FRE7241 Algorithmic Portfolio Management

Lecture#7, Fall 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 22, 2024

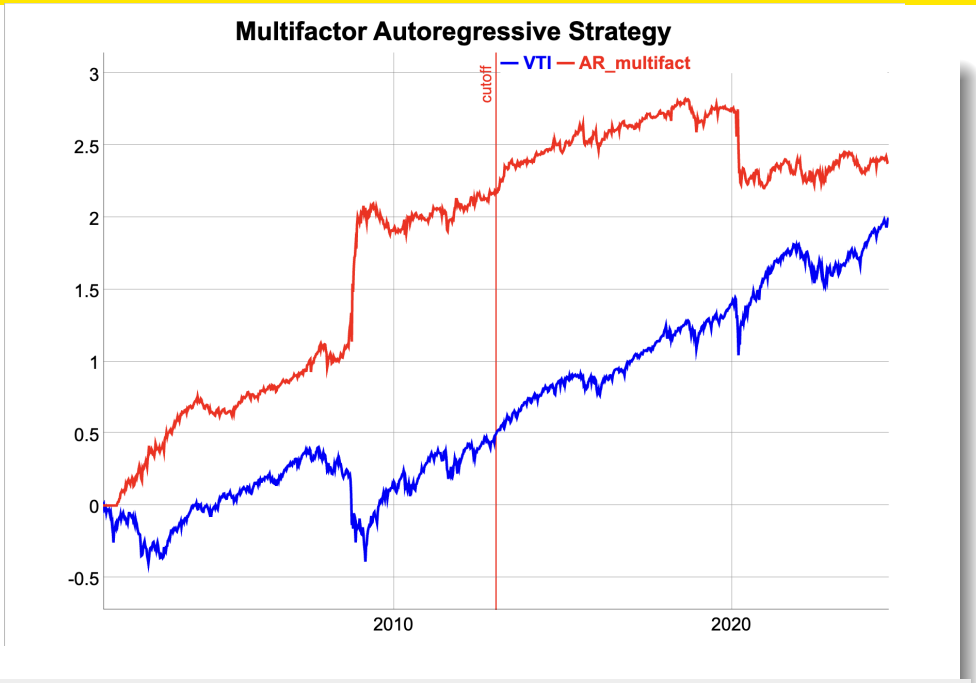
NYU | TANDON SCHOOL OF ENGINEERING

Multifactor Autoregressive Strategy

Multifactor autoregressive strategies can have a large number of predictors, and are often called *kitchen sink* strategies.

Multifactor strategies are overfit to the in-sample data because they have a large number of parameters.

The out-of-sample performance of the multifactor strategy is much worse than its in-sample performance, because the strategy is overfit to the in-sample data.




```
> # Calculate the VTI daily percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> datev <- index(retp)
> nrows <- NROW(retp)
> # Define in-sample and out-of-sample intervals
> insample <- 1:(nrows %/% 2)
> outsample <- (nrows %/% 2 + 1):nrows
> cutoff <- nrows %/% 2
> # Define the response and predictor matrices
> respv <- retp
> orderp <- 8 # 9 predictors!!!
> predm <- lapply(1:orderp, rutils::lagit, input=respv)
> predm <- rutils::do_call(cbind, predm)
> predm <- cbind(rep(1, nrows), predm)
> colnames(predm) <- c("phi0", paste0("lag", 1:orderp))
> # Calculate the in-sample fitted autoregressive coefficients
> predinv <- MASS::ginv(predm[insample, ])
> coeff <- predinv %*% respv[insample, ]
> # Calculate the in-sample forecasts of VTI (fitted values)
> fcasts <- predm %*% coeff
> fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.8)[, 2])
> fcastv[1:100] <- 1
> fcasts <- fcasts/fcastv
```

```
> # Calculate the autoregressive strategy PnLs
> pnls <- retp*fcasts
> pnls <- pnls*sd(retp[retp<0])/sd(pnls[pnls<0])
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "AR_multifact")
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Multifactor Autoregressive Strategy") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=300)
```

Regularization of the Inverse Predictor Matrix

The *SVD* of a rectangular matrix $\mathbb{A}$ is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\mathbf{\Sigma}\mathbb{V}^T$$

Where $\mathbb{U}$ and $\mathbb{V}$ are the *singular matrices*, and $\mathbf{\Sigma}$ is a diagonal matrix of *singular values*.

The *generalized inverse* matrix $\mathbb{A}^{-1}$ satisfies the inverse equation: $\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$, and it can be expressed as a product of the *SVD* matrices as follows:

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

If any of the *singular values* are zero then the *generalized inverse* does not exist.

Regularization is the removal of zero singular values, to make calculating the inverse matrix possible.

The *generalized inverse* is obtained by removing the zero *singular values*:

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

Where $\mathbb{U}_n$, $\mathbb{V}_n$ and $\mathbf{\Sigma}_n$ are the *SVD* matrices without the zero *singular values*.

The generalized inverse satisfies the inverse matrix equation: $\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$.

```
> # Calculate singular value decomposition of the predictor matrix
> svdec <- svd(predm)
> barplot(svdec$d, main="Singular Values of Predictor Matrix")
> # Calculate generalized inverse from SVD
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Verify inverse property of the inverse
> all.equal(zoo::coredata(predm), predm %*% invsvd %*% predm)
> # Compare with the generalized inverse using MASS::ginv()
> invreg <- MASS::ginv(predm)
> all.equal(invreg, invsvd)
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Check for zero singular values
> round(svdec$d, 12)
> notzero <- (svdec$d > (precv*svdec$d[1]))
> # Calculate generalized inverse from SVD
> invsvd <- svdec$v[, notzero] %*%
+   (t(svdec$u[, notzero]) / svdec$d[notzero])
> # Verify inverse property of invsvd
> all.equal(zoo::coredata(predm), predm %*% invsvd %*% predm)
> all.equal(invsvd, invreg)
```

Reduced Inverse of the Predictor Matrix

Regularization is the removal of zero singular values, to make calculating the inverse matrix possible.

If the higher order singular values are very small then the inverse matrix will amplify the noise in the response matrix.

Dimension reduction is achieved by the removal of small singular values, to improve the out-of-sample performance of the inverse matrix.

The *reduced inverse* is obtained by removing the very small *singular values*.

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

This effectively reduces the number of parameters in the model.

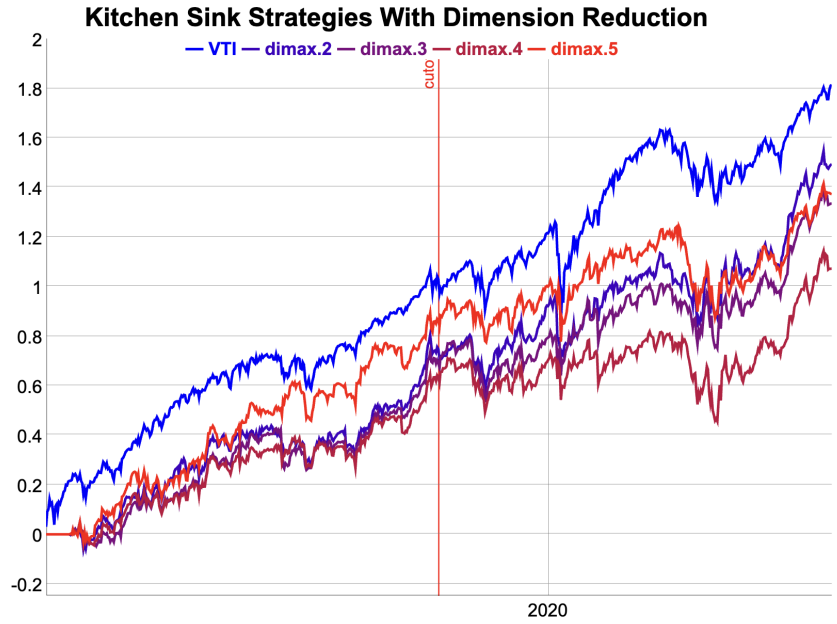
The *reduced inverse* satisfies the inverse equation only approximately (it is *biased*), but it's often used in machine learning because it produces a lower *variance* of the forecasts than the exact inverse.

```
> # Calculate reduced inverse from SVD
> dimax <- 3 # Number of dimensions to keep
> invred <- svdec$v[, 1:dimax] %*%
+   (t(svdec$u[, 1:dimax]) / svdec$d[1:dimax])
> # Inverse property fails for invred
> all.equal(zoo::coredata(predm), predm %*% invred %*% predm)
> # Calculate reduced inverse using RcppArmadillo
> invrcpp <- HighFreq::calc_invsvd(predm, dimax=dimax)
> all.equal(invred, invrcpp, check.attributes=FALSE)
```

Autoregressive Strategy With Dimension Reduction

Dimension reduction improves the out-of-sample, risk-adjusted performance of the multifactor autoregressive strategy.

The best performance is achieved with the smallest order parameter (strongest reduction) *dimax* = 2




```
> # Calculate the in-sample SVD
> svdec <- svd(predm[insample, ])
> # Calculate the in-sample fitted AR coefficients for different dir
> dimv <- 2:5
> # dimv <- c(2, 5, 10, NCOL(predm))
> coeffm <- sapply(dimv, function(dimax) {
+   predinv <- svdec$v[, 1:dimax] %*%
+     (t(svdec$u[, 1:dimax]) / svdec$d[1:dimax])
+   predinv %*% respv[insample]
+ })  # end lapply
> colnames(coeffm) <- paste0("dimax=", dimv)
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(coeffm))
> matplot(y=coeffm, type="l", lty="solid", lwd=1, col=colorv,
+   xlab="predictor", ylab="coeff",
+   main="AR Coefficients For Different Dimensions")
> # Calculate the forecasts of VTI
> fcasts <- predm %*% coeffm
> fcasts <- apply(fcasts, 2, function(x) {
+   fcastv <- sqrt(HighFreq::run_var(matrix(x), lambda=0.8)[, 2])
+   fcastv[1:100] <- 1
+   x/fcastv
+ }) # end apply
> # Simulate the autoregressive strategies
> retn <- coredata(retp)
> pnls <- apply(fcasts, 2, function(x) (x*retn))
> pnls <- xts(pnls, datev)
> # Scale the PnL volatility to that of VTI
> pnls <- lapply(pnls, function(x) x/sd(x))
> pnls <- sd(retp)*do.call(cbind, pnls)
```

```
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(retp, pnls)
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Autoregressive Strategies With Dimension Reduction") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=500)
```

Shrinkage of the Autoregressive Coefficients

The AR coefficients can be found by minimizing the sum of the squared errors of the in-sample forecasts.

The objective function is the sum of the squared errors of the in-sample forecasts, plus a penalty term proportional to the square of the AR coefficients:

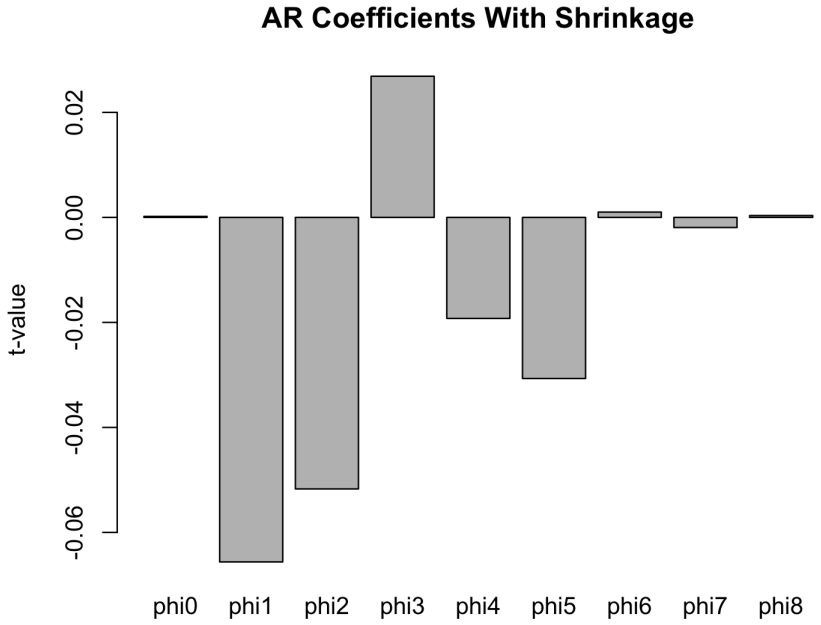
$$\text{ObjFunc} = \sum_{i=1}^{n}(r_t - f_t)^2 + \sum_{i=1}^{k} \varphi_i^2 \lambda^i$$

As the objective function is minimized, the *shrinkage* penalty shrinks the AR coefficients closer to zero.

A larger *shrinkage factor* λ applies more *shrinkage* to the AR coefficients, to shrink them closer to zero.

The *shrinkage* penalty is greater for the higher order coefficients, to reflect the fact that returns from the more distant past should be less important for forecasting the current returns.

The coefficient shrinkage also produces a dimension reduction effect, because the higher order coefficients are shrunk closer to zero.




```
> # Objective function for the in-sample AR coefficients
> objfun <- function(coeff, lambdaf) {
+   fcasts <- predm[insample, ] %*% coeff
+   lambdav <- lambdaf^(0:orderp)
+   10000*sum((respv[insample, ] - fcasts)^2) + sum(lambdav*coeff^
+ }  # end objfun
```

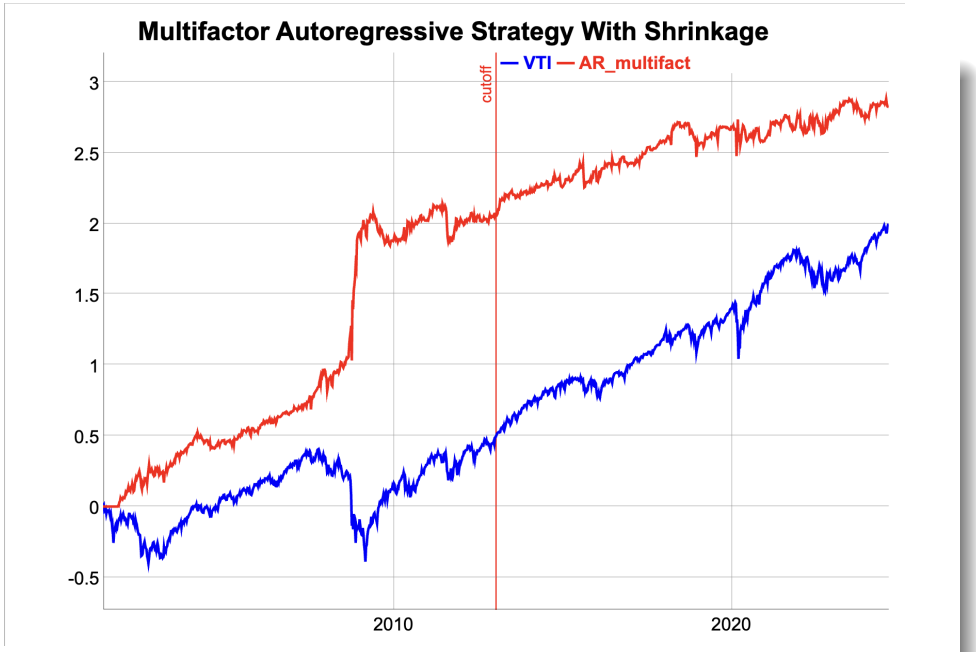
```
> # Perform optimization using the quasi-Newton method
> optiml <- optim(par=numeric(orderp+1),
+           fn=objfun, lambdaf=5.0, method="L-BFGS-B",
+           upper=rep(10000, orderp+1),
+           lower=rep(-10000, orderp+1))
> # Extract the AR coefficients
> coeff <- optiml$par
> coeffn <- paste0("phi", 0:orderp)
> names(coeff) <- coeffn
> barplot(coeff ~ coeffn, xlab="", ylab="t-value", col="grey",
+   main="AR Coefficients With Shrinkage")
```

Multifactor Autoregressive Strategy With Shrinkage

The out-of-sample performance of the multifactor autoregressive strategy can be improved by applying *shrinkage* to the AR coefficients.

The value of the *shrinkage factor* λ can be chosen to maximize the out-of-sample performance.

```
> # Calculate the forecasts of VTI (fitted values)
> fcasts <- predm %*% coeff
> fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.8)[, 2])
> fcastv[1:100] <- 1
> fcasts <- fcasts/fcastv
> # Calculate the autoregressive strategy PnLs
> pnls <- retp*fcasts
> pnls <- pnls*sd(retp[retp<0])/sd(pnls[pnls<0])
```




```
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "AR_multifact")
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Multifactor Autoregressive Strategy With Shrinkage") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=300)
```

Kitchen Sink Model of Stock Returns

The "kitchen sink" model uses many possible predictor variables, so the predictor matrix has a very large number of columns.

The predictor matrix includes the lagged and scaled daily returns and the squared returns.

stock returns r_t can be fitted into an *autoregressive* model *AR(n)* with a constant intercept term φ_0:

$$r_t = \varphi_0 + \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \varepsilon_t$$

The *residuals* ε_t are assumed to be normally distributed, independent, and stationary.

The autoregressive model can be written in matrix form as:

$$\mathbf{r} = \varphi \, \mathbb{P} + \varepsilon$$

Where $\varphi = \{\varphi_0, \varphi_1, \varphi_2, \ldots \varphi_n\}$ is the vector of autoregressive coefficients.

The *autoregressive* model is equivalent to *multivariate* linear regression, with the *response* equal to the returns $\mathbf{r}$, and the columns of the *predictor matrix* $\mathbb{P}$ equal to the lags of the returns.

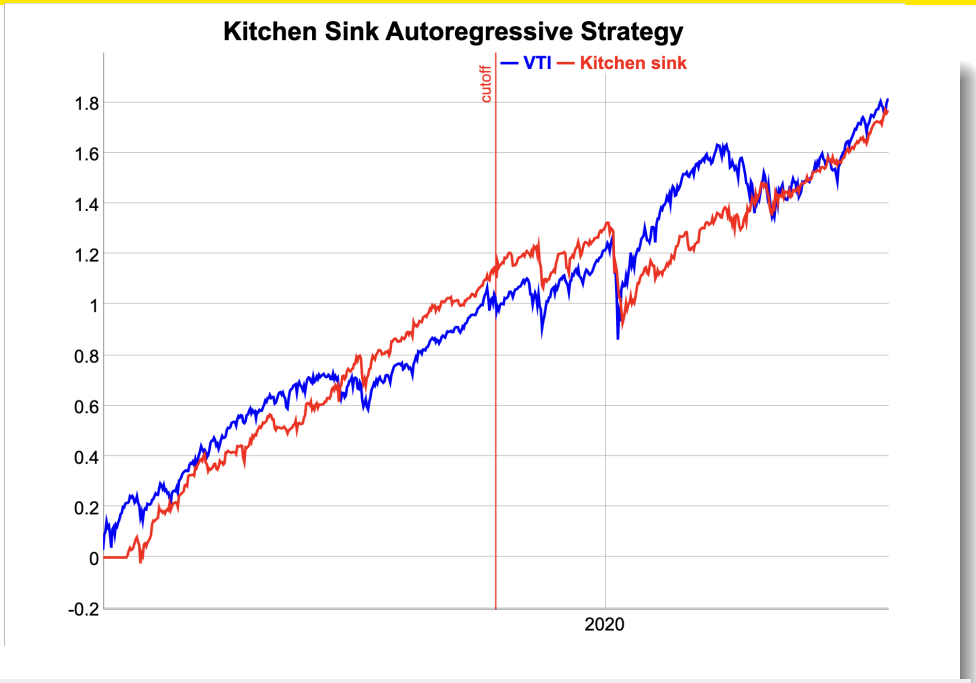
```
> # Calculate the returns of VTI, TLT, VXX, and SVXY
> retp <- na.omit(rutils::etfenv$returns[, c("VTI", "TLT", "VXX", "S
> datev <- zoo::index(retp)
> nrows <- NROW(retp)
> # Calculate VTI returns and trading volumes
> ohlc <- rutils::etfenv$VTI
> volumv <- rutils::diffit(quantmod::Vo(ohlc))[datev]
> # Define the response and the VTI predictor matrix
> respv <- retp$VTI
> orderp <- 5
> predm <- lapply(1:orderp, rutils::lagit, input=respv)
> predm <- rutils::do_call(cbind, predm)
> predm <- cbind(rep(1, nrows), predm)
> colnames(predm) <- c("phi0", paste0("lag", 1:orderp))
> # Add the TLT predictor matrix
> predx <- lapply(1:orderp, rutils::lagit, input=retp$TLT)
> predx <- rutils::do_call(cbind, predx)
> colnames(predx) <- paste0("TLT", 1:orderp)
> predm <- cbind(predm, predx)
> # Add the VXX predictor matrix
> predx <- lapply(1:orderp, rutils::lagit, input=retp$VXX)
> predx <- rutils::do_call(cbind, predx)
> colnames(predx) <- paste0("VXX", 1:orderp)
> predm <- cbind(predm, predx)
> # Perform the multivariate linear regression
> regmod <- lm(respv ~ predm - 1)
> summary(regmod)
```

Kitchen Sink Autoregressive Strategy

Multifactor autoregressive strategies can have a large number of predictors, and are often called *kitchen sink* strategies.

Multifactor strategies are overfit to the in-sample data because they have a large number of parameters.

The out-of-sample performance of the multifactor strategy is much worse than its in-sample performance, because the strategy is overfit to the in-sample data.




```
> # Define in-sample and out-of-sample intervals
> insample <- 1:(nrows %/% 2)
> outsample <- (nrows %/% 2 + 1):nrows
> cutoff <- nrows %/% 2
> # Calculate the in-sample fitted autoregressive coefficients
> predinv <- MASS::ginv(predm[insample, ])
> coeff <- predinv %*% respv[insample, ]
> coeffn <- colnames(predm)
> barplot(coeff ~ coeffn, xlab="", ylab="t-value", col="grey",
+   main="Coefficients of Kitchen Sink Autoregressive Model")
> # Calculate the in-sample forecasts of VTI (fitted values)
> fcasts <- predm %*% coeff
> fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.8)[, 2])
> fcastv[1:100] <- 1
> fcasts <- fcasts/fcastv
```

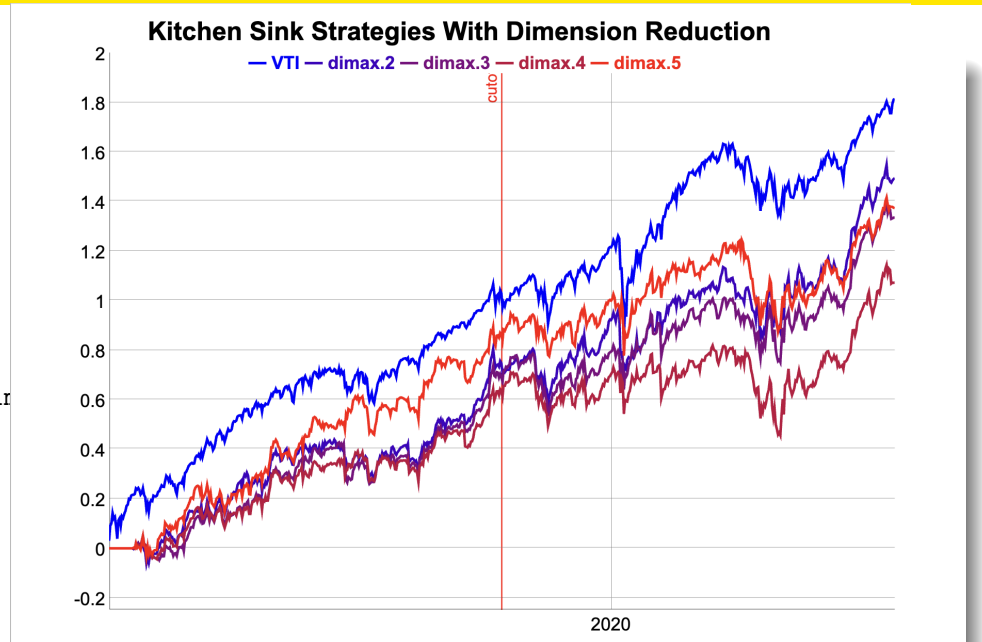
```
> # Calculate the autoregressive strategy PnLs
> pnls <- respv*fcasts
> pnls <- pnls*sd(respv[respv<0])/sd(pnls[pnls<0])
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Kitchen sink")
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Kitchen Sink Autoregressive Strategy") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=300)
```

Kitchen Sink Strategy With Dimension Reduction

Dimension reduction improves the out-of-sample, risk-adjusted performance of the multifactor autoregressive strategy.

The best performance is achieved with the smallest order parameter (strongest reduction) $dimax = 2$

```
> # Calculate the in-sample SVD
> svdec <- svd(predm[insample, ])
> # Calculate the in-sample fitted AR coefficients for different dir
> dimv <- 2:5
> # dimv <- c(2, 5, 10, NCOL(predm))
> coeffm <- sapply(dimv, function(dimax) {
+   predinv <- svdec$v[, 1:dimax] %*%
+     (t(svdec$u[, 1:dimax]) / svdec$d[1:dimax])
+   predinv %*% respv[insample]
+ })  # end lapply
> colnames(coeffm) <- paste0("dimax=", dimv)
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(coeffm))
> matplot(y=coeffm, type="l", lty="solid", lwd=1, col=colorv,
+   xlab="predictor", ylab="coeff",
+   main="AR Coefficients For Different Dimensions")
> # Calculate the forecasts of VTI
> fcasts <- predm %*% coeffm
> fcasts <- apply(fcasts, 2, function(x) {
+   fcastv <- sqrt(HighFreq::run_var(matrix(x), lambda=0.8)[, 2])
+   fcastv[1:100] <- 1
+   x/fcastv
+ }) # end apply
> # Simulate the autoregressive strategies
> retn <- coredata(respv)
> pnls <- apply(fcasts, 2, function(x) (x*retn))
> pnls <- xts(pnls, datev)
> # Scale the PnL volatility to that of VTI
> pnls <- lapply(pnls, function(x) x/sd(x))
> pnls <- sd(respv)*do.call(cbind, pnls)
```




```
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(respv, pnls)
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Kitchen Sink Strategies With Dimension Reduction") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=500)
```

Shrinkage of the Kitchen Sink Coefficients

The AR coefficients can be found by minimizing the sum of the squared errors of the in-sample forecasts.

The objective function is the sum of the squared errors of the in-sample forecasts, plus a penalty term proportional to the square of the AR coefficients:

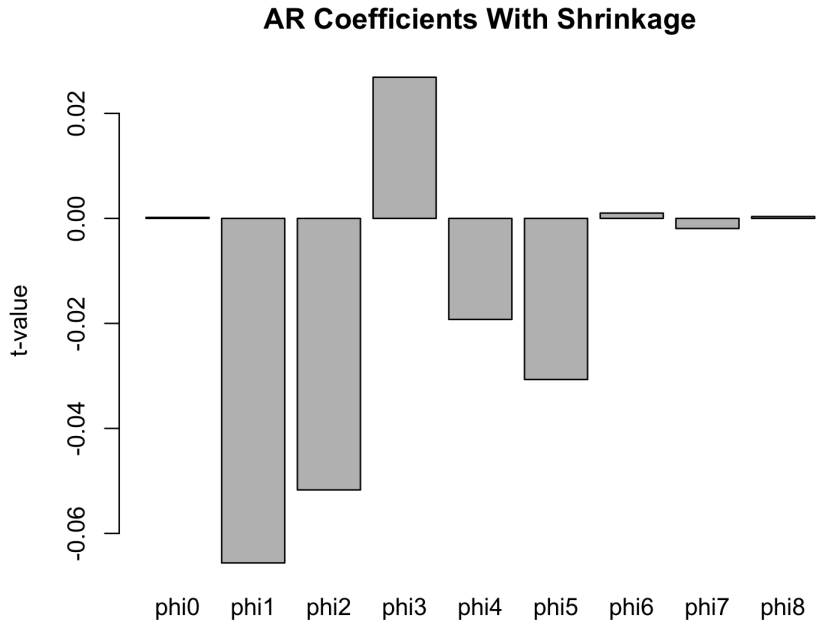
$$\text{ObjFunc} = \sum_{i=1}^{n}(r_t - f_t)^2 + \sum_{i=1}^{k} \varphi_i^2 \lambda^i$$

As the objective function is minimized, the *shrinkage* penalty shrinks the AR coefficients closer to zero.

A larger *shrinkage factor* λ applies more *shrinkage* to the AR coefficients, to shrink them closer to zero.

The *shrinkage* penalty is greater for the higher order coefficients, to reflect the fact that returns from the more distant past should be less important for forecasting the current returns.

The coefficient shrinkage also produces a dimension reduction effect, because the higher order coefficients are shrunk closer to zero.




```
> # Objective function for the in-sample AR coefficients
> objfun <- function(coeff, respv, predm, lambdaf) {
+   fcasts <- predm %*% coeff
+   10000*sum((respv - fcasts)^2) + sum(lambdaf*coeff^2)
+ }  # end objfun
```

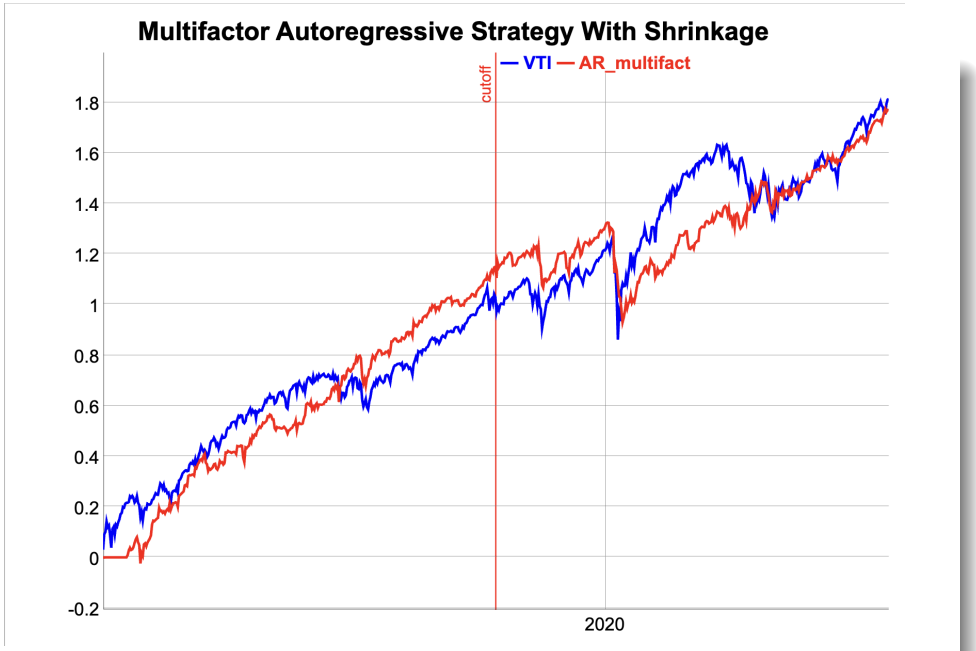
```
> # Perform optimization using the quasi-Newton method
> ncoeff <- NROW(coeff)
> optiml <- optim(par=numeric(ncoeff),
+           fn=objfun,
+           respv=respv[insample, ],
+           predm=predm[insample, ],
+           lambdaf=2.0,
+           method="L-BFGS-B",
+           upper=rep(10000, ncoeff),
+           lower=rep(-10000, ncoeff))
> # Extract the AR coefficients
> coeff <- optiml$par
> names(coeff) <- coeffn
> barplot(coeff ~ coeffn, xlab="", ylab="t-value", col="grey",
+   main="AR Coefficients With Shrinkage")
```

Kitchen Sink Strategy With Shrinkage

The out-of-sample performance of the multifactor autoregressive strategy can be improved by applying *shrinkage* to the AR coefficients.

The value of the *shrinkage factor* λ can be chosen to maximize the out-of-sample performance.

```
> # Calculate the forecasts of VTI (fitted values)
> fcasts <- predm %*% coeff
> fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.8)[, 2])
> fcastv[1:100] <- 1
> fcasts <- fcasts/fcastv
> # Calculate the autoregressive strategy PnLs
> pnls <- respv*fcasts
> pnls <- pnls*sd(respv[respv<0])/sd(pnls[pnls<0])
```




```
> # Calculate the in-sample and out-of-sample Sharpe and Sortino rat
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "AR_multifact")
> sqrt(252)*sapply(wealthv[insample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> sqrt(252)*sapply(wealthv[outsample, ], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the autoregressive strategies
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(wealthv))
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Multifactor Autoregressive Strategy With Shrinkage") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="cutoff", strokePattern="solid", co
+   dyLegend(show="always", width=300)
```

Brownian Motion

Brownian motion B_T is a stochastic process, with the increments $\mathrm{d}B_t$ which are independent and normally distributed, with mean zero and variance equal to $\mathrm{d}t$.

$$\mathrm{d}B_t = \xi_t \sqrt{\mathrm{d}t}$$

Where the ξ_t are random and independent *innovations* following the standard normal distribution $\phi(0,1)$, with the expected values: $\mathbb{E}[\xi_t] = 0$, $\mathbb{E}[\xi_t^2] = 1$, and $\mathbb{E}[\xi_{t1}\xi_{t2}] = 0$.

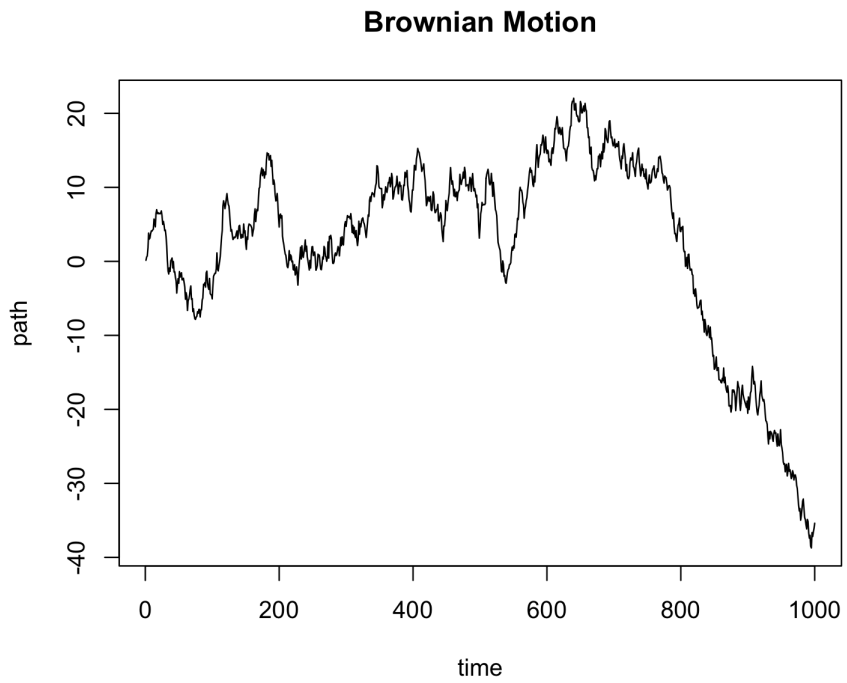
The Brownian motion path B_T is equal to the sum of its increments $\mathrm{d}B_t$:

$$B_T = \sum_{i=1}^{n} \mathrm{d}B_t$$

Where the number of time increments n is equal to the total time of evolution T divided by the increment size $\mathrm{d}t$: $n = T/\mathrm{d}t$.

The variance of Brownian motion is equal to the time of its evolution T:

$$\sigma^2 = \mathbb{E}[(\sum_{i=1}^{n} \xi_t \sqrt{\mathrm{d}t})^2] = \sum_{i=1}^{n} \mathbb{E}[\xi_t^2]\mathrm{d}t = T$$




```
> # Simulate a Brownian motion path
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> pathv <- cumsum(rnorm(nrows))
> plot(pathv, type="l", xlab="time", ylab="path",
+      main="Brownian Motion")
```

The Maximum Value of Brownian Motion

The distribution of the maximum value m of a Brownian motion path B_t can be calculated using the *reflection principle*.

The *reflection principle* states that the mirror image (reflection) of a Brownian motion path has the same probability as the original path.

The probability that the Brownian motion path B_t is at some point greater than some value m is the sum of the joint probability, that after the Brownian motion reaches the value m, it ends up greater than m, plus the joint probability that it ends up less than m:

$$p(B_t > m) = p((B_t > m) \& (B_T > m)) + p((B_t > m) \& (B_T < m))$$

By the *reflection principle*, both probabilities are equal, and also $p((B_t > m) \& (B_T > m)) = p(B_T > m)$, so that:

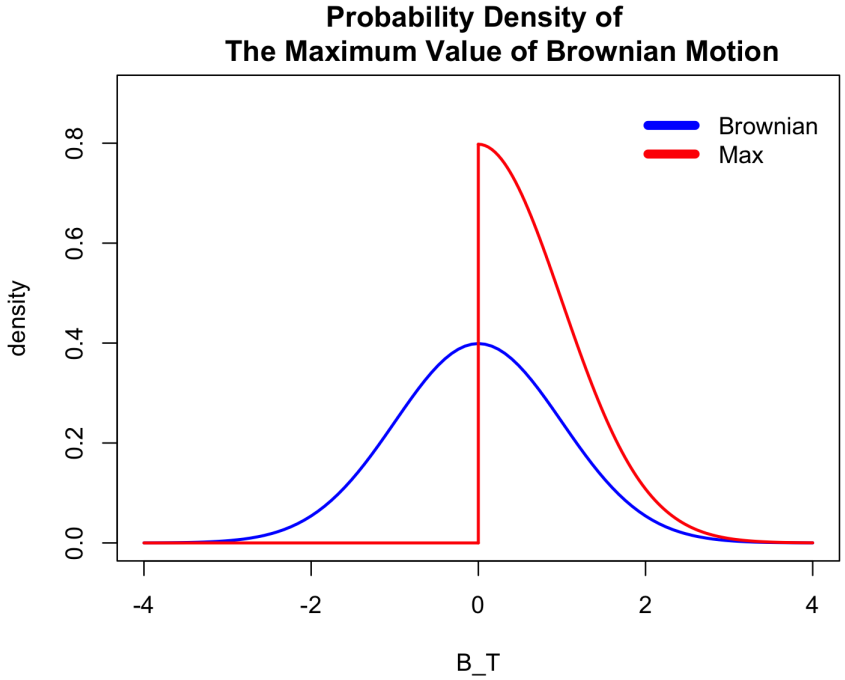
$$p(B_t > m) = 2p(B_T > m)$$

And the probability density of the maximum value m is equal to:

$$\varphi(m) = \sqrt{\frac{2}{\pi T}} e^{-\frac{m^2}{2T}}$$

The average value of the maximum value is equal to:

$$\bar{m} = \sqrt{\frac{2T}{\pi}}$$




```
> # Plot the density of Brownian Motion
> curve(expr=dnorm(x), xlim=c(-4, 4), ylim=c(0, 0.9),
+   xlab="B_T", ylab="density", lwd=2, col="blue")
> # Plot the density of the maximum of Brownian Motion
> curve(expr=2*dnorm(x), xlim=c(0, 4), xlab="", ylab="",
+   lwd=2, col="red", add=TRUE)
> lines(x=c(0, 0), y=c(0, sqrt(2/pi)), lwd=2, col="red")
> lines(x=c(-4, 0), y=c(0, 0), lwd=2, col="red")
> title(main="Probability Density of
+ The Maximum Value of Brownian Motion", line=0.5)
> legend("topright", inset=0.0, bty="n", y.intersp=0.4,
+  title=NULL, c("Brownian", "Max"), lwd=6,
+  col=c("blue", "red"))
```

The Range of Brownian Motion

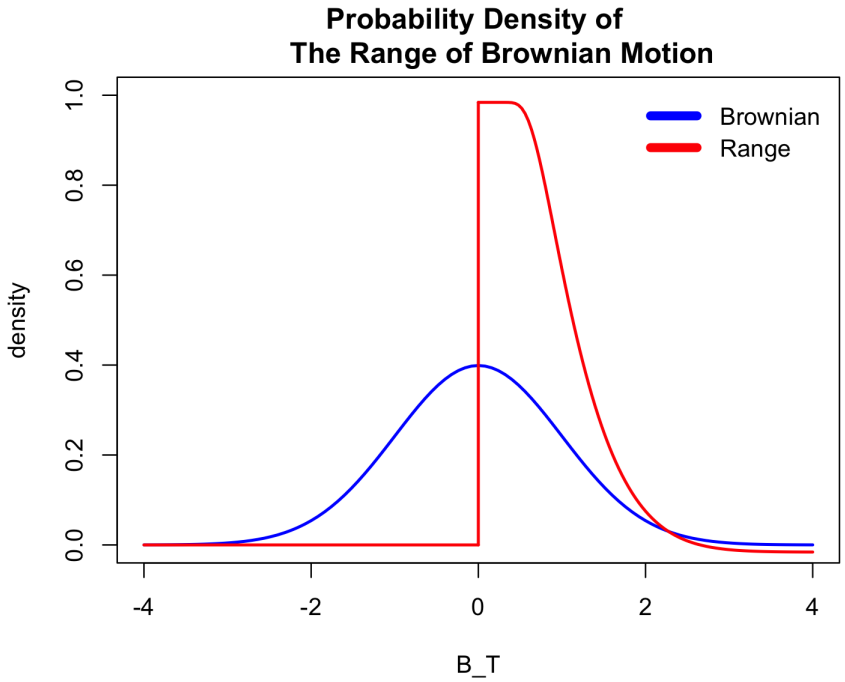
The range of a Brownian motion path B_t is equal to the difference between its maximum value minus its minimum value. The range is also called the drawdown.

The probability density of the range r is equal to the infinite series:

$$p(r) = 2 \sum_{n=1}^{\infty} \frac{\sin{(n-0.5)\pi}}{(n-0.5)\pi} (1 - e^{-\frac{(n-0.5)^2 \pi^2 T}{2r^2}})$$

The average value of the range is equal to:

$$\bar{r} = \sqrt{\frac{\pi T}{2}}$$




```
> # Series element
> fun1 <- function(n, r) { 2*sin((n-0.5)*pi)/((n-0.5)*pi) *
+   (1-exp(-((n-0.5)^2)*pi^2/2/r^2)) }
> # fun2 <- function(x) { sum(sapply(1:10, function(n) fun1(n, x))) }
> # fun2 <- function(x) { fun1(1, x) + fun1(2, x) + fun1(3, x) + fu
> # Sum of fun1
> fun2 <- function(x) {
+   valf <- 0
+   for (n in 1:20) { valf <- valf + fun1(n, x) }
+   return(valf)
+   } # end fun2
> # Theoretical average value of the range
> fun2(2)
> # Average value of the range from integration (not quite close)
> integrate(fun2, lower=0.01, upper=4)
```

```
> # Plot the density of Brownian Motion
> curve(expr=dnorm(x), xlim=c(-4, 4), ylim=c(0, 1.0),
+   xlab="B_T", ylab="density", lwd=2, col="blue")
> # Plot the density of the range of Brownian Motion
> curve(expr=fun2(x), xlim=c(0, 4), xlab="", ylab="",
+   lwd=2, col="red", add=TRUE)
> lines(x=c(0, 0), y=c(0, fun2(0.01)), lwd=2, col="red")
> lines(x=c(-4, 0), y=c(0, 0), lwd=2, col="red")
> title(main="Probability Density of
+ The Range of Brownian Motion", line=0.5)
> legend("topright", inset=0.0, bty="n", y.intersp=0.7,
+  title=NULL, c("Brownian", "Range"), lwd=6,
+  col=c("blue", "red"))
```

Geometric Brownian Motion

If the percentage asset returns $r_t \mathrm{d}t = \mathrm{d} \log p_t$ follow *Brownian motion*:

$$r_t \mathrm{d}t = \mathrm{d} \log p_t = (\mu - \frac{\sigma^2}{2})\mathrm{d}t + \sigma \, \mathrm{d}B_t$$

Then asset prices p_t follow *Geometric Brownian motion* (GBM):

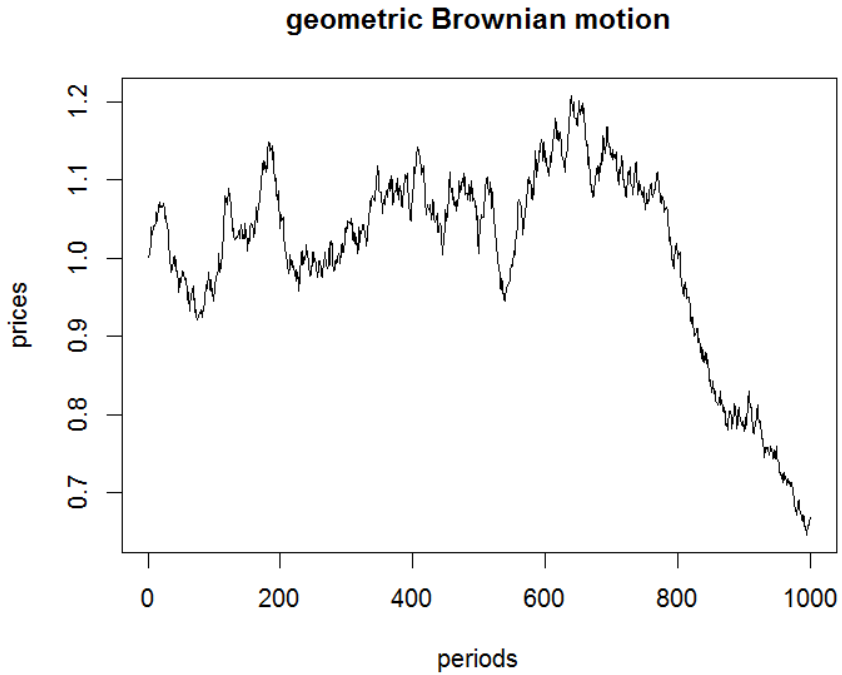
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}B_t$$

Where σ is the volatility of asset returns, and B_t is a *Brownian Motion*, with $\mathrm{d}B_t$ following the normal distribution $\phi(0, \sqrt{\mathrm{d}t})$, with the volatility $\sqrt{\mathrm{d}t}$, equal to the square root of the time increment $\mathrm{d}t$.

The solution of *Geometric Brownian motion* is equal to:

$$p_t = p_0 \exp[(\mu - \frac{\sigma^2}{2})t + \sigma \, B_t]$$

The convexity correction: $-\frac{\sigma^2}{2}$ ensures that the growth rate of prices is equal to μ, (according to Ito's lemma).




```
> # Define the daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 1000
> # Simulate geometric Brownian motion
> retp <- sigmav*rnorm(nrows) + drift - sigmav^2/2
> pricev <- exp(cumsum(retp))
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Geometric Brownian Motion")
```

The *Log-normal* Probability Distribution

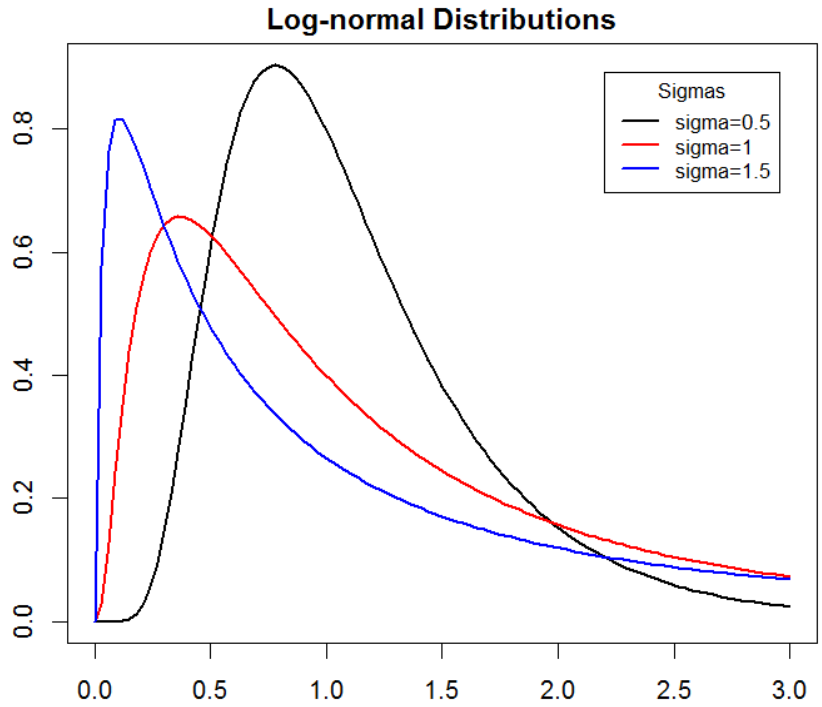
If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

$$\log \phi(y, \mu, \sigma) = \frac{\exp(-(\log y - \mu)^2/2\sigma^2)}{y\sigma\sqrt{2\pi}}$$

With mean equal to: $\bar{y} = \mathbb{E}[y] = e^{(\mu+\sigma^2/2)}$, and median equal to: $\tilde{y} = e^{\mu}$

With variance equal to: $\sigma_y^2 = (e^{\sigma^2} - 1)e^{(2\mu+\sigma^2)}$, and skewness (third moment) equal to: $\varsigma = \mathbb{E}[(y - \mathbb{E}[y])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$

```
> # Standard deviations of log-normal distribution
> sigmavs <- c(0.5, 1, 1.5)
> # Create plot colors
> colorv <- c("black", "red", "blue")
> # Plot all curves
> for (indeks in 1:NROW(sigmavs)) {
+   curve(expr=dlnorm(x, sdlog=sigmavs[indeks]),
+   type="l", lwd=2, xlim=c(0, 3),
+   xlab="", ylab="", col=colorv[indeks],
+   add=as.logical(indeks-1))
+ }  # end for
```




```
> # Add title and legend
> title(main="Log-normal Distributions", line=0.5)
> legend("topright", inset=0.05, title="Sigmas",
+  paste("sigma", sigmavs, sep="="),
+  cex=0.8, lwd=2, lty=rep(1, NROW(sigmavs)),
+  col=colorv)
```

The Standard Deviation of *Log-normal* Prices

If percentage asset returns are *normally* distributed and follow *Brownian motion*, then asset prices follow *Geometric Brownian motion*, and they are *Log-normally* distributed at every point in time.

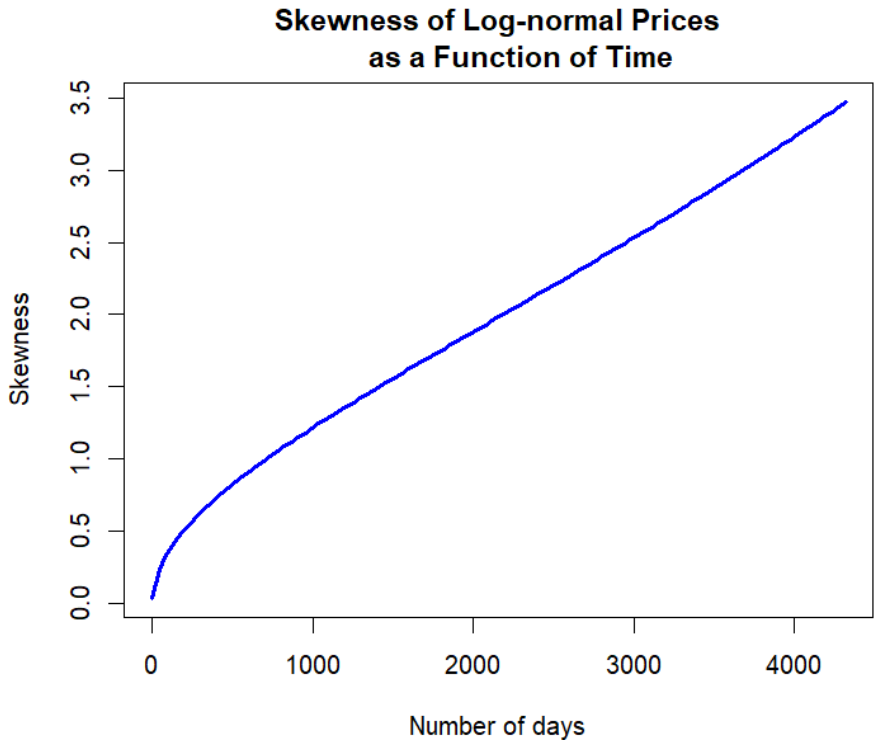
The standard deviation of *log-normal* prices is equal to the return volatility σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

The *Log-normal* distribution has a strong positive skewness (third moment) equal to:

$$\varsigma = \mathbb{E}[(y - \mathbb{E}[y])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$$

For large standard deviation, the skewness increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

```
> # Return volatility of VTI etf
> sigmav <- sd(rutils::diffit(log(rutils::etfenv$VTI[, 4])))
> sigma2 <- sigmav^2
> nrows <- NROW(rutils::etfenv$VTI)
> # Standard deviation of log-normal prices
> sqrt(nrows)*sigmav
```




```
> # Skewness of log-normal prices
> calcskew <- function(t) {
+   expv <- exp(t*sigma2)
+   (expv + 2)*sqrt(expv - 1)
+ }  # end calcskew
> curve(expr=calcskew, xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Skewness", col="blue",
+ main="Skewness of Log-normal Prices
+ as a Function of Time")
```

The Mean and Median of *Log-normal* Prices

The mean of the *Log-normal* distribution: $\bar{y} = \mathbb{E}[y] = \exp(\mu + \sigma^2/2)$ is greater than its median, which is equal to: $\tilde{y} = \exp(\mu)$.

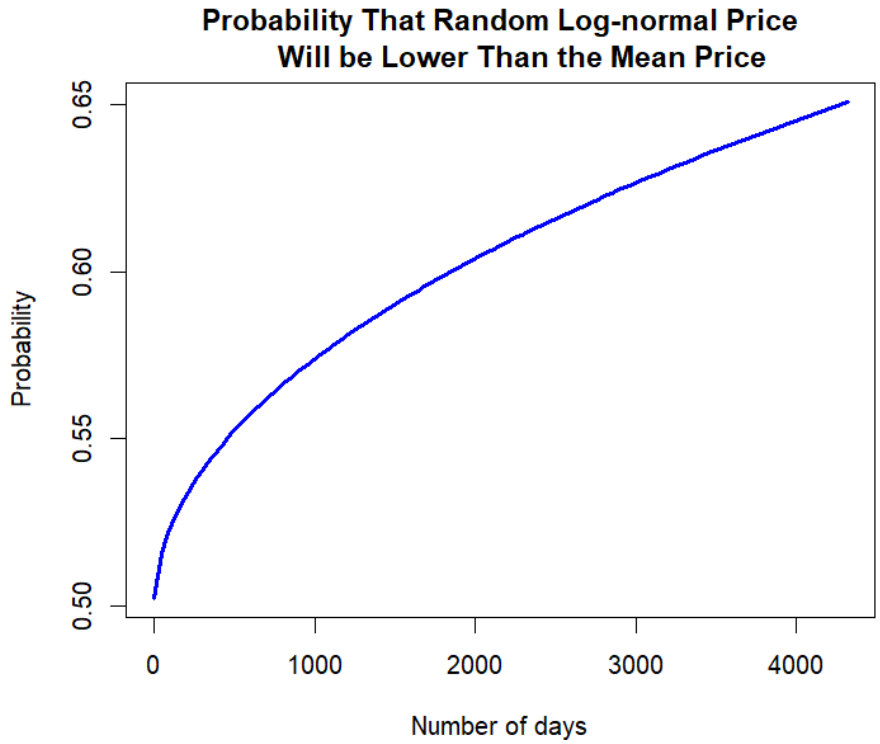
So if stock prices follow *Geometric Brownian motion* and are distributed *log-normally*, then a stock selected at random will have a high probability of havng a lower price than the mean expected price.

The cumulative *Log-normal* probability distribution is equal to $F(x) = \Phi(\frac{\log y - \mu}{\sigma})$, where $\Phi()$ is the cumulative standard normal distribution.

So the probability that the price of a randomly selected stock will be lower than the mean price is equal to $F(\bar{y}) = \Phi(\sigma/2)$.

Therefore an investor without skill, who selects stocks at random, has a high probability of underperforming the index.

Performing as well as the index requires *significant* investment skill, while outperforming the index requires *exceptional* investment skill.



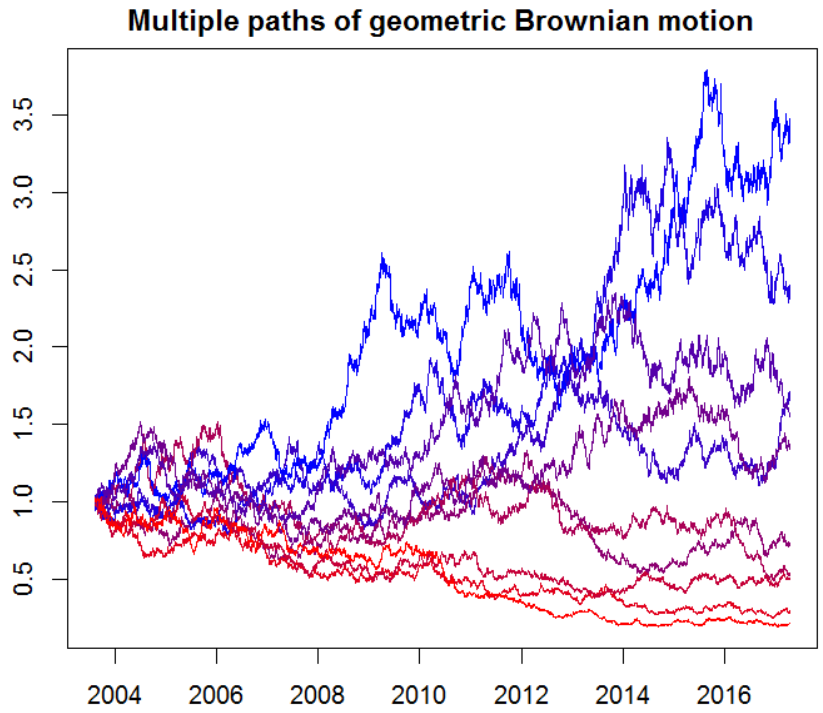

```
> # Probability that random log-normal price will be lower than the
> curve(expr=pnorm(sigmav*sqrt(x)/2),
+ xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Probability", col="blue",
+ main="Probability That Random Log-normal Price
+ Will be Lower Than the Mean Price")
```

Paths of Geometric Brownian Motion

The standard deviation of *log-normal* prices σ is equal to the volatility of returns σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

For large standard deviation, the skewness ς increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

```
> # Define the daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 5000
> npaths <- 10
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))
> # Create xts time series
> pricev <- xts(pricev, order.by=seq.Date(Sys.Date()-nrows+1, Sys.Da
> # Sort the columns according to largest terminal values
> pricev <- pricev[, order(pricev[nrows, ])]
> # Plot xts time series
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(pricev))
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(pricev, main="Multiple paths of geometric Brownian motion",
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
```

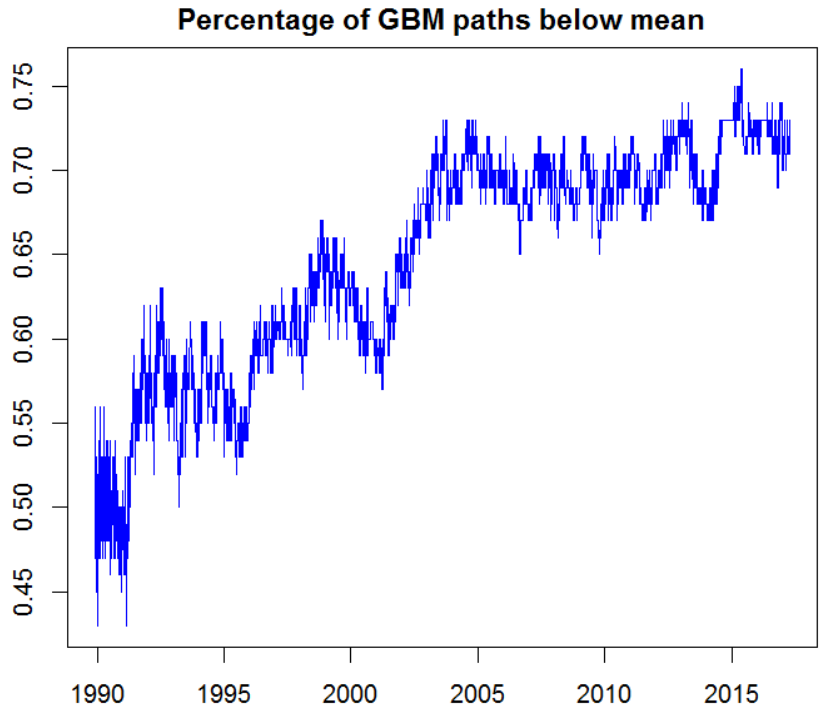


Distribution of Paths of Geometric Brownian Motion

Prices following *Geometric Brownian motion* have a large positive skewness, so that the expected value of prices is skewed by a few paths with very high prices, while the prices of the majority of paths are below their expected value.

For large standard deviation, the skewness ς increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

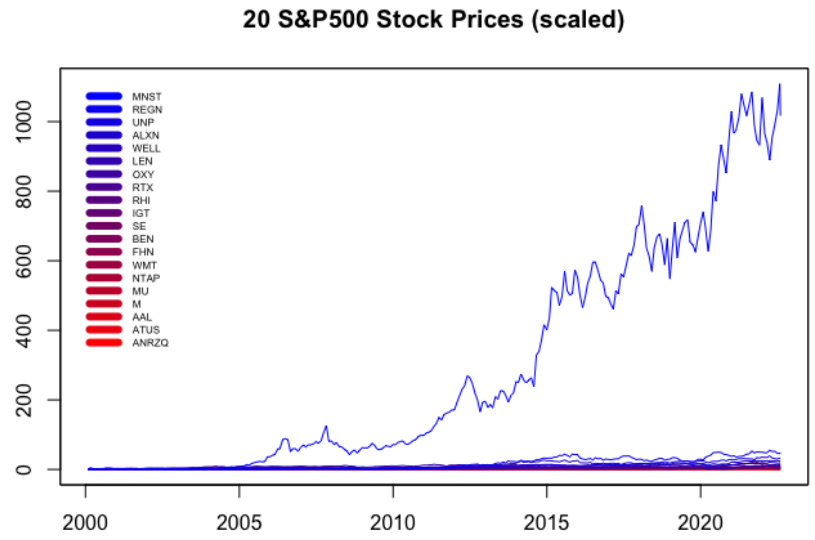
```
> # Define the daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 10000
> npaths <- 100
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))
> # Calculate fraction of paths below the expected value
> fractv <- rowSums(pricev < 1.0) / npaths
> # Create xts time series of percentage of paths below the expecte
> fractv <- xts(fractv, order.by=seq.Date(Sys.Date()-NROW(fractv)+1, Sys.Date(), by=1))
> # Plot xts time series of percentage of paths below the expected value
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(fractv, main="Percentage of GBM paths below mean",
+    xlab=NA, ylab=NA, col="blue")
```



Time Evolution of Stock Prices

Stock prices evolve over time similar to *Geometric Brownian motion*, and they also exhibit a very skewed distribution of prices.

```
> # Load S&P500 stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> ls(sp500env)
> # Extract the closing prices
> pricev <- eapply(sp500env, quantmod::Cl)
> # Flatten the prices into a single xts series
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward and backward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> pricev <- zoo::na.locf(pricev, fromLast=TRUE)
> sum(is.na(pricev))
> # Drop ".Close" from column names
> colnames(pricev)
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Select prices after the year 2000
> pricev <- pricev["2000/", ]
> # Scale the columns so that prices start at 1
> pricev <- lapply(pricev, function(x) x/as.numeric(x[1]))
> pricev <- rutils::do_call(cbind, pricev)
> # Sort the columns according to the final prices
> nrows <- NROW(pricev)
> ordern <- order(pricev[nrows, ])
> pricev <- pricev[, ordern]
> # Select 20 symbols
> symbolv <- colnames(pricev)
> symbolv <- symbolv[round(seq.int(from=1, to=NROW(symbolv), length.out=20))]
```



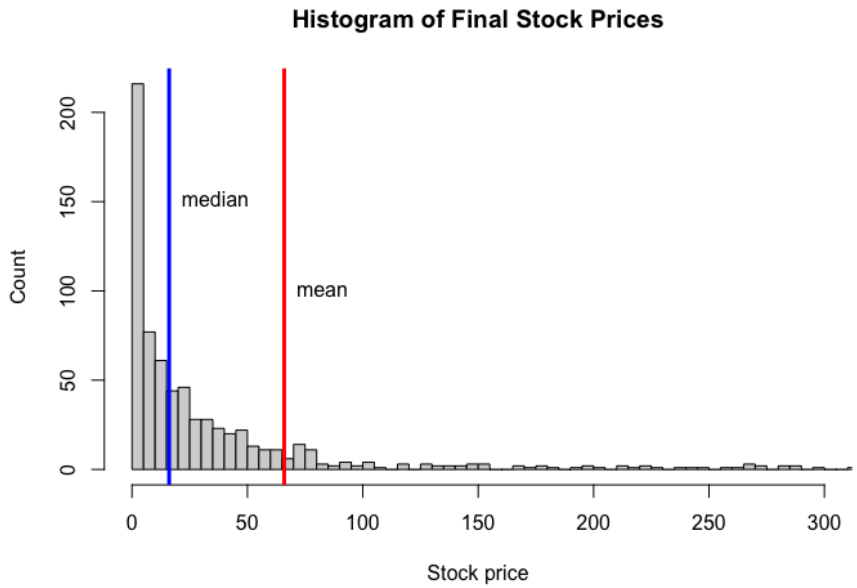

```
> # Plot xts time series of prices
> colorv <- colorRampPalette(c("red", "blue"))(NROW(symbolv))
> endd <- rutils::calc_endpoints(pricev, interval="weeks")
> plot.zoo(pricev[endd, symbolv], main="20 S&P500 Stock Prices (scal
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
> legend(x="topleft", inset=0.02, cex=0.5, bty="n", y.intersp=0.5,
+  legend=rev(symbolv), col=rev(colorv), lwd=6, lty=1)
```

Distribution of Final Stock Prices

The distribution of the final stock prices is extremely skewed, with over 80% of the *S&P500* constituent stocks from 1990 now below the average price of that portfolio.

The *mean* of the final stock prices is much greater than the *median*.

```
> # Calculate the final stock prices
> pricef <- drop(zoo::coredata(pricev[nrows, ]))
> # Calculate the mean and median stock prices
> max(pricef); min(pricef)
> which.max(pricef)
> which.min(pricef)
> mean(pricef)
> median(pricef)
> # Calculate the percentage of stock prices below the mean
> sum(pricef < mean(pricef))/NROW(pricef)
```




```
> # Plot a histogram of final stock prices
> hist(pricef, breaks=1e3, xlim=c(0, 300),
+      xlab="Stock price", ylab="Count",
+      main="Histogram of Final Stock Prices")
> # Plot a histogram of final stock prices
> abline(v=median(pricef), lwd=3, col="blue")
> text(x=median(pricef), y=150, lab="median", pos=4)
> abline(v=mean(pricef), lwd=3, col="red")
> text(x=mean(pricef), y=100, lab="mean", pos=4)
```

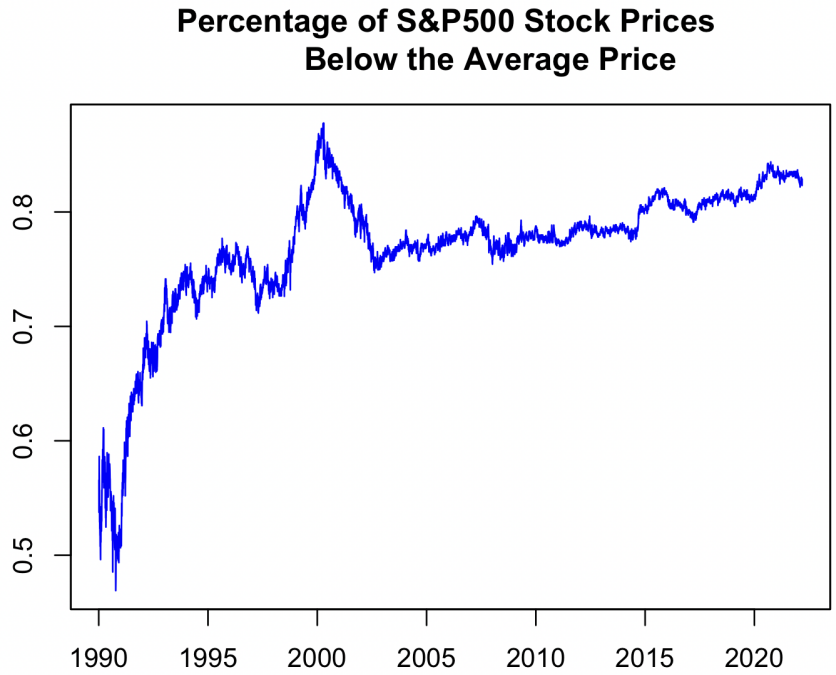
Distribution of Stock Prices Over Time

Usually, a small number of stocks in an index reach very high prices, while the prices of the majority of stocks remain below the index price (the average price of the index portfolio).

For example, the current prices of over 80% of the *S&P500* constituent stocks from 1990 are now below the average price of that portfolio.

Therefore an investor without skill, who selects stocks at random, has a high probability of underperforming the index, because they will most likely miss selecting the best performing stocks.

Performing as well as the index requires *significant* investment skill, while outperforming the index requires *exceptional* investment skill.




```
> # Calculate average of valid stock prices
> validp <- (pricev != 1)  # Valid stocks
> nstocks <- rowSums(validp)
> nstocks[1] <- NCOL(pricev)
> indeks <- rowSums(pricev*validp)/nstocks
> # Calculate fraction of stock prices below the average price
> fractv <- rowSums((pricev < indeks) & validp)/nstocks
> # Create xts time series of average stock prices
> indeks <- xts(indeks, order.by=zoo::index(pricev))
```

```
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # x11(width=6, height=4)
> # Plot xts time series of average stock prices
> plot.zoo(indeks, main="Average S&P500 Stock Prices (normalized fr
+    xlab=NA, ylab=NA, col="blue")
> # Create xts time series of percentage of stock prices below the a
> fractv <- xts(fractv, order.by=zoo::index(pricev))
> # Plot percentage of stock prices below the average price
> plot.zoo(fractv[-(1:2),],
+    main="Percentage of S&P500 Stock Prices
+    Below the Average Price",
+    xlab=NA, ylab=NA, col="blue")
```

Autocorrelation Function of Time Series

The *autocorrelation* of lag k of a time series of returns r_t is equal to:

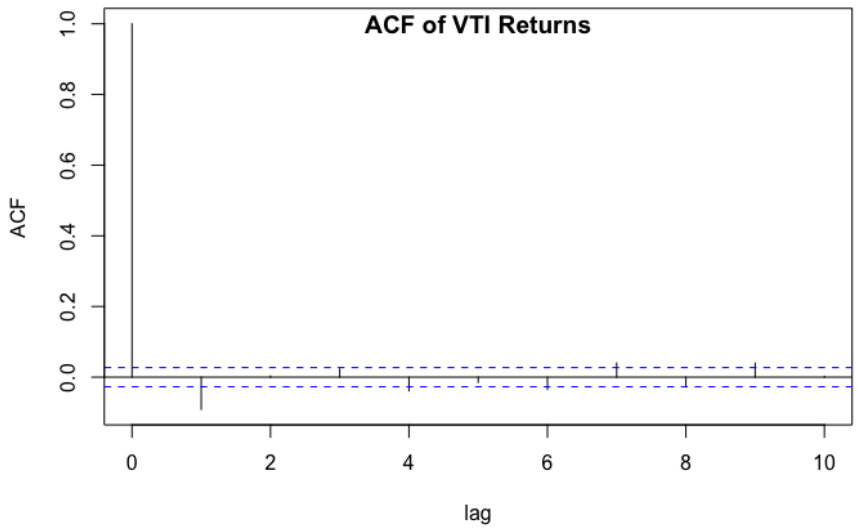
$$\rho_k = \frac{\sum_{t=k+1}^{n}(r_t - \bar{r})(r_{t-k} - \bar{r})}{(n-k)\,\sigma^2}$$

The *autocorrelation function* (*ACF*) is the vector of autocorrelation coefficients ρ_k.

The function `stats::acf()` calculates and plots the autocorrelation function of a time series.

The function `stats::acf()` has the drawback that it plots the lag zero autocorrelation (which is trivially equal to 1).

```
> # Open plot window under MS Windows
> x11(width=6, height=4)
> par(mar=c(3, 2, 1, 1), oma=c(1, 0, 0, 0))
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> retp <- drop(zoo::coredata(retp))
> # Plot autocorrelations of VTI returns using stats::acf()
> stats::acf(retp, lag=10, xlab="lag", main="")
> title(main="ACF of VTI Returns", line=-1)
> # Calculate two-tailed 95% confidence interval
> qnorm(0.975)/sqrt(NROW(retp))
```



The *VTI* time series of returns has small, but statistically significant negative autocorrelations.

The horizontal dashed lines are two-tailed confidence intervals of the autocorrelation estimator at 95% significance level: $\frac{\Phi^{-1}(0.975)}{\sqrt{n}}$.

But the visual inspection of the *ACF* plot alone is not enough to test whether autocorrelations are statistically significant or not.

Improved Autocorrelation Function

The function `acf()` has the drawback that it plots the lag zero autocorrelation (which is simply equal to 1).

Inspection of the data returned by `acf()` shows how to omit the lag zero autocorrelation.

The function `acf()` returns the *ACF* data invisibly, i.e. the return value can be assigned to a variable, but otherwise it isn't automatically printed to the console.

The function `rutils::plot_acf()` from package *rutils* is a wrapper for `acf()`, and it omits the lag zero autocorrelation.

```
> # Get the ACF data returned invisibly
> acfl <- acf(retp, plot=FALSE)
> summary(acfl)
> # Print the ACF data
> print(acfl)
> dim(acfl$acf)
> dim(acfl$lag)
> head(acfl$acf)
```

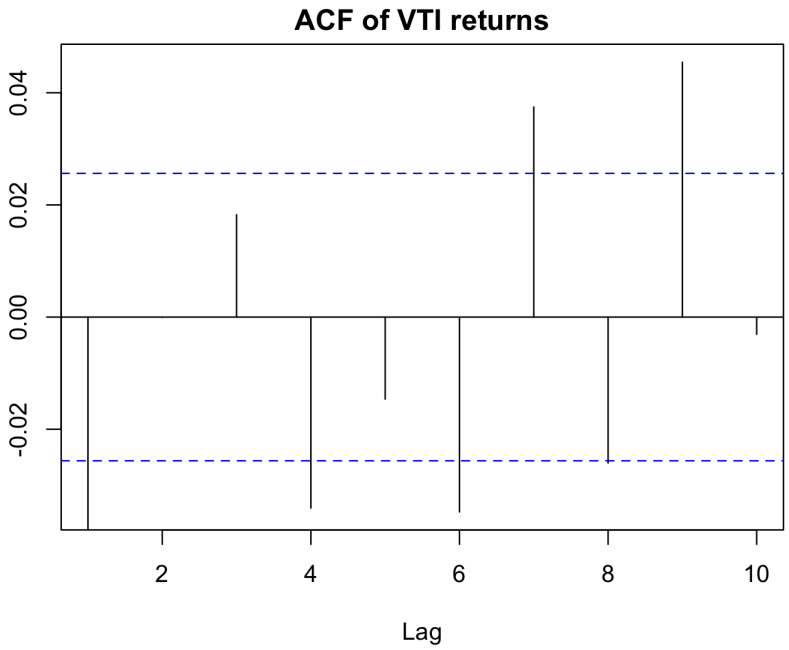
```
> plot_acf <- function(xtsv, lagg=10, plotobj=TRUE,
+                xlab="Lag", ylab="", main="", ...) {
+   # Calculate the ACF without a plot
+   acfl <- acf(x=xtsv, lag.max=lagg, plot=FALSE, ...)
+   # Remove first element of ACF data
+   acfl$acf <- array(data=acfl$acf[-1],
+     dim=c((dim(acfl$acf)[1]-1), 1, 1))
+   acfl$lag <- array(data=acfl$lag[-1],
+     dim=c((dim(acfl$lag)[1]-1), 1, 1))
+   # Plot ACF
+   if (plotobj) {
+     ci <- qnorm((1+0.95)/2)/sqrt(NROW(xtsv))
+     ylim <- c(min(-ci, range(acfl$acf[-1])),
+         max(ci, range(acfl$acf[-1])))
+     plot(acfl, xlab=xlab, ylab=ylab,
+    ylim=ylim, main="", ci=0)
+     title(main=main, line=0.5)
+     abline(h=c(-ci, ci), col="blue", lty=2)
+   }  # end if
+   # Return the ACF data invisibly
+   invisible(acfl)
+ }  # end plot_acf
```

Autocorrelations of Stock Returns

The *VTI* returns appear to have some small, yet significant negative autocorrelations at `lag=1`.

But the visual inspection of the *ACF* plot alone is not enough to test whether autocorrelations are statistically significant or not.

```
> # Autocorrelations of VTI returns
> rutils::plot_acf(retp, lag=10, main="ACF of VTI returns")
```



Ljung-Box Test for Autocorrelations of Time Series

The *Ljung-Box* test, tests if the autocorrelations of a time series are *statistically significant*.

The *null hypothesis* of the *Ljung-Box* test is that the autocorrelations are equal to zero.

The test statistic is:

$$Q = n(n+2)\sum_{k=1}^{maxlag}\frac{\hat{\rho}_k^2}{n-k}$$

Where `n` is the sample size, and the $\hat{\rho}_k$ are sample autocorrelations.

The *Ljung-Box* statistic follows the *chi-squared* distribution with *maxlag* degrees of freedom.

The *Ljung-Box* statistic is small for time series that have *statistically insignificant* autocorrelations.

The function `Box.test()` calculates the *Ljung-Box* test and returns the test statistic and its p-value.

```
> # Ljung-Box test for VTI returns
> # 'lag' is the number of autocorrelation coefficients
> Box.test(retp, lag=10, type="Ljung")
> # Ljung-Box test for random returns
> Box.test(rnorm(NROW(retp)), lag=10, type="Ljung")
> library(Ecdat)  # Load Ecdat
> macrodata <- as.zoo(Macrodat[, c("lhur", "fygm3")])
> colnames(macrodata) <- c("unemprate", "3mTbill")
> macrodiff <- na.omit(diff(macrodata))
> # Changes in 3 month T-bill rate are autocorrelated
> Box.test(macrodiff[, "3mTbill"], lag=10, type="Ljung")
> # Changes in unemployment rate are autocorrelated
> Box.test(macrodiff[, "unemprate"], lag=10, type="Ljung")
```

The *p*-value for *VTI* returns is small, and we conclude that the *null hypothesis* is `FALSE`, and that *VTI* returns do have some small autocorrelations.

The *p*-value for changes in econometric data is extremely small, and we conclude that the *null hypothesis* is `FALSE`, and that econometric data *are* autocorrelated.

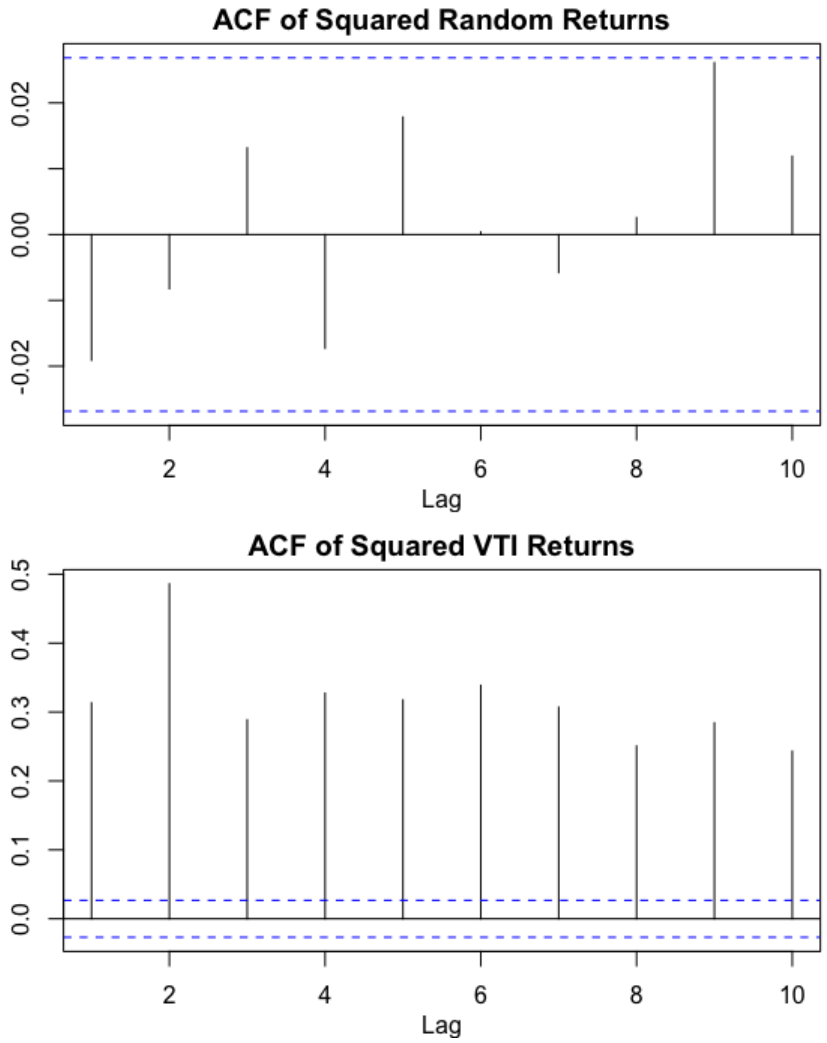
Autocorrelations of Squared VTI Returns

Squared random returns are not autocorrelated.

But squared *VTI* returns do have statistically significant autocorrelations.

The autocorrelations of squared asset returns are a very important feature.

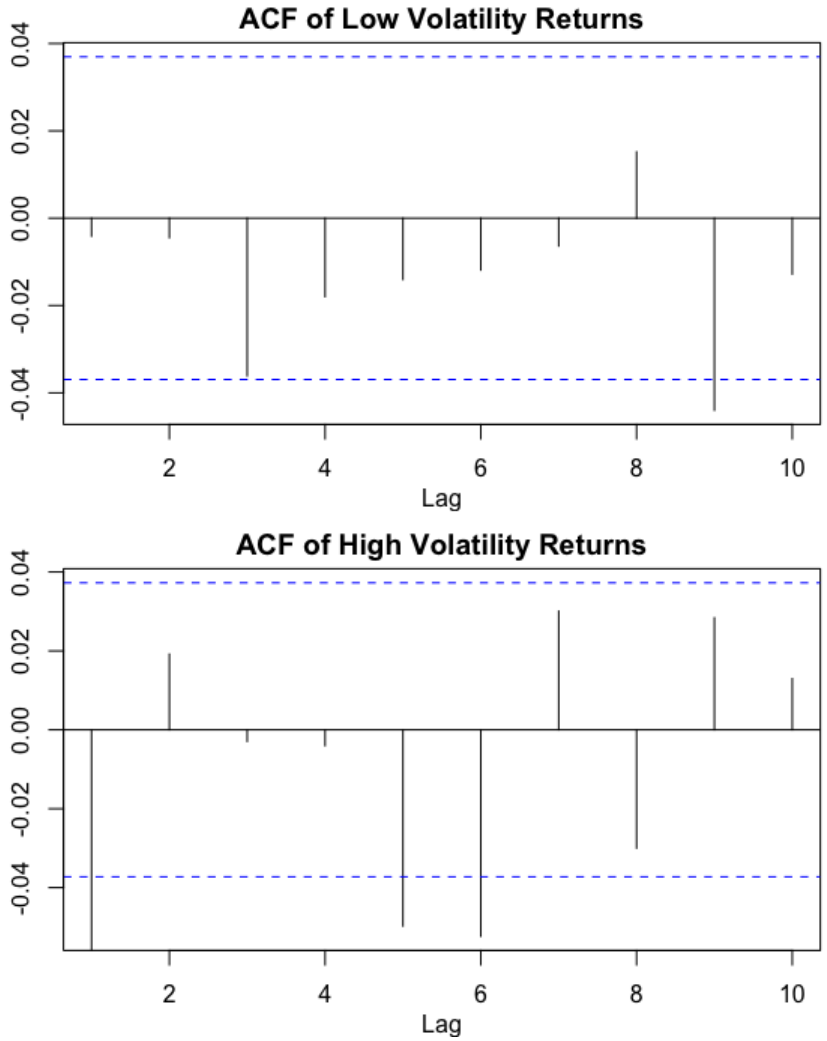
```
> # Open plot window under MS Windows
> x11(width=6, height=7)
> # Set two vertical plot panels
> par(mfrow=c(2,1))
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> # Plot ACF of squared random returns
> rutils::plot_acf(rnorm(NROW(retp))^2, lag=10,
+   main="ACF of Squared Random Returns")
> # Plot ACF of squared VTI returns
> rutils::plot_acf(retp^2, lag=10,
+   main="ACF of Squared VTI Returns")
> # Ljung-Box test for squared VTI returns
> Box.test(retp^2, lag=10, type="Ljung")
```



Autocorrelations in Intervals of Low and High Volatility

Stock returns have significant negative autocorrelations in time intervals with high volatility, but much less in time intervals with low volatility.

```
> # Calculate the monthly end points
> endd <- rutils::calc_endpoints(retp, interval="weeks")
> npts <- NROW(endd)
> # Calculate the monthly VTI volatilities and their median volatil
> stdev <- sapply(2:npts, function(endp) {
+   sd(retp[endd[endp-1]:endd[endp]])
+ })  # end sapply
> medianv <- median(stdev)
> # Calculate the stock returns of low volatility intervals
> retlow <- lapply(2:npts, function(endp) {
+   if (stdev[endp-1] <= medianv)
+     retp[endd[endp-1]:endd[endp]]
+ })  # end lapply
> retlow <- rutils::do_call(c, retlow)
> # Calculate the stock returns of high volatility intervals
> rethigh <- lapply(2:npts, function(endp) {
+   if (stdev[endp-1] > medianv)
+     retp[endd[endp-1]:endd[endp]]
+ })  # end lapply
> rethigh <- rutils::do_call(c, rethigh)
> # Plot ACF of low volatility returns
> rutils::plot_acf(retlow, lag=10,
+  main="ACF of Low Volatility Returns")
> Box.test(retlow, lag=10, type="Ljung")
> # Plot ACF of high volatility returns
> rutils::plot_acf(rethigh, lag=10,
+  main="ACF of High Volatility Returns")
> Box.test(rethigh, lag=10, type="Ljung")
```

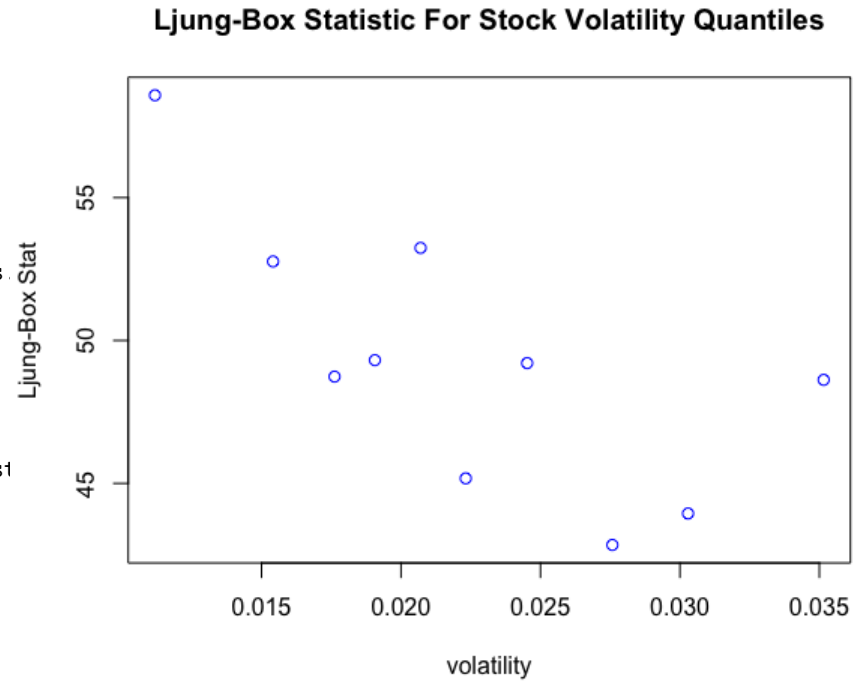


Autocorrelations of Low and High Volatility Stocks

Low volatility stocks have more significant negative autocorrelations than high volatility stocks.

But even the lowest volatility quantile of stocks has less significant negative autocorrelations than *VTI* does.

```
> # Load daily S&P500 stock returns
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns
> # Calculate the stock volatilities and Ljung-Box test statistics
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1
> statm <- mclapply(returns, function(retp) {
+   retp <- na.omit(retp)
+   c(stdev=sd(retp), lbstat=Box.test(retp, lag=10, type="Ljung")$st
+ }, mc.cores=ncores)  # end mclapply
> statm <- do.call(rbind, statm)
> colnames(statm)[2] <- "lbstat"
> # Calculate the median volatility
> stdev <- statm[, "stdev"]
> lbstat <- statm[, "lbstat"]
> stdevm <- median(stdev)
> # Calculate the Ljung-Box statistics for stock volatility quantiles
> quants <- quantile(stdev, c(0.001, seq(0.1, 0.9, 0.1), 0.999))
> lbstatq <- sapply(2:NROW(quants), function(it) {
+   mean(lbstat[(stdev > quants[it-1]) & (stdev < quants[it])])
+ }) # end sapply
> # Calculate the Ljung-Box statistics for low and high volatility
> lowvol <- (stdev < stdevm)
> mean(statm[lowvol, "lbstat"])
> mean(statm[!lowvol, "lbstat"])
> # Compare the Ljung-Box statistics for lowest volatility stocks with VTI
> lbstatq[1]
> Box.test(na.omit(rutils::etfenv$returns$VTI), lag=10, type="Ljung")$statistic
```



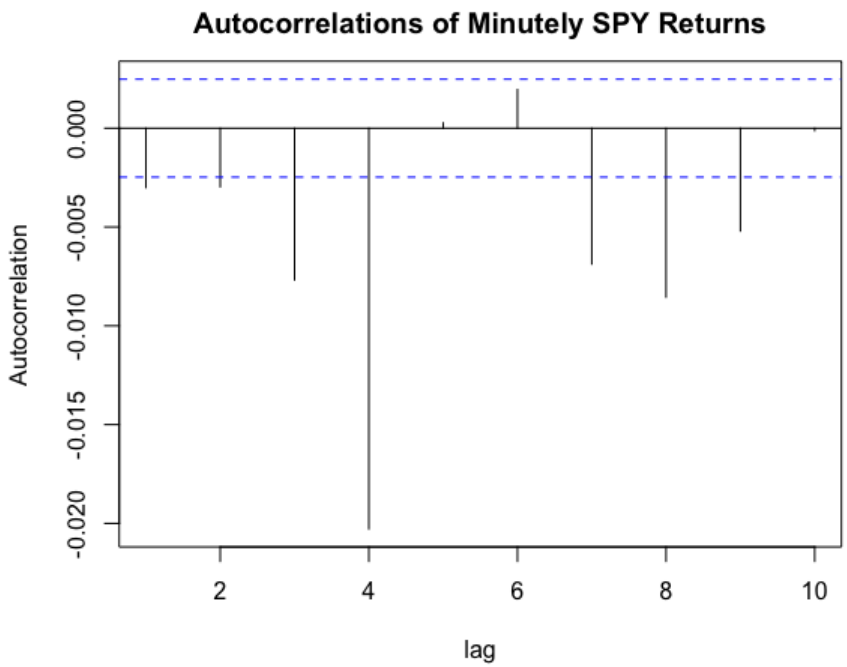

```
> # Plot Ljung-Box test statistic for volatility quantiles
> plot(x=quants[-NROW(quants)], y=lbstatq, lwd=1, col="blue",
+      # xlim=c(0.01, 0.05), ylim=c(0, 100),
+      xlab="volatility", ylab="Ljung-Box Stat",
+      main="Ljung-Box Statistic For Stock Volatility Quantiles")
```

Autocorrelations of High Frequency Returns

The package *HighFreq* contains three time series of intraday 1-minute *OHLC* price bars, called `SPY`, `TLT`, and `VXX`, for the *SPY*, *TLT*, and *VXX* ETFs.

Minutely *SPY* returns have statistically significant negative autocorrelations.

```
> # Calculate SPY log prices and percentage returns
> ohlc <- HighFreq::SPY
> ohlc[, 1:4] <- log(ohlc[, 1:4])
> nrows <- NROW(ohlc)
> closep <- quantmod::Cl(ohlc)
> retp <- rutils::diffit(closep)
> colnames(retp) <- "SPY"
> # Open plot window under MS Windows
> x11(width=6, height=4)
> # Open plot window on Mac
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # Plot the autocorrelations of minutely SPY returns
> acfl <- rutils::plot_acf(as.numeric(retp), lag=10,
+       xlab="lag", ylab="Autocorrelation", main="")
> title("Autocorrelations of Minutely SPY Returns", line=1)
> # Calculate the sum of autocorrelations
> sum(acfl$acf)
```



Autocorrelations as Function of Aggregation Interval

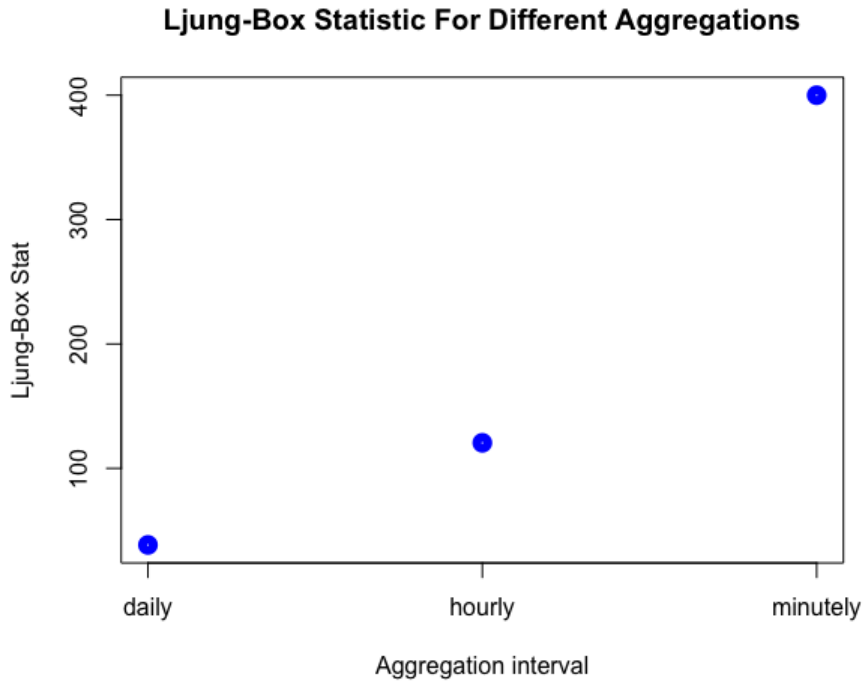
For *minutely SPY* returns, the *Ljung-Box* statistic is large and its *p*-value is very small, so we can conclude that *minutely SPY* returns have statistically significant autocorrelations.

The level of the autocorrelations depends on the sampling frequency, with higher frequency returns having more significant negative autocorrelations.

SPY returns aggregated to longer time intervals are less autocorrelated.

As the returns are aggregated to a lower periodicity, they become less autocorrelated, with daily returns having almost insignificant autocorrelations.

The function `rutils::to_period()` aggregates an *OHLC* time series to a lower periodicity.




```
> # Ljung-Box test for minutely SPY returns
> Box.test(retp, lag=10, type="Ljung")
> # Calculate hourly SPY percentage returns
> closeh <- quantmod::Cl(xts::to.period(x=ohlc, period="hours"))
> retsh <- rutils::diffit(closeh)
> # Ljung-Box test for hourly SPY returns
> Box.test(retsh, lag=10, type="Ljung")
> # Calculate daily SPY percentage returns
> closed <- quantmod::Cl(xts::to.period(x=ohlc, period="days"))
> retd <- rutils::diffit(closed)
> # Ljung-Box test for daily SPY returns
> Box.test(retd, lag=10, type="Ljung")
```

```
> # Ljung-Box test statistics for aggregated SPY returns
> lbstat <- sapply(list(daily=retd, hourly=retsh, minutely=retp),
+   function(rets) {
+     Box.test(rets, lag=10, type="Ljung")$statistic
+ })  # end sapply
> # Plot Ljung-Box test statistic for different aggregation interval
> plot(lbstat, lwd=6, col="blue", xaxt="n",
+      xlab="Aggregation interval", ylab="Ljung-Box Stat",
+      main="Ljung-Box Statistic For Different Aggregations")
> # Add X-axis with labels
> axis(side=1, at=(1:3), labels=c("daily", "hourly", "minutely"))
```

Volatility as a Function of the Aggregation Interval

The estimated volatility σ scales as the *power* of the length of the aggregation time interval Δt:

$$\frac{\sigma_t}{\sigma} = \Delta t^H$$

Where `H` is the *Hurst* exponent, σ is the return volatility, and σ_t is the volatility of the aggregated returns.

If returns follow *Brownian motion* then the volatility scales as the *square root* of the length of the aggregation interval (`H = 0.5`).

If returns are *mean reverting* then the volatility scales slower than the *square root* (`H < 0.5`).

If returns are *trending* then the volatility scales faster than the *square root* (`H > 0.5`).

The length of the daily time interval is often approximated to be equal to `390 = 6.5*60` minutes, since the exchange trading session is equal to `6.5` hours, and daily volatility is dominated by the trading session.

The daily volatility is exaggerated by price jumps over the weekends and holidays, so it should be scaled.

The minutely volatility is exaggerated by overnight price jumps.

```
> # Daily SPY volatility from daily returns
> sd(retd)
> # Minutely SPY volatility scaled to daily interval
> sqrt(6.5*60)*sd(retp)
> # Minutely SPY returns without overnight price jumps (unit per se
> retp <- retp/rutils::diffit(xts::.index(retp))
> retp[1] <- 0
> # Daily SPY volatility from minutely returns
> sqrt(6.5*60)*60*sd(retp)
> # Daily SPY returns without weekend and holiday price jumps (unit
> retd <- retd/rutils::diffit(xts::.index(retd))
> retd[1] <- 0
> # Daily SPY volatility without weekend and holiday price jumps
> 24*60*60*sd(retd)
```

The package *HighFreq* contains three time series of intraday 1-minute *OHLC* price bars, called `SPY`, `TLT`, and `VXX`, for the *SPY*, *TLT*, and *VXX* ETFs.

The function `rutils::to_period()` aggregates an *OHLC* time series to a lower periodicity.

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.

Hurst Exponent From Volatility

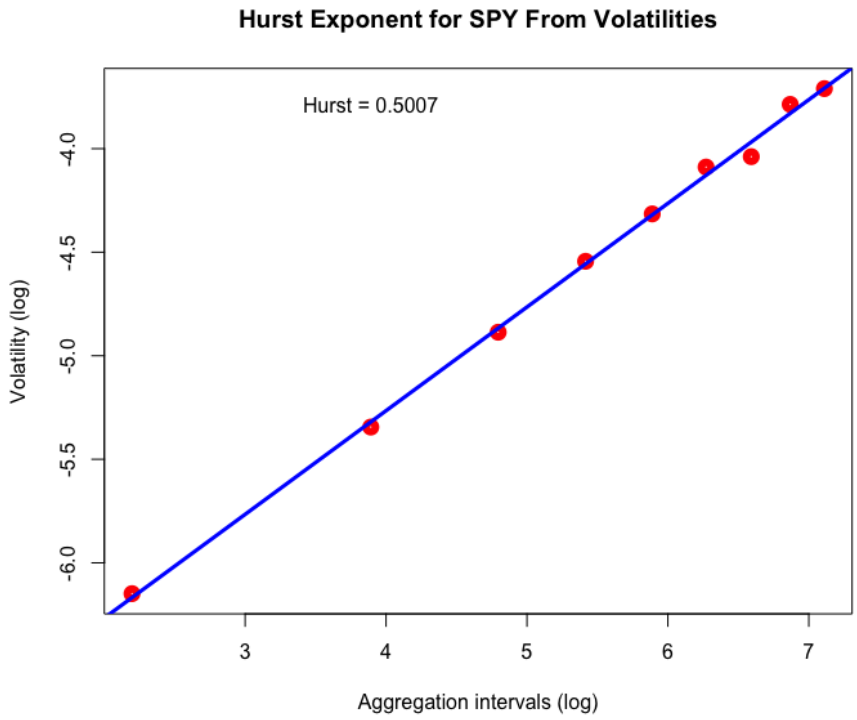
For a single aggregation interval, the *Hurst exponent* `H` is equal to:

$$H = \frac{\log \sigma_t - \log \sigma}{\log \Delta t}$$

For a vector of aggregation intervals Δt, the *Hurst exponent* `H` is equal to the regression slope between the logarithms of the aggregated volatilities σ_t versus the logarithms of the aggregation intervals Δt:

$$H = \frac{\text{cov}(\log \sigma_t, \log \Delta t)}{\text{var}(\log \Delta t)}$$

```
> # Calculate volatilities for vector of aggregation intervals
> aggv <- seq.int(from=3, to=35, length.out=9)^2
> volv <- sapply(aggv, function(agg) {
+   naggs <- nrows %/% agg
+   endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
+   # endd <- rutils::calc_endpoints(closep, interval=agg)
+   sd(rutils::diffit(closep[endd]))
+ })  # end sapply
> # Calculate the Hurst from single data point
> volog <- log(volv)
> agglog <- log(aggv)
> (last(volog) - first(volog))/(last(agglog) - first(agglog))
> # Calculate the Hurst from regression slope using formula
> hurstexp <- cov(volog, agglog)/var(agglog)
> # Or using function lm()
> model <- lm(volog ~ agglog)
> coef(model)[2]
```




```
> # Plot the volatilities
> x11(width=6, height=4)
> par(mar=c(4, 4, 2, 1), oma=c(1, 1, 1, 1))
> plot(volog ~ agglog, lwd=6, col="red",
+      xlab="Aggregation intervals (log)", ylab="Volatility (log)",
+      main="Hurst Exponent for SPY From Volatilities")
> abline(model, lwd=3, col="blue")
> text(agglog[2], volog[NROW(volog)-1],
+      paste0("Hurst = ", round(hurstexp, 4)))
```

Rescaled Range Analysis

The range $R_{\Delta t}$ of prices p_t over an interval Δt, is the difference between the highest attained price minus the lowest:

$$R_t = \max_{\Delta t}[p_\tau] - \min_{\Delta t}[p_\tau]$$

The *Rescaled Range* $RS_{\Delta t}$ is equal to the range $R_{\Delta t}$ divided by the standard deviation of the price differences σ_t: $RS_{\Delta t} = R_t/\sigma_t$.

The *Rescaled Range* $RS_{\Delta t}$ for a time series of prices is calculated by:

- Dividing the time series into non-overlapping intervals of length Δt,
- Calculating the *rescaled range* $RS_{\Delta t}$ for each interval,
- Calculating the average of the *rescaled ranges* $RS_{\Delta t}$ for all the intervals.

Rescaled Range Analysis (R/S) consists of calculating the average *rescaled range* $RS_{\Delta t}$ as a function of the length of the aggregation interval Δt.

```
> # Calculate cumulative SPY returns
> closep <- cumsum(retp)
> nrows <- NROW(closep)
> # Calculate the rescaled range
> agg <- 500
> naggs <- nrows %/% agg
> endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
> # Or
> # endd <- rutils::calc_endpoints(closep, interval=agg)
> rrange <- sapply(2:NROW(endd), function(np) {
+   indeks <- (endd[np-1]+1):endd[np]
+   diff(range(closep[indeks]))/sd(retp[indeks])
+ })  # end sapply
> mean(rrange)
> # Calculate the Hurst from single data point
> log(mean(rrange))/log(agg)
```

Hurst Exponent From Rescaled Range

The average *Rescaled Range* $RS_{\Delta t}$ is proportional to the length of the aggregation interval Δt raised to the power of the *Hurst exponent* `H`:

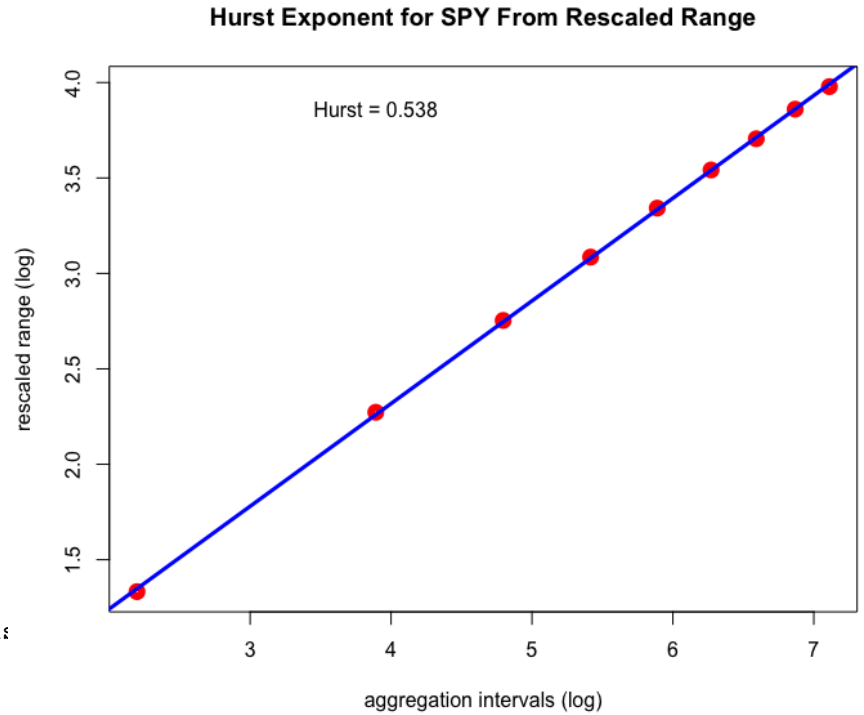
$$RS_{\Delta t} \propto \Delta t^{H}$$

So the *Hurst exponent* `H` is equal to:

$$H = \frac{\log RS_{\Delta t}}{\log \Delta t}$$

The Hurst exponents calculated from the *rescaled range* and the *volatility* are similar but not exactly equal because they use different methods to estimate price dispersion.

```
> # Calculate the rescaled range for vector of aggregation intervals
> rrange <- sapply(aggv, function(agg) {
+ # Calculate the end points
+   naggs <- nrows %/% agg
+   endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
+ # Calculate the rescaled ranges
+   rrange <- sapply(2:NROW(endd), function(np) {
+     indeks <- (endd[np-1]+1):endd[np]
+     diff(range(closep[indeks]))/sd(retp[indeks])
+   })  # end sapply
+   mean(na.omit(rrange))
+ })  # end sapply
> # Calculate the Hurst as regression slope using formula
> rangelog <- log(rrange)
> agglog <- log(aggv)
> hurstexp <- cov(rangelog, agglog)/var(agglog)
> # Or using function lm()
> model <- lm(rangelog ~ agglog)
> coef(model)[2]
```



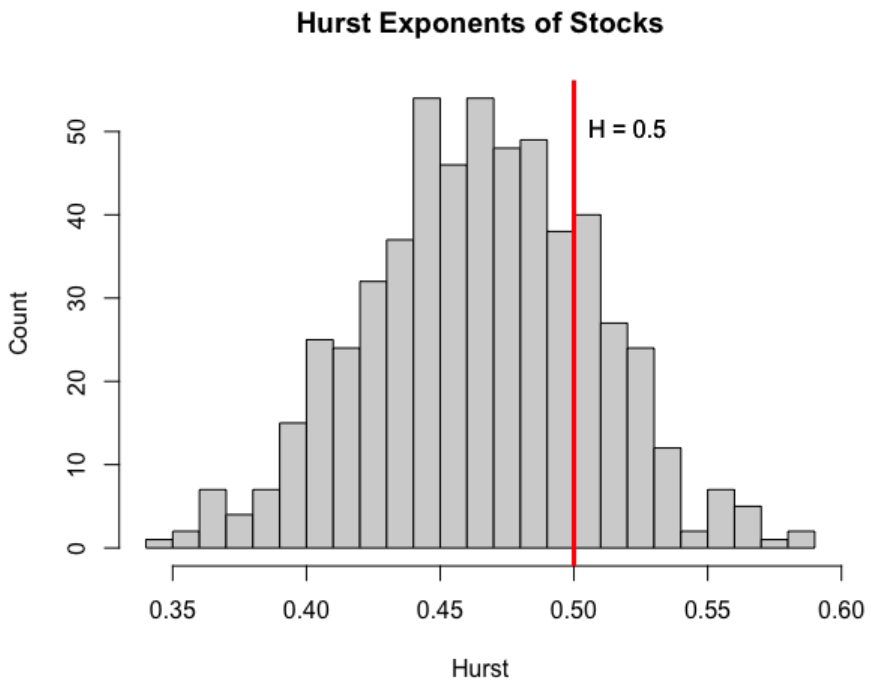

```
> plot(rangelog ~ agglog, lwd=6, col="red",
+      xlab="aggregation intervals (log)",
+      ylab="rescaled range (log)",
+      main="Hurst Exponent for SPY From Rescaled Range")
> abline(model, lwd=3, col="blue")
> text(agglog[2], rangelog[NROW(rangelog)-1],
+      paste0("Hurst = ", round(hurstexp, 4)))
```

Hurst Exponents of Stocks

The Hurst exponents of stocks are typically slightly less than 0.5, because their idiosyncratic risk components are mean-reverting.

The function `HighFreq::calc_hurst()` calculates the Hurst exponent in C++ using volatility ratios.

```
> # Load S&P500 constituent OHLC stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> class(sp500env$AAPL)
> head(sp500env$AAPL)
> # Calculate log stock prices after the year 2000
> pricev <- eapply(sp500env, function(ohlc) {
+   closep <- log(quantmod::Cl(ohlc)["2000/"])
+ # Ignore short lived and penny stocks (less than $1)
+   if ((NROW(closep) > 4000) & (last(closep) > 0))
+     return(closep)
+ })  # end eapply
> # Calculate the number of NULL prices
> sum(sapply(pricev, is.null))
> # Calculate the names of the stocks (remove NULL pricev)
> namev <- sapply(pricev, is.null)
> namev <- namev[!namev]
> namev <- names(namev)
> pricev <- pricev[namev]
> # Calculate the Hurst exponents of stocks
> aggv <- trunc(seq.int(from=3, to=10, length.out=5)^2)
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of stock with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of stock with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
```



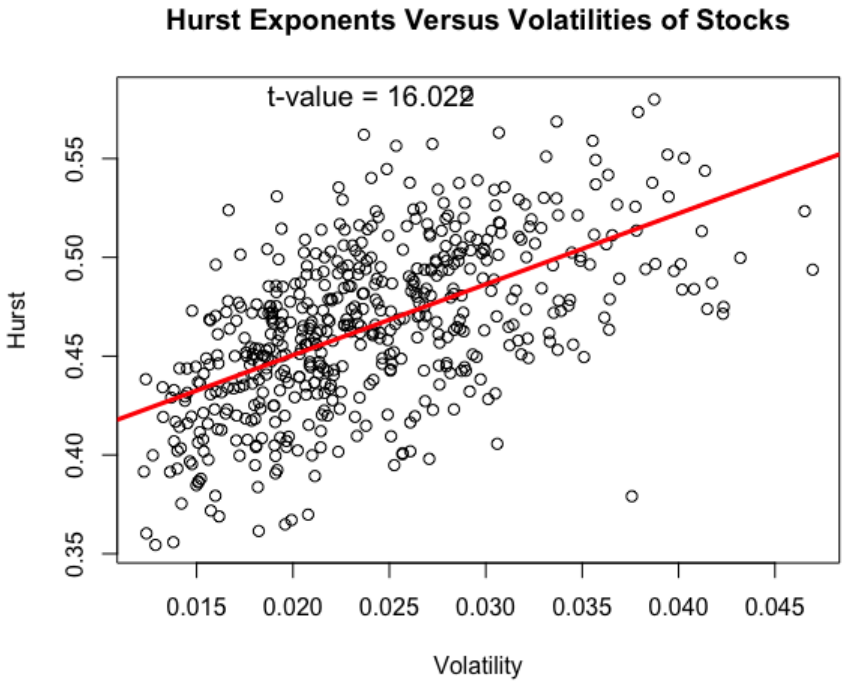

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=20, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of Stocks")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

Stock Volatility and Hurst Exponents

There is a strong relationship between stock volatility and hurst exponents.

Highly volatile stocks tend to have large Hurst exponents.

```
> # Calculate the volatility of stocks
> volv <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep)))
+ })  # end sapply
> # Dygraph of stock with highest volatility
> namev <- names(which.max(volv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of stock with lowest volatility
> namev <- names(which.min(volv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Calculate the regression of the Hurst exponents versus volatilit
> model <- lm(hurstv ~ volv)
> summary(model)
```



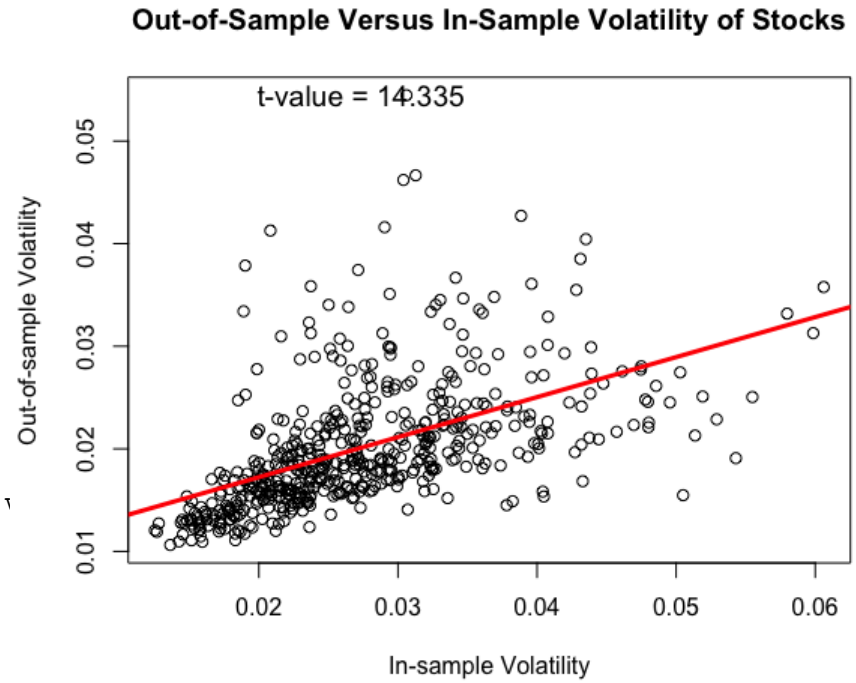

```
> # Plot scatterplot of the Hurst exponents versus volatilities
> plot(hurstv ~ volv, xlab="Volatility", ylab="Hurst",
+      main="Hurst Exponents Versus Volatilities of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(volv), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Volatility of Stocks

There is a strong relationship between *out-of-sample* and *in-sample* stock volatility.

Highly volatile stocks *in-sample* also tend to have high volatility *out-of-sample*.

```
> # Calculate the in-sample volatility of stocks
> volatis <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep["/2010"])))
+ })  # end sapply
> # Calculate the out-of-sample volatility of stocks
> volatos <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep["2010/"])))
+ })  # end sapply
> # Calculate the regression of the out-of-sample versus in-sample v
> model <- lm(volatos ~ volatis)
> summary(model)
```



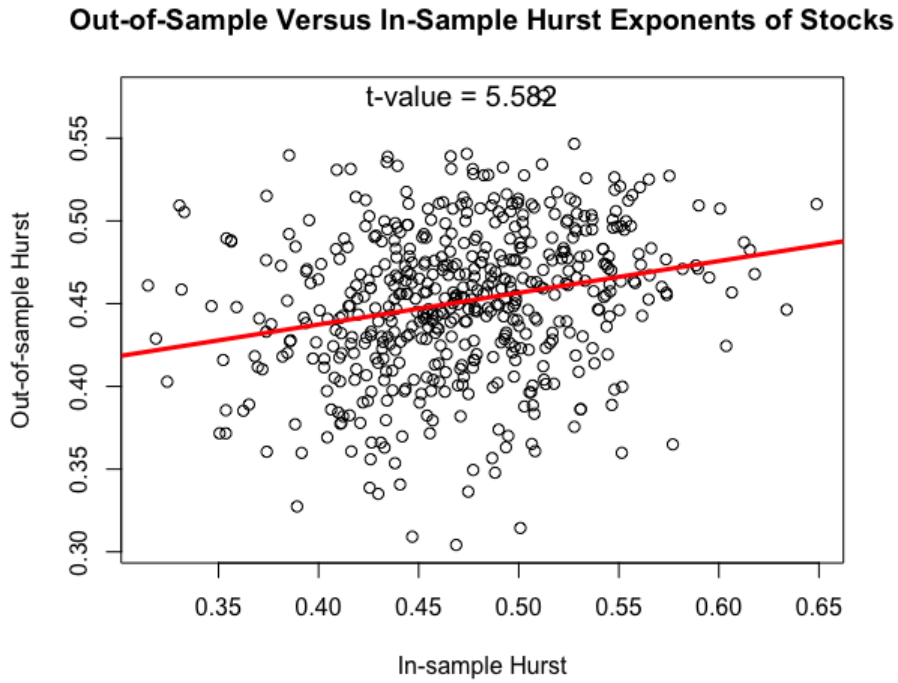

```
> # Plot scatterplot of the out-of-sample versus in-sample volatilit
> plot(volatos ~ volatis, xlab="In-sample Volatility", ylab="Out-of-
+      main="Out-of-Sample Versus In-Sample Volatility of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(volatis), y=max(volatos),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Hurst Exponents of Stocks

The *out-of-sample* Hurst exponents of stocks have a significant positive correlation to the *in-sample* Hurst exponents.

That means that stocks with larger *in-sample* Hurst exponents tend to also have larger *out-of-sample* Hurst exponents (but not always).

This is because stock volatility persists *out-of-sample*, and Hurst exponents are larger for higher volatility stocks.




```
> # Calculate the in-sample Hurst exponents of stocks
> hurstis <- sapply(pricev, function(closep) {
+   HighFreq::calc_hurst(closep["/2010"], aggv=aggv)
+ })  # end sapply
> # Calculate the out-of-sample Hurst exponents of stocks
> hurstos <- sapply(pricev, function(closep) {
+   HighFreq::calc_hurst(closep["2010/"], aggv=aggv)
+ })  # end sapply
> # Calculate the regression of the out-of-sample versus in-sample Hurst exponents
> model <- lm(hurstos ~ hurstis)
> summary(model)
```

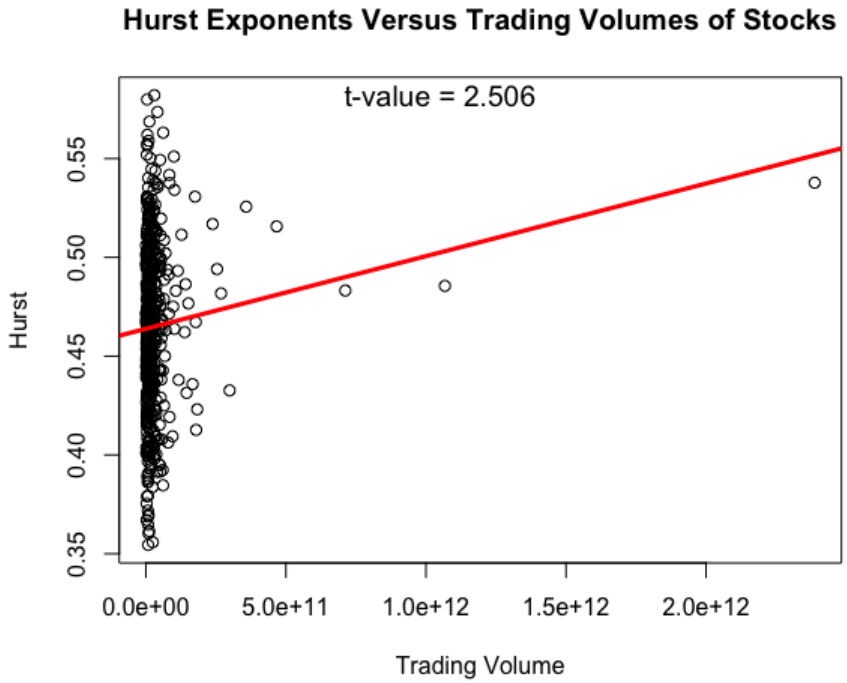
```
> # Plot scatterplot of the out-of-sample versus in-sample Hurst exp
> plot(hurstos ~ hurstis, xlab="In-sample Hurst", ylab="Out-of-sampl
+      main="Out-of-Sample Versus In-Sample Hurst Exponents of Stock
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(hurstis), y=max(hurstos),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Stock Trading Volumes and Hurst Exponents

The relationship between stock trading volumes and Hurst exponents is not very significant.

The relationship is dominated by a few stocks with very large trading volumes, like *AAPL*, which also tend to be more volatile and therefore have larger Hurst exponents.

```
> # Calculate the stock trading volumes after the year 2000
> volum <- eapply(sp500env, function(ohlc) {
+     sum(quantmod::Vo(ohlc)["2000/"])
+ })  # end eapply
> # Remove NULL values
> volum <- volum[names(pricev)]
> volum <- unlist(volum)
> which.max(volum)
> # Calculate the number of NULL prices
> sum(is.null(volum))
> # Calculate the Hurst exponents of stocks
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> # Calculate the regression of the Hurst exponents versus trading volumes
> model <- lm(hurstv ~ volum)
> summary(model)
```



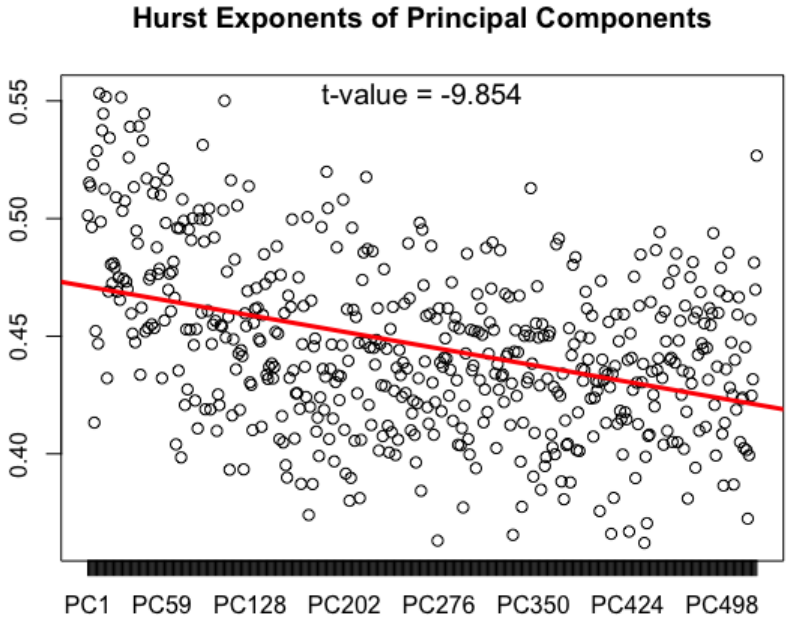

```
> # Plot scatterplot of the Hurst exponents versus trading volumes
> plot(hurstv ~ volum, xlab="Trading Volume", ylab="Hurst",
+      main="Hurst Exponents Versus Trading Volumes of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=quantile(volum, 0.998), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Hurst Exponents of Stock Principal Components

The Hurst exponents of the lower order principal components are typically larger than of the higher order principal components.

This is because the lower order principal components represent systematic risk factors, while the higher order principal components represent idiosyncratic risk factors, which are mean-reverting.

The Hurst exponents of most higher order principal components are less than 0.5, so they can potentially be traded in mean-reverting strategies.




```
> # Calculate log stock returns
> retp <- lapply(pricev, rutils::diffit)
> retp <- rutils::do_call(cbind, retp)
> retp[is.na(retp)] <- 0
> sum(is.na(retp))
> # Drop ".Close" from column names
> colnames(retp[, 1:4])
> colnames(retp) <- rutils::get_name(colnames(retp))
> # Calculate PCA prices using matrix algebra
> eigend <- eigen(cor(retp))
> retpca <- retp %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retpca),
+                      order.by=index(retp))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retp))
> # Calculate the Hurst exponents of PCAs
> hurstv <- sapply(pricepca, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of PCA with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
> # Dygraph of PCA with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
```

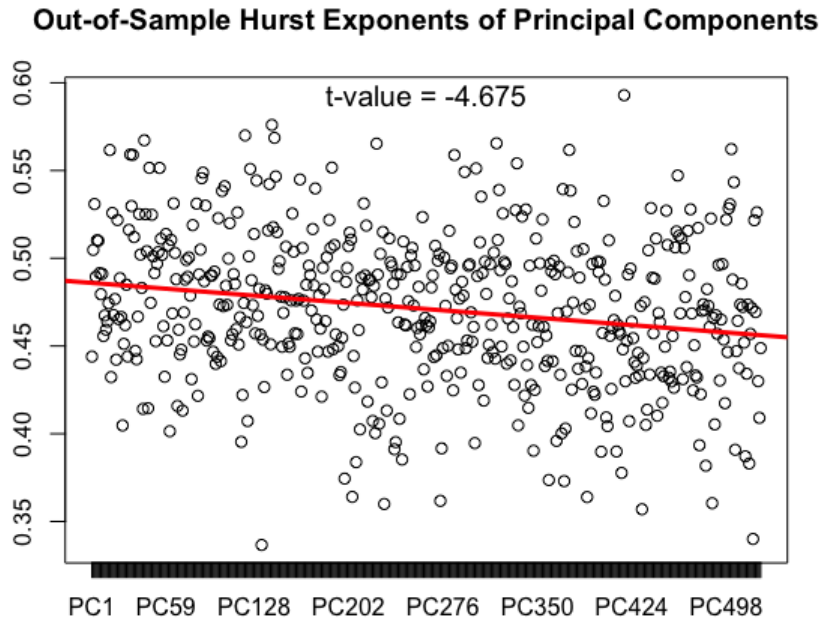
```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus thei
> orderv <- 1:NROW(hurstv)
> model <- lm(hurstv ~ orderv)
> summary(model)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(orderv), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Hurst Exponents of Stock Principal Components

The *out-of-sample* Hurst exponents of principal components also decrease with the increasing PCA order, the statistical significance is much lower.

That's because the PCA weights are not persistent *out-of-sample* - the PCA weights in the *out-of-sample* interval are often quite different from the *in-sample* weights.

```
> # Calculate in-sample eigen decomposition using matrix algebra
> eigend <- eigen(cor(retp["/2010"]))
> # Calculate out-of-sample PCA prices
> retpca <- retp["2010/"] %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retpca),
+                      order.by=index(retp["2010/"]))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retp))
> # Calculate the Hurst exponents of PCAs
> hurstv <- sapply(pricepca, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of PCA with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
> # Dygraph of PCA with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
```



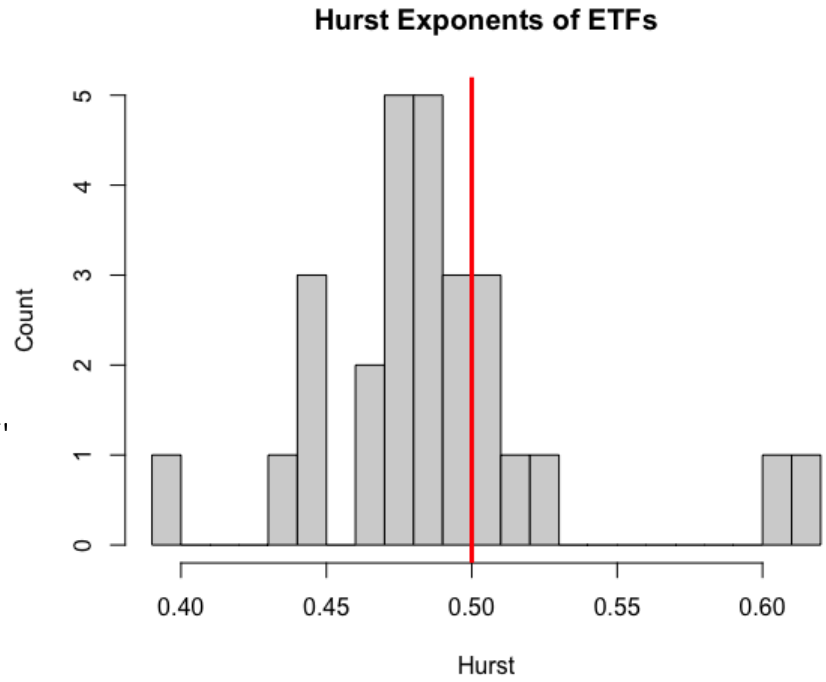

```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Out-of-Sample Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus their
> model <- lm(hurstv ~ orderv)
> summary(model)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(orderv), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Hurst Exponents of ETFs

The Hurst exponents of ETFs are also typically slightly less than 0.5, but they're closer to 0.5 than stocks, because they're portfolios stocks, so they have less idiosyncratic risk.

For this data sample, the commodity ETFs have the largest Hurst exponents while stock sector ETFs have the smallest Hurst exponents.

```
> # Get ETF log prices
> symbolv <- rutils::etfenv$symbolv
> symbolv <- symbolv[!(symbolv %in% c("MTUM", "QUAL", "VLUE", "USMV'
> pricev <- lapply(mget(symbolv, rutils::etfenv), function(x) {
+   log(na.omit(quantmod::Cl(x)))
+ })  # end lapply
> # Calculate the Hurst exponents of ETFs
> aggv <- trunc(seq.int(from=3, to=10, length.out=5)^2)
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> hurstv <- sort(unlist(hurstv))
> # Dygraph of ETF with smallest Hurst exponent
> namev <- names(first(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of ETF with largest Hurst exponent
> namev <- names(last(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
```



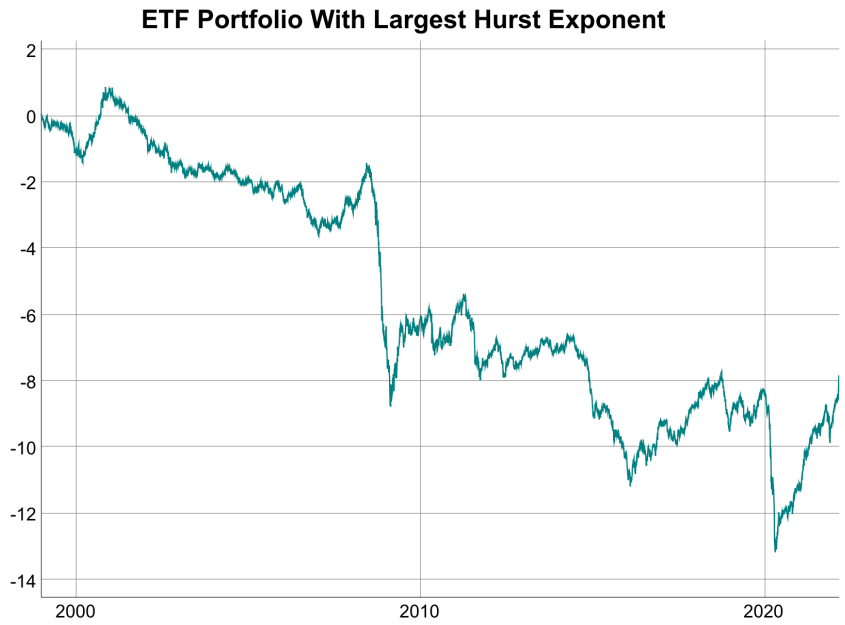

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=2e1, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of ETFs")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

ETF Portfolio With Largest Hurst Exponent

The portfolio weights can be optimized to maximize the portfolio's Hurst exponent.

The optimized portfolio exhibits very strong trending of returns, especially in periods of high volatility.

```
> # Calculate log ETF returns
> symbolv <- rutils::etfenv$symbolv
> symbolv <- symbolv[!(symbolv %in% c("MTUM", "QUAL", "VLUE", "USMV'
> retp <- rutils::etfenv$returns[, symbolv]
> retp[is.na(retp)] <- 0
> sum(is.na(retp))
> # Calculate the Hurst exponent of an ETF portfolio
> calc_phurst <- function(weightv, retp) {
+   -HighFreq::calc_hurst(matrix(cumsum(retp %*% weightv)), aggv=agg
+ }  # end calc_phurst
> # Calculate the portfolio weights with maximum Hurst
> nweights <- NCOL(retp)
> weightv <- rep(1/sqrt(nweights), nweights)
> calc_phurst(weightv, retp=retp)
> optiml <- optim(par=weightv, fn=calc_phurst, retp=retp,
+           method="L-BFGS-B",
+           upper=rep(10.0, nweights),
+           lower=rep(-10.0, nweights))
> # Optimal weights and maximum Hurst
> weightv <- optiml$par
> names(weightv) <- colnames(retp)
> -calc_phurst(weightv, retp=retp)
```




```
> # Dygraph of ETF portfolio with largest Hurst exponent
> wealthv <- xts::xts(cumsum(retp %*% weightv), zoo::index(retp))
> dygraphs::dygraph(wealthv, main="ETF Portfolio With Largest Hurst
```

Out-of-Sample ETF Portfolio With Largest Hurst Exponent

The portfolio weights can be optimized *in-sample* to maximize the portfolio's Hurst exponent.

But the *out-of-sample* Hurst exponent is close to `H = 0.5`, which means it's close to a random Brownian motion process.

```
> # Calculate the in-sample maximum Hurst portfolio weights
> optiml <- optim(par=weightv, fn=calc_phurst, retp=retp["/2010"],
+           method="L-BFGS-B",
+           upper=rep(10.0, nweights),
+           lower=rep(-10.0, nweights))
> # Optimal weights and maximum Hurst
> weightv <- optiml$par
> names(weightv) <- colnames(retp)
> # Calculate the in-sample Hurst exponent
> -calc_phurst(weightv, retp=retp["/2010"])
> # Calculate the out-of-sample Hurst exponent
> -calc_phurst(weightv, retp=retp["2010/"])
```

Autoregressive Processes

An *autoregressive* process *AR(n)* of order n for a time series r_t is defined as:

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

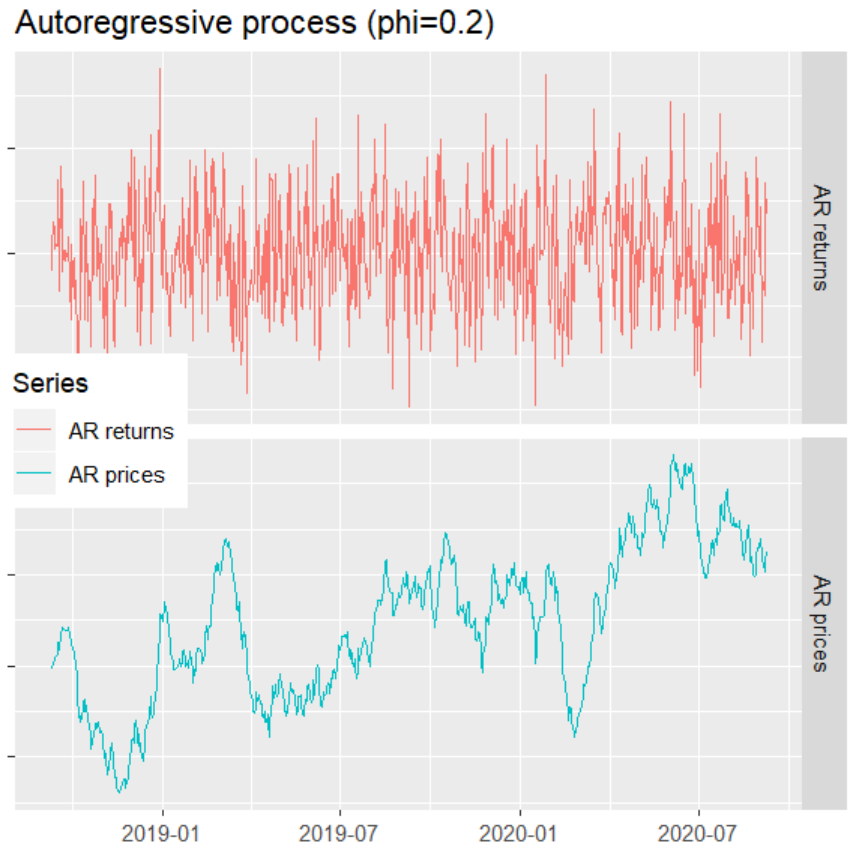
Where φ_i are the *AR(n)* coefficients, and ξ_t are standard normal *innovations*.

The *AR(n)* process is a special case of an *ARIMA* process, and is simply called an *AR(n)* process.

If the *AR(n)* process is *stationary* then the time series r_t is mean reverting to zero.

The function `arima.sim()` simulates *ARIMA* processes, with the `"model"` argument accepting a `list` of *AR(n)* coefficients φ_i.

```
> # Simulate AR processes
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Res
> datev <- Sys.Date() + 0:728  # Two year daily series
> # AR time series of returns
> arimav <- xts(x=arima.sim(n=NROW(datev), model=list(ar=0.2)),
+           order.by=datev)
> arimav <- cbind(arimav, cumsum(arimav))
> colnames(arimav) <- c("AR returns", "AR prices")
```



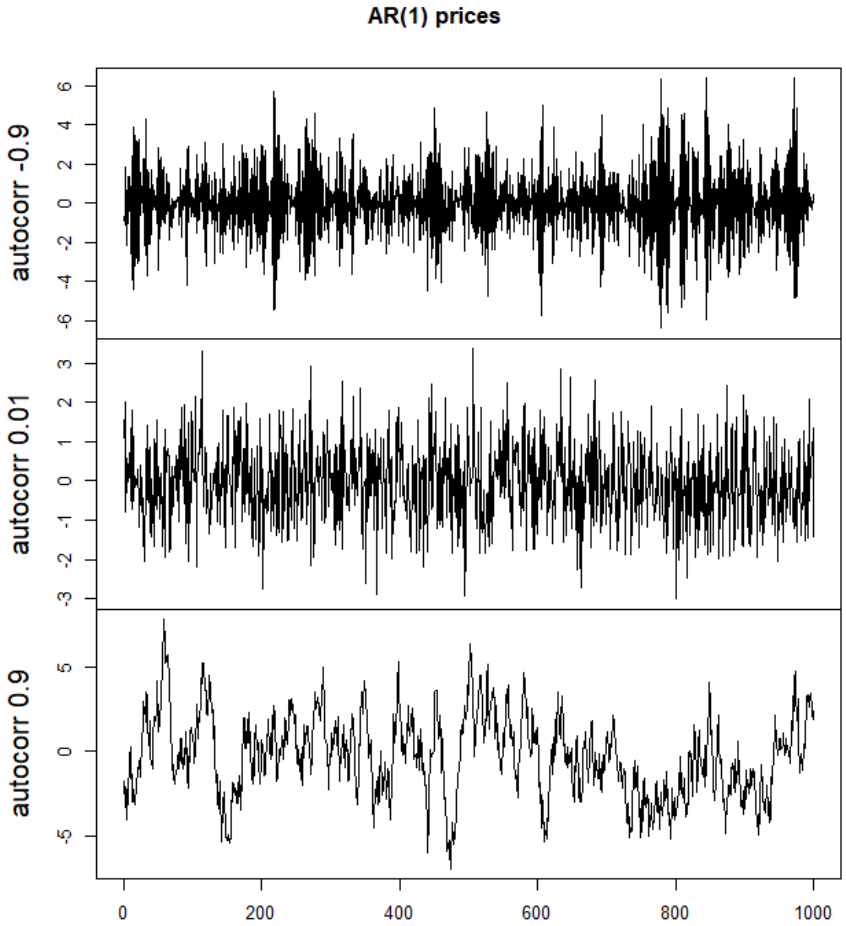

```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=arimav, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),
+   plot.background=element_blank(),
+   axis.text.y=element_blank())
```

Examples of Autoregressive Processes

The speed of mean reversion of an *AR(1)* process depends on the *AR(n)* coefficient φ_1, with a negative coefficient producing faster mean reversion, and a positive coefficient producing stronger diversion.

A positive coefficient φ_1 produces a diversion away from the mean, so that the time series r_t wanders away from the mean for longer periods of time.

```
> coeff <- c(-0.9, 0.01, 0.9)  # AR coefficients
> # Create three AR time series
> arimav <- sapply(coeff, function(phi) {
+   set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  #
+   arima.sim(n=NROW(datev), model=list(ar=phi))
+ })  # end sapply
> colnames(arimav) <- paste("autocorr", coeff)
> plot.zoo(arimav, main="AR(1) prices", xlab=NA)
> # Or plot using ggplot
> arimav <- xts(x=arimav, order.by=datev)
> library(ggplot)
> autoplot(arimav, main="AR(1) prices",
+    facets=Series ~ .) +
+     facet_grid(Series ~ ., scales="free_y") +
+ xlab("") +
+ 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())
```



Simulating Autoregressive Processes

An *autoregressive* process *AR(n)*:

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

Can be simulated by using an explicit recursive loop in R.

AR(n) processes can also be simulated by using the function `filter()` directly, with the argument `method="recursive"`.

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

The function `HighFreq::sim_ar()` simulates an *AR(n)* processes using C++ code.

```
> # Define AR(3) coefficients and innovations
> coeff <- c(0.1, 0.39, 0.5)
> nrows <- 1e2
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection"); innov
> # Simulate AR process using recursive loop in R
> arimav <- numeric(nrows)
> arimav[1] <- innov[1]
> arimav[2] <- coeff[1]*arimav[1] + innov[2]
> arimav[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1] + innov[3]
> for (it in 4:NROW(arimav)) {
+   arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]
+ }  # end for
> # Simulate AR process using filter()
> arimaf <- filter(x=innov, filter=coeff, method="recursive")
> class(arimaf)
> all.equal(arimav, as.numeric(arimaf))
> # Fast simulation of AR process using C_rfilter()
> arimacpp <- .Call(stats:::C_rfilter, innov, coeff,
+      double(NROW(coeff) + NROW(innov)))[-(1:3)]
> all.equal(arimav, arimacpp)
> # Fastest simulation of AR process using HighFreq::sim_ar()
> arimav <- HighFreq::sim_ar(coeff=matrix(coeff), innov=matrix(innov
> arimav <- drop(arimav)
> all.equal(arimav, arimacpp)
> # Benchmark the speed of the three methods of simulating AR proces
> library(microbenchmark)
> summary(microbenchmark(
+   Rloop={for (it in 4:NROW(arimav)) {
+     arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]
+   }},
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   cpp=HighFreq::sim_ar(coeff=matrix(coeff), innov=matrix(innov))
+   ), times=10)[, c(1, 4, 5)]
```

Simulating Autoregressive Processes Using arima.sim()

The function `arima.sim()` simulates *ARIMA* processes by calling the function `filter()`.

ARIMA processes can also be simulated by using the function `filter()` directly, with the argument `method="recursive"`.

Simulating stationary *autoregressive* processes requires a *warmup period*, to allow the process to reach its stationary state.

The required length of the *warmup period* depends on the smallest root of the characteristic equation, with a longer *warmup period* needed for smaller roots, that are closer to 1.

The *rule of thumb* (heuristic rule, guideline) is for the *warmup period* to be equal to 6 divided by the logarithm of the smallest characteristic root plus the number of *AR(n)* coefficients: $\frac{6}{\log(minroot)} + \text{numcoeff}$

```
> # Calculate modulus of roots of characteristic equation
> rootv <- Mod(polyroot(c(1, -coeff)))
> # Calculate warmup period
> warmup <- NROW(coeff) + ceiling(6/log(min(rootv)))
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nrows <- 1e4
> innov <- rnorm(nrows + warmup)
> # Simulate AR process using arima.sim()
> arimav <- arima.sim(n=nrows,
+   model=list(ar=coeff),
+   start.innov=innov[1:warmup],
+   innov=innov[(warmup+1):NROW(innov)])
> # Simulate AR process using filter()
> arimaf <- filter(x=innov, filter=coeff, method="recursive")
> all.equal(arimaf[-(1:warmup)], as.numeric(arimav))
> # Benchmark the speed of the three methods of simulating AR proces
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   arima_sim=arima.sim(n=nrows,
+                       model=list(ar=coeff),
+                       start.innov=innov[1:warmup],
+                       innov=innov[(warmup+1):NROW(innov)]),
+   arima_loop={for (it in 4:NROW(arimav)) {
+   arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]}}
+   ), times=10)[, c(1, 4, 5)]
```

Autocorrelations of Autoregressive Processes

The autocorrelation ρ_i of an *AR(1)* process (defined as $r_t = \varphi r_{t-1} + \xi_t$), satisfies the recursive equation: $\rho_i = \varphi \rho_{i-1}$, with $\rho_1 = \varphi$.

Therefore *AR(1)* processes have exponentially decaying autocorrelations: $\rho_i = \varphi^i$.

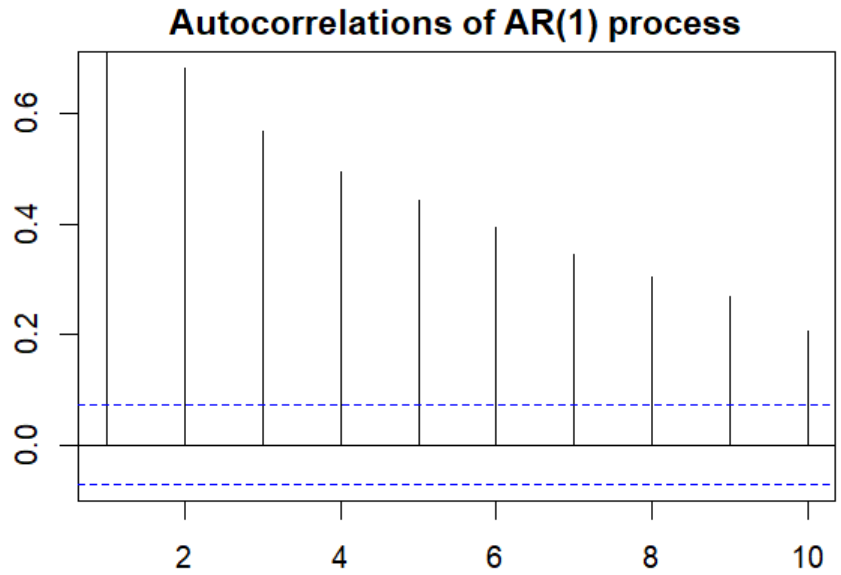
The *AR(1)* process can be simulated recursively:

$$r_1 = \xi_1$$
$$r_2 = \varphi r_1 + \xi_2 = \xi_2 + \varphi \xi_1$$
$$r_3 = \xi_3 + \varphi \xi_2 + \varphi^2 \xi_1$$
$$r_4 = \xi_4 + \varphi \xi_3 + \varphi^2 \xi_2 + \varphi^3 \xi_1$$

Therefore the *AR(1)* process can be expressed as a *moving average* (*MA*) of the *innovations* ξ_t: $r_t = \sum_{i=1}^{n} \varphi^{i-1} \xi_t$.

If $\varphi < 1.0$ then the influence of the innovation ξ_t decays exponentially.

If $\varphi = 1.0$ then the influence of the random innovations ξ_t persists indefinitely, so that the variance of r_t is proportional to time.



An *AR(1)* process has an exponentially decaying *ACF*.

```
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> # Simulate AR(1) process
> arimav <- arima.sim(n=1e3, model=list(ar=0.8))
> # ACF of AR(1) process
> acfl <- rutils::plot_acf(arimav, lag=10, xlab="", ylab="",
+   main="Autocorrelations of AR(1) process")
> acfl$acf[1:5]
```

Partial Autocorrelations

An autocorrelation of lag 1 induces higher order autocorrelations of lag `2, 3, ...`, which may obscure the direct higher order autocorrelations.

If two random variables are both correlated to a third variable, then they are indirectly correlated with each other.

The indirect correlation can be removed by defining new variables with no correlation to the third variable.

The *partial correlation* is the correlation after the correlations to the common variables are removed.

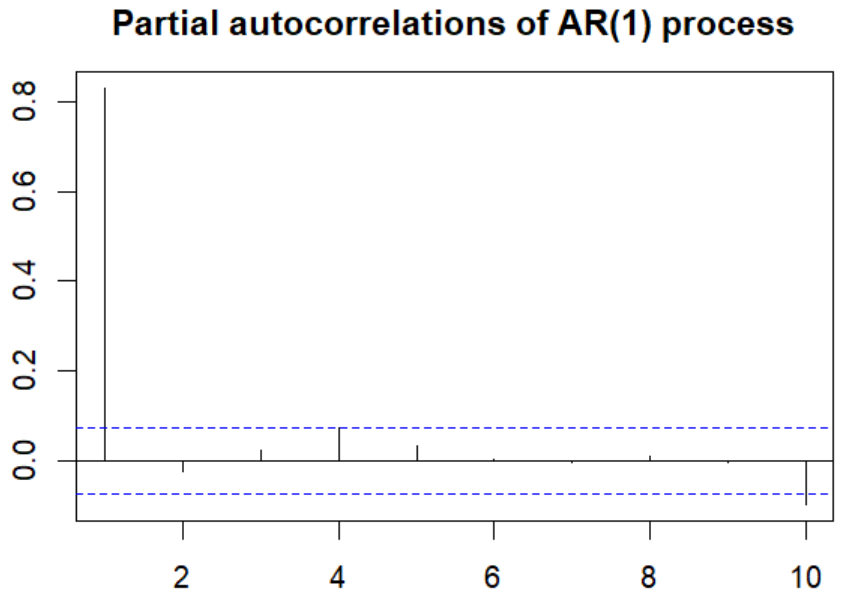
A linear combination of the time series and its own lag can be created, such that its lag 1 autocorrelation is zero.

The lag 2 autocorrelation of this new series is called the *partial autocorrelation* of lag 2, and represents the true second order autocorrelation.

The *partial autocorrelation* of lag `k` is the autocorrelation of lag `k`, after all the autocorrelations of lag `1, ..., k-1` have been removed.

The *partial autocorrelations* ϱ_i are the estimators of the coefficients ϕ_i of the *AR(n)* process.

The function `pacf()` calculates and plots the *partial autocorrelations* using the Durbin-Levinson algorithm.



An *AR(1)* process has an exponentially decaying *ACF* and a non-zero *PACF* at lag one.

```
> # PACF of AR(1) process
> pacfl <- pacf(arimav, lag=10, xlab="", ylab="", main="")
> title("Partial autocorrelations of AR(1) process", line=1)
> pacfl <- as.numeric(pacfl$acf)
> pacfl[1:5]
```

Stationary Processes and Unit Root Processes

A process is *stationary* if its probability distribution does not change with time, which means that it has constant mean and variance.

The *autoregressive* process *AR(n)*:
$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$

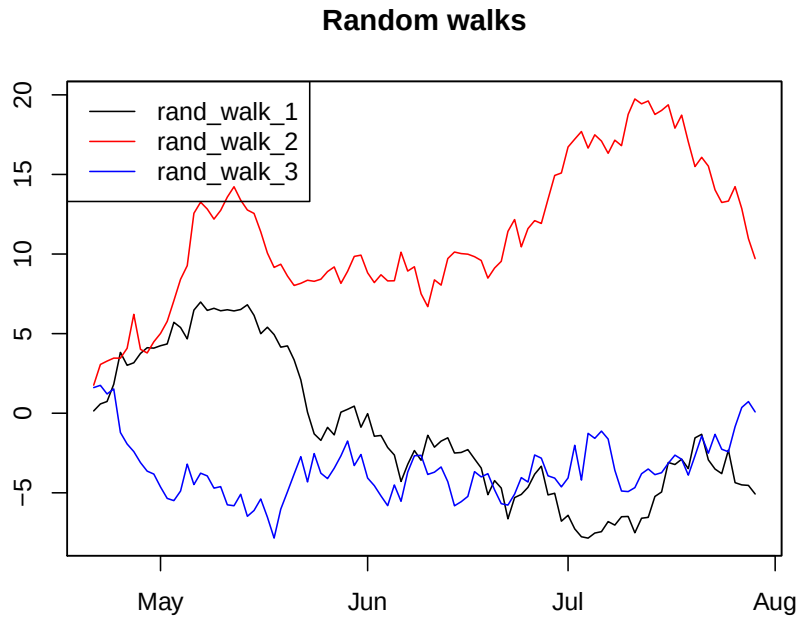
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_n z^n = 0$

An autoregressive process is *stationary* only if the absolute values of all the roots of its characteristic equation are greater than 1.

If the sum of the autoregressive coefficients is equal to 1: $\sum_{i=1}^{n} \varphi_i = 1$, then the process has a root equal to 1 (it has a *unit root*), so it's not *stationary*.

Non-stationary processes with unit roots are called *unit root* processes.

A simple example of a *unit root* process is the *Brownian Motion*: $p_t = p_{t-1} + \xi_t$




```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randw <- cumsum(zoo(matrix(rnorm(3*100), ncol=3),
+ order.by=(Sys.Date()+0:99)))
> colnames(randw) <- paste("randw", 1:3, sep="_")
> plot.zoo(randw, main="Random walks",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue"))
> # Add legend
> legend(x="topleft", legend=colnames(randw),
+  col=c("black", "red", "blue"), lty=1)
```

Integrated and Unit Root Processes

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: $p_t = \sum_{i=1}^{t} r_i$.

If returns follow an *AR(n)* process:
$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$

Then asset prices follow the process:
$p_t = (1 + \varphi_1)p_{t-1} + (\varphi_2 - \varphi_1)p_{t-2} + \ldots + (\varphi_n - \varphi_{n-1})p_{t-n} - \varphi_n p_{t-n-1} + \xi_t$

The sum of the coefficients of the price process is equal to 1, so it has a *unit root* for all values of the φ_i coefficients.

The *integrated* process of an *AR(n)* process is always a *unit root* process.

For example, if returns follow an *AR(1)* process:
$r_t = \varphi r_{t-1} + \xi_t$.

Then asset prices follow the process:
$p_t = (1 + \varphi)p_{t-1} - \varphi p_{t-2} + \xi_t$

Which is a *unit root* process for all values of φ, because the sum of its coefficients is equal to 1.

If $\varphi = 0$ then the above process is a *Brownian Motion* (random walk).

```
> # Simulate arima with large AR coefficient
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nrows <- 1e4
> arimav <- arima.sim(n=nrows, model=list(ar=0.99))
> tseries::adf.test(arimav)
> # Integrated series has unit root
> tseries::adf.test(cumsum(arimav))
> # Simulate arima with negative AR coefficient
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> arimav <- arima.sim(n=nrows, model=list(ar=-0.99))
> tseries::adf.test(arimav)
> # Integrated series has unit root
> tseries::adf.test(cumsum(arimav))
```

The Variance of Unit Root Processes

An *AR(1)* process: $r_t = \varphi r_{t-1} + \xi_t$ has the following characteristic equation: $1 - \varphi z = 0$, with a root equal to: $z = 1/\varphi$

If $\varphi = 1$, then the characteristic equation has a *unit root* (and therefore it isn't *stationary*), and the process follows: $r_t = r_{t-1} + \xi_t$

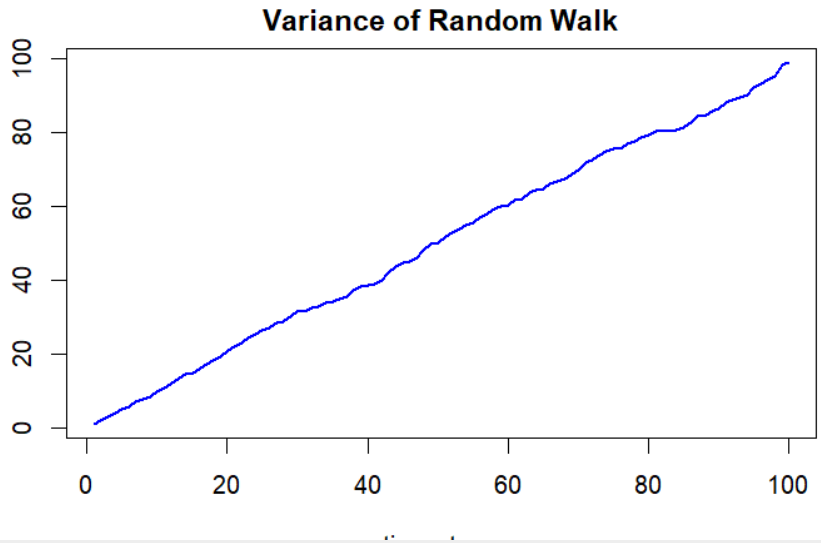
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

The expected value of the *AR(1)* process $r_t = \varphi r_{t-1} + \xi_t$ is equal to zero: $\mathbb{E}[r_t] = \frac{\mathbb{E}[\xi_t]}{1-\varphi} = 0$.

And its variance is equal to: $\sigma^2 = \mathbb{E}[r_i^2] = \frac{\sigma_\xi^2}{1-\varphi^2}$.

If $\varphi = 1$, then the *variance* grows over time and becomes infinite over time, so the process is not *stationary*.

The variance of the *Brownian Motion* $r_t = r_{t-1} + \xi$ is proportional to time: $\sigma_i^2 = \mathbb{E}[r_i^2] = i\sigma_\xi^2$




```
> # Simulate random walks using apply() loops
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randws <- matrix(rnorm(1000*100), ncol=1000)
> randws <- apply(randws, 2, cumsum)
> varv <- apply(randws, 1, var)
> # Simulate random walks using vectorized functions
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randws <- matrixStats::colCumsums(matrix(rnorm(1000*100), ncol=100
> varv <- matrixStats::rowVars(randws)
> par(mar=c(5, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot(varv, xlab="time steps", ylab="",
+      t="l", col="blue", lwd=2,
+      main="Variance of Random Walk")
```

The Brownian Motion Process

In the *Brownian Motion* process, the returns r_t are equal to the random *innovations*:

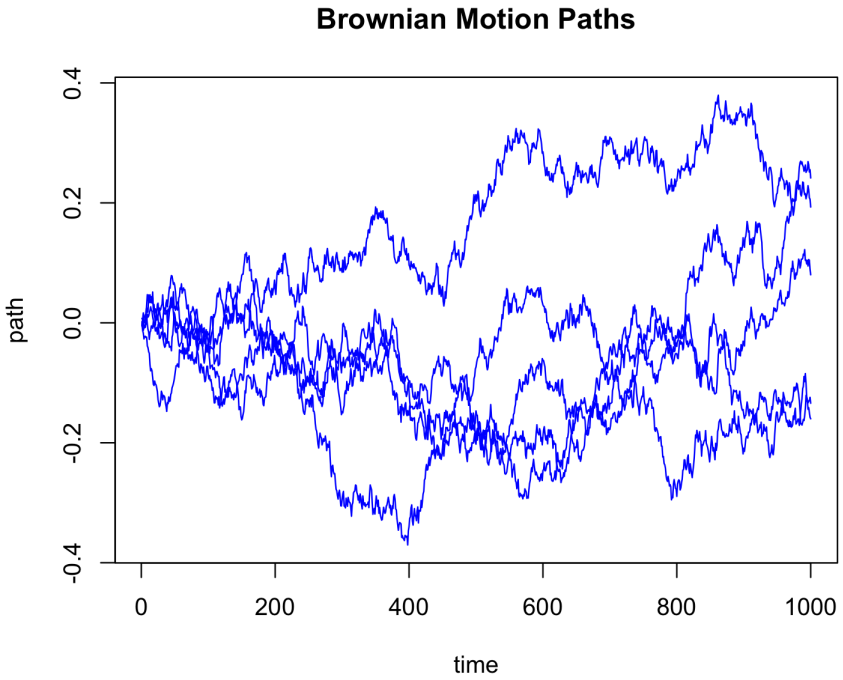
$$r_t = p_t - p_{t-1} = \sigma \xi_t$$

$$p_t = p_{t-1} + r_t$$

Where σ is the volatility of returns, and ξ_t are random normal *innovations* with zero mean and unit variance.

The *Brownian Motion* process for prices can be written as an *AR(1)* autoregressive process with coefficient $\varphi = 1$:

$$p_t = \varphi p_{t-1} + \sigma \xi_t$$




```
> # Define Brownian Motion parameters
> nrows <- 1000; sigmav <- 0.01
> # Simulate 5 paths of Brownian motion
> pricev <- matrix(rnorm(5*nrows, sd=sigmav), nc=5)
> pricev <- matrixStats::colCumsums(pricev)
> # Plot 5 paths of Brownian motion
> matplot(y=pricev, main="Brownian Motion Paths",
+   xlab="time", ylab="path",
+   type="l", lty="solid", lwd=1, col="blue")
> # Save plot to png file on Mac
> quartz.save("figure/brown_paths.png", type="png", width=6, height=4)
```

The Ornstein-Uhlenbeck Process

In the *Ornstein-Uhlenbeck* process, the returns r_t are equal to the difference between the equilibrium price μ minus the latest price p_{t-1}, times the mean reversion parameter θ, plus random *innovations*:

$$r_t = p_t - p_{t-1} = \theta\,(\mu - p_{t-1}) + \sigma\,\xi_t$$

$$p_t = p_{t-1} + r_t$$

Where σ is the volatility of returns, and ξ_t are random normal *innovations* with zero mean and unit variance.

The *Ornstein-Uhlenbeck* process for prices can be written as an *AR(1)* process plus a drift:

$$p_t = \theta\,\mu + (1 - \theta)\,p_{t-1} + \sigma\,\xi_t$$

The *Ornstein-Uhlenbeck* process cannot be simulated using the function `filter()` because of the drift term, so it must be simulated using explicit loops, either in `R` or in `C++`.

The compiled *Rcpp* `C++` code can be over 100 times faster than loops in `R`!

```
> # Define Ornstein-Uhlenbeck parameters
> prici <- 0.0; priceq <- 1.0;
> sigmav <- 0.02; thetav <- 0.01; nrows <- 1000
> # Initialize the data
> innov <- rnorm(nrows)
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> retp[1] <- sigmav*innov[1]
> pricev[1] <- prici
> # Simulate Ornstein-Uhlenbeck process in R
> for (i in 2:nrows) {
+   retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1] + retp[i]
+ }  # end for
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> pricecpp <- HighFreq::sim_ou(prici=prici, priceq=priceq,
+   theta=thetav, innov=matrix(sigmav*innov))
> all.equal(pricev, drop(pricecpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (i in 2:nrows) {
+     retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+     pricev[i] <- pricev[i-1] + retp[i]}},
+   Rcpp=HighFreq::sim_ou(prici=prici, priceq=priceq,
+     theta=thetav, innov=matrix(sigmav*innov)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

The Solution of the Ornstein-Uhlenbeck Process

The *Ornstein-Uhlenbeck* process in continuous time is:

$$\mathrm{d}p_t = \theta\,(\mu - p_t)\,\mathrm{d}t + \sigma\,\mathrm{d}B_t$$

Where B_t is a *Brownian Motion*, with $\mathrm{d}B_t$ following the normal distribution $\phi(0, \sqrt{\mathrm{d}t})$, with the volatility $\sqrt{\mathrm{d}t}$, equal to the square root of the time increment $\mathrm{d}t$.

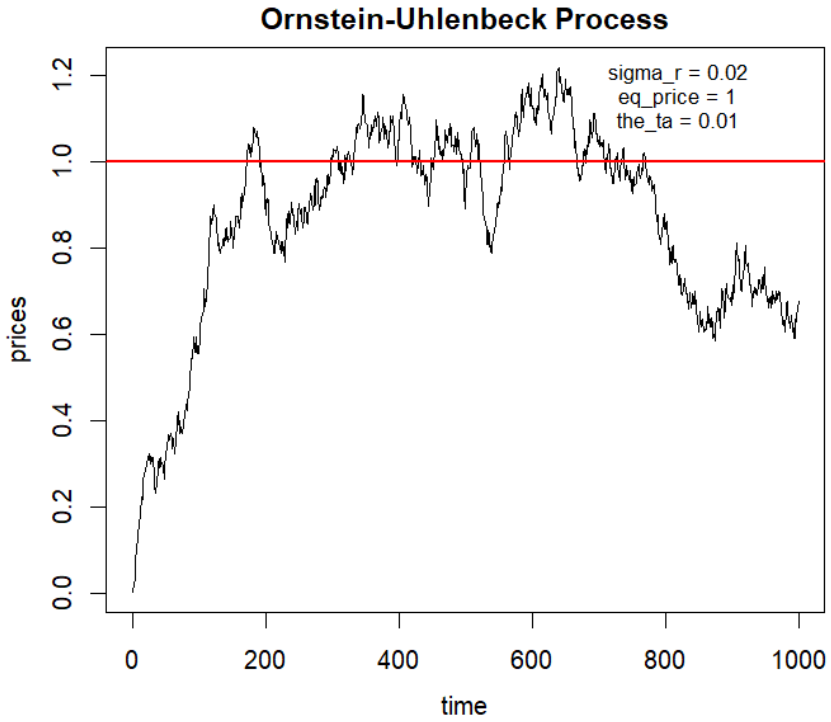
The solution of the *Ornstein-Uhlenbeck* process is given by:

$$p_t = p_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) + \sigma \int_0^t e^{\theta(s-t)} \mathrm{d}W_s$$

The mean and variance are given by:
$\mathbb{E}[p_t] = p_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) \to \mu$
$\mathbb{E}[(p_t - \mathbb{E}[p_t])^2] = \frac{\sigma^2}{2\theta}(1 - e^{-\theta t}) \to \frac{\sigma^2}{2\theta}$

The *Ornstein-Uhlenbeck* process is mean reverting to a non-zero equilibrium price μ.

The *Ornstein-Uhlenbeck* process needs a *warmup period* before it reaches equilibrium.

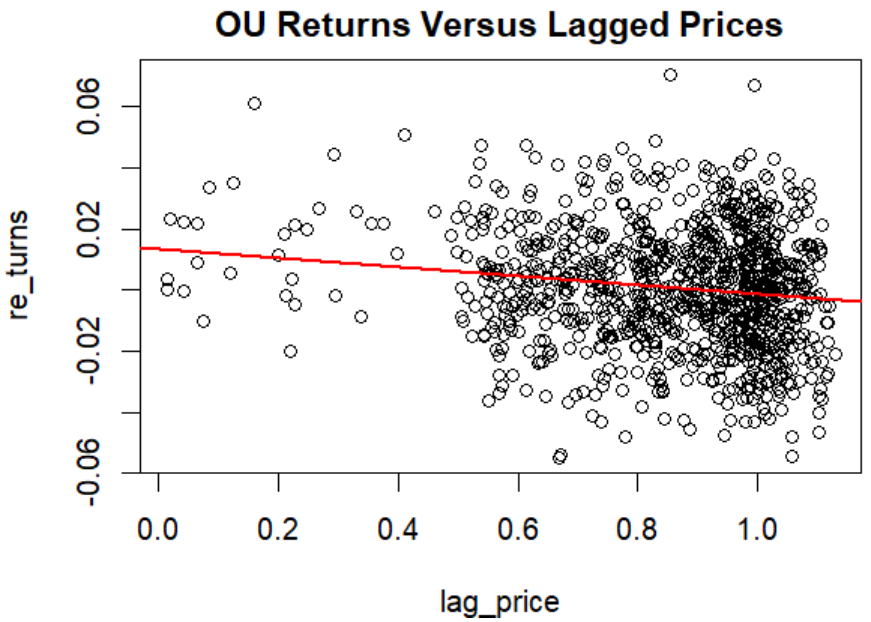



```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright", title=paste(c(paste0("sigmav = ", sigmav),
+      paste0("priceq = ", ),
+      paste0("thetav = ", thetav)),
+    collapse="\n"),
+  legend="", cex=0.8, inset=0.1, bg="white", bty="n")
> abline(h=, col='red', lwd=2)
```

Ornstein-Uhlenbeck Process Returns Correlation

Under the *Ornstein-Uhlenbeck* process, the returns are negatively correlated to the lagged prices.

```
> retp <- rutils::diffit(pricev)
> pricelag <- rutils::lagit(pricev)
> formulav <- retp ~ pricelag
> regmod <- lm(formulav)
> summary(regmod)
> # Plot regression
> plot(formulav, main="OU Returns Versus Lagged Prices")
> abline(regmod, lwd=2, col="red")
```



Calibrating the Ornstein-Uhlenbeck Parameters

The volatility parameter of the Ornstein-Uhlenbeck process can be estimated directly from the standard deviation of the returns.

The θ and μ parameters can be estimated from the linear regression of the returns versus the lagged prices.

Calculating regression parameters directly from formulas has the advantage of much faster calculations.

```
> # Calculate volatility parameter
> c(volatility=sigmav, estimate=sd(retp))
> # Extract OU parameters from regression
> coeff <- summary(regmod)$coefficients
> # Calculate regression alpha and beta directly
> betac <- cov(retp, pricelag)/var(pricelag)
> alphac <- (mean(retp) - betac*mean(pricelag))
> cbind(direct=c(alpha=alphac, beta=betac), lm=coeff[, 1])
> all.equal(c(alpha=alphac, beta=betac), coeff[, 1],
+     check.attributes=FALSE)
> # Calculate regression standard errors directly
> betac <- c(alpha=alphac, beta=betac)
> fitv <- (alphac + betac*pricelag)
> resids <- (retp - fitv)
> prices2 <- sum((pricelag - mean(pricelag))^2)
> betasd <- sqrt(sum(resids^2)/prices2/(nrows-2))
> alphasd <- sqrt(sum(resids^2)/(nrows-2)*(1:nrows + mean(pricelag)
> cbind(direct=c(alphasd=alphasd, betasd=betasd), lm=coeff[, 2])
> all.equal(c(alphasd=alphasd, betasd=betasd), coeff[, 2],
+     check.attributes=FALSE)
> # Compare mean reversion parameter theta
> c(theta=(-thetav), round(coeff[2, ], 3))
> # Compare equilibrium price mu
> c(priceq=priceq, estimate=-coeff[1, 1]/coeff[2, 1])
> # Compare actual and estimated parameters
> coeff <- cbind(c(thetav*priceq, -thetav), coeff[, 1:2])
> rownames(coeff) <- c("drift", "theta")
> colnames(coeff)[1] <- "actual"
> round(coeff, 4)
```

The Schwartz Process

The *Ornstein-Uhlenbeck* prices can be negative, while actual prices are usually not negative.

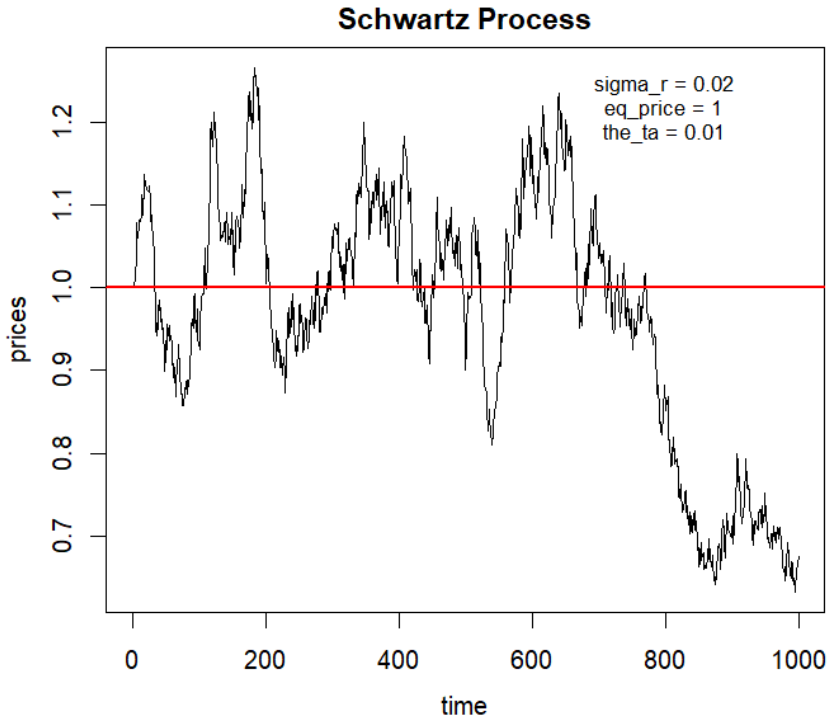
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of prices, which can be negative.

The *Schwartz* process is the exponential of the *Ornstein-Uhlenbeck* process, so it avoids negative prices by compounding the percentage returns r_t instead of summing them:

$$r_t = \log p_t - \log p_{t-1} = \theta (\mu - p_{t-1}) + \sigma \xi_t$$

$$p_t = p_{t-1} \exp(r_t)$$

Where the parameter θ is the strength of mean reversion, σ is the volatility, and ξ_t are random normal *innovations* with zero mean and unit variance.




```
> # Simulate Schwartz process
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> pricev[1] <- exp(sigmav*innov[1])
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Re
> for (i in 2:nrows) {
+   retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1]*exp(retp[i])
+ }  # end for
```

```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Schwartz Process")
> legend("topright",
+  title=paste(c(paste0("sigmav = ", sigmav),
+      paste0("priceq = ", priceq),
+      paste0("thetav = ", thetav)),
+    collapse="\n"),
+  legend="", cex=0.8, inset=0.12, bg="white", bty="n")
> abline(h=priceq, col='red', lwd=2)
```

The Dickey-Fuller Process

The *Dickey-Fuller* process is a combination of an *Ornstein-Uhlenbeck* process and an *autoregressive* process.

The returns r_t are equal to the sum of a mean reverting term plus *autoregressive* terms:

$$r_t = \theta(\mu - p_{t-1}) + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \sigma \xi_t$$

$$p_t = p_{t-1} + r_t$$

Where μ is the equilibrium price, σ is the volatility of returns, θ is the strength of mean reversion, and ξ_t are standard normal *innovations*.

Then the prices follow an *autoregressive* process:

$$p_t = \theta\mu + (1 + \varphi_1 - \theta)p_{t-1} + (\varphi_2 - \varphi_1)p_{t-2} + \ldots + (\varphi_n - \varphi_{n-1})p_{t-n} - \varphi_n p_{t-n-1} + \sigma \xi_t$$

The sum of the *autoregressive* coefficients is equal to $1 - \theta$, so if the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_t exhibits mean reversion and has no *unit root*.

```
> # Define Dickey-Fuller parameters
> prici <- 0.0;  priceq <- 1.0
> thetav <- 0.01;  nrows <- 1000
> coeff <- c(0.1, 0.39, 0.5)
> # Initialize the data
> innov <- rnorm(nrows, sd=0.01)
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> # Simulate Dickey-Fuller process using recursive loop in R
> retp[1] <- innov[1]
> pricev[1] <- prici
> retp[2] <- thetav*(priceq - pricev[1]) + coeff[1]*retp[1] +
+   innov[2]
> pricev[2] <- pricev[1] + retp[2]
> retp[3] <- thetav*(priceq - pricev[2]) + coeff[1]*retp[2] +
+   coeff[2]*retp[1] + innov[3]
> pricev[3] <- pricev[2] + retp[3]
> for (it in 4:nrows) {
+   retp[it] <- thetav*(priceq - pricev[it-1]) +
+     retp[(it-1):(it-3)] %*% coeff + innov[it]
+   pricev[it] <- pricev[it-1] + retp[it]
+ }  # end for
> # Simulate Dickey-Fuller process in Rcpp
> pricecpp <- HighFreq::sim_df(prici=prici, priceq=priceq,
+    theta=thetav, coeff=matrix(coeff), innov=matrix(innov))
> # Compare prices
> all.equal(pricev, drop(pricecpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (it in 4:nrows) {
+   retp[it] <- thetav*(priceq - pricev[it-1]) + retp[(it-1):(it-3)]
+   pricev[it] <- pricev[it-1] + retp[it]
+   }},
+   Rcpp=HighFreq::sim_df(prici=prici, priceq=priceq, theta=thetav,
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Augmented Dickey-Fuller ADF Test for Unit Roots

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

The *ADF* test fits an autoregressive model for the prices p_t:

$$r_t = \theta(\mu - p_{t-1}) + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \sigma \xi_t$$

$$p_t = p_{t-1} + r_t$$

Where μ is the equilibrium price, σ is the volatility of returns, and θ is the strength of mean reversion.

ε_i are the *residuals*, which are assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_t exhibits mean reversion and has no *unit root*.

The *null hypothesis* is that prices have a unit root ($\theta = 0$, no mean reversion), while the alternative hypothesis is that it's *stationary* ($\theta > 0$, mean reversion).

The *ADF* test statistic is equal to the *t*-value of the θ parameter: $t_\theta = \hat{\theta}/SE_\theta$ (which follows a different distribution from the t-distribution).

The function `tseries::adf.test()` performs the *ADF* test.

```
> # Simulate AR(1) process with coefficient=1, with unit root
> innov <- matrix(rnorm(1e4, sd=0.01))
> arimav <- HighFreq::sim_ar(coeff=matrix(1), innov=innov)
> plot(arimav, t="l", main="Brownian Motion")
> # Perform ADF test with lag = 1
> tseries::adf.test(arimav, k=1)
> # Perform standard Dickey-Fuller test
> tseries::adf.test(arimav, k=0)
> # Simulate AR(1) with coefficient close to 1, without unit root
> arimav <- HighFreq::sim_ar(coeff=matrix(0.99), innov=innov)
> plot(arimav, t="l", main="AR(1) coefficient = 0.99")
> tseries::adf.test(arimav, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with mean reversion
> prici <- 0.0; priceq <- 0.0; thetav <- 0.1
> pricev <- HighFreq::sim_ou(prici=prici, priceq=priceq,
+   theta=thetav, innov=innov)
> plot(pricev, t="l", main=paste("OU coefficient =", thetav))
> tseries::adf.test(pricev, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with zero reversion
> thetav <- 0.0
> pricev <- HighFreq::sim_ou(prici=prici, priceq=priceq,
+   theta=thetav, innov=innov)
> plot(pricev, t="l", main=paste("OU coefficient =", thetav))
> tseries::adf.test(pricev, k=1)
```

The common practice is to use a small number of lags in the *ADF* test, and if the residuals are autocorrelated, then to increase them until the correlations are no longer significant.

If the number of lags in the regression is zero: $n = 0$ then the *ADF* test becomes the standard *Dickey-Fuller* test: $r_t = \theta(\mu - p_{t-1}) + \varepsilon_i$.

Sensitivity of the ADF Test for Detecting Unit Roots

The *ADF null hypothesis* is that prices have a unit root, while the alternative hypothesis is that they're *stationary*.

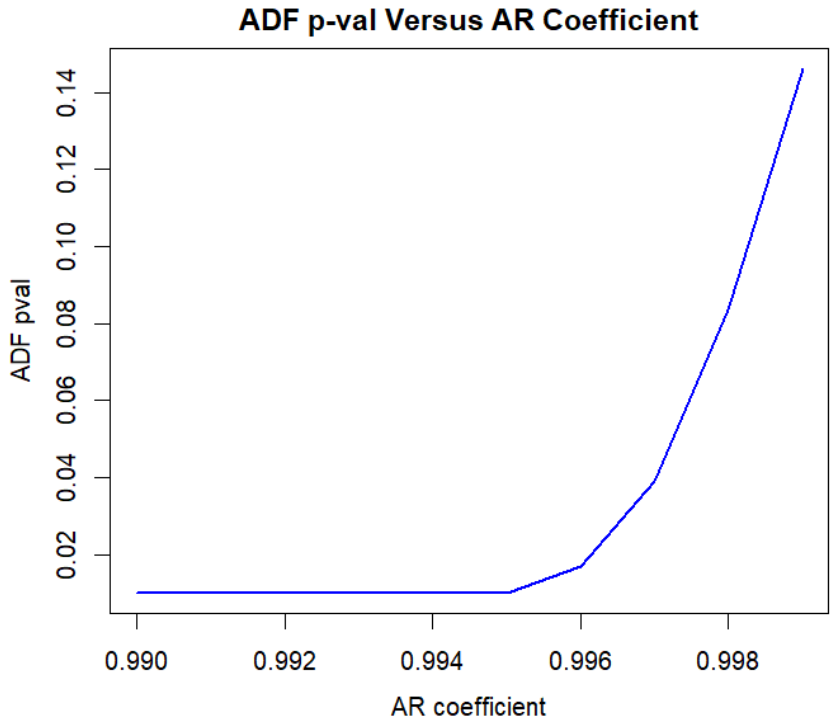
The *ADF* test has low *sensitivity*, i.e. the ability to correctly identify time series with no *unit root*, causing it to produce *false negatives* (*type II* errors).

This is especially true for time series which exhibit mean reversion over longer time horizons. The *ADF* test will identify them as having a *unit root* even though they are mean reverting.

Therefore the *ADF* test often requires a lot of data before it's able to correctly identify *stationary* time series with *no unit root*.

A *true negative* test result is that the *null hypothesis* is `TRUE` (pricev have a unit root), while a *true positive* result is that the *null hypothesis* is `FALSE` (pricev are stationary).

The function `tseries::adf.test()` assumes that the data is *normally distributed*, which may underestimate the standard errors of the parameters, and produce *false positives* (*type I* errors) by incorrectly rejecting the null hypothesis of a unit root process.




```
> # Simulate AR(1) process with different coefficients
> coeffv <- seq(0.99, 0.999, 0.001)
> retp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> adft <- sapply(coeffv, function(coeff) {
+   arimav <- filter(x=retp, filter=coeff, method="recursive")
+   adft <- suppressWarnings(tseries::adf.test(arimav))
+   c(adfstat=unname(adft$statistic), pval=adft$p.value)
+ })  # end sapply
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # x11(width=6, height=4)
> plot(x=coeffv, y=adft["pval", ], main="ADF p-val Versus AR Coeffic
+      xlab="AR coefficient", ylab="ADF pval", t="l", col="blue", lw
> plot(x=coeffv, y=adft["adfstat", ], main="ADF Stat Versus AR Coeff
+      xlab="AR coefficient", ylab="ADF stat", t="l", col="blue", lw
```

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.

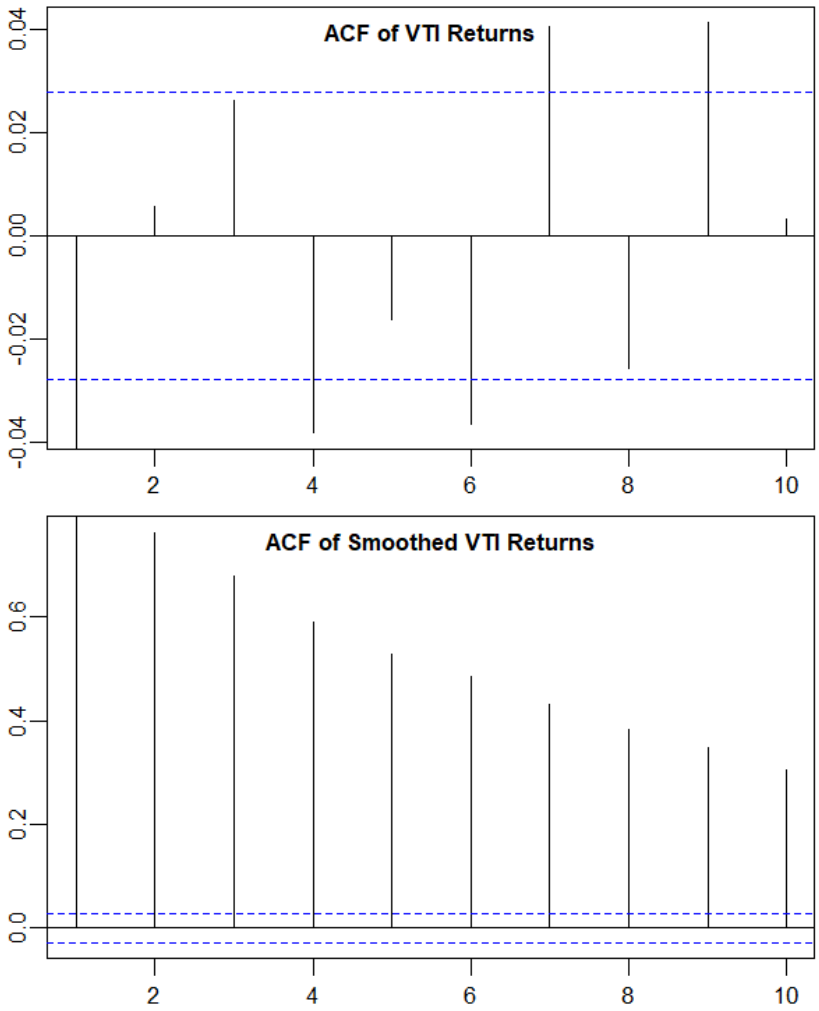

```
> # Extract log VTI prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> nrows <- NROW(closep)
> # Calculate EMA prices using HighFreq::run_mean()
> pricema <- HighFreq::run_mean(closep, lambda=0.9)
> # Combine prices with EMA prices
> pricev <- cbind(closep, pricema)
> colnames(pricev)[2] <- "VTI EMA"
> # Calculate standard deviations of returns
> sapply(rutils::diffit(pricev), sd)
```

```
> # Plot dygraph
> dygraphs::dygraph(pricev["2009"], main="VTI Prices and EMA Prices"
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Autocorrelations of Smoothed Time Series

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

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



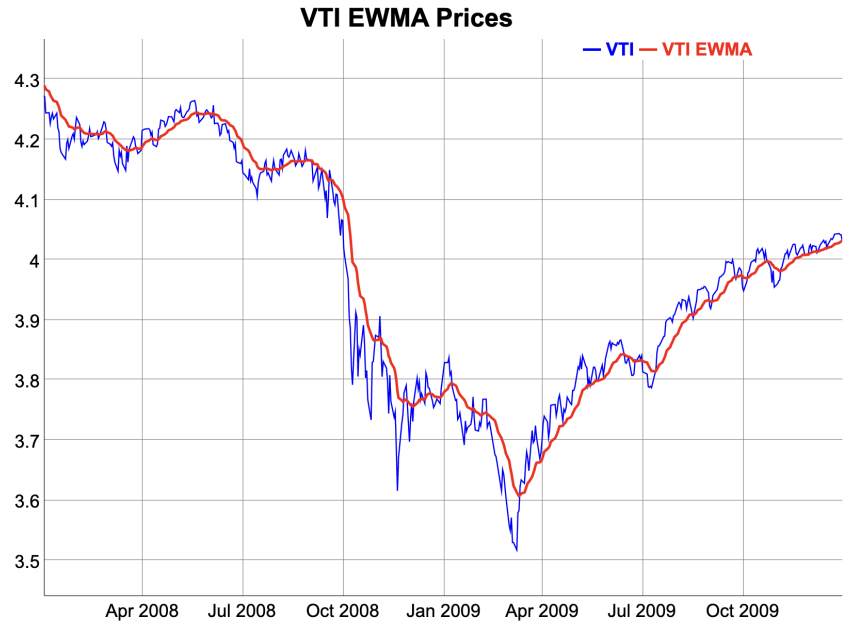
EMA Price Technical Indicator

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

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

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

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




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

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

Recursive EMA Price Indicator

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

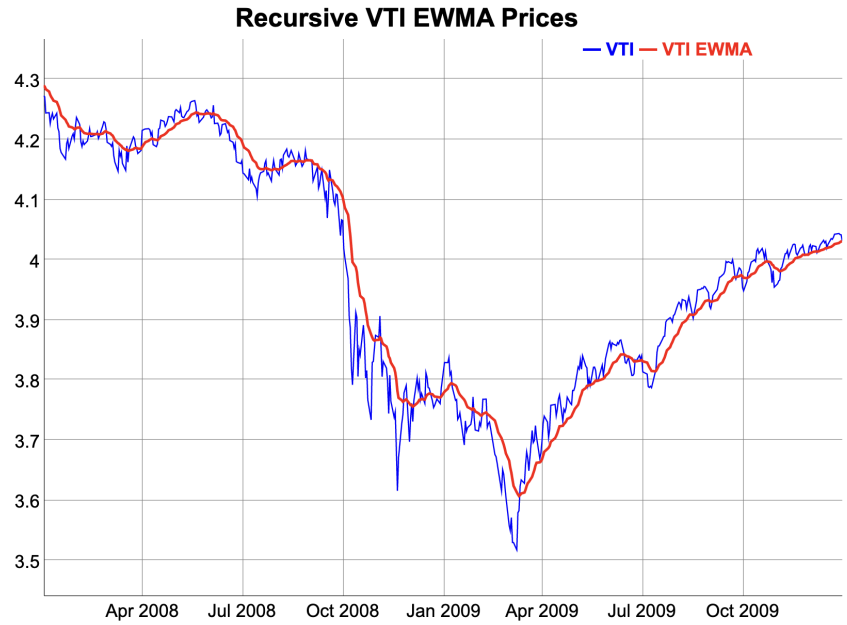
$$p_t^{EMA} = \lambda p_{t-1}^{EMA} + (1 - \lambda) p_t$$

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

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

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

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




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

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

Volume-Weighted Average Price Indicator

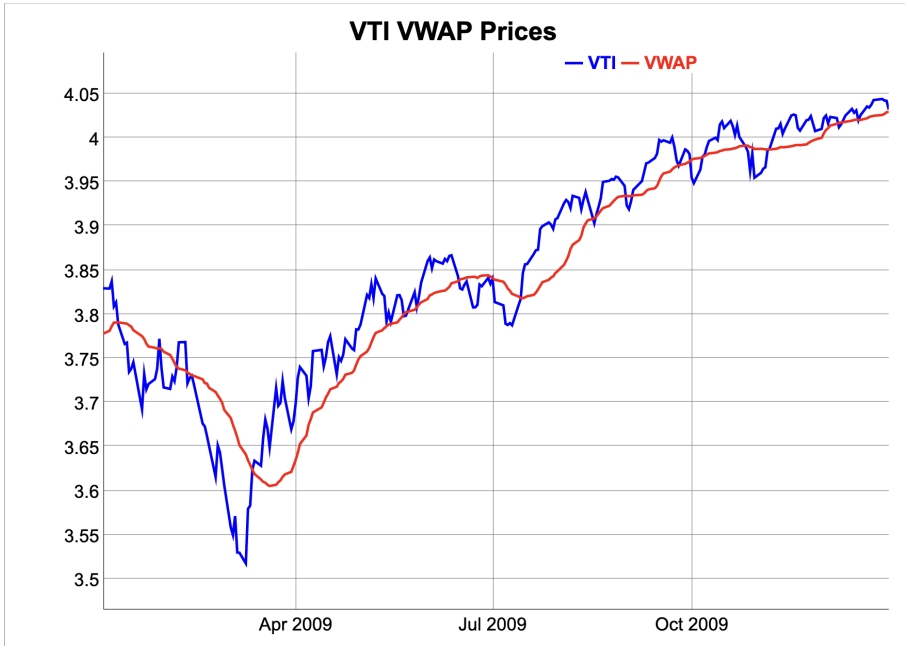
The Volume-Weighted Average Price (*VWAP*) is defined as the sum of prices multiplied by trading volumes, divided by the sum of volumes:

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

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

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

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




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

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

Recursive VWAP Price Indicator

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

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

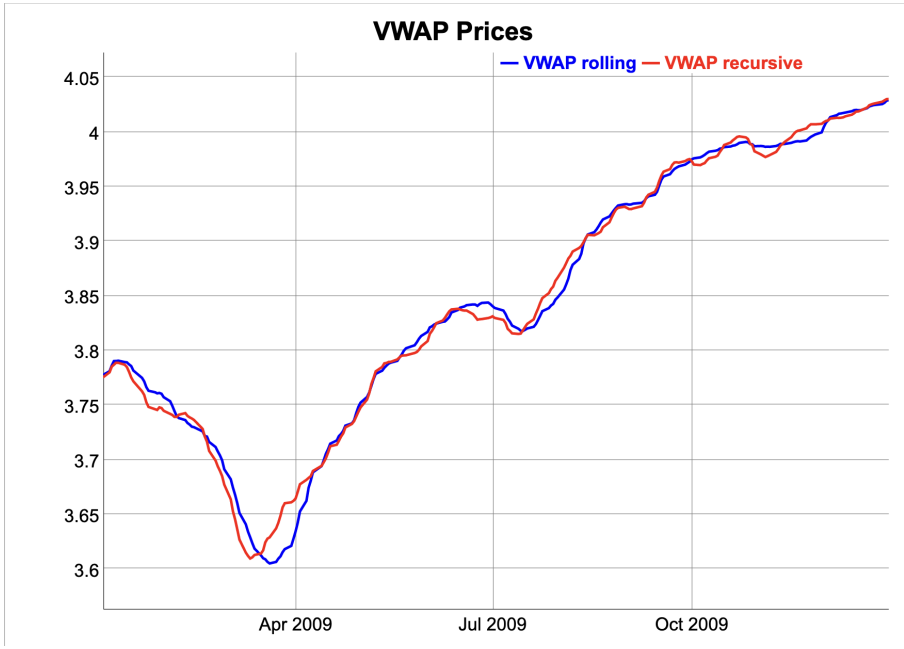
The recursive *VWAP* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

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

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

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

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




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

Smooth Asset Returns

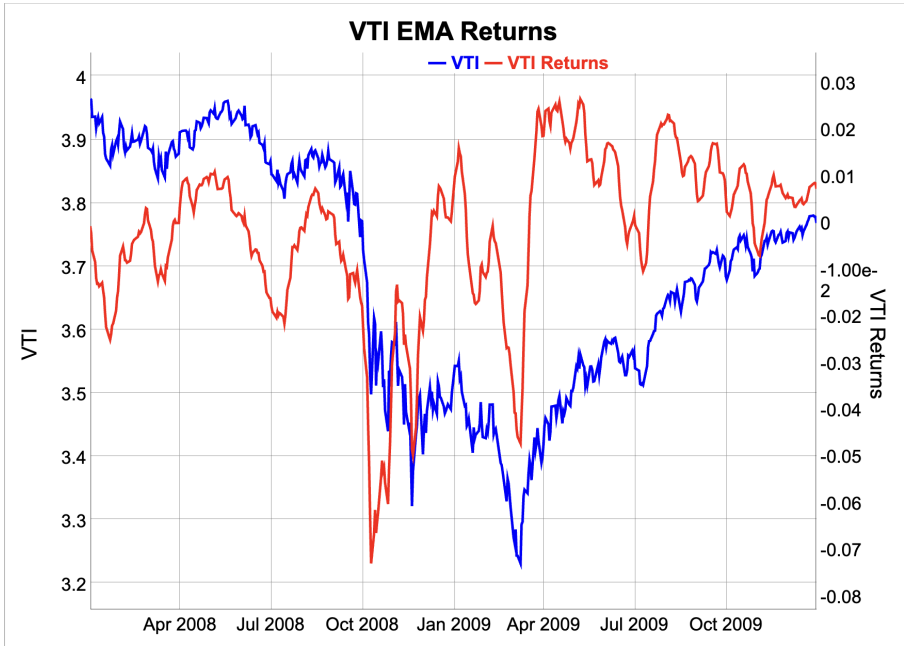
Asset returns are calculated by filtering prices through a *differencing* filter.

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

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

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

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




```
> # Calculate two EMA prices
> lambdaf <- 0.8 # Fast EMA
> emaf <- HighFreq::run_mean(closep, lambda=lambdaf)
> lambdas <- 0.9 # Slow EMAs
> emas <- HighFreq::run_mean(closep, lambda=lambdas)
```

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

Fractional Asset Returns

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).

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

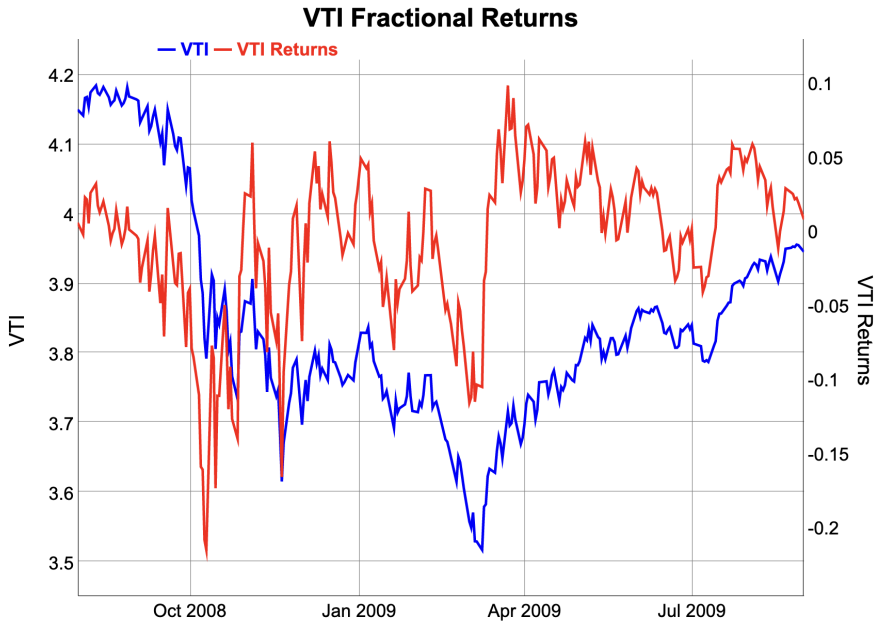
The simple returns can then be expressed as equal to the returns operator $(1 - L)$ applied to the prices: $r_t = (1 - L)p_t$.

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

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

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

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




```
> # Calculate the fractional weights
> lookb <- 21
> deltav <- 0.1
> weightv <- (deltav - 0:(lookb-2)) / 1:(lookb-1)
> weightv <- (-1)^(1:(lookb-1))*cumprod(weightv)
> weightv <- c(1, weightv)
> weightv <- (weightv - mean(weightv))
```

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

Augmented Dickey-Fuller Test for Asset Returns

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

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

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

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

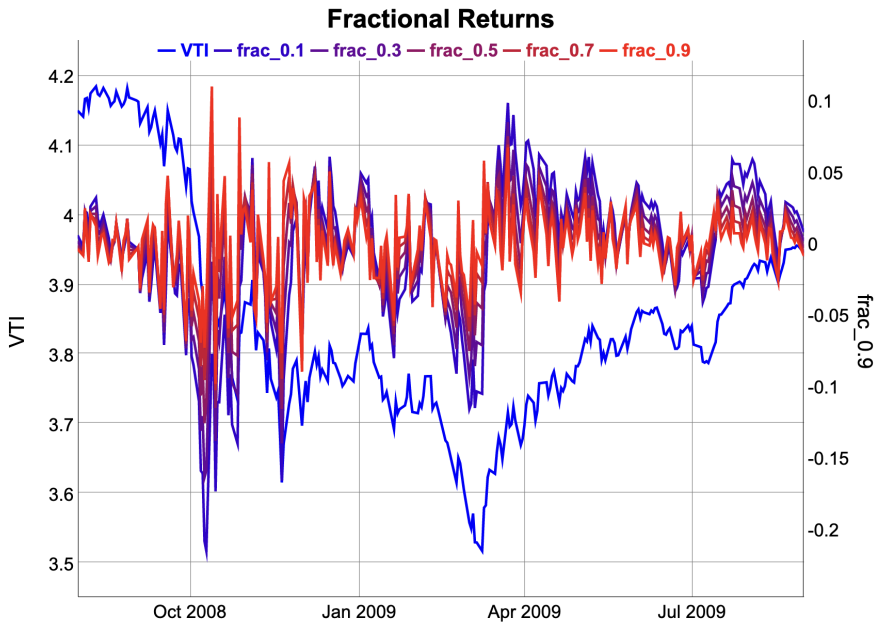
The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

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

Augmented Dickey-Fuller Test for Fractional Returns

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

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




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

Trading Volume Z-Scores

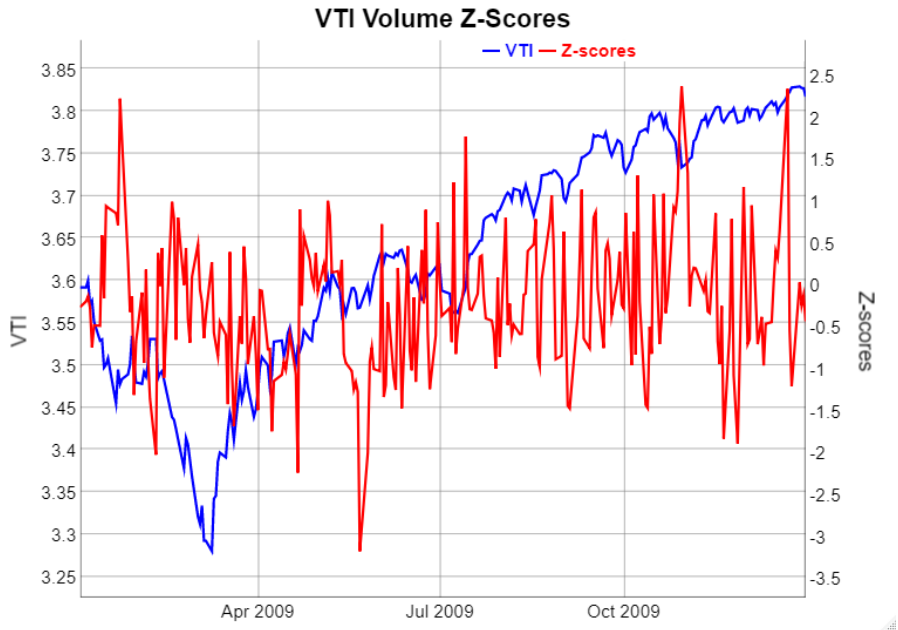
The trailing *volume z-score* is equal to the volume v_t minus the trailing average volumes $\bar{v}_t$ divided by the volatility of the volumes σ_t:

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

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

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

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




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

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

Volatility Z-Scores

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

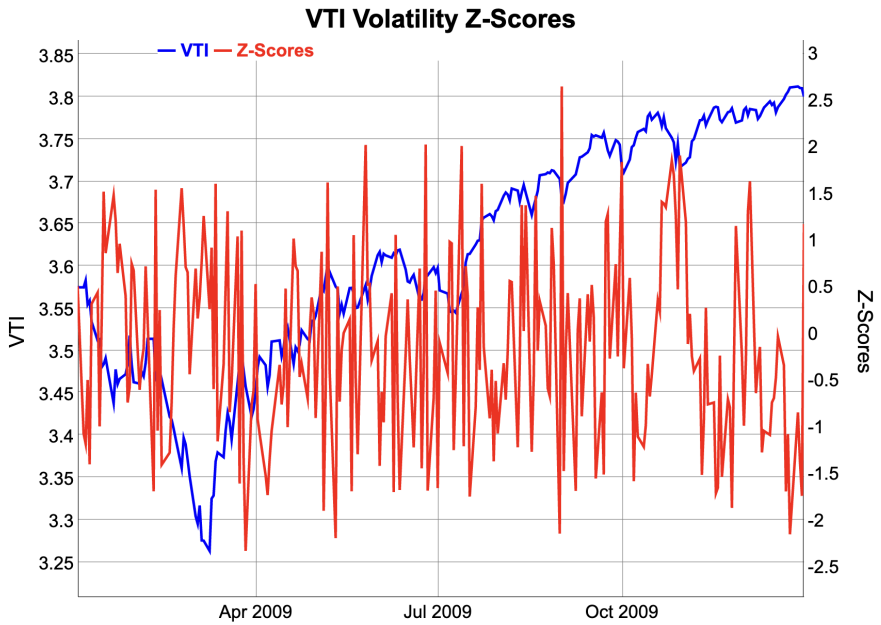
The *volatility z-score* is equal to the spot volatility v_t minus the trailing average volatility $\bar{v}_t$ divided by the standard deviation of the volatility σ_t:

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

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

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

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




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

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

Trailing Volatility Z-Scores

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

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

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

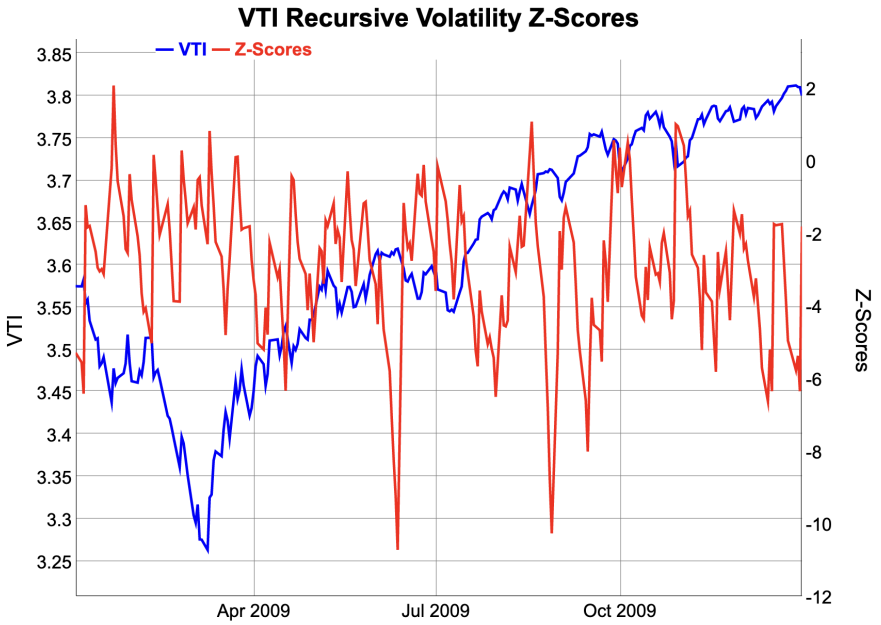
$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$

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

Where $\bar{r}_t$ and σ_t^2 are the trailing mean and variance at time t.

The decay factor λ determines how quickly the mean and variance estimates are updated, with smaller values of λ producing faster updating, giving more weight to recent prices, and vice versa.

```
> # Calculate the recursive trailing VTI volatility
> lambdaf <- 0.8 # Fast lambda
> lambdas <- 0.81 # Slow lambda
> volatf <- sqrt(HighFreq::run_var(retp, lambda=lambdaf))[, 2]
> volats <- sqrt(HighFreq::run_var(retp, lambda=lambdas))[, 2]
```




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

Centered Price Z-Scores

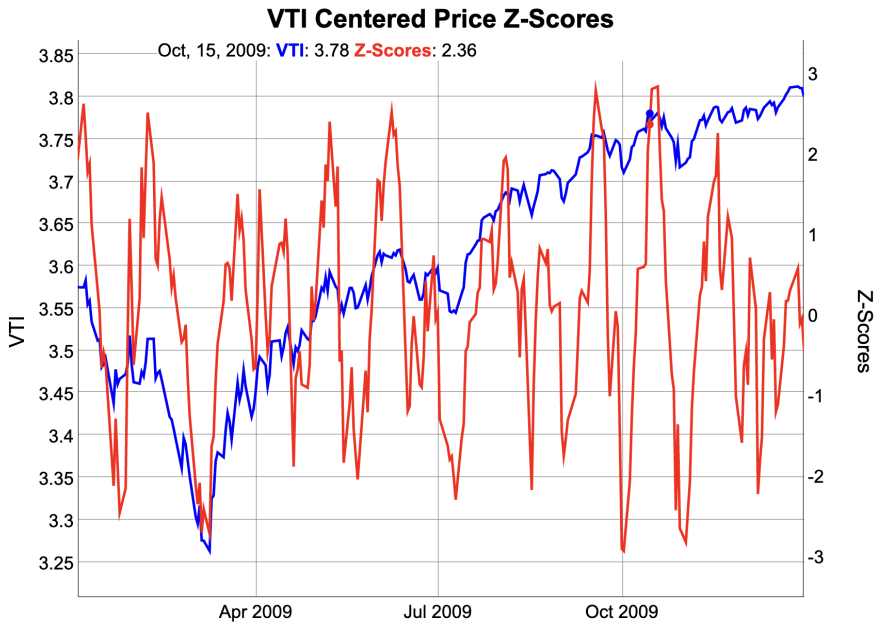
An extreme local price is a price which differs significantly from neighboring prices.

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

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

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

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




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

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

Labeling the Tops and Bottoms of Prices

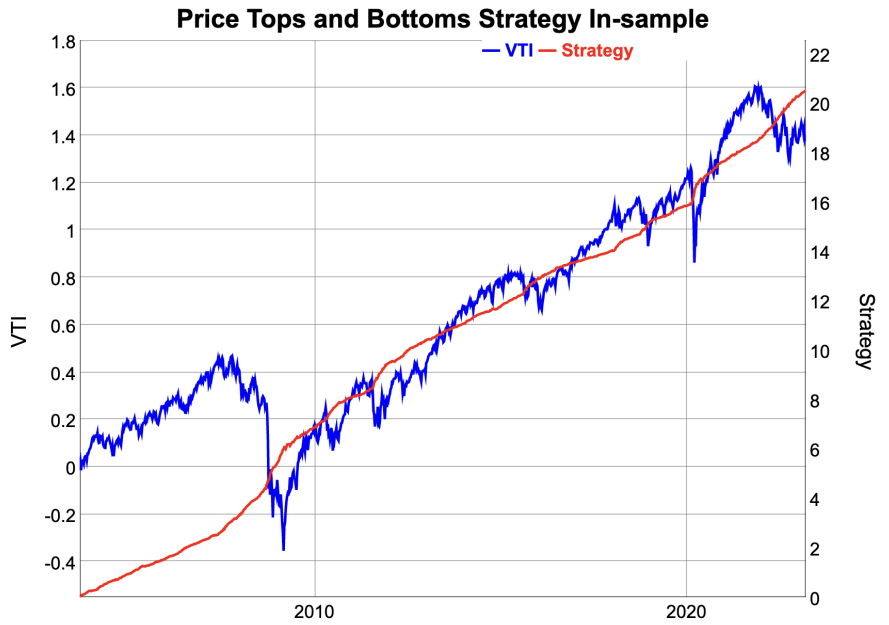
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

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

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

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

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




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

Trailing Price Z-Scores

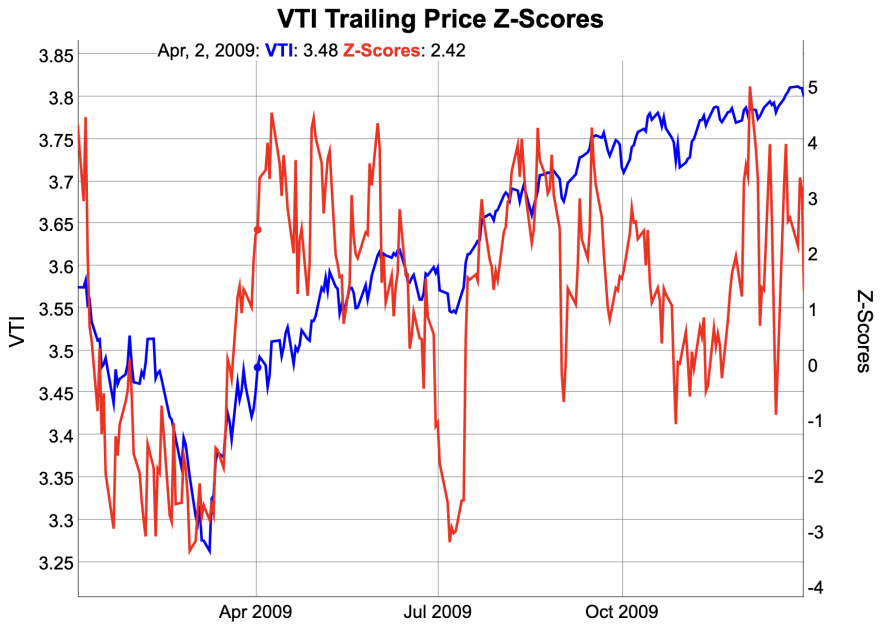
The trailing price z-score is equal to the difference between the current price p_t minus the trailing average price $\bar{p}_{t-k}$, divided by the volatility of the price σ_t:

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

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

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

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




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

Recursive Trailing Price Z-Scores

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

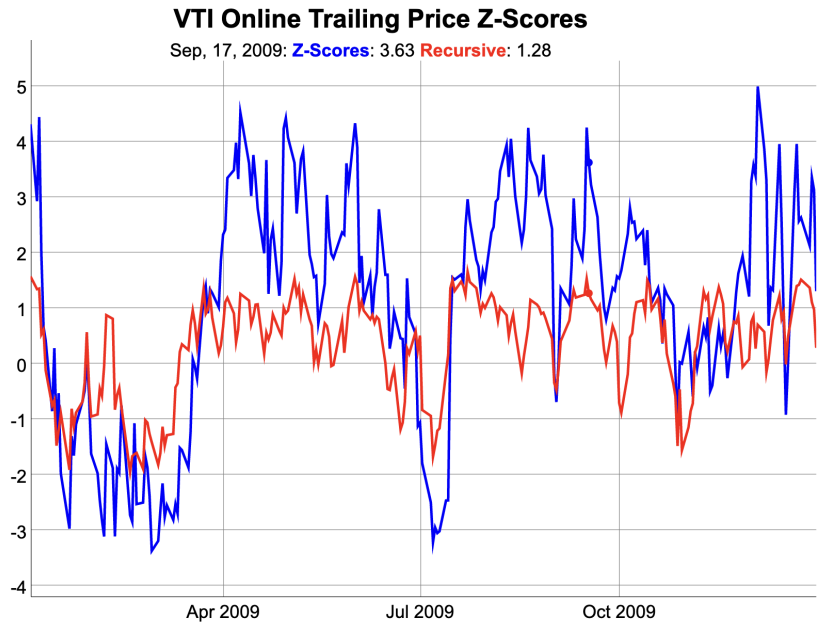
$$z_t = \frac{p_t - \bar{p}_t}{\sigma_t}$$

The function `HighFreq::run_var()` calculates the trailing mean and variance of the prices p_t, by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the prices minus their trailing means $(p_t - \bar{p}_t)^2$, using the decay factor λ:

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

Where $\bar{p}_t$ and σ^2_t are the trailing mean and variance at time t.

The decay factor λ determines how quickly the mean and variance estimates are updated, with smaller values of λ producing faster updating, giving more weight to recent prices, and vice versa. If λ is close to 1 then the decay is weak and past prices have a greater weight, and the trailing mean values have a stronger dependence on past prices. This is equivalent to a long look-back interval.
And vice versa if λ is close to 0.




```
> # Calculate the EMA returns and volatilities
> lambdaf <- 0.9
> volv <- HighFreq::run_var(closep, lambda=lambdaf)
> # Calculate the recursive trailing z-scores of VTI prices
> pricer <- (closep - volv[, 1])
> pricer <- ifelse(volv > 0, pricer/volv, 0)
> volv <- sqrt(volv[, 2])
> # Plot dygraph of the trailing z-scores of VTI prices
> pricev <- xts::xts(cbind(pricez, pricer), datev)
> colnames(pricev) <- c("Z-Scores", "Recursive")
> colv <- colnames(pricev)
> dygraphs::dygraph(pricev["2009"], main="VTI Online Trailing Price
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Trailing Regression Z-Scores

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

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

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

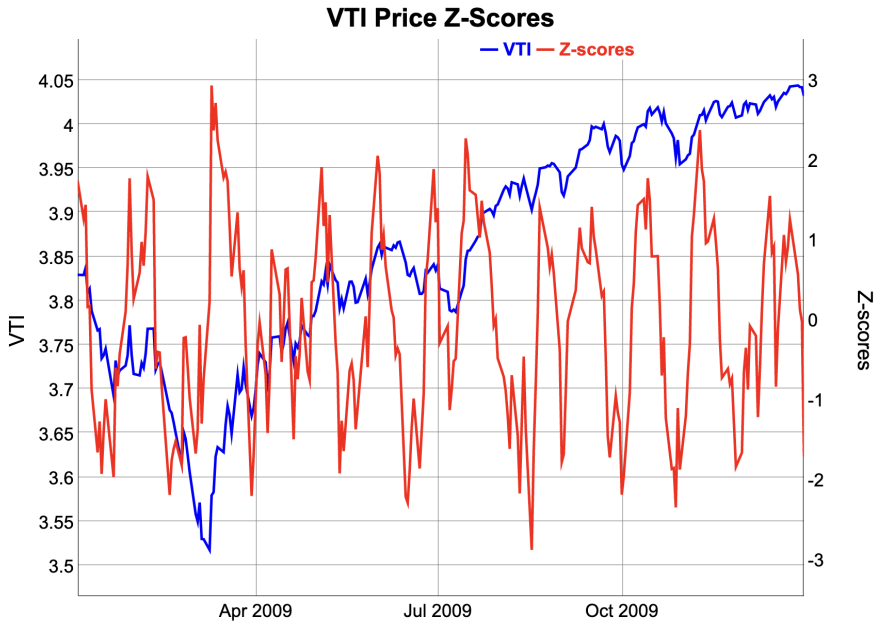
Where p_t^{fit} are the fitted values, α_t and β_t are the *regression coefficients*, and σ_t is the standard deviation of the residuals.

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

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

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

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




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

Recursive Trailing Regression

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

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

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

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

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

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

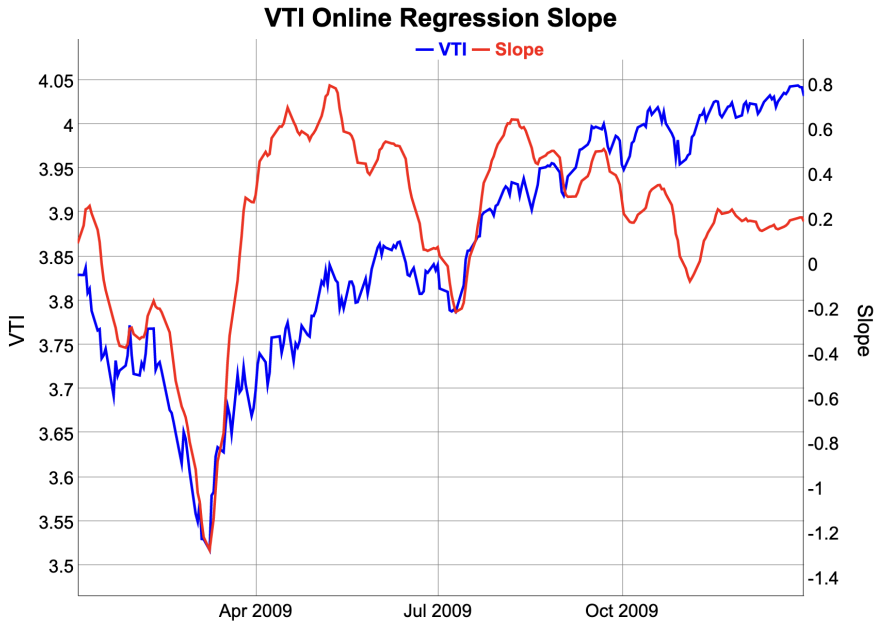
The covariance matrices are updated using the following recursive (online) formulas:

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

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

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

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



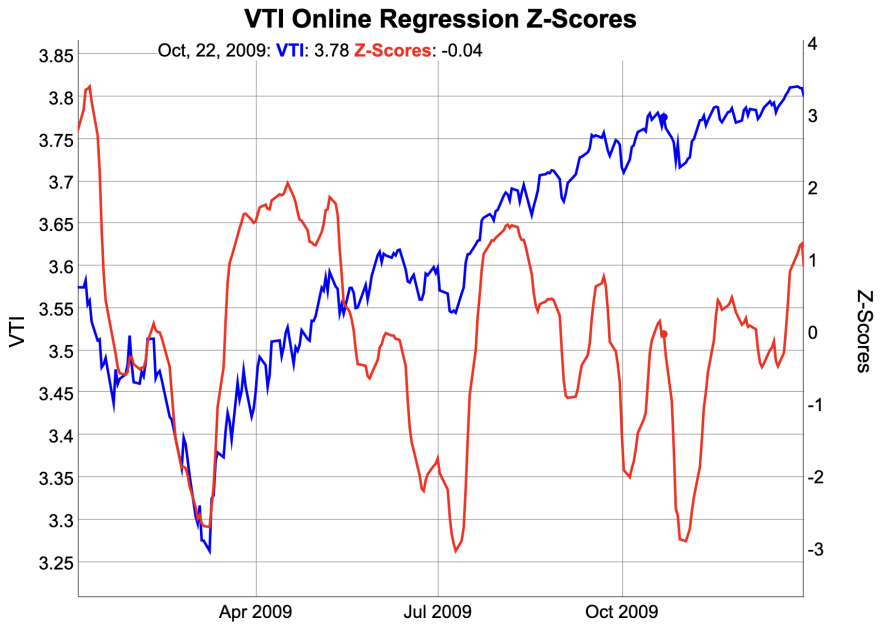

```
> # Plot dygraph of regression betas
> datav <- cbind(closep, 252*regs[, "beta"])
> colnames(datav) <- c("VTI", "Slope")
> colv <- colnames(datav)
> dygraphs::dygraph(datav["2009"], main="VTI Online Regression Slope
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2],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

Recursive Trailing Regression Z-Scores

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

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

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




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

The Hampel Filter

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

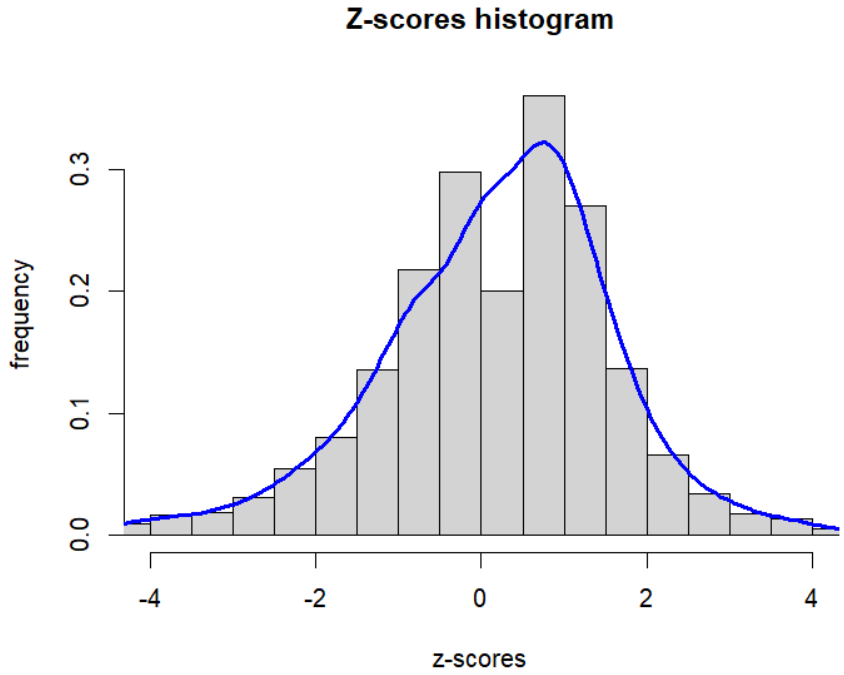
$$\text{MAD} = \text{median}(\text{abs}(p_t - \text{median}(\mathbf{p})))$$

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

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

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

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




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

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

One-sided and Two-sided Data Filters

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

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

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

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

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

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

```
> # Calculate one-sided Hampel z-scores
> medianv <- HighFreq::roll_mean(closep, lookb=lookb, method="nonpar
> madv <- HighFreq::roll_var(closep, lookb=lookb, method="nonparamet
> zscores <- (closep - medianv)/madv
> zscores[1:lookb, ] <- 0
> tail(zscores, lookb)
> range(zscores)
> # Calculate two-sided Hampel z-scores
> halfb <- lookb %/% 2
> medianv <- rutils::lagit(medianv, lagg=(-halfb))
> madv <- rutils::lagit(madv, lagg=(-halfb))
> zscores <- (closep - medianv)/madv
> zscores[1:lookb, ] <- 0
> tail(zscores, lookb)
> range(zscores)
```

Calculating the Trailing Variance of Asset Returns

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

The trailing variance of returns is given by:

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

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

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

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

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

Calculating the Trailing Variance Using Package *roll*

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

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

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

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

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

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

Trailing *EMA* Realized Volatility Estimator

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

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

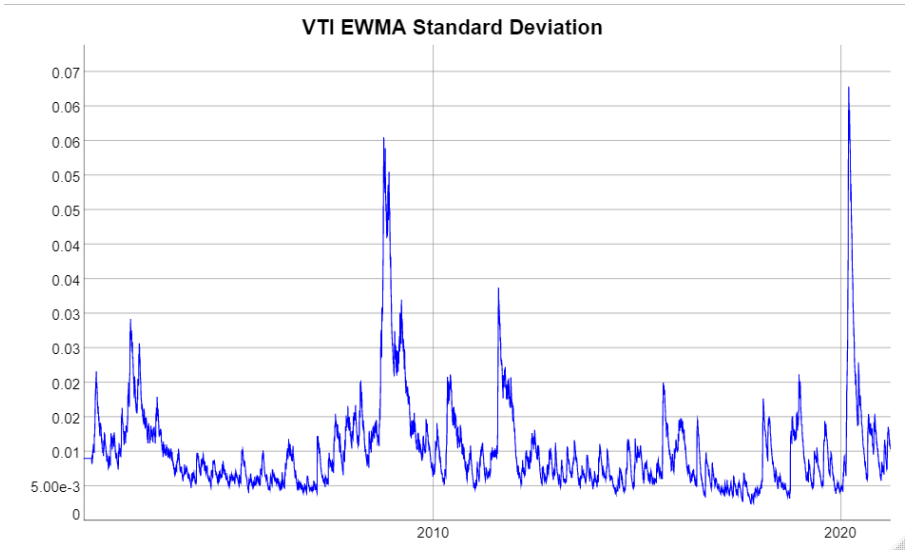
$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2 = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j r_{t-j}^2$$

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

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

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

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




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

Estimating *EMA* Variance Using Package *roll*

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

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

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

Where w_j is the vector of exponentially decaying weights:

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

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

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

Trailing Realized Volatility Estimator

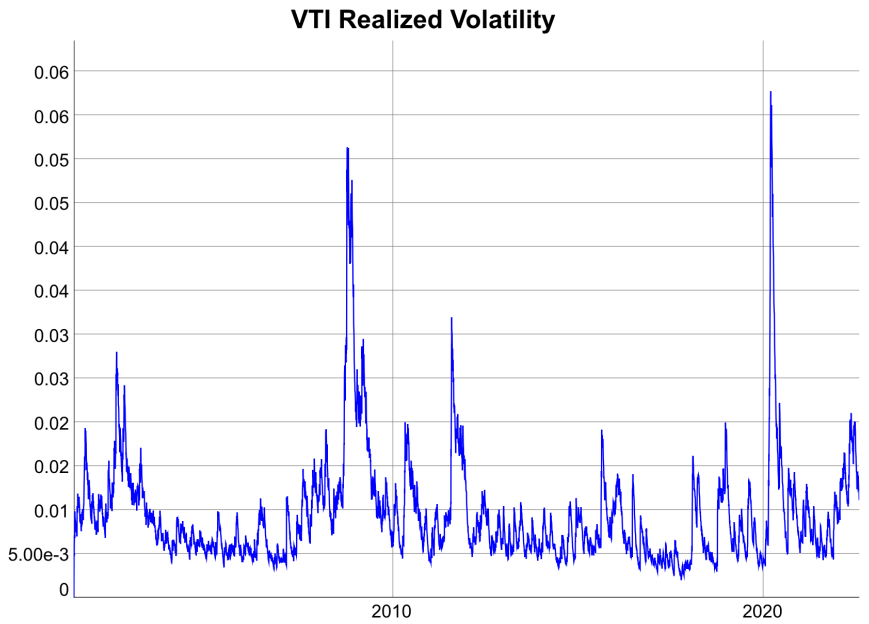
The function `HighFreq::run_var()` calculates the trailing mean and variance of the returns r_t, by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the returns minus their trailing means $(r_t - \bar{r}_t)^2$, using the decay factor λ:

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

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

Where $\bar{r}_t$ and σ^2_t are the trailing mean and variance at time t.

The decay factor λ determines how quickly the mean and variance estimates are updated, with smaller values of λ producing faster updating, giving more weight to recent prices, and vice versa.




```
> # Calculate realized variance recursively
> lambdaf <- 0.9
> volv <- HighFreq::run_var(retp, lambda=lambdaf)
> volv <- sqrt(volv[, 2])
> # Plot EMA volatility
> volv <- xts:::xts(volv, order.by=datev)
> dygraphs::dygraph(volv, main="VTI Realized Volatility") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
```

Estimating Daily Volatility From Intraday Returns

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

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

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

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

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

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

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

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

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

Range Volatility Estimators of *OHLC* Time Series

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

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

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

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

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

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

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

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

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

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

Calculating the Trailing Range Variance Using *HighFreq*

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

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

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

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

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

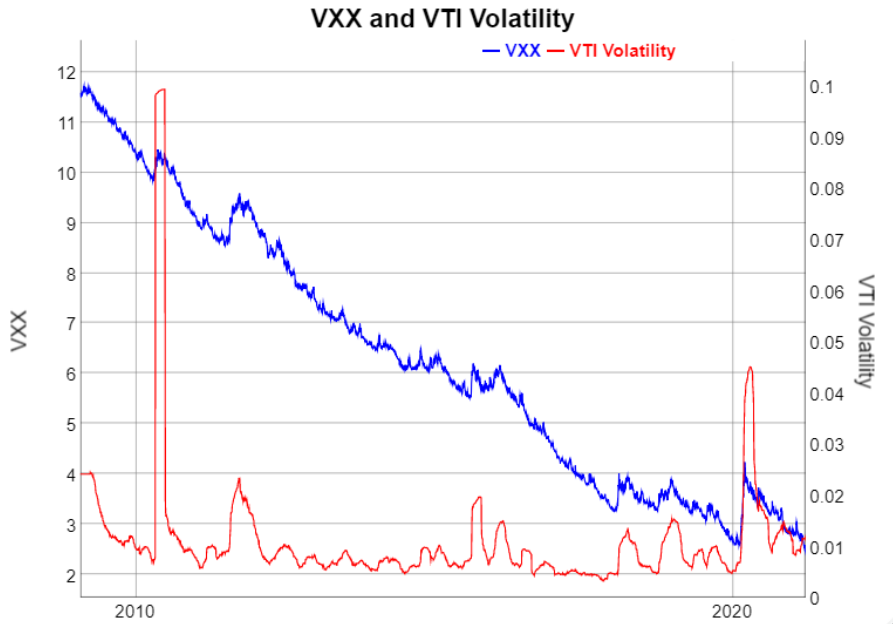
```
> library(HighFreq)  # Load HighFreq
> spy <- HighFreq::SPY["2008/2009"]
> # Calculate daily SPY volatility using package HighFreq
> sqrt(6.5*60*HighFreq::calcvar_ohlc(log(spy),
+   method="yang_zhang"))
> # Calculate daily SPY volatility from minutely prices using packag
> sqrt((6.5*60)*mean(na.omit(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

VXX Prices and the Trailing Volatility

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

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

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



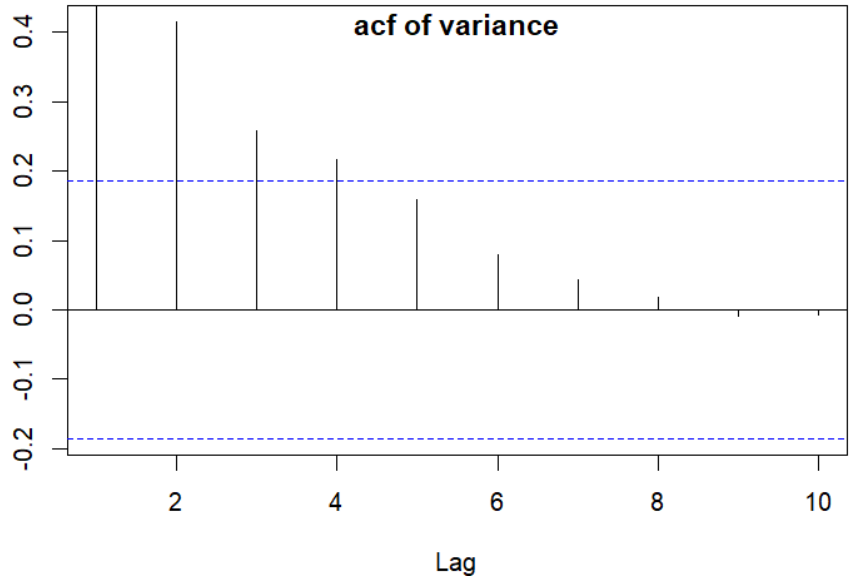

```
> # Calculate VXX log prices
> vxx <- na.omit(rutils::etfenv$prices$VXX)
> datev <- zoo::index(vxx)
> lookb <- 41
> vxx <- log(vxx)
> # Calculate trailing VTI volatility
> closep <- get("VTI", rutils::etfenv)[datev]
> closep <- log(closep)
> volv <- sqrt(HighFreq::roll_var_ohlc(ohlc=closep, lookb=lookb,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

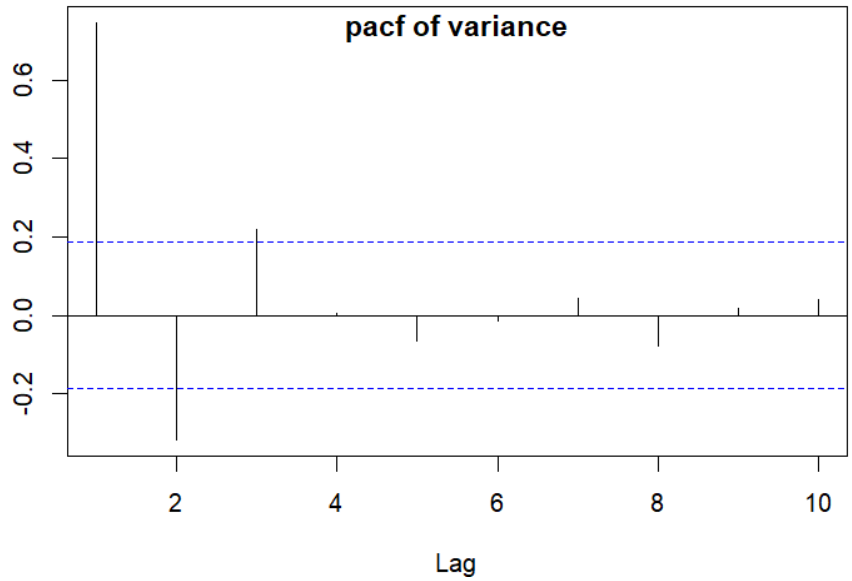
```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volv)
> colnames(datav)[2] <- "VTI Volatility"
> colv <- colnames(datav)
> captiont <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=captiont) %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2",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

Autocorrelation of Volatility

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

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





The *GARCH* Volatility Model

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

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

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

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

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

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

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

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

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

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

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

GARCH Volatility Time Series

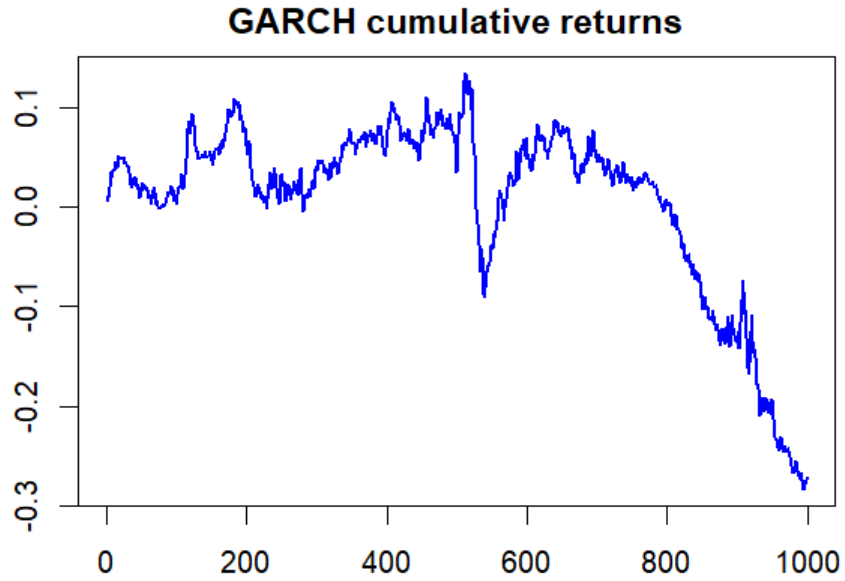
The *GARCH* volatility model produces volatility clustering - periods of high volatility followed by a quick decay.

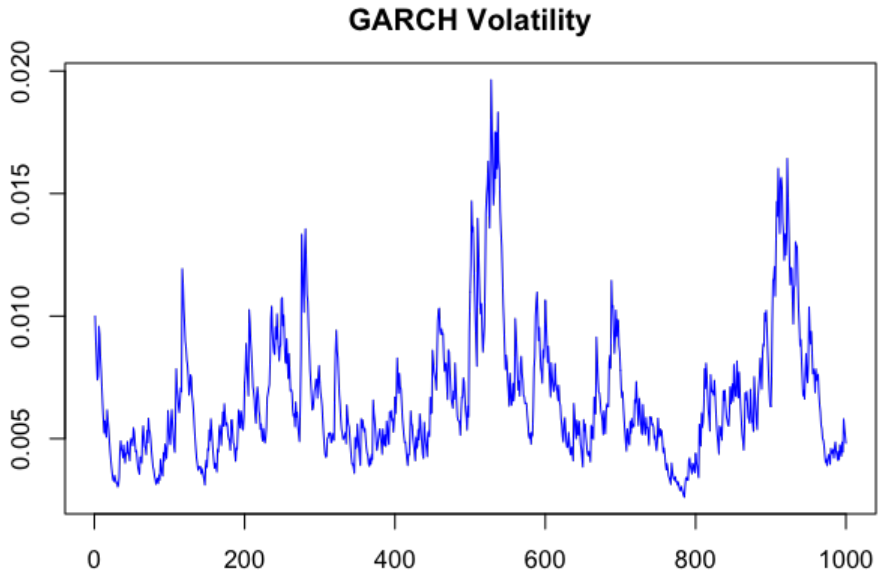
But the decay of the volatility in the *GARCH* model is faster than what is observed in practice.

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

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

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





GARCH Returns Distribution

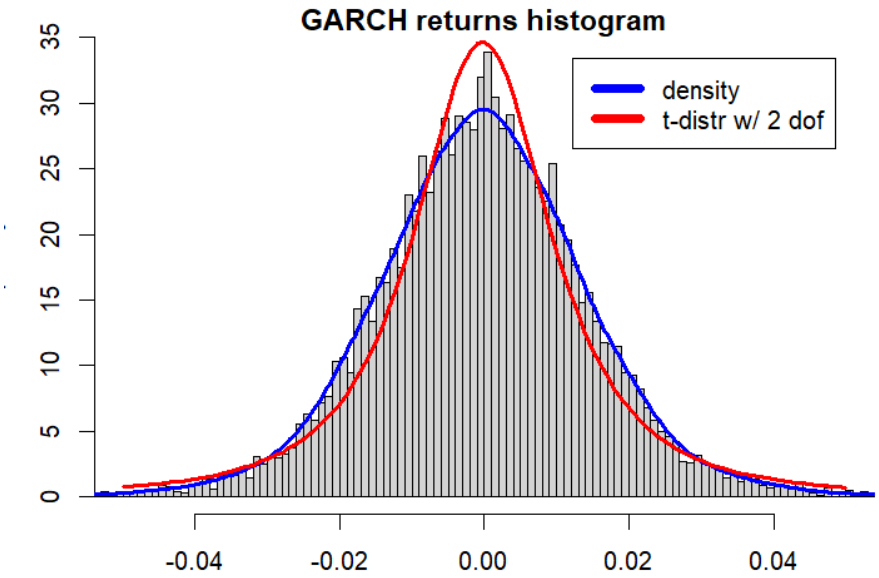
The *GARCH* volatility model produces *leptokurtic* returns with fat tails in their the distribution.

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

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

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

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




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

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

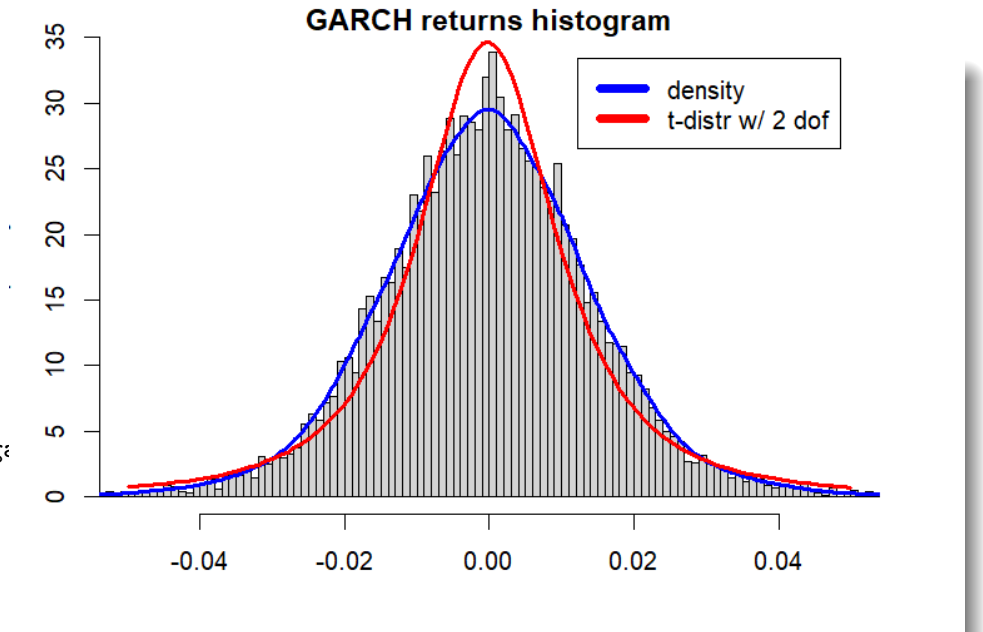
GARCH Model Simulation

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

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

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

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




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

GARCH Returns Kurtosis

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

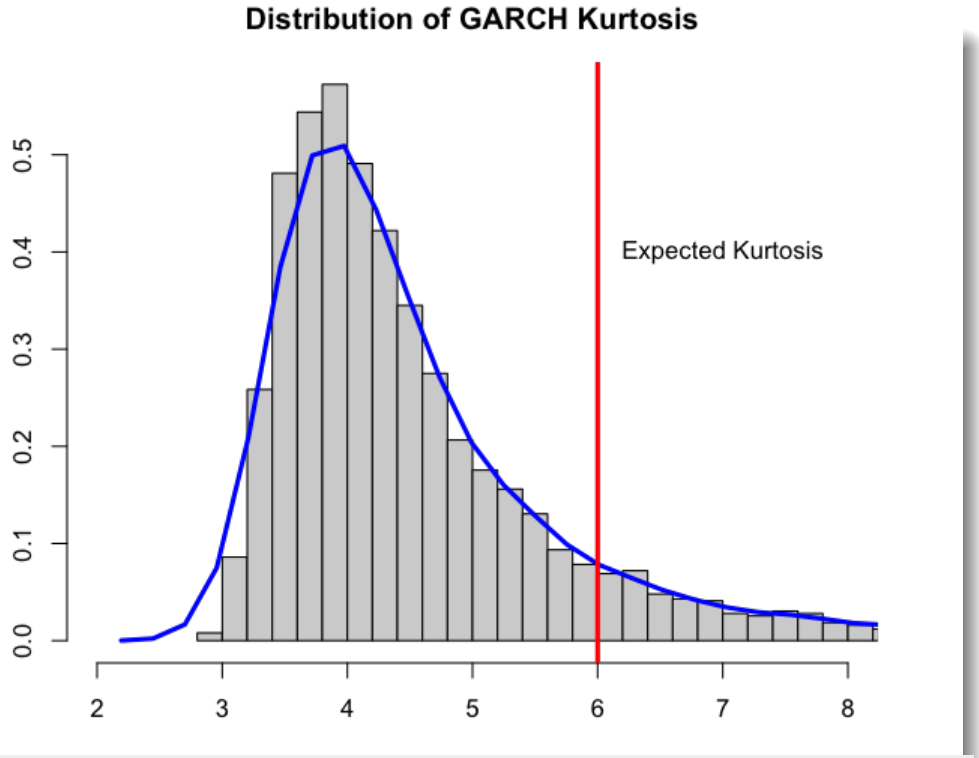
$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

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

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

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

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




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

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

GARCH Variance Estimation

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

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

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

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

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

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

GARCH Model Calibration

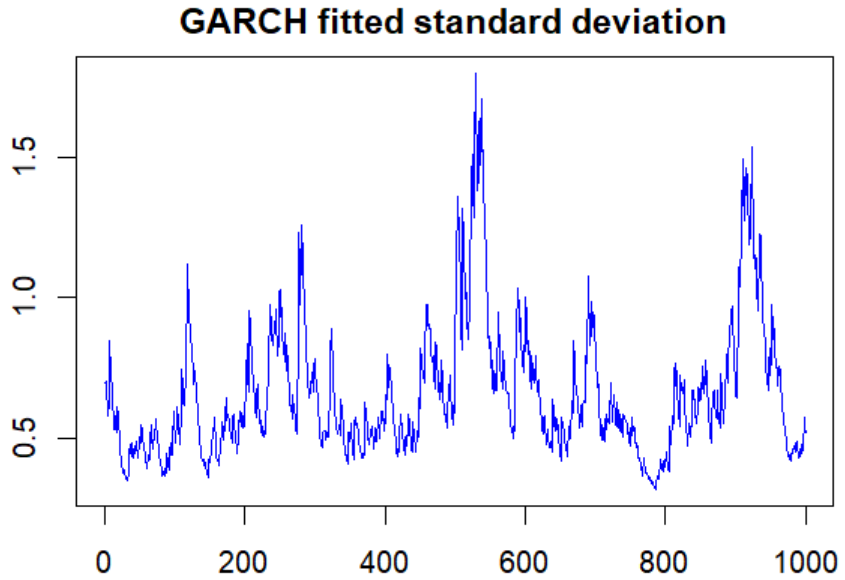
GARCH models can be calibrated from the returns using the *maximum-likelihood* method.

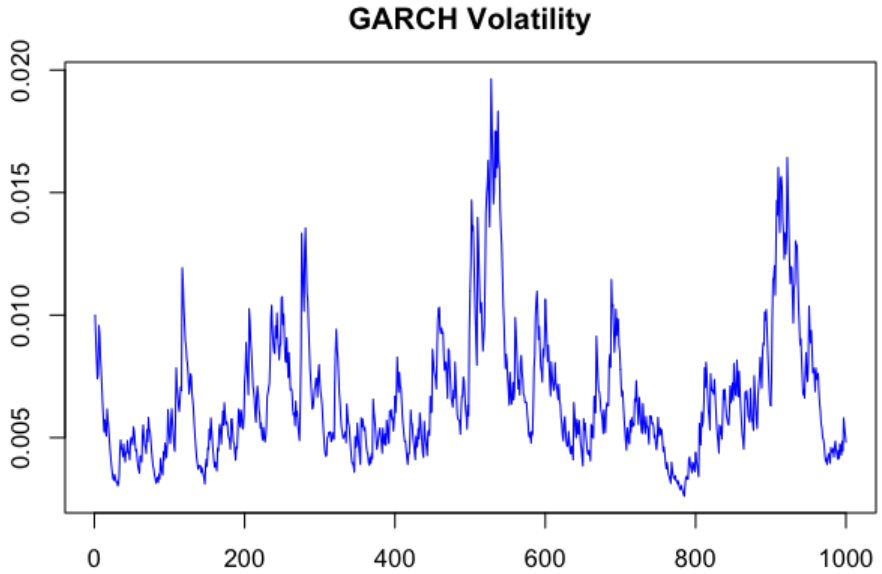
But it's a complex optimization procedure which requires a large amount of data for accurate results.

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

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

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





GARCH Likelihood Function

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

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

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

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

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

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

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

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

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

GARCH Likelihood Function Matrix

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

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

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

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

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

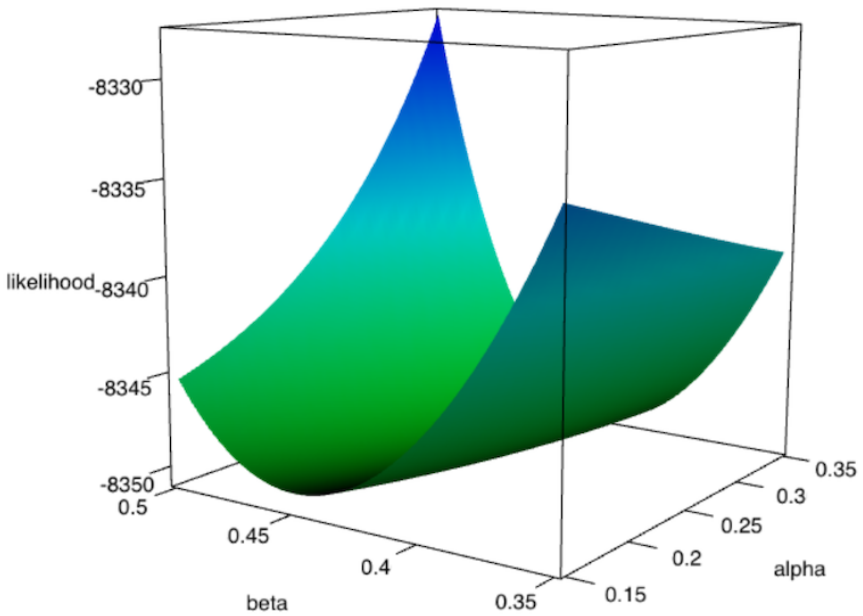
```
> # Calculate the variance of returns
> retp <- garchsim[, 1, drop=FALSE]
> varv <- var(retp)
> retp <- (retp - mean(retp))
> # Calculate likelihood as function of alpha and betac parameters
> likefun <- function(alphac, betac) {
+   omega <- variance*(1 - alpha - betac)
+   -HighFreq::lik_garch(omega=omega,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

GARCH Likelihood Perspective Plot

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

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

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




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

GARCH Likelihood Function Optimization

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

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

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

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

The first generation of solutions is selected randomly.

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

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

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

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

GARCH Variance of Stock Returns

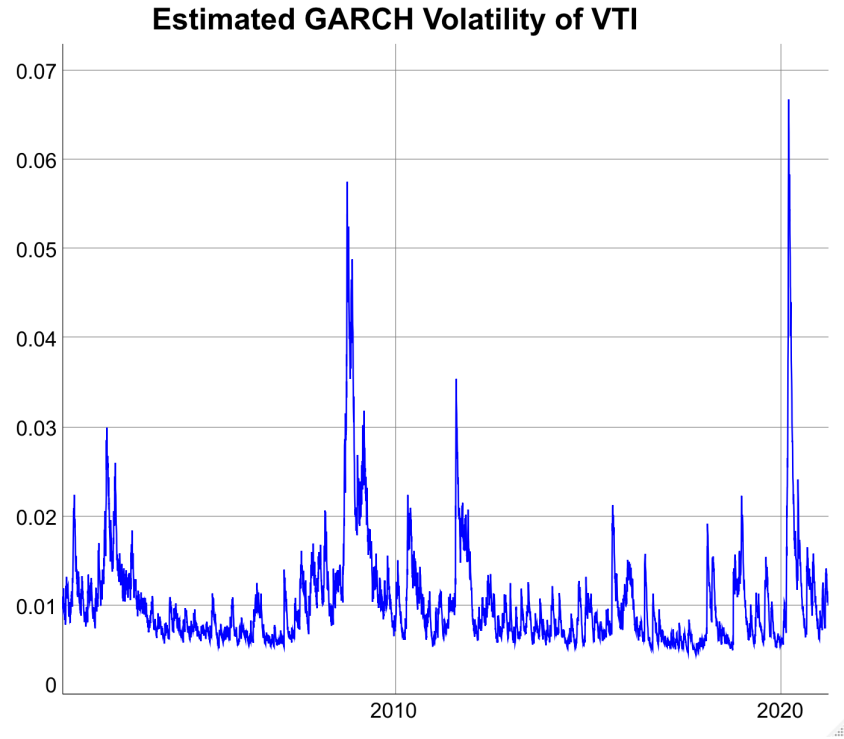
The *GARCH* model can be used to estimate the trailing variance of empirical (historical) returns.

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

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

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

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




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

GARCH Variance Forecasts

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

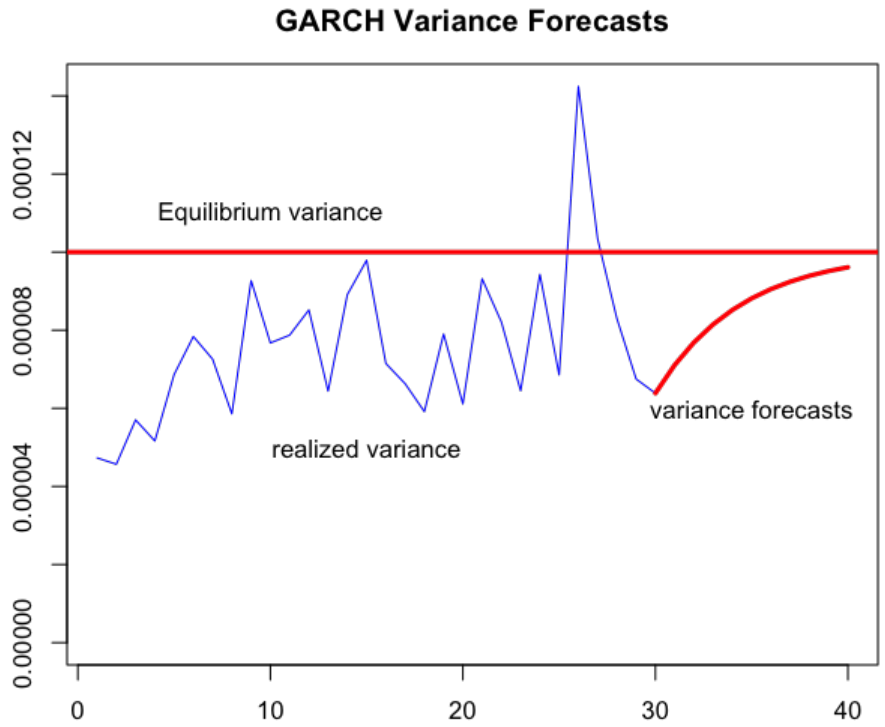
So the variance forecasts depend on the variance in the previous period:
$\sigma_{t+1}^2 = \mathbb{E}[\omega + \alpha r_{t+1}^2 + \beta \sigma_t^2] = \omega + (\alpha + \beta)\sigma_t^2$

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

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

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

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




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

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

Package *Rcpp* for Calling C++ Programs from R

The package *Rcpp* allows calling C++ functions from R, by compiling the C++ code and creating R functions.

Rcpp functions are R functions that were compiled from C++ code using package *Rcpp*.

Rcpp functions are much faster than code written in R, so they're suitable for large numerical calculations.

The package *Rcpp* relies on *Rtools* for compiling the C++ code:

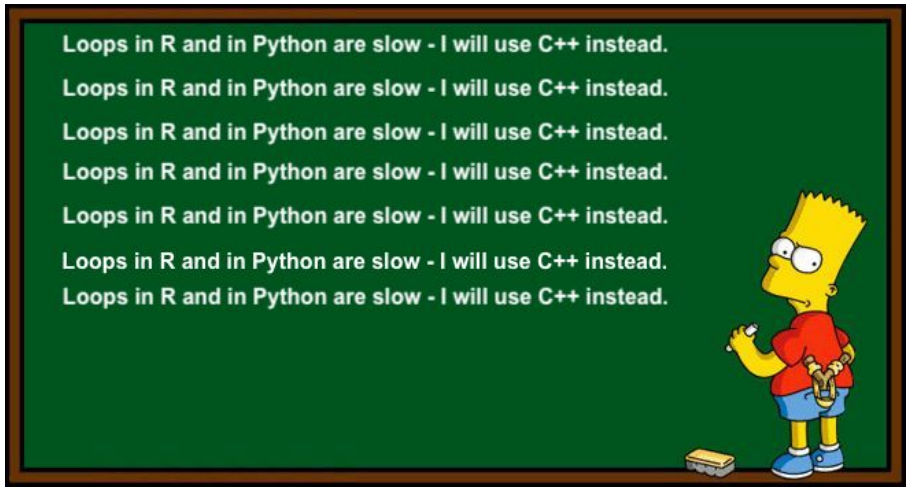
https://cran.r-project.org/bin/windows/Rtools/

You can learn more about the package *Rcpp* here:

http://adv-r.had.co.nz/Rcpp.html

http://www.rcpp.org/

http://gallery.rcpp.org/




```
> # Verify that Rtools or XCode are working properly:
> devtools::find_rtools()  # Under Windows
> devtools::has_devel()
> # Install the packages Rcpp and RcppArmadillo
> install.packages(c("Rcpp", "RcppArmadillo"))
> # Load package Rcpp
> library(Rcpp)
> # Get documentation for package Rcpp
> # Get short description
> packageDescription("Rcpp")
> # Load help page
> help(package="Rcpp")
> # List all datasets in "Rcpp"
> data(package="Rcpp")
> # List all objects in "Rcpp"
> ls("package:Rcpp")
> # Remove Rcpp from search path
> detach("package:Rcpp")
```

Function cppFunction() for Compiling C++ code

The function cppFunction() compiles C++ code into an R function.

The function cppFunction() creates an R function only for the current R session, and it must be recompiled for every new R session.

The function sourceCpp() compiles C++ code contained in a file into R functions.

```
> # Define Rcpp function
> Rcpp::cppFunction("
+   int times_two(int x)
+     { return 2 * x;}
+   ")  # end cppFunction
> # Run Rcpp function
> times_two(3)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Multiply two numbers
> mult_rcpp(2, 3)
> mult_rcpp(1:3, 6:4)
> # Multiply two vectors
> mult_vec_rcpp(2, 3)
> mult_vec_rcpp(1:3, 6:4)
```

Performing Loops in *Rcpp Sugar*

Loops written in *Rcpp* can be two orders of magnitude faster than loops in R!

Rcpp Sugar allows using R-style vectorized syntax in *Rcpp* code.

```
> # Define Rcpp function with loop
> Rcpp::cppFunction("
+ double inner_mult(NumericVector x, NumericVector y) {
+ int xsize = x.size();
+ int ysize = y.size();
+ if (xsize != ysize) {
+     return 0;
+   } else {
+     double total = 0;
+     for(int i = 0; i < xsize; ++i) {
+ total += x[i] * y[i];
+   }
+   return total;
+   }
+ }")  # end cppFunction
> # Run Rcpp function
> inner_mult(1:3, 6:4)
> inner_mult(1:3, 6:3)
> # Define Rcpp Sugar function with loop
> Rcpp::cppFunction("
+ double inner_sugar(NumericVector x, NumericVector y) {
+   return sum(x * y);
+ }")  # end cppFunction
> # Run Rcpp Sugar function
> inner_sugar(1:3, 6:4)
> inner_sugar(1:3, 6:3)
```

```
> # Define R function with loop
> inner_multr <- function(x, y) {
+     sumv <- 0
+     for(i in 1:NROW(x)) {
+ sumv <- sumv + x[i] * y[i]
+     }
+     sumv
+ }  # end inner_multr
> # Run R function
> inner_multr(1:3, 6:4)
> inner_multr(1:3, 6:3)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=inner_multr(1:10000, 1:10000),
+   innerp=1:10000 %*% 1:10000,
+   Rcpp=inner_mult(1:10000, 1:10000),
+   sugar=inner_sugar(1:10000, 1:10000),
+   times=10))[, c(1, 4, 5)]
```

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

Simulating the Ornstein-Uhlenbeck Process in *Rcpp* is about 30 times faster than in R!

```
> # Define Ornstein-Uhlenbeck function in R
> sim_our <- function(nrows=1000, priceq=5.0,
+               volat=0.01, theta=0.01) {
+   retp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- priceq
+   for (i in 2:nrows) {
+     retp[i] <- theta*(priceq - pricev[i-1]) + volat*rnorm(1)
+     pricev[i] <- pricev[i-1] + retp[i]
+   }  # end for
+   pricev
+ }  # end sim_our
> # Simulate Ornstein-Uhlenbeck process in R
> priceq <- 5.0; sigmav <- 0.01
> thetav <- 0.01; nrows <- 1000
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # R
> ousim <- sim_our(nrows, priceq=priceq, volat=sigmav, theta=theta
```

```
> # Define Ornstein-Uhlenbeck function in Rcpp
> Rcpp::cppFunction("
+ NumericVector sim_oucpp(double priceq,
+                   double volat,
+                   double thetav,
+                   NumericVector innov) {
+   int nrows = innov.size();
+   NumericVector pricev(nrows);
+   NumericVector retv(nrows);
+   pricev[0] = priceq;
+   for (int it = 1; it < nrows; it++) {
+     retv[it] = thetav*(priceq - pricev[it-1]) + volat*innov[it-1];
+     pricev[it] = pricev[it-1] + retv[it];
+   }  // end for
+   return pricev;
+ }")  # end cppFunction
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Res
> oucpp <- sim_oucpp(priceq=priceq,
+   volat=sigmav, theta=thetav, innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, priceq=priceq, volat=sigmav, theta=thetav),
+   Rcpp=sim_oucpp(priceq=priceq, volat=sigmav, theta=thetav, innov=
+   times=10))[, c(1, 4, 5)]
```

Rcpp Attributes

Rcpp attributes are instructions for the C++ compiler, embedded in the *Rcpp* code as C++ comments, and preceded by the "//" symbol.

The `Rcpp::depends` attribute specifies additional C++ library dependencies.

The `Rcpp::export` attribute specifies that a function should be exported to R, where it can be called as an R function.

Only functions which are preceded by the `Rcpp::export` attribute are exported to R.

The function `sourceCpp()` compiles C++ code contained in a file into R functions.

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namespace

// The function sim_oucpp() simulates an Ornstein-Uhlenb
// export the function roll_maxmin() to R
// [[Rcpp::export]]
NumericVector sim_oucpp(double priceq,
                        double volat,
                        double thetav,
                        NumericVector innov) {
  int(nrows = innov.size();
  NumericVector pricev*nrows);
  NumericVector retp*nrows);
  pricev[0] = priceq;
  for (int it = 1; it < nrows; it++) {
    retp[it] = thetav*(priceq - pricev[it-1]) + volat*in
    pricev[it] = pricev[it-1] + retp[it];
  }  // end for
  return pricev;
}  // end sim_oucpp
```

```
> # Source Rcpp function for Ornstein-Uhlenbeck process from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Reset random numbers
> oucpp <- sim_oucpp(priceq=priceq,
+   volat=sigmav,
+   theta=thetav,
+   innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, priceq=priceq, volat=sigmav, theta=thetav),
+   Rcpp=sim_oucpp(priceq=priceq, volat=sigmav, theta=thetav, innov=rnorm(nrows)),
+   times=10))[, c(1, 4, 5)]
```

Generating Random Numbers Using Logistic Map in *Rcpp*

The *logistic map* in *Rcpp* is about seven times faster than the loop in R, and even slightly faster than the standard `runif()` function in R!

```
> # Calculate uniformly distributed pseudo-random sequence
> unifun <- function(seedv, nrows=10) {
+   datav <- numeric(nrows)
+   datav[1] <- seedv
+   for (i in 2:nrows) {
+     datav[i] <- 4*datav[i-1]*(1-datav[i-1])
+   }  # end for
+   acos(1-2*datav)/pi
+ }  # end unifun
>
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts,
> # Microbenchmark Rcpp code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=runif(1e5),
+   rloop=unifun(0.3, 1e5),
+   Rcpp=unifuncpp(0.3, 1e5),
+   times=10))[, c(1, 4, 5)]
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namespace

// This is a simple example of exporting a C++ function
// You can source this function into an R session using
// function Rcpp::sourceCpp()
// (or via the Source button on the editor toolbar).
// Learn more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/

// function unifun() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector unifuncpp(double seedv, int(nrows) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector datav(nrows);
// initialize output vector
  datav[0] = seedv;
// perform loop
  for (int i=1; i < nrows; ++i) {
    datav[i] = 4*datav[i-1]*(1-datav[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*datav)/pi;
}
```

Package *RcppArmadillo* for Fast Linear Algebra

The package *RcppArmadillo* allows calling from R the high-level *Armadillo* C++ linear algebra library.

Armadillo provides ease of use and speed, with syntax similar to *Matlab*.

RcppArmadillo functions are often faster than even compiled R functions, because they use better optimized C++ code:
http://arma.sourceforge.net/speed.html

You can learn more about *RcppArmadillo*:
http://arma.sourceforge.net/
http://dirk.eddelbuettel.com/code/rcpp.armadillo.html
https://cran.r-project.org/web/packages/\emph{RcppArmadillo}/index.html
https://github.com/RcppCore/\emph{RcppArmadillo}

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts
> vec1 <- runif(1e5)
> vec2 <- runif(1e5)
> inner_vec(vec1, vec2)
> vec1 %*% vec2
```

```
// Rcpp header with information for C++ compiler
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;
// [[Rcpp::depends(RcppArmadillo)]]

// The function inner_vec() calculates the inner (dot) p
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_vec(arma::vec vec1, arma::vec vec2) {
  return arma::dot(vec1, vec2);
}  // end inner_vec

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vecv2, const arma::mat
  return arma::as_scalar(trans(vecv2) * (matv * vecv1));
}  // end inner_mat
```

```
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = inner_vec(vec1, vec2),
+   rcode = (vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Microbenchmark shows:
> # inner_vec() is several times faster than %*%, especially for lo
> #     expr     mean   median
> # 1 inner_vec 110.7067 110.4530
> # 2 rcode 585.5127 591.3575
```

Simulating *ARIMA* Processes Using *RcppArmadillo*

ARIMA processes can be simulated using *RcppArmadillo* even faster than by using the function `filter()`.

```
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Define AR(2) coefficients
> coeff <- c(0.9, 0.09)
> nrows <- 1e4
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> innov <- rnorm(nrows)
> # Simulate ARIMA using filter()
> arimar <- filter(x=innov, filter=coeff, method="recursive")
> # Simulate ARIMA using sim_ar()
> innov <- matrix(innov)
> coeff <- matrix(coeff)
> arimav <- sim_ar(coeff, innov)
> all.equal(drop(arimav), as.numeric(arimar))
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = sim_ar(coeff, innov),
+   filter = filter(x=innov, filter=coeff, method="recursive"),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
#include <RcppArmadillo.h> // include C++ header file fr
using namespace arma; // use C++ namespace from Armadill
// declare dependency on RcppArmadillo
// [[Rcpp::depends(RcppArmadillo)]]

//' @export
// [[Rcpp::export]]
arma::vec sim_ar(const arma::vec& innov, const arma::vec
  uword nrows = innov.n_elem;
  uword lookb = coeff.n_elem;
  arma::vec arimav[nrows);

  // startup period
  arimav(0) = innov(0);
  arimav(1) = innov(1) + coeff(lookb-1) * arimav(0);
  for (uword it = 2; it < lookb-1; it++) {
    arimav(it) = innov(it) + arma::dot(coeff.subvec(lookb
  }  // end for

  // remaining periods
  for (uword it = lookb; it < nrows; it++) {
    arimav(it) = innov(it) + arma::dot(coeff, arimav.sub
  }  // end for

  return arimav;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

RcppArmadillo functions can be made even faster by operating on pointers to matrices and performing calculations in place, without copying large matrices.

RcppArmadillo functions can be compiled using the same *Rtools* as those for *Rcpp* functions:
https://cran.r-project.org/bin/windows/Rtools/

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> matv <- matrix(runif(1e5), nc=1e3)
> # Center matrix columns using apply()
> matd <- apply(matv, 2, function(x) (x-mean(x)))
> # Center matrix columns in place using Rcpp demeanr()
> demeanr(matv)
> all.equal(matd, matv)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = (apply(matv, 2, mean)),
+   rcpp = demeanr(matv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Perform matrix inversion
> # Create random positive semi-definite matrix
> matv <- matrix(runif(25), nc=5)
> matv <- t(matv) %*% matv
> # Invert the matrix
> matrixinv <- solve(matv)
> inv_mat(matv)
> all.equal(matrixinv, matv)
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcode = solve(matv),
+   rcpp = inv_mat(matv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
#include <RcppArmadillo.h> // include C++ header file fr
using namespace arma; // use C++ namespace from Armadill
// declare dependency on RcppArmadillo
// [[Rcpp::depends(RcppArmadillo)]]

// Examples of \emph{RcppArmadillo} functions below

// The function demeanr() calculates a matrix with cente
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
int demeanr(arma::mat& matv) {
  for (uword i = 0; i < matv.n_cols; i++) {
    matv.col(i) -= arma::mean(matv.col(i));
  }  // end for
  return matv.n_cols;
}  // end demeanr

// The function inv_mat() calculates the inverse of symm
// definite matrix.
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inv_mat(arma::mat& matv) {
  matv = arma::inv_sympd(matv);
  return matv.n_cols;
}  // end inv_mat
```

Fast Correlation Matrix Inverse Using *RcppArmadillo*

RcppArmadillo can be used to quickly calculate the reduced inverse of correlation matrices.

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("/Users/jerzy/Develop/lecture_slides/scripts/HighI
> # Calculate matrix of random returns
> matv <- matrix(rnorm(300), nc=5)
> # Reduced inverse of correlation matrix
> dimax <- 4
> cormat <- cor(matv)
> eigend <- eigen(cormat)
> invmat <- eigend$vectors[, 1:dimax] %*%
+   (t(eigend$vectors[, 1:dimax]) / eigend$values[1:dimax])
> # Reduced inverse using \emph{RcppArmadillo}
> invarma <- calc_inv(cormat, dimax=dimax)
> all.equal(invmat, invarma)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = {eigend <- eigen(cormat)
+ eigend$vectors[, 1:dimax] %*% (t(eigend$vectors[, 1:dimax]) / eige
+   rcpp = calc_inv(cormat, dimax=dimax),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
// include Rcpp C++ header files
using namespace stdev;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& matv,
                   arma::uword dimax = 0, // Max number
                   double eigen_thresh = 0.01) { // Thre

  // Allocate SVD variables
  arma::vec svdval;  // Singular values
  arma::mat svdu, svdv;  // Singular matrices
  // Calculate the SVD
  arma::svd(svdu, svdval, svdv, tseries);
  // Calculate the number of non-small singular values
  arma::uword svdnum = arma::sum(svdval > eigen_thresh*a

  // If no regularization then set dimax to (svdnum - 1)
  if (dimax == 0) {
    // Set dimax
    dimax = svdnum - 1;
  } else {
    // Adjust dimax
    dimax = stdev::min(dimax - 1, svdnum - 1);
  }  // end if

  // Remove all small singular values
  svdval = svdval.subvec(0, dimax);
  svdu = svdu.cols(0, dimax);
  svdv = svdv.cols(0, dimax);

  // Calculate the reduced inverse from the SVD decompos
  return svdv*arma::diagmat(1/svdval)*svdu.t();
```

Portfolio Optimization Using *RcppArmadillo*

Fast portfolio optimization using matrix algebra can be implemented using *RcppArmadillo*.

```
// Fast portfolio optimization using matrix algebra and \emph{RcppArmadillo}
arma::vec calc_weights(const arma::mat& returns, // Asset returns
                       Rcpp::List controlv) { // List of portfolio optimization parameters

  // Apply different calculation methods for weights
  switch(calc_method(method)) {
  case methodenum::maxsharpe: {
    // Mean returns of columns
    arma::vec colmeans = arma::trans(arma::mean(returns, 0));
    // Shrink colmeans to the mean of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::mean(colmeans));
    // Calculate weights using reduced inverse
    weights = calc_inv(covmat, dimax, eigen_thresh)*colmeans;
    break;
  }  // end maxsharpe
  case methodenum::maxsharpemed: {
    // Median returns of columns
    arma::vec colmeans = arma::trans(arma::median(returns, 0));
    // Shrink colmeans to the median of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::median(colmeans));
    // Calculate weights using reduced inverse
    weights = calc_inv(covmat, dimax, eigen_thresh)*colmeans;
    break;
  }  // end maxsharpemed
  case methodenum::minvarlin: {
    // Minimum variance weights under linear constraint
    // Multiply reduced inverse times unit vector
    weights = calc_inv(covmat, dimax, eigen_thresh)*arma::ones(ncols);
    break;
  }  // end minvarlin
  case methodenum::minvarquad: {
    // Minimum variance weights under quadratic constraint
    // Calculate highest order principal component
    arma::vec eigenval;
    arma::mat eigenvec;
```

Strategy Backtesting Using *RcppArmadillo*

Fast backtesting of strategies can be implemented using *RcppArmadillo*.

```
arma::mat back_test(const arma::mat& excess, // Asset excess returns
                    const arma::mat& returns, // Asset returns
                    Rcpp::List controlv, // List of portfolio optimization model parameters
                    arma::uvec startp, // Start points
                    arma::uvec endd, // End points
                    double lambdaf = 0.0, // Decay factor for averaging the portfolio weights
                    double coeff = 1.0, // Multiplier of strategy returns
                    double bidask = 0.0) { // The bid-ask spread

  double lambda1 = 1-lambdaf;
  arma::uword nweights = returns.n_cols;
  arma::vec weights(nweights, fill::zeros);
  arma::vec weights_past = ones(nweights)/stdev::sqrt(nweights);
  arma::mat pnls = zeros(returns.n_rows, 1);

  // Perform loop over the end points
  for (arma::uword it = 1; it < endd.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate the portfolio weights
    weights = coeff*calc_weights(excess.rows(startp(it-1), endd(it-1)), controlv);
    // Calculate the weights as the weighted sum with past weights
    weights = lambda1*weights + lambdaf*weights_past;
    // Calculate out-of-sample returns
    pnls.rows(endd(it-1)+1, endd(it)) = returns.rows(endd(it-1)+1, endd(it))*weights;
    // Add transaction costs
    pnls.row(endd(it-1)+1) -= bidask*sum(abs(weightv - weights_past))/2;
    // Copy the weights
    weights_past = weights;
  }  // end for

  // Return the strategy pnls
  return pnls;

}  // end back_test
```

Package *reticulate* for Running Python from RStudio

The package *reticulate* allows running Python functions and scripts from RStudio.

The package *reticulate* relies on Python for interpreting the Python code.

```
> # Install package reticulate
> install.packages("reticulate")
> # Start Python session
> reticulate::repl_python()
> # Exit Python session
> exit
```

You must set your Global Options in RStudio to your Python executable, for example:
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3.10

You can learn more about the package *reticulate* here:
https://rstudio.github.io/reticulate/

Running Python Under *reticulate*

```
"""
Script for loading OHLC data from a CSV file and plotting a candlestick plot.
"""
# Import packages
import pandas as pd
import numpy as np
import plotly.graph_objects as go
# Load OHLC data from csv file - the time index is formatted inside read_csv()
symbol = "SPY"
range = "day"
filename = "/Users/jerzy/Develop/data/" + symbol + "_" + range + ".csv"
ohlc = pd.read_csv(filename)
datev = ohlc.Date
# Calculate log stock prices
ohlc[["Open", "High", "Low", "Close"]] = np.log(ohlc[["Open", "High", "Low", "Close"]])
# Calculate moving average
lookback = 55
closep = ohlc.Close
pricema = closep.ewm(span=lookback, adjust=False).mean()
# Plotly simple candlestick with moving average
# Create empty graph object
plotfig = go.Figure()
# Add trace for candlesticks
plotfig = plotfig.add_trace(go.Candlestick(x=datev,
  open=ohlc.Open, high=ohlc.High, low=ohlc.Low, close=ohlc.Close,
  name=symbol+" Log OHLC Prices", showlegend=False))
# Add trace for moving average
plotfig = plotfig.add_trace(go.Scatter(x=datev, y=pricema,
  name="Moving Average", line=dict(color="blue")))
# Customize plot
plotfig = plotfig.update_layout(title=symbol + " Log OHLC Prices",
  title_font_size=24, title_font_color="blue", yaxis_title="Price",
  font_color="black", font_size=18, xaxis_rangeslider_visible=False)
# Customize legend
plotfig = plotfig.update_layout(legend=dict(x=0.2, y=0.9, traceorder="normal",
  itemsizing="constant", font=dict(family="sans-serif", size=18, color="blue")))
# Render the plot
plotfig.show()
```

Homework Assignment

No homework!