FRE7241 Algorithmic Portfolio Management

Lecture#4, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 28, 2021

NYU | TANDON SCHOOL OF ENGINEERING

Centered Price Z-scores

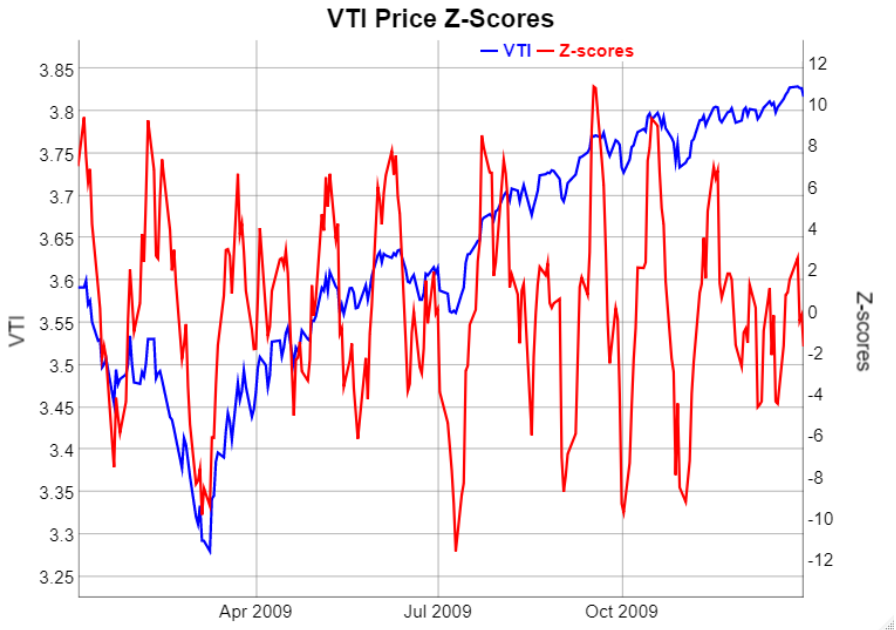
An extreme local price is a price which differs significantly from neighboring prices.

Extreme prices can be identified in-sample using the centered *price z-score* equal to the price difference with neighboring prices divided by the volatility of returns σ_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 prices, with smaller k producing larger z-scores for more local price extremes.




```
> # Extract VTI log OHLC prices
> oh_lc <- log(rutils::etf_env$VTI)
> clos_e <- quantmod::Cl(oh_lc)
> re_turns <- rutils::diff_it(clos_e)
> # Calculate the centered volatility
> look_back <- 21
> half_back <- look_back %/% 2
> vol_at <- roll::roll_sd(re_turns, width=look_back, min_obs=1)
> vol_at <- rutils::lag_it(vol_at, lagg=(-half_back))
> # Calculate the z-scores of prices
> price_scores <- (2*clos_e -
+   rutils::lag_it(clos_e, half_back, pad_zeros=FALSE) -
+   rutils::lag_it(clos_e, -half_back, pad_zeros=FALSE))
> price_scores <- ifelse(vol_at > 0, price_scores/vol_at, 0)
```

```
> # Plot dygraph of z-scores of VTI prices
> price_s <- cbind(clos_e, price_scores)
> colnames(price_s) <- c("VTI", "Z-scores")
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s["2009"], main="VTI Price Z-Scores") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

Labeling the Tops and Bottoms of Prices

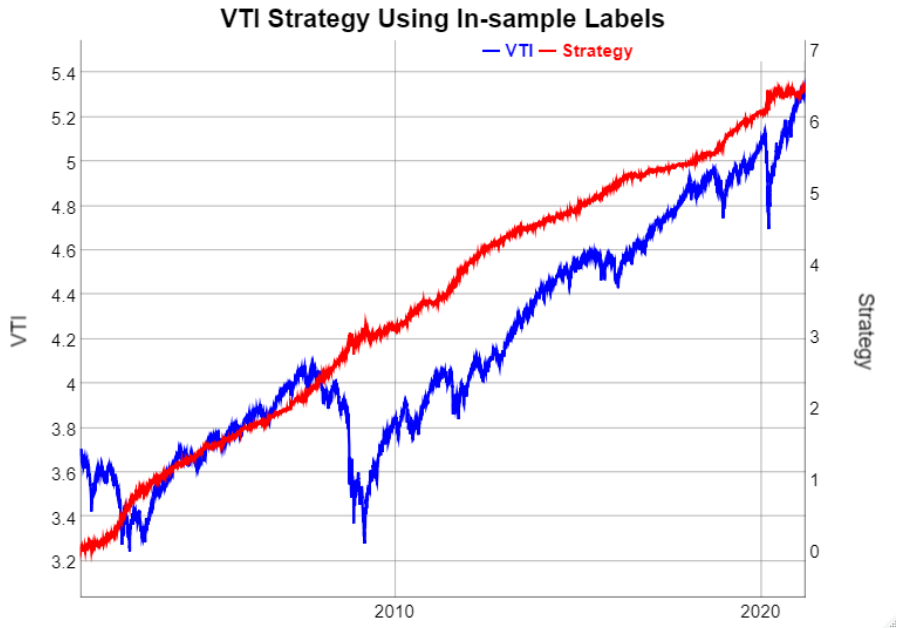
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

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

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

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

```
> # Calculate thresholds for labeling tops and bottoms
> threshold_s <- quantile(price_scores, c(0.1, 0.9))
> # Calculate the vectors of tops and bottoms
> top_s <- zoo::coredata(price_scores > threshold_s[2])
> colnames(top_s) <- "tops"
> bottom_s <- zoo::coredata(price_scores < threshold_s[1])
> colnames(bottom_s) <- "bottoms"
> # Simulate in-sample VTI strategy
> position_s <- rep(NA_integer_, NROW(re_turns))
> position_s[1] <- 0
> position_s[top_s] <- (-1)
> position_s[bottom_s] <- 1
> position_s <- zoo::na.locf(position_s)
> position_s <- rutils::lag_it(position_s)
> pnl_s <- cumsum(re_turns*position_s)
```




```
> # Plot dygraph of in-sample VTI strategy
> price_s <- cbind(clos_e, pnl_s)
> colnames(price_s) <- c("VTI", "Strategy")
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s, main="VTI Strategy Using In-sample Labe
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

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 σ_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 σ_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 volatility z-scores
> vol_at <- HighFreq::roll_var_ohlc(ohlc=oh_lc, look_back=look_back
> volat_mean <- roll::roll_mean(vol_at, width=look_back, min_obs=1)
> volat_sd <- roll::roll_sd(rutils::diff_it(vol_at), width=look_bac
> volat_sd[1] <- 0
> volat_scores <- ifelse(volat_sd > 0, (vol_at - volat_mean)/volat_
> colnames(volat_scores) <- "volat"
> # Calculate volume z-scores
> vol_ume <- quantmod::Vo(oh_lc)
> volume_mean <- roll::roll_mean(vol_ume, width=look_back, min_obs=
> volume_sd <- roll::roll_sd(rutils::diff_it(vol_ume), width=look_ba
> volume_sd[1] <- 0
> volume_scores <- ifelse(volume_sd > 0, (vol_ume - volume_mean)/vol
> colnames(volume_scores) <- "volume"
```

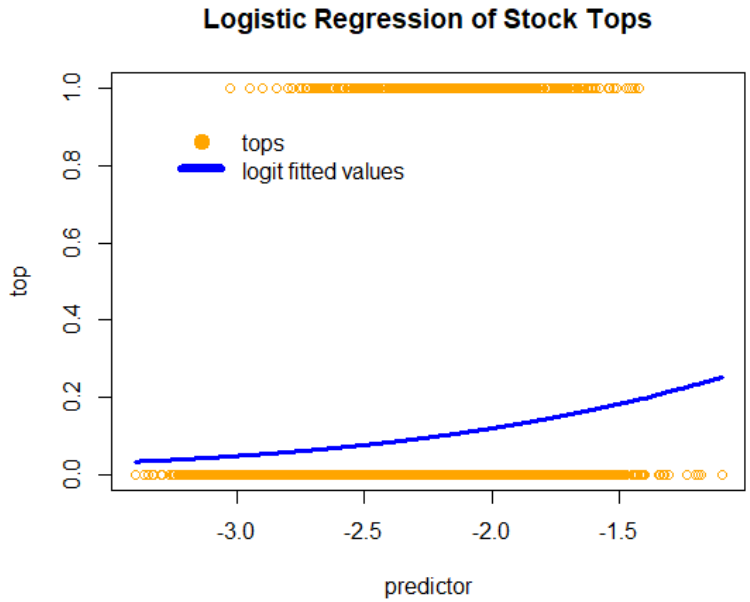
Forecasting Price Extremes Using Logistic Regression

The weighted average of the volatility and trading volume z-scores can be used to calculate the probability of a top (overbought 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 design matrix for tops including intercept column
> de_sign <- cbind(top_s, intercept=rep(1, NROW(top_s)),
+            volat_scores, volume_scores)
> # Define regression formula
> col_names <- colnames(de_sign)
> for_mula <- as.formula(paste(paste(col_names[1],
+   paste(col_names[-1], collapse="+"), sep=" ~ "), "-1"))
> # Fit in-sample logistic regression for tops
> g_lm <- glm(for_mula, data=de_sign, family=binomial(logit))
> summary(g_lm)
> co_eff <- g_lm$coefficients
> pre_dict <- drop(de_sign[, -1] %*% co_eff)
> or_der <- order(pre_dict)
> # Calculate in-sample forecasts from logistic regression model
> forecast_s <- 1/(1+exp(-pre_dict))
> all.equal(g_lm$fitted.values, forecast_s, check.attributes=FALSE)
> hist(forecast_s)
```




```
> x11(width=6, height=5)
> plot(x=pre_dict[or_der], y=top_s[or_der],
+      main="Logistic Regression of Stock Tops",
+      col="orange", xlab="predictor", ylab="top")
> lines(x=pre_dict[or_der], y=g_lm$fitted.values[or_der], col="blue"
> legend(x="topleft", inset=0.1, bty="n", lwd=6,
+  legend=c("tops", "logit fitted values"),
+  col=c("orange", "blue"), lty=c(NA, 1), pch=c(1, NA))
```

Forecasting Errors

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: `top_s = FALSE`.

A *positive* result corresponds to rejecting the null hypothesis, while a *negative* result corresponds to accepting the null hypothesis.

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 there is no default but it's classified as a default.

A *type II* error is the incorrect acceptance of a `FALSE` *null hypothesis* (i.e. a "false negative"), when there is a default but it's classified as no default.

```
> # Define discrimination threshold value
> thresh_old <- quantile(forecast_s, 0.95)
> # Calculate confusion matrix in-sample
> confu_sion <- table(actual=!top_s, forecast=(forecast_s < thresh_
> confu_sion
> # Calculate FALSE positive (type I error)
> sum(!top_s & (forecast_s > thresh_old))
> # Calculate FALSE negative (type II error)
> sum(top_s & (forecast_s < thresh_old))
```

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.

c10*Actual	Forecast: Null is FALSE	Forecast: 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 FALSE positive and FALSE negative rates
> confu_sion <- confu_sion / rowSums(confu_sion)
> c(typeI=confu_sion[2, 1], typeII=confu_sion[1, 2])
```

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

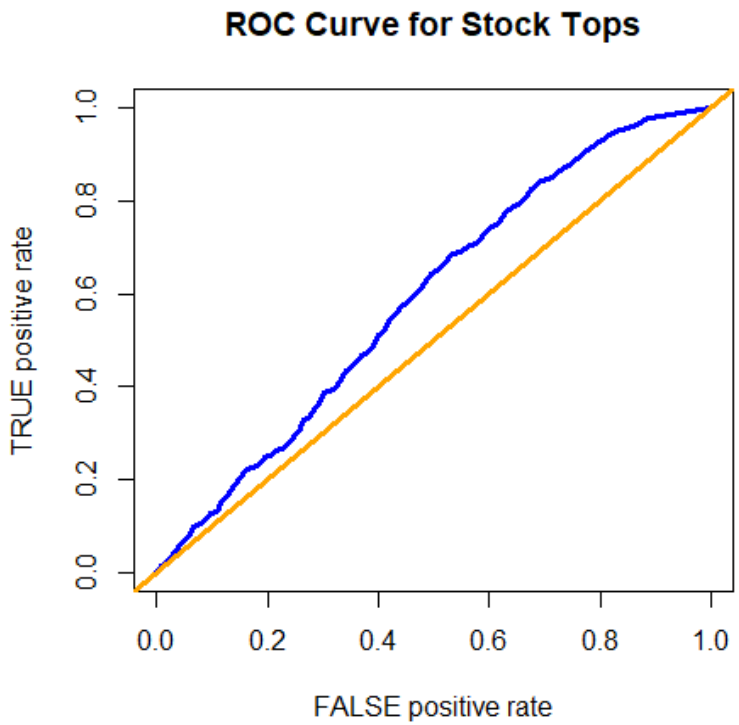
Receiver Operating Characteristic (ROC) Curve for Stock Tops

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 thresh_old
> con_fuse <- function(actual, forecasts, threshold) {
+     conf <- table(actual, (forecasts < threshold))
+     conf <- conf / rowSums(conf)
+     c(typeI=conf[2, 1], typeII=conf[1, 2])
+   }  # end con_fuse
> con_fuse(!top_s, forecast_s, threshold=thresh_old)
> # Define vector of discrimination thresholds
> threshold_s <- quantile(forecast_s, seq(0.1, 0.99, by=0.01))
> # Calculate error rates
> error_rates <- sapply(threshold_s, con_fuse,
+   actual=!top_s, forecasts=forecast_s)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshold_s
> # Calculate the informedness
> inform_ed <- 2 - rowSums(error_rates[, c("typeI", "typeII")])
> plot(threshold_s, inform_ed, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshold_top <- threshold_s[which.max(inform_ed)]
> tops_forecast <- (forecast_s > threshold_top)
```



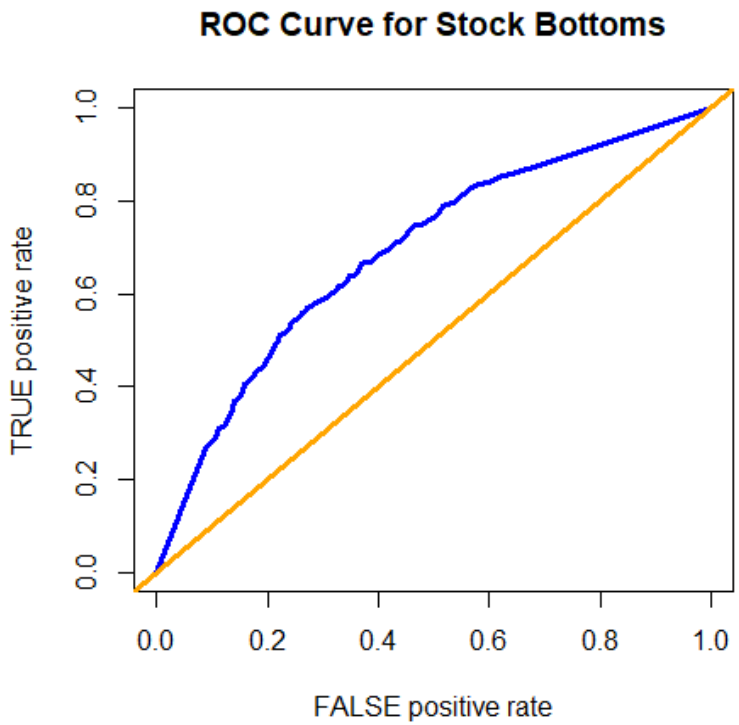

```
> # Calculate area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> true_pos <- (1 - error_rates[, "typeII"])
> true_pos <- (true_pos + rutils::lag_it(true_pos))/2
> false_pos <- rutils::diff_it(error_rates[, "typeI"])
> abs(sum(true_pos*false_pos))
> # Plot ROC Curve for stock tops
> x11(width=5, height=5)
> 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.




```
> # Define design matrix for tops including intercept column
> de_sign <- cbind(bottom_s, intercept=rep(1, NROW(bottom_s)),
+            volat_scores, volume_scores)
> # Define regression formula
> col_names <- colnames(de_sign)
> for_mula <- as.formula(paste(paste(col_names[1],
+   paste(col_names[-1], collapse="+"), sep=" ~ "), "-1"))
> # Fit in-sample logistic regression for tops
> g_lm <- glm(for_mula, data=de_sign, family=binomial(logit))
> summary(g_lm)
> # Calculate in-sample forecast from logistic regression model
> pre_dict <- drop(de_sign[, -1] %*% g_lm$coefficients)
> forecast_s <- 1/(1+exp(-pre_dict))
> # Calculate error rates
> error_rates <- sapply(threshold_s, con_fuse,
+   actual=!bottom_s, forecasts=forecast_s)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshold_s
> # Calculate the informedness
> inform_ed <- 2 - rowSums(error_rates[, c("typeI", "typeII")])
> plot(threshold_s, inform_ed, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshold_bottom <- threshold_s[which.max(inform_ed)]
> bottoms_forecast <- (forecast_s > threshold_bottom)
```

```
> # Calculate area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> true_pos <- (1 - error_rates[, "typeII"])
> true_pos <- (true_pos + rutils::lag_it(true_pos))/2
> false_pos <- rutils::diff_it(error_rates[, "typeI"])
> abs(sum(true_pos*false_pos))
> # Plot ROC Curve for stock tops
> x11(width=5, height=5)
> 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")
```

Labeling the Tops and Bottoms of Prices

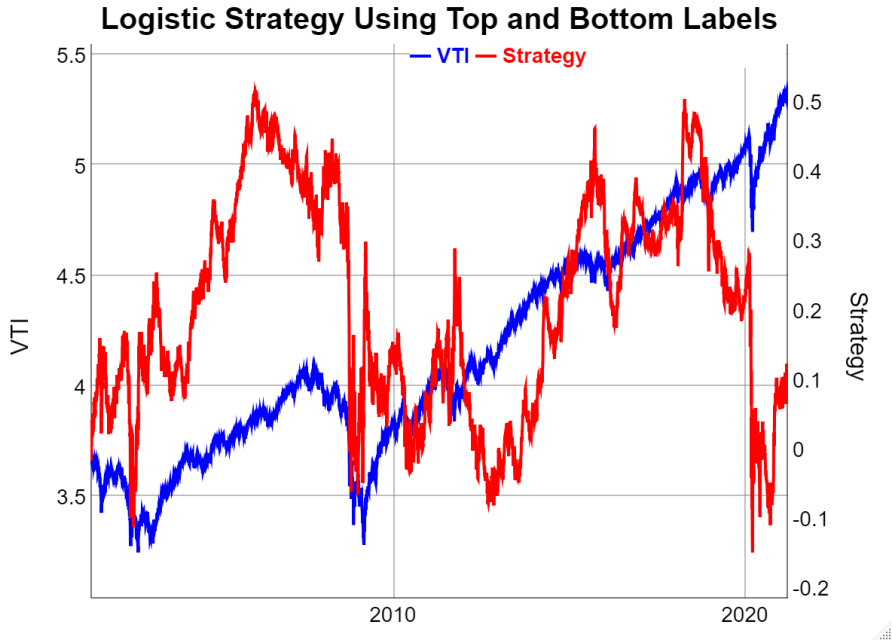
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

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.

```
> # Simulate in-sample VTI strategy
> position_s <- rep(NA_integer_, NROW(re_turns))
> position_s[1] <- 0
> position_s[tops_forecast] <- (-1)
> position_s[bottoms_forecast] <- 1
> position_s <- zoo::na.locf(position_s)
> position_s <- rutils::lag_it(position_s)
> pnl_s <- cumsum(re_turns*position_s)
```




```
> # Plot dygraph of in-sample VTI strategy
> price_s <- cbind(clos_e, pnl_s)
> colnames(price_s) <- c("VTI", "Strategy")
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s, main="Logistic Strategy Using Top and
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

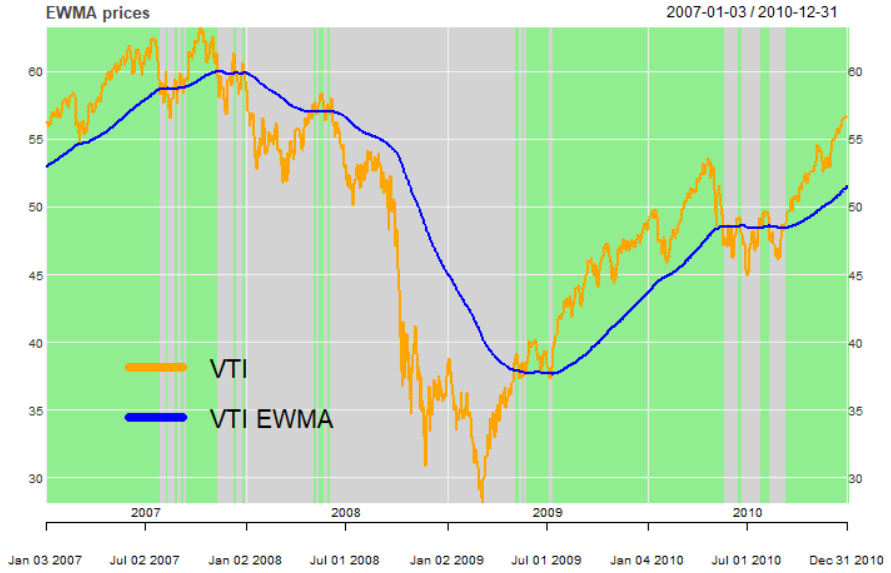
Simulating the *EWMA* Crossover Strategy

In the trend following *EWMA Crossover* strategy, the risk position switches depending if the current price is above or below the *EWMA*.

If the current price crosses above the *EWMA*, then the strategy switches its risk position to a fixed unit of long risk, and if it crosses below, to a fixed unit of short risk.

The strategy holds the same position until the *EWMA* crosses over the current price (either from above or below), and then it switches its position.

The strategy is therefore always either in a long risk, or in a short risk position.




```
> # Determine trade dates right after EWMA has crossed prices
> in_dic <- sign(clos_e - ew_ma[, 2])
> trade_dates <- (rutils::diff_it(in_dic) != 0)
> trade_dates <- which(trade_dates) + 1
> trade_dates <- trade_dates[trade_dates < n_rows]
> # Calculate positions, either: -1, 0, or 1
> position_s <- rep(NA_integer_, n_rows)
> position_s[1] <- 0
> position_s[trade_dates] <- in_dic[trade_dates-1]
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> position_s <- xts::xts(position_s, order.by=index(oh_lc))
```

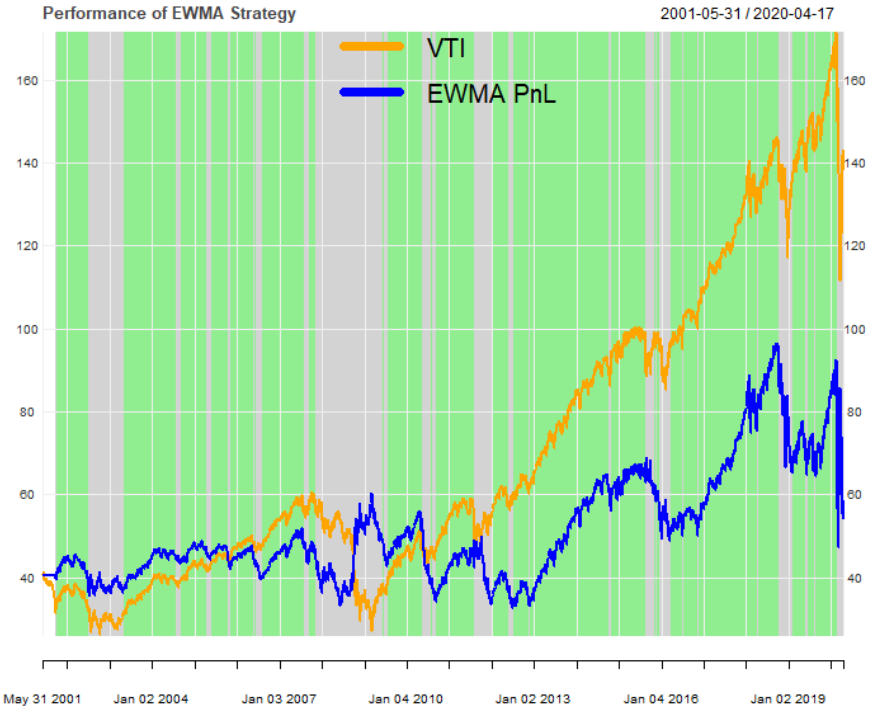
```
> # Plot EWMA prices with position shading
> quantmod::chart_Series(ew_ma["2007/2010"], theme=plot_theme,
+        name="EWMA prices")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("bottomleft", legend=colnames(ew_ma),
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Performance of *EWMA* Crossover Strategy

The strategy trades at the *Open* price on the next day after prices cross the *EWMA*, since in practice it may not be possible to trade immediately.

The Profit and Loss (*PnL*) on a trade date is the sum of the realized *PnL* from closing the old position, plus the unrealized *PnL* after opening the new position.

```
> # Calculate daily profits and losses
> # Calculate pnl for days without trade
> pnl_s <- rutils::diff_it(clos_e)*position_s
> # Calculate realized pnl for days with trade
> close_lag <- rutils::lag_it(clos_e)
> pos_lagged <- rutils::lag_it(position_s)
> pnl_s[trade_dates] <- pos_lagged[trade_dates]*
+   (op_en[trade_dates] - close_lag[trade_dates])
> # Calculate unrealized pnl for days with trade
> pnl_s[trade_dates] <- pnl_s[trade_dates] +
+   position_s[trade_dates]*
+   (clos_e[trade_dates] - op_en[trade_dates])
> # Annualized Sharpe ratio of EWMA strategy
> sqrt(252)*sum(pnl_s)/sd(pnl_s)/NROW(pnl_s)
> # Cumulative pnls
> cum_pnls <- star_t + cumsum(pnl_s)
> cum_pnls <- cbind(clos_e, cum_pnls)
> colnames(cum_pnls) <- c("VTI", "EWMA PnL")
```




```
> # Plot EWMA PnL with position shading
> quantmod::chart_Series(cum_pnls, theme=plot_theme,
+        name="Performance of EWMA Strategy")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(cum_pnls),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

EWMA Crossover Strategy With Transaction Costs

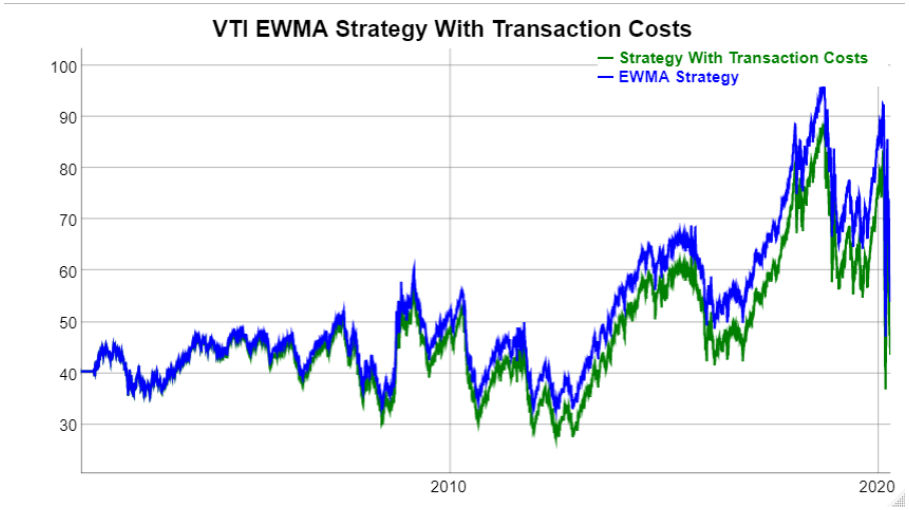
The *bid-offer spread* is the percentage difference between the *offer* minus the *bid* price, divided by the *mid* price.

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

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

$$c^r = \frac{\delta}{2} |\Delta n_t| \, p_t$$

Where Δn_t is the number of shares traded, and p_t is their price.




```
> # bid_offer equal to 10 bps for liquid ETFs
> bid_offer <- 0.001
> # Calculate transaction costs
> cost_s <- 0.5*bid_offer*abs(pos_lagged - position_s)*clos_e
> # pnl_s <- (pnl_s - cost_s)
> # Plot strategy with transaction costs
> cum_pnls <- star_t + cumsum(pnl_s)
> cum_pnls <- cbind(cum_pnls, cum_pnls - cumsum(cost_s))
> colnames(cum_pnls) <- c(sym_bol, "costs")
> dygraphs::dygraph(cum_pnls, main=paste(sym_bol, "EWMA Strategy Wit
+   dySeries(name="costs", label="Strategy With Transaction Costs",
+   dySeries(name=sym_bol, label="EWMA Strategy", strokeWidth=2, col
```

Backtesting Function for *EWMA* Crossover Strategy

The *EWMA* strategy can be simulated by a single function, which allows the analysis of its performance depending on its parameters.

The function `simu_ewma()` performs a simulation of the *EWMA* strategy, given an *OHLC* time series of prices, and a decay parameter λ.

The function `simu_ewma()` returns the *EWMA* strategy positions and returns, in a two-column *xts* time series.

```
> simu_ewma <- function(oh_lc, lamb_da=0.01, wid_th=351, bid_offer=0.001, tre_nd=
+   n_rows <- NROW(oh_lc)
+   # Calculate EWMA prices
+   weight_s <- exp(-lamb_da*1:wid_th)
+   weight_s <- weight_s/sum(weight_s)
+   clos_e <- quantmod::Cl(oh_lc)
+   ew_ma <- .Call(stats:::C_cfilter, clos_e, filter=weight_s, sides=1, circular=
+   ew_ma[1:(wid_th-1)] <- ew_ma[wid_th]
+   # Determine trade dates right after EWMA has crossed prices
+   in_dic <- tre_nd*sign(clos_e - ew_ma)
+   trade_dates <- (rutils::diff_it(in_dic) != 0)
+   trade_dates <- which(trade_dates) + 1
+   trade_dates <- trade_dates[trade_dates < n_rows]
+   # Calculate positions, either: -1, 0, or 1
+   position_s <- rep(NA_integer_, n_rows)
+   position_s[1] <- 0
+   position_s[trade_dates] <- in_dic[trade_dates-1]
+   position_s <- zoo::na.locf(position_s, na.rm=FALSE)
+   op_en <- quantmod::Op(oh_lc)
+   close_lag <- rutils::lag_it(clos_e)
+   pos_lagged <- rutils::lag_it(position_s)
+   # Calculate daily profits and losses
+   pnl_s <- rutils::diff_it(clos_e)*position_s
+   pnl_s[trade_dates] <- pos_lagged[trade_dates]*
+     (op_en[trade_dates] - close_lag[trade_dates])
+   pnl_s[trade_dates] <- pnl_s[trade_dates] +
+     position_s[trade_dates]*
+     (clos_e[trade_dates] - op_en[trade_dates])
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*abs(pos_lagged - position_s)*clos_e
+   pnl_s <- (pnl_s - cost_s)
+   # Calculate strategy returns
+   pnl_s <- cbind(position_s, pnl_s)
+   colnames(pnl_s) <- c("positions", "pnls")
+   pnl_s
+ }  # end simu_ewma
```

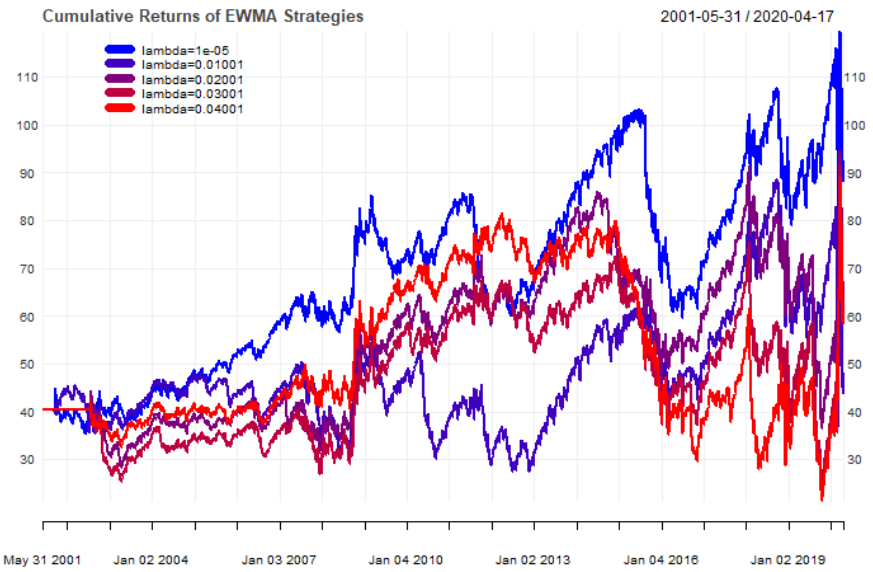
Simulating Multiple Trend Following *EWMA* Strategies

Multiple *EWMA* strategies can be simulated by calling the function `simu_ewma()` in a loop over a vector of λ parameters.

But `simu_ewma()` returns an *xts* time series, and `sapply()` cannot merge *xts* time series together.

So instead the loop is performed using `lapply()` which returns a list of *xts*, and the list is merged into a single *xts* using the functions `do.call()` and `cbind()`.

```
> source("C:/Develop/lecture_slides/scripts/ewma_model.R")
> lamb_das <- seq(from=1e-5, to=0.05, by=0.01)
> # Perform lapply() loop over lamb_das
> pnl_s <- lapply(lamb_das, function(lamb_da) {
+   # Simulate EWMA strategy and calculate re_turns
+   star_t + cumsum(simu_ewma(oh_lc=oh_lc,
+     lamb_da=lamb_da, wid_th=wid_th)[, "pnls"])
+ })  # end lapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
```




```
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> quantmod::chart_Series(pnl_s, theme=plot_theme,
+   name="Cumulative Returns of EWMA Strategies")
> legend("topleft", legend=colnames(pnl_s), inset=0.1,
+   bg="white", cex=0.8, lwd=rep(6, NCOL(pnl_s)),
+   col=plot_theme$col$line.col, bty="n")
```

Simulating *EWMA* Strategies Using Parallel Computing

Simulating *EWMA* strategies naturally lends itself to parallel computing, since the simulations are independent from each other.

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

The resulting list of time series can then be collapsed into a single *xts* series using the functions `rutils::do_call()` and `cbind()`.

```
> # Initialize compute cluster under Windows
> library(parallel)
> clus_ter <- makeCluster(detectCores()-1)
> clusterExport(clus_ter,
+   varlist=c("oh_lc", "wid_th", "simu_ewma"))
> # Perform parallel loop over lamb_das under Windows
> pnl_s <- parLapply(clus_ter, lamb_das, function(lamb_da) {
+   library(quantmod)
+   # Simulate EWMA strategy and calculate re_turns
+   star_t + cumsum(simu_ewma(oh_lc=oh_lc,
+     lamb_da=lamb_da, wid_th=wid_th)[, "pnls"])
+ })  # end parLapply
> # Perform parallel loop over lamb_das under Mac-OSX or Linux
> re_turns <- mclapply(lamb_das, function(lamb_da) {
+   library(quantmod)
+   # Simulate EWMA strategy and calculate re_turns
+   star_t + cumsum(simu_ewma(oh_lc=oh_lc,
+     lamb_da=lamb_da, wid_th=wid_th)[, "pnls"])
+ })  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster under Windo
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
```

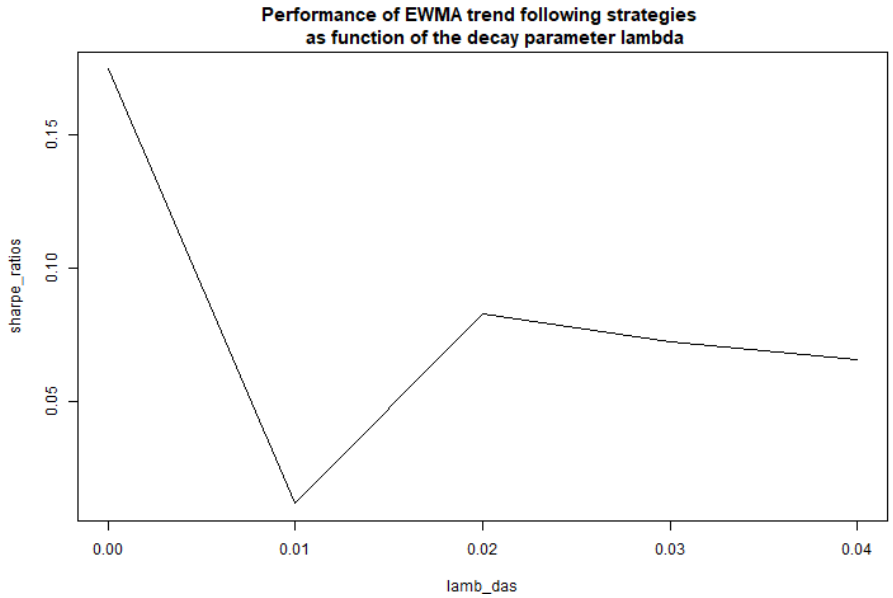
Performance of Trend Following *EWMA* Strategies

The *Sharpe ratios* of *EWMA* strategies with different λ parameters can be calculated by performing an `sapply()` loop over the *columns* of returns.

`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns.

Performing loops in `R` over the *columns* of returns is acceptable, but `R` loops over the *rows* of returns should be avoided.

The performance of trend following *EWMA* strategies depends on the λ parameter, with larger λ parameters performing worse than smaller ones.

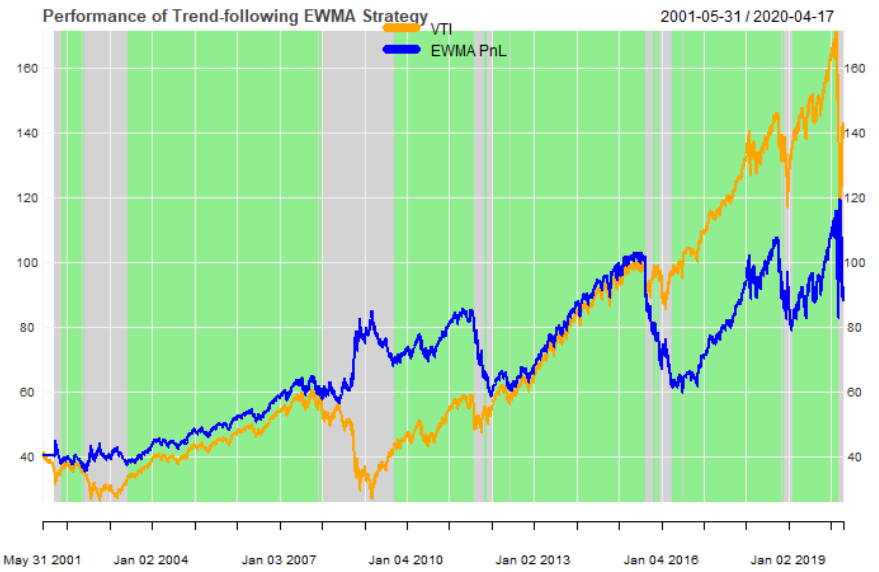



```
> sharpe_ratios <- sqrt(252)*sapply(pnl_s, function(x_ts) {
+   # Calculate annualized Sharpe ratio of strategy returns
+   x_ts <- rutils::diff_it(x_ts)
+   sum(x_ts)/sd(x_ts)
+ })/NROW(pnl_s)  # end sapply
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      main="Performance of EWMA trend following strategies
+      as function of the decay parameter lambda")
> trend_returns <- rutils::diff_it(pnl_s)
> trend_sharpe <- sharpe_ratios
```

Optimal Trend Following *EWMA* Strategy

The best performing trend following *EWMA* strategy has a relatively small λ parameter, corresponding to slower weight decay (giving more weight to past prices), and producing less frequent trading.

```
> # Simulate best performing strategy
> ewma_trend <- simu_ewma(oh_lc=oh_lc,
+   lamb_da=lamb_das[which.max(sharpe_ratios)],
+   wid_th=wid_th)
> position_s <- ewma_trend[, "positions"]
> pnl_s <- star_t + cumsum(ewma_trend[, "pnls"])
> pnl_s <- cbind(clos_e, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
> # Plot EWMA PnL with position shading
> plot_theme$col$line.col <- c("orange", "blue")
> quantmod::chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of Trend Following EWMA Strategy")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

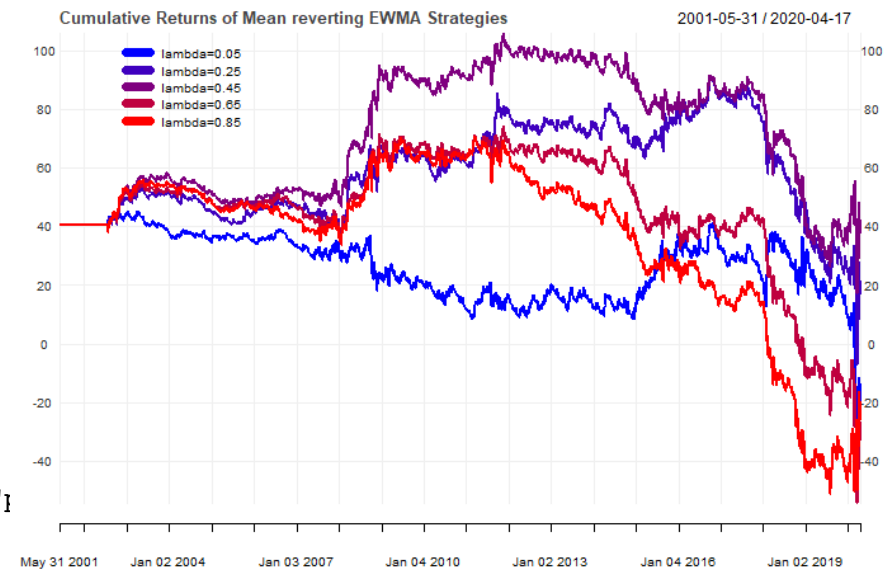


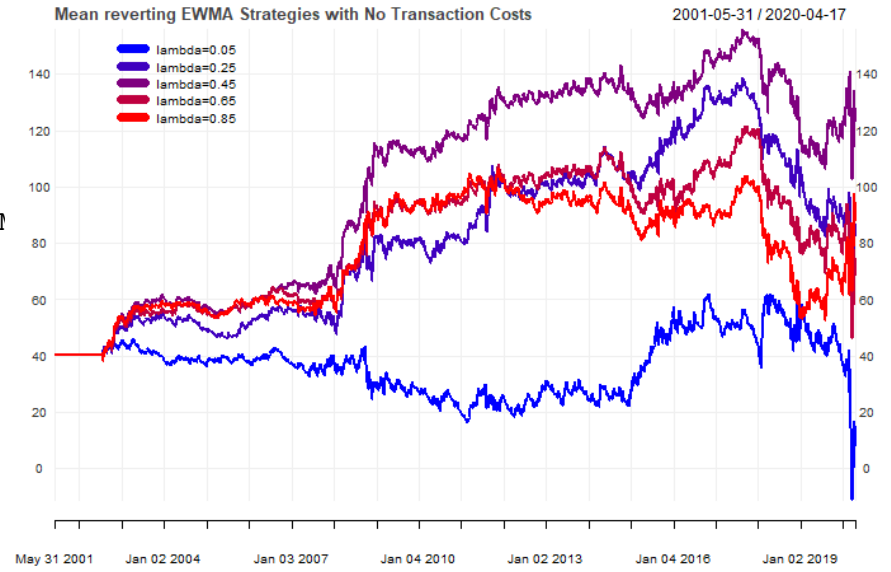
Backtesting Multiple Mean Reverting *EWMA* Strategies

Mean reverting *EWMA* strategies can be simulated using function `simu_ewma()` with argument `tre_nd=(-1)`.

If transaction costs could be reduced by using limit orders, then the profitability of mean reverting strategies could be significantly improved.

```
> source("C:/Develop/lecture_slides/scripts/ewma_model.R")
> lamb_das <- seq(0.05, 1.0, 0.05)
> # Perform lapply() loop over lamb_das
> pnl_s <- lapply(lamb_das, function(lamb_da) {
+   # Backtest EWMA strategy and calculate re_turns
+   star_t + cumsum(simu_ewma(
+     oh_lc=oh_lc, lamb_da=lamb_da, wid_th=wid_th, tre_nd=(-1))[, "
+ })  # end lapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
> # Plot EWMA strategies with custom line colors
> column_s <- seq(1, NCOL(pnl_s), by=4)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NROW(column_s))
> quantmod::chart_Series(pnl_s[, column_s],
+   theme=plot_theme, name="Cumulative Returns of Mean Reverting EWM
> legend("topleft", legend=colnames(pnl_s[, column_s]),
+   inset=0.1, bg="white", cex=0.8, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```





Performance of Mean Reverting *EWMA* Strategies

The *Sharpe ratios* of *EWMA* strategies with different λ parameters can be calculated by performing an `sapply()` loop over the *columns* of returns.

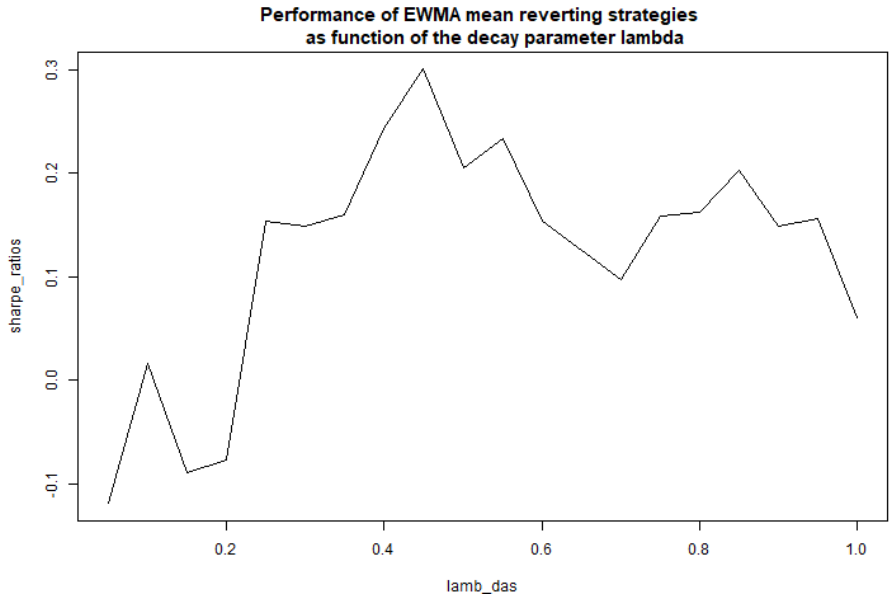
`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns.

Performing loops in `R` over the *columns* of returns is acceptable, but `R` loops over the *rows* of returns should be avoided.

The performance of mean reverting *EWMA* strategies depends on the λ parameter, with performance decreasing for very small or very large λ parameters.

For too large λ parameters, the trading frequency is too high, causing high transaction costs.

For too small λ parameters, the trading frequency is too low, causing the strategy to miss profitable trades.




```
> sharpe_ratios <- sqrt(252)*sapply(pnl_s, function(x_ts) {
+   # Calculate annualized Sharpe ratio of strategy returns
+   x_ts <- rutils::diff_it(x_ts)
+   sum(x_ts)/sd(x_ts)
+ })/NROW(pnl_s)  # end sapply
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      main="Performance of EWMA mean reverting strategies
+      as function of the decay parameter lambda")
> revert_returns <- rutils::diff_it(pnl_s)
> revert_sharpe <- sharpe_ratios
```

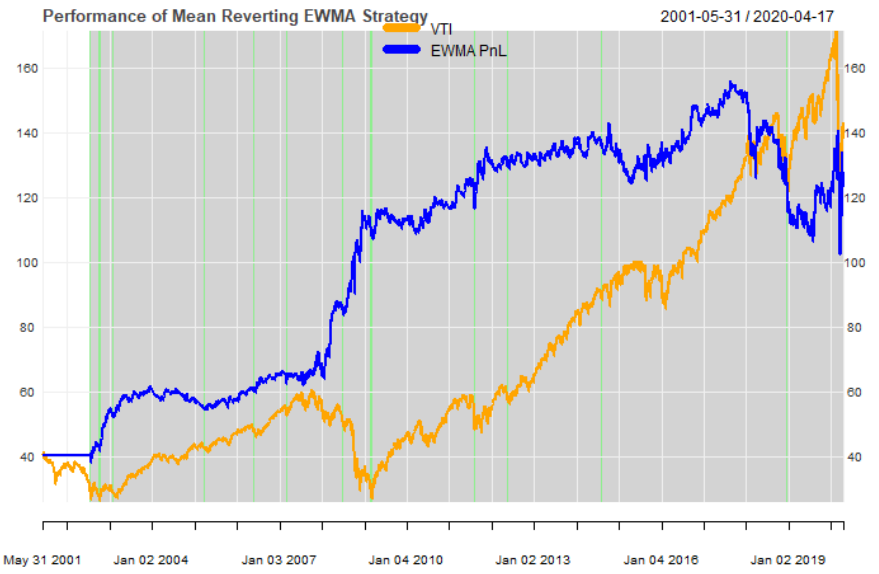
Optimal Mean Reverting *EWMA* Strategy

Reverting the direction of the trend following *EWMA* strategy creates a mean reverting strategy.

The best performing mean reverting *EWMA* strategy has a relatively large λ parameter, corresponding to faster weight decay (giving more weight to recent prices), and producing more frequent trading.

But a too large λ parameter also causes very high trading frequency, and high transaction costs.

```
> # Backtest best performing strategy
> ewma_revert <- simu_ewma(oh_lc=oh_lc, bid_offer=0.0,
+   lamb_da=lamb_das[which.max(sharpe_ratios)],
+   wid_th=wid_th, tre_nd=(-1))
> position_s <- ewma_revert[, "positions"]
> pnl_s <- star_t + cumsum(ewma_revert[, "pnls"])
> pnl_s <- cbind(clos_e, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
```

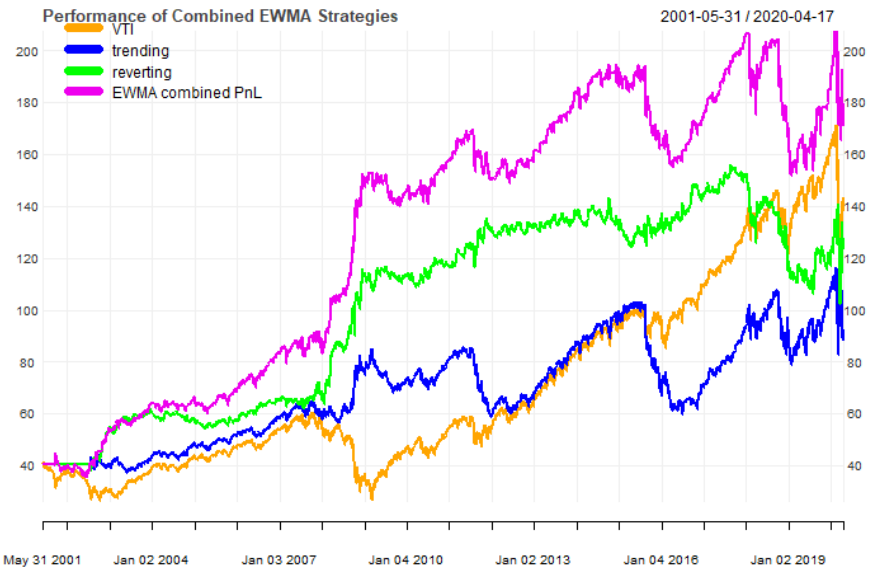



```
> # Plot EWMA PnL with position shading
> plot_theme$col$line.col <- c("orange", "blue")
> quantmod::chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of Mean Reverting EWMA Strategy")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Combining Trend Following and Mean Reverting Strategies

The returns of trend following and mean reverting strategies are usually negatively correlated to each other, so combining them can achieve significant diversification of risk.

```
> # Calculate correlation between trend following and mean revertin
> trend_ing <- ewma_trend[, "pnls"]
> colnames(trend_ing) <- "trend"
> revert_ing <- ewma_revert[, "pnls"]
> colnames(revert_ing) <- "revert"
> close_rets <- rutils::diff_it(clos_e)
> cor(cbind(trend_ing, revert_ing, close_rets))
> # Calculate combined strategy
> com_bined <- trend_ing + revert_ing
> colnames(com_bined) <- "combined"
> # Calculate annualized Sharpe ratio of strategy returns
> re_turns <- cbind(close_rets, trend_ing, revert_ing, com_bined)
> sqrt(252)*sapply(re_turns, function(x_ts)
+   sum(x_ts)/sd(x_ts))/NROW(com_bined)
> pnl_s <- lapply(re_turns, function(x_ts) {star_t + cumsum(x_ts)}
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- c("VTI", "trending", "reverting", "EWMA combi
```



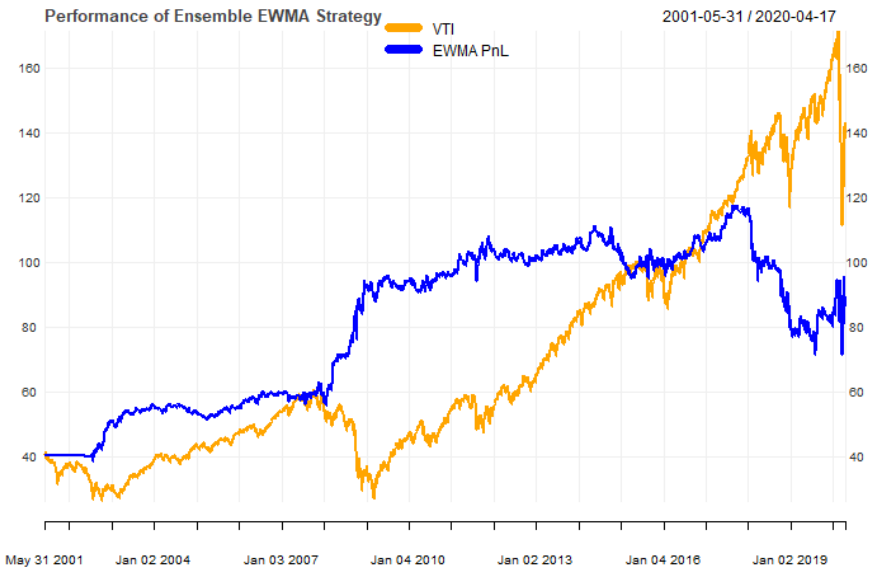

```
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue", "green", "magenta2"
> quantmod::chart_Series(pnl_s, theme=plot_theme,
+         name="Performance of Combined EWMA Strategies")
> legend("topleft", legend=colnames(pnl_s),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Ensemble of *EWMA* Strategies

Instead of selecting the best performing *EWMA* strategy, one can choose a weighted average of strategies (ensemble), which corresponds to allocating positions according to the weights.

The weights can be chosen to be proportional to the Sharpe ratios of the *EWMA* strategies.

```
> weight_s <- c(trend_sharpe, revert_sharpe)
> weight_s[weight_s<0] <- 0
> weight_s <- weight_s/sum(weight_s)
> re_turns <- cbind(trend_returns, revert_returns)
> avg_returns <- re_turns %*% weight_s
> avg_returns <- xts::xts(avg_returns, order.by=index(re_turns))
> pnl_s <- (star_t + cumsum(avg_returns))
> pnl_s <- cbind(clos_e, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
> # Plot EWMA PnL without position shading
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> quantmod::chart_Series(pnl_s, theme=plot_theme,
+   name="Performance of Ensemble EWMA Strategy")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

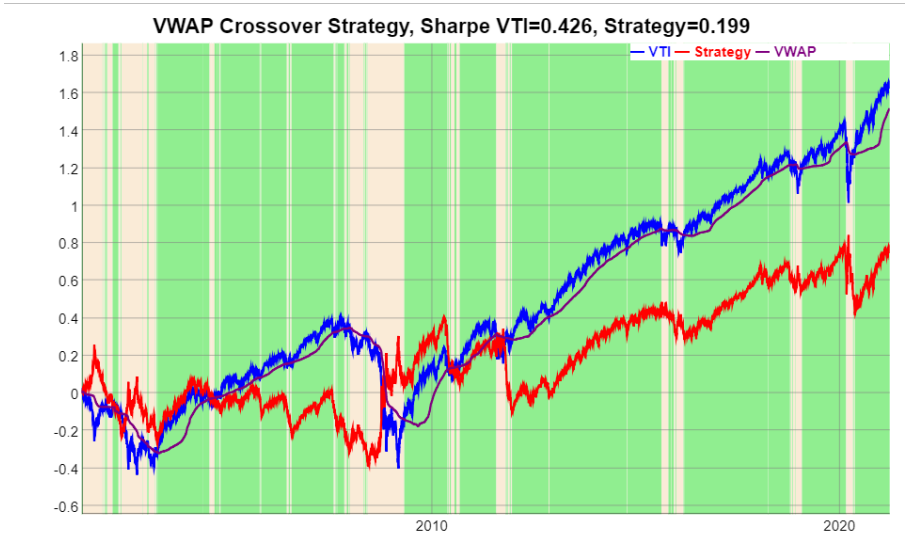


Moving Average Crossover Strategy

In the *Moving Average Crossover* strategy, when the current price crosses above the *VWAP*, then the strategy switches its position to long risk, and vice versa.

A single-period time lag is applied to the *VWAP indicator*, so that the strategy trades immediately after the *VWAP indicator* is evaluated at the end of the day.

This assumption may be too optimistic because in practice it's difficult to trade immediately just before the close of markets.




```
> # Calculate VWAP positions
> position_s <- sign(vwap_fast - vwap_slow)
> # Lag the positions to avoid data snooping
> position_s <- rutils::lag_it(position_s)
> # Calculate daily profits and losses of strategy
> pnl_s <- re_turns*position_s
> colnames(pnl_s) <- "Strategy"
> cum_pnls <- cumsum(pnl_s)
> weal_th <- cbind(cum_rets, cum_pnls, v_wap)
> colnames(weal_th) <- c(sym_bol, "Strategy", "VWAP")
> # Annualized Sharpe ratios of VTI and VWAP strategy
> sharp_e <- sqrt(252)*sapply(cbind(re_turns, pnl_s), function (x)
> # Calculate index for background shading
> in_dic <- (cum_rets > v_wap)
> whi_ch <- which(rutils::diff_it(in_dic) != 0)
> in_dic <- rbind(first(in_dic), in_dic[whi_ch, ], last(in_dic))
> date_s <- index(in_dic)
> in_dic <- ifelse(drop(coredata(in_dic)), "lightgreen", "antiquewl
```

```
> # Create dygraph object without rendering it
> dy_graph <- dygraphs::dygraph(weal_th, main=paste("VWAP Crossover
+   dyOptions(colors=c("blue", "red", "purple"), strokeWidth=2)
> # Add shading
> for (i in 1:(NROW(in_dic)-1)) {
+     dy_graph <- dy_graph %>%
+ dyShading(from=date_s[i], to=date_s[i+1], color=in_dic[i])
+ }  # end for
> # Render the dygraph object
> dy_graph
> # Plot VTI and VWAP strategy using quantmod
> quantmod::chart_Series(x=cbind(cum_rets, cum_pnls),
+   name="VWAP Crossover Strategy for VTI", theme=plot_theme)
> add_TA(position_s > 0, on=-1, col="lightgreen", border="lightgreen
> add_TA(position_s < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("top", legend=c(sym_bol, "VWAP strategy"), lty=1, lwd=6,
+  cex=0.9, inset=0.1, bg="white", col=c("blue", "red"), bty="n")
```

MA Crossover Strategy With Lag

The *MA Crossover* strategy suffers losses when prices are range-bound without a trend, because whenever it switches position the prices soon change direction. (This is called a "whipsaw".)

To prevent whipsaws and over-trading, the *MA Crossover* strategy may choose to delay switching positions until the indicator repeats the same value for several periods.

There's a tradeoff between switching positions too early and risking a whipsaw, and waiting too long and missing a trend.

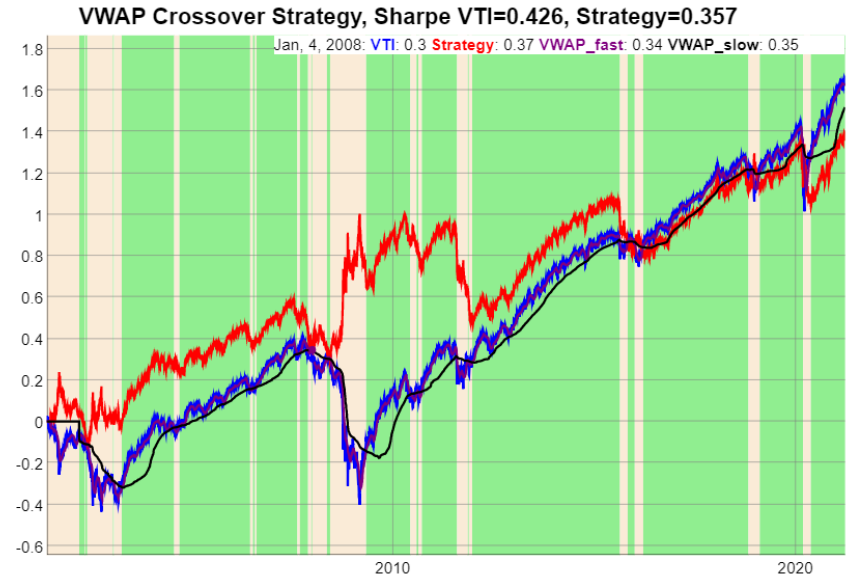

```
> # Calculate positions from lagged indicator
> lagg <- 2
> in_dic <- sign(cum_rets - v_wap)
> indic_sum <- roll::roll_sum(in_dic, width=lagg)
> indic_sum[1:lagg] <- 0
> position_s <- rep(NA_integer_, NROW(clos_e))
> position_s[1] <- 0
> position_s <- ifelse(indic_sum == lagg, 1, position_s)
> position_s <- ifelse(indic_sum == (-lagg), -1, position_s)
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> # Lag the positions to trade in next period
> position_s <- rutils::lag_it(position_s, lagg=1)
> # Calculate PnLs of lagged strategy
> pnl_s <- re_turns*position_s
> colnames(pnl_s) <- "Strategy"
```

```
> cum_pnls_lag <- cumsum(pnl_s)
> weal_th <- cbind(cum_pnls, cum_pnls_lag)
> colnames(weal_th) <- c("Strategy", "Strategy_lag")
> # Annualized Sharpe ratios of VWAP strategies
> sharp_e <- sqrt(252)*sapply(cbind(re_turns, pnl_s),
+   function (x) mean(x)/sd(x))
> # Plot both strategies
> dygraphs::dygraph(weal_th, main=paste("VWAP Crossover Strategy, Sh
+   dyOptions(colors=c("blue", "red"), strokeWidth=3)
```

Dual VWAP Crossover Strategy

The fast-moving *VWAP* is calculated over a short look-back interval, while the slow-moving *VWAP* is calculated over a longer interval.

The trend following reverses direction when the fast-moving *VWAP* crosses the slow-moving one.




```
> # Calculate fast and slow VWAPs
> vwap_fast <- TTR::VWAP(cum_rets, volume=vol_ume, n=20)
> vwap_fast[1:20] <- 0
> vwap_slow <- TTR::VWAP(cum_rets, volume=vol_ume, n=200)
> vwap_slow[1:200] <- 0
> # Calculate VWAP positions
> position_s <- sign(vwap_fast - vwap_slow)
> # Lag the positions to avoid data snooping
> position_s <- rutils::lag_it(position_s)
> # Calculate daily profits and losses of strategy
> pnl_s <- re_turns*position_s
> colnames(pnl_s) <- "Strategy"
> cum_pnls <- cumsum(pnl_s)
> weal_th <- cbind(cum_rets, cum_pnls, vwap_fast, vwap_slow)
> colnames(weal_th) <- c(sym_bol, "Strategy", "VWAP_fast", "VWAP_s
> # Annualized Sharpe ratios of VTI and VWAP strategy
> sharp_e <- sqrt(252)*sapply(cbind(re_turns, pnl_s),
+   function (x) mean(x)/sd(x))
> # Calculate index for background shading
> in_dic <- (vwap_fast > vwap_slow)
> whi_ch <- which(rutils::diff_it(in_dic) != 0)
> in_dic <- rbind(first(in_dic), in_dic[whi_ch, ], last(in_dic))
> date_s <- index(in_dic)
> in_dic <- ifelse(drop(coredata(in_dic)), "lightgreen", "antiquewhite")
```

```
> # Create dygraph object without rendering it
> dy_graph <- dygraphs::dygraph(weal_th, main=paste("VWAP Crossover
+   dyOptions(colors=c("blue", "red", "purple", "lightpurple"), stro
+   dyLegend(show="always", width=500)
> # Add shading
> for (i in 1:(NROW(in_dic)-1)) {
+     dy_graph <- dy_graph %>%
+ dyShading(from=date_s[i], to=date_s[i+1], color=in_dic[i])
+ }  # end for
> # Render the dygraph object
> dy_graph
```

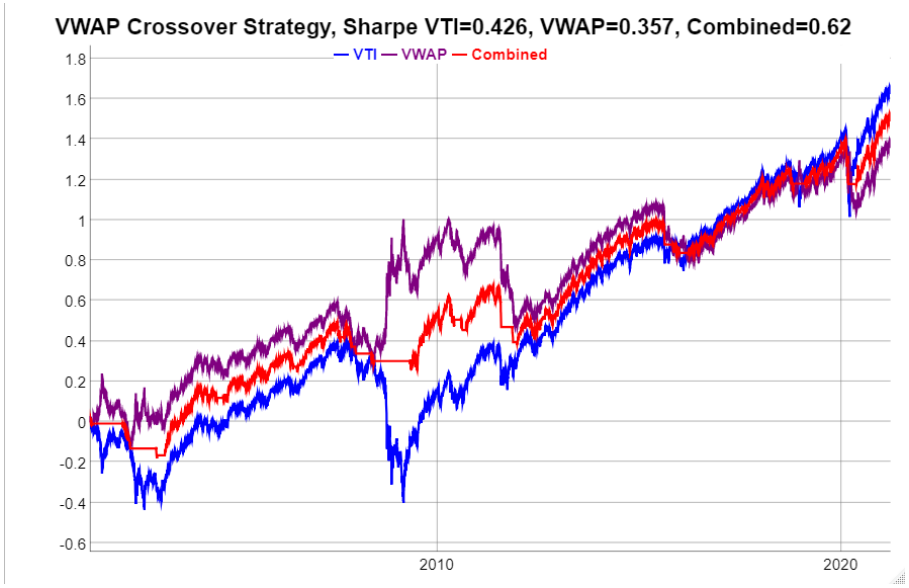
Combining VWAP Crossover Strategy with Stocks

Even though the *VWAP* strategy doesn't perform as well as a static buy-and-hold strategy, it can provide risk reduction when combined with it.

This is because the *VWAP* strategy has a negative correlation with respect to the underlying asset.

In addition, the *VWAP* strategy performs well in periods of extreme market selloffs, so it can provide a hedge for a static buy-and-hold strategy.

The *VWAP* strategy serves as a dynamic put option in periods of extreme market selloffs.




```
> # Calculate correlation of VWAP strategy with VTI
> cor(pnl_s, re_turns)
> # Combine VWAP strategy with VTI
> weal_th <- cbind(re_turns, pnl_s, 0.5*(re_turns+pnl_s))
> colnames(weal_th) <- c(sym_bol, "VWAP", "Combined")
> sharp_e <- sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
```

```
> # Plot dygraph of VWAP strategy combined with VTI
> dygraphs::dygraph(cumsum(weal_th),
+   main=paste("VWAP Crossover Strategy, Sharpe", paste(paste(names
+   dyOptions(colors=c("blue", "purple", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

The Brownian Motion Process

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

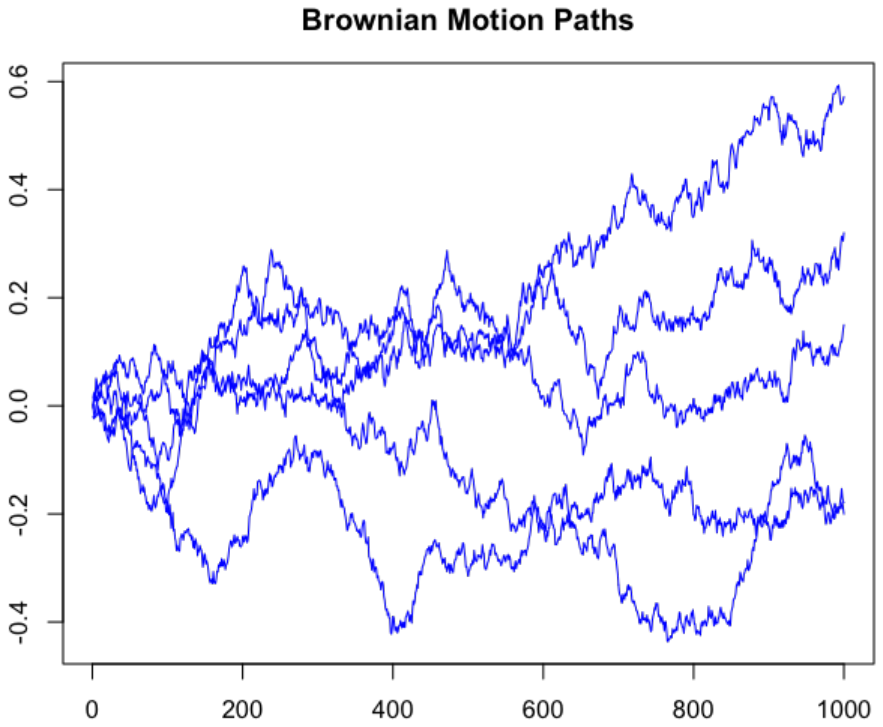
$$r_i = p_i - p_{i-1} = \sigma \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where σ is the volatility of returns, and ξ_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
> n_rows <- 1000; sig_ma <- 0.01
> # Simulate 5 paths of Brownian motion
> price_s <- matrix(rnorm(5*n_rows, sd=sig_ma), nc=5)
> price_s <- matrixStats::colCumsums(price_s)
> # 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=price_s, 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 μ minus the latest price p_{i-1}, times the mean reversion parameter θ, 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 σ is the volatility of returns, and ξ_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
> eq_price <- 1.0; sig_ma <- 0.02
> the_ta <- 0.01; n_rows <- 1000
> # Initialize the data
> in_nov <- rnorm(n_rows)
> re_turns <- numeric(n_rows)
> price_s <- numeric(n_rows)
> # Simulate Ornstein-Uhlenbeck process in R
> price_s[1] <- sig_ma*in_nov[1]
> for (i in 2:n_rows) {
+   re_turns[i] <- the_ta*(eq_price - price_s[i-1]) +
+     sig_ma*in_nov[i]
+   price_s[i] <- price_s[i-1] + re_turns[i]
+ }  # end for
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> prices_cpp <- HighFreq::sim_ou(eq_price=eq_price, volat=sig_ma,
+   theta=the_ta, innov=matrix(in_nov))
> all.equal(price_s, drop(prices_cpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (i in 2:n_rows) {
+     re_turns[i] <- the_ta*(eq_price - price_s[i-1]) + sig_ma*in_no
+     price_s[i] <- price_s[i-1] + re_turns[i]}},
+   Rcpp=HighFreq::sim_ou(eq_price=eq_price, volat=sig_ma,
+     theta=the_ta, innov=matrix(in_nov)),
+   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 $\phi(0, \sqrt{\mathrm{d}t})$.

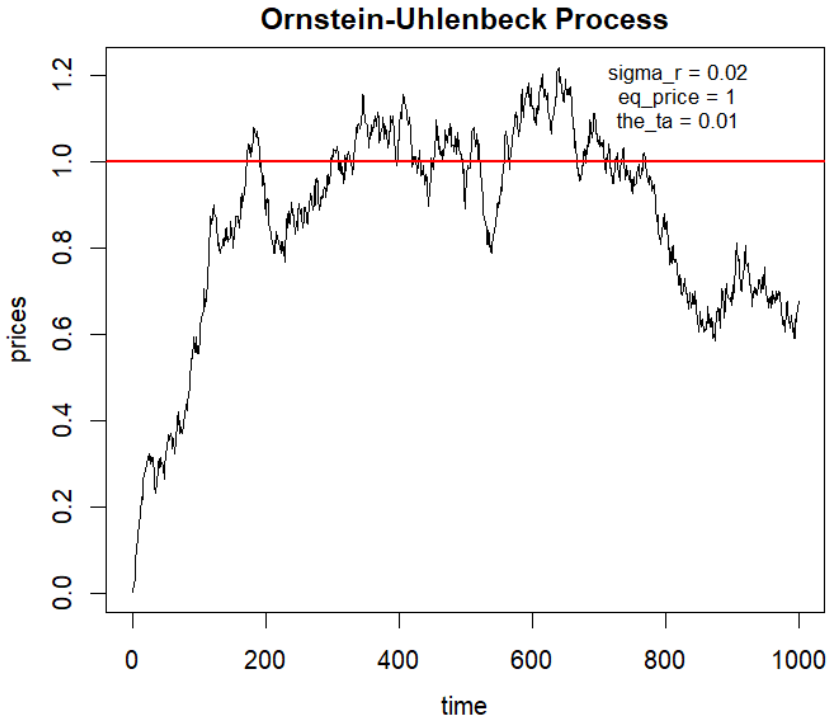
The solution of the *Ornstein-Uhlenbeck* process is given by:

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

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 μ.

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

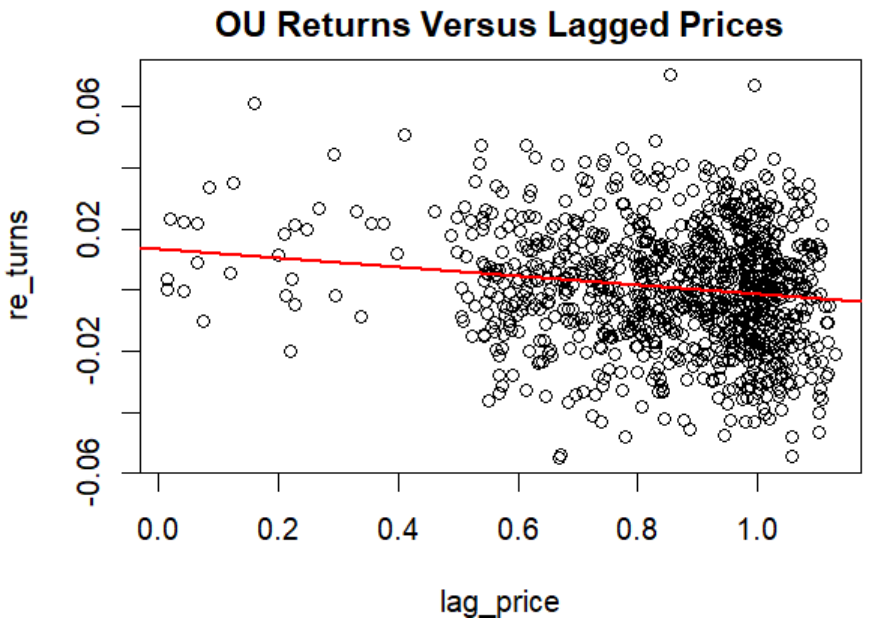



```
> plot(price_s, type="l",
+      xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright",
+  title=paste(c(paste0("sig_ma = ", sig_ma),
+                paste0("eq_price = ", eq_price),
+                paste0("the_ta = ", the_ta)),
+              collapse="\n"),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

Ornstein-Uhlenbeck Process Returns Correlation

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

```
> re_turns <- rutils::diff_it(price_s)
> lag_prices <- rutils::lag_it(price_s)
> for_mula <- re_turns ~ lag_prices
> l_m <- lm(for_mula)
> summary(l_m)
> # Plot regression
> plot(for_mula, main="OU Returns Versus Lagged Prices")
> abline(l_m, lwd=2, col="red")
```



Calibrating the Ornstein-Uhlenbeck Parameters

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

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

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

```
> # Calculate volatility parameter
> c(volatility=sig_ma, estimate=sd(re_turns))
> # Extract OU parameters from regression
> co_eff <- summary(l_m)$coefficients
> # Calculate regression alpha and beta directly
> be_ta <- cov(re_turns, lag_prices)/var(lag_prices)
> al_pha <- (mean(re_turns) - be_ta*mean(lag_prices))
> cbind(direct=c(alpha=al_pha, beta=be_ta), lm=co_eff[, 1])
> all.equal(c(alpha=al_pha, beta=be_ta), co_eff[, 1],
+     check.attributes=FALSE)
> # Calculate regression standard errors directly
> beta_s <- c(alpha=al_pha, beta=be_ta)
> fit_ted <- (al_pha + be_ta*lag_prices)
> residual_s <- (re_turns - fit_ted)
> prices_squared <- sum((lag_prices - mean(lag_prices))^2)
> beta_sd <- sqrt(sum(residual_s^2)/prices_squared/(n_rows-2))
> alpha_sd <- sqrt(sum(residual_s^2)/(n_rows-2)*(1/n_rows + mean(lag
> cbind(direct=c(alpha_sd=alpha_sd, beta_sd=beta_sd), lm=co_eff[, 2
> all.equal(c(alpha_sd=alpha_sd, beta_sd=beta_sd), co_eff[, 2],
+     check.attributes=FALSE)
> # Compare mean reversion parameter theta
> c(theta=(-the_ta), round(co_eff[2, ], 3))
> # Compare equilibrium price mu
> c(eq_price=eq_price, estimate=-co_eff[1, 1]/co_eff[2, 1])
> # Compare actual and estimated parameters
> co_eff <- cbind(c(the_ta*eq_price, -the_ta), co_eff[, 1:2])
> rownames(co_eff) <- c("drift", "theta")
> colnames(co_eff)[1] <- "actual"
> round(co_eff, 4)
```

The Schwartz Process

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

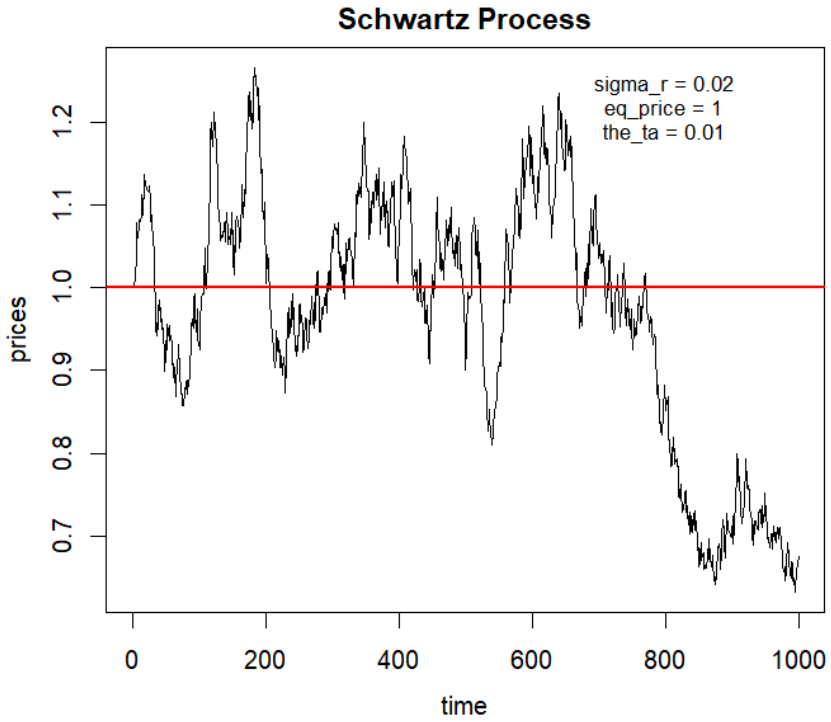
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of prices, which can be negative.

The *Schwartz* process is the exponential of the *Ornstein-Uhlenbeck* process, so it avoids negative prices by compounding the percentage returns r_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 θ is the strength of mean reversion, σ is the volatility, and ξ_i are random normal *innovations* with zero mean and unit variance.




```
> # Simulate Schwartz process
> re_turns <- numeric(n_rows)
> price_s <- numeric(n_rows)
> price_s[1] <- exp(sig_ma*in_nov[1])
> set.seed(1121)  # Reset random numbers
> for (i in 2:n_rows) {
+   re_turns[i] <- the_ta*(eq_price - price_s[i-1]) + sig_ma*in_nov
+   price_s[i] <- price_s[i-1]*exp(re_turns[i])
+ }  # end for
```

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

Autocorrelation Function of Time Series

The estimator of *autocorrelation* of a time series is equal to:

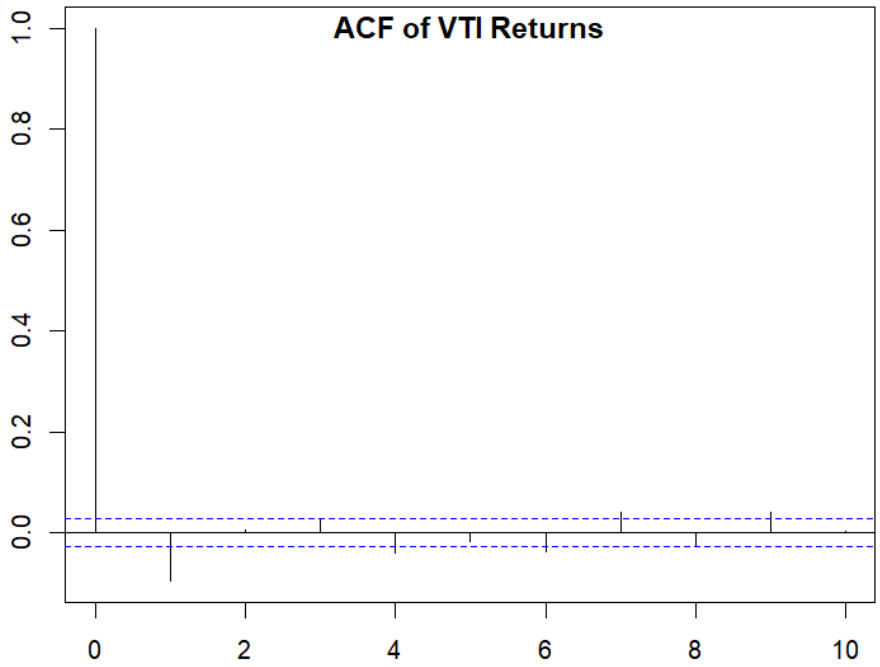
$$\rho_k = \frac{\sum_{i=k+1}^{n}(x_i - \bar{x})(x_{i-k} - \bar{x})}{(n-k)\,\sigma^2}$$

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

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

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

```
> x11(width=6, height=5)
> par(mar=c(3, 2, 1, 1), oma=c(1, 0, 0, 0))
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> # Plot autocorrelations using stats::acf()
> stats::acf(re_turns, lag=10, xlab="lag", main="")
> title(main="ACF of VTI Returns", line=-1)
> # Two-tailed 95% confidence interval
> qnorm(0.975)/sqrt(NROW(re_turns))
```



The *VTI* time series of returns does not appear to have statistically significant autocorrelations.

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

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

Ljung-Box Test for Autocorrelations of Time Series

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

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

The test statistic is:

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

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

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

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

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

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

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

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

Improved Autocorrelation Function

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

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

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

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

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

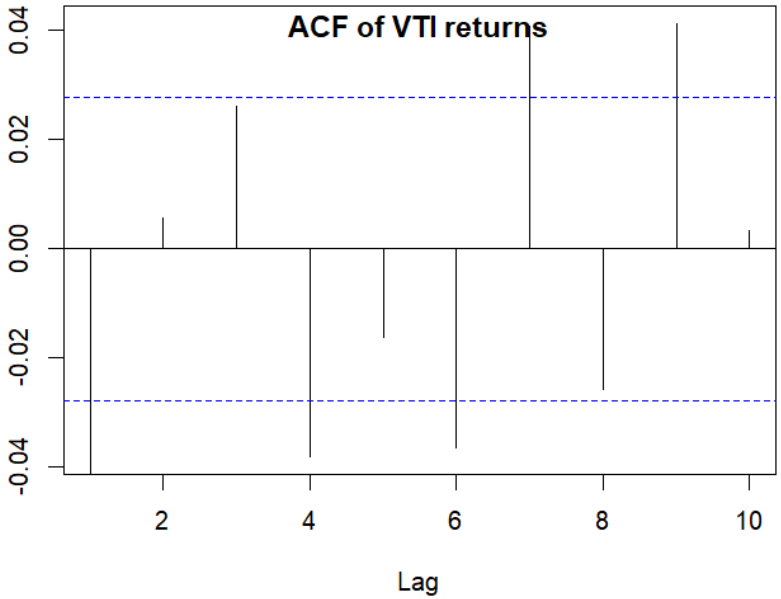
```
> plot_acf <- function(x_ts, lagg=10, plo_t=TRUE,
+                xlab="Lag", ylab="", main="", ...) {
+   # Calculate the ACF without a plot
+   acf_data <- acf(x=x_ts, lag.max=lagg, plot=FALSE, ...)
+   # Remove first element of ACF data
+   acf_data$acf <- array(data=acf_data$acf[-1],
+     dim=c((dim(acf_data$acf)[1]-1),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

Autocorrelation of *VTI* Returns

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

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

```
> # Improved autocorrelation function
> x11(width=6, height=5)
> rutils::plot_acf(re_turns, lag=10, main="")
> title(main="ACF of VTI returns", line=-1)
> # Ljung-Box test for VTI returns
> Box.test(re_turns, lag=10, type="Ljung")
```



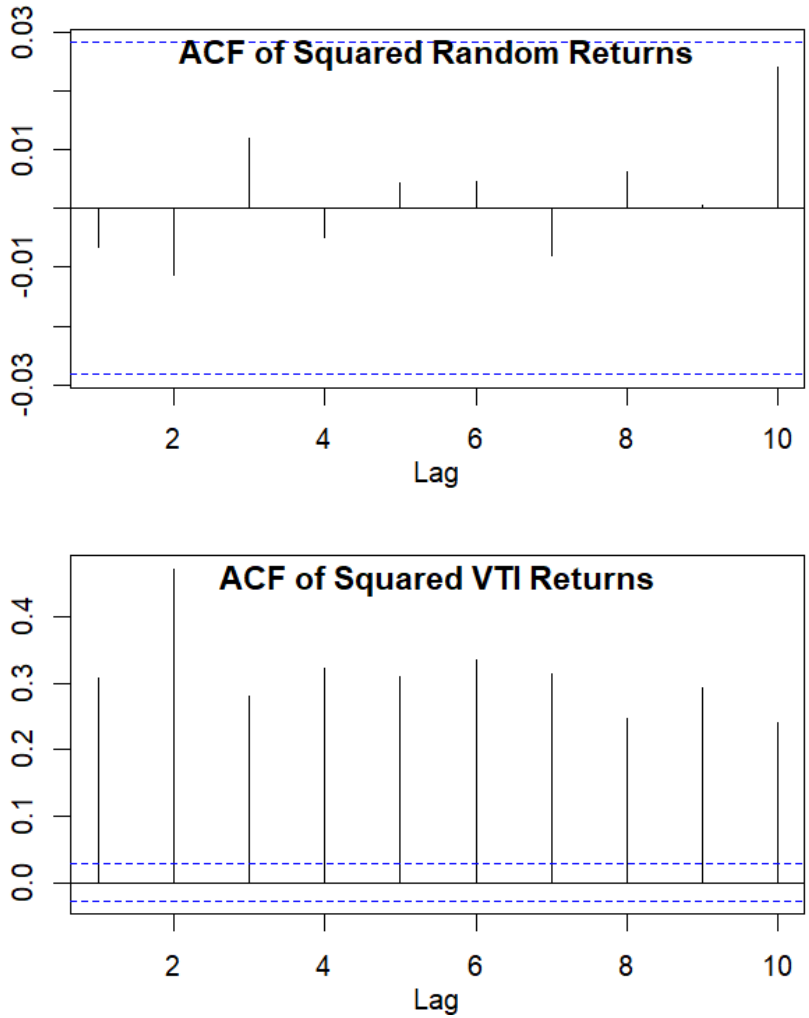
Autocorrelation of Squared *VTI* Returns

Squared random returns are not autocorrelated.

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

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

```
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> # Autocorrelation of squared random returns
> rutils::plot_acf(rnorm(NROW(re_turns))^2, lag=10, main="")
> title(main="ACF of Squared Random Returns", line=-1)
> # Autocorrelation of squared VTI returns
> rutils::plot_acf(re_turns^2, lag=10, main="")
> title(main="ACF of Squared VTI Returns", line=-1)
> # Ljung-Box test for squared VTI returns
> Box.test(re_turns^2, lag=10, type="Ljung")
```



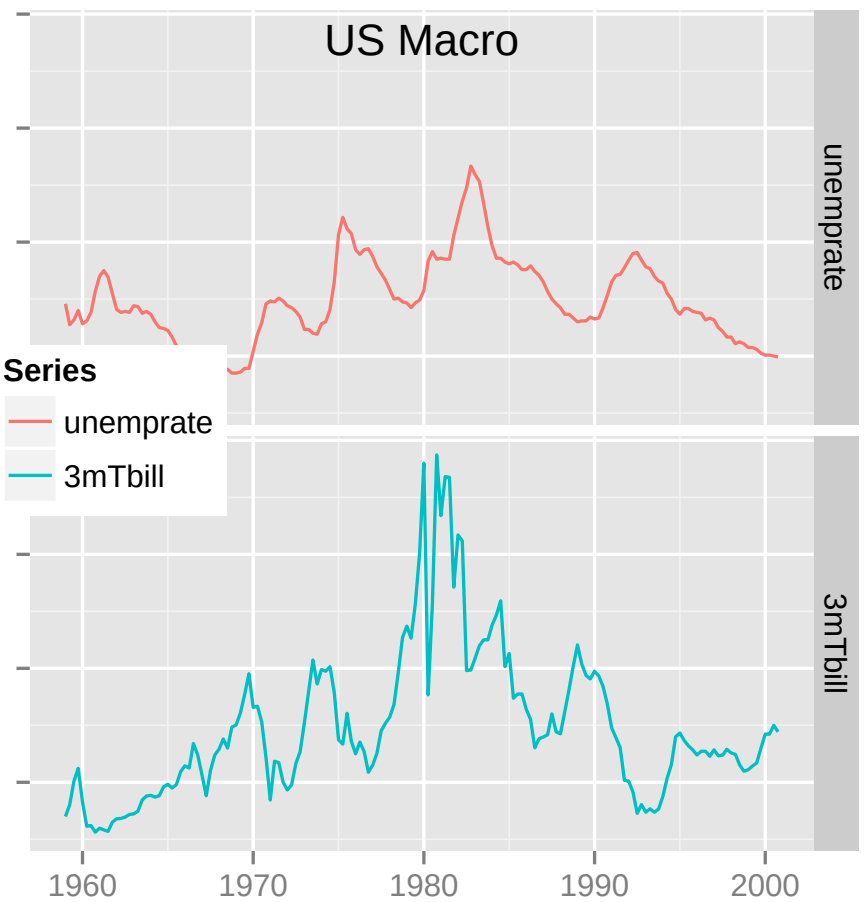
U.S. Macroeconomic Data

The package *Ecdat* contains the `Macrodat` U.S. macroeconomic data.

`"lhur"` is the unemployment rate (average of months in quarter).

`"fygm3"` 3 month treasury bill interest rate (last month in quarter)

```
> library(Ecdat)  # Load Ecdat
> colnames(Macrodat)  # United States Macroeconomic Time Series
> # Coerce to "zoo"
> macro_zoo <- as.zoo(Macrodat[, c("lhur", "fygm3")])
> colnames(macro_zoo) <- c("unemprate", "3mTbill")
> # ggplot2 in multiple panes
> autoplot(  # Generic ggplot2 for "zoo"
+   object=macro_zoo, main="US Macro",
+   facets=Series ~ .) + # end autoplot
+   xlab("") +
+ theme(  # Modify plot theme
+   legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+   plot.background=element_blank(),
+   axis.text.y=element_blank()
+ )  # end theme
```



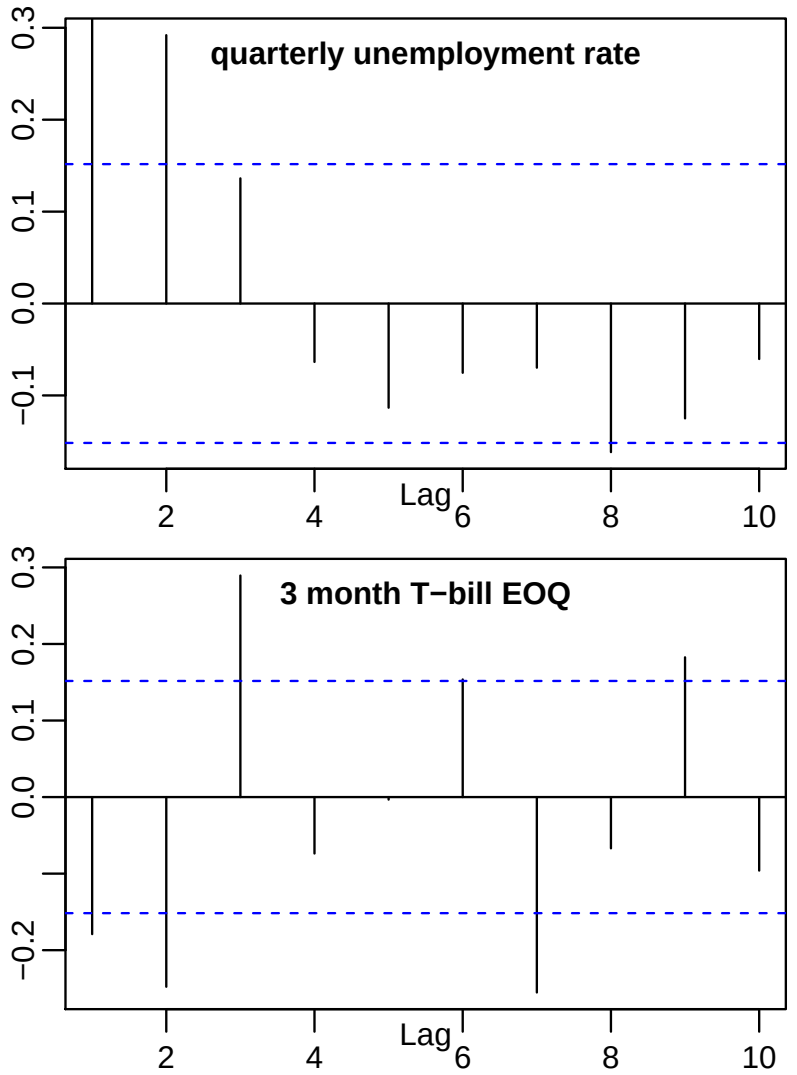
Autocorrelation of Econometric Data

Most econometric data displays a high degree of autocorrelation.

But the time series of asset returns display very low autocorrelations.

```
> macro_diff <- na.omit(diff(macro_zoo))
> rutils::plot_acf(coredata(macro_diff[, "unemprate"]),
+   lag=10, main="quarterly unemployment rate")
> rutils::plot_acf(coredata(macro_diff[, "3mTbill"]),
+   lag=10, main="3 month T-bill EOQ")
```

The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.



Autoregressive Processes

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

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

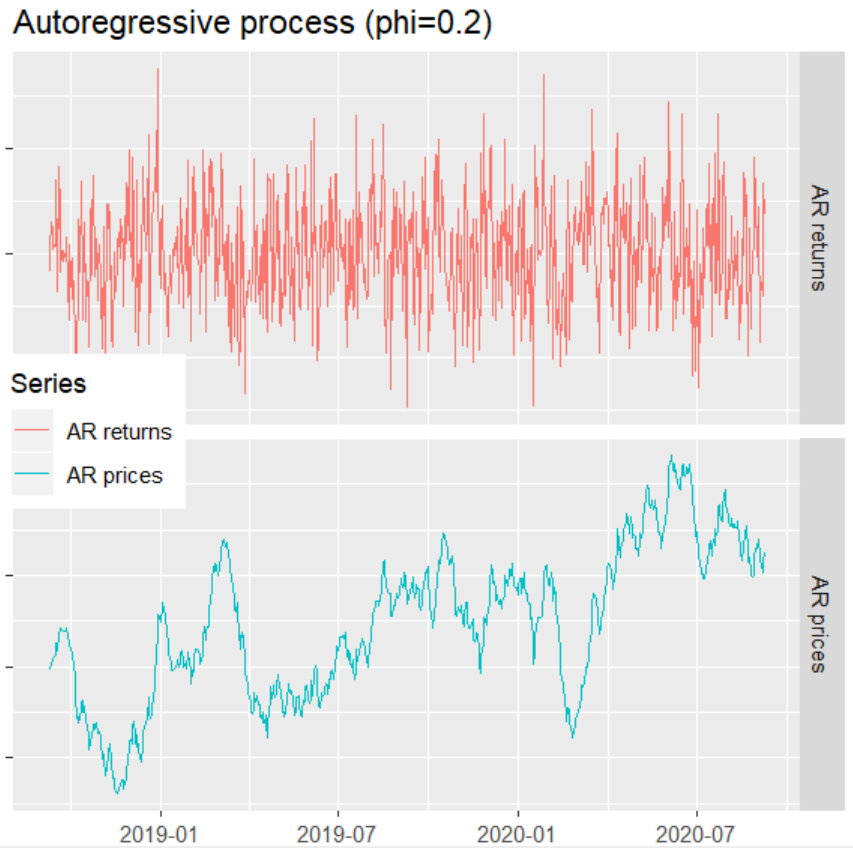
Where φ_i are the *AR(p)* coefficients, and ξ_i are standard normal *innovations*.

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

If the *AR(p)* 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(p)* coefficients φ_i.

```
> date_s <- Sys.Date() + 0:728  # Two year daily series
> # AR time series of returns
> ari_ma <- xts(x=arima.sim(n=NROW(date_s), model=list(ar=0.2)),
+           order.by=date_s)
> ari_ma <- cbind(ari_ma, cumsum(ari_ma))
> colnames(ari_ma) <- c("AR returns", "AR prices")
```



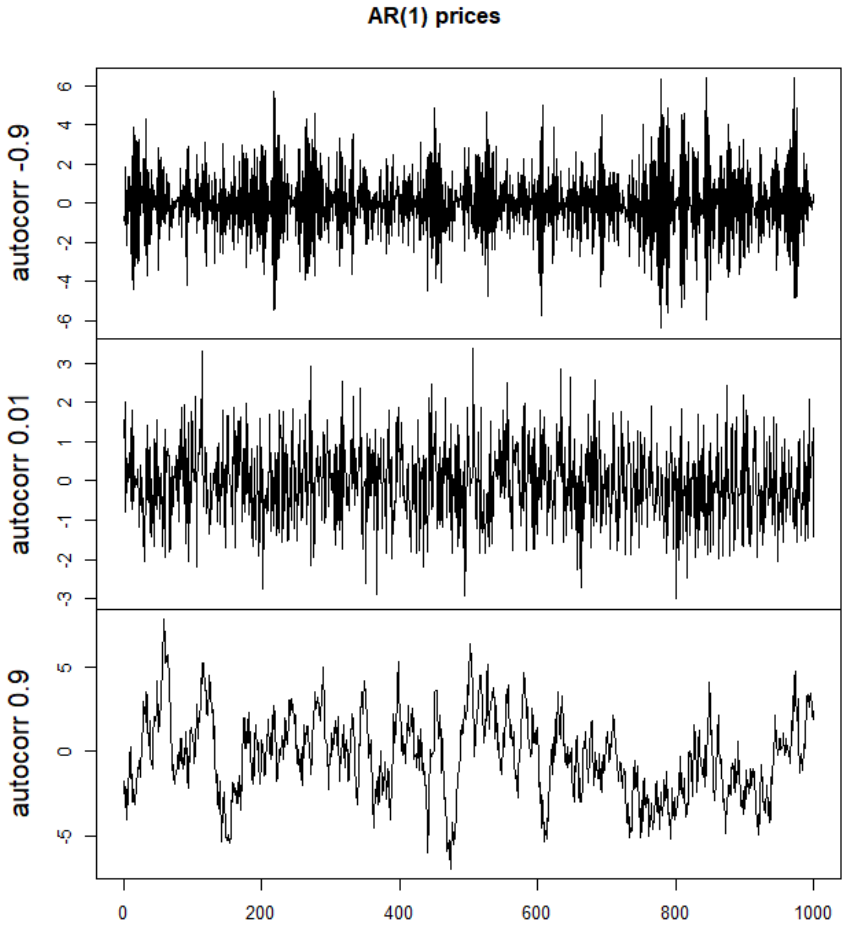

```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=ari_ma, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),
+   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(p)* coefficient φ_1, with a negative coefficient producing faster mean reversion, and a positive coefficient producing stronger diversion.

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

```
> ar_coeff <- c(-0.9, 0.01, 0.9)  # AR coefficients
> # Create three AR time series
> ari_ma <- sapply(ar_coeff, function(phi) {
+   set.seed(1121)  # Reset random numbers
+   arima.sim(n=NROW(date_s), model=list(ar=phi))
+ })  # end sapply
> colnames(ari_ma) <- paste("autocorr", ar_coeff)
> plot.zoo(ari_ma, main="AR(1) prices", xlab=NA)
> # Or plot using ggplot
> ari_ma <- xts(x=ari_ma, order.by=date_s)
> library(ggplot)
> autoplot(ari_ma, 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(p)*:

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

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

AR(p) 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"`.

```
> # Define AR(3) coefficients and innovations
> co_eff <- c(0.1, 0.39, 0.5)
> n_rows <- 1e2
> set.seed(1121); in_nov <- rnorm(n_rows)
> # Simulate AR process using recursive loop in R
> ari_ma <- numeric(NROW(in_nov))
> ari_ma[1] <- in_nov[1]
> ari_ma[2] <- co_eff[1]*ari_ma[1] + in_nov[2]
> ari_ma[3] <- co_eff[1]*ari_ma[2] + co_eff[2]*ari_ma[1] + in_nov[3]
> for (it in 4:NROW(ari_ma)) {
+   ari_ma[it] <- ari_ma[(it-1):(it-3)] %*% co_eff + in_nov[it]
+ }  # End for
> # Simulate AR process using filter()
> arima_faster <- filter(x=in_nov, filter=co_eff, method="recursive"
> class(arima_faster)
> all.equal(ari_ma, as.numeric(arima_faster))
> # Fast simulation of AR process using C_rfilter()
> arima_fastest <- .Call(stats:::C_rfilter, in_nov, co_eff,
+                        double(NROW(co_eff) + NROW(in_nov)))[-(1:3)]
> all.equal(ari_ma, arima_fastest)
```

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(p)* coefficients: $\frac{6}{\log(minroot)} + \text{numcoeff}$

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

Autocorrelations of Autoregressive Processes

The autocorrelation ρ_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: $\rho_i = \varphi^i$.

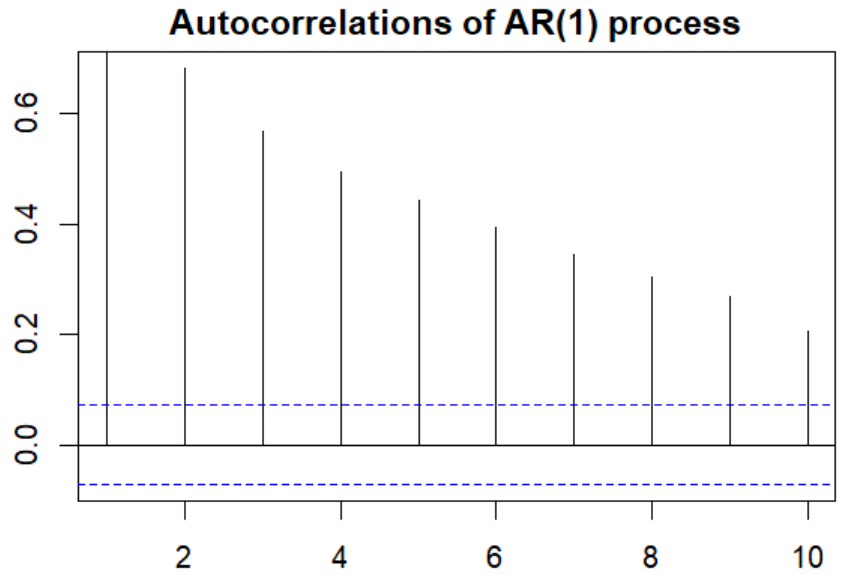
The *AR(1)* process can be solved recursively:

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

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

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

If $\varphi = 1.0$ then the influence of the random innovations ξ_i persists indefinitely, so that the variance of r_i increases linearly with time.



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

```
> # Simulate AR(1) process
> ari_ma <- arima.sim(n=1e3, model=list(ar=0.8))
> # ACF of AR(1) process
> ac_f <- rutils::plot_acf(ari_ma, lag=10, xlab="", ylab="",
+   main="Autocorrelations of AR(1) process")
> ac_f$acf[1:5]
```

Partial 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.

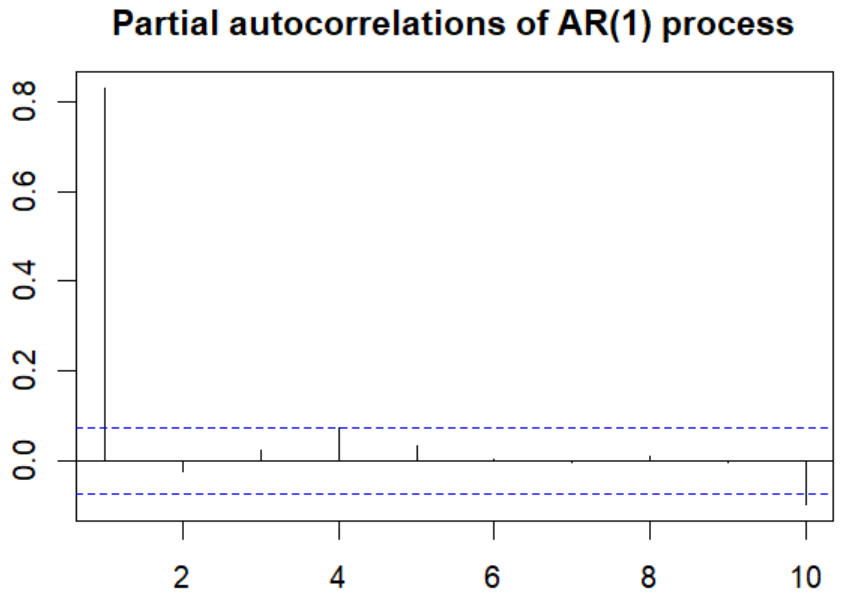
The *partial autocorrelations* ϱ_i of an *AR(1)* process can be computed recursively from the autocorrelations ρ_i using the Durbin-Levinson algorithm:

$$\varrho_1 = \rho_1$$

$$\varrho_2 = \rho_2 - \varrho_1 \rho_1$$

$$\varrho_3 = \rho_3 - \varrho_1 \rho_2 - \varrho_2 \rho_1$$

The function `pacf()` calculates and plots the *partial autocorrelations*, but it performs regressions instead of 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
> pac_f <- pacf(ari_ma, lag=10, xlab="", ylab="", main="")
> title("Partial autocorrelations of AR(1) process", line=1)
> pac_f <- drop(pac_f$acf)
> pac_f[1:5]
```

Partial Autocorrelations of *AR(1)* Processes

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

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

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

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

The *partial autocorrelations* ϱ_i of an *AR(1)* process can be computed recursively from the autocorrelations ρ_i using the Durbin-Levinson algorithm:

$$\varrho_k = \rho_k - \sum_{i=1}^{k-1} \varrho_i \rho_{k-i}$$

```
> # Compute pacf recursively from acf
> ac_f <- rutils::plot_acf(ari_ma, lag=10, plo_t=FALSE)
> ac_f <- drop(ac_f$acf)
> pac_f <- numeric(3)
> pac_f[1] <- ac_f[1]
> pac_f[2] <- ac_f[2] - ac_f[1]^2
> pac_f[3] <- ac_f[3] - pac_f[2]*ac_f[1] - ac_f[2]*pac_f[1]
> # Compute pacf recursively in a loop
> pac_f <- numeric(NROW(ac_f))
> pac_f[1] <- ac_f[1]
> for (it in 2:NROW(pac_f)) {
+   pac_f[it] <- ac_f[it] - pac_f[1:(it-1)] %*% ac_f[(it-1):1]
+ }  # end for
```

Higher Order Autocorrelations

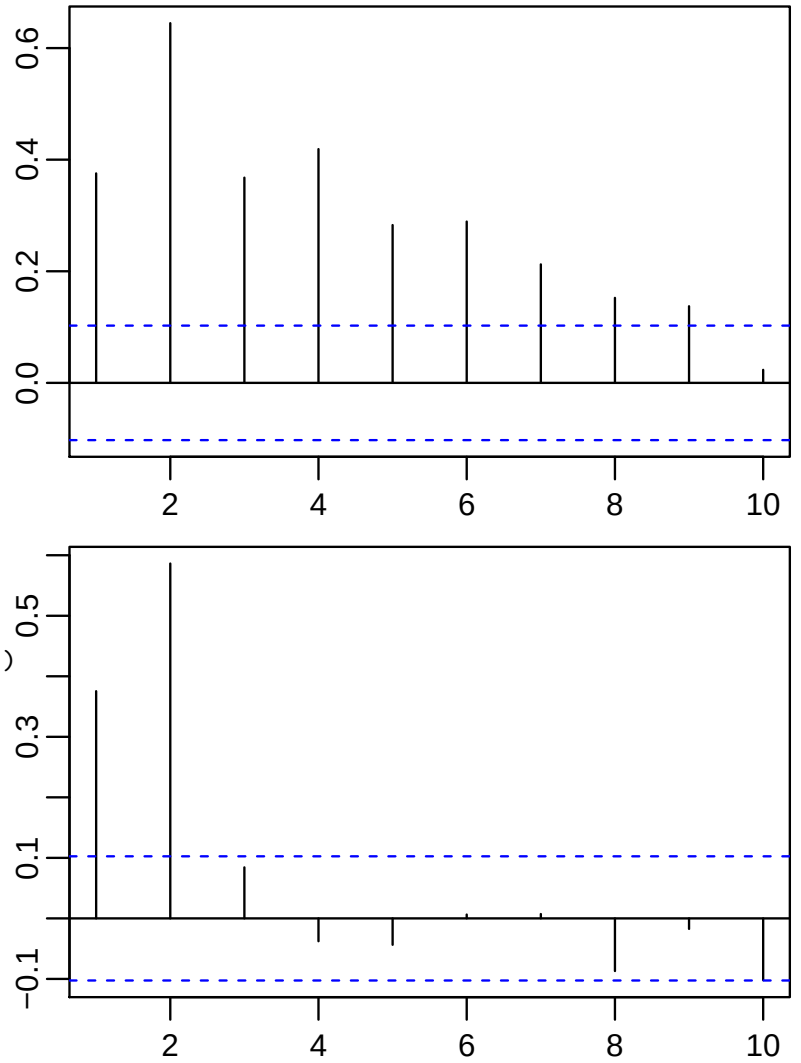
An *AR(3)* process of order *three* is defined by the formula:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \varphi_3 r_{i-3} + \xi_i$$

Autoregressive processes *AR(p)* of order *p* have an exponentially decaying *ACF* and a non-zero *PACF* up to lag *p*.

The number of non-zero *partial autocorrelations* is equal to the *order* parameter *p* of the *AR(p)* process.

```
> # Simulate AR process of returns
> ari_ma <- arima.sim(n=1e3, model=list(ar=c(0.1, 0.5, 0.1)))
> # ACF of AR(3) process
> rutils::plot_acf(ari_ma, lag=10, xlab="", ylab="",
+    main="ACF of AR(3) process")
> # PACF of AR(3) process
> pacf(ari_ma, lag=10, xlab="", ylab="", main="PACF of AR(3) process")
```



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(p)*:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_p r_{i-p} + \xi_i$

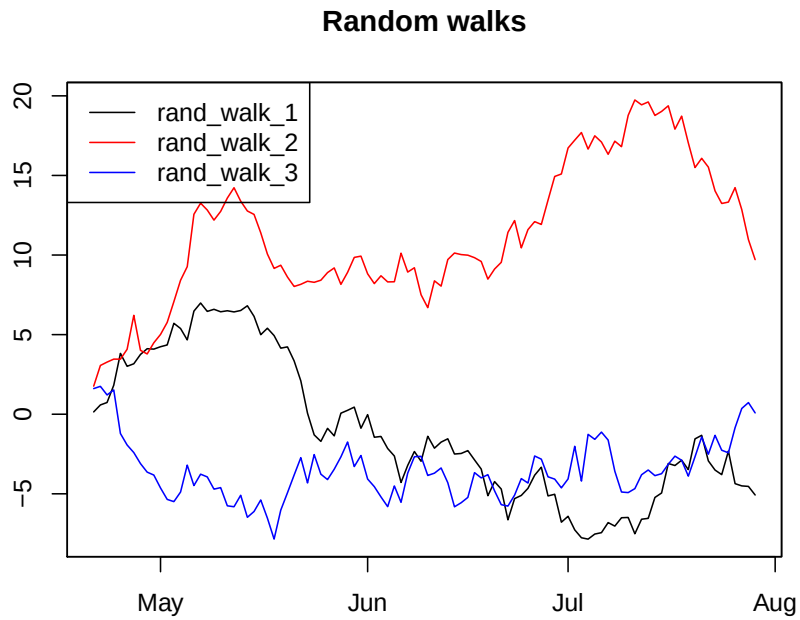
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_p z^p = 0$

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: $\sum_{i=1}^{p} \varphi_i = 1$, 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*: $p_i = p_{i-1} + \xi_i$




```
> rand_walk <- cumsum(zoo(matrix(rnorm(3*100), ncol=3),
+             order.by=(Sys.Date()+0:99)))
> colnames(rand_walk) <- paste("rand_walk", 1:3, sep="_")
> plot.zoo(rand_walk, main="Random walks",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue"))
> # Add legend
> legend(x="topleft", legend=colnames(rand_walk),
+  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(p)* process:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_p r_{i-p} + \xi_i$

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

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

The *integrated* process of an *AR(p)* 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 φ, because the sum of its coefficients is equal to 1.

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

```
> # Simulate arima with large AR coefficient
> set.seed(1121)
> ari_ma <- arima.sim(n=n_rows, model=list(ar=0.99))
> tseries::adf.test(ari_ma)
> # Integrated series has unit root
> tseries::adf.test(cumsum(ari_ma))
> # Simulate arima with negative AR coefficient
> set.seed(1121)
> ari_ma <- arima.sim(n=n_rows, model=list(ar=-0.99))
> tseries::adf.test(ari_ma)
> # Integrated series has unit root
> tseries::adf.test(cumsum(ari_ma))
```

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: $r_i = r_{i-1} + \xi_i$

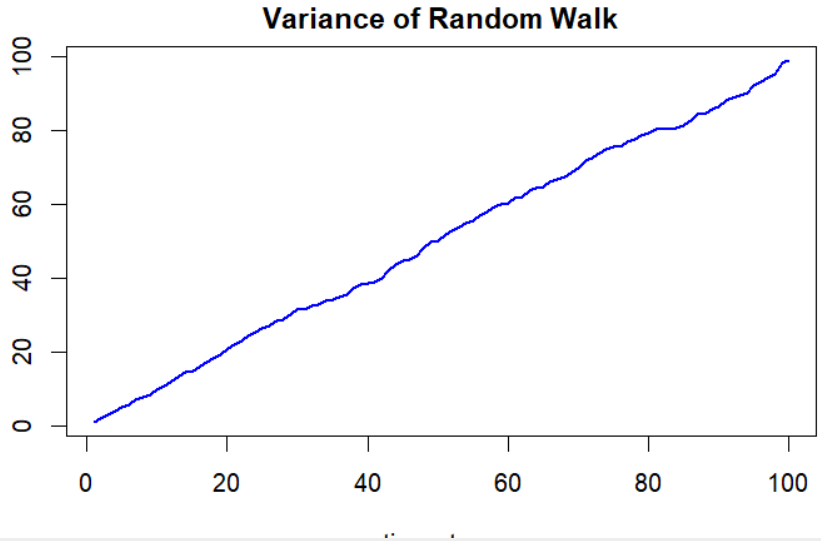
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

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

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:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_p r_{i-p} + \sigma \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where μ is the equilibrium price, σ is the volatility of returns, and ξ_i are standard normal *innovations*.

Then the prices follow an *autoregressive* process:

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

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

```
> # Define Dickey-Fuller parameters
> eq_price <- 1.0; sig_ma <- 0.02
> the_ta <- 0.01; n_rows <- 1000
> # Initialize the data
> in_nov <- rnorm(n_rows)
> re_turns <- numeric(n_rows)
> price_s <- numeric(n_rows)
> # Simulate Dickey-Fuller process in R
> price_s[1] <- sig_ma*in_nov[1]
> for (i in 2:n_rows) {
+   re_turns[i] <- the_ta*(eq_price - price_s[i-1]) + sig_ma*in_nov
+   price_s[i] <- price_s[i-1] + re_turns[i]
+ }  # end for
> # Simulate Dickey-Fuller process in Rcpp
> prices_cpp <- HighFreq::sim_ou(eq_price=eq_price, volat=sig_ma,
+     theta=the_ta, innov=matrix(in_nov))
> all.equal(price_s, drop(prices_cpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (i in 2:n_rows) {
+     re_turns[i] <- the_ta*(eq_price - price_s[i-1]) + sig_ma*in_no
+     price_s[i] <- price_s[i-1] + re_turns[i]}},
+   Rcpp=HighFreq::sim_ou(eq_price=eq_price, volat=sig_ma, theta=the
+   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 with an extra mean reversion term:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_p r_{i-p} + \varepsilon_i$$

Where μ is the equilibrium price.

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

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_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 θ parameter: $t_\theta = \hat{\theta}/SE_\theta$ (which follows a distribution different from the t-distribution).

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: $p = 0$ then the *ADF* test becomes the standard *Dickey-Fuller* test: $r_i = \theta p_{i-1} + \xi_i$.

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

```
> set.seed(1121); in_nov <- rnorm(1e4, sd=0.01)
> # Simulate AR(1) process with coefficient=1, with unit root
> ari_ma <- filter(x=in_nov, filter=1.0, method="recursive")
> x11(); plot(ari_ma, t="l", main="AR(1) coefficient = 1.0")
> # Perform ADF test with lag = 1
> tseries::adf.test(ari_ma, k=1)
> # Perform standard Dickey-Fuller test
> tseries::adf.test(ari_ma, k=0)
> # Simulate AR(1) with coefficient close to 1, without unit root
> ari_ma <- filter(x=in_nov, filter=0.99, method="recursive")
> x11(); plot(ari_ma, t="l", main="AR(1) coefficient = 0.99")
> tseries::adf.test(ari_ma, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with mean reversion
> eq_price <- 0.0; the_ta <- 0.001
> price_s <- HighFreq::sim_ou(eq_price=eq_price, volat=1.0,
+   theta=the_ta, innov=in_nov)
> x11(); plot(price_s, t="l", main=paste("OU coefficient =", the_ta)
> tseries::adf.test(price_s, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with zero reversion
> the_ta <- 0.0
> price_s <- HighFreq::sim_ou(eq_price=eq_price, volat=1.0,
+   theta=the_ta, innov=in_nov)
> x11(); plot(price_s, t="l", main=paste("OU coefficient =", the_ta)
> tseries::adf.test(price_s, k=1)
```

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