FRE7241 Algorithmic Portfolio Management

Lecture#5, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 5, 2021

NYU | TANDON SCHOOL OF ENGINEERING

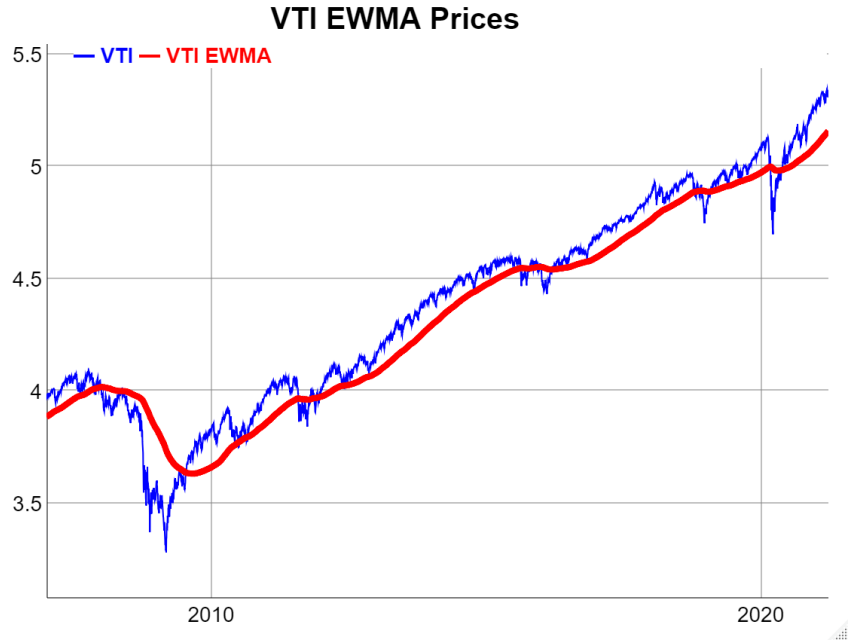
EWMA Price Technical Indicator

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

$$p_i^{EWMA} = (1 - \exp(-\lambda)) \sum_{j=0}^{\infty} \exp(-\lambda j) p_{i-j}$$

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

```
> # Extract log VTI prices
> oh_lc <- rutils::etf_env$VTI
> clos_e <- log(quantmod::Cl(oh_lc))
> colnames(clos_e) <- "VTI"
> n_rows <- NROW(clos_e)
> # Calculate EWMA weights
> look_back <- 333
> lamb_da <- 0.004
> weight_s <- exp(-lamb_da*(1:look_back))
> weight_s <- weight_s/sum(weight_s)
> # Calculate EWMA prices
> ew_ma <- HighFreq::roll_wsum(clos_e, weights=weight_s)
> # Copy over NA values
> ew_ma <- zoo::na.locf(ew_ma, fromLast=TRUE)
> price_s <- cbind(clos_e, ew_ma)
> colnames(price_s) <- c("VTI", "VTI EWMA")
```




```
> # Dygraphs plot with custom line colors
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s["2007/"], main="VTI EWMA Prices") %>%
+   dySeries(name=col_names[1], label=col_names[1], strokeWidth=1, c
+   dySeries(name=col_names[2], label=col_names[2], strokeWidth=4, c
+   dyLegend(show="always", width=500)
> # Standard plot of  EWMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> col_ors <- c("blue", "red")
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(price_s["2007/"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(price_s),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

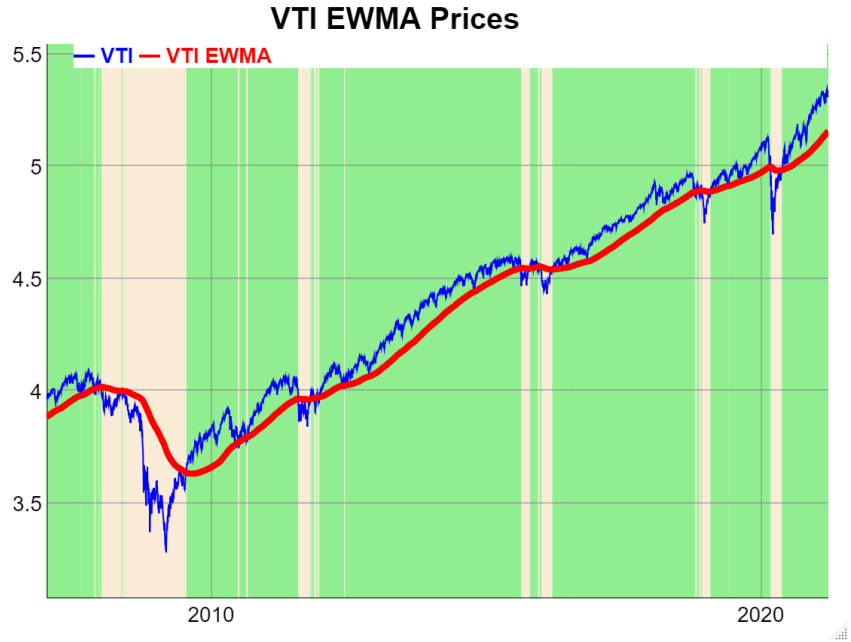
Simulating the EWMA Crossover Strategy

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.




```
> # Calculate positions, either: -1, 0, or 1
> position_s <- sign(clos_e - ew_ma)
> position_s <- xts::xts(position_s, order.by=index(clos_e))
> position_s <- rutils::lag_it(position_s, lagg=1)
> # Create colors for background shading
> date_s <- (rutils::diff_it(position_s) != 0)
> shad_e <- position_s[date_s]
> date_s <- c(index(shad_e), end(position_s))
> shad_e <- ifelse(drop(zoo::coredata(shad_e)) == 1, "lightgreen",
> # Create dygraph object without plotting it
> dy_graph <- dygraphs::dygraph(price_s["2007/"], main="VTI EWMA Pr
+   dySeries(name=col_names[1], label=col_names[1], strokeWidth=1,
+   dySeries(name=col_names[2], label=col_names[2], strokeWidth=4,
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>% dyShading(from=date_s[i], to=date_s[i+1], color=shad_e[i])
+ }  # end for
> # Plot the dygraph object
> dy_graph
```

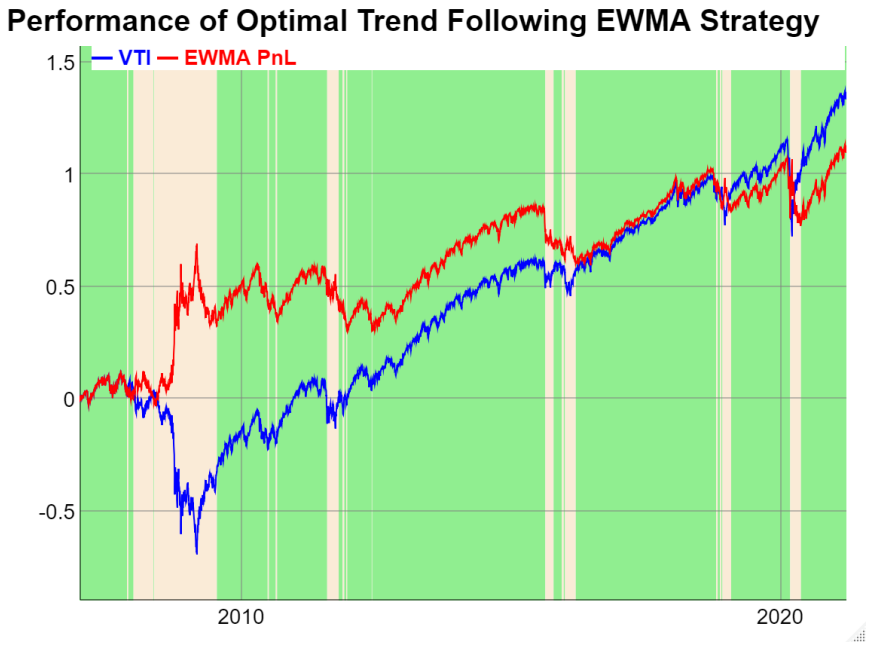
```
> # Standard plot of EWMA prices with position shading
> x11(width=6, height=5)
> quantmod::chart_Series(price_s["2007/"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> add_TA(position_s > 0, on=-1, col="lightgreen", border="lightgreen
> add_TA(position_s < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("topleft", legend=colnames(price_s),
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Performance of the EWMA Crossover Strategy

The crossover strategy trades at the *Close* price on the same day that prices cross the *EWMA*, which may be difficult in practice.

The crossover strategy performance is worse than the underlying asset (*VTI*), but it has a negative correlation to it, which is very valuable when building a portfolio.

```
> # Calculate daily profits and losses of EWMA strategy
> vt_i <- rutils::diff_it(clos_e)
> colnames(vt_i) <- "VTI"
> pnl_s <- vt_i*position_s
> colnames(pnl_s) <- "EWMA"
> weal_th <- cbind(vt_i, pnl_s)
> colnames(weal_th) <- c("VTI", "EWMA PnL")
> # Annualized Sharpe ratio of EWMA strategy
> sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(weal_th)
> # Plot dygraph of EWMA strategy wealth
> # Create dygraph object without plotting it
> col_ors <- c("blue", "red")
> dy_graph <- dygraphs::dygraph(cumsum(weal_th["2007/"]), main="Pe
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>%
+ dyShading(from=date_s[i], to=date_s[i+1], color=shad_e[i])
+ }  # end for
> # Plot the dygraph object
> dy_graph
```



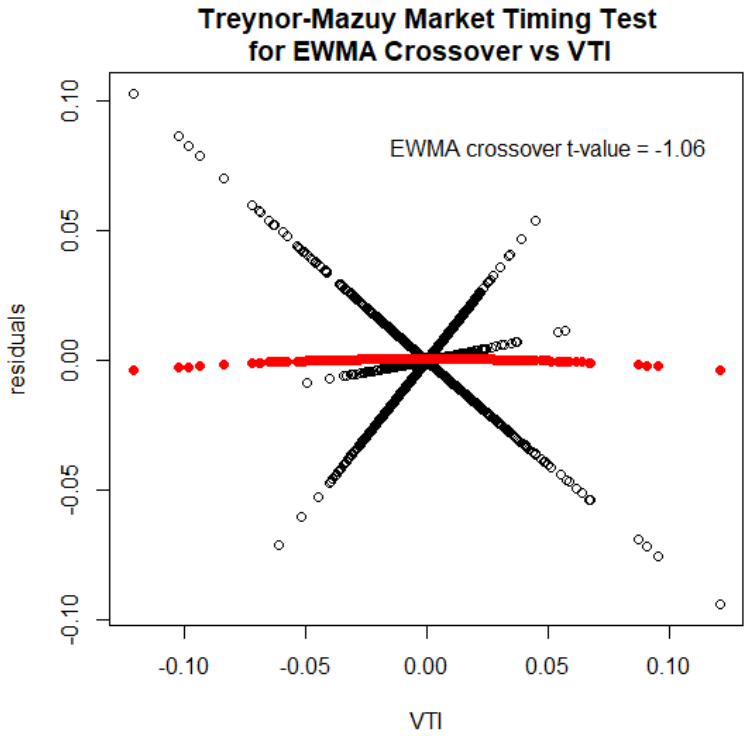

```
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(weal_th["2007/"]), theme=plot_theme
+        name="Performance of EWMA Strategy")
> add_TA(position_s > 0, on=-1, col="lightgreen", border="lightgreen
> add_TA(position_s < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(weal_th),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

EWMA Crossover Strategy Market Timing Skill

The EWMA crossover strategy shorts the market during significant selloffs, but otherwise doesn't display market timing skill.

The t-value of the *Treynor-Mazuy* test is negative, but not statistically significant.

```
> # Test EWMA crossover market timing of VTI using Treynor-Mazuy tes
> de_sign <- cbind(pnl_s, vt_i, vt_i^2)
> de_sign <- na.omit(de_sign)
> colnames(de_sign) <- c("EWMA","VTI","treynor")
> mod_el <- lm(EWMA ~ VTI + treynor, data=de_sign)
> summary(mod_el)
> # Plot residual scatterplot
> residual_s <- (de_sign$EWMA - mod_el$coeff[2]*de_sign$VTI)
> residual_s <- mod_el$residuals
> x11(width=6, height=6)
> plot.default(x=de_sign$VTI, y=residual_s, xlab="VTI", ylab="residu
> title(main="Treynor-Mazuy Market Timing Test\n for EWMA Crossover
> # Plot fitted (predicted) response values
> fit_ted <- (mod_el$coeff["(Intercept)"] +
+         mod_el$coeff["treynor"]*vt_i^2)
> points.default(x=de_sign$VTI, y=fit_ted, pch=16, col="red")
> text(x=0.05, y=0.8*max(residual_s), paste("EWMA crossover t-value =", round(summary(mod_el)$coeff["treynor", "t value"], 2)))
```



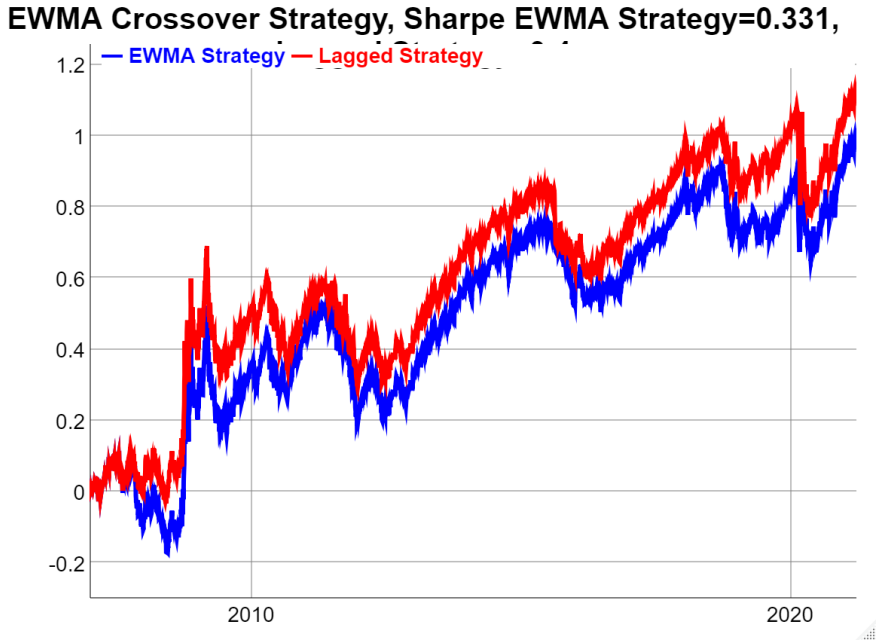
EWMA Crossover Strategy With Lag

The 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 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 an emerging trend.

```
> # Determine trade dates right after EWMA has crossed prices
> in_dic <- sign(clos_e - ew_ma)
> # Calculate positions from lagged indicator
> lagg <- 2
> in_dic <- roll::roll_sum(in_dic, width=lagg, min_obs=1)
> # Calculate positions, either: -1, 0, or 1
> position_s <- rep(NA_integer_, n_rows)
> position_s[1] <- 0
> position_s <- ifelse(in_dic == lagg, 1, position_s)
> position_s <- ifelse(in_dic == (-lagg), -1, position_s)
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> position_s <- xts::xts(position_s, order.by=index(clos_e))
> # Lag the positions to trade in next period
> position_s <- rutils::lag_it(position_s, lagg=1)
> # Calculate PnLs of lagged strategy
> pnl_s <- vt_i*position_s
> colnames(pnl_s) <- "Lagged Strategy"
```




```
> weal_th <- cbind(weal_th[, 2], pnl_s)
> colnames(weal_th) <- c("EWMA Strategy", "Lagged Strategy")
> # Annualized Sharpe ratios of EWMA strategies
> sharp_e <- sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
> # Plot both strategies
> dygraphs::dygraph(cumsum(weal_th["2007/"]), main=paste("EWMA Cross
+   dyOptions(colors=c("blue", "red"), strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
```

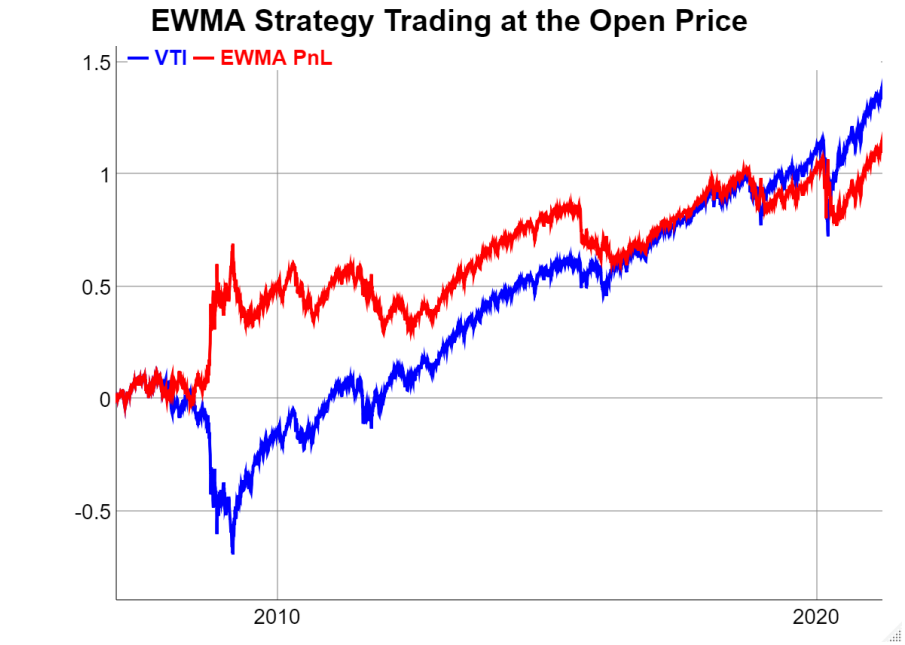
EWMA Strategy Trading at the Open Price

In practice it may not be possible to trade immediately at the *Close* price on the same day that prices cross the *EWMA*.

Then the strategy may trade at the *Open* price on the next day.

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
> pnl_s <- vt_i*position_s
> # Calculate realized pnl for days with trade
> op_en <- log(quantmod::Op(oh_lc))
> 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])
> weal_th <- cbind(vt_i, pnl_s)
> colnames(weal_th) <- c("VTI", "EWMA PnL")
> # Annualized Sharpe ratio of EWMA strategy
> sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(weal_th)
```




```
> # Plot dygraph of EWMA strategy wealth
> dygraphs::dygraph(cumsum(weal_th["2007/"]), main="EWMA Strategy Tr
+   dyOptions(colors=col_ors, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Standard plot of EWMA strategy wealth
> quantmod::chart_Series(weal_th, theme=plot_theme,
+        name="EWMA Strategy Trading at the Open Price")
> legend("top", legend=colnames(weal_th),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

EWMA Crossover Strategy With Transaction Costs

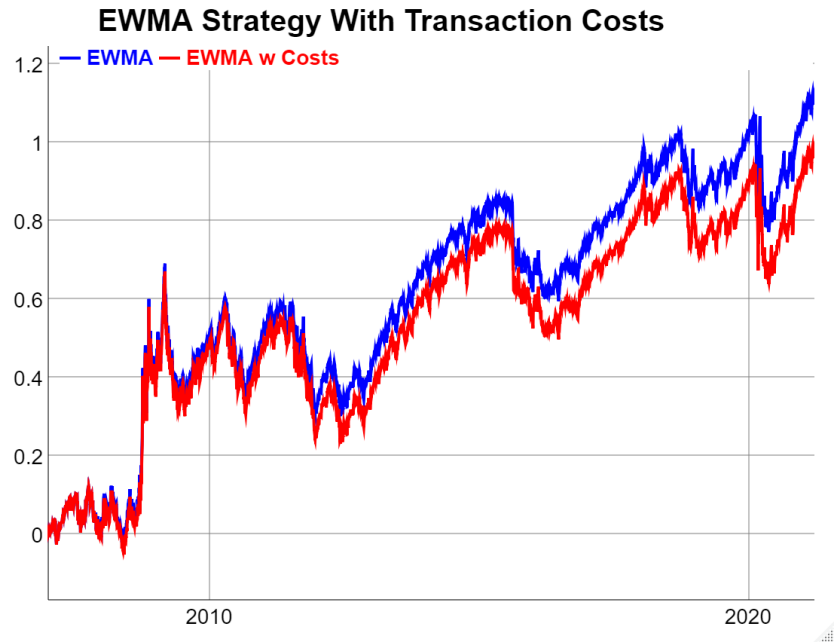
The *bid-offer spread* is the percentage difference between the *offer* minus the *bid* price, divided by the *mid* price.

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
> # Plot strategy with transaction costs
> weal_th <- cbind(pnl_s, pnl_s - cost_s)
> colnames(weal_th) <- c("EWMA", "EWMA w Costs")
> col_ors <- c("blue", "red")
> dygraphs::dygraph(cumsum(weal_th["2007/"]), main="EWMA Strategy W
+   dyOptions(colors=col_ors, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Simulation 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(ohlc, lambda=0.01, look_back=333, bid_offer=
+                 trend=1, lagg=1) {
+   close <- log(quantmod::Cl(ohlc))
+   returns <- rutils::diff_it(close)
+   n_rows <- NROW(ohlc)
+   # Calculate EWMA prices
+   weights <- exp(-lambda*(1:look_back))
+   weights <- weights/sum(weights)
+   ewma <- HighFreq::roll_wsum(close, weights=weights)
+   # Calculate the indicator
+   indic <- trend*sign(close - ewma)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   # Calculate positions, either: -1, 0, or 1
+   pos <- rep(NA_integer_, n_rows)
+   pos[1] <- 0
+   pos <- ifelse(indic == lagg, 1, pos)
+   pos <- ifelse(indic == (-lagg), -1, pos)
+   pos <- zoo::na.locf(pos, na.rm=FALSE)
+   pos <- xts::xts(pos, order.by=index(close))
+   # Lag the positions to trade on next day
+   pos <- rutils::lag_it(pos, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- returns*pos
+   costs <- 0.5*bid_offer*abs(rutils::diff_it(pos))*close
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(pos, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end simu_ewma
```

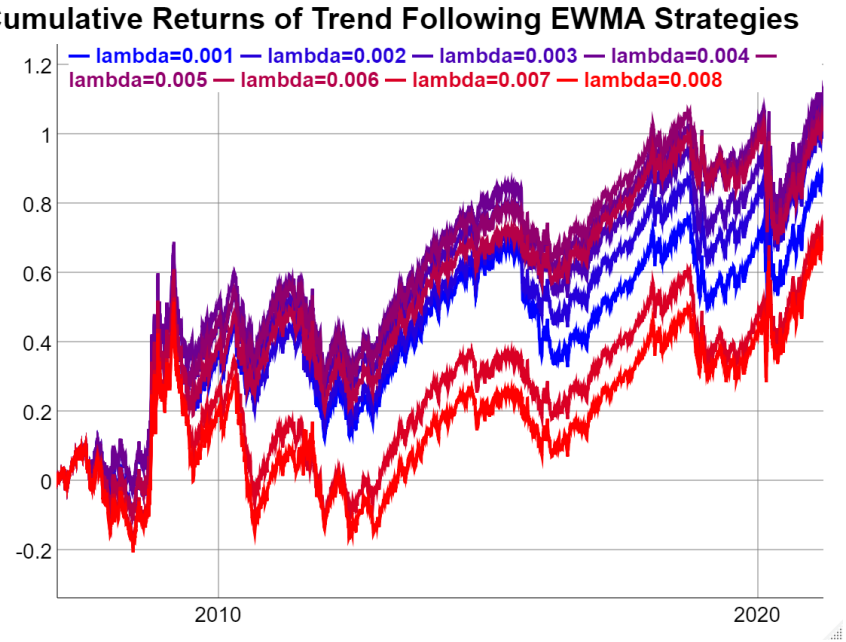
Simulating Multiple Trend Following EWMA Strategies

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=0.001, to=0.008, by=0.001)
> # Perform lapply() loop over lamb_das
> pnl_s <- lapply(lamb_das, function(lamb_da) {
+   # Simulate EWMA strategy and calculate returns
+   simu_ewma(ohlc=oh_lc, lambda=lamb_da, look_back=look_back, bid_
+ })  # end lapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
```




```
> # Plot dygraph of multiple EWMA strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s["2007/"]), main="Cumulative Returns
+   dyOptions(colors=col_ors, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(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", "look_back", "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 returns
+   simu_ewma(ohlc=oh_lc, lambda=lamb_da, look_back=look_back)[, "pr
+ })  # end parLapply
> stopCluster(clus_ter)  # Stop R processes over cluster under Windo
> # Perform parallel loop over lamb_das under Mac-OSX or Linux
> pnl_s <- mclapply(lamb_das, function(lamb_da) {
+   library(quantmod)
+   # Simulate EWMA strategy and calculate returns
+   simu_ewma(ohlc=oh_lc, lambda=lamb_da, look_back=look_back)[, "pr
+ })  # end mclapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
```

Optimal Weights of Trend Following EWMA Strategies

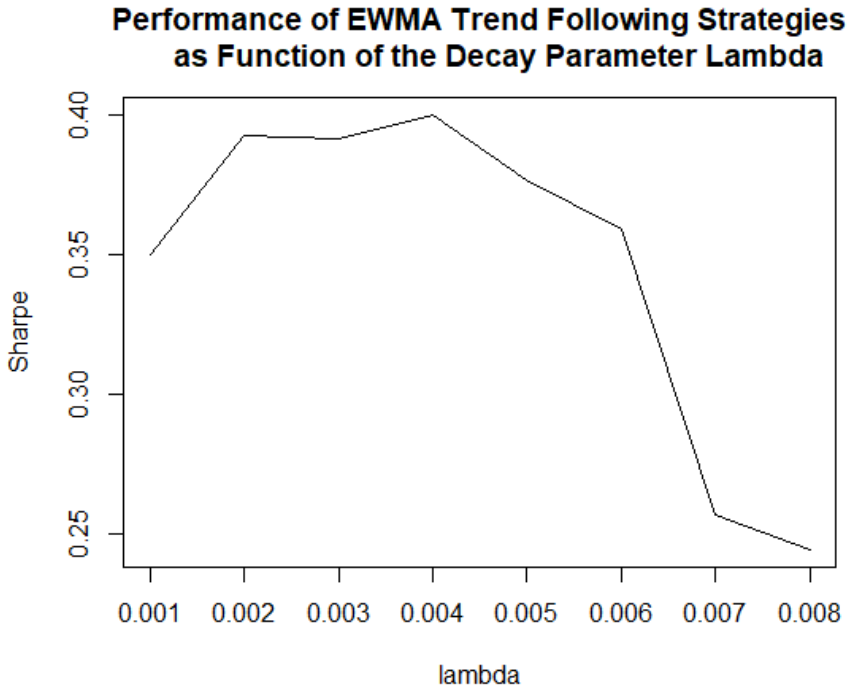
The performance of trend following *EWMA* strategies depends on the λ parameter, with smaller λ parameters performing better than larger ones.

The optimal λ parameter applies significant weight to returns $8 - 12$ months in the past, which is consistent with research on trend following 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.



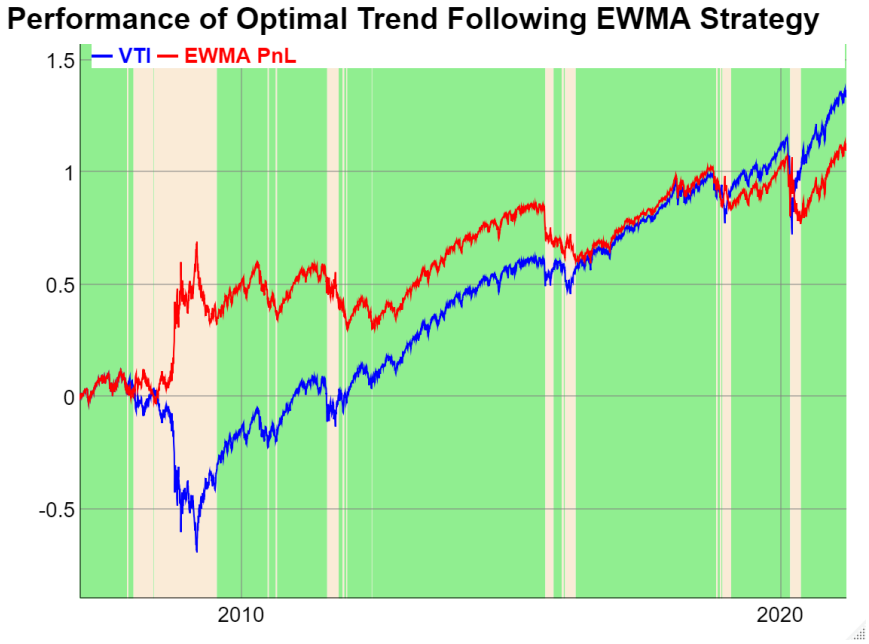

```
> # Calculate annualized Sharpe ratios of strategy returns
> sharpe_ratios <- sqrt(252)*sapply(pnl_s, function(x_ts) {
+   mean(x_ts)/sd(x_ts)
+ })  # end sapply
> # Plot Sharpe ratios
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      xlab="lambda", ylab="Sharpe",
+      main="Performance of EWMA Trend Following Strategies
+      as Function of the Decay Parameter Lambda")
> # Find optimal lambda
> lamb_da <- lamb_das[which.max(sharpe_ratios)]
```

```
> # Plot optimal weights
> weight_s <- exp(-lamb_da*(1:look_back))
> weight_s <- weight_s/sum(weight_s)
> plot(weight_s, t="l", xlab="days", ylab="weights",
+      main="Optimal Weights of EWMA Trend Following Strategy")
> trend_returns <- 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(ohlc=oh_lc, lambda=lamb_da, look_back=look
> position_s <- ewma_trend[, "positions"]
> pnl_s <- ewma_trend[, "pnls"]
> weal_th <- cbind(vt_i, pnl_s)
> colnames(weal_th) <- c("VTI", "EWMA PnL")
> # Create colors for background shading
> date_s <- (rutils::diff_it(position_s) != 0)
> shad_e <- position_s[date_s]
> date_s <- c(index(shad_e), end(position_s))
> shad_e <- ifelse(drop(zoo::coredata(shad_e)) == 1, "lightgreen", '
> col_ors <- c("blue", "red")
> # Plot dygraph of EWMA strategy wealth
> # Create dygraph object without plotting it
> dy_graph <- dygraphs::dygraph(cumsum(weal_th["2007/"]), main="Per
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>%
+ dyShading(from=date_s[i], to=date_s[i+1], color=shad_e[i])
+ }  # end for
> # Plot the dygraph object
> dy_graph
```



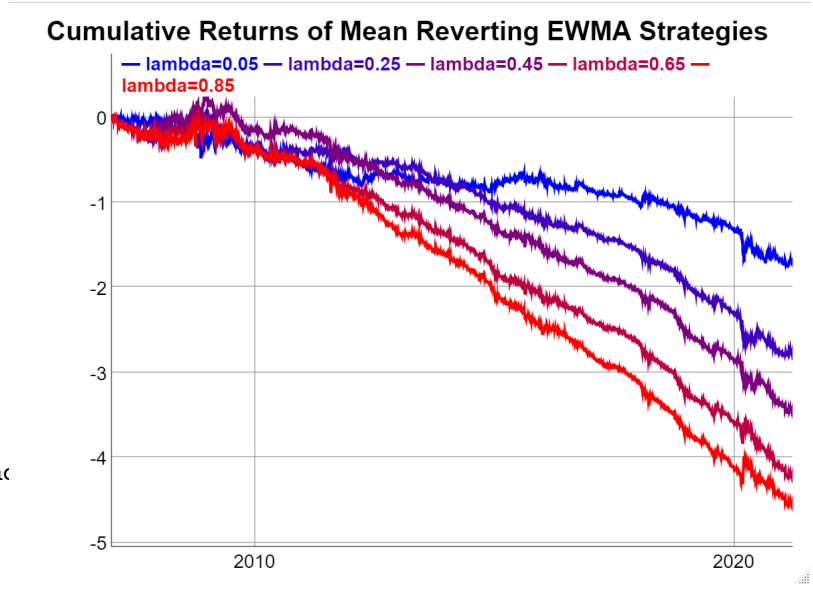

```
> # Plot EWMA PnL with position shading
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(weal_th["2007/"]), theme=plot_theme,
+         name="Performance of EWMA Strategy")
> add_TA(position_s > 0, on=-1, col="lightgreen", border="lightgreen
> add_TA(position_s < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(weal_th),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

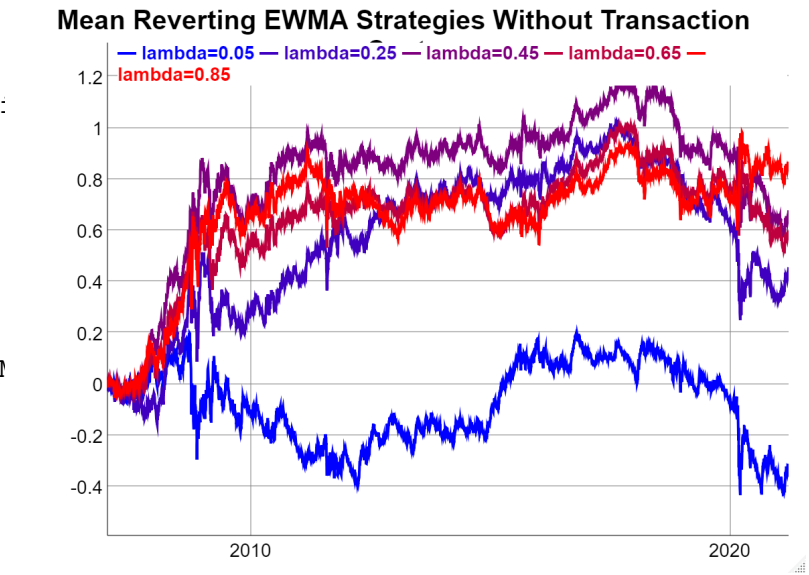
Mean Reverting EWMA Crossover Strategies

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

The profitability of mean reverting strategies can be significantly improved by using limit orders, to reduce transaction costs.

```
> 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) {
+   # Simulate EWMA strategy and calculate returns
+   simu_ewma(ohlc=oh_lc, lambda=lamb_da, look_back=look_back, trend
+ })  # end lapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("lambda=", lamb_das)
> # Plot dygraph of mean reverting EWMA strategies
> column_s <- seq(1, NCOL(pnl_s), by=4)
> col_ors <- colorRampPalette(c("blue", "red"))(NROW(column_s))
> dygraphs::dygraph(cumsum(pnl_s["2007/", column_s]), main="Cumulati
+   dyOptions(colors=col_ors, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> 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.

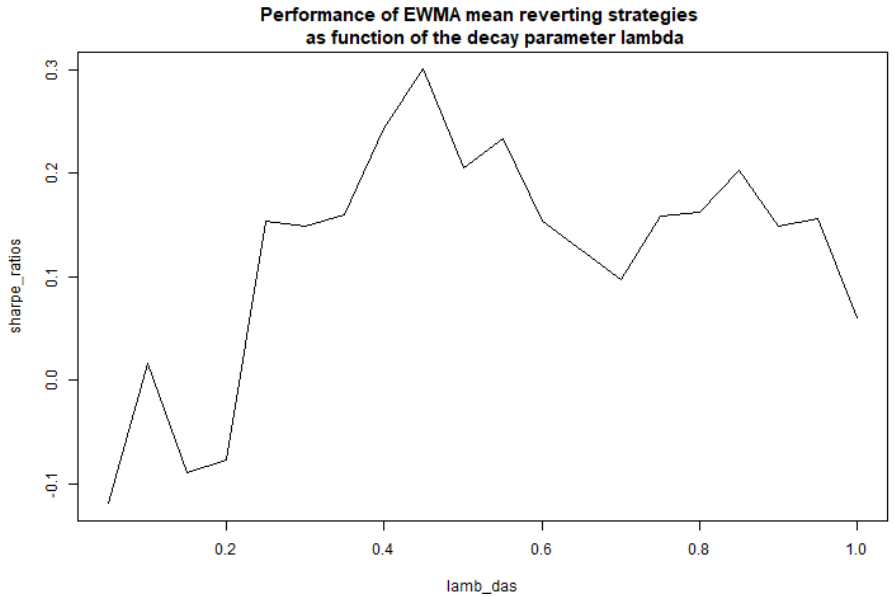
`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns.

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.



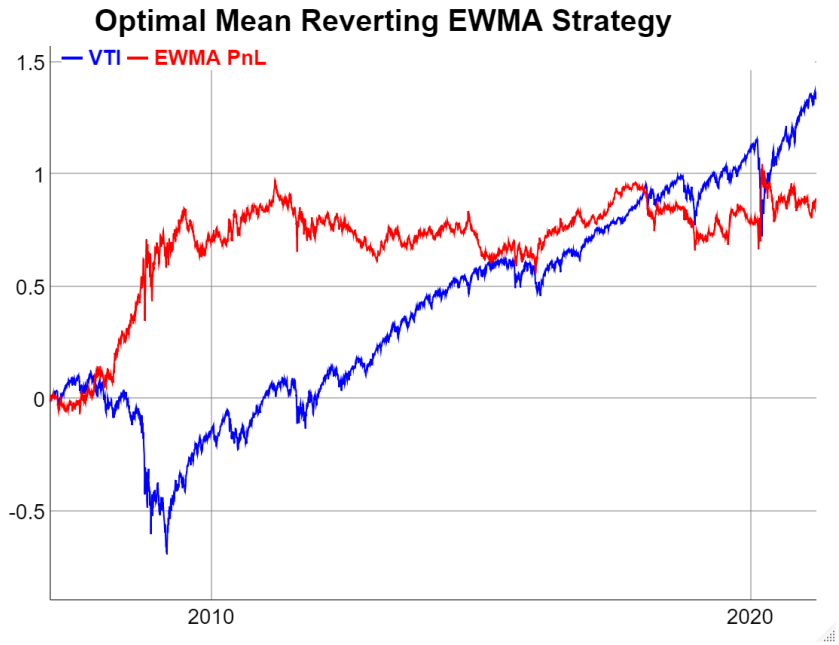

```
> # Calculate Sharpe ratios of strategy returns
> sharpe_ratios <- sqrt(252)*sapply(pnl_s, function(x_ts) {
+   mean(x_ts)/sd(x_ts)
+ })  # end sapply
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      xlab="lambda", ylab="Sharpe",
+      main="Performance of EWMA Mean Reverting Strategies
+      as Function of the Decay Parameter Lambda")
> revert_returns <- 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.




```
> # Find optimal lambda
> lamb_da <- lamb_das[which.max(sharpe_ratios)]
> # Simulate best performing strategy
> ewma_revert <- simu_ewma(ohlc=oh_lc, bid_offer=0.0,
+   lambda=lamb_da, look_back=look_back, trend=(-1))
> position_s <- ewma_revert[, "positions"]
> pnl_s <- ewma_revert[, "pnls"]
> weal_th <- cbind(vt_i, pnl_s)
> colnames(weal_th) <- c("VTI", "EWMA PnL")
> # Plot dygraph of EWMA strategy wealth
> col_ors <- c("blue", "red")
> dygraphs::dygraph(cumsum(weal_th["2007/"]), main="Optimal Mean Re
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA PnL with position shading
```

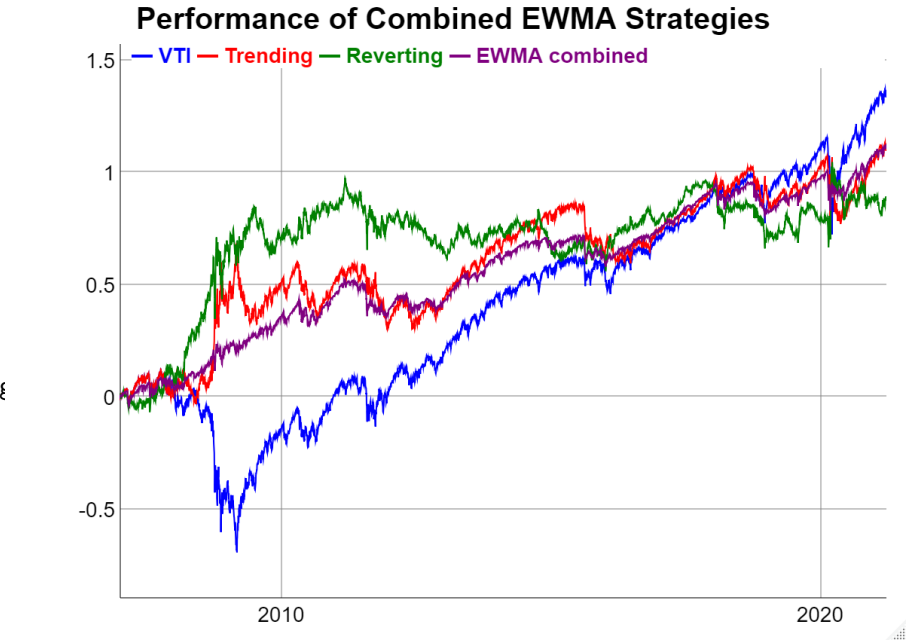
```
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(weal_th["2007/"]), theme=plot_theme,
+        name="Optimal Mean Reverting EWMA Strategy")
> add_TA(position_s > 0, on=-1, col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(weal_th),
+  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.

The main advantage of EWMA crossover strategies is that they provide positive returns and a diversification of risk with respect to static stock portfolios.

```
> # 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"
> cor(cbind(vt_i, trend_ing, revert_ing))
> # Calculate combined strategy
> com_bined <- (vt_i + trend_ing + revert_ing)/3
> colnames(com_bined) <- "combined"
> # Calculate annualized Sharpe ratio of strategy returns
> re_turns <- cbind(vt_i, trend_ing, revert_ing, com_bined)
> colnames(re_turns) <- c("VTI", "Trending", "Reverting", "EWMA co
> sqrt(252)*sapply(re_turns, function(x_ts) mean(x_ts)/sd(x_ts))
```



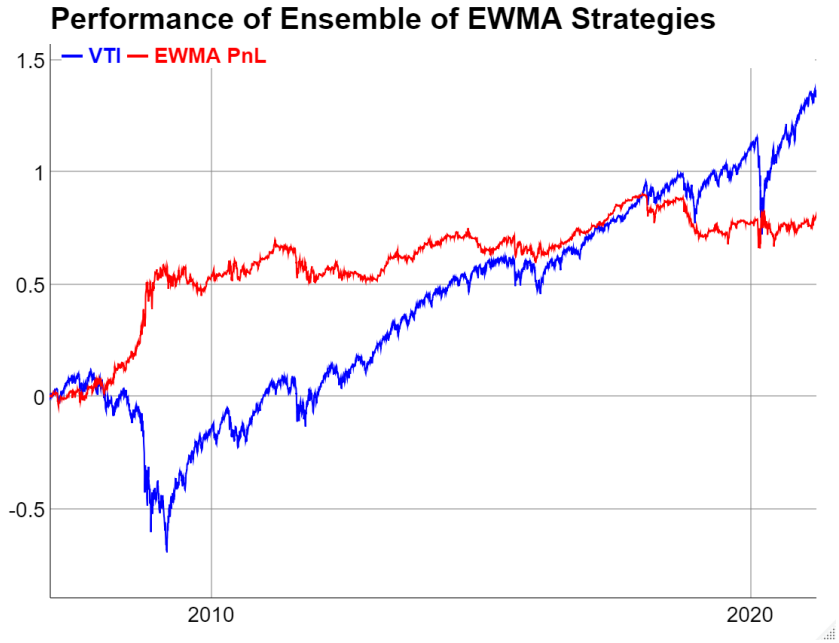

```
> # Plot dygraph of EWMA strategy wealth
> col_ors <- c("blue", "red", "green", "purple")
> dygraphs::dygraph(cumsum(re_turns["2007/"]), main="Performance of
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Standard plot of EWMA strategy wealth
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> 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)
> re_turns <- re_turns %*% weight_s
> re_turns <- xts::xts(re_turns, order.by=index(vt_i))
> re_turns <- cbind(vt_i, re_turns)
> colnames(re_turns) <- c("VTI", "EWMA PnL")
> # Plot dygraph of EWMA strategy wealth
> col_ors <- c("blue", "red")
> dygraphs::dygraph(cumsum(re_turns["2007/"]), main="Performance of Ensemble of EWMA Strategies") %>%
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Standard plot of EWMA strategy wealth
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(re_turns["2007/"]), theme=plot_theme,
+        name="Performance of Ensemble of 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")
```

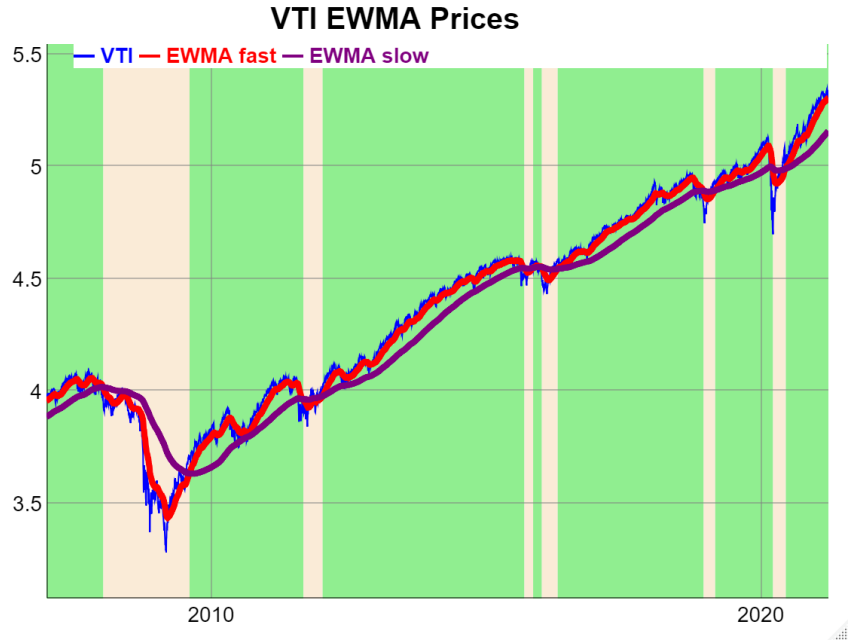


Simulating the Dual EWMA Crossover Strategy

In the *Dual EWMA Crossover* strategy, the risk position depends on the difference between two moving averages.

The risk position flips when the fast moving *EWMA* crosses the slow moving *EWMA*.

```
> # Calculate fast and slow EWMAs
> look_back <- 333
> lambda1 <- 0.04
> lambda2 <- 0.004
> weight_s <- exp(-lambda1*(1:look_back))
> weight_s <- weight_s/sum(weight_s)
> ewma1 <- HighFreq::roll_wsum(clos_e, weights=weight_s)
> weight_s <- exp(-lambda2*(1:look_back))
> weight_s <- weight_s/sum(weight_s)
> ewma2 <- HighFreq::roll_wsum(clos_e, weights=weight_s)
> # Calculate EWMA prices
> price_s <- cbind(clos_e, ewma1, ewma2)
> colnames(price_s) <- c("VTI", "EWMA fast", "EWMA slow")
> # Calculate positions, either: -1, 0, or 1
> in_dic <- sign(ewma1 - ewma2)
> lagg <- 2
> in_dic <- roll::roll_sum(in_dic, width=lagg, min_obs=1)
> position_s <- rep(NA_integer_, n_rows)
> position_s[1] <- 0
> position_s <- ifelse(in_dic == lagg, 1, position_s)
> position_s <- ifelse(in_dic == (-lagg), -1, position_s)
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> position_s <- xts::xts(position_s, order.by=index(clos_e))
> position_s <- rutils::lag_it(position_s, lagg=1)
```




```
> # Create colors for background shading
> date_s <- (rutils::diff_it(position_s) != 0)
> shad_e <- position_s[date_s]
> date_s <- c(index(shad_e), end(position_s))
> shad_e <- ifelse(drop(zoo::coredata(shad_e)) == 1, "lightgreen", "
> # Plot dygraph
> col_names <- colnames(price_s)
> dy_graph <- dygraphs::dygraph(price_s["2007/"], main="VTI Dual EWM
+   dySeries(name=col_names[1], label=col_names[1], strokeWidth=1, c
+   dySeries(name=col_names[2], label=col_names[2], strokeWidth=4, c
+   dySeries(name=col_names[3], label=col_names[3], strokeWidth=4, c
+   dyLegend(show="always", width=500)
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>% dyShading(from=date_s[i], to=date_s[i
+ }  # end for
> dy_graph
```

Performance of the Dual EWMA Crossover Strategy

The 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"*.)

The crossover strategy performance is worse than the underlying asset (*VTI*), but it has a negative correlation to it, which is very valuable when building a portfolio.

```
> # Calculate daily profits and losses of strategy
> pnl_s <- vt_i*position_s
> colnames(pnl_s) <- "Strategy"
> weal_th <- cbind(vt_i, pnl_s)
> # Annualized Sharpe ratio of Dual EWMA strategy
> sharp_e <- sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(weal_th)
```


```
> # Plot Dual EWMA strategy
> dy_graph <- dygraphs::dygraph(cumsum(weal_th["2007/"]), main=paste
+   dyOptions(colors=c("blue", "red"), strokeWidth=1)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>% dyShading(from=date_s[i], to=date_s[
+ }  # end for
> # Plot the dygraph object
> dy_graph
```

Simulation Function for the Dual EWMA Crossover Strategy

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

The function `simu_ewma2()` performs a simulation of the *Dual EWMA* strategy, given an *OHLC* time series of prices, and two decay parameters $\lambda 1$ and $\lambda 2$.

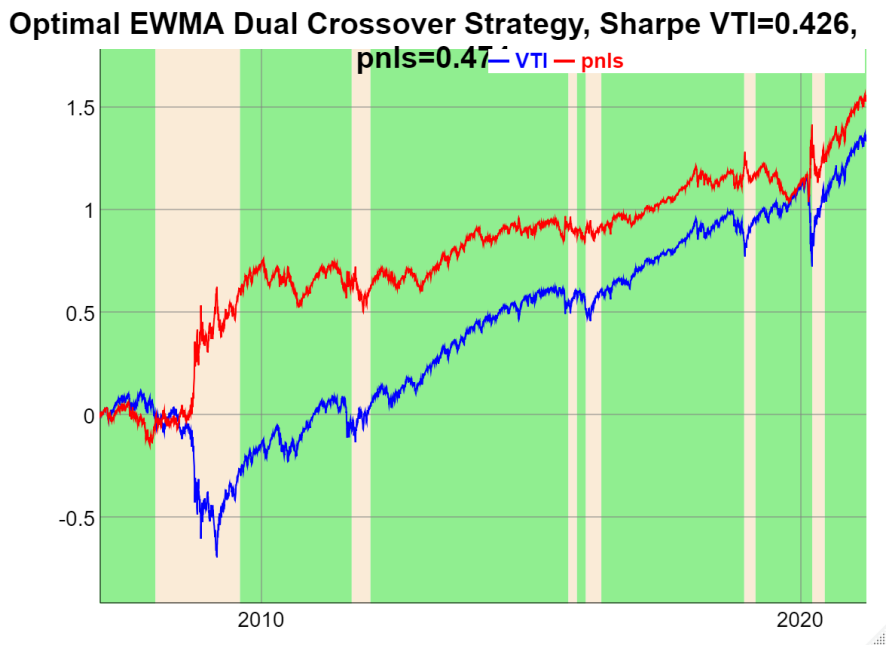
The function `simu_ewma2()` returns the *EWMA* strategy positions and returns, in a two-column *xts* time series.

```
> simu_ewma2 <- function(ohlc, lambda1=0.1, lambda2=0.01, look_back=
+                bid_offer=0.001, trend=1, lagg=1) {
+   close <- log(quantmod::Cl(ohlc))
+   returns <- rutils::diff_it(close)
+   n_rows <- NROW(ohlc)
+   # Calculate EWMA prices
+   weights <- exp(-lambda1*(1:look_back))
+   weights <- weights/sum(weights)
+   ewma1 <- HighFreq::roll_wsum(clos_e, weights=weights)
+   weights <- exp(-lambda2*(1:look_back))
+   weights <- weights/sum(weights)
+   ewma2 <- HighFreq::roll_wsum(clos_e, weights=weights)
+   # Calculate positions, either: -1, 0, or 1
+   indic <- sign(ewma1 - ewma2)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   pos <- rep(NA_integer_, n_rows)
+   pos[1] <- 0
+   pos <- ifelse(indic == lagg, 1, pos)
+   pos <- ifelse(indic == (-lagg), -1, pos)
+   pos <- zoo::na.locf(pos, na.rm=FALSE)
+   pos <- xts::xts(pos, order.by=index(close))
+   # Lag the positions to trade on next day
+   pos <- rutils::lag_it(pos, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- returns*pos
+   costs <- 0.5*bid_offer*abs(rutils::diff_it(pos))*close
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(pos, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end simu_ewma2
```

Optimal Dual EWMA Strategy

Multiple *Dual EWMA* strategies can be simulated by calling the function `simu_ewma2()` in two loops over the vectors of λ parameters.

```
> source("C:/Develop/lecture_slides/scripts/ewma_model.R")
> lamb_das1 <- seq(from=0.05, to=0.15, by=0.01)
> lamb_das2 <- seq(from=0.03, to=0.1, by=0.01)
> # Perform sapply() loops over lambdas
> sharpe_ratios <- sapply(lamb_das1, function(lambda1) {
+   sapply(lamb_das2, function(lambda2) {
+     if (lambda1 > lambda2) {
+ # Simulate Dual EWMA strategy
+ pnl_s <- simu_ewma2(ohlc=oh_lc, lambda1=lambda1, lambda2=lambda2,
+                     look_back=look_back, bid_offer=0.0, trend=1,
+ sqrt(252)*mean(pnl_s)/sd(pnl_s)
+     } else NA
+   })  # end sapply
+ })  # end sapply
> colnames(sharpe_ratios) <- lamb_das1
> rownames(sharpe_ratios) <- lamb_das2
> # Calculate the PnLs for the optimal strategy
> whi_ch <- which(sharpe_ratios == max(sharpe_ratios, na.rm=TRUE),
> lambda1 <- lamb_das1[whi_ch[2]]
> lambda2 <- lamb_das2[whi_ch[1]]
> pnl_s <- simu_ewma2(ohlc=oh_lc, lambda1=lambda1, lambda2=lambda2
+               look_back=look_back, bid_offer=0.0, trend=1, lagg=
> weal_th <- cbind(vt_i, pnl_s)
> # Annualized Sharpe ratio of Dual EWMA strategy
> sharp_e <- sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(weal_th)
```




```
> # Plot Optimal Dual EWMA strategy
> dy_graph <- dygraphs::dygraph(cumsum(weal_th["2007/"]), main=paste
+   dyOptions(colors=c("blue", "red"), strokeWidth=1)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>% dyShading(from=date_s[i], to=date_s[
+ }  # end for
> # Plot the dygraph object
> dy_graph
```

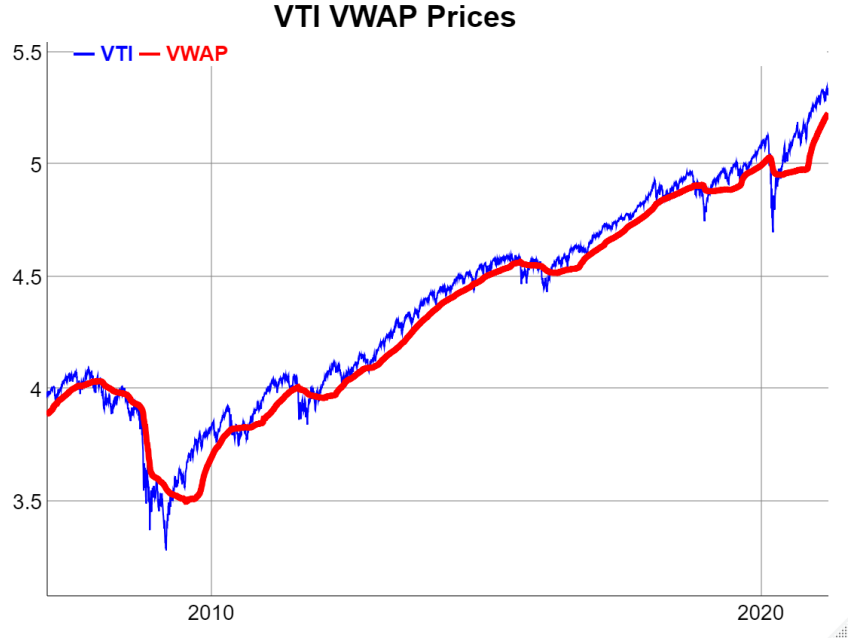
Volume-Weighted Average Price Indicator

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

$$p_i^{VWAP} = \frac{\sum_{j=0}^{n} v_j p_{i-j}}{\sum_{j=0}^{n} v_j}$$

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

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




```
> # Calculate log OHLC prices and volumes
> oh_lc <- rutils::etf_env$VTI
> clos_e <- log(quantmod::Cl(oh_lc))
> colnames(clos_e) <- "VTI"
> vol_ume <- quantmod::Vo(oh_lc)
> colnames(vol_ume) <- "Volume"
> n_rows <- NROW(clos_e)
> # Calculate the VWAP prices
> look_back <- 170
> vwap <- roll::roll_sum(clos_e*vol_ume, width=look_back, min_obs=1
> volume_roll <- roll::roll_sum(vol_ume, width=look_back, min_obs=1
> vwap <- vwap/volume_roll
> colnames(vwap) <- "VWAP"
> price_s <- cbind(clos_e, vwap)
```

```
> # Dygraphs plot with custom line colors
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s["2007/"], main="VTI VWAP Prices") %>%
+   dySeries(name=col_names[1], label=col_names[1], strokeWidth=1, c
+   dySeries(name=col_names[2], label=col_names[2], strokeWidth=4, c
+   dyLegend(show="always", width=500)
> # Plot VWAP prices with custom line colors
> x11(width=6, height=5)
> col_ors <- c("blue", "red")
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(price_s["2009"], theme=plot_theme,
+        lwd=2, name="VTI VWAP Prices")
> legend("bottomright", legend=colnames(price_s),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Simulating the VWAP Crossover Strategy

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

If the current price crosses above the *VWAP*, 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.

To prevent whipsaws and over-trading, the crossover strategy delays switching positions until the indicator repeats the same value for several periods.

```
> # Calculate positions from lagged indicator
> in_dic <- sign(clos_e - vwap)
> lagg <- 2
> in_dic <- roll::roll_sum(in_dic, width=lagg, min_obs=1)
> # Calculate positions, either: -1, 0, or 1
> position_s <- rep(NA_integer_, n_rows)
> position_s[1] <- 0
> position_s <- ifelse(in_dic == lagg, 1, position_s)
> position_s <- ifelse(in_dic == (-lagg), -1, position_s)
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> position_s <- xts::xts(position_s, order.by=index(clos_e))
> # Lag the positions to trade in next period
> position_s <- rutils::lag_it(position_s, lagg=1)
> # Calculate PnLs of VWAP strategy
> pnl_s <- vt_i*position_s
> colnames(pnl_s) <- "VWAP Strategy"
> weal_th <- cbind(vt_i, pnl_s)
> colnames(weal_th) <- c("VTI", "VWAP Strategy")
> col_names <- colnames(weal_th)
> # Annualized Sharpe ratios of VTI and VWAP strategy
> sharp_e <- sqrt(252)*sapply(weal_th, function (x) mean(x)/sd(x))
```


```
> # Create colors for background shading
> date_s <- (rutils::diff_it(position_s) != 0)
> shad_e <- position_s[date_s]
> date_s <- c(index(shad_e), end(position_s))
> shad_e <- ifelse(drop(zoo::coredata(shad_e)) == 1, "lightgreen", "
> # Plot dygraph of VWAP strategy
> # Create dygraph object without plotting it
> dy_graph <- dygraphs::dygraph(cumsum(weal_th["2007/"]), main=paste
+   dyOptions(colors=c("blue", "red"), strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shad_e)) {
+     dy_graph <- dy_graph %>% dyShading(from=date_s[i], to=date_s[i
+ }  # end for
> # Plot the dygraph object
> dy_graph
```

Combining VWAP Crossover Strategy with Stocks

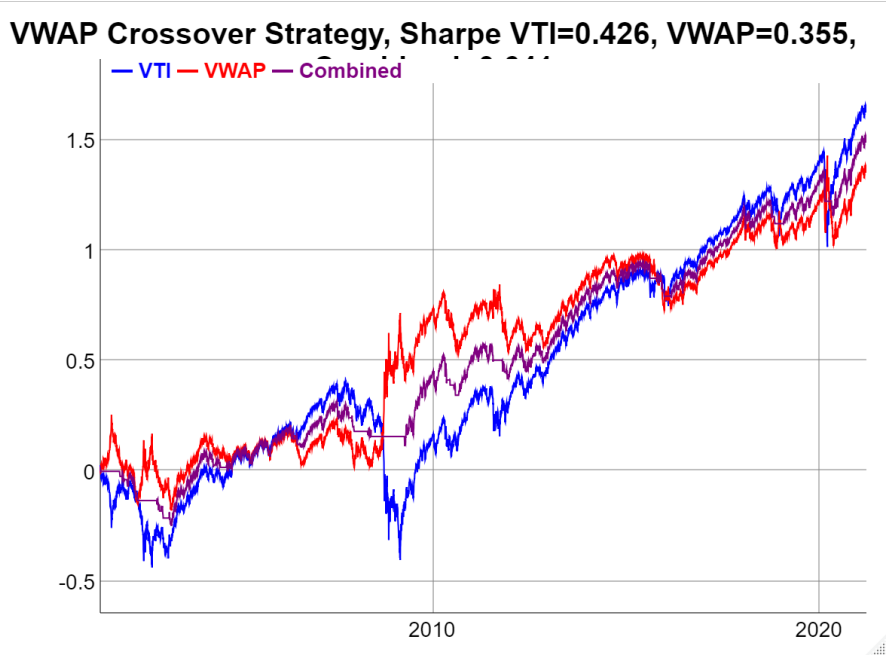
Even though the *VWAP* strategy doesn't perform as well as a static buy-and-hold strategy, it can provide risk reduction when combined with it.

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

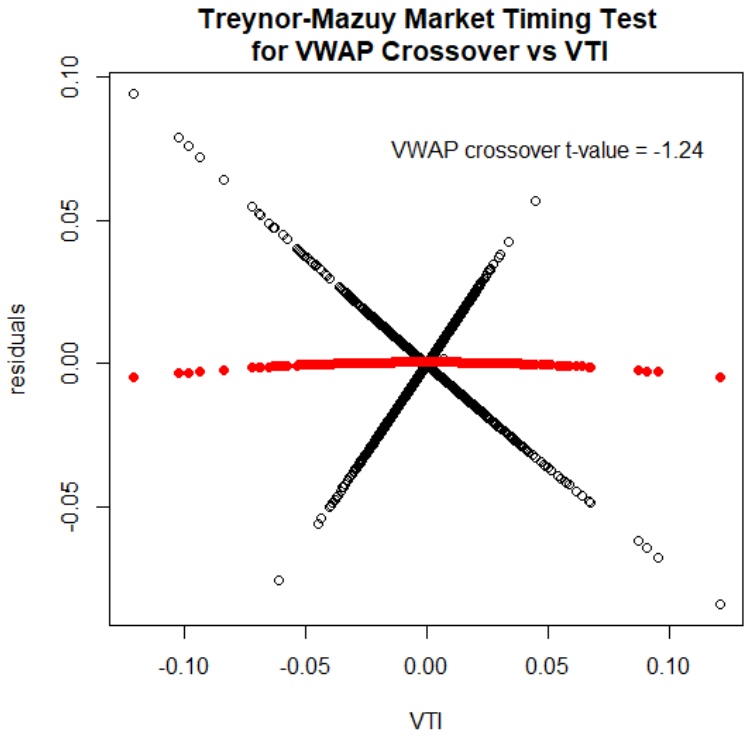



```
> # Plot dygraph of VWAP strategy combined with VTI
> dygraphs::dygraph(cumsum(weal_th),
+   main=paste("VWAP Crossover Strategy, Sharpe", paste(paste(names
+   dyOptions(colors=c("blue", "red", "purple"), strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
```

VWAP Crossover Strategy Market Timing Skill

The VWAP crossover strategy shorts the market during significant selloffs, but otherwise doesn't display market timing skill.

The t-value of the *Treynor-Mazuy* test is negative, but not statistically significant.




```
> # Test VWAP crossover market timing of VTI using Treynor-Mazuy tes
> de_sign <- cbind(pnl_s, vt_i, vt_i^2)
> de_sign <- na.omit(de_sign)
> colnames(de_sign) <- c("VWAP","VTI","treynor")
> mod_el <- lm(VWAP ~ VTI + treynor, data=de_sign)
> summary(mod_el)
> # Plot residual scatterplot
> residual_s <- (de_sign$VWAP - mod_el$coeff[2]*de_sign$VTI)
> residual_s <- mod_el$residuals
> x11(width=6, height=6)
> plot.default(x=de_sign$VTI, y=residual_s, xlab="VTI", ylab="residu
> title(main="Treynor-Mazuy Market Timing Test\n for VWAP Crossover
> # Plot fitted (predicted) response values
> fit_ted <- (mod_el$coeff["(Intercept)"] + mod_el$coeff["treynor"]*
> points.default(x=de_sign$VTI, y=fit_ted, pch=16, col="red")
> text(x=0.05, y=0.8*max(residual_s), paste("VWAP crossover t-value =", round(summary(mod_el)$coeff["treynor", "t value"], 2)))
```

Simulation Function for VWAP Crossover Strategy

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

The function `simu_vwap()` performs a simulation of the *VWAP* strategy, given an *OHLC* time series of prices, and the length of the look-back interval (`look_back`).

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

```
> simu_vwap <- function(ohlc, look_back=333, bid_offer=0.001, trend=
+   close <- log(quantmod::Cl(ohlc))
+   vol_ume <- quantmod::Vo(oh_lc)
+   returns <- rutils::diff_it(close)
+   n_rows <- NROW(ohlc)
+   # Calculate VWAP prices
+   vwap <- roll::roll_sum(clos_e*vol_ume, width=look_back, min_obs=
+   volume_roll <- roll::roll_sum(vol_ume, width=look_back, min_obs=
+   vwap <- vwap/volume_roll
+   # Calculate the indicator
+   indic <- trend*sign(close - vwap)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   # Calculate positions, either: -1, 0, or 1
+   pos <- rep(NA_integer_, n_rows)
+   pos[1] <- 0
+   pos <- ifelse(indic == lagg, 1, pos)
+   pos <- ifelse(indic == (-lagg), -1, pos)
+   pos <- zoo::na.locf(pos, na.rm=FALSE)
+   pos <- xts::xts(pos, order.by=index(close))
+   # Lag the positions to trade on next day
+   pos <- rutils::lag_it(pos, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- returns*pos
+   costs <- 0.5*bid_offer*abs(rutils::diff_it(pos))*close
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(pos, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end simu_vwap
```

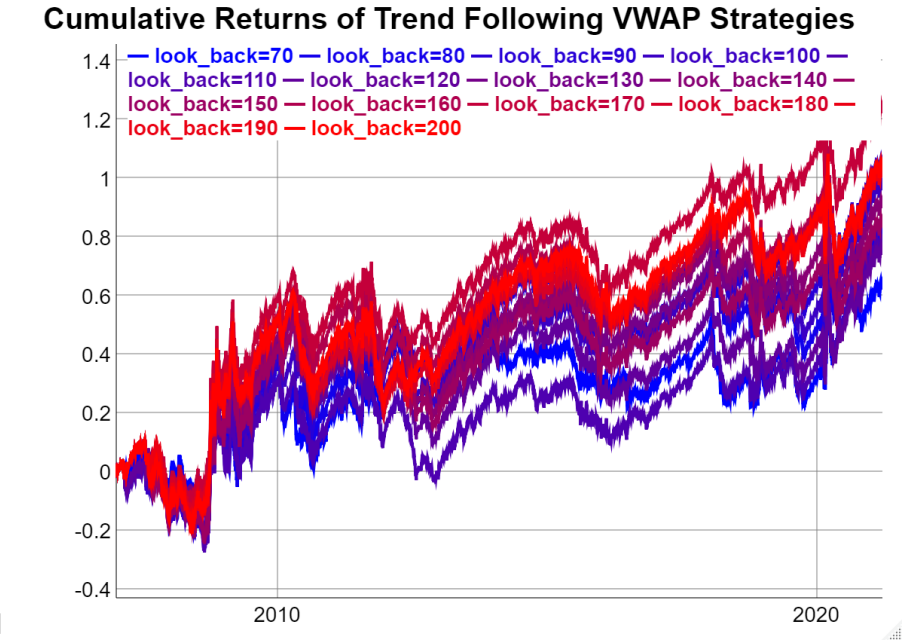
Simulating Multiple Trend Following VWAP Strategies

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

But `simu_vwap()` 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")
> look_backs <- seq(70, 200, 10)
> # Perform lapply() loop over lamb_das
> pnl_s <- lapply(look_backs, function(look_back) {
+   # Simulate VWAP strategy and calculate returns
+   simu_vwap(ohlc=oh_lc, look_back=look_back, bid_offer=0, lagg=2)
+ })  # end lapply
> pnl_s <- do.call(cbind, pnl_s)
> colnames(pnl_s) <- paste0("look_back=", look_backs)
```




```
> # Plot dygraph of multiple VWAP strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s["2007/"]), main="Cumulative Returns
+   dyOptions(colors=col_ors, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Plot VWAP strategies with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- col_ors
> quantmod::chart_Series(cumsum(pnl_s), theme=plot_theme,
+   name="Cumulative Returns of VWAP 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")
```

Monte Carlo Simulation

Monte Carlo simulation consists of generating random samples from a given probability distribution.

The *Monte Carlo* data samples can then used to calculate different parameters of the probability distribution (moments, quantiles, etc.), and its functionals.

The *quantile* of a probability distribution is the value of the *random variable* `x`, such that the probability of values less than `x` is equal to the given *probability* p.

The *quantile* of a data sample can be calculated by first sorting the sample, and then finding the value corresponding closest to the given *probability* p.

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

The function `sort()` returns a vector sorted into ascending order.

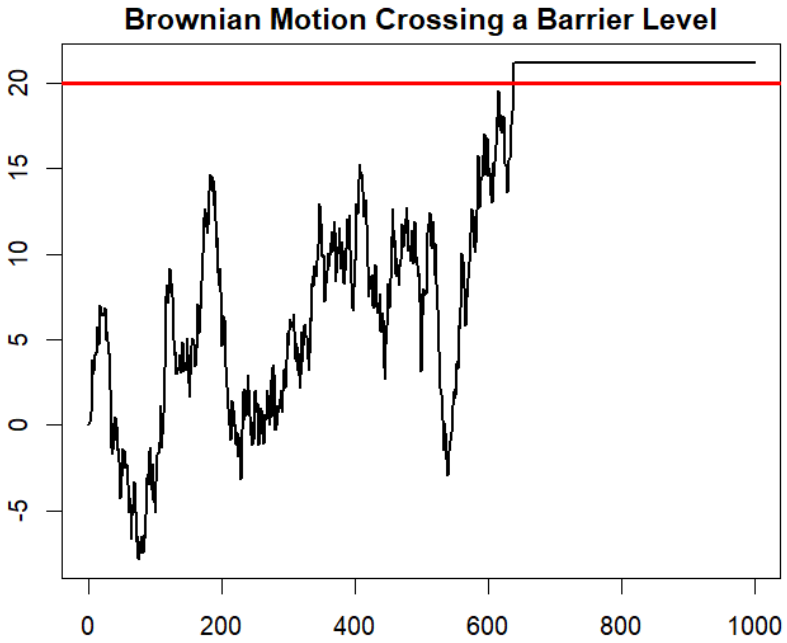
```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> # Sample mean - MC estimate
> mean(da_ta)
> # Sample standard deviation - MC estimate
> sd(da_ta)
> # Monte Carlo estimate of cumulative probability
> pnorm(1)
> sum(da_ta < 1)/n_rows
> # Monte Carlo estimate of quantile
> conf_level <- 0.98
> qnorm(conf_level)  # Exact value
> cut_off <- conf_level*n_rows
> da_ta <- sort(da_ta)
> da_ta[cut_off]  # Naive Monte Carlo value
> quantile(da_ta, probs=conf_level)
> # Analyze the source code of quantile()
> stats:::quantile.default
> # Microbenchmark quantile
> library(microbenchmark)
> summary(microbenchmark(
+   monte_carlo = da_ta[cut_off],
+   quan_tile = quantile(da_ta, probs=conf_level),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

Simulating Brownian Motion Using while() Loops

while() loops are often used in simulations, when the number of required loops is unknown in advance.

Below is an example of a simulation of the path of *Brownian Motion* crossing a barrier level.

```
> set.seed(1121)  # Reset random number generator
> bar_rier <- 20  # Barrier level
> n_rows <- 1000  # Number of simulation steps
> pa_th <- numeric(n_rows)  # Allocate path vector
> pa_th[1] <- 0  # Initialize path
> in_dex <- 2  # Initialize simulation index
> while ((in_dex <= n_rows) && (pa_th[in_dex - 1] < bar_rier)) {
+ # Simulate next step
+   pa_th[in_dex] <- pa_th[in_dex - 1] + rnorm(1)
+   in_dex <- in_dex + 1  # Advance in_dex
+ }  # end while
> # Fill remaining pa_th after it crosses bar_rier
> if (in_dex <= n_rows)
+   pa_th[in_dex:n_rows] <- pa_th[in_dex - 1]
> # Plot the Brownian motion
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(pa_th, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=bar_rier, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```

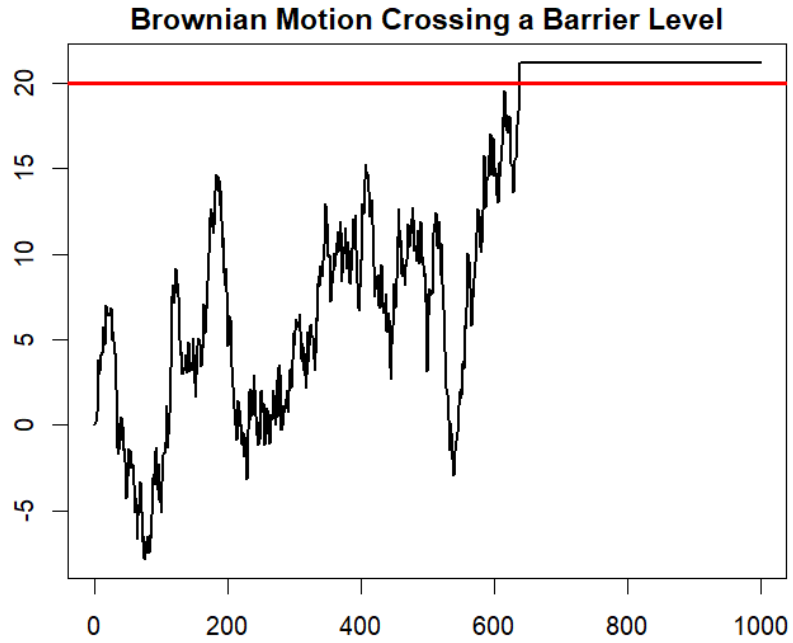


Simulating Brownian Motion Using Vectorized Functions

Simulations in R can be accelerated by pre-computing a vector of random numbers, instead of generatng them one at a time in a loop.

Vectors of random numbers allow using *vectorized* functions, instead of inefficient (slow) `while()` loops.

```
> set.seed(1121)  # Reset random number generator
> bar_rier <- 20  # Barrier level
> n_rows <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pa_th <- cumsum(rnorm(n_rows))
> # Find index when pa_th crosses bar_rier
> cro_ss <- which(pa_th > bar_rier)
> # Fill remaining pa_th after it crosses bar_rier
> if (NROW(cro_ss)>0) {
+   pa_th[(cro_ss[1]+1):n_rows] <- pa_th[cro_ss[1]]
+ }  # end if
> # Plot the Brownian motion
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(pa_th, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=bar_rier, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```



The tradeoff between speed and memory usage: more memory may be used than necessary, since the simulation may stop before all the pre-computed random numbers are used up.

But the simulation is much faster because the path is simulated using *vectorized* functions,

Geometric Brownian Motion

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

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

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

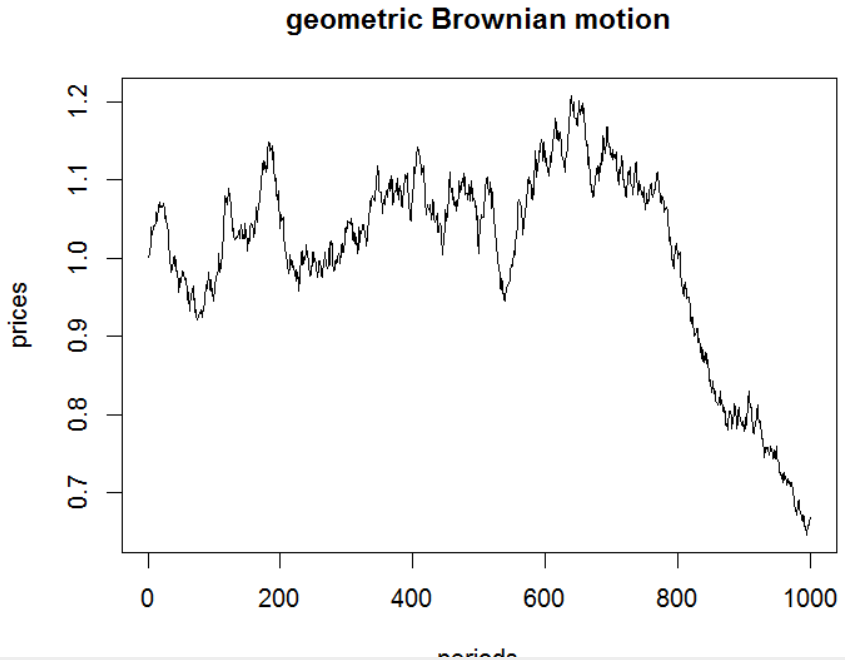
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}W_t$$

Where σ is the volatility of asset returns, and 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 *Geometric Brownian motion* is equal to:

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

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




```
> # Define daily volatility and growth rate
> sig_ma <- 0.01; dri_ft <- 0.0; n_rows <- 1000
> # Simulate geometric Brownian motion
> re_turns <- sig_ma*rnorm(n_rows) + dri_ft - sig_ma^2/2
> price_s <- exp(cumsum(re_turns))
> plot(price_s, type="l", xlab="time", ylab="prices",
+      main="geometric Brownian motion")
```

Simulating Random *OHLC* Prices

Random *OHLC* prices are useful for testing financial models.

The function `sample()` selects a random sample from a vector of data elements.

The function `sample()` with `replace=TRUE` selects samples with replacement (the default is `replace=FALSE`).

```
> # Simulate geometric Brownian motion
> sig_ma <- 0.01/sqrt(48)
> dri_ft <- 0.0
> n_rows <- 1e4
> date_s <- seq(from=as.POSIXct(paste(Sys.Date()-250, "09:30:00")),
+   length.out=n_rows, by="30 min")
> price_s <- exp(cumsum(sig_ma*rnorm(n_rows) + dri_ft - sig_ma^2/2))
> price_s <- xts(price_s, order.by=date_s)
> price_s <- cbind(price_s,
+   volume=sample(x=10*(2:18), size=n_rows, replace=TRUE))
> # Aggregate to daily OHLC data
> oh_lc <- xts::to.daily(price_s)
> quantmod::chart_Series(oh_lc, name="random prices")
> # dygraphs candlestick plot using pipes syntax
> library(dygraphs)
> dygraphs::dygraph(oh_lc[, 1:4]) %>% dyCandlestick()
> # dygraphs candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dygraph(oh_lc[, 1:4]))
```

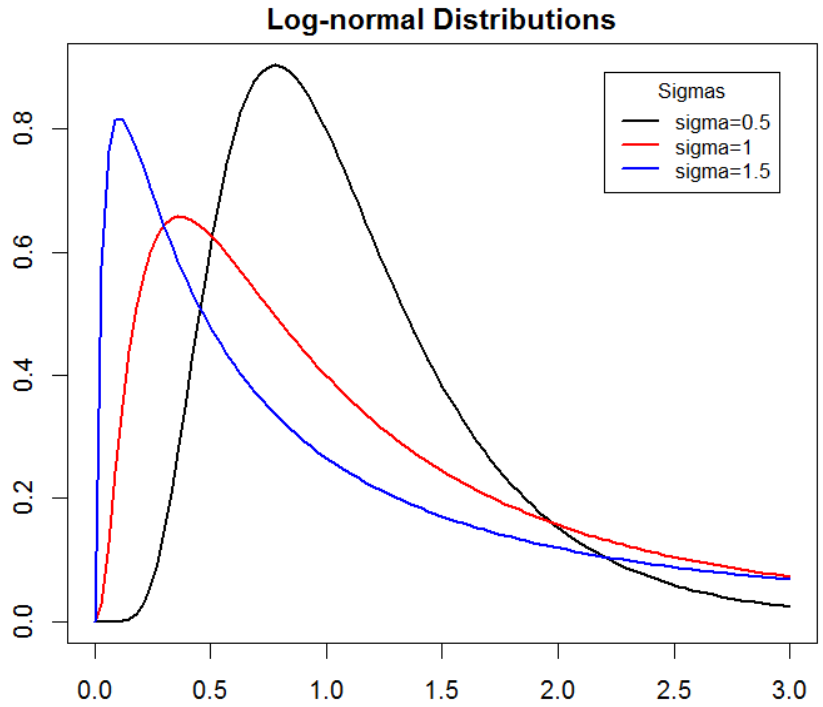
The *Log-normal* Probability Distribution

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

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

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

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




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

The Standard Deviation of *Log-normal* Prices

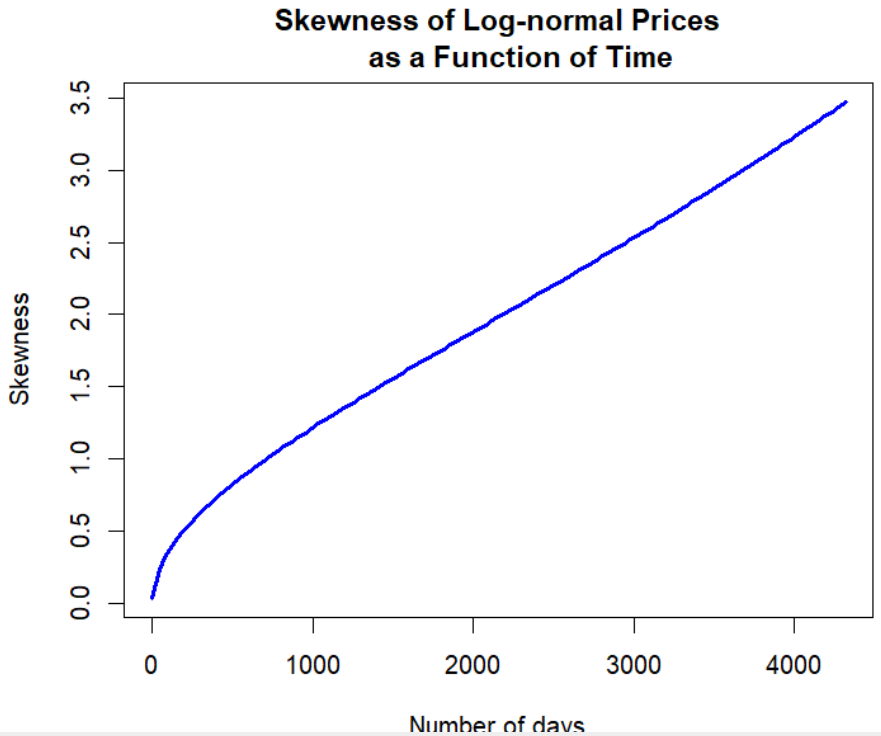
If percentage asset returns are *normally* distributed and follow *Brownian motion*, then asset prices follow *Geometric Brownian motion*, and they are *Log-normally* distributed at every point in time.

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

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

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

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




```
> # Return volatility of VTI etf
> sig_ma <- sd(rutils::diff_it(log(rutils::etf_env$VTI[, 4])))
> sigma2 <- sig_ma^2
> n_rows <- NROW(rutils::etf_env$VTI)
> # Standard deviation of log-normal prices
> sqrt(n_rows)*sig_ma
```

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

The Mean and Median of *Log-normal* Prices

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

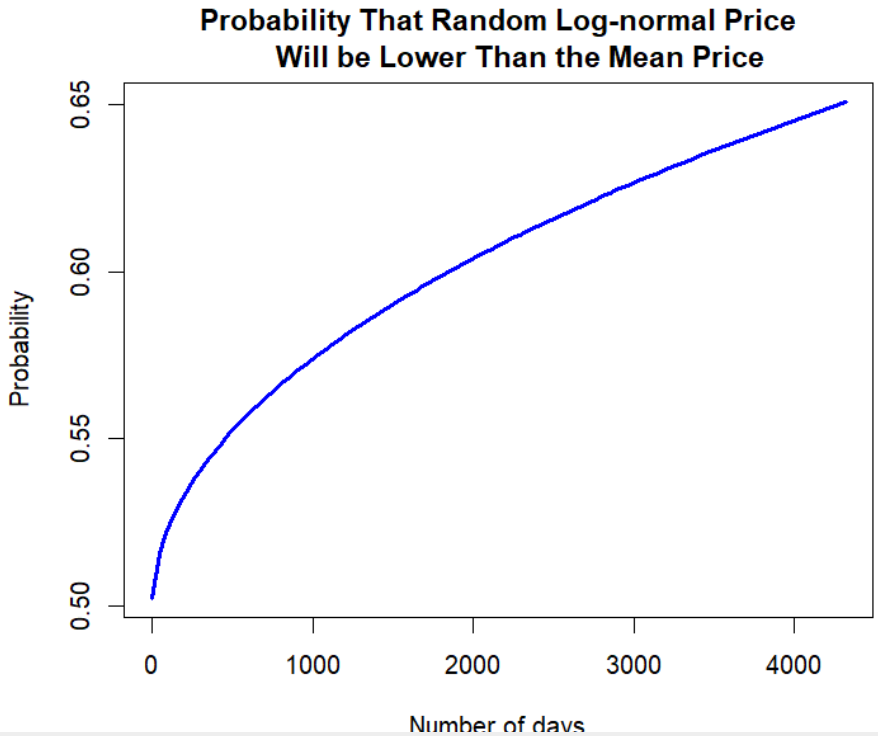
So if stock prices follow *Geometric Brownian motion* and are distributed *log-normally*, then a stock selected at random will have a high probability of havng a lower price than the mean expected price.

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

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

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

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



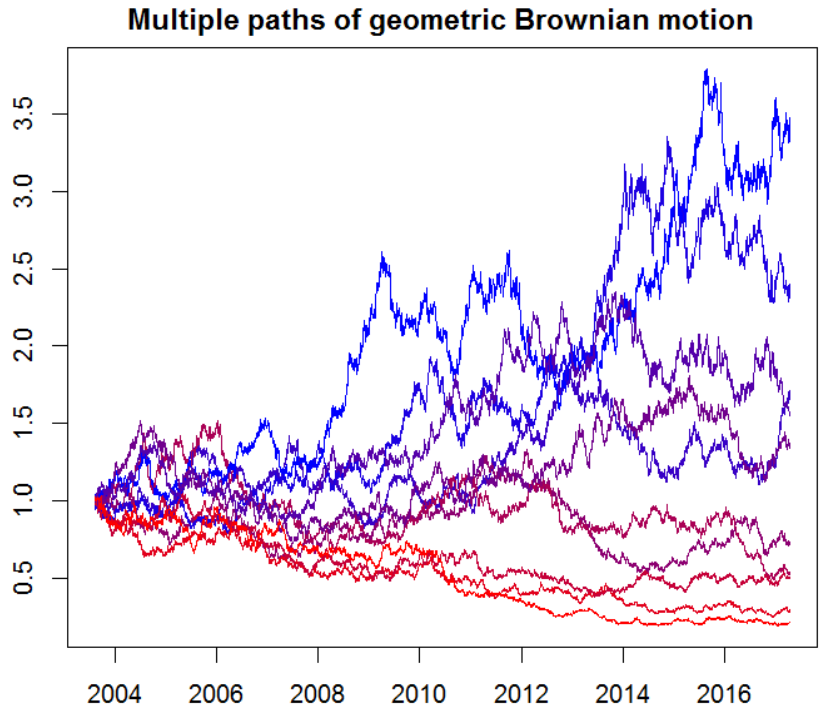

```
> # Probability that random log-normal price will be lower than the
> curve(expr=pnorm(sig_ma*sqrt(x)/2),
+ xlim=c(1, n_rows), lwd=3,
+ xlab="Number of days", ylab="Probability", col="blue",
+ main="Probability That Random Log-normal Price
+ Will be Lower Than the Mean Price")
```

Paths of Geometric Brownian Motion

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

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

```
> # Define daily volatility and growth rate
> sig_ma <- 0.01; dri_ft <- 0.0; n_rows <- 5000
> path_s <- 10
> # Simulate multiple paths of geometric Brownian motion
> price_s <- matrix(rnorm(path_s*n_rows, sd=sig_ma) +
+     dri_ft - sig_ma^2/2, nc=path_s)
> price_s <- exp(matrixStats::colCumsums(price_s))
> # Create xts time series
> price_s <- xts(price_s, order.by=seq.Date(Sys.Date()-NROW(price_s)
> # Plot xts time series
> col_ors <- colorRampPalette(c("red", "blue"))(NCOL(price_s))
> col_ors <- col_ors[order(order(price_s[NROW(price_s), ]))]
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(price_s, main="Multiple paths of geometric Brownian motion",
+    xlab=NA, ylab=NA, plot.type="single", col=col_ors)
```

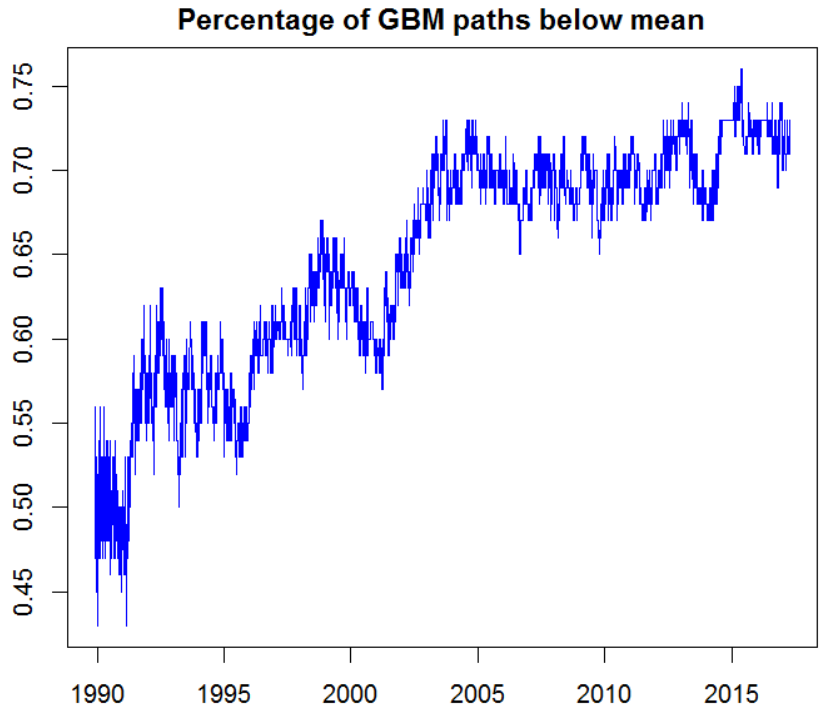


Distribution of Paths of Geometric Brownian Motion

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

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

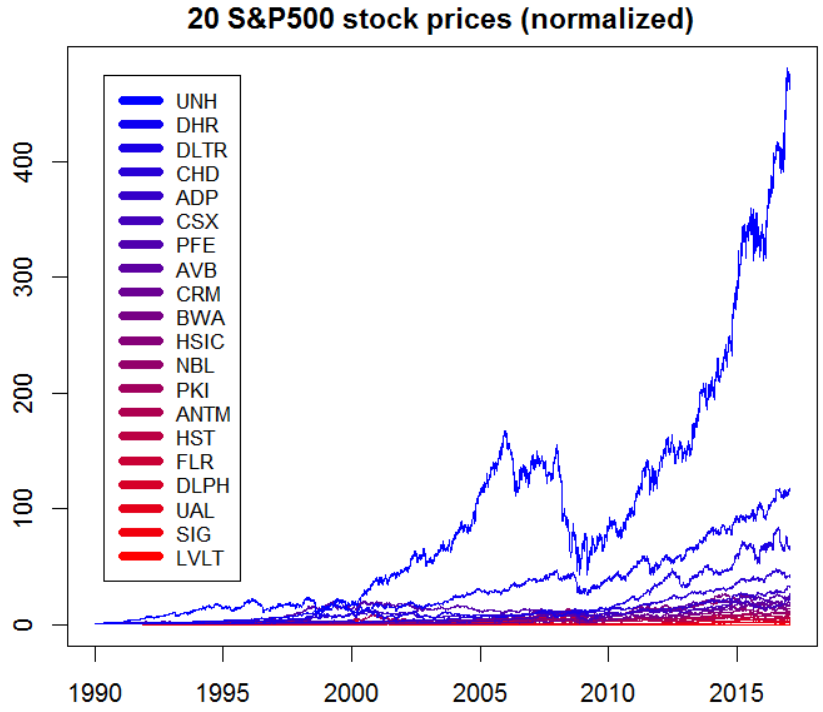
```
> # Define daily volatility and growth rate
> sig_ma <- 0.01; dri_ft <- 0.0; n_rows <- 10000
> path_s <- 100
> # Simulate multiple paths of geometric Brownian motion
> price_s <- matrix(rnorm(path_s*n_rows, sd=sig_ma) +
+     dri_ft - sig_ma^2/2, nc=path_s)
> price_s <- exp(matrixStats::colCumsums(price_s))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



Time Evolution of Stock Prices

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

```
> # Load S&P500 stock prices
> load("C:/Develop/lecture_slides/data/sp500.RData")
> ls(sp500_env)
> # Extract closing prices
> price_s <- eapply(sp500_env, quantmod::Cl)
> # Flatten price_s into a single xts series
> price_s <- rutils::do_call(cbind, price_s)
> # Carry forward and backward non-NA prices
> price_s <- zoo::na.locf(price_s, na.rm=FALSE)
> price_s <- zoo::na.locf(price_s, fromLast=TRUE)
> sum(is.na(price_s))
> # Drop ".Close" from column names
> colnames(price_s[, 1:4])
> colnames(price_s) <- rutils::get_name(colnames(price_s))
> # Or
> # colnames(price_s) <- do.call(rbind,
> #   strsplit(colnames(price_s), split="[.]"))[, 1]
> # Normalize columns
> price_s <- xts(t(t(price_s) / as.numeric(price_s[1, ])),
+          order.by=index(price_s))
> # Calculate permution index for sorting the lowest to highest fi
> or_der <- order(price_s[NROW(price_s), ])
> # Select a few symbols
> sym_bols <- colnames(price_s)[or_der]
> sym_bols <- sym_bols[seq.int(from=1, to=(NROW(sym_bols)-1), leng
```




```
> # Plot xts time series of price_s
> col_ors <- colorRampPalette(c("red", "blue"))(NROW(sym_bols))
> col_ors <- col_ors[order(order(price_s[NROW(price_s), sym_bols]))]
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(price_s[, sym_bols], main="20 S&P500 stock prices (normal
+    xlab=NA, ylab=NA, plot.type="single", col=col_ors)
> legend(x="topleft", inset=0.05, cex=0.8,
+  legend=rev(sym_bols), col=rev(col_ors), lwd=6, lty=1)
```

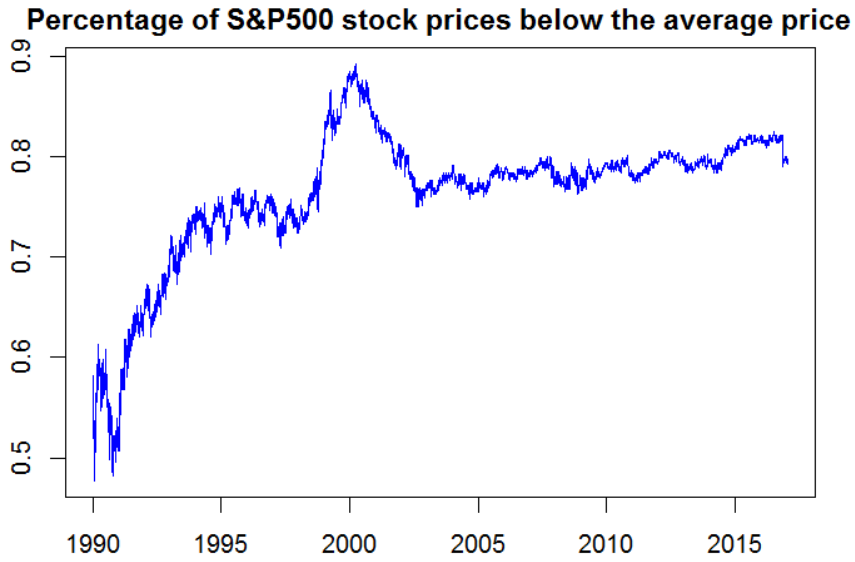
Distribution of Stock Prices

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

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

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

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




```
> # Calculate average of valid stock prices
> val_id <- (price_s != 1)  # Valid stocks
> n_stocks <- rowSums(val_id)
> n_stocks[1] <- NCOL(price_s)
> in_dex <- rowSums(price_s * val_id) / n_stocks
> # Calculate percentage of stock prices below the average price
> per_centage <- rowSums((price_s < in_dex) & val_id) / n_stocks
> # Create xts time series of average stock prices
> in_dex <- xts(in_dex, order.by=index(price_s))
```

```
> # Plot xts time series of average stock prices
> plot.zoo(in_dex, main="Average S&P500 stock prices (normalized fr
+    xlab=NA, ylab=NA, col="blue")
> # Create xts time series of percentage of stock prices below the a
> per_centage <- xts(per_centage, order.by=index(price_s))
> # Plot percentage of stock prices below the average price
> plot.zoo(per_centage[-(1:2),],
+    main="Percentage of S&P500 stock prices below the average price
+    xlab=NA, ylab=NA, col="blue")
```

The Brownian Motion Process

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

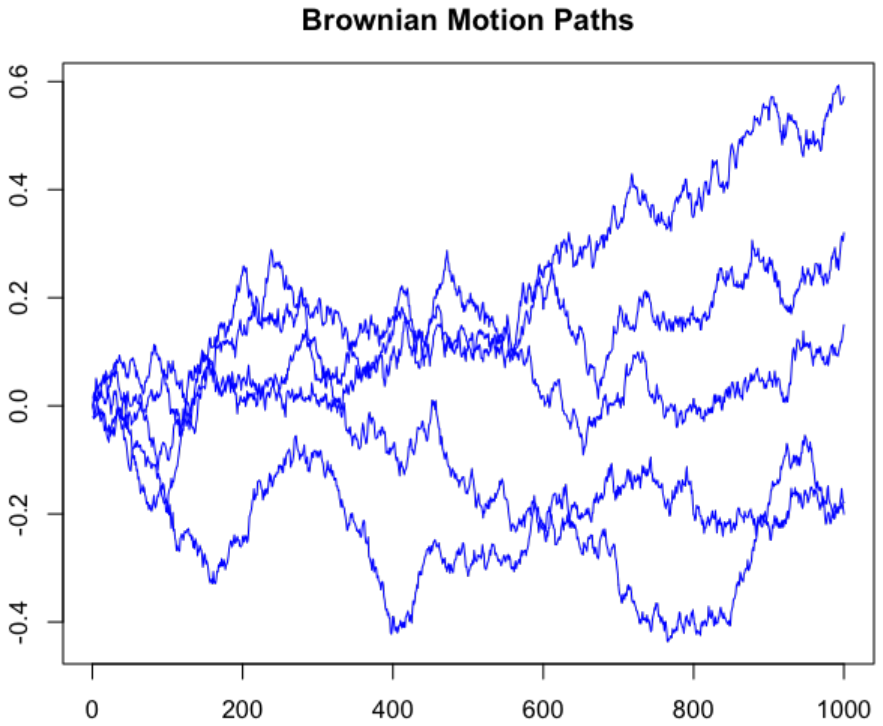
$$r_i = p_i - p_{i-1} = \sigma \xi_i$$

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

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
> init_price <- 0.0; 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)
> price_s[1] <- init_price
> # Simulate Ornstein-Uhlenbeck process in R
> 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(init_price=init_price, eq_price=eq_
+   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(init_price=init_price, 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 $\phi(0, \sqrt{\mathrm{d}t})$.

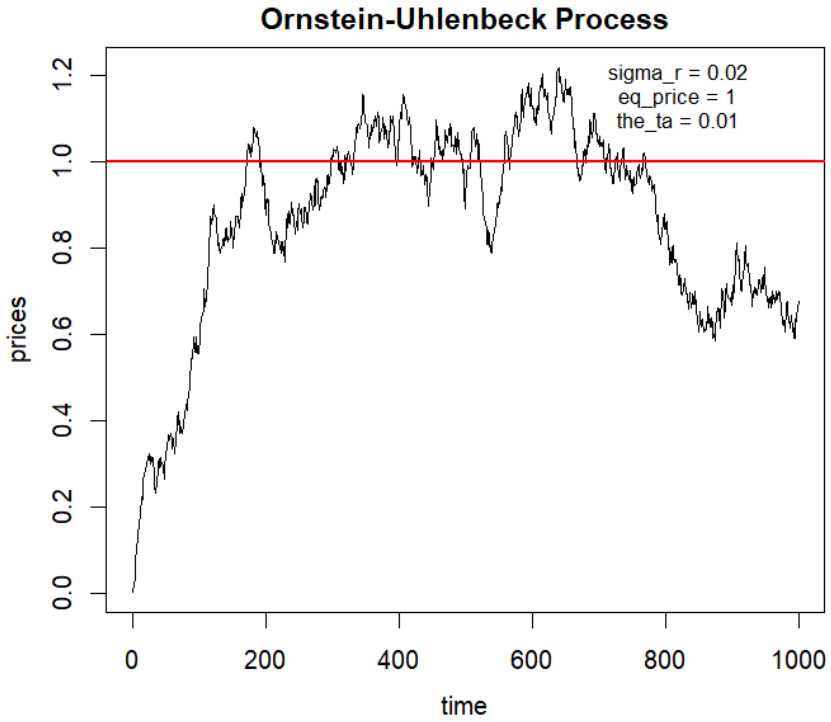
The solution of the *Ornstein-Uhlenbeck* process is given by:

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

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.

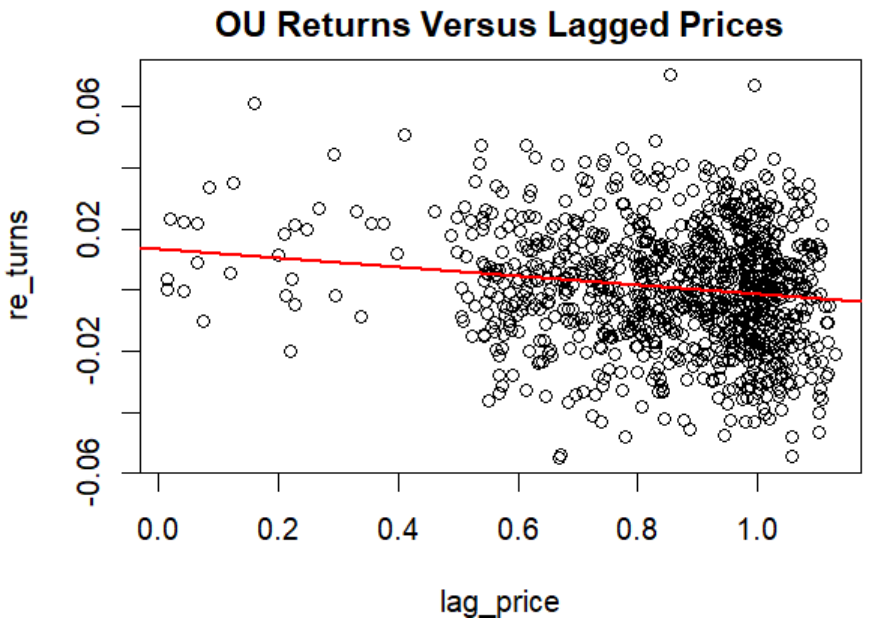



```
> plot(price_s, type="l",
+      xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright",
+  title=paste(c(paste0("sig_ma = ", sig_ma),
+                paste0("eq_price = ", eq_price),
+                paste0("the_ta = ", the_ta)),
+              collapse="\n"),
+  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.

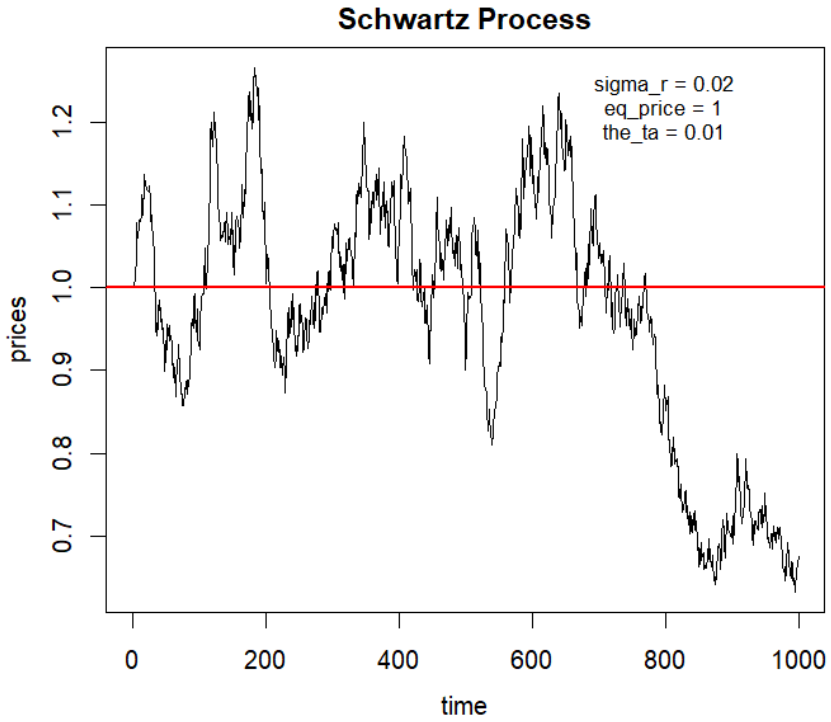
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of prices, which can be negative.

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:

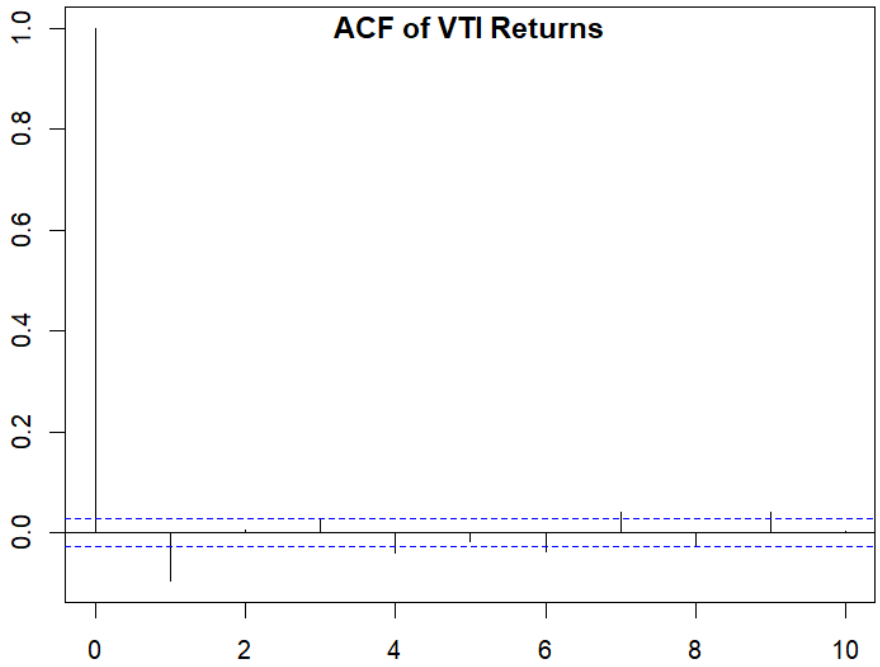
$$\rho_k = \frac{\sum_{i=k+1}^{n}(x_i - \bar{x})(x_{i-k} - \bar{x})}{(n-k)\,\sigma^2}$$

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

The *n*-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)
```

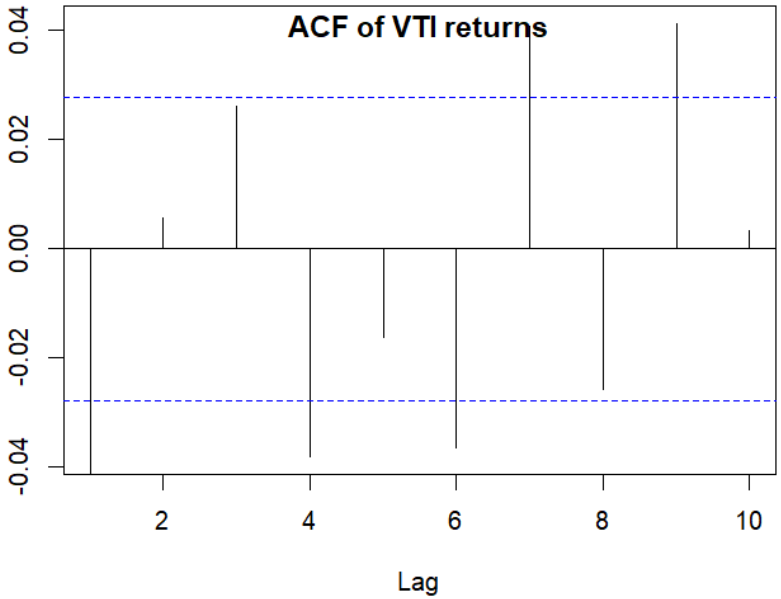
```
> plot_acf <- function(x_ts, lagg=10, plo_t=TRUE,
+                xlab="Lag", ylab="", main="", ...) {
+   # Calculate the ACF without a plot
+   acf_data <- acf(x=x_ts, lag.max=lagg, plot=FALSE, ...)
+   # Remove first element of ACF data
+   acf_data$acf <- array(data=acf_data$acf[-1],
+     dim=c((dim(acf_data$acf)[1]-1), 1, 1))
+   acf_data$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")
```



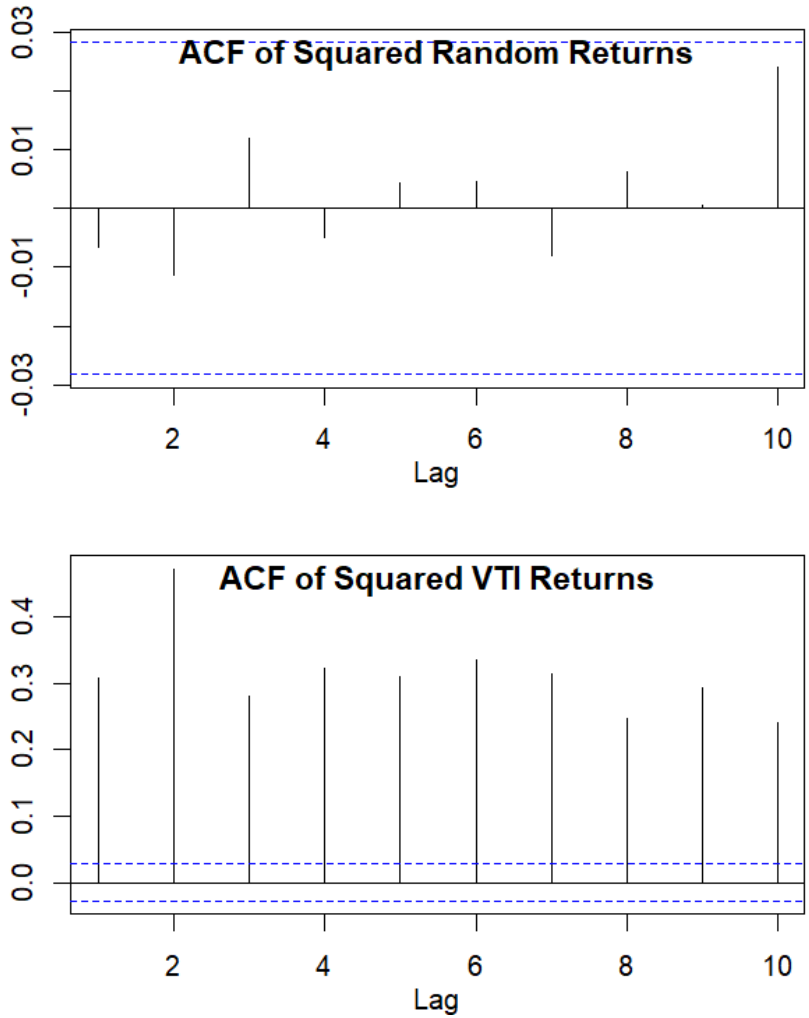
Autocorrelation of Squared *VTI* Returns

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")
```



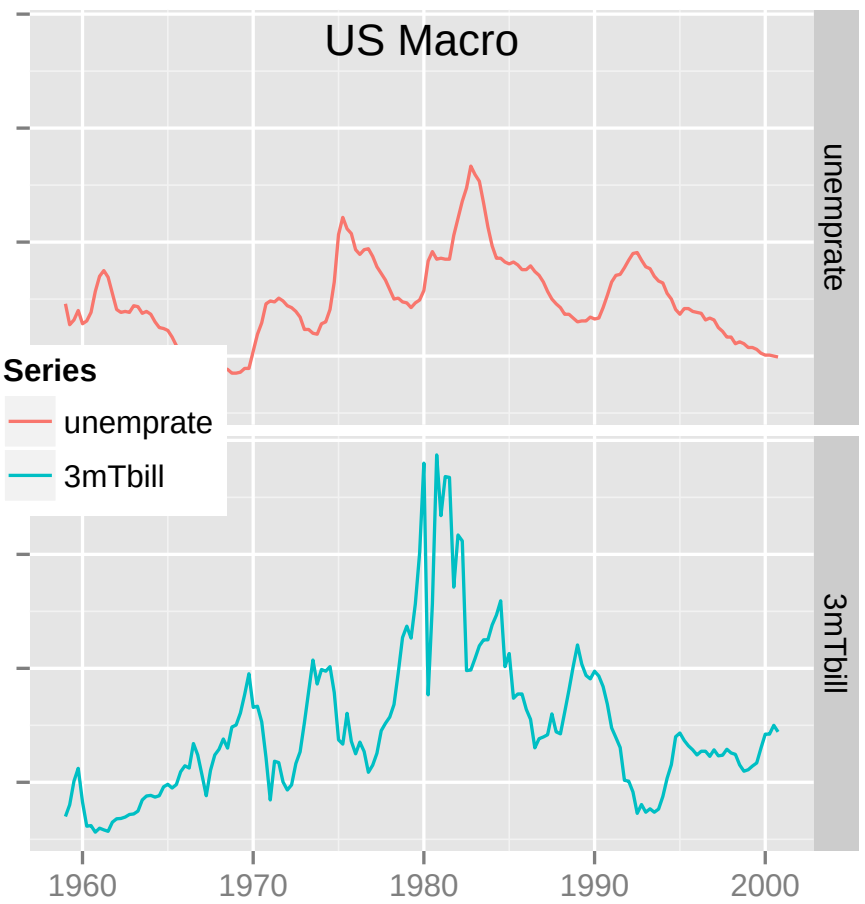
U.S. Macroeconomic Data

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
```



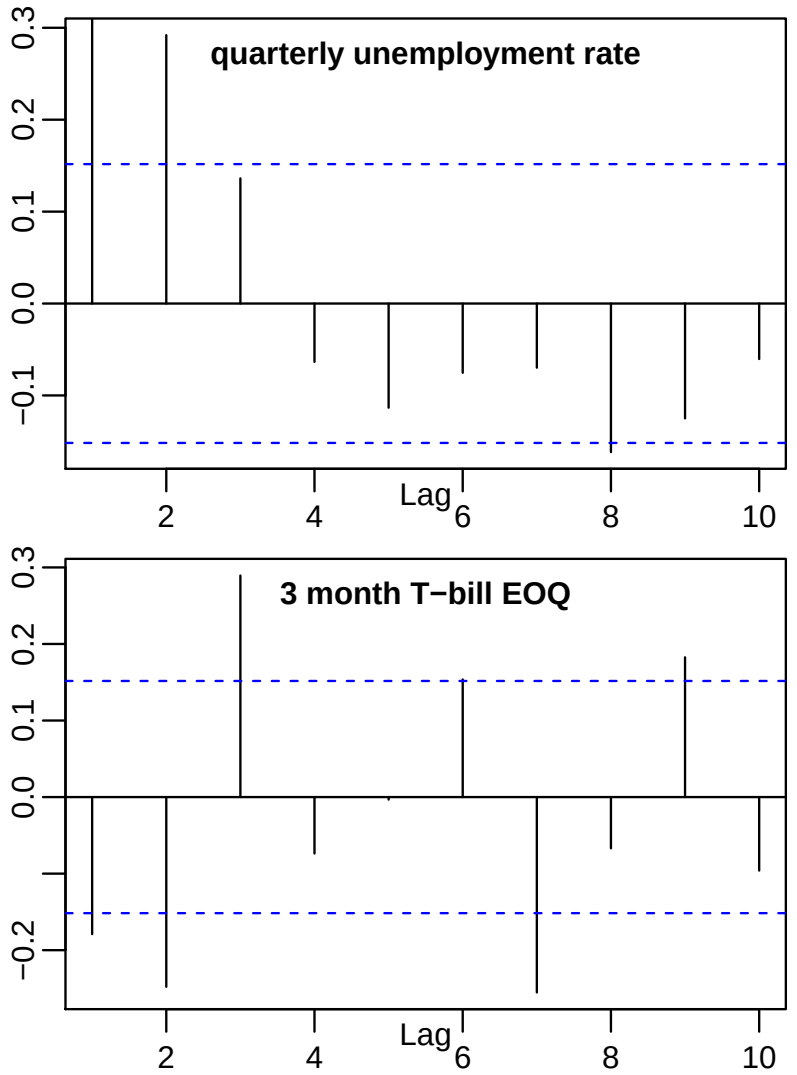
Autocorrelation of Econometric Data

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(n)* of order n for a time series r_i is defined as:

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

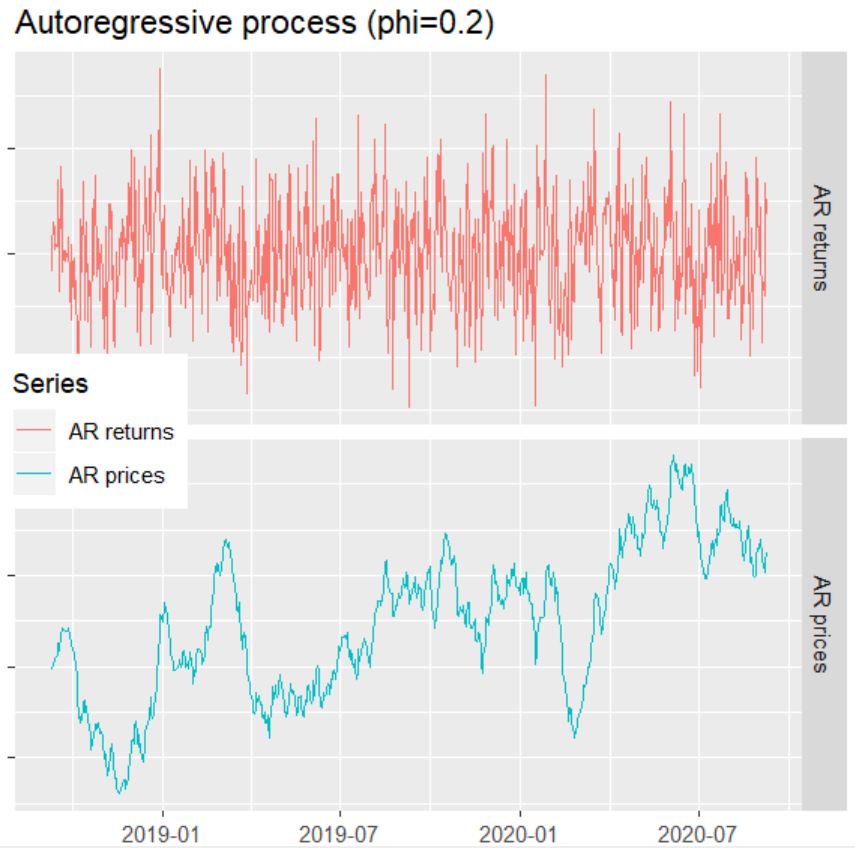
Where φ_i are the *AR(n)* coefficients, and ξ_i are standard normal *innovations*.

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 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")
```



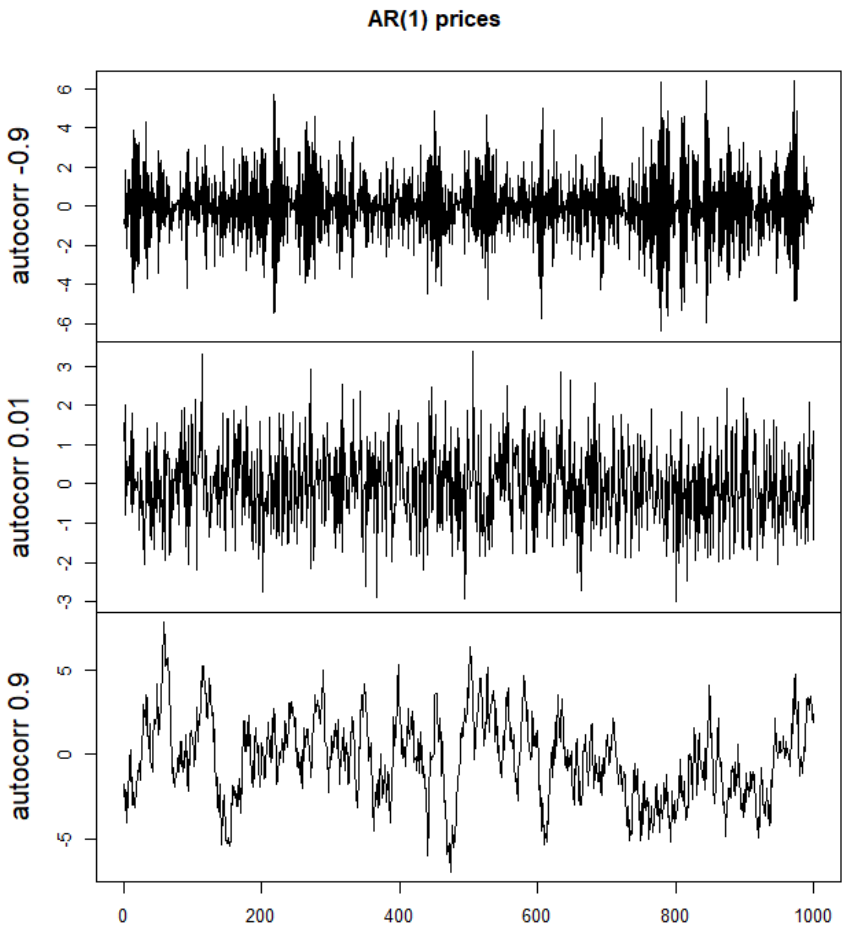

```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=ari_ma, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),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

A positive coefficient φ_1 produces a diversion away from the mean, so that the time series r_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(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"`.

```
> # 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(n)* 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: $\rho_i = \varphi^i$.

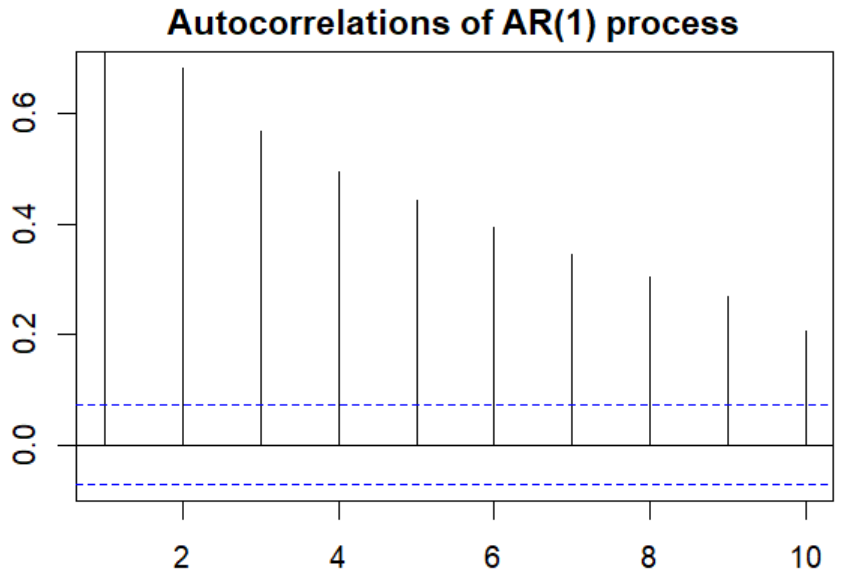
The *AR(1)* process can be solved recursively:

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

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.

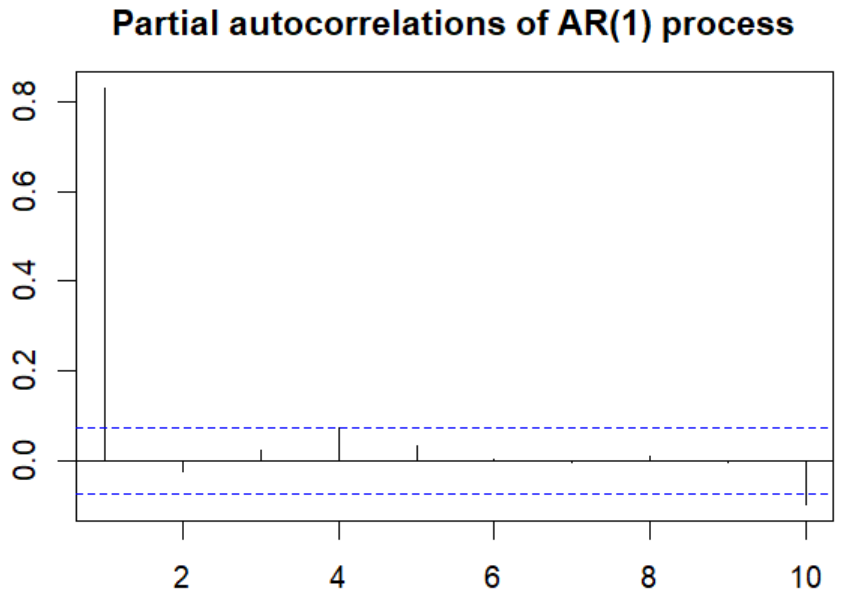
The indirect correlation can be removed by defining new variables with no correlation to the third variable.

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

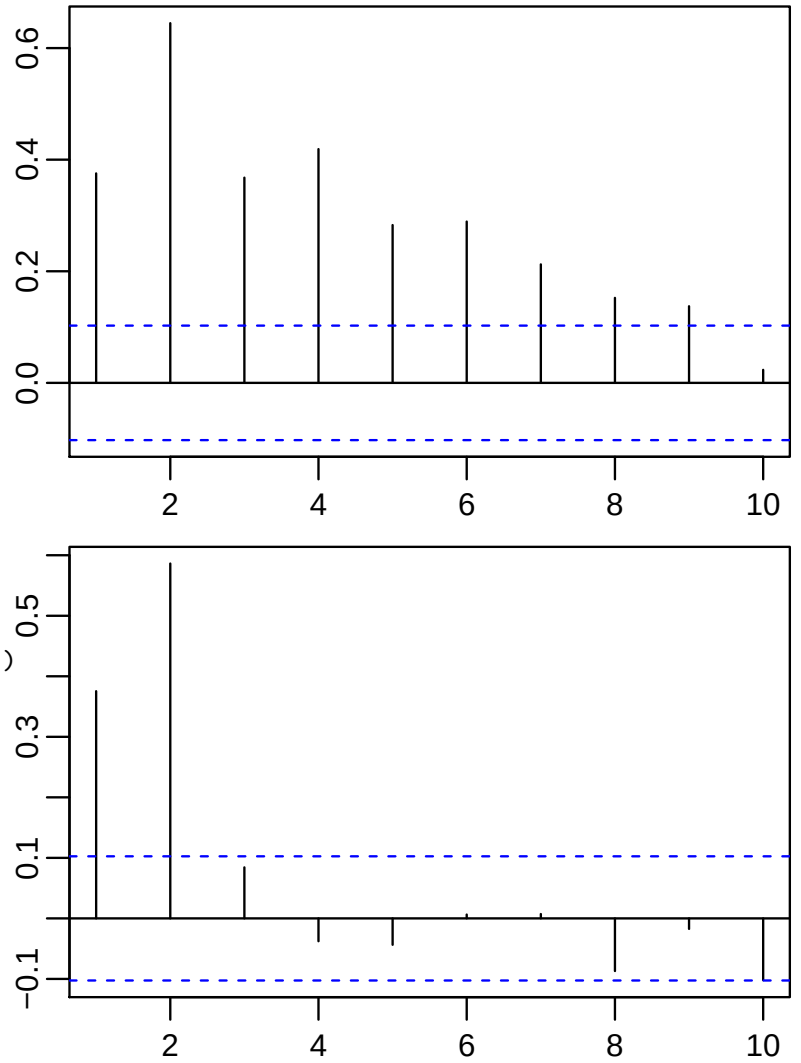
An *AR(3)* process of order *three* is defined by the formula:

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

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

The number of non-zero *partial autocorrelations* is equal to the *order* parameter *n* of the *AR(n)* 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(n)*:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$

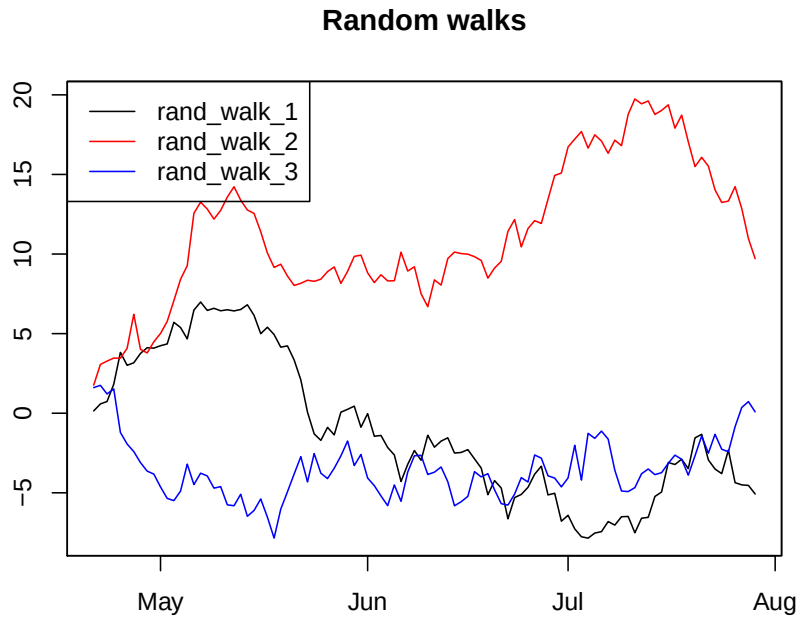
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_n z^n = 0$

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




```
> 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(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 φ_i coefficients.

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

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

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

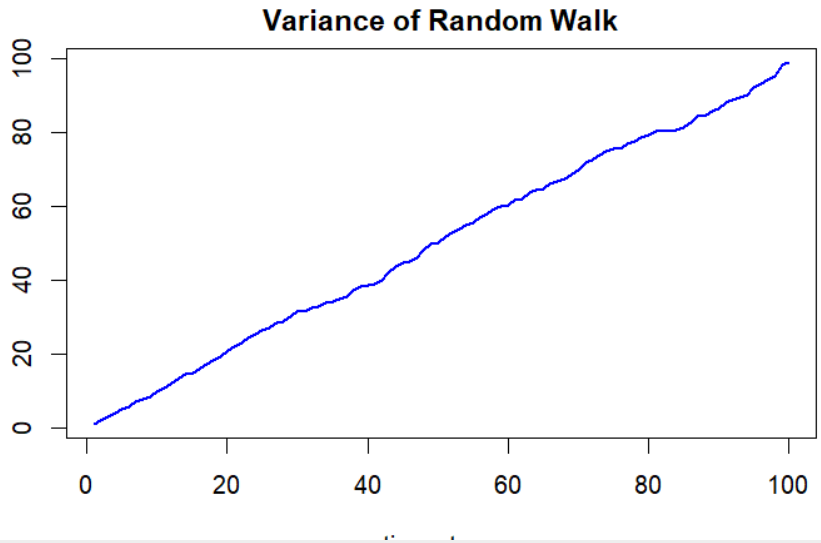
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

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:

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

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

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

Then the prices follow an *autoregressive* process:

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

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

```
> # Define Dickey-Fuller parameters
> init_price <- 0.0; 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(init_price=init_price, eq_price=eq
+   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 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 μ is the equilibrium price, σ is the volatility of returns, and θ is the strength of mean reversion.

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

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_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 function `tseries::adf.test()` performs the *ADF* test.

```
> set.seed(1121); in_nov <- matrix(rnorm(1e4, sd=0.01))
> # Simulate AR(1) process with coefficient=1, with unit root
> ari_ma <- HighFreq::sim_ar(coeff=matrix(1), innov=in_nov)
> 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 <- HighFreq::sim_ar(coeff=matrix(0.99), innov=in_nov)
> 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
> init_price <- 0.0; eq_price <- 0.0; the_ta <- 0.1
> price_s <- HighFreq::sim_ou(init_price=init_price, eq_price=eq_pri
+   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(init_price=init_price, eq_price=eq_pri
+   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)
```

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: $r_i = \theta(\mu - p_{i-1}) + \varepsilon_i$.

Fitting Time Series to Autoregressive Models

An *autoregressive* process *AR(n)* for the time series of returns r_i:

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

Can be solved as a *multivariate* linear regression, with the *response* equal to r_i, and the columns of the *design matrix* equal to the lags of r_i.

An intercept term can be added to the above formula by adding a unit column to the regression design matrix.

Adding the intercept term produces slightly different coefficients, depending on the mean of the returns.

The function `stats::ar.ols()` fits an *AR(n)* model, but it produces slightly different coefficients than linear regression, because it uses a different calibration procedure.

```
> # Specify AR process parameters
> n_rows <- 1e3
> co_eff <- matrix(c(0.1, 0.39, 0.5)); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- matrix(rnorm(n_rows))
> # ari_ma <- filter(x=in_nov, filter=co_eff, method="recursive")
> # Simulate AR process using HighFreq::sim_ar()
> ari_ma <- HighFreq::sim_ar(coeff=co_eff, innov=in_nov)
> # Fit AR model using ar.ols()
> ar_fit <- ar.ols(ari_ma, order.max=n_coeff, aic=FALSE)
> class(ar_fit)
> is.list(ar_fit)
> drop(ar_fit$ar); drop(co_eff)
> # Define design matrix without intercept column
> de_sign <- sapply(1:n_coeff, rutils::lag_it, in_put=ari_ma)
> # Fit AR model using regression
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% ari_ma)
> all.equal(drop(ar_fit$ar), coeff_fit, check.attributes=FALSE)
```

The Standard Errors of the *AR(n)* Coefficients

The *standard errors* of the fitted *AR(n)* coefficients are proportional to the standard deviation of the fitted residuals.

Their *t*-values are equal to the ratio of the fitted coefficients divided by their standard errors.

```
> # Calculate the regression residuals
> fit_ted <- drop(de_sign %*% coeff_fit)
> residual_s <- drop(ari_ma - fit_ted)
> # Variance of residuals
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> # Design matrix squared
> design_2 <- crossprod(de_sign)
> # Calculate covariance matrix of AR coefficients
> co_var <- var_resid*MASS::ginv(design_2)
> coeff_fitd <- sqrt(diag(co_var))
> # Calculate t-values of AR coefficients
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
```

Order Selection of *AR(n)* Model

Order selection means determining the *order parameter* n of the *AR(n)* model that best fits the time series.

The order parameter n can be set equal to the number of significantly non-zero *partial autocorrelations* of the time series.

The order parameter can also be determined by only selecting coefficients with statistically significant t-values.

Fitting an *AR(n)* model can be performed by first determining the order n, and then calculating the coefficients.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series.

The function `auto.arima()` from the package *forecast* performs order selection, and calibrates an *AR(n)* model to a univariate time series.

```
> # Fit AR(5) model into AR(3) process
> de_sign <- sapply(1:5, rutils::lag_it, in_put=ari_ma)
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% ari_ma)
> # Calculate t-values of AR(5) coefficients
> residual_s <- drop(ari_ma - drop(de_sign %*% coeff_fit))
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> co_var <- var_resid*MASS::ginv(crossprod(de_sign))
> coeff_fitd <- sqrt(diag(co_var))
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
> # Fit AR(5) model using arima()
> arima_fit <- arima(ari_ma, order=c(5, 0, 0), include.mean=FALSE)
> arima_fit$coef
> # Fit AR(5) model using auto.arima()
> library(forecast)  # Load forecast
> arima_fit <- forecast::auto.arima(ari_ma, max.p=5, max.q=0, max.d=
> # Fit AR(5) model into VTI returns
> re_turns <- drop(zoo::coredata(na.omit(rutils::etf_env$re_turns$VT
> de_sign <- sapply(1:5, rutils::lag_it, in_put=re_turns)
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% re_turns)
> # Calculate t-values of AR(5) coefficients
> residual_s <- drop(re_turns - drop(de_sign %*% coeff_fit))
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> co_var <- var_resid*MASS::ginv(crossprod(de_sign))
> coeff_fitd <- sqrt(diag(co_var))
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
```

The Yule-Walker Equations

To lighten the notation we can assume that the time series r_i has zero mean $\mathbb{E}[r_i] = 0$ and unit variance $\mathbb{E}[r_i^2] = 1$. ($\mathbb{E}$ is the expectation operator.)

Then the *autocorrelations* of r_i are equal to: $\rho_k = \mathbb{E}[r_i r_{i-k}]$.

If we multiply the *autoregressive* process *AR(n)*: $r_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$, by r_{i-k} and take the expectations, then we obtain the Yule-Walker equations:

$$\begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \vdots \\ \rho_p \end{pmatrix} = \begin{pmatrix} 1 & \rho_1 & \dots & \rho_{n-1} \\ \rho_1 & 1 & \dots & \rho_{n-2} \\ \rho_2 & \rho_1 & \dots & \rho_{n-3} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n-1} & \rho_{n-2} & \dots & 1 \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \vdots \\ \varphi_n \end{pmatrix}$$

The Yule-Walker equations relate the *autocorrelation coefficients* ρ_i with the coefficients of the *AR(n)* process φ_i.

The Yule-Walker equations can be solved for the *AR(n)* coefficients φ_i using matrix inversion.

```
> # Compute autocorrelation coefficients
> ac_f <- acf(ari_ma, lag=10, plot=FALSE)
> ac_f <- drop(ac_f$acf)
> acf1 <- ac_f[-NROW(ac_f)]
> # Define Yule-Walker matrix
> yule_walker <- sapply(2:9, function(lagg) {
+   c(acf1[lagg:1], acf1[2:(NROW(acf1)-lagg+1)])
+ })  # end sapply
> yule_walker <- cbind(acf1, yule_walker, rev(acf1))
> # Generalized inverse of Yule-Walker matrix
> yule_walker_inv <- MASS::ginv(yule_walker)
> # Solve Yule-Walker equations
> coeff_yw <- drop(yule_walker_inv %*% ac_f[-1])
> round(coeff_yw, 5)
> coeff_fit
```

Forecasting Autoregressive Processes

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

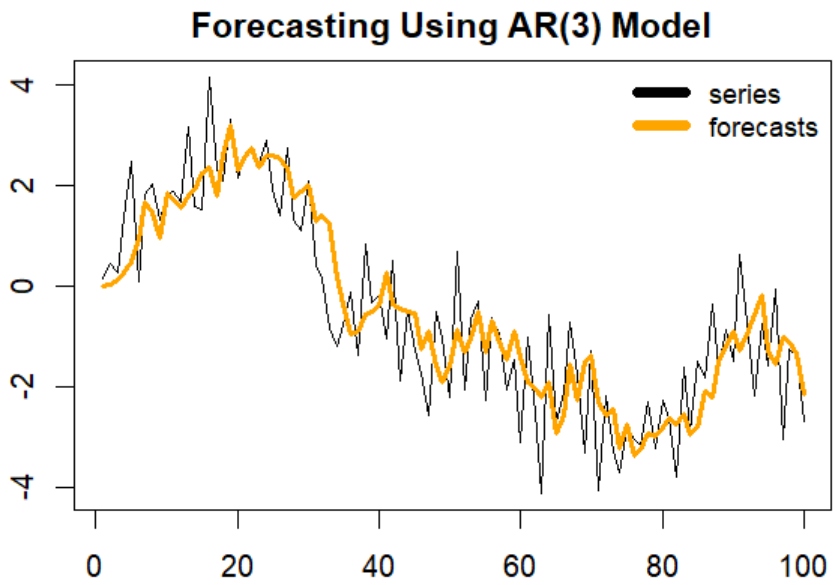
$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

Can be simulated using the function `filter()` with the argument `method="recursive"`.

Filtering can be performed even faster by directly calling the compiled C++ function `stats:::C_rfilter()`.

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

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




```
> n_rows <- 1e2
> co_eff <- c(0.1, 0.39, 0.5); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- rnorm(n_rows)
> # Simulate AR process using filter()
> ari_ma <- filter(x=in_nov, filter=co_eff, method="recursive")
> ari_ma <- as.numeric(ari_ma)
> # Simulate AR process using C_rfilter()
> arima_fast <- .Call(stats:::C_rfilter, in_nov, co_eff,
+   double(n_rows + n_coeff))
> all.equal(ari_ma, arima_fast[-(1:n_coeff)],
+   check.attributes=FALSE)
```

```
> # Forecast AR(3) process using loop in R
> forecast_s <- numeric(NROW(ari_ma)+1)
> forecast_s[1] <- 0
> forecast_s[2] <- co_eff[1]*ari_ma[1]
> forecast_s[3] <- co_eff[1]*ari_ma[2] + co_eff[2]*ari_ma[1]
> for (it in 4:NROW(forecast_s)) {
+   forecast_s[it] <- ari_ma[(it-1):(it-3)] %*% co_eff
+ }  # end for
> # Plot with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> plot(ari_ma, main="Forecasting Using AR(3) Model",
+   xlab="", ylab="", type="l")
> lines(forecast_s, col="orange", lwd=3)
> legend(x="topright", legend=c("series", "forecasts"),
+  col=c("black", "orange"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Fast Forecasting of Autoregressive Processes

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

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

The above *convolution* can be quickly calculated by using the function `filter()` with the argument `method="convolution"`.

The convolution can be calculated even faster by directly calling the compiled `C++` function `stats:::C_cfilter()`.

The forecasts can also be calculated using the design matrix multiplied by the *AR(n)* coefficients.

```
> # Forecast using filter()
> filter_fast <- filter(x=ari_ma, sides=1,
+   filter=co_eff, method="convolution")
> filter_fast <- as.numeric(filter_fast)
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Filter using C_cfilter() compiled C++ function directly
> filter_fast <- .Call(stats:::C_cfilter, ari_ma, filter=co_eff,
+                sides=1, circular=FALSE)
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Filter using HighFreq::roll_conv() Rcpp function
> filter_fast <- HighFreq::roll_conv(matrix(ari_ma), matrix(co_eff))
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Define predictor matrix for forecasting
> predic_tor <- sapply(0:(n_coeff-1), function(lagg) {
+   rutils::lag_it(ari_ma, lagg=lagg)
+ })  # end sapply
> # Forecast using predictor matrix
> filter_fast <- c(0, drop(predic_tor %*% co_eff))
> # Compare with loop in R
> all.equal(forecast_s, filter_fast, check.attributes=FALSE)
```

Forecasting Using predict.Arima()

The forecasts of the *AR(n)* process can also be calculated using the function `predict()`.

The function `predict()` is a *generic function* for forecasting based on a given model.

The *method* `predict.Arima()` is *dispatched* by `R` for calculating predictions from *ARIMA* models produced by the function `stats::arima()`.

The *method* `predict.Arima()` returns a prediction object which is a `list` containing the predicted value and its standard error.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series, using the *maximum likelihood* method (which may give slightly different coefficients than the linear regression model).

```
> # Fit ARIMA model using arima()
> arima_fit <- arima(ari_ma, order=c(3,0,0), include.mean=FALSE)
> arima_fit$coef
> co_eff
> # One-step-ahead forecast using predict.Arima()
> pre_dict <- predict(arima_fit, n.ahead=1)
> # Or directly call predict.Arima()
> # pre_dict <- predict.Arima(arima_fit, n.ahead=1)
> # Inspect the prediction object
> class(pre_dict)
> names(pre_dict)
> class(pre_dict$pred)
> unlist(pre_dict)
> # One-step-ahead forecast using matrix algebra
> fore_cast <- drop(ari_ma[n_rows:(n_rows-2)] %*% arima_fit$coef)
> # Compare one-step-ahead forecasts
> all.equal(pre_dict$pred[[1]], fore_cast)
> # Get information about predict.Arima()
> ?stats:::predict.Arima
```

The Forecasting Residuals

The *forecasting residuals* ε_i are equal to the differences between the actual values r_i minus their *forecasts* f_i: $\varepsilon_i = r_i - f_i$.

Accurate forecasting of an *AR(n)* process requires knowing its coefficients.

If the coefficients of the *AR(n)* process are known exactly, then its *in-sample residuals* ε_i are equal to its *innovations* ξ_i: $\varepsilon_i = r_i - f_i = \xi_i$.

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series.

If the *AR(n)* coefficients are fitted to the empirical time series, then its *residuals* are *not* equal to its *innovations*.

```
> # Calculate the in-sample forecasting residuals
> residual_s <- (ari_ma - forecast_s[-NROW(forecast_s)])
> # Compare residuals with innovations
> all.equal(in_nov, residual_s, check.attributes=FALSE)
> plot(residual_s, t="l", lwd=3, xlab="", ylab="",
+      main="ARIMA Forecast Errors")
```

Fitting and Forecasting Autoregressive Models

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series first, before forecasting.

Forecasting using an autoregressive model is performed by first fitting an *AR(n)* model to past data, and calculating its coefficients.

The fitted coefficients are then applied to calculating the *out-of-sample* forecasts.

The model fitting procedure depends on two unknown *meta-parameters*: the order n of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define AR process parameters
> n_rows <- 1e3
> co_eff <- c(0.5, 0.0, 0.0); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- rnorm(n_rows)
> # Simulate AR process using C_rfilter()
> ari_ma <- .Call(stats:::C_rfilter, in_nov, co_eff,
+   double(n_rows + n_coeff))[-(1:n_coeff)]
> # Define order of the AR(n) forecasting model
> or_der <- 5
> # Define predictor matrix for forecasting
> de_sign <- sapply(1:or_der, rutils::lag_it, in_put=ari_ma)
> colnames(de_sign) <- paste0("pred_", 1:NCOL(de_sign))
> # Add response equal to series
> de_sign <- cbind(ari_ma, de_sign)
> colnames(de_sign)[1] <- "response"
> # Specify length of look-back interval
> look_back <- 100
> # Invert the predictor matrix
> rang_e <- (n_rows-look_back):(n_rows-1)
> design_inv <- MASS::ginv(de_sign[rang_e, -1])
> # Calculate fitted coefficients
> coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
> # Calculate forecast
> drop(de_sign[n_rows, -1] %*% coeff_fit)
```

Backtesting Autoregressive Forecasting Models

Backtesting is the simulation of a model on historical data to test its forecasting accuracy.

The autoregressive forecasting model can be *backtested* by calculating forecasts over either a *rolling* or an *expanding* look-back interval.

If the start date is fixed at the first row then the look-back interval is *expanding*.

The coefficients of the *AR(n)* process are fitted to past data, and then applied to calculating out-of-sample forecasts.

The *backtesting* procedure allows determining the optimal *meta-parameters* of the forecasting model: the order *n* of the *AR(n)* model and the length of look-back interval (`look_back`).

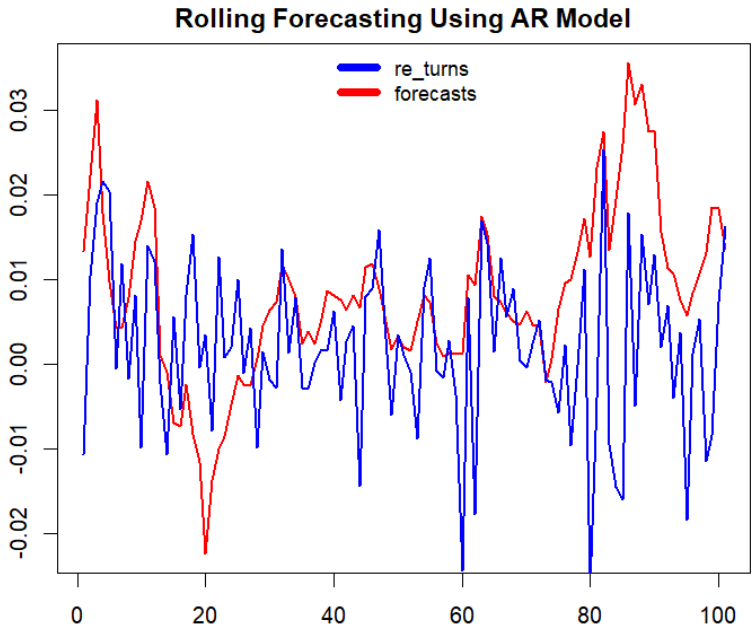
```
> # Calculate a vector of daily VTI log returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(re_turns)
> re_turns <- as.numeric(re_turns)
> n_rows <- NROW(re_turns)
> # Define predictor as a rolling sum
> n_agg <- 5
> predic_tor <- rutils::roll_sum(re_turns, look_back=n_agg)
> # Shift the res_ponse forward out-of-sample
> res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Define de_sign matrix
> de_sign <- cbind(res_ponse, predic_tor)
> # Perform rolling forecasting
> look_back <- 100
> forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+   # Define rolling look-back range
+   start_p <- max(1, end_p-look_back)
+   # Or expanding look-back range
+   # start_p <- 1
+   rang_e <- start_p:(end_p-1)
+   # Invert the predictor matrix
+   design_inv <- MASS::ginv(de_sign[rang_e, -1])
+   # Calculate fitted coefficients
+   coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+   # Calculate forecast
+   drop(de_sign[end_p, -1] %*% coeff_fit)
+ })  # end sapply
> # Add warmup period
> forecast_s <- c(rep(0, look_back), forecast_s)
```

The Accuracy of the Autoregressive Forecasting Model

The accuracy of a forecasting model can be measured using the *mean squared error* and the *correlation*.

The mean squared error (*MSE*) of a forecasting model is the average of the squared forecasting residuals ε_i, equal to the differences between the actual values r_i minus the *forecasts* f_i: $\varepsilon_i = r_i - f_i$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(r_i - f_i)^2$$




```
> # Mean squared error
> mean((re_turns - forecast_s)^2)
> # Correlation
> cor(forecast_s, re_turns)
> # Plot forecasting series with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> plot(forecast_s[(n_rows-look_back):n_rows], col="red",
+      xlab="", ylab="", type="l", lwd=2,
+      main="Rolling Forecasting Using AR Model")
> lines(re_turns[(n_rows-look_back):n_rows], col="blue", lwd=2)
> legend(x="top", legend=c("re_turns", "forecasts"),
+  col=c("blue", "red"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Backtesting Function for the Forecasting Model

The *meta-parameters* of the *backtesting* function are the order *n* of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define backtesting function
> sim_forecasts <- function(res_ponse, predic_tor=res_ponse, n_agg=5
+                 or_der=5, look_back=100) {
+   n_rows <- NROW(res_ponse)
+   # Define predictor as a rolling sum
+   predic_tor <- rutils::roll_sum(res_ponse, look_back=n_agg)
+   # Shift the res_ponse forward out-of-sample
+   res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
+   # Define predictor matrix for forecasting
+   predic_tor <- sapply(1+n_agg*(0:or_der), rutils::lag_it,
+                  in_put=predic_tor)
+   predic_tor <- cbind(rep(1, n_rows), predic_tor)
+   # Define de_sign matrix
+   de_sign <- cbind(res_ponse, predic_tor)
+   # Perform rolling forecasting
+   forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+     # Define rolling look-back range
+     start_p <- max(1, end_p-look_back)
+     # Or expanding look-back range
+     # start_p <- 1
+     rang_e <- start_p:(end_p-1)
+     # Invert the predictor matrix
+     design_inv <- MASS::ginv(de_sign[rang_e, -1])
+     # Calculate fitted coefficients
+     coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+     # Calculate forecast
+     drop(de_sign[end_p, -1] %*% coeff_fit)
+   })  # end sapply
+   # Add warmup period
+   forecast_s <- c(rep(0, look_back), forecast_s)
+   rutils::roll_sum(forecast_s, look_back=n_agg)
+ }  # end sim_forecasts
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(re_turns, or_der=5, look_back=100)
> c(mse=mean((re_turns - forecast_s)^2), cor=cor(re_turns, forecast_
```

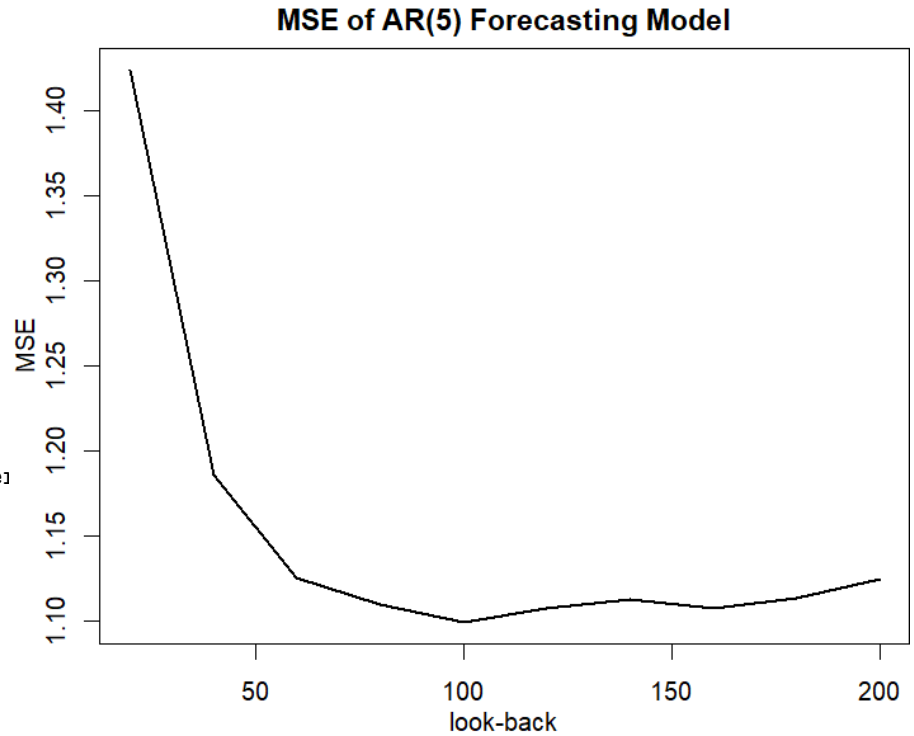
The Optimal Parameters of the Forecasting Model

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order n of the $AR(n)$ model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

```
> look_backs <- seq(20, 200, 20)
> back_tests <- sapply(look_backs, back_test, se_ries=ari_ma, or_de
> back_tests <- t(back_tests)
> rownames(back_tests) <- look_backs
> # Plot forecasting series with legend
> plot(x=look_backs, y=back_tests[, 1],
+   xlab="look-back", ylab="MSE", type="l", lwd=2,
+   main="MSE of AR(5) Forecasting Model")
```



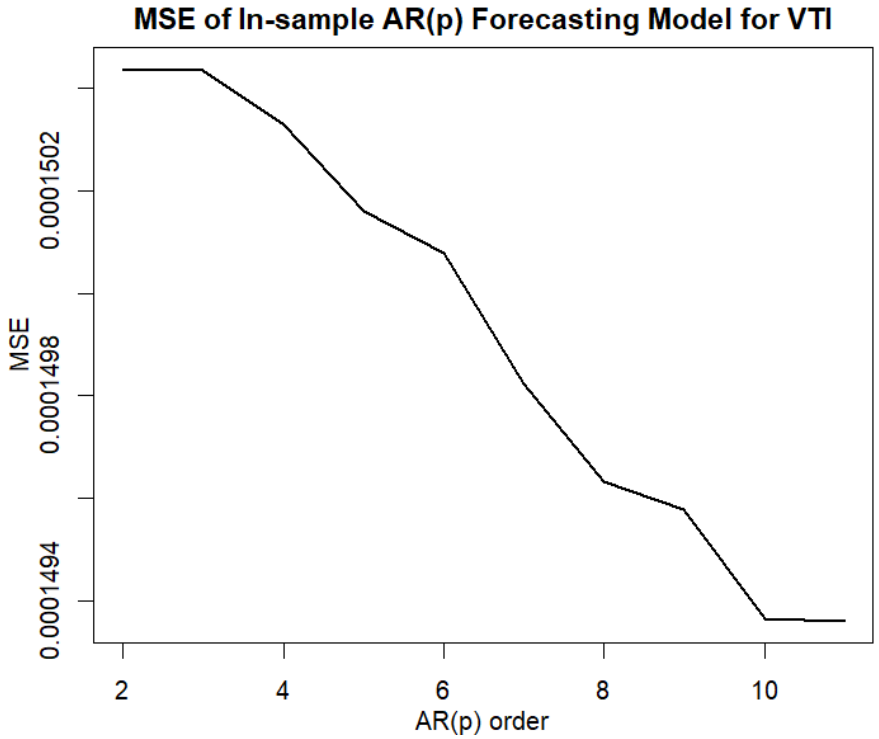
In-sample Forecasting Using Autoregressive Models

In-sample forecasting consists of first fitting an *AR(n)* model to the data, and calculating its coefficients.

The *in-sample* forecasts are calculated by multiplying the response vector of returns by the fitted coefficients.

The mean squared errors (*MSE*) of the *in-sample* forecasts decrease steadily with the increasing order parameter *n* of the *AR(n)* model.

```
> # Calculate a vector of daily VTI log returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(vt_i)
> vt_i <- as.numeric(vt_i)
> n_rows <- NROW(vt_i)
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1:order_max, rutils::lag_it, in_put=vt_i)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> colnames(predic_tor) <- paste0("pred_", 1:NCOL(predic_tor))
> res_ponse <- vt_i
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   # Calculate fitted coefficients
+   in_verse <- MASS::ginv(predic_tor[, 1:or_der])
+   coeff_fit <- drop(in_verse %*% res_ponse)
+   # Calculate in-sample forecasts of vt_i
+   drop(predic_tor[, 1:or_der] %*% coeff_fit)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```




```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- names(forecast_s)
> # Plot forecasting MSE
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> plot(x=2:NCOL(predic_tor), y=ms_e[, 1],
+   xlab="AR(n) order", ylab="MSE", type="l", lwd=2,
+   main="MSE of In-sample AR(n) Forecasting Model for VTI")
```

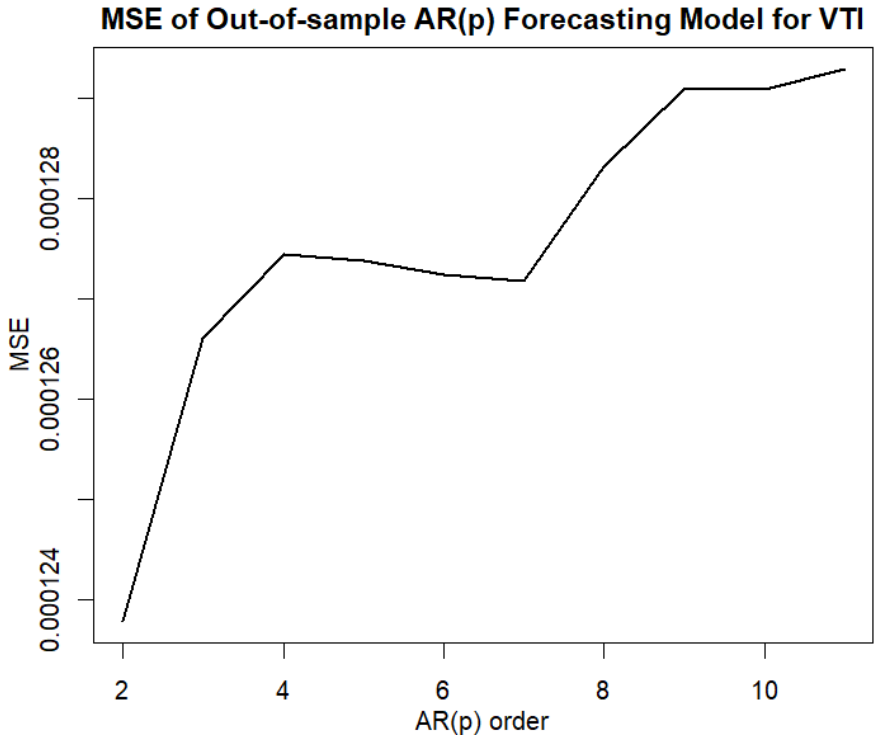
Out-of-sample Forecasting Using Autoregressive Models

Out-of-sample forecasting consists of first fitting an *AR(n)* model to the training data, and calculating its coefficients.

The *out-of-sample* forecasts are calculated by multiplying the *out-of-sample* response vector of returns by the fitted coefficients.

The mean squared errors (*MSE*) of the *out-of-sample* forecasts increase steadily with the increasing order parameter *n* of the *AR(n)* model.

The reason for the increasing out-of-sample MSE is *overfitting* of the coefficients to the training data.




```
> in_sample <- 1:(n_rows %/% 2)
> out_sample <- (n_rows %/% 2 + 1):n_rows
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   # Calculate fitted coefficients
+   in_verse <- MASS::ginv(predic_tor[in_sample, 1:or_der])
+   coeff_fit <- drop(in_verse %*% res_ponse[in_sample])
+   # Calculate out-of-sample forecasts of vt_i
+   drop(predic_tor[out_sample, 1:or_der] %*% coeff_fit)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i[out_sample] - x)^2), cor=cor(vt_i[out_sample],
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- names(forecast_s)
> # Plot forecasting MSE
> plot(x=2:NCOL(predic_tor), y=ms_e[, 1],
+   xlab="AR(n) order", ylab="MSE", type="l", lwd=2,
+   main="MSE of Out-of-sample AR(n) Forecasting Model for VTI")
```

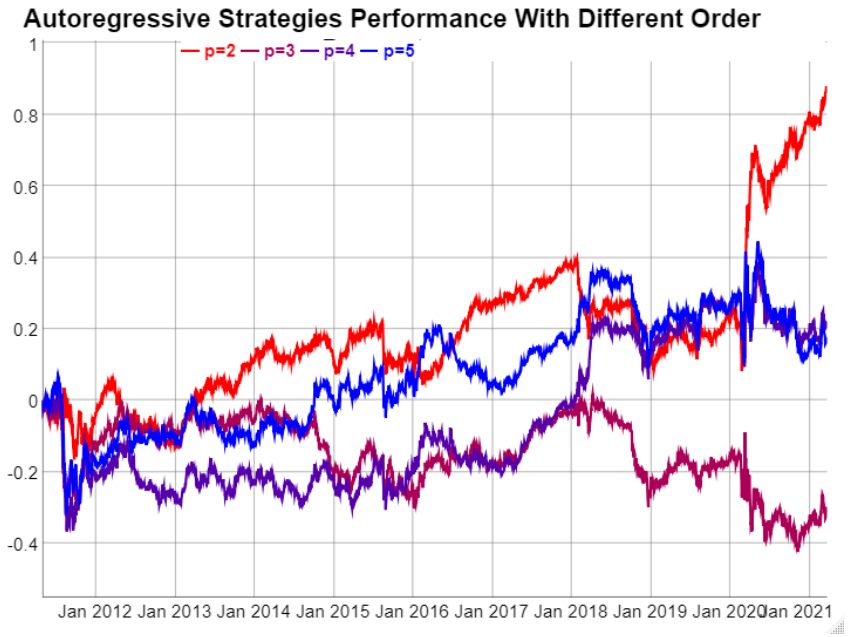
Autoregressive Strategy Out-of-sample Performance

The autoregressive strategy invests a dollar amount of *VTI* equal to the sign of the forecasts.

The performance of the autoregressive strategy is better with a smaller order parameter *n* of the *AR(n)* model.

Decreasing the order parameter of the autoregressive model is a form of *shrinkage* because it reduces the number of predictive variables.

```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
```




```
> # Plot dygraph of out-of-sample PnLs
> color_s <- colorRampPalette(c("red", "blue"))(NCOL(pnl_s[, 1:4]))
> col_names <- colnames(pnl_s[, 1:4])
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance With Different Order
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

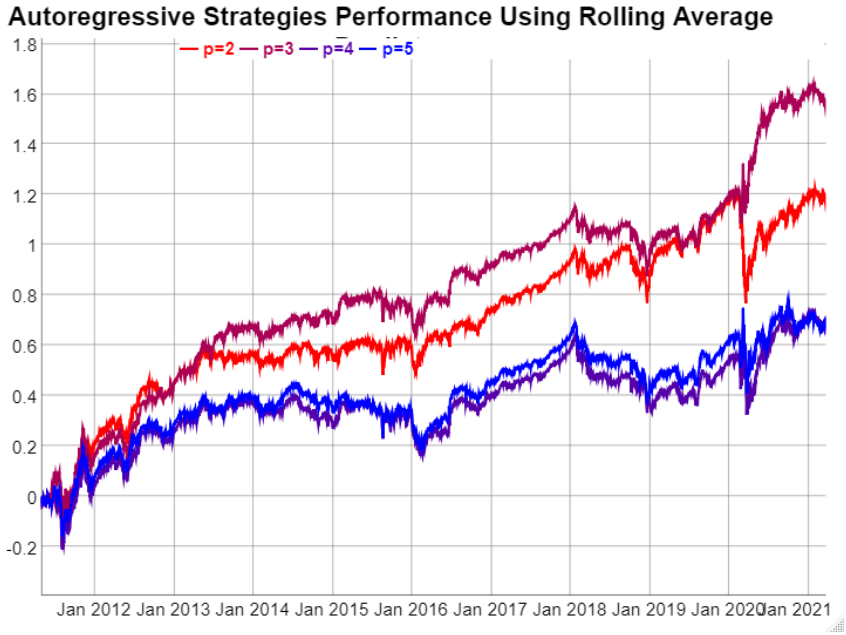
Autoregressive Strategy Using Rolling Average Returns

The *out-of-sample* forecasts can be improved by using the rolling average of the returns as a predictor.

This is because the average of returns has a lower *variance*.

But the average also has a higher *bias* because it includes returns that may be unrelated to the present.

Using the rolling average of returns as a predictor reduces the forecast variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).




```
> # Define predictor as a rolling mean
> n_agg <- 5
> predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
> # Shift the res_ponse forward out-of-sample
> res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
> # Define predictor matrix for forecasting
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   in_verse <- MASS::ginv(predic_tor[in_sample, 1:or_der])
+   coeff_fit <- drop(in_verse %*% res_ponse[in_sample])
+   drop(predic_tor[out_sample, 1:or_der] %*% coeff_fit)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```

```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
> # Plot dygraph of out-of-sample PnLs
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance Using Rolling Averag
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

Autoregressive Strategy Using Rolling Average Forecasts

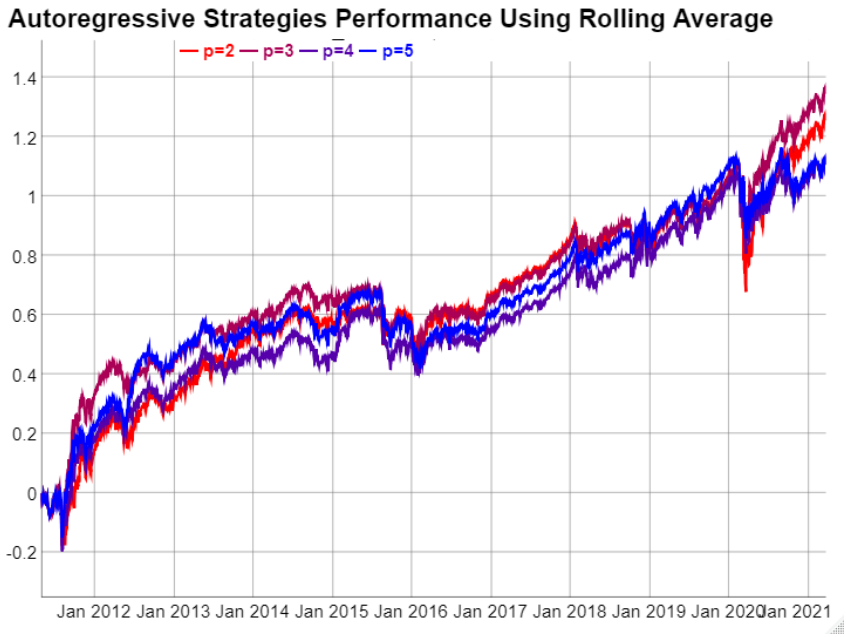
The *out-of-sample* forecasts can be further improved by using the average of past forecasts.

This is because the average of forecasts has a lower *variance*.

But the average also has a higher *bias* because it includes past forecasts that may be unrelated to the present.

Using the rolling average of past forecasts reduces the forecast variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).

```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   x <- roll::roll_mean(x, width=n_agg, min_obs=1)
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
```




```
> # Plot dygraph of out-of-sample PnLs
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance Using Rolling Averag
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

Backtesting Autoregressive Forecasting Models

Backtesting is the simulation of a model on historical data to test its forecasting accuracy.

The autoregressive forecasting model can be *backtested* by calculating forecasts over either a *rolling* or an *expanding* look-back interval.

If the start date is fixed at the first row then the look-back interval is *expanding*.

The coefficients of the *AR(n)* process are fitted to past data, and then applied to calculating out-of-sample forecasts.

The *backtesting* procedure allows determining the optimal *meta-parameters* of the forecasting model: the order *n* of the *AR(n)* model and the length of look-back interval (`look_back`).

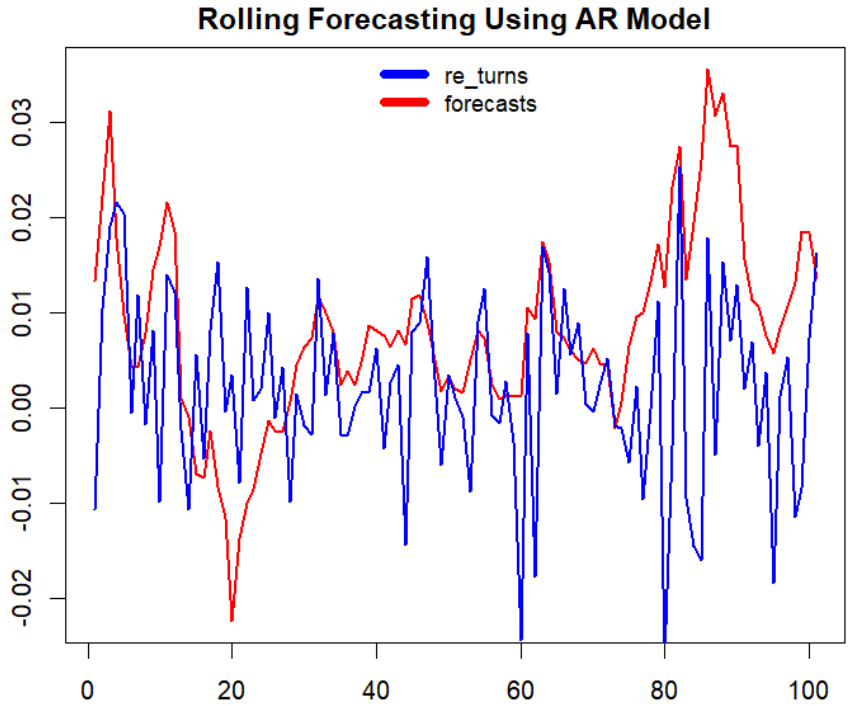
```
> # Calculate a vector of daily VTI log returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(vt_i)
> vt_i <- as.numeric(vt_i)
> n_rows <- NROW(vt_i)
> # Define predictor as a rolling mean
> n_agg <- 5
> predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
> # Shift the res_ponse forward out-of-sample
> res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Define de_sign matrix
> de_sign <- cbind(res_ponse, predic_tor)
> # Perform rolling forecasting
> look_back <- 100
> forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+   # Define rolling look-back range
+   start_p <- max(1, end_p-look_back)
+   # Or expanding look-back range
+   # start_p <- 1
+   rang_e <- start_p:(end_p-1)
+   # Invert the predictor matrix
+   design_inv <- MASS::ginv(de_sign[rang_e, -1])
+   # Calculate fitted coefficients
+   coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+   # Calculate forecast
+   drop(de_sign[end_p, -1] %*% coeff_fit)
+ })  # end sapply
> # Add warmup period
> forecast_s <- c(rep(0, look_back), forecast_s)
```

The Accuracy of the Autoregressive Forecasting Model

The accuracy of a forecasting model can be measured using the *mean squared error* and the *correlation*.

The mean squared error (*MSE*) of a forecasting model is the average of the squared forecasting residuals ε_i, equal to the differences between the actual values r_i minus the *forecasts* f_i: $\varepsilon_i = r_i - f_i$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(r_i - f_i)^2$$




```
> # Mean squared error
> mean((vt_i - forecast_s)^2)
> # Correlation
> cor(forecast_s, vt_i)
> # Plot forecasting series with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> plot(forecast_s[(n_rows-look_back):n_rows], col="red",
+      xlab="", ylab="", type="l", lwd=2,
+      main="Rolling Forecasting Using AR Model")
> lines(vt_i[(n_rows-look_back):n_rows], col="blue", lwd=2)
> legend(x="top", legend=c("VTI returns", "forecasts"),
+  col=c("blue", "red"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Backtesting Function for the *AR* Forecasting Model

The *meta-parameters* of the *backtesting* function are the order *n* of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define backtesting function
> sim_forecasts <- function(res_ponse, predic_tor=res_ponse, n_agg=5
+                 or_der=5, look_back=100) {
+   n_rows <- NROW(res_ponse)
+   # Define predictor as a rolling mean
+   predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
+   # Shift the res_ponse forward out-of-sample
+   res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
+   # Define predictor matrix for forecasting
+   predic_tor <- sapply(1+n_agg*(0:or_der), rutils::lag_it,
+                  in_put=predic_tor)
+   predic_tor <- cbind(rep(1, n_rows), predic_tor)
+   # Define de_sign matrix
+   de_sign <- cbind(res_ponse, predic_tor)
+   # Perform rolling forecasting
+   forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+     # Define rolling look-back range
+     start_p <- max(1, end_p-look_back)
+     # Or expanding look-back range
+     # start_p <- 1
+     rang_e <- start_p:(end_p-1)
+     # Invert the predictor matrix
+     design_inv <- MASS::ginv(de_sign[rang_e, -1])
+     # Calculate fitted coefficients
+     coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+     # Calculate forecast
+     drop(de_sign[end_p, -1] %*% coeff_fit)
+   })  # end sapply
+   # Add warmup period
+   forecast_s <- c(rep(0, look_back), forecast_s)
+   roll::roll_mean(forecast_s, width=n_agg, min_obs=1)
+ }  # end sim_forecasts
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(vt_i, or_der=5, look_back=100)
> c(mse=mean((vt_i - forecast_s)^2), cor=cor(vt_i, forecast_s))
```

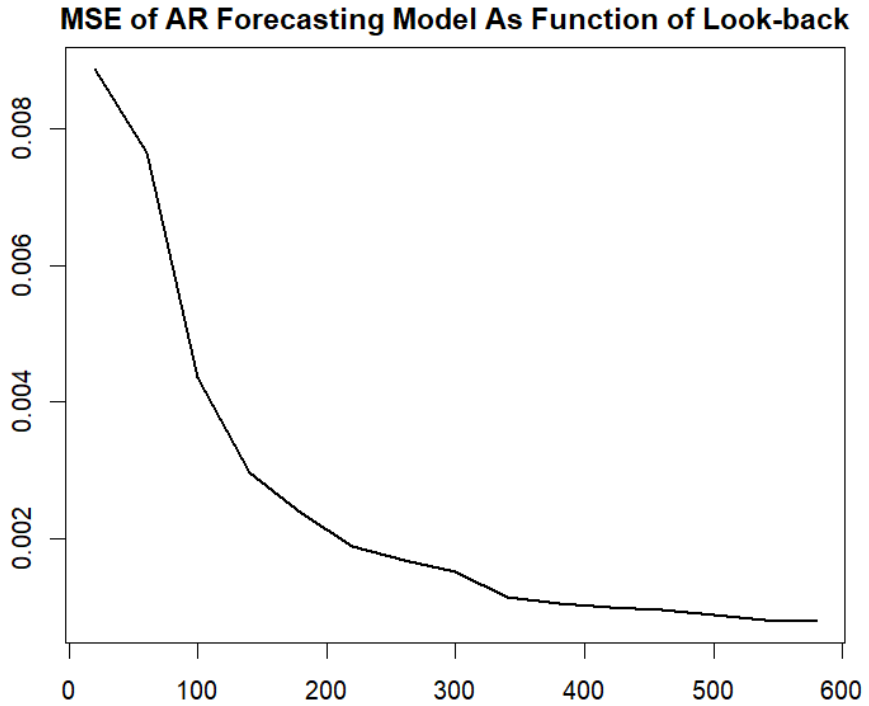
The Dependence On the Look-back Interval

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

The accuracy of the forecasting model increases steadily with longer look-back intervals (`look_back`), because more data improves the estimates of the autoregressive coefficients.




```
> look_backs <- seq(20, 600, 40)
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # clusterExport(clus_ter, c("star_t", "bar_rier"))
> # Perform parallel loop under Windows
> forecast_s <- parLapply(clus_ter, look_backs, sim_forecasts, res.
+                   predic_tor=vt_i, n_agg=5, or_der=5)
> # Perform parallel bootstrap under Mac-OSX or Linux
> forecast_s <- mclapply(look_backs, sim_forecasts, res_ponse=vt_i
+   predic_tor=vt_i, n_agg=5, or_der=5, mc.cores=n_cores)
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- look_backs
> # Select optimal look_back interval
> look_back <- look_backs[which.min(ms_e[, 1])]
> # Plot forecasting MSE
> plot(x=look_backs, y=ms_e[, 1],
+   xlab="look-back", ylab="MSE", type="l", lwd=2,
+   main="MSE of AR Forecasting Model As Function of Look-back")
```

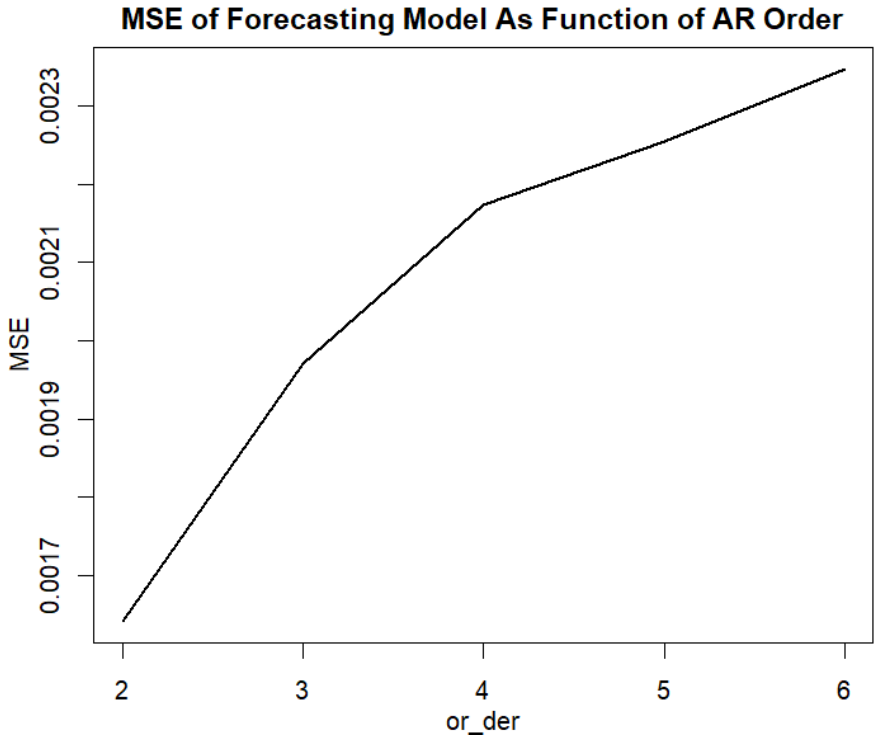
The Dependence On the Order Parameter

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

The accuracy of the forecasting model increases steadily with longer look-back intervals (`look_back`), because more data improves the estimates of the autoregressive coefficients.




```
> order_s <- 2:6
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # clusterExport(clus_ter, c("star_t", "bar_rier"))
> # Perform parallel loop under Windows
> forecast_s <- parLapply(clus_ter, order_s, sim_forecasts, res_por
+                   predic_tor=vt_i, n_agg=5, look_back=look_back)
> stopCluster(clus_ter)  # Stop R processes over cluster under Winc
> # Perform parallel bootstrap under Mac-OSX or Linux
> forecast_s <- mclapply(order_s, sim_forecasts, res_ponse=vt_i,
+   predic_tor=vt_i, n_agg=5, look_back=look_back, mc.cores=n_cores
> stopCluster(clus_ter)  # Stop R processes over cluster under Winc
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- order_s
> # Select optimal order parameter
> or_der <- order_s[which.min(ms_e[, 1])]
> # Plot forecasting MSE
> plot(x=order_s, y=ms_e[, 1],
+   xlab="or_der", ylab="MSE", type="l", lwd=2,
+   main="MSE of Forecasting Model As Function of AR Order")
```

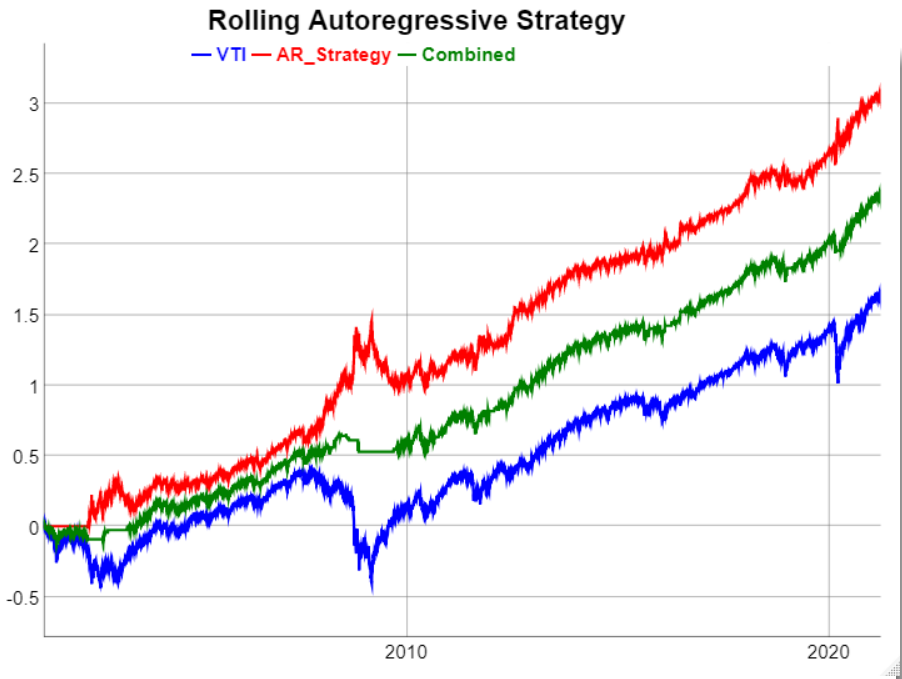
Performance of the Rolling Autoregressive Strategy

The return forecasts are calculated just before the close of the markets, so that trades can be executed before the close.

The autoregressive strategy is dominated by a few periods with very large returns, without producing profits for the remaining periods.

Using the return forecasts as portfolio weights produces very large weights in periods of high volatility, and creates excessive risk.

To reduce excessive risk, a binary strategy uses portfolio weights equaly to the sign of the forecasts.




```
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(vt_i, or_der=or_der, look_back=look_ba
> # Calculate strategy PnLs
> pnl_s <- sign(forecast_s)*vt_i
> pnl_s <- cbind(vt_i, pnl_s, (vt_i+pnl_s)/2)
> colnames(pnl_s) <- c("VTI", "AR_Strategy", "Combined")
> cor(pnl_s)
> # Annualized Sharpe ratios of VTI and AR strategy
> sqrt(252)*apply(pnl_s, 2, function (x) mean(x)/sd(x))
> pnl_s <- xts::xts(pnl_s, date_s)
> pnl_s <- cumsum(pnl_s)
```

```
> # Plot the cumulative strategy PnLs
> dygraphs::dygraph(pnl_s, main="Rolling Autoregressive Strategy") %
+   dyOptions(colors=c("blue","red","green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

draft: The Dependence On the Order Parameter

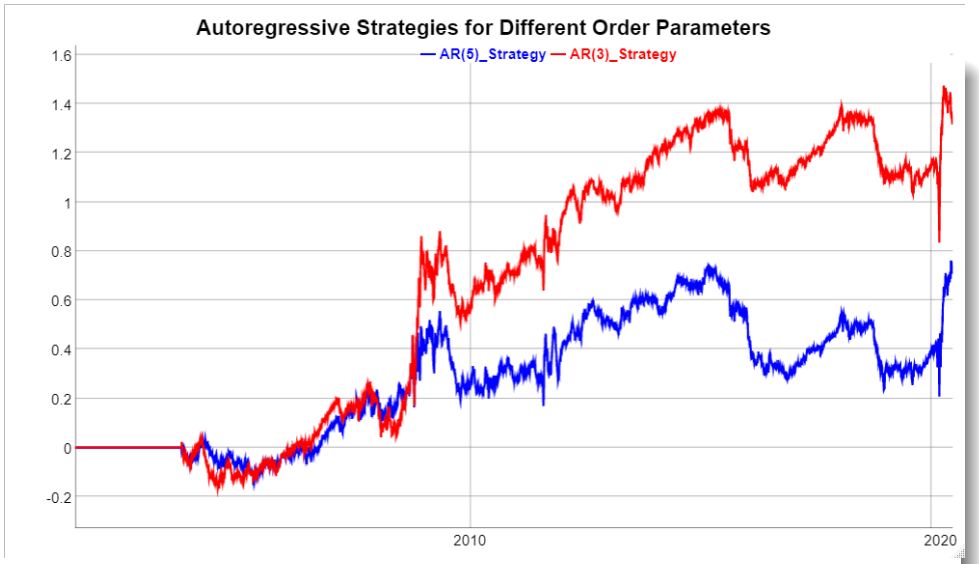
The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model.

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

Longer look-back intervals (`look_back`) are usually better for the autoregressive forecasting model.

The return forecasts are calculated just before the close of the markets, so that trades can be executed before the close.

The autoregressive strategy is dominated by a few periods with very large returns, without producing profits for the remaining periods.




```
> # Calculate PnLs for or_der=5
> forecast_s <- sim_forecasts(vt_i, or_der=5, look_back=look_back)
> pnls_5 <- cumsum(sign(forecast_s)*vt_i)
> # Calculate PnLs for or_der=3
> forecast_s <- sim_forecasts(vt_i, or_der=3, look_back=look_back)
> pnls_3 <- cumsum(sign(forecast_s)*vt_i)
```

```
> # Plot the cumulative strategy returns
> weal_th <- cbind(pnls_5, pnls_3)
> weal_th <- xts::xts(weal_th, date_s)
> col_names <- c("AR(5)_Strategy", "AR(3)_Strategy")
> colnames(weal_th) <- col_names
> dygraphs::dygraph(weal_th, main="Autoregressive Strategies for Di
+   dySeries(name=col_names[1], label=col_names[1], col="blue", stro
+   dySeries(name=col_names[2], label=col_names[2], col="red", strok
+   dyLegend(width=500)
```

Homework Assignment

Required

Study all the lecture slides in `FRE7241_Lecture_5.pdf`, and run all the code in `FRE7241_Lecture_5.R`

Recommended

- Read about *optimization methods*:
  Bolker Optimization Methods.pdf
  Yollin Optimization.pdf
  Boudt DEoptim Large Portfolio Optimization.pdf
- Read about *PCA* in:
  pca-handout.pdf
  pcaTutorial.pdf