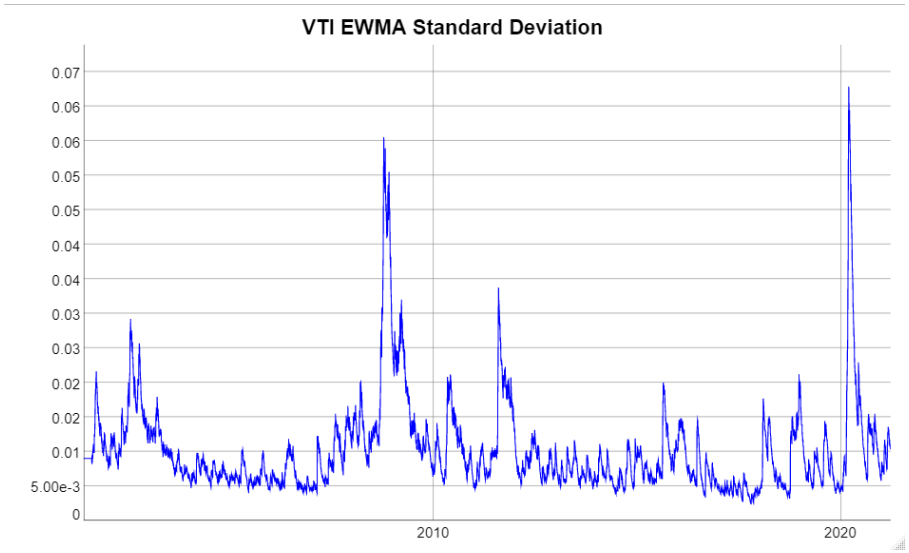
$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2 = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j r_{t-j}^2$$

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 *EWMA* weightv, 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 (weightv).

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




```
> # Calculate EWMA 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, retvti^2, filter=weightv, sides=1
> varv[1:(look_back-1)] <- varv[look_back]
> # Plot EWMA volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(retvti))
> dygraphs::dygraph(varv, main="VTI EWMA Volatility") %>%
+   dyOptions(colors="blue")
> quantmod::chart_Series(xtsv, name="VTI EWMA Volatility")
```

Estimating *EWMA* Variance Using Package *roll*

If the *time series* has non-zero *expected* mean, then the rolling *EWMA* 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 rolling *EWMA* variance.

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

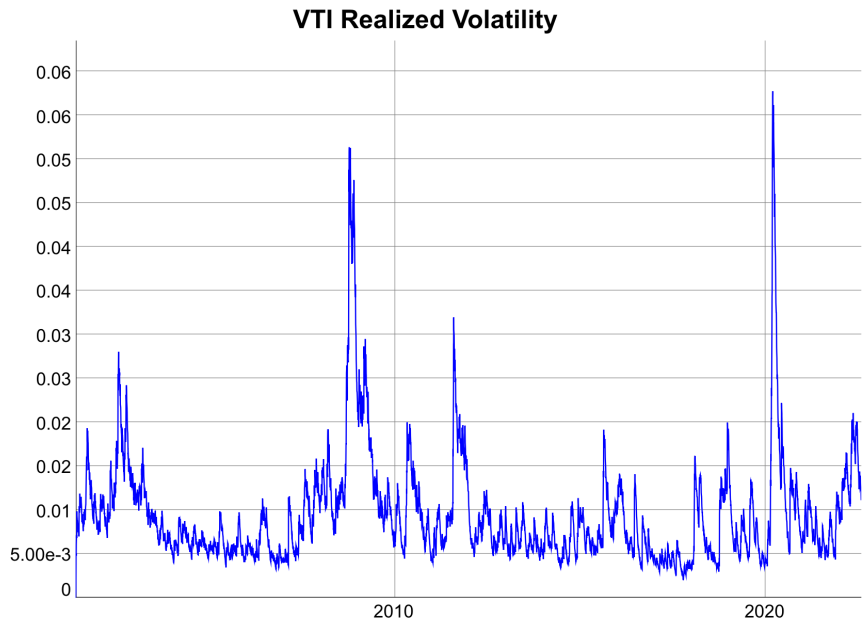
Recursive Realized Volatility Estimator

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

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

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

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




```
> # Calculate realized variance recursively
> lambda <- 0.9
> volat <- HighFreq::run_var(retvti, lambda=lambda)
> volat <- sqrt(volat)
> # Plot EWMA volatility
> volat <- xts:::xts(volat, order.by=datev)
> dygraphs::dygraph(volat, main="VTI Realized Volatility") %>%
+   dyOptions(colors="blue")
```

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 Rolling 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 *rolling* 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["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 rolling 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 Rolling Range Volatility") %>%
+   dyOptions(colors="blue")
> # 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)]
```

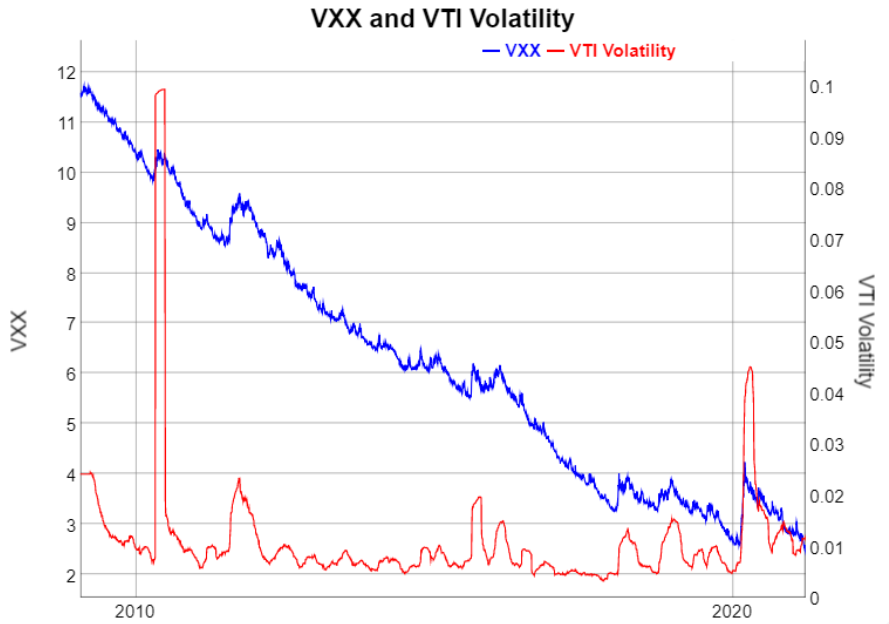
VXX Prices and the Rolling Volatility

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 rolling 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 rolling volatility.

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



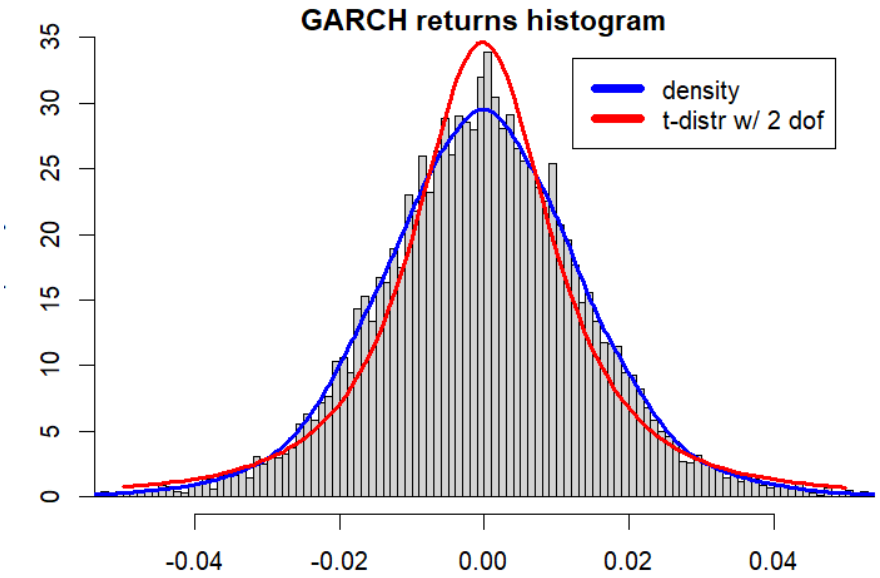

```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volat)
> colnames(datav)[2] <- "VTI Volatility"
> colnamev <- colnames(datav)
> cap_tion <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=cap_tion) %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   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
```

draft: Cointegration of VXX Prices and the Rolling Volatility

The rolling 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.



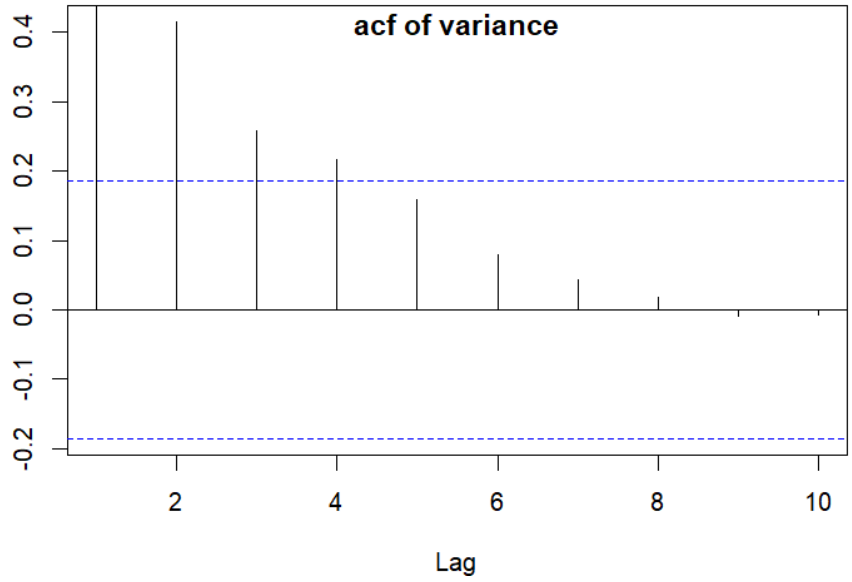

```
> # Calculate VXX log prices
> vxx <- na.omit(rutils::etfenv$prices$VXX)
> dates <- zoo::index(vxx)
> look_back <- 41
> vxx <- log(vxx)
> vxx <- (vxx - roll::roll_mean(vxx, width=look_back))
> vxx[1:look_back] <- vxx[look_back+1]
> # Calculate rolling VTI volatility
> closep <- get("VTI", rutils::etfenv)[dates]
> 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
> betav <- drop(cov(vxx, volat)/var(volat))
> alpha <- drop(mean(vxx) - betav*mean(volat))
> # Calculate regression residuals
> fittedv <- (alpha + betav*volat)
> residuals <- (vxx - fittedv)
> # Perform ADF test on residuals
> tseries::adf.test(residuals, k=1)
```

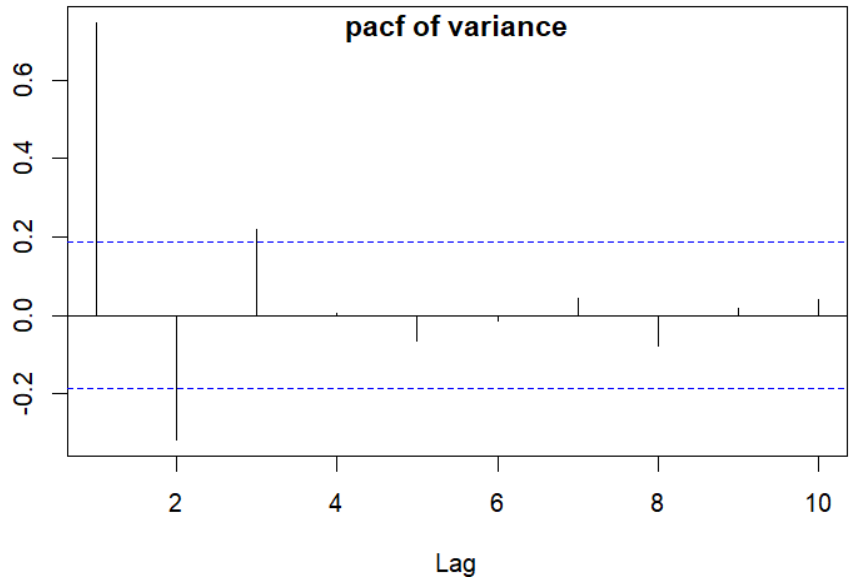
```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volat)
> colnamev <- colnames(datav)
> cap_tion <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=cap_tion) %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE)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

Autocorrelation of Volatility

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

```
> # Calculate VTI percentage returns
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> # Calculate rolling VTI variance using package roll
> look_back <- 22
> varv <- roll::roll_var(retvti, width=look_back)
> varv[1:(look_back-1)] <- 0
> colnames(varv) <- "VTI.variance"
> # Number of look_backs that fit over returns
> nrows <- NROW(retvti)
> nagg <- nrows %/% look_back
> # Define endd with beginning stub
> endd <- c(0, nrows-look_back*nagg + (0:nagg)*look_back)
> nrows <- NROW(endd)
> # Subset variance to endd
> 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 $(r_t - \mu)^2$ 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
> alpha <- 0.3; betav <- 0.5;
> omega <- 1e-4*(1-alpha-betav)
> nrows <- 1000
> # Calculate matrix of standard normal innovations
> set.seed(1121)  # Reset random numbers
> innov <- rnorm(nrows)
> retsp <- numeric(nrows)
> varv <- numeric(nrows)
> varv[1] <- omega/(1-alpha-betav)
> retsp[1] <- sqrt(varv[1])*innov[1]
> # Simulate GARCH model
> for (i in 2:nrows) {
+   retsp[i] <- sqrt(varv[i-1])*innov[i]
+   varv[i] <- omega + alpha*retsp[i]^2 +
+     betav*varv[i-1]
+ }  # end for
> # Simulate the GARCH process using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alpha,
+   beta=betav, innov=matrix(innov))
> all.equal(garch_data, cbind(retsp, 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:

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

Where σ_t^2 is the time-dependent variance, equal to the weighted average of the point *realized* variance $(r_t - \mu)^2$ 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
> alpha <- 0.3; betav <- 0.5;
> omega <- 1e-4*(1-alpha-betav)
> nrows <- 1000
> # Calculate matrix of standard normal innovations
> set.seed(1121)  # Reset random numbers
> innov <- rnorm(nrows)
> retsp <- numeric(nrows)
> varv <- numeric(nrows)
> varv[1] <- omega/(1-alpha-betav)
> retsp[1] <- sqrt(varv[1])*innov[1]
> # Simulate GARCH model
> for (i in 2:nrows) {
+   retsp[i] <- sqrt(varv[i-1])*innov[i]
+   varv[i] <- omega + alpha*retsp[i]^2 + betav*varv[i-1]
+ }  # end for
> # Simulate the GARCH process using Rcpp
> garch_data <- HighFreq::sim_garch(omega=omega, alpha=alpha,
+   beta=betav, innov=matrix(innov))
> all.equal(garch_data, cbind(retsp, 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

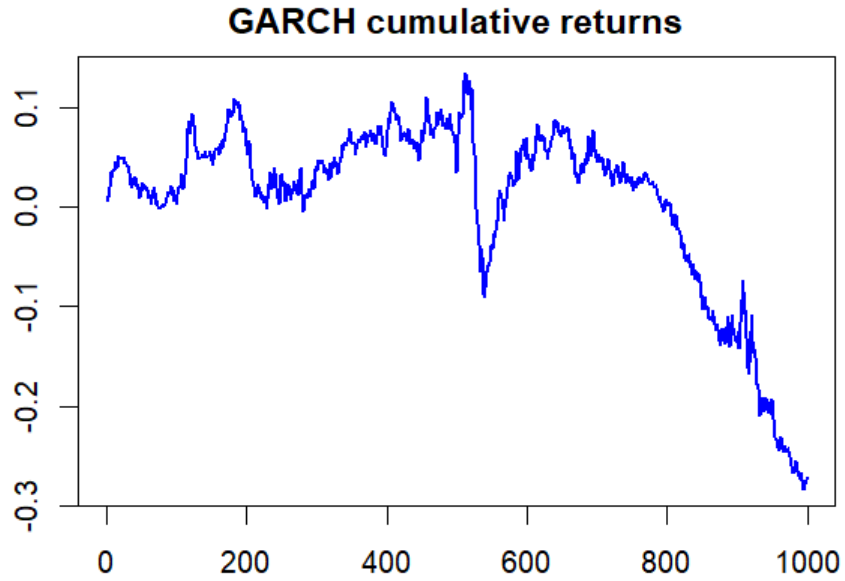
The *GARCH* volatility model produces volatility clustering - periods of high volatility followed by a quick decay.

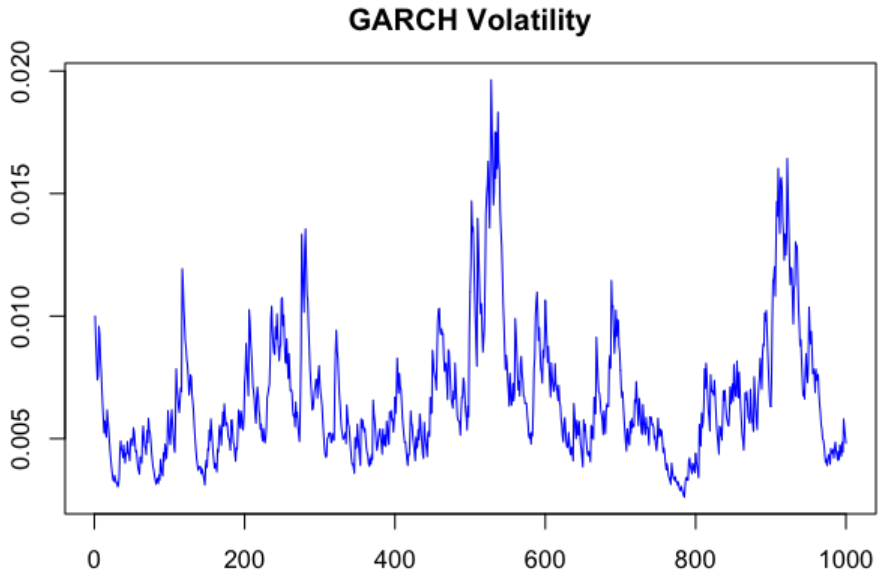
But the decay of the volatility in the *GARCH* model is faster than what is observed in practice.

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(retsp), 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

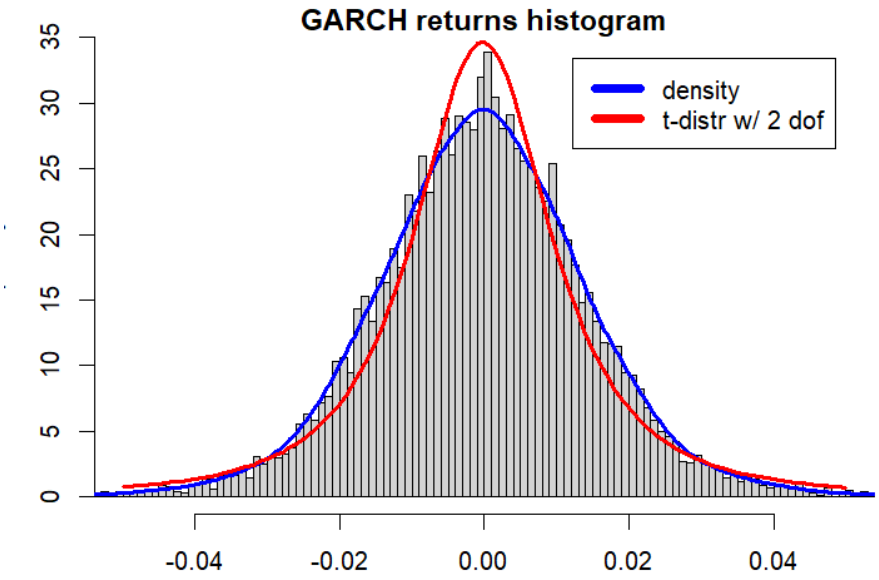
The *GARCH* volatility model produces *leptokurtic* returns with fat tails in their the distribution.

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

```
> # Plot histogram of GARCH returns
> histp <- hist(retsp, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.03, 0.03),
+   ylab="frequency", freq=FALSE, main="GARCH Returns Histogram")
> lines(density(retsp, 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",
+  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
```

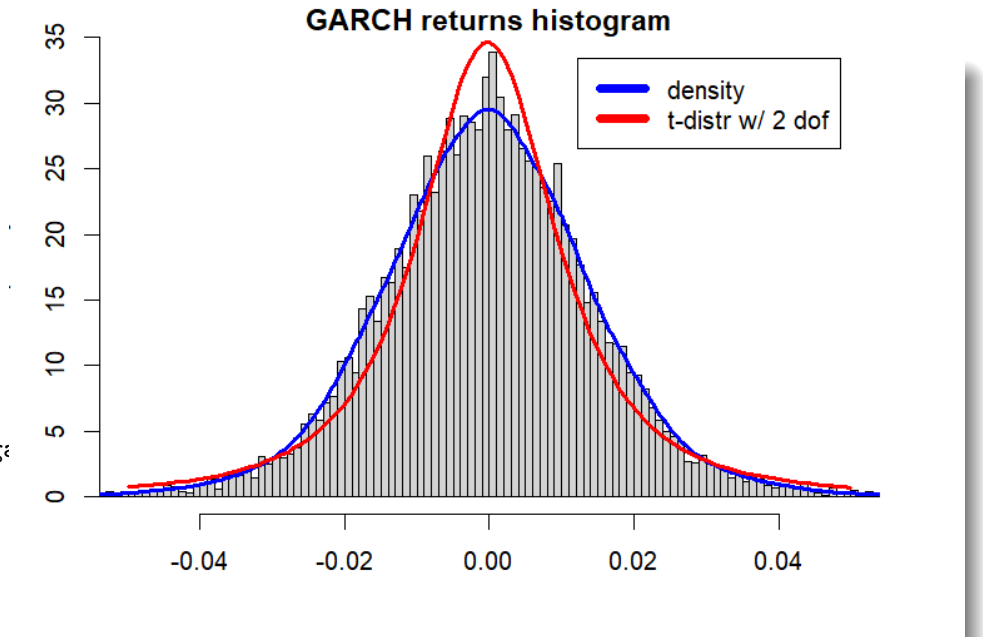
GARCH Model Simulation

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




```
> # Fit t-distribution into GARCH returns
> fitobj <- MASS::fitdistr(retsp, densfun="t", df=2, lower=c(-1, 1e-
> 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",
+  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:

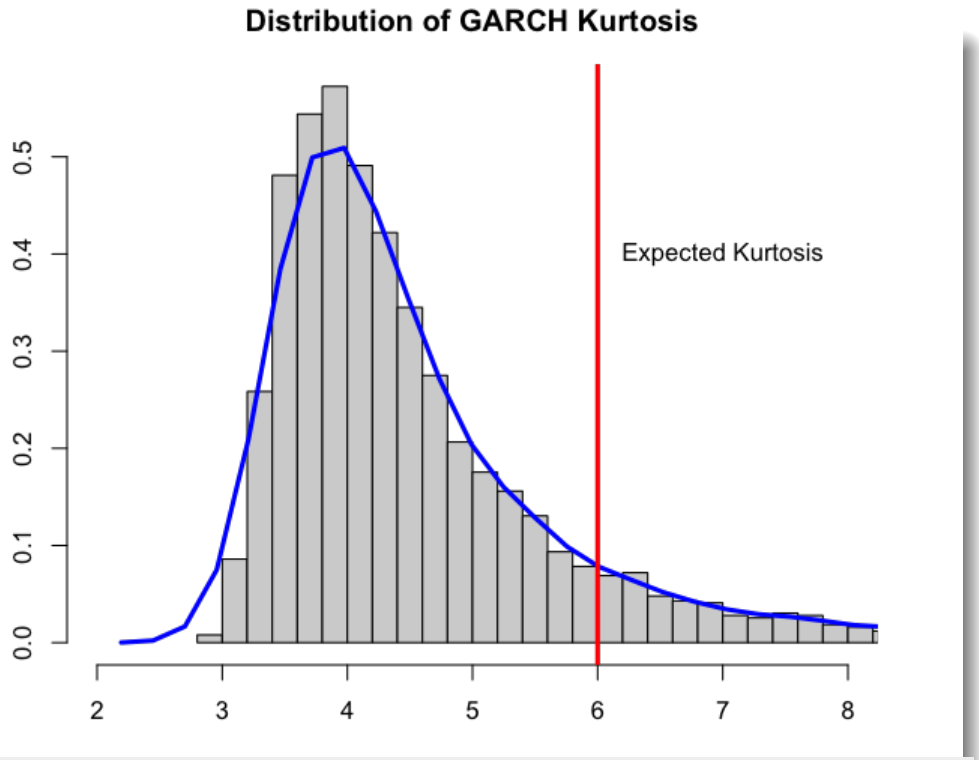
$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

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

```
> # Calculate the distribution of GARCH kurtosis
> kurt <- sapply(1:1e4, function(x) {
+   garch_data <- HighFreq::sim_garch(omega=omega, alpha=alpha,
+     beta=betav, innov=matrix(rnorm(nrows)))
+   retsp <- garch_data[, 1]
+   c(var(retsp), mean(((retsp-mean(retsp))/sd(retsp))^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 rolling 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 rolling variance σ_t^2:

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

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 rolling 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=alpha,
+   beta=betav, innov=matrix(innov))
> # Extract the returns
> retsp <- garch_data[, 1]
> # Estimate the rolling variance from the returns
> varv <- numeric(nrows)
> varv[1] <- omega/(1-alpha-betav)
> for (i in 2:nrows) {
+   varv[i] <- omega + alpha*retsp[i]^2 +
+     betav*varv[i-1]
+ }  # end for
> all.equal(garch_data[, 2], variance, check.attributes=FALSE)
```

GARCH Model Calibration

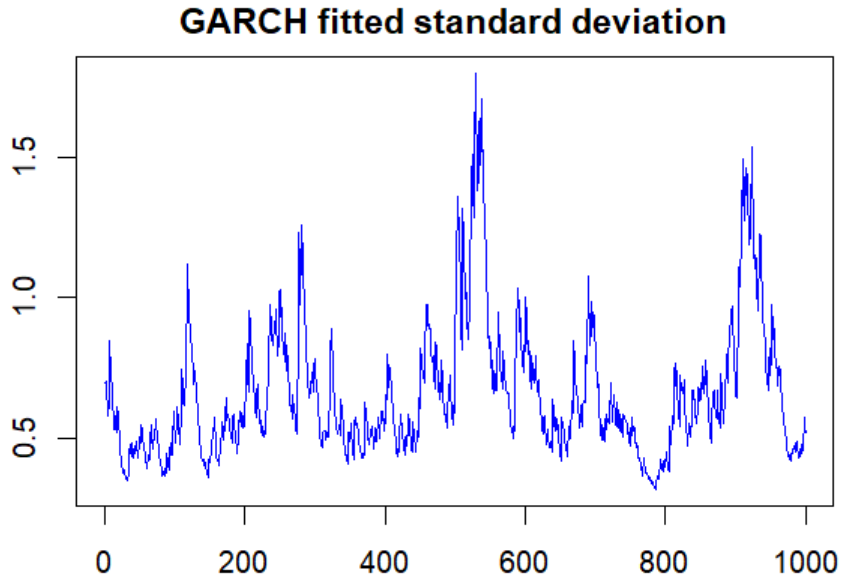
GARCH models can be calibrated from the returns using the *maximum-likelihood* method.

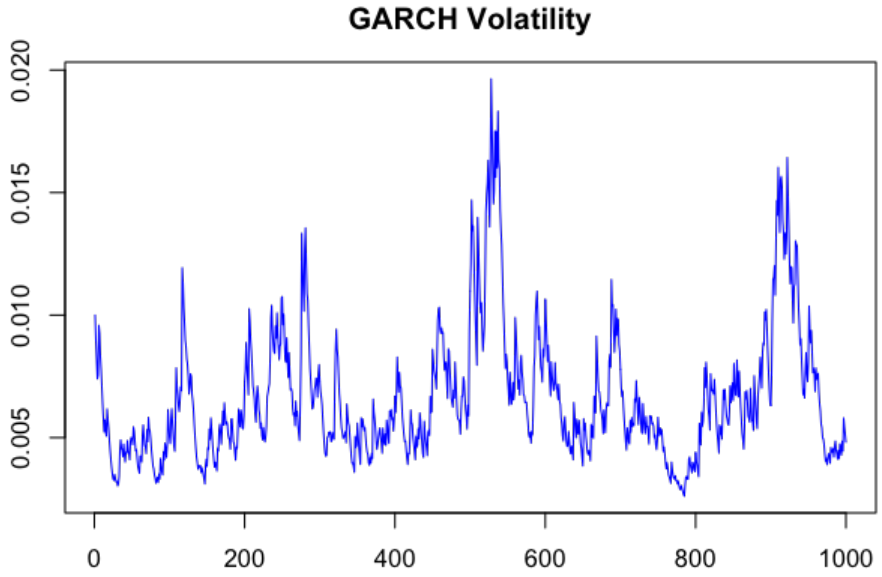
But it's a complex optimization procedure which requires a large amount of data for accurate results.

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=retsp)
> # Fitted GARCH parameters
> garch_fit@fit$coef
> # Actual GARCH parameters
> c(mu=mean(retsp), omega=omega,alpha=alpha, beta=betav)
> # Plot GARCH fitted volatility
> plot(sqrt(garch_fit@fit$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: $r_t = \mu + \sigma_{t-1}\xi_t$. (We can assume that the returns have been de-meaned, so that $\mu = 0$.)

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 rolling variance σ_t^2 depends on the *GARCH(1,1)* parameters:

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

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, alpha, betav) {
+   # Estimate the rolling variance from the returns
+   varv <- numeric(nrows)
+   varv[1] <- omega/(1-alpha-betav)
+   for (i in 2:nrows) {
+     varv[i] <- omega + alpha*retsp[i]^2 + betav*varv[i-1]
+   }  # end for
+   varv <- ifelse(varv > 0, variance, 0.000001)
+   # Lag the variance
+   varv <- rutils::lagit(varv, pad_zeros=FALSE)
+   # Calculate the likelihood
+   -sum(retsp^2/variance + log(varv))
+ }  # end likefun
> # Calculate the likelihood in R
> likefun(omega, alpha, betav)
> # Calculate the likelihood in Rcpp
> HighFreq::lik_garch(omega=omega, alpha=alpha,
+   beta=betav, returns=matrix(retsp))
> # Benchmark speed of likelihood calculations
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode=likefun(omega, alpha, betav),
+   Rcpp=HighFreq::lik_garch(omega=omega, alpha=alpha, beta=betav, r
+   ), 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.

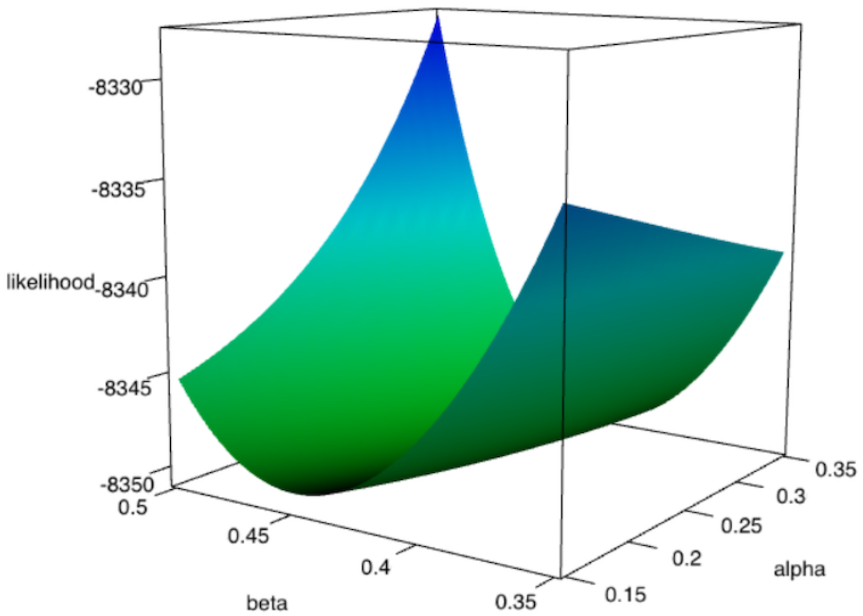
```
> # Calculate the variance of returns
> retsp <- garch_data[, 1, drop=FALSE]
> varv <- var(retsp)
> retsp <- (retsp - mean(retsp))
> # Calculate likelihood as function of alpha and betav parameters
> likefun <- function(alpha, betav) {
+   omega <- variance*(1 - alpha - betav)
+   -HighFreq::lik_garch(omega=omega,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

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(alphas, betas, 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(alphas[coord[2]], betas[coord[1]])
> likmat[coord]
> likefun(alphas[coord[2]], betas[coord[1]])
> # Optimal and actual parameters
> options(scipen=2)  # Use fixed not scientific notation
> cbind(actual=c(alpha=alpha, beta=betav, omega=omega),
+   optimal=c(alphas[coord[2]], betas[coord[1]], variance*(1 - sum(alphas[coord[2]], betas[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, retsp) {
+   alpha <- x[1]; betav <- x[2]; omega <- x[3]
+   -HighFreq::lik_garch(omega=omega, alpha=alpha, beta=betav, retu
+ }  # end likefun
> # Initial parameters
> initp <- c(alpha=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=retsp,
+   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(alpha=alpha, beta=betav, omega=omega),
+ optimal=c(fitobj$par["alpha"], fitobj$par["beta"], fitobj$par["ome
> # Find max likelihood parameters using DEoptim
> optiml <- DEoptim::DEoptim(fn=likefun,
+   upper=c(0.35, 0.55, varv), # Upper constraint
+   lower=c(0.15, 0.35, varv/100), # Lower constraint
+   returns=retsp,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal and actual parameters
> cbind(actual=c(alpha=alpha, beta=betav, omega=omega),
+ optimal=c(optiml$optim$bestmem[1], optiml$optim$bestmem[2], optiml
```

GARCH Variance of Stock Returns

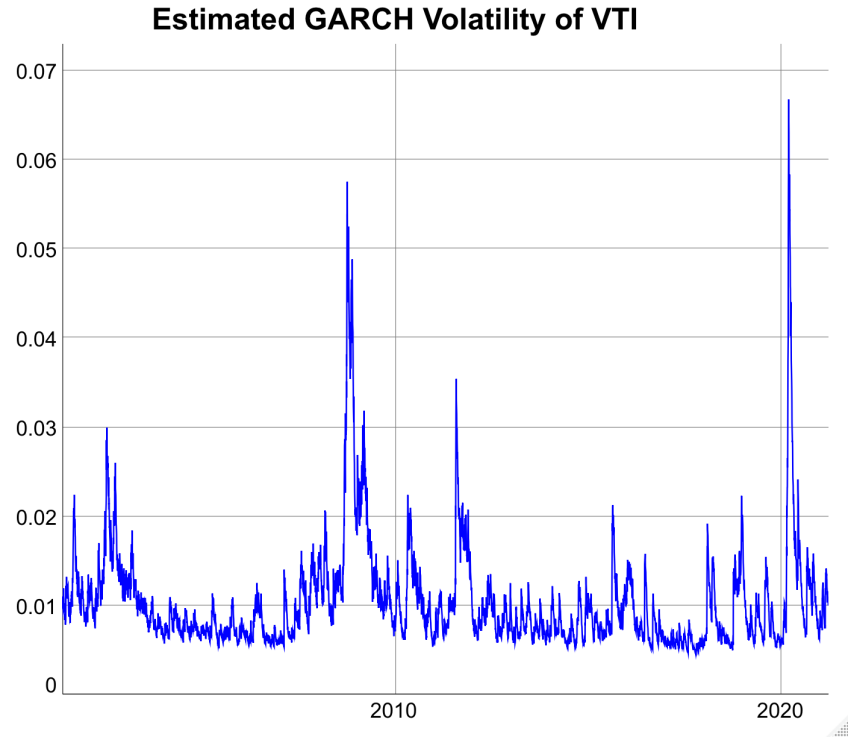
The *GARCH* model can be used to estimate the rolling variance of empirical (historical) returns.

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

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

The *GARCH* formula can be viewed as a generalization of the *EWMA* rolling variance.

```
> # Calculate VTI returns
> retvti <- 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=retvti,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal parameters
> par_am <- unname(optiml$optim$bestmem)
> alpha <- par_am[1]; betav <- par_am[2]; omega <- par_am[3]
> c(alpha, betav, omega)
> # Equilibrium GARCH variance
> omega/(1-alpha-betav)
> drop(var(retvti))
```




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

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 $\mathbb{E}[\xi_t^2] = 1$.

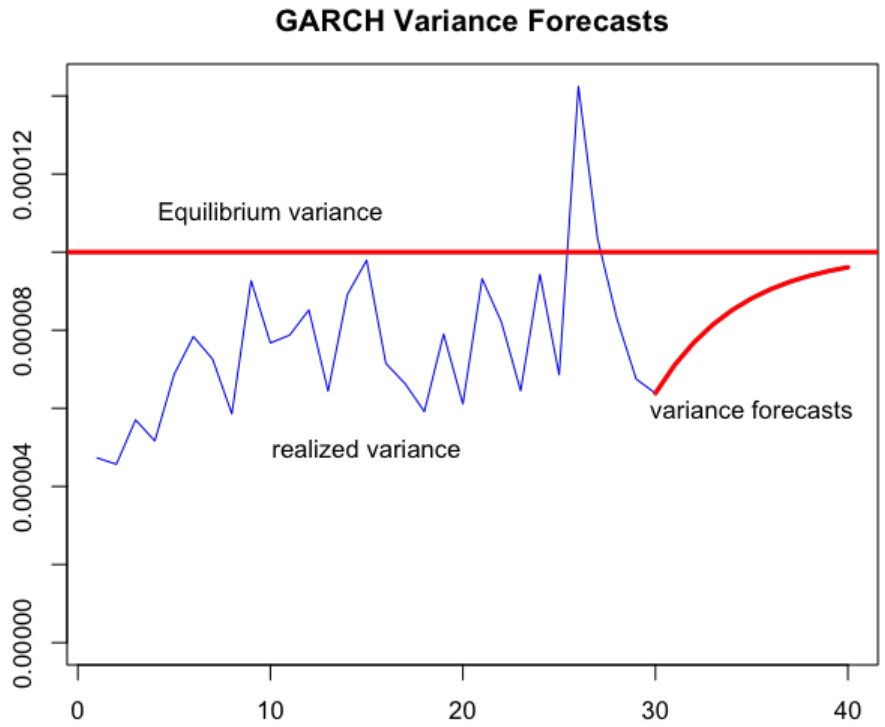
So the variance forecasts depend on the variance in the previous period:
$\sigma_{t+1}^2 = \mathbb{E}[\omega + \alpha r_{t+1}^2 + \beta \sigma_t^2] = \omega + (\alpha + \beta)\sigma_t^2$

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=alpha,
+   beta=betav, innov=matrix(innov))
> varv <- garch_data[, 2]
> # Calculate the equilibrium variance
> vareq <- omega/(1-alpha-betav)
> # Calculate the variance forecasts
> varf <- numeric(10)
> varf[1] <- vareq + (alpha + betav)*(xts::last(varv) - vareq)
> for (i in 2:10) {
+   varf[i] <- vareq + (alpha + betav)*(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

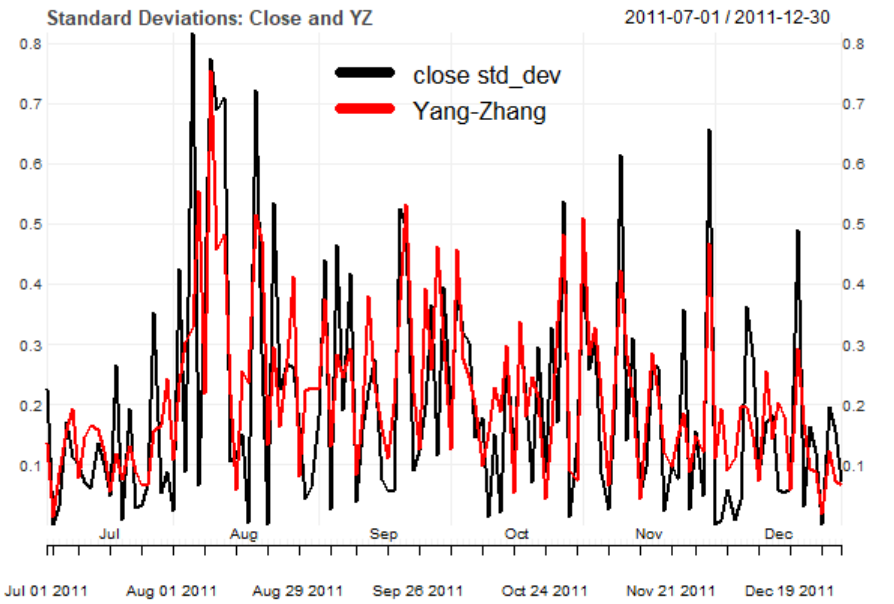
The range volatility estimators have much lower variability (standard errors) than the standard *Close-to-Close* estimator.

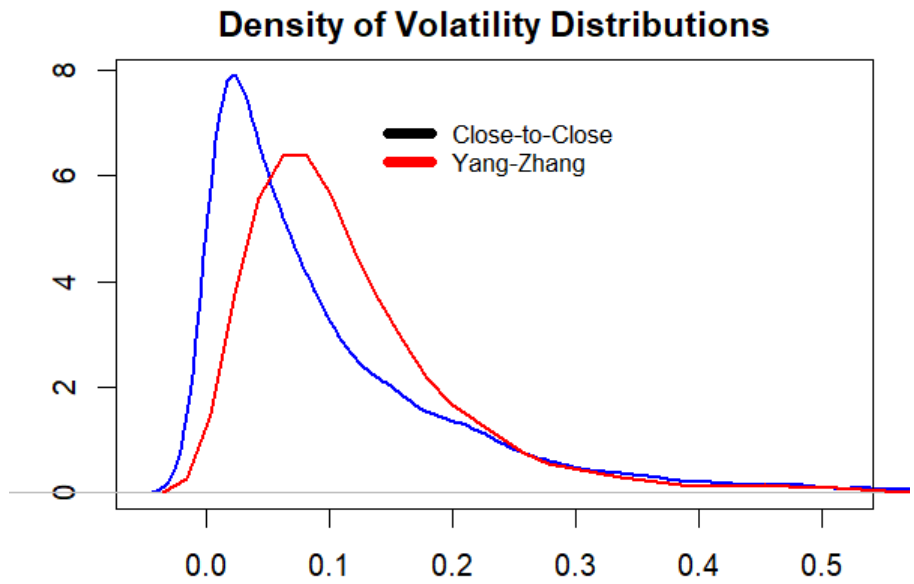
Is the above correct? Because the plot shows otherwise.

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$line.col <- c("black", "red")
> quantmod::chart_Series(stdev["2011-07/2011-12"],
+   theme=plot_theme, name="Standard Deviations: Close and YZ")
> legend("top", legend=colnames(stdev),
+  bg="white", lty=1, lwd=6, inset=0.1, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
> # Plot volatility around 2010 flash crash
> quantmod::chart_Series(stdev["2010-04/2010-06"],
+   theme=plot_theme, name="Volatility Around 2010 Flash Crash")
> legend("top", legend=colnames(stdev),
+  bg="white", lty=1, lwd=6, inset=0.1, cex=0.8,
+  col=plot_theme$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).

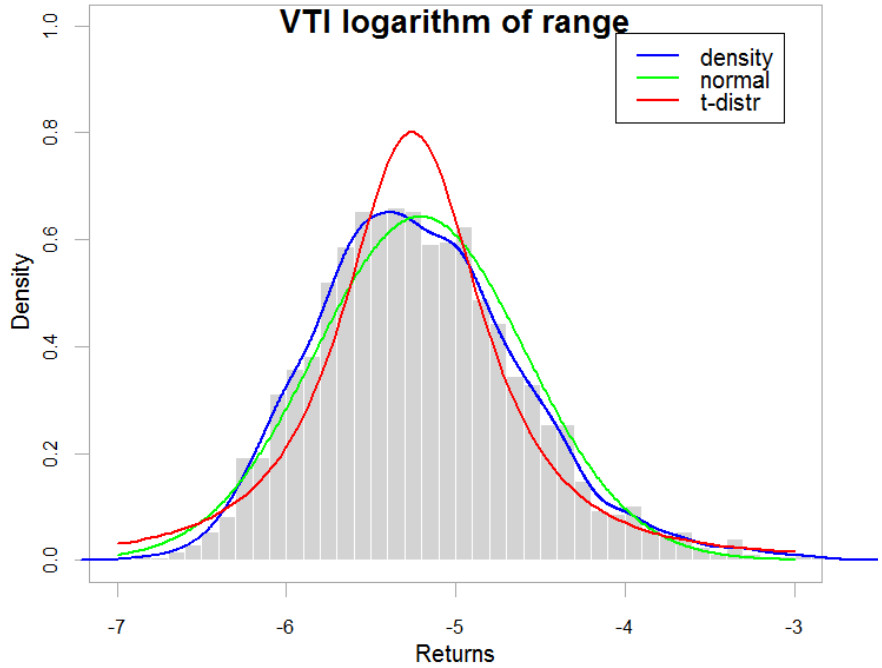
To-do: plot scatterplot of *intra-day range* volatility estimator and standard close-to-close volatility estimator.

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

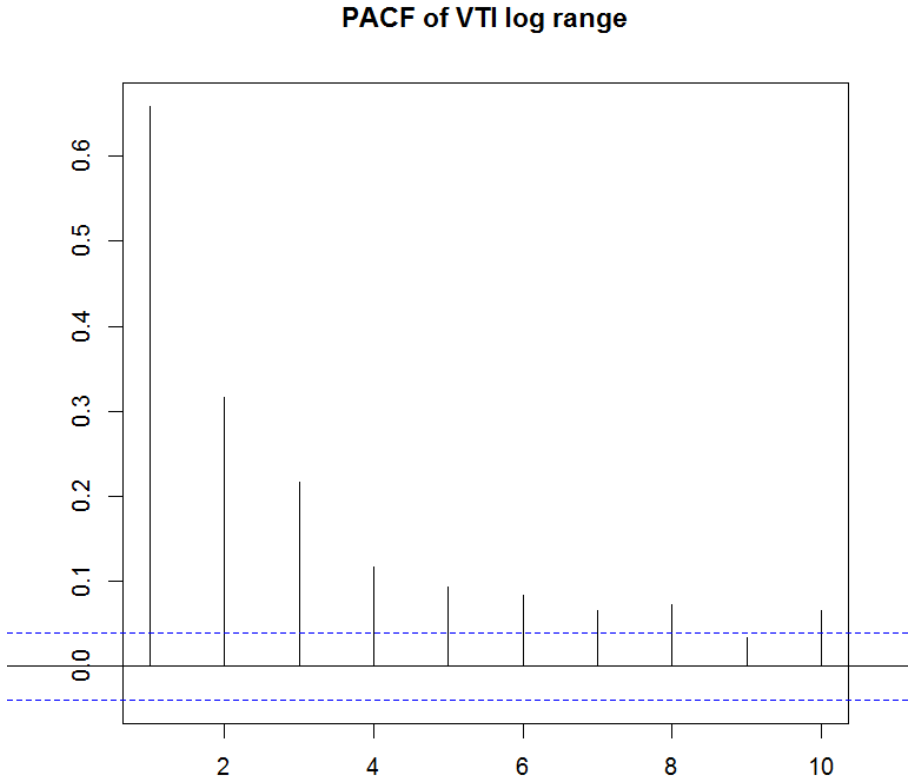



```
> # Plot histogram of VTI returns
> colorv <- c("lightgray", "blue", "green", "red")
> PerformanceAnalytics::chart.Histogram(retsp,
+   main="", xlim=c(-7, -3), col=colorv[1:3],
+   methods = c("add.density", "add.normal"))
> curve(expr=dt((x-fitobj$estimate[1])/
+   fitobj$estimate[2], df=2)/fitobj$estimate[2],
+ 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,
+   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(retsp^2, lag=10, xlab=NA, ylab=NA,
+      main="PACF of VTI log range")
> quantmod::chart_Series(retsp^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.

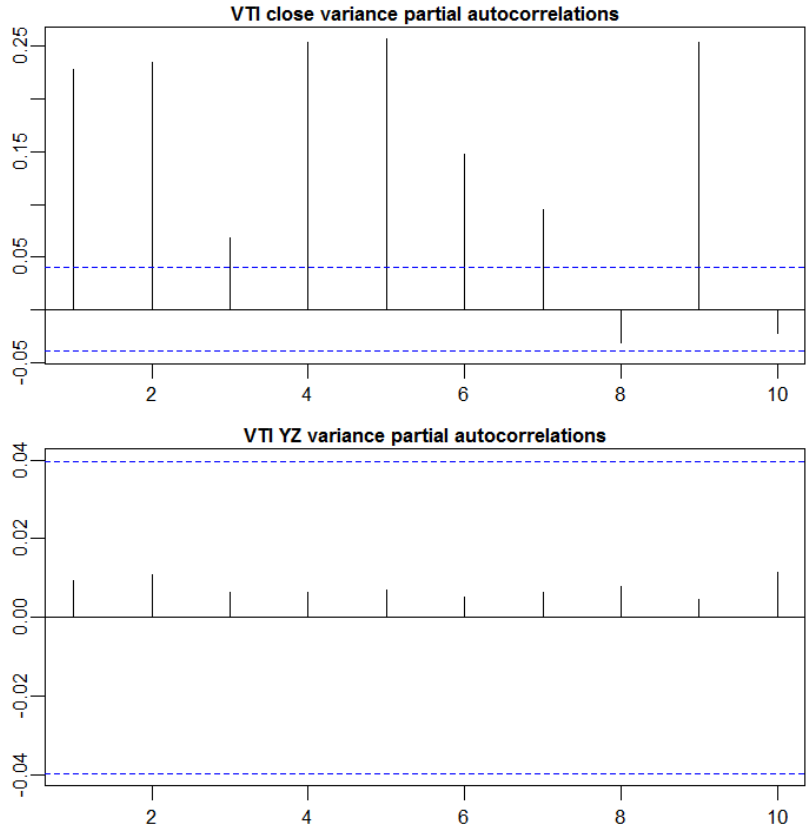
```
> # Standard errors of variance estimators using bootstrap
> bootd <- sapply(1:1e2, function(x) {
+   # Create random OHLC
+   ohlc <- HighFreq::random_ohlc()
+   # Calculate variance estimate
+   c(var=var(ohlc[, 4]),
+     yang_zhang=HighFreq::calcvariance(
+ ohlc, method="yang_zhang", scalev=FALSE))
+ })  # 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
> retsp <- log(rutils::etfenv$VTI[,2] /
+              rutils::etfenv$VTI[,3])
> pacf(retsp^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*.

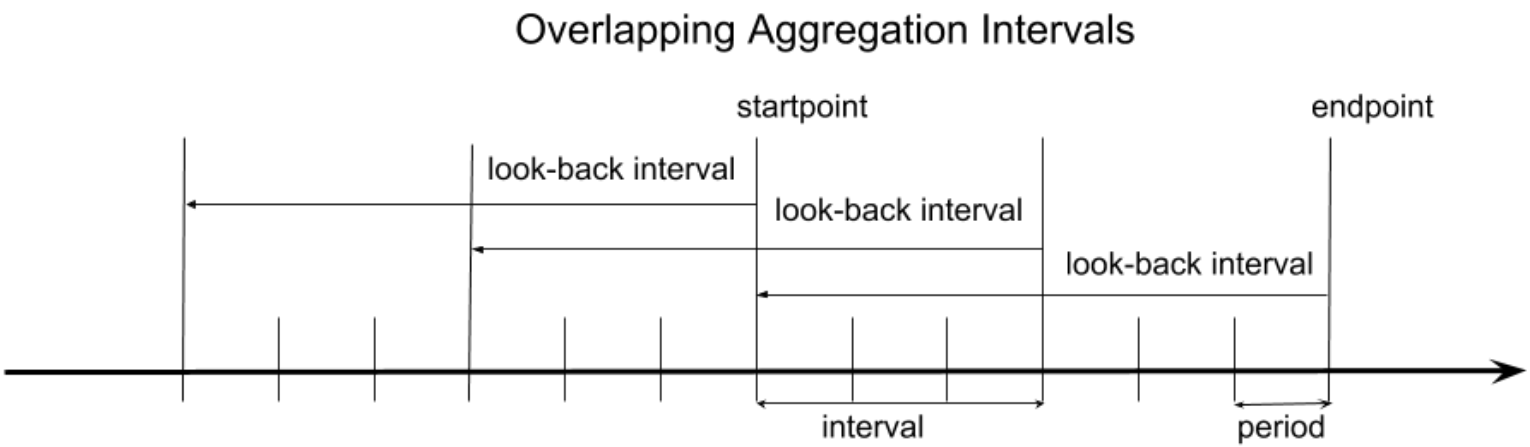
The look-back *intervals* may or may not *overlap* with their neighboring intervals.

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

An example of a rolling 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 *Rolling* Look-back Time Intervals

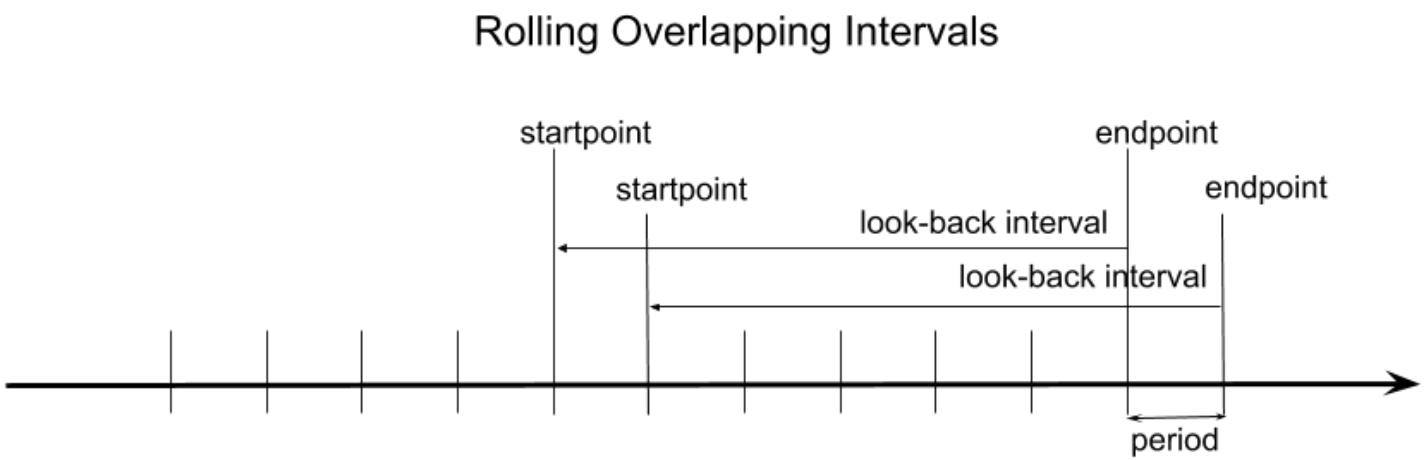
A *rolling aggregation* is performed over a vector of *end points* in time.

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 rolling 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)
```



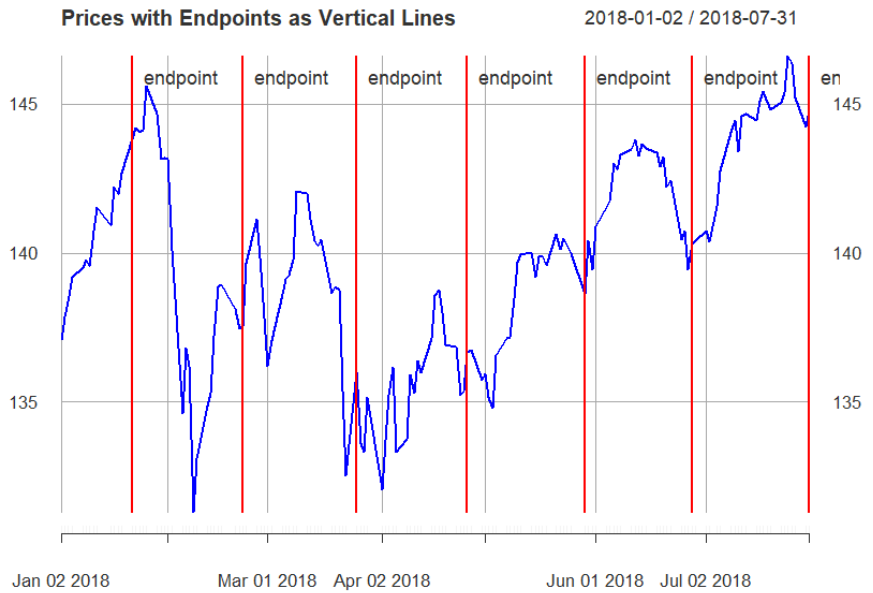
Defining Equally Spaced *end points* of a Time Series

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="months")
```




```
> # 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

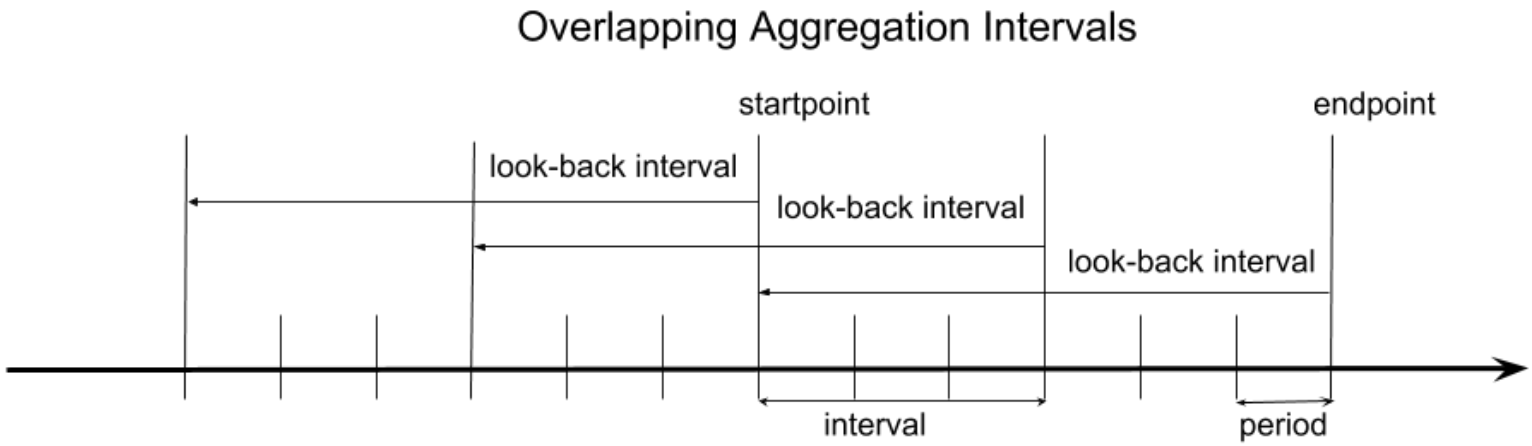
Overlapping time intervals can be defined if the *start points* are equal to the *end points* lagged by the *look-back interval*.

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

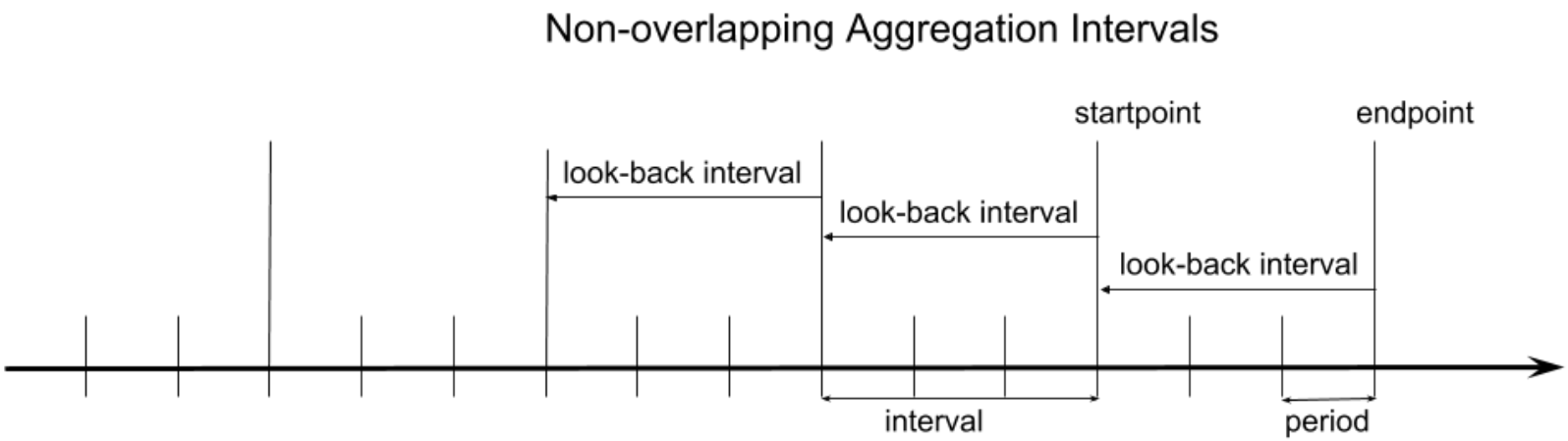
Non-overlapping time intervals can be defined if *start points* are equal to the previous *end points*.

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 Rolling 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 Rolling 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,
+   col=plot_theme$col$line.col, bty="n")
```

Defining Functionals for Rolling Aggregations

The functional `roll_agg()` performs rolling 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 *EWMA* window).

```
> # Define functional for rolling 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 rolling interval
> aggs <- roll_agg(closep, look_back=look_back, FUN=aggfun)
> class(aggs)
> dim(aggs)
```

Benchmarking Speed of Rolling Aggregations

The speed of rolling 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 Rolling Aggregations

Several packages contain functionals designed for performing rolling 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 rolling 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 rolling 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 rolling interval
> aggs <- xts:::rollapply.xts(closep, width=look_back,
+               FUN=aggfun, align="right")
> # Perform aggregations over a rolling 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)]
```

Rolling 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 rolling sum of an an *xts* series.

Using the function `cumsum()` is over 25 times faster than using `apply()` loops.

But rolling volatilities and higher moments can't be easily calculated using `cumsum()`.

```
> # Rolling 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 rolling 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 EWMA prices using filter()
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> filtered <- stats::filter(closep, filter=weightv,
+                   method="convolution", sides=1)
> filtered <- as.numeric(filtered)
> # filter() returns time series of class "ts"
> class(filtered)
> # Filter using compiled C++ function directly
> getAnywhere(C_cfilter)
> str(stats:::C_cfilter)
> filterfast <- .Call(stats:::C_cfilter, closep,
+                filter=weightv, sides=1, circular=FALSE)
> all.equal(as.numeric(filtered), filterfast, check.attributes=FALSE
> # Calculate EWMA prices using roll::roll_sum()
> weightrev <- rev(weightv)
> filtercpp <- roll::roll_sum(closep, width=look_back, weights=weigh
> all.equal(filtered[-(1:look_back)],
+     as.numeric(filtercpp)[-(1:look_back)],
+     check.attributes=FALSE)
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   filterfast=.Call(stats:::C_cfilter, closep, filter=weightv, side
+   cumsumv=cumsum(closep),
+   roll=roll::roll_sum(closep, width=look_back, weights=weightrev)
+   ), times=10)[, c(1, 4, 5)]
```

Performing Rolling Aggregations Using Package *TTR*

The package *TTR* contains functions for calculating rolling aggregations over *vectors* and *time series* objects:

- `runSum()` for rolling sums,
- `runMin()` and `runMax()` for rolling minima and maxima,
- `runSD()` for rolling standard deviations,
- `runMedian()` and `runMAD()` for rolling medians and Median Absolute Deviations (*MAD*),
- `runCor()` for rolling correlations,

The rolling *TTR* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` or `Fortran` code).

But the rolling *TTR* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> # Calculate the rolling maximum and minimum over a vector of data
> roll_maxminr <- function(vectorv, look_back) {
+   nrows <- NROW(vectorv)
+   max_min <- matrix(numeric(2:nrows), nc=2)
+   # Loop over periods
+   for (it in 1:nrows) {
+     sub_vec <- vectorv[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)]
```

Rolling *Weighted* Aggregations Using Package *roll*

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

- `roll_sum()`, `roll_max()`, `roll_mean()`, and `roll_median()` for *weighted* rolling sums, maximums, means, and medians,
- `roll_var()` for *weighted* rolling variance,
- `roll_scale()` for rolling scaling and centering of time series,
- `roll_lm()` for rolling regression,
- `roll_pcr()` for rolling 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 rolling VTI variance using package roll
> library(roll)  # Load roll
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> look_back <- 22
> # Calculate rolling sum using RcppRoll
> sum_roll <- roll::roll_sum(retvti, width=look_back, min_obs=1)
> # Calculate rolling sum using rutils
> sum_rutils <- rutils::roll_sum(retvti, look_back=look_back)
> all.equal(sum_roll[-(1:look_back), ],
+     sum_rutils[-(1:look_back), ], check.attributes=FALSE)
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retvti),
+   roll=roll::roll_sum(retvti, width=look_back),
+   RcppRoll=RcppRoll::roll_sum(retvti, n=look_back),
+   rutils=rutils::roll_sum(retvti, look_back=look_back),
+   times=10))[, c(1, 4, 5)]
```

Rolling *Weighted* Aggregations Using Package *RcppRoll*

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

- `roll_sum()` for *weighted* rolling sums,
- `roll_min()` and `roll_max()` for *weighted* rolling minima and maxima,
- `roll_sd()` for *weighted* rolling standard deviations,
- `roll_median()` for *weighted* rolling medians,

The *RcppRoll* functions accept *xts* objects, but they return matrices, not *xts* objects.

The rolling *RcppRoll* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` code).

But the rolling *RcppRoll* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> library(RcppRoll)  # Load package RcppRoll
> # Calculate rolling sum using RcppRoll
> sum_roll <- RcppRoll::roll_sum(retvti, align="right", n=look_back)
> # Calculate rolling sum using rutils
> sum_rutils <- rutils::roll_sum(retvti, look_back=look_back)
> all.equal(sum_roll, coredata(sum_rutils[-(1:(look_back-1))]),
+     check.attributes=FALSE)
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retvti),
+   RcppRoll=RcppRoll::roll_sum(retvti, n=look_back),
+   rutils=rutils::roll_sum(retvti, look_back=look_back),
+   times=10))[, c(1, 4, 5)]
> # Calculate EWMA prices using RcppRoll
> closep <- quantmod::Cl(rutils::etfenv$VTI)
> weightv <- exp(0.1*1:look_back)
> prices_ewma <- RcppRoll::roll_mean(closep,
+ align="right", n=look_back, weights=weightv)
> prices_ewma <- cbind(closep,
+   rbind(coredata(closep[1:(look_back-1), ]), prices_ewma))
> colnames(prices_ewma) <- c("VTI", "VTI EWMA")
> # Plot an interactive dygraph plot
> dygraphs::dygraph(prices_ewma)
> # Or static plot of EWMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> quantmod::chart_Series(prices_ewma, theme=plot_theme, name="EWMA p
> legend("top", legend=colnames(prices_ewma),
+  bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Performing Rolling Aggregations Using Package *caTools*

The package *caTools* contains functions for calculating rolling interval aggregations over a `vector` of data:

- `runmin()` and `runmax()` for rolling minima and maxima,
- `runsd()` for rolling standard deviations,
- `runmad()` for rolling Median Absolute Deviations (*MAD*),
- `runquantile()` for rolling quantiles,

Time series need to be coerced to *vectors* before they are passed to *caTools* functions.

The rolling *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.

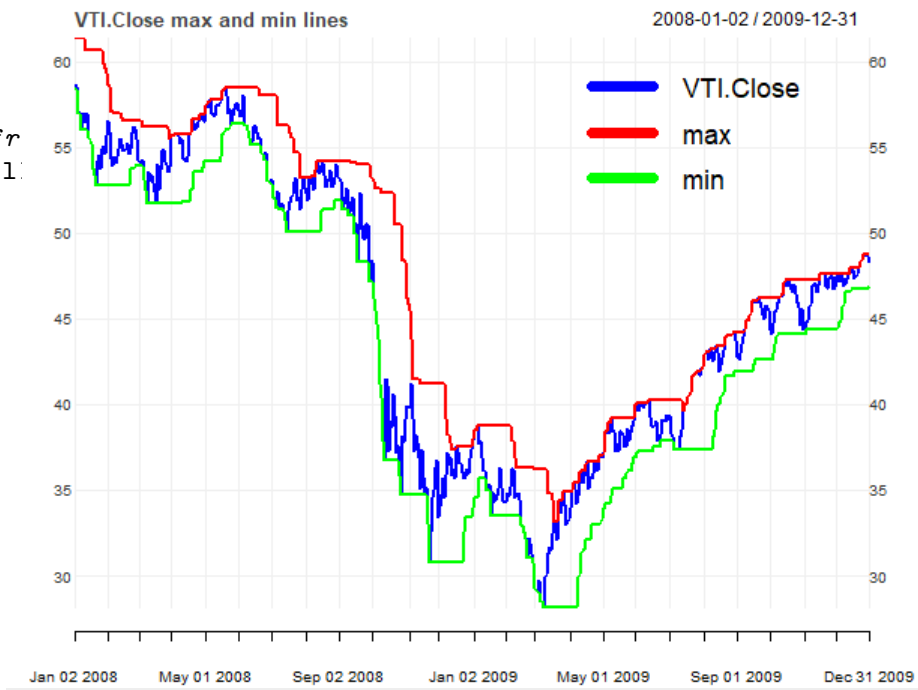
```
> library(caTools)  # Load package "caTools"
> # Get documentation for package "caTools"
> packageDescription("caTools")  # Get short description
> help(package="caTools")  # Load help page
> data(package="caTools")  # List all datasets in "caTools"
> ls("package:caTools")  # List all objects in "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 rolling volatilities
> sigmav <- runsd(x=closep, k=look_back,
+           endrule="constant", align="center")
> # Vector of rolling quantiles
> quantilevs <- runquantile(x=closep, k=look_back,
+   probs=0.9, endrule="constant", align="center")
```

Performing Rolling Aggregations Using *RcppArmadillo*

RcppArmadillo functions for calculating rolling 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++ namesvpace from Armadil
// declare dependency on RcppArmadillo
// [[Rcpp::depends(RcppArmadillo)]]

// export the function roll_maxmin() to R
// [[Rcpp::export]]
arma::mat roll_maxmin(const arma::vec& vectorv,
                      const arma::uword& look_back) {
  arma::uword.n_rows = vectorv.size();
  arma::mat max_min[nrows, 2);
  arma::vec sub_vec;
  // startup period
  max_min(0, 0) = vectorv[0];
  max_min(0, 1) = vectorv[0];
  for (uword it = 1; it < look_back; it++) {
    sub_vec = vectorv.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 = vectorv.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 rolling 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*).

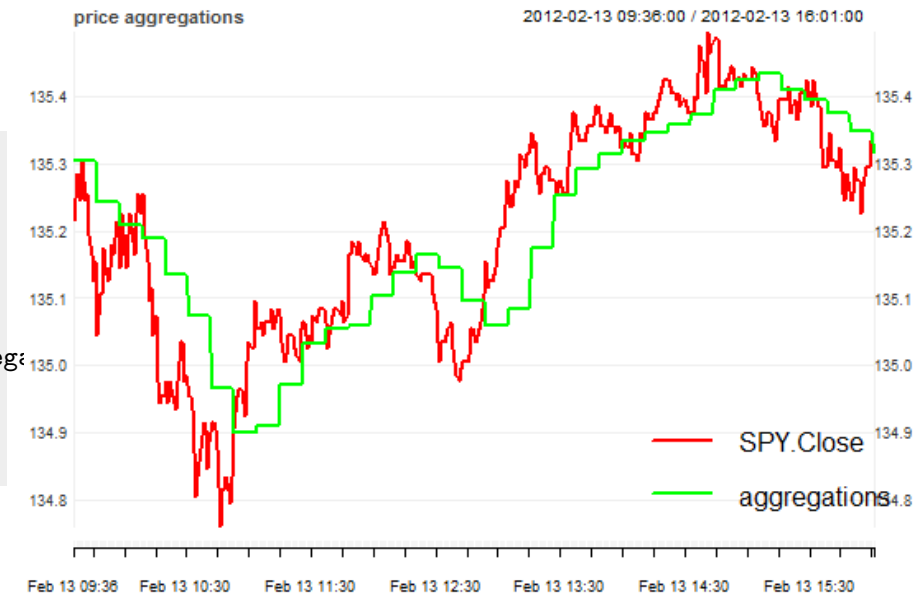
The variable `look_back` is equal to the number of end points in the *look-back interval*, while (`look_back - 1`) is equal to the number of intervals in the look-back.

```
> # 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")
```

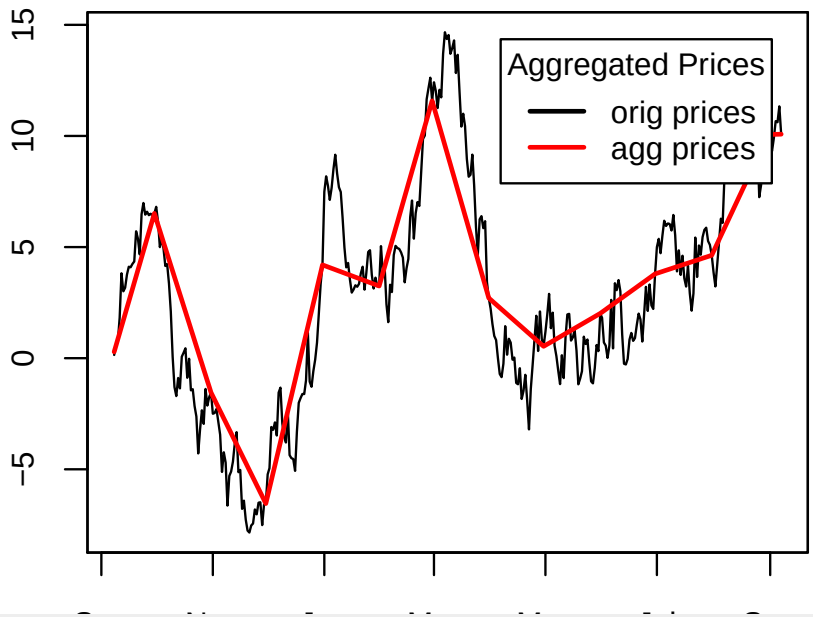


Performing Interval Aggregations of *zoo* Time Series

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.

```
> # Create zoo time series of random returns
> dates <- Sys.Date() + 0:365
> zoo_series <- zoo(rnorm(NROW(dates)), order.by=dates)
> # Create monthly dates
> dates_agg <- as.Date(as.yearmon(zoo::index(zoo_series)))
> # Perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=dates_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]
```



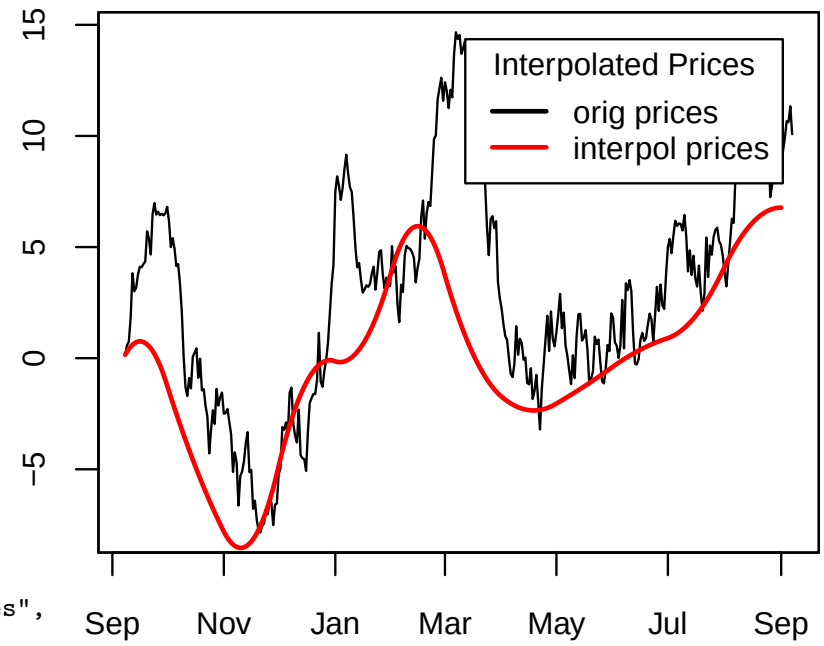

```
> # Plot original and aggregated cumulative returns
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, bty="n",
+   title="Aggregated Prices",
+   leg=c("orig prices", "agg prices"),
+   lwd=2, bg="white", 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=dates_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")
```

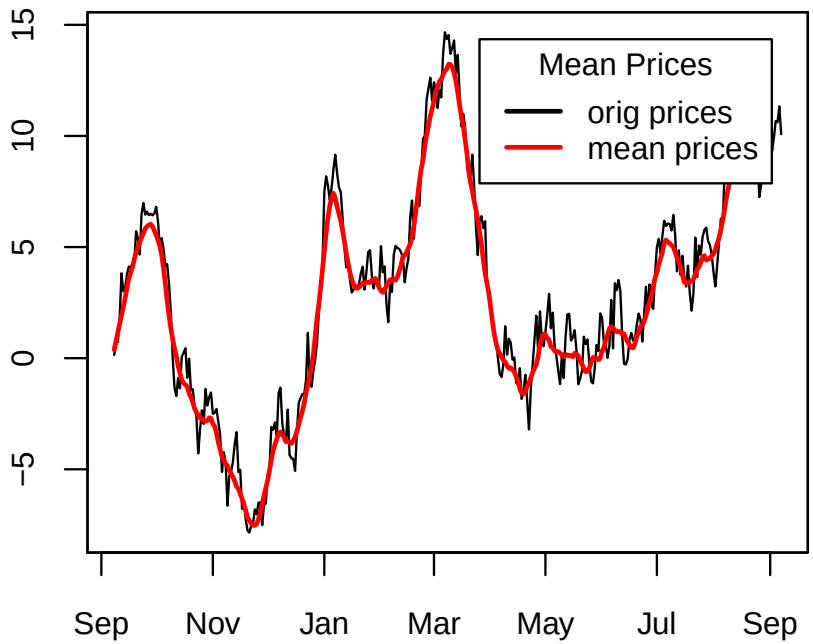


Performing Rolling Aggregations Over *zoo* Time Series

The package *zoo* has several functions for rolling calculations:

- `rollapply()` performing aggregations over a rolling (sliding) interval,
- `rollmean()` calculating rolling means,
- `rollmedian()` calculating rolling median,
- `rollmax()` calculating rolling 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")
```



aggregations are taken from the past,

The argument `align="right"` determines that