# FRE7241 Algorithmic Portfolio Management

Lecture#4, Spring 2023

Jerzy Pawlowski *jp3900@nyu.edu*

*NYU Tandon School of Engineering*

April 11, 2023

## Centered Price Z-scores

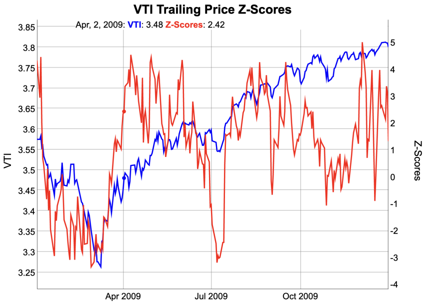
An extreme local price is a price which differs significantly from neighboring prices.

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 $\sigma_i$:

$$z_i = \frac{2p_i - p_{i-k} - p_{i+k}}{\sigma_i}$$

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

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



```
> # Extract the VTI log OHLC prices
> ohlc <- log(rutils::etfenv$VTI)
> nrows <- NROW(ohlc)
> closep <- quantmod::Cl(ohlc)
> retp <- rutils::diffit(closep)
> # Calculate the centered volatility
> look_back <- 7
> half_back <- look_back %/% 2
> stdev <- sqrt(HighFreq::roll_var(retp, look_back))
> stdev <- rutils::lagit(stdev, lagg=(-half_back))
> # Calculate the z-scores of prices
> pricez <- (2*closep -
+   rutils::lagit(closep, half_back, pad_zeros=FALSE) -
+   rutils::lagit(closep, -half_back, pad_zeros=FALSE))
> pricez <- ifelse(stdev > 0, pricez/stdev, 0)
```

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

## Labeling the Tops and Bottoms of Prices

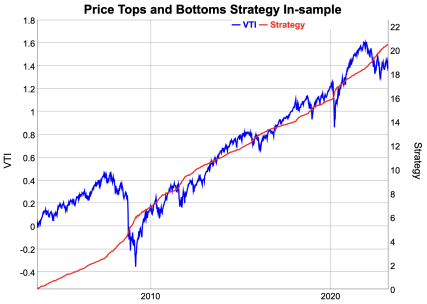
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

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

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

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

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



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

## Predictors of Price Extremes

The return volatility and trading volumes may be used as predictors in a classification model, in order to identify *overbought* and *oversold* conditions.

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

$$z_i = \frac{v_i - \bar{v}_i}{\sigma_i}$$

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

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

$$z_i = \frac{v_i - \bar{v}_i}{\sigma_i}$$

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

```
> # Calculate the volatility z-scores
> volat <- HighFreq::roll_var_ohlc(ohlc=ohlc, look_back=look_back, s
> volatm <- HighFreq::roll_mean(volat, look_back)
> volatsd <- sqrt(HighFreq::roll_var(rutils::diffit(volat), look_bac
> volatsd[1] <- 0
> volatz <- ifelse(volatsd > 0, (volat - volatm)/volatsd, 0)
> colnames(volatz) <- "volat"
> # Calculate the volume z-scores
> volum <- quantmod::Vo(ohlc)
> volumean <- HighFreq::roll_mean(volum, look_back)
> volumsd <- sqrt(HighFreq::roll_var(rutils::diffit(volum), look_bac
> volumsd[1] <- 0
> volumz <- ifelse(volumsd > 0, (volum - volumean)/volumsd, 0)
> colnames(volumz) <- "volume"
```

## Regression Z-Scores

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

$$z_i = \frac{p_i - (\alpha + \beta t_i)}{\sigma_i}$$

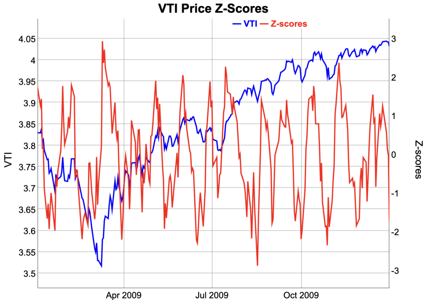
Where $\alpha$ and $\beta$ are the *regression coefficients*, and $\sigma_i$ is the standard deviation of the residuals.

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

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

The function `HighFreq::roll_zscores()` calculates the residuals of a rolling regression.

```
> # Calculate the trailing price regression z-scores
> datev <- matrix(zoo::index(closep))
> look_back <- 21
> controlv <- HighFreq::param_reg()
> regz <- HighFreq::roll_reg(respv=closep, predm=datev, look_back=
> regz <- drop(regz[, NCOL(regz)])
> regz[1:look_back] <- 0
```



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

## The *Logistic* Function

The *logistic* function expresses the probability of a numerical variable ranging over the whole interval of real numbers:

$$p(x) = \frac{1}{1 + \exp(-\lambda x)}$$

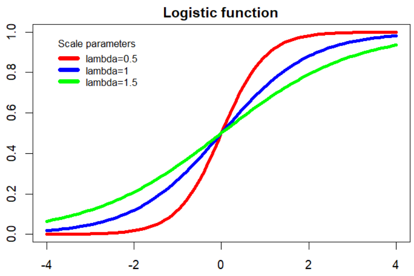
Where $\lambda$ is the scale (dispersion) parameter.

The *logistic* function is often used as an activation function in neural networks, and logistic regression can be viewed as a perceptron (single neuron network).

The *logistic* function can be inverted to obtain the *Odds Ratio* (the ratio of probabilities for favorable to unfavorable outcomes):

$$\frac{p(x)}{1 - p(x)} = \exp(\lambda x)$$

The function `plogis()` gives the cumulative probability of the *Logistic* distribution,



```
> lambdav <- c(0.5, 1, 1.5)
> colorv <- c("red", "blue", "green")
> # Plot three curves in loop
> for (it in 1:3) {
+   curve(expr=plogis(x, scale=lambdav[it]),
+ xlim=c(-4, 4), type="l", xlab="", ylab="", lwd=4,
+ col=colorv[it], add=(it>1))
+ }  # end for
> # Add title
> title(main="Logistic function", line=0.5)
> # Add legend
> legend("topleft", title="Scale parameters",
+        paste("lambda", lambdav, sep="="), y.intersp=0.4,
+        inset=0.05, cex=0.8, lwd=6, bty="n", lty=1, col=colorv)
```

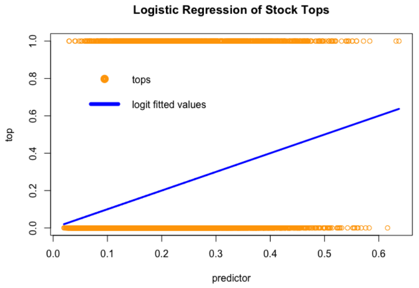
## Forecasting Stock Price Tops and Bottoms Using Logistic Regression

Consider a model which uses the weighted average of the volatility, trading volume, and regression z-scores, to forecast a stock top (overbought condition) or a bottom (oversold condition).

The residuals are the differences between the actual response values (0 and 1), and the calculated probabilities of default.

The residuals are not normally distributed, so the data is fitted using the *maximum likelihood* method, instead of least squares.

```
> # Define predictor for tops including intercept column
> predm <- cbind(volatz, volumz, regz)
> predm[1, ] <- 0
> predm <- rutils::lagit(predm)
> # Fit in-sample logistic regression for tops
> logmod <- glm(tops ~ predm, family=binomial(logit))
> summary(logmod)
> coeff <- logmod$coefficients
> fcast <- drop(cbind(rep(1, nrows), predm) %*% coeff)
> ordern <- order(fcast)
> # Calculate the in-sample forecasts from logistic regression mod
> fcast <- 1/(1+exp(-fcast))
> all.equal(logmod$fitted.values, fcast, check.attributes=FALSE)
> hist(fcast)
```



```
> plot(x=fcast[ordern], y=tops[ordern],
+      main="Logistic Regression of Stock Tops",
+      col="orange", xlab="predictor", ylab="top")
> lines(x=fcast[ordern], y=logmod$fitted.values[ordern], col="blue"
> legend(x="topleft", inset=0.1, bty="n", lwd=6,
+  legend=c("tops", "logit fitted values"), y.intersp=0.5,
+  col=c("orange", "blue"), lty=c(NA, 1), pch=c(1, NA))
```

## Forecasting Errors of Stock Tops and Bottoms

A *binary classification model* categorizes cases based on its forecasts whether the *null hypothesis* is TRUE or FALSE.

Let the *null hypothesis* be that the data point is not a top: tops = FALSE.

A *positive* result corresponds to rejecting the null hypothesis (tops = TRUE), while a *negative* result corresponds to accepting the null hypothesis (tops = FALSE).

The forecasts are subject to two different types of errors: *type I* and *type II* errors.

A *type I* error is the incorrect rejection of a TRUE *null hypothesis* (i.e. a "false positive"), when tops = FALSE but it's classified as tops = TRUE.

A *type II* error is the incorrect acceptance of a FALSE *null hypothesis* (i.e. a "false negative"), when tops = TRUE but it's classified as tops = FALSE.

```
> # Define discrimination threshold value
> threshv <- quantile(fcast, confl[2])
> # Calculate the confusion matrix in-sample
> confmat <- table(actual=!tops, forecast=(fcast < threshv))
> confmat
> # Calculate the FALSE positive (type I error)
> sum(tops & (fcast < threshv))
> # Calculate the FALSE negative (type II error)
> sum(!tops & (fcast > threshv))
```

## The Confusion Matrix of a Binary Classification Model

The confusion matrix summarizes the performance of a classification model on a set of test data for which the actual values of the *null hypothesis* are known.

| **Actual** \ **Forecast** | **Null is FALSE** | **Null is TRUE** |
|---|---|---|
| **Null is FALSE** | True Positive (sensitivity) | False Negative (type II error) |
| **Null is TRUE** | False Positive (type I error) | True Negative (specificity) |

```
> # Calculate the FALSE positive and FALSE negative rates
> confmat <- confmat / rowSums(confmat)
> c(typeI=confmat[2, 1], typeII=confmat[1, 2])
```

Let the *null hypothesis* be that the data point is not a top: `tops = FALSE`.

The *true positive* rate (known as the *sensitivity*) is the fraction of `FALSE` *null hypothesis* cases that are correctly classified as `FALSE`.

The *false negative* rate is the fraction of `FALSE` *null hypothesis* cases that are incorrectly classified as `TRUE` (*type II* error).

The sum of the *true positive* plus the *false negative* rate is equal to 1.

The *true negative* rate (known as the *specificity*) is the fraction of `TRUE` *null hypothesis* cases that are correctly classified as `TRUE`.

The *false positive* rate is the fraction of `TRUE` *null hypothesis* cases that are incorrectly classified as `FALSE` (*type I* error).

The sum of the *true negative* plus the *false positive* rate is equal to 1.

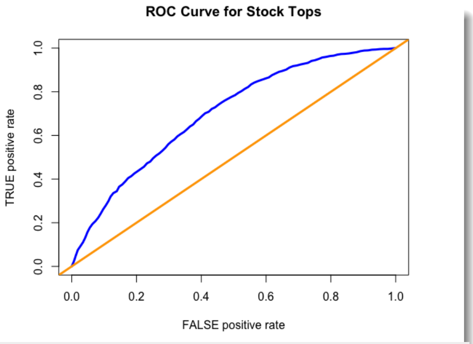
## Receiver Operating Characteristic (ROC) Curve for Stock Tops

The *ROC curve* is the plot of the *true positive* rate, as a function of the *false positive* rate, and illustrates the performance of a binary classifier.

The area under the *ROC curve* (AUC) measures the classification ability of a binary classifier.

The *informedness* is equal to the sum of the sensitivity plus the specificity, and measures the performance of a binary classification model.

```
> # Confusion matrix as function of threshold
> confun <- function(actual, fcast, threshv) {
+   forb <- (fcast < threshv)
+   conf <- matrix(c(sum(!actual & !forb), sum(actual & !forb),
+              sum(!actual & forb), sum(actual & forb)), ncol=2)
+   conf <- conf / rowSums(conf)
+   c(typeI=conf[2, 1], typeII=conf[1, 2])
+ }  # end confun
> confun(!tops, fcast, threshv=threshv)
> # Define vector of discrimination thresholds
> threshv <- quantile(fcast, seq(0.01, 0.99, by=0.01))
> # Calculate the error rates
> error_rates <- sapply(threshv, confun,
+   actual=!tops, fcast=fcast)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastops <- (fcast > threshm)
```



```
> # Calculate the area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> truepos <- (1 - error_rates[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(error_rates[, "typeI"])
> abs(sum(truepos*falsepos))
> # Plot ROC Curve for stock tops
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      main="ROC Curve for Stock Tops", type="l", lwd=3, col="blue"
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

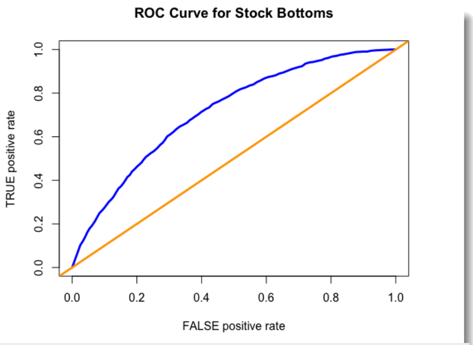
## Receiver Operating Characteristic (ROC) Curve for Stock Bottoms

The *ROC curve* is the plot of the *true positive* rate, as a function of the *false positive* rate, and illustrates the performance of a binary classifier.

The area under the *ROC curve* (AUC) measures the classification ability of a binary classifier.

The *informedness* is equal to the sum of the sensitivity plus the specificity, and measures the performance of a binary classification model.

```
> # Fit in-sample logistic regression for bottoms
> logmod <- glm(bottoms ~ predm, family=binomial(logit))
> summary(logmod)
> # Calculate the in-sample forecast from logistic regression model
> coeff <- logmod$coefficients
> fcast <- drop(cbind(rep(1, nrows), predm) %*% coeff)
> fcast <- 1/(1+exp(-fcast))
> # Calculate the error rates
> error_rates <- sapply(threshv, confun,
+   actual=!bottoms, fcast=fcast)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastbot <- (fcast > threshm)
```



```
> # Calculate the area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> truepos <- (1 - error_rates[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(error_rates[, "typeI"])
> abs(sum(truepos*falsepos))
> # Plot ROC Curve for stock tops
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      main="ROC Curve for Stock Bottoms", type="l", lwd=3, col="blu
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

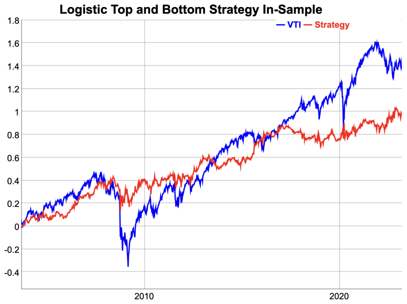
## Logistic Tops and Bottoms Strategy In-sample

The logistic strategy forecasts the tops and bottoms of pricev, using a logistic regression model with the volatility and trading volumes as predictors.

Averaging the forecasts over time improves strategy performance because of the bias-variance tradeoff.

It makes sense to average the forecasts over time because they are forecasts for future time intervals, not just a single point in time.

```
> # Average the signals over time
> topsav <- HighFreq::roll_sum(matrix(forecastops), 5)/5
> botsav <- HighFreq::roll_sum(matrix(forecastbot), 5)/5
> # Simulate in-sample VTI strategy
> posv <- (botsav - topsav)
> # Standard strategy
> # posv <- rep(NA_integer_, NROW(retp))
> # posv[1] <- 0
> # posv[forecastops] <- (-1)
> # posv[forecastbot] <- 1
> # posv <- zoo::na.locf(posv)
> posv <- rutils::lagit(posv)
> pnls <- retp*posv
```

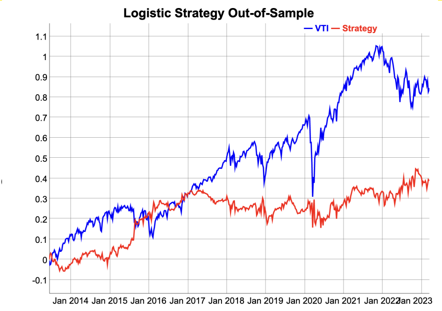


```
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of in-sample VTI strategy
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Logistic Top and Bottom Strategy In-Sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

## Logistic Tops and Bottoms Strategy Out-of-Sample

The logistic strategy forecasts the tops and bottoms of pricev, using a logistic regression model with the volatility and trading volumes as predictors.

```
> # Define in-sample and out-of-sample intervals
> insample <- 1:(nrows %/% 2)
> outsample <- (nrows %/% 2 + 1):nrows
> # Fit in-sample logistic regression for tops
> logmod <- glm(tops[insample] ~ predm[insample, ], family=binomial(
> fitv <- logmod$fitted.values
> coefftop <- logmod$coefficients
> # Calculate the error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!tops[insample], fcast=fitv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshtop <- threshv[which.max(informv)]
> # Fit in-sample logistic regression for bottoms
> logmod <- glm(bottoms[insample] ~ predm[insample, ], family=binor
> fitv <- logmod$fitted.values
> coeffbot <- logmod$coefficients
> # Calculate the error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!bottoms[insample], fcast=fitv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshbot <- threshv[which.max(informv)]
> # Calculate the out-of-sample forecasts from logistic regression
> predictout <- cbind(rep(1, NROW(outsample)), predm[outsample, ])
> fcast <- drop(predictout %*% coefftop)
> fcast <- 1/(1+exp(-fcast))
> forecastops <- (fcast > threshtop)
> fcast <- drop(predictout %*% coeffbot)
> fcast <- 1/(1+exp(-fcast))
> forecastbot <- (fcast > threshbot)
```



```
> # Simulate in-sample VTI strategy
> topsav <- HighFreq::roll_sum(matrix(forecastops), 5)/5
> botsav <- HighFreq::roll_sum(matrix(forecastbot), 5)/5
> posv <- (botsav - topsav)
> posv <- rutils::lagit(posv)
> pnls <- retp[outsample, ]*posv
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retp[outsample, ], pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of in-sample VTI strategy
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Logistic Strategy Out-of-Sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

## Autoregressive Processes

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

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

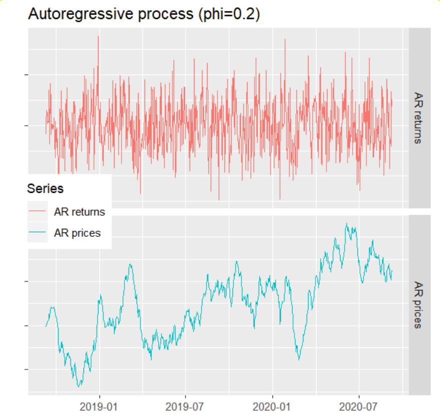
Where $\varphi_i$ are the *AR(n)* coefficients, and $\xi_i$ are standard normal *innovations*.

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 $r_i$ is mean reverting to zero.

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

```
> # Simulate AR processes
> set.seed(1121)  # Reset random numbers
> 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")
```



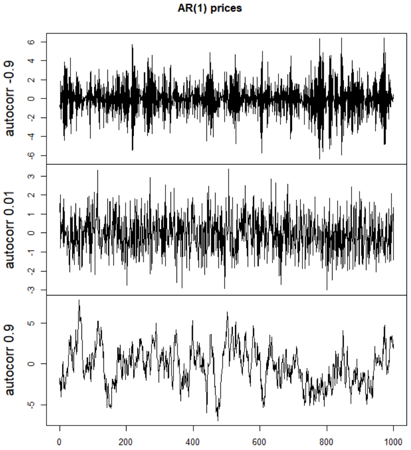
```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=arimav, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),
+   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 $\varphi_1$, with a negative coefficient producing faster mean reversion, and a positive coefficient producing stronger diversion.

A positive coefficient $\varphi_1$ produces a diversion away from the mean, so that the time series $r_i$ 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)  # Reset random numbers
+   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)*:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

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); innov <- rnorm(nrows)
> # 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(inno
> 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)
> 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 $\rho_i$ of an *AR(1)* process (defined as $r_i = \varphi r_{i-1} + \xi_i$), satisfies the recursive equation:
$\rho_i = \varphi \rho_{i-1}$, with $\rho_1 = \varphi$.

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

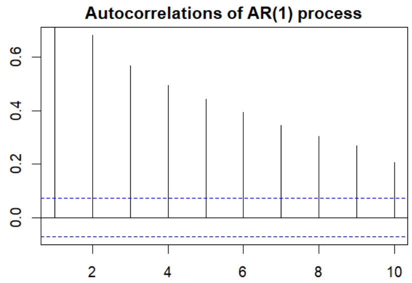
The *AR(1)* process can be simulated recursively:

$$\begin{aligned} r_1 &= \xi_1 \\ r_2 &= \varphi r_1 + \xi_2 = \xi_2 + \varphi \xi_1 \\ r_3 &= \xi_3 + \varphi \xi_2 + \varphi^2 \xi_1 \\ r_4 &= \xi_4 + \varphi \xi_3 + \varphi^2 \xi_2 + \varphi^3 \xi_1 \end{aligned}$$

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

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

If $\varphi = 1.0$ then the influence of the random innovations $\xi_i$ persists indefinitely, so that the variance of $r_i$ increases linearly with 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
> acfv <- rutils::plot_acf(arimav, lag=10, xlab="", ylab="",
+   main="Autocorrelations of AR(1) process")
> acfv$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.

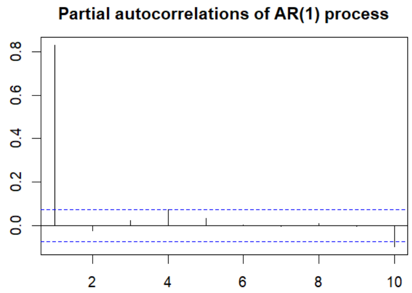
A linear combination of the time series and its own lag can be created, such that its lag 1 autocorrelation is zero.

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* $\varrho_i$ are the estimators of the coefficients $\phi_i$ 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
> pacfv <- pacf(arimav, lag=10, xlab="", ylab="", main="")
> title("Partial autocorrelations of AR(1) process", line=1)
> pacfv <- as.numeric(pacfv$acf)
> pacfv[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)*:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$

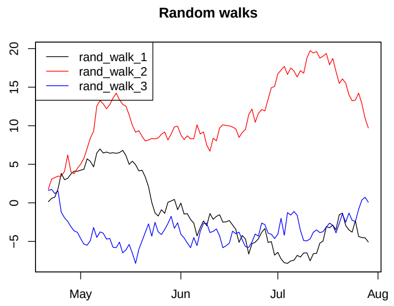
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_n z^n = 0$

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*: $p_i = p_{i-1} + \xi_i$



```
> 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: $p_n = \sum_{i=1}^{n} r_i$.

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

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

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

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

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

Then asset prices follow the process:
$p_i = (1 + \varphi) p_{i-1} - \varphi p_{i-2} + \xi_i$

Which is a *unit root* process for all values of $\varphi$, 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)
> 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)
> 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: $r_i = \varphi r_{i-1} + \xi_i$ 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: $r_i = r_{i-1} + \xi_i$

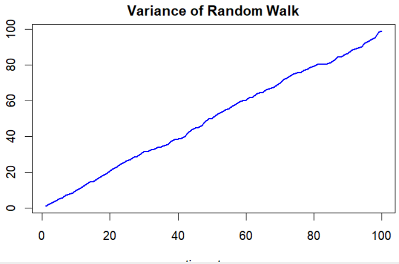
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

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

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 isn't *stationary*.

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



```
> # Simulate random walks using apply() loops
> set.seed(1121)  # Initialize random number generator
> randws <- matrix(rnorm(1000*100), ncol=1000)
> randws <- apply(randws, 2, cumsum)
> varv <- apply(randws, 1, var)
> # Simulate random walks using vectorized functions
> set.seed(1121)  # Initialize random number generator
> randws <- matrixStats::colCumsums(matrix(rnorm(1000*100), ncol=10
> 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 $r_i$ are equal to the random *innovations*:

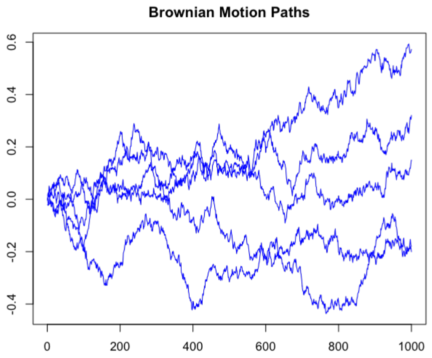
$$r_i = p_i - p_{i-1} = \sigma \, \xi_i$$
$$p_i = p_{i-1} + r_i$$

Where $\sigma$ is the volatility of returns, and $\xi_i$ 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 $\varphi = 1$:

$$p_i = \varphi p_{i-1} + \sigma \, \xi_i$$

```
> # 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)
> # Open plot window on Mac
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot 5 paths of Brownian motion
> matplot(y=pricev, main="Brownian Motion Paths",
+   xlab="", ylab="", 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 $r_i$ are equal to the difference between the equilibrium price $\mu$ minus the latest price $p_{i-1}$, times the mean reversion parameter $\theta$, plus random *innovations*:

$$r_i = p_i - p_{i-1} = \theta\,(\mu - p_{i-1}) + \sigma\,\xi_i$$
$$p_i = p_{i-1} + r_i$$

Where $\sigma$ is the volatility of returns, and $\xi_i$ 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:

$$p_i = \theta\,\mu + (1 - \theta)\,p_{i-1} + \sigma\,\xi_i$$

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(init_price=prici, eq_price=priceq,
+   theta=thetav, innov=matrix(innov))
> all.equal(pricev, drop(pricev_cpp))
> # 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(init_price=prici, eq_price=priceq,
+     theta=thetav, innov=matrix(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:

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

Where $W_t$ is a *Brownian Motion*, with $\mathrm{d}W_t$ following the standard normal distribution $\phi(0, \sqrt{\mathrm{d}t})$.

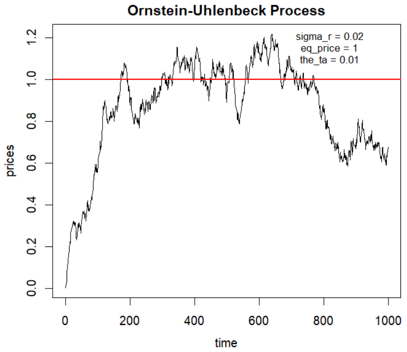
The solution of the *Ornstein-Uhlenbeck* process is given by:

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

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 $\mu$.

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

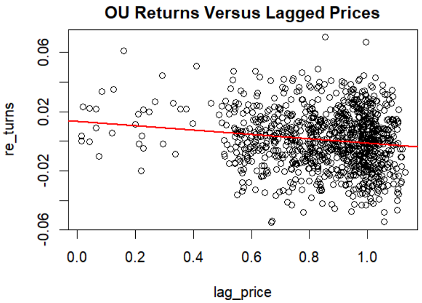


```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright", title=paste(c(paste0("sigmav = ", sigmav),
+      paste0("eq_price = ", ),
+      paste0("thetav = ", thetav)),
+    collapse="\n"),
+  legend="", cex=0.8, inset=0.1, bg="white", bty="n")
> 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 $\theta$ and $\mu$ 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
> betav <- cov(retp, pricelag)/var(pricelag)
> alpha <- (mean(retp) - betav*mean(pricelag))
> cbind(direct=c(alpha=alpha, beta=betav), lm=coeff[, 1])
> all.equal(c(alpha=alpha, beta=betav), coeff[, 1],
+     check.attributes=FALSE)
> # Calculate regression standard errors directly
> betav <- c(alpha=alpha, beta=betav)
> fitv <- (alpha + betav*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.

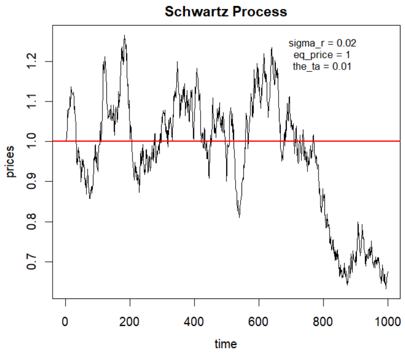
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of pricev, which can be negative.

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

$$r_i = \log p_i - \log p_{i-1} = \theta \, (\mu - p_{i-1}) + \sigma \, \xi_i$$
$$p_i = p_{i-1} \exp(r_i)$$

Where the parameter $\theta$ is the strength of mean reversion, $\sigma$ is the volatility, and $\xi_i$ 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)  # Reset random numbers
> 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 $r_i$ are equal to the sum of a mean reverting term plus *autoregressive* terms:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \sigma\,\xi_i$$
$$p_i = p_{i-1} + r_i$$

Where $\mu$ is the equilibrium price, $\sigma$ is the volatility of returns, $\theta$ is the strength of mean reversion, and $\xi_i$ are standard normal *innovations*.

Then the prices follow an *autoregressive* process:

$$p_i = \theta\mu + (1+\varphi_1-\theta)p_{i-1} + (\varphi_2-\varphi_1)p_{i-2} + \ldots + (\varphi_n - \varphi_{n-1})p_{i-n} - \varphi_n p_{i-n-1} + \sigma\,\xi_i$$

The sum of the *autoregressive* coefficients is equal to $1-\theta$, so if the mean reversion parameter $\theta$ is positive: $\theta > 0$, then the time series $p_i$ 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
> pricev[2] <- pricev[1] + retp[2]
> retp[3] <- thetav*(priceq - pricev[2]) + coeff[1]*retp[2] + coeff
> pricev[3] <- pricev[2] + retp[3]
> for (it in 4:nrows) {
+   retp[it] <- thetav*(priceq - pricev[it-1]) + retp[(it-1):(it-3)
+   pricev[it] <- pricev[it-1] + retp[it]
+ }  # end for
> # Simulate Dickey-Fuller process in Rcpp
> pricecpp <- HighFreq::sim_df(init_price=prici, eq_price=priceq, th
> # Compare prices
> all.equal(pricev, drop(pricev_cpp))
> # 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(init_price=prici, eq_price=priceq, theta=
+   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 $p_i$:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \sigma\,\xi_i$$
$$p_i = p_{i-1} + r_i$$

Where $\mu$ is the equilibrium price, $\sigma$ is the volatility of returns, and $\theta$ is the strength of mean reversion.

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

If the mean reversion parameter $\theta$ is positive: $\theta > 0$, then the time series $p_i$ 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 $\theta$ 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.

```
> set.seed(1121); innov <- matrix(rnorm(1e4, sd=0.01))
> # Simulate AR(1) process with coefficient=1, with unit root
> arimav <- HighFreq::sim_ar(coeff=matrix(1), innov=innov)
> x11(); plot(arimav, t="l", main="AR(1) coefficient = 1.0")
> # 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)
> x11(); 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(init_price=prici, eq_price=priceq,
+   theta=thetav, innov=innov)
> x11(); 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(init_price=prici, eq_price=priceq,
+   theta=thetav, innov=innov)
> x11(); 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: $r_i = \theta(\mu - p_{i-1}) + \varepsilon_i$.

## 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*.

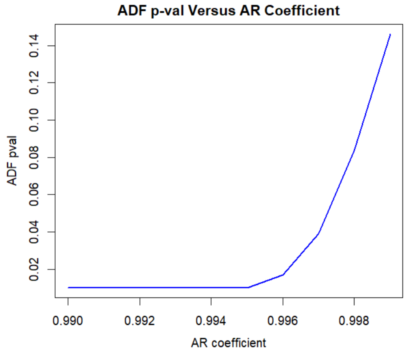
The *ADF* test has low *sensitivity*, i.e. the ability to correctly identify time series with no *unit root*, causing it to produce *false negatives* (*type II* errors).

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
```

## Idiosyncratic Stock Returns

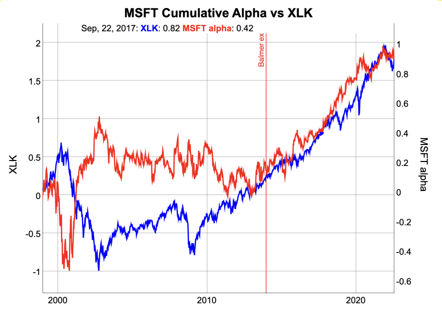
The daily stock returns $r_i - r_f$ in excess of the risk-free rate $r_f$, can be decomposed into *systematic* returns $\beta(r_m - r_f)$ ($r_m$ are the market returns) plus *idiosyncratic* returns $\alpha + \varepsilon_i$ (which are uncorrelated to the market returns):

$$r_i - r_f = \alpha + \beta(r_m - r_f) + \varepsilon_i$$

The *alpha* $\alpha$ are the abnormal returns in excess of the risk premium $\beta(r_m - r_f)$, and $\varepsilon_i$ are the regression residuals with zero mean.

We can simplify the formula by setting the risk-free rate to zero $r_f = 0$ and redefining the alpha $\alpha$.

```
> # Load daily S&P500 stock returns
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData")
> # Select ETF returns
> retetf <- rutils::etfenv$returns[, c("VTI", "XLK", "XLF", "XLE")]
> # Calculate the MSFT betas with respect to different ETFs
> betas <- sapply(retetf, function(retetf) {
+   retv <- na.omit(cbind(returns$MSFT, retetf))
+   # Calculate the MSFT beta
+   drop(cov(retv$MSFT, retv[, 2])/var(retv[, 2]))
+ }) # end sapply
> # Combine MSFT and XLK returns
> retv <- cbind(returns$MSFT, rutils::etfenv$returns$XLK)
> retv <- na.omit(retv)
> colnames(retv) <- c("MSFT", "XLK")
> # Calculate the beta and alpha of returns MSFT ~ XLK
> betav <- drop(cov(retv$MSFT, retv$XLK)/var(retv$XLK))
> alphav <- retv$MSFT - betav*retv$XLK
> # Scatterplot of returns
> plot(MSFT ~ XLK, data=retv, main="MSFT ~ XLK Returns",
+      xlab="XLK", ylab="MSFT", pch=1, col="blue")
```



The stock of Microsoft *MSFT* began outperforming the *XLK* ETF after Steve Ballmer was replaced as CEO in 2014.

```
> # dygraph plot of MSFT idiosyncratic returns vs XLK
> endd <- rutils::calc_endpoints(retv, interval="weeks")
> datev <- zoo::index(retv)[endd]
> dateb <- datev[findInterval(as.Date("2014-01-01"), datev)] # Steve
> datav <- cbind(retv$XLK, alphav)
> colnames(datav)[2] <- "MSFT alpha"
> colnamev <- colnames(datav)
> dygraphs::dygraph(cumsum(datav)[endd], main="MSFT Cumulative Alpha
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=colnamev[2], axis="y2", col="red", strokeWidth=2)
+   dyEvent(dateb, label="Balmer exit", strokePattern="solid", color
+   dyLegend(show="always", width=500)
```

## Trailing Idiosyncratic Stock Returns

In practice, the stock beta should be updated over time and applied out-of-sample to calculate the trailing idiosyncratic stock returns.

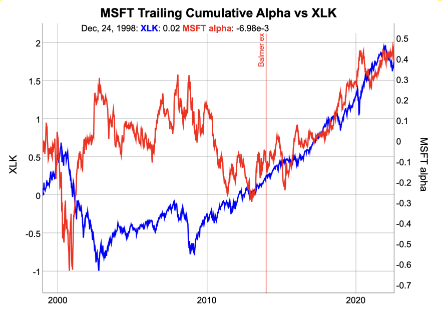
The trailing beta $\beta$ of a stock with returns $r_t$ with respect to a stock index with returns $R_t$ can be updated using these recursive formulas with the weight decay factor $\lambda$:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$
$$\bar{R}_t = \lambda \bar{R}_{t-1} + (1 - \lambda) R_t$$
$$\sigma^2_t = \lambda \sigma^2_{t-1} + (1 - \lambda)(R_t - \bar{R}_t)^2$$
$$\operatorname{cov}_t = \lambda \operatorname{cov}_{t-1} + (1 - \lambda)(r_t - \bar{r}_t)(R_t - \bar{R}_t)$$
$$\beta_t = \frac{\operatorname{cov}_t}{\sigma^2_t}$$
$$\alpha_t = r_t - \beta_t R_t$$

The parameter $\lambda$ determines the rate of decay of the weight of past returns. If $\lambda$ is close to 1 then the decay is weak and past returns have a greater weight. And vice versa if $\lambda$ is close to 0.

The function `HighFreq::run_covar()` calculates the trailing variances, covariances, and means of two *time series*.

Using a dynamic beta produces a similar picture of *MSFT* stock performance versus the *XLK* ETF.



```
> # Calculate the trailing alphas and betas
> lambda <- 0.9
> covars <- HighFreq::run_covar(retv, lambda)
> betav <- covars[, 1]/covars[, 3]
> alphav <- retv$MSFT - betav*retv$XLK
> # dygraph plot of trailing MSFT idiosyncratic returns vs XLK
> datav <- cbind(retv$XLK, alphav)
> colnames(datav)[2] <- "MSFT alpha"
> colnamev <- colnames(datav)
> dygraphs::dygraph(cumsum(datav)[endd], main="MSFT Trailing Cumula
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=colnamev[2], axis="y2", col="red", strokeWidth=2)
+   dyEvent(dateb, label="Balmer exit", strokePattern="solid", colo
+   dyLegend(show="always", width=500)
```

## Cointegration of Stocks Prices

A group of stocks is *cointegrated* if there is a portfolio of the stocks whose price is range-bound.

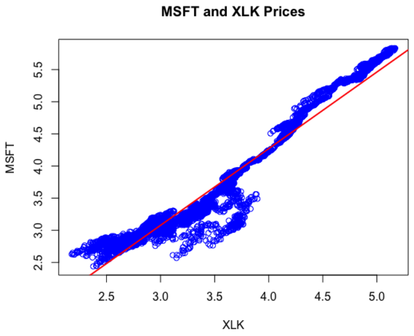
For two stocks, the cointegrating factor $\beta$ can be obtained from the regression of the stock prices. The cointegrated portfolio price $R$ is the residual of the regression:

$$R = p_1 - \beta p_2$$

Regressing the stock *prices* produces a *cointegrated* portfolio, with a small variance of its *prices*.

Regressing the stock *returns* produces a *correlated* portfolio, with a small variance of its *returns*.

The standard confidence intervals are not valid for the regression of prices, because prices are not stationary and are not normally distributed.



```
> # Load daily S&P500 stock prices
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices
> # Combine MSFT and XLK prices
> pricev <- cbind(rutils::etfenv$prices$XLK, prices$MSFT)
> pricev <- log(na.omit(pricev))
> colnames(pricev) <- c("XLK", "MSFT")
> datev <- zoo::index(pricev)
> # Calculate the beta regression coefficient of prices MSFT ~ XLK
> betav <- drop(cov(pricev$MSFT, pricev$XLK)/var(pricev$XLK))
> # Calculate the cointegrated portfolio prices
> pricec <- pricev$MSFT - betav*pricev$XLK
> colnames(pricec) <- "MSFT Coint XLK"
```

```
> # Scatterplot of MSFT and XLK prices
> plot(MSFT ~ XLK, data=pricev, main="MSFT and XLK Prices",
+      xlab="XLK", ylab="MSFT", pch=1, col="blue")
> abline(a=mean(pricec), b=betav, col="red", lwd=2)
> # Plot time series of prices
> endd <- rutils::calc_endpoints(pricev, interval="weeks")
> dygraphs::dygraph(pricev[endd], main="MSFT and XLK Log Prices") %>
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

## Cointegrated Portfolio Prices

The Engle-Granger two-step procedure can be used to test the cointegrated portfolio of two stocks:

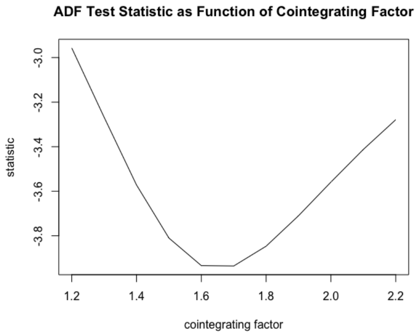
- Perform a regression of the stock prices to calculate the cointegrating factor $\beta$,
- Apply the *ADF* test to the cointegrated portfolio price, to determine if it has a unit root (the portfolio price diverges), or if it's mean reverting.

The *p*-value of the *ADF* test for the cointegrated portfolio of *MSFT* and *XLK* is not small, so the *null hypothesis* that it has a *unit root* (it diverges) cannot be rejected.

The *null hypothesis* of the *ADF* test is that the time series has a *unit root* (it diverges). So a small *p*-value suggests that the *null hypothesis* is FALSE and that the time series is range-bound.

The *ADF* test statistic for the cointegrated portfolio is smaller than for either *MSFT* or *XLK* alone, which indicates that it's more mean-reverting.

```
> # Plot histogram of the cointegrated portfolio prices
> hist(pricec, breaks=100, xlab="Prices",
+   main="Histogram of Cointegrated Portfolio Prices")
> # Plot of cointegrated portfolio prices
> datav <- cbind(pricev$XLK, pricec)[endd]
> colnames(datav)[2] <- "Cointegrated Portfolio"
> colnamev <- colnames(datav)
> dygraphs::dygraph(datav, main="MSFT and XLK Cointegrated Portfol
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
```


ADF Test Statistic as Function of Cointegrating Factor

```
> # Perform ADF test on the individual stocks
> sapply(pricev, tseries::adf.test, k=1)
> # Perform ADF test on the cointegrated portfolio
> tseries::adf.test(pricec, k=1)
> # Perform ADF test for vector of cointegrating factors
> betas <- seq(1.2, 2.2, 0.1)
> adfstat <- sapply(betas, function(betav) {
+   pricec <- (pricev$MSFT - betav*pricev$XLK)
+   tseries::adf.test(pricec, k=1)$statistic
+ })  # end sapply
> # Plot ADF statistics for vector of cointegrating factors
> plot(x=betas, y=adfstat, type="l", xlab="cointegrating factor", y
+  main="ADF Test Statistic as Function of Cointegrating Factor")
```

## Bollinger Band Strategy for Cointegrated Pairs

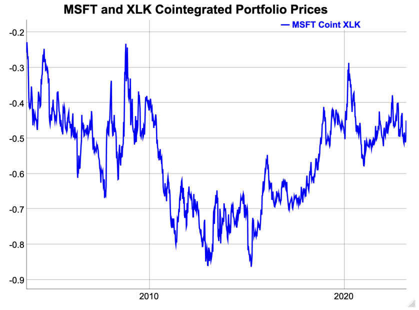
The returns of the cointegrated portfolio have negative autocorrelations, so it can be traded using a mean-reverting strategy.

The *Bollinger Band* strategy switches to long \$1 dollar of the stock when the portfolio price is cheap (below the lower band), and sells short -\$1 dollar of stock when the portfolio is rich (expensive - above the upper band). It goes flat \$0 dollar of stock (unwinds) if the stock reaches a fair (mean) price.

The strategy is therefore always either long \$1 dollar of stock, or short -\$1 dollar of stock, or flat \$0 dollar of stock.

The portfolio is cheap if its price is below its mean price minus the portfolio standard deviation, and it's rich (expensive) if its price is above the mean plus the standard deviation. The portfolio price is fair if it's equal to the mean price.

The pairs strategy is path dependent because it depends on the risk position. So the simulation requires performing a loop.



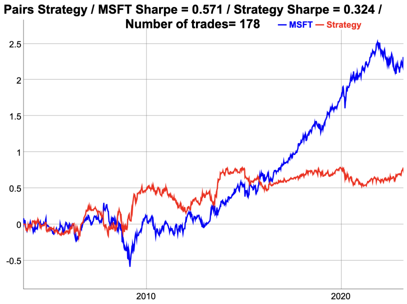
```
> # Plot of PACF of the cointegrated portfolio returns
> pricen <- zoo::coredata(pricec) # Numeric price
> retd <- rutils::diffit(pricen)
> pacf(retd, lag=10, xlab=NA, ylab=NA,
+      main="PACF of Cointegrated Portfolio Returns")
> # Dygraphs plot of cointegrated portfolio prices
> endd <- rutils::calc_endpoints(pricec, interval="weeks")
> dygraphs::dygraph(pricec[endd], main=
+   "MSFT and XLK Cointegrated Portfolio Prices") %>%
+   dyOptions(colors=c("blue"), strokeWidth=2) %>%
+   dyLegend(show="always", width=200)
```

## Pairs Strategy With Fixed Beta

The pairs strategy performs well because it's in-sample - it uses the in-sample beta.

A more realistic strategy requires calculating the trailing betas on past prices, updating the pairs price, and updating the trailing mean and volatility.

```
> # Calculate the trailing mean prices and volatilities
> lambda <- 0.9
> meanv <- HighFreq::run_mean(pricen, lambda=lambda)
> volat <- HighFreq::run_var(pricen, lambda=lambda)
> volat <- sqrt(volat)
> # Simulate the pairs Bollinger strategy
> pricem <- pricen - meanv # De-meaned price
> nrows <- NROW(pricec)
> threshd <- rutils::lagit(volat)
> posv <- rep(NA_integer_, nrows)
> posv[1] <- 0
> posv <- ifelse(pricem > threshd, -1, posv)
> posv <- ifelse(pricem < -threshd, 1, posv)
> posv <- zoo::na.locf(posv)
> # Lag the positions to trade in the next period
> posv <- rutils::lagit(posv, lagg=1)
> # Calculate the pnls and the number of trades
> retv <- rutils::diffit(pricev)
> pnls <-  posv*(retv$MSFT - betav*retv$XLK)
> ntrades <- sum(abs(rutils::diffit(posv)) > 0)
```



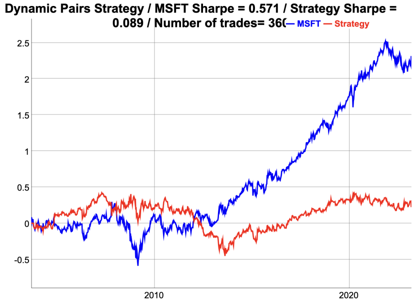
```
> # Calculate the Sharpe ratios
> wealthv <- cbind(retv$MSFT, pnls)
> colnames(wealthv) <- c("MSFT", "Strategy")
> sharper <- sqrt(252)*sapply(wealthv, function(x) mean(x)/sd(x[x<0]
> sharper <- round(sharper, 3)
> # Dygraphs plot of pairs Bollinger strategy
> colnamev <- colnames(wealthv)
> caption <- paste("Pairs Strategy", "/ \n",
+   paste0(paste(colnamev[1:2], "Sharpe =", sharper), collapse=" / '
+   "Number of trades=", ntrades)
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main=caption) %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=200)
```

## Pairs Strategy With Dynamic Beta

In practice, the stock beta should be updated over time and applied out-of-sample to calculate the trailing cointegrated pair prices.

Using a dynamic beta produces poor performance for many stock and ETF pairs.

```
> # Calculate the trailing cointegrated pair prices
> covars <- HighFreq::run_covar(pricev, lambda)
> betav <- covars[, 1]/covars[, 3]
> pricec <- (pricev$MSFT - betav*pricev$XLK)
> # Recalculate the mean of cointegrated portfolio prices
> meanv <- HighFreq::run_mean(pricec, lambda=lambda)
> vars <- sqrt(HighFreq::run_var(pricec, lambda=lambda))
> # Simulate the pairs Bollinger strategy
> pricen <- zoo::coredata(pricec) # Numeric price
> pricem <- pricen - meanv # De-meaned price
> threshd <- rutils::lagit(volat)
> posv <- rep(NA_integer_, nrows)
> posv[1] <- 0
> posv <- ifelse(pricem > threshd, -1, posv)
> posv <- ifelse(pricem < -threshd, 1, posv)
> posv <- zoo::na.locf(posv)
> posv <- rutils::lagit(posv, lagg=1)
> # Calculate the pnls and the number of trades
> retv <- rutils::diffit(pricev)
> pnls <-  posv*(retv$MSFT - betav*retv$XLK)
> ntrades <- sum(abs(rutils::diffit(posv)) > 0)
```



```
> # Calculate the Sharpe ratios
> wealthv <- cbind(retv$MSFT, pnls)
> colnames(wealthv) <- c("MSFT", "Strategy")
> sharper <- sqrt(252)*sapply(wealthv, function(x) mean(x)/sd(x[x<0
> sharper <- round(sharper, 3)
> # Dygraphs plot of pairs Bollinger strategy
> colnamev <- colnames(wealthv)
> caption <- paste("Dynamic Pairs Strategy", "/ \n",
+   paste0(paste(colnamev[1:2], "Sharpe =", sharper), collapse=" / '
+   "Number of trades=", ntrades)
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main=captiont) %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=200)
```

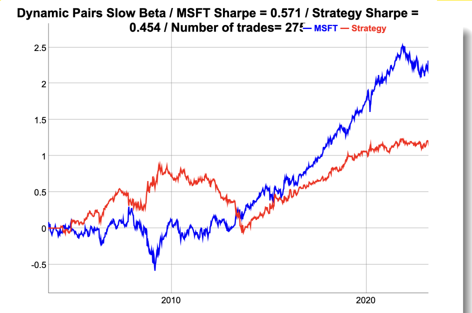
## Pairs Strategy With Slow Beta

The pairs strategy can be improved by introducing an independent decay parameter $\lambda_\beta$ for calculating the trailing stock $\beta$.

The pairs strategy performs better with fast decay for calculating the trailing pairs volatility and slow decay for calculating the trailing stock $\beta$.

Independent decay parameters for the trailing stock $\beta$ and for the trailing pairs volatility improve the pairs strategy performance, but they also increase the risk of overfitting the model, because the more model parameters, the greater the risk of overfitting.

```
> # Calculate the trailing cointegrated pair prices
> covars <- HighFreq::run_covar(pricev, lambda=0.95)
> betav <- covars[, 1]/covars[, 3]
> pricec <- (pricev$MSFT - betav*pricev$XLK)
> # Recalculate the mean of cointegrated portfolio prices
> meanv <- HighFreq::run_mean(pricec, lambda=0.3)
> vars <- sqrt(HighFreq::run_var(pricec, lambda=0.3))
> # Simulate the pairs Bollinger strategy
> pricen <- zoo::coredata(pricec) # Numeric price
> pricem <- pricen - meanv # De-meaned price
> threshd <- rutils::lagit(volat)
> posv <- rep(NA_integer_, nrows)
> posv[1] <- 0
> posv <- ifelse(pricem > threshd, -1, posv)
> posv <- ifelse(pricem < -threshd, 1, posv)
> posv <- zoo::na.locf(posv)
> posv <- rutils::lagit(posv, lagg=1)
> # Calculate the pnls and the number of trades
> retv <- rutils::diffit(pricev)
> pnls <-  posv*(retv$MSFT - betav*retv$XLK)
> ntrades <- sum(abs(rutils::diffit(posv)) > 0)
```



```
> # Calculate the Sharpe ratios
> wealthv <- cbind(retv$MSFT, pnls)
> colnames(wealthv) <- c("MSFT", "Strategy")
> sharper <- sqrt(252)*sapply(wealthv, function(x) mean(x)/sd(x[x<0]
> sharper <- round(sharper, 3)
> # Dygraphs plot of pairs Bollinger strategy
> colnamev <- colnames(wealthv)
> captiont <- paste("Dynamic Pairs Slow Beta", "/ \n",
+   paste0(paste(colnamev[1:2], "Sharpe =", sharper), collapse=" / "
+   "Number of trades=", ntrades)
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main=captiont) %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=200)
```

## Stock Index Weighting Methods

Stock market indices can be either capitalization-weighted, price-weighted, or equal-weighted.

The cap-weighted index is equal to the average of the market capitalizations of all its companies (stock price times number of shares). The *S&P500* index is cap-weighted.

The price-weighted index is equal to the average of the stock prices. The *DJIA* index is price-weighted.

The equal-weighted index is equal to the value (wealth) of the equal-weighted portfolio of stocks.

The equal-weighted portfolio owns equal dollar amounts of each stock, and it rebalances its allocations as market prices change.

The cap-weighted and price-weighted indices are overweight large-cap stocks, compared to the equal-weight index which has larger weights for small-cap stocks.

```
> # Load daily S&P500 log percentage stock returns
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.
> # Subset (select) the stock returns after the start date of VTI
> retvti <- na.omit(rutils::etfenv$returns$VTI)
> colnames(retvti) <- "VTI"
> retv <- returns[zoo::index(retvti)]
> datev <- zoo::index(retv)
> retvti <- retvti[datev]
> nrows <- NROW(retv)
> nstocks <- NCOL(retv)
> head(retv[, 1:5])
> # Replace NA returns with zeros
> retsna <- retv
> retsna[is.na(retsna)] <- 0
```

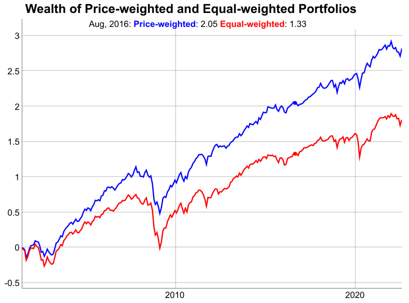
## The Equal-Weight Portfolio

The equal-weight portfolio rebalances its allocations - it sells the stocks with higher returns and buys stocks with lower returns. So it's a *mean reverting* (contrarian) strategy.

The equal-weight portfolio underperforms the cap-weighted and price-weighted indices because it gradually overweights underperforming stocks, as it rebalances to maintain equal dollar allocations.

In periods when a small number of stocks dominate returns, the cap-weighted and price-weighted indices outperform the equal-weighted index.

```
> # Calculate normalized log prices that start at 0
> pricev <- cumsum(retsna)
> head(pricev[, 1:5])
> # Wealth of price-weighted (fixed shares) portfolio
> wealthpw <- rowMeans(exp(pricev))
> # Wealth of equal-weighted portfolio
> wealthew <- exp(rowMeans(pricev))
```



```
> # Calculate combined log wealth
> wealthv <- cbind(wealthpw, wealthew)
> wealthv <- log(wealthv)
> wealthv <- xts::xts(wealthv, datev)
> colnames(wealthv) <- c("Price-weighted", "Equal-weighted")
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(rutils::diffit(wealthv),
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of combined log wealth
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(wealthv[endd],
+   main="Wealth of Price-weighted and Equal-weighted Portfolios")
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

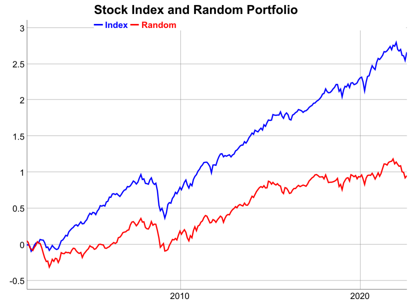
## Random Stock Selection

A random portfolio is a sub-portfolio of stocks selected at random.

Random portfolios are used as a benchmark for stock pickers (portfolio managers).

If a portfolio manager outperforms the median of random portfolios, then they may have stock picking skill.

```
> # Calculate the price-weighted average of all stock prices
> wealthpw <- xts::xts(wealthpw, order.by=datev)
> colnames(wealthpw) <- "Index"
> # Select a random, price-weighted portfolio of 5 stocks
> set.seed(1121)
> samplev <- sample.int(n=nstocks, size=5, replace=FALSE)
> portf <- pricev[, samplev]
> portf <- rowMeans(exp(portf))
```



```
> # Plot dygraph of stock index and random portfolio
> wealthv <- cbind(wealthpw, portf)
> wealthv <- log(wealthv)
> colnames(wealthv)[2] <- "Random"
> colorv <- c("blue", "red")
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(wealthv[endd], main="Stock Index and Random Port
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

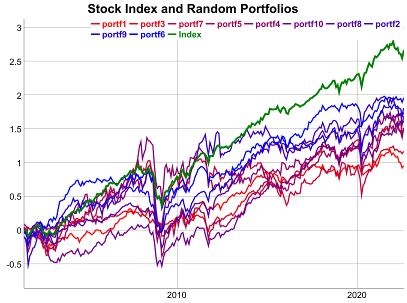
## Random Stock Portfolios

Most random portfolios underperform the index, so picking a portfolio which outperforms the stock index requires great skill.

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.

Therefore the proper benchmark for a stock picker is the median of random portfolios, not the stock index, which is the mean of all the stock prices.

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



```
> # Select 10 random price-weighted sub-portfolios
> set.seed(1121)
> nportf <- 10
> portfs <- sapply(1:nportf, function(x) {
+   portf <- pricev[, sample.int(n=nstocks, size=5, replace=FALSE)]
+   rowMeans(exp(portf))
+ })  # end sapply
> portfs <- xts::xts(portfs, order.by=datev)
> colnames(portfs) <- paste0("portf", 1:nportf)
> # Sort the sub-portfolios according to perfomance
> portfs <- portfs[, order(portfs[nrows])]
> round(head(portfs), 3)
> round(tail(portfs), 3)
```

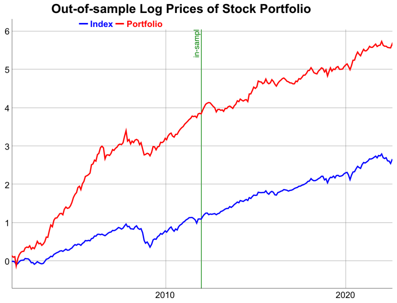
```
> # Plot dygraph of stock index and random portfolios
> colorv <- colorRampPalette(c("red", "blue"))(nportf)
> wealthv <- cbind(wealthpw, portfs)
> wealthv <- log(wealthv)
> colnames(wealthv)[1] <- "Index"
> colnamev <- colnames(wealthv)
> colorv <- c("green", colorv)
> dygraphs::dygraph(wealthv[endd], main="Stock Index and Random Port
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
+   dyLegend(show="always", width=500)
```

## Stock Portfolio Selection Out-of-Sample

The strategy selects the 10 best performing stocks at the end of the in-sample interval, and invests in them in the out-of-sample interval.

The strategy buys equal and fixed number of shares of stocks, and at the end of the in-sample interval, selects the 10 best performing stocks. It then invests the same number of shares in the out-of-sample interval.

The out-of-sample performance of the best performing stocks is not any better than the index.



```
> # Define in-sample and out-of-sample intervals
> cutoff <- nrows %/% 2
> datev[cutoff]
> insample <- 1:cutoff
> outsample <- (cutoff + 1):nrows
> # Calculate the 10 best performing stocks in-sample
> perfstat <- sort(drop(coredata(pricev[cutoff, ])), decreasing=TRU
> symbolv <- names(head(perfstat, 10))
> # Calculate the wealth of the 10 best performing stocks
> wealthv <- pricev[, symbolv]
> wealthv <- rowMeans(exp(wealthv))
```

```
> # Combine the price-weighted wealth with the 10 best performing st
> wealthv <- cbind(wealthpw, wealthv)
> wealthv <- log(wealthv)
> colnames(wealthv)[2] <- "Portfolio"
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(rutils::diffit(wealthv[outsample, ]),
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot out-of-sample stock portfolio returns
> dygraphs::dygraph(wealthv[endd], main="Out-of-sample Log Prices of
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyEvent(datev[cutoff], label="in-sample", strokePattern="solid",
+   dyLegend(width=500)
```

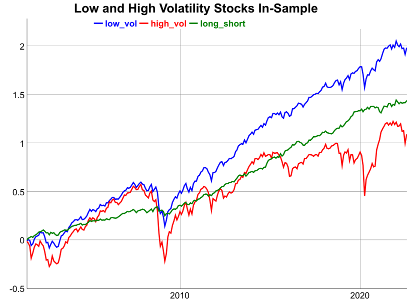
## Low and High Volatility Stock Portfolios

Research by Robeco, Eric Falkenstein, and others has shown that low volatility stocks have outperformed high volatility stocks.

*Betting against volatility* is a strategy which invests in low volatility stocks and shorts high volatility stocks.

*USMV* is an *ETF* that holds low volatility stocks, although it hasn't met expectations.

```
> # Calculate the stock volatilities, betas, and alphas
> riskret <- sapply(retv, function(retv) {
+   retv <- na.omit(retv)
+   std <- sd(retv)
+   retvti <- retvti[zoo::index(retv)]
+   varvti <- drop(var(retvti))
+   meanvti <- mean(retvti)
+   betav <- drop(cov(retv, retvti))/varvti
+   resid <- retv - betav*retvti
+   alphav <- mean(retv) - betav*meanvti
+   c(alpha=alphav, beta=betav, std=std, ivol=sd(resid))
+ })  # end sapply
> riskret <- t(riskret)
> tail(riskret)
> # Calculate the median volatility
> riskv <- riskret[, "std"]
> medianv <- median(riskv)
> # Calculate the returns of low and high volatility stocks
> retlow <- rowMeans(retsna[, names(riskv[riskv<=medianv])])
> rethigh <- rowMeans(retsna[, names(riskv[riskv>medianv])])
> wealthv <- cbind(retlow, rethigh, retlow - 0.5*rethigh)
> wealthv <- xts::xts(wealthv, order.by=datev)
> colnamev <- c("low_vol", "high_vol", "long_short")
> colnames(wealthv) <- colnamev
```



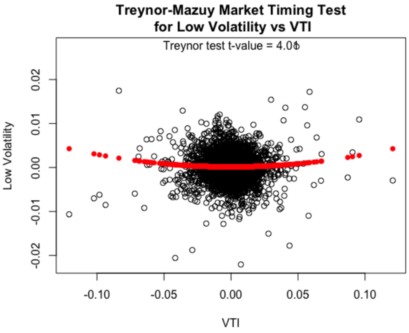
```
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of cumulative returns of low and high volatility stocks
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Volati
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

## Low Volatility Stock Portfolio Market Timing Skill

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

The *Treynor-Mazuy* test shows that the *betting against volatility* strategy has some *market timing* skill.

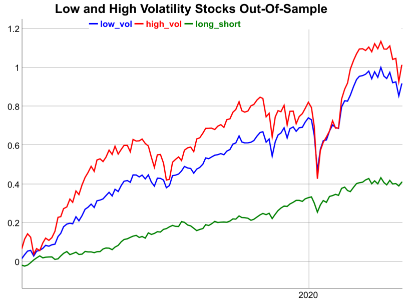
```
> # Merton-Henriksson test
> predm <- cbind(VTI=retvti, 0.5*(retvti+abs(retvti)), retvti^2)
> colnames(predm)[2:3] <- c("merton", "treynor")
> regmod <- lm(wealthv$long_short ~ VTI + merton, data=predm); summa
> # Treynor-Mazuy test
> regmod <- lm(wealthv$long_short ~ VTI + treynor, data=predm); summ
> # Plot residual scatterplot
> resids <- regmod$residuals
> plot.default(x=retvti, y=resids, xlab="VTI", ylab="Low Volatility"
> title(main="Treynor-Mazuy Market Timing Test\n for Low Volatility
> # Plot fitted (predicted) response values
> coefreg <- summary(regmod)$coeff
> fitv <- regmod$fitted.values - coefreg["VTI", "Estimate"]*retvti
> tvalue <- round(coefreg["treynor", "t value"], 2)
> points.default(x=retvti, y=fitv, pch=16, col="red")
> text(x=0.0, y=max(resids), paste("Treynor test t-value =", tvalue))
```



## Low and High Volatility Stock Portfolios Out-Of-Sample

The low volatility stocks selected in-sample also have a higher *Sharpe ratio* in the out-of-sample period than the high volatility stocks, although their absolute returns are similar.

```
> # Calculate the in-sample stock volatilities, betas, and alphas
> riskretis <- sapply(retv[insample], function(retv) {
+   combv <- na.omit(cbind(retv, retvti))
+   if (NROW(combv) > 0) {
+     retv <- na.omit(retv)
+     std <- sd(retv)
+     retvti <- retvti[zoo::index(retv)]
+     varvti <- drop(var(retvti))
+     meanvti <- mean(retvti)
+     betav <- drop(cov(retv, retvti))/varvti
+     resid <- retv - betav*retvti
+     alphav <- mean(retv) - betav*meanvti
+     return(c(alpha=alphav, beta=betav, std=std, ivol=sd(resid)))
+   } else {
+     return(c(alpha=0, beta=0, std=0, ivol=0))
+   }  # end if
+ })  # end sapply
> riskretis <- t(riskretis)
> tail(riskretis)
> # Calculate the median volatility
> riskv <- riskretis[, "std"]
> medianv <- median(riskv)
> # Calculate the out-of-sample returns of low and high volatility
> retlow <- rowMeans(retsna[outsample, names(riskv[riskv<=medianv]
> rethigh <- rowMeans(retsna[outsample, names(riskv[riskv>medianv])])
> wealthv <- cbind(retlow, rethigh, retlow - 0.5*rethigh)
> wealthv <- xts::xts(wealthv, order.by=datev[outsample])
> colnamev <- c("low_vol", "high_vol", "long_short")
> colnames(wealthv) <- colnamev
```



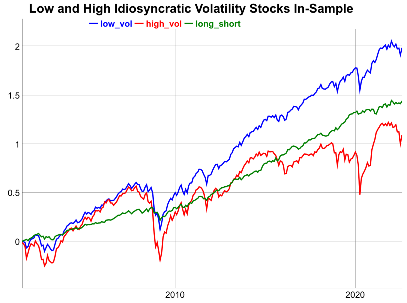
```
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of cumulative returns of low and high volatility stocks
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Volati
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

## Low and High Idiosyncratic Volatility Stock Portfolios

Research by Robeco, Eric Falkenstein, and others has shown that low idiosyncratic volatility stocks have outperformed high volatility stocks.

*Betting against idiosyncratic volatility* is a strategy which invests in low idiosyncratic volatility stocks and shorts high volatility stocks.

```
> # Calculate the median idiosyncratic volatility
> riskv <- riskret[, "ivol"]
> medianv <- median(riskv)
> # Calculate the returns of low and high idiosyncratic volatility s
> retlow <- rowMeans(retsna[, names(riskv[riskv<=medianv])])
> rethigh <- rowMeans(retsna[, names(riskv[riskv>medianv])])
> wealthv <- cbind(retlow, rethigh, retlow - 0.5*rethigh)
> wealthv <- xts::xts(wealthv, order.by=datev)
> colnamev <- c("low_vol", "high_vol", "long_short")
> colnames(wealthv) <- colnamev
```



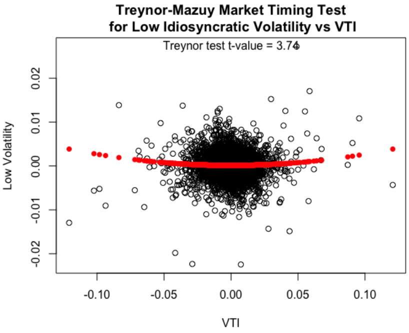
```
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of returns of low and high idiosyncratic volatility stocks
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Idiosy
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

## Low Idiosyncratic Volatility Stock Portfolio Market Timing Skill

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

The *Treynor-Mazuy* test shows that the *betting against idiosyncratic volatility* strategy has some *market timing* skill.

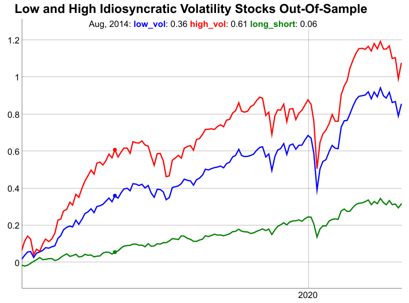
```
> # Merton-Henriksson test
> predm <- cbind(VTI=retvti, 0.5*(retvti+abs(retvti)), retvti^2)
> colnames(predm)[2:3] <- c("merton", "treynor")
> regmod <- lm(wealthv$long_short ~ VTI + merton, data=predm); summa
> # Treynor-Mazuy test
> regmod <- lm(wealthv$long_short ~ VTI + treynor, data=predm); sumr
> # Plot residual scatterplot
> resids <- regmod$residuals
> plot.default(x=retvti, y=resids, xlab="VTI", ylab="Low Volatility'
> title(main="Treynor-Mazuy Market Timing Test\n for Low Idiosyncrat
> # Plot fitted (predicted) response values
> coefreg <- summary(regmod)$coeff
> fitv <- regmod$fitted.values - coefreg["VTI", "Estimate"]*retvti
> tvalue <- round(coefreg["treynor", "t value"], 2)
> points.default(x=retvti, y=fitv, pch=16, col="red")
> text(x=0.0, y=max(resids), paste("Treynor test t-value =", tvalue))
```



## Low and High Idiosyncratic Volatility Stock Portfolios Out-Of-Sample

The low idiosyncratic volatility stocks selected in-sample also have a higher *Sharpe ratio* in the out-of-sample period than the high idiosyncratic volatility stocks, although their absolute returns are similar.

```
> # Calculate the median in-sample idiosyncratic volatility
> riskv <- riskretis[, "ivol"]
> medianv <- median(riskv)
> # Calculate the out-of-sample returns of low and high idiosyncrat
> retlow <- rowMeans(retsna[outsample, names(riskv[riskv<=medianv])]
> rethigh <- rowMeans(retsna[outsample, names(riskv[riskv>medianv])]
> wealthv <- cbind(retlow, rethigh, retlow - 0.5*rethigh)
> wealthv <- xts::xts(wealthv, order.by=datev[outsample])
> colnamev <- c("low_vol", "high_vol", "long_short")
> colnames(wealthv) <- colnamev
```



```
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of out-of-sample returns of low and high volatility stocks
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Idiosy
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

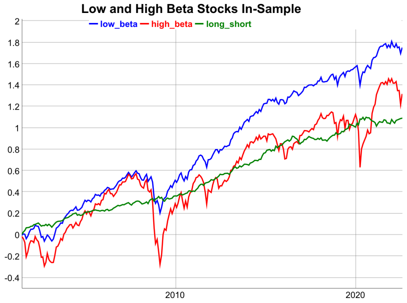
## Low and High Beta Stock Portfolios

Research by NYU professors Andrea Frazzini and Lasse Heje Pedersen has shown that low beta stocks have outperformed high beta stocks, contrary to the *CAPM* model.

The low beta stocks are mostly from defensive stock sectors, like consumer staples, healthcare, etc., which investors buy when they fear a market selloff.

The strategy of investing in low beta stocks and shorting high beta stocks is known as betting against beta.

```
> # Calculate the median beta
> riskv <- riskret[, "beta"]
> medianv <- median(riskv)
> # Calculate the returns of low and high beta stocks
> betalow <- rowMeans(retsna[, names(riskv[riskv<=medianv])])
> betahigh <- rowMeans(retsna[, names(riskv[riskv>medianv])])
> wealthv <- cbind(betalow, betahigh, betalow - 0.5*betahigh)
> wealthv <- xts::xts(wealthv, order.by=datev)
> colnamev <- c("low_beta", "high_beta", "long_short")
> colnames(wealthv) <- colnamev
```



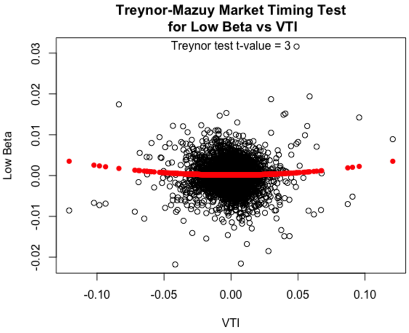
```
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of cumulative returns of low and high beta stocks
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Beta S
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

## Low Beta Stock Portfolio Market Timing Skill

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

The *Treynor-Mazuy* test shows that the *betting against beta* strategy does not have significant *market timing* skill.

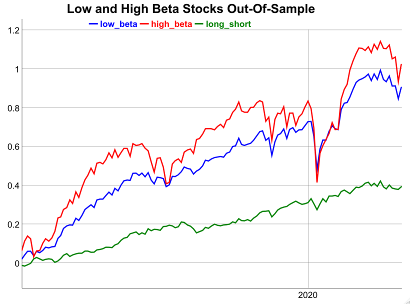
```
> # Merton-Henriksson test
> predm <- cbind(VTI=retvti, 0.5*(retvti+abs(retvti)), retvti^2)
> colnames(predm)[2:3] <- c("merton", "treynor")
> regmod <- lm(wealthv$long_short ~ VTI + merton, data=predm); summa
> # Treynor-Mazuy test
> regmod <- lm(wealthv$long_short ~ VTI + treynor, data=predm); sumr
> # Plot residual scatterplot
> resids <- regmod$residuals
> plot.default(x=retvti, y=resids, xlab="VTI", ylab="Low Beta")
> title(main="Treynor-Mazuy Market Timing Test\n for Low Beta vs VT
> # Plot fitted (predicted) response values
> coefreg <- summary(regmod)$coeff
> fitv <- regmod$fitted.values - coefreg["VTI", "Estimate"]*retvti
> tvalue <- round(coefreg["treynor", "t value"], 2)
> points.default(x=retvti, y=fitv, pch=16, col="red")
> text(x=0.0, y=max(resids), paste("Treynor test t-value =", tvalue))
```



## Low and High Beta Stock Portfolios Out-Of-Sample

The low beta stocks selected in-sample also have a higher *Sharpe ratio* in the out-of-sample period than the high beta stocks, although their absolute returns are similar.

```
> # Calculate the median beta
> riskv <- riskretis[, "beta"]
> medianv <- median(riskv)
> # Calculate the out-of-sample returns of low and high beta stocks
> betalow <- rowMeans(retsna[outsample, names(riskv[riskv<=medianv]
> betahigh <- rowMeans(retsna[outsample, names(riskv[riskv>medianv]
> wealthv <- cbind(betalow, betahigh, betalow - 0.5*betahigh)
> wealthv <- xts::xts(wealthv, order.by=datev[outsample])
> colnamev <- c("low_beta", "high_beta", "long_short")
> colnames(wealthv) <- colnamev
```



```
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of out-of-sample returns of low and high beta stocks
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Low and High Beta S
+   dySeries(name=colnamev[1], col="blue", strokeWidth=1) %>%
+   dySeries(name=colnamev[2], col="red", strokeWidth=1) %>%
+   dySeries(name=colnamev[3], col="green", strokeWidth=2) %>%
+   dyLegend(width=500)
```

## Risk Parity Strategy for Stocks

In the *Risk Parity* strategy the dollar portfolio allocations are rebalanced daily so that their dollar volatilities remain equal.

The dollar amount of stock that has unit dollar volatility is equal to the *standardized prices* $\frac{p_i}{\sigma_i^d}$. Where $\sigma_i^d$ is the dollar volatility.

So the allocations $a_i$ should be proportional to the *standardized prices*: $a_i \propto \frac{p_i}{\sigma_i^d}$,

But the *standardized prices* are equal to the inverse of the percentage volatilities $\sigma_i$: $\frac{p_i}{\sigma_i^d} = \frac{1}{\sigma_i}$, so the allocations $a_i$ are proportional to the inverse of the percentage volatilities $a_i \propto \frac{1}{\sigma_i}$.

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

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

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

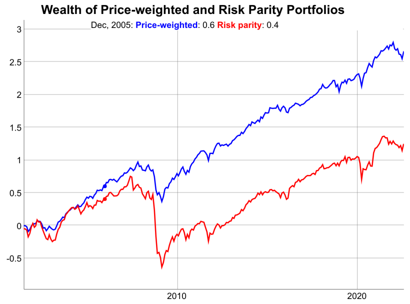
```
> # Calculate the trailing percentage volatilities
> lambda <- 0.6
> volat <- HighFreq::run_var(retsna, lambda=lambda)
> volat <- sqrt(volat)
> # Calculate the risk parity portfolio allocations
> alloc <- ifelse(volat > 1e-4, 1/volat, 0)
> # Scale allocations to 1 dollar total
> allocs <- rowSums(alloc)
> alloc <- ifelse(allocs > 0, alloc/allocs, 0)
> # Lag the allocations
> alloc <- rutils::lagit(alloc)
```

## Risk Parity Strategy for Stocks Performance

The risk parity strategy does not perform well for stocks because their correlations are positive.

The risk parity strategy performs better when the correlations of asset returns are negative, and worse when the correlations are positive.

```
> # Calculate the wealth of risk parity
> wealthrp <- exp(cumsum(rowSums(alloc*retv, na.rm=TRUE)))
> # Combined wealth
> wealthv <- cbind(wealthpw, wealthrp)
> wealthv <- xts::xts(wealthv, datev)
> colnames(wealthv) <- c("Price-weighted", "Risk parity")
> wealthv <- log(wealthv)
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(rutils::diffit(wealthv),
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
```



```
> # Plot of log wealth
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(wealthv[endd],
+   main="Wealth of Price-weighted and Risk Parity Portfolios") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

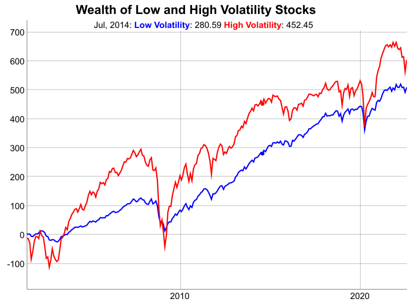
## Stocks With Low and High Trailing Volatilities

The trailing volatilities can be used to create low and high volatility portfolios, which are rebalanced daily.

The low volatility portfolio consists of stocks with trailing volatilities less than the median, and the high portfolio with trailing volatilities greater than the median.

The low volatility portfolio has higher risk-adjusted returns than the high volatility portfolio, which contradicts the *CAPM* model.

```
> # Calculate the median volatilities
> medianv <- matrixStats::rowMedians(volat)
> # Calculate the wealth of low volatility stocks
> alloc <- matrix(integer(nrows*nstocks), ncol=nstocks)
> alloc[volat <= medianv] <- 1
> alloc <- rutils::lagit(alloc)
> retlow <- rowSums(alloc*retv, na.rm=TRUE)
> wealth_lovol <- exp(cumsum(retlow))
> # Calculate the wealth of high volatility stocks
> alloc <- matrix(integer(nrows*nstocks), ncol=nstocks)
> alloc[volat > medianv] <- 1
> alloc <- rutils::lagit(alloc)
> rethigh <- rowSums(alloc*retv, na.rm=TRUE)
> wealth_hivol <- exp(cumsum(rethigh))
```



```
> # Combined wealth
> wealthv <- cbind(wealth_lovol, wealth_hivol)
> wealthv <- xts::xts(wealthv, datev)
> colnames(wealthv) <- c("Low Volatility", "High Volatility")
> wealthv <- log(wealthv)
> # Calculate the Sharpe and Sortino ratios
> retvol <- rutils::diffit(wealthv)
> sqrt(252)*sapply(retvol, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot of log wealth
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(wealthv[endd],
+   main="Wealth of Low and High Volatility Stocks") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

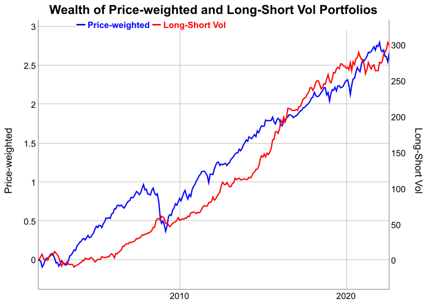
## Long-Short Stock Volatility Strategy

The *Long-Short Volatility* strategy buys the low volatility stock portfolio and shorts the high volatility portfolio.

The high volatility portfolio returns are multiplied by a factor to compensate for their higher volatility.

The *Long-Short Volatility* strategy has higher risk-adjusted returns than the price-weighted portfolio.

```
> # Calculate the volatilities of the low and high volatility stocks
> volat <- HighFreq::run_var(retvol, lambda=lambda)
> volat <- sqrt(volat)
> volat[1:2, ] <- 1
> colnames(volat) <- c("Low Volatility", "High Volatility")
> # Multiply the high volatility portfolio returns by a factor
> factv <- volat[, 1]/volat[, 2]
> factv <- rutils::lagit(factv)
> # Calculate the long-short volatility returns
> retls <- (retlow - factv*rethigh)
> wealthls <- exp(cumsum(retls))
> # Combined wealth
> wealthv <- cbind(wealthpw, wealthls)
> wealthv <- xts::xts(wealthv, datev)
> colnamev <- c("Price-weighted", "Long-Short Vol")
> colnames(wealthv) <- colnamev
> wealthv <- log(wealthv)
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(rutils::diffit(wealthv),
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
```



```
> # Plot of log wealth
> endd <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(wealthv[endd],
+   main="Wealth of Price-weighted and Long-Short Vol Portfolios") %
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeW
+   dyLegend(show="always", width=500)
```

## Homework Assignment

### Required

- Study all the lecture slides in *FRE7241_Lecture_4.pdf*, and run all the code in *FRE7241_Lecture_4.R*

### Recommended

- Download from NYU Classes and read about momentum strategies:
  *Moskowitz Time Series Momentum.pdf*
  *Bouchaud Momentum Mean Reversion Equity Returns.pdf*
  *Hurst Pedersen AQR Momentum Evidence.pdf*