FRE7241 Algorithmic Portfolio Management

Lecture#7, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 26, 2021

NYU | TANDON SCHOOL OF ENGINEERING

Momentum Strategy for an *ETF* Portfolio

Momentum strategies can be *backtested* by specifying the portfolio rebalancing frequency, the formation period, and the holding period:

- Specify a portfolio of *ETFs*, stocks, or other assets, and a time series of their returns,
- Specify *end points* for the portfolio rebalancing frequency,
- Specify *look-back* intervals for portfolio formation, and *look-forward* intervals for portfolio holding,
- Specify a performance function to calculate the past performance of the assets,
- Calculate the past performance over the *look-back* formation intervals,
- Calculate the portfolio weights from the past performance,
- Calculate the future returns over the *look-forward* holding intervals,
- Calculate the out-of-sample momentum strategy returns by applying the portfolio weights to the future returns,
- Calculate the transaction costs and subtract them from the strategy returns.

```
> # Extract ETF returns
> sym_bols <- c("VTI", "IEF", "DBC")
> re_turns <- rutils::etf_env$re_turns[, sym_bols]
> re_turns <- na.omit(re_turns)
> # Or, select rows with IEF data
> # re_turns <- re_turns[index(rutils::etf_env$IEF)]
> # Copy over NA values
> # re_turns[1, is.na(re_turns[1, ])] <- 0
> # re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
```

Look-back and Look-forward Intervals

Performance aggregations are be calculated over a vector of overlapping in-sample *look-back* intervals attached at *end points*.

For example, aggregations at monthly *end points* over overlapping 12-month *look-back* intervals.

An example of a data aggregation are the cumulative past returns at each *end point*.

The variable `look_back` is equal to the number of *end points* in the *look-back* interval.

The *start points* are the *end points* lagged by the length of the *look-back* interval.

The *look-back* intervals are spanned by the vectors of *start points* and *end points*.

Performance aggregations are also be calculated over non-overlapping out-of-sample *look-forward* intervals.

The *look-forward* intervals should not overlap with the *look-back* intervals, in order to avoid data snooping.

```
> # Define end of month end points
> end_p <- rutils::calc_endpoints(re_turns, inter_val="months")
> end_p <- end_p[-1]
> n_rows <- NROW(end_p)
> date_s <- zoo::index(re_turns)[end_p]
> # Start points equal end points lagged by 12-month look-back inter
> look_back <- 12
> start_p <- c(rep_len(1, look_back-1),
+   end_p[1:(n_rows - look_back + 1)])
> # Calculate matrix of look-back intervals
> look_backs <- cbind(start_p, end_p)
> colnames(look_backs) <- c("start", "end")
> # Calculate matrix of look-forward intervals
> look_fwds <- cbind(end_p + 1, rutils::lag_it(end_p, -1))
> look_fwds[n_rows, ] <- end_p[n_rows]
> colnames(look_fwds) <- c("start", "end")
> # Inspect the intervals
> head(cbind(look_backs, look_fwds))
> tail(cbind(look_backs, look_fwds))
```

Momentum Portfolio Weights

The portfolio weights of *momentum* strategies are calculated based on the past performance of the assets.

The weights are scaled to limit the portfolio *leverage* and its market *beta*.

The portfolio weights of *momentum* strategies can be scaled in several different ways.

To limit the portfolio leverage, the weights can be scaled so that the sum of squares is equal to 1: $\sum_{i=1}^{n} w_i^2 = 1$.

The weights can also be de-meaned, so that their sum is equal to zero, to create long-short portfolios with small betas: $\sum_{i=1}^{n} w_i = 0$.

```
> # Define performance function as Sharpe ratio
> perform_ance <- function(re_turns) sum(re_turns)/sd(re_turns)
> # Calculate past performance over look-back intervals
> pas_t <- apply(look_backs, 1, function(ep) {
+   sapply(re_turns[ep[1]:ep[2]], perform_ance)
+ })  # end sapply
> pas_t <- t(pas_t)
> pas_t[is.na(pas_t)] <- 0
> # Weights are proportional to past performance
> weight_s <- pas_t
> # weight_s[weight_s < 0] <- 0
> # Scale weight_s so sum of squares is equal to 1.
> weight_s <- weight_s/sqrt(rowSums(weight_s^2))
> # Or scale weight_s so sum is equal to 1
> # weight_s <- weight_s/rowSums(weight_s)
> # Set NA values to zero
> weight_s[is.na(weight_s)] <- 0
> sum(is.na(weight_s))
```

Backtesting the Momentum Strategy

Backtesting is the testing of the accuracy of a forecasting model using simulation on historical data.

Backtesting is a type of *cross-validation* applied to time series data.

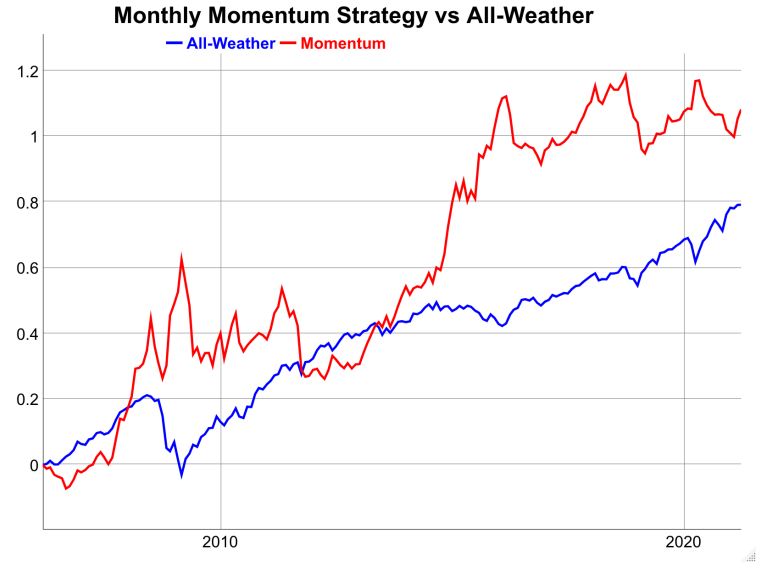
Backtesting is performed by *training* the model on past data and *testing* it on future out-of-sample data.

The *training* data is specified by the *look-back* intervals (`pas_t`), and the model forecasts are applied to the future data defined by the *look-forward* intervals (`fu_ture`).

The out-of-sample *momentum* strategy returns can be calculated by multiplying the `fu_ture` returns by the forecast *ETF* portfolio weights.

The momentum returns are lagged so that they are attached to the end of the future interval, instead of at its beginning.

```
> # Calculate future out-of-sample performance
> fu_ture <- apply(look_fwds, 1, function(ep) {
+   sapply(re_turns[ep[1]:ep[2]], sum)
+ })  # end sapply
> fu_ture <- t(fu_ture)
> fu_ture[is.na(fu_ture)] <- 0
> tail(fu_ture)
```




```
> # Calculate the momentum pnls
> pnl_s <- rowSums(weight_s*fu_ture)
> # Lag the future and momentum returns to proper dates
> fu_ture <- rutils::lag_it(fu_ture)
> pnl_s <- rutils::lag_it(pnl_s)
> # The momentum strategy has low correlation to stocks
> cor(pnl_s, fu_ture)
> # Define all-weather benchmark
> weights_aw <- c(0.30, 0.55, 0.15)
> all_weather <- fu_ture %*% weights_aw
> # Calculate the wealth of momentum returns
> weal_th <- xts::xts(cbind(all_weather, pnl_s), order.by=date_s)
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th), main="Monthly Momentum Strategy
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Backtesting Functional for Momentum Strategy

```
> backtest_momentum <- function(returns,
+                 perform_ance=function(re_turns) (sum(re_turns)/sd(re_turns)),
+                 look_back=12, re_balance="months", bid_offer=0.001,
+                 endp=rutils::calc_endpoints(re_turns, inter_val=re_balance)[-1],
+                 with_weights=FALSE, ...) {
+   stopifnot("package:rutils" %in% search() || require("rutils", quietly=TRUE))
+   # Define look-back and look-forward intervals
+   n_rows <- NROW(end_p)
+   start_p <- c(rep_len(1, look_back-1), end_p[1:(n_rows-look_back+1)])
+   # Calculate look-back intervals
+   look_backs <- cbind(start_p, end_p)
+   # Calculate look-forward intervals
+   look_fwds <- cbind(end_p + 1, rutils::lag_it(end_p, -1))
+   look_fwds[n_rows, ] <- end_p[n_rows]
+   # Calculate past performance over look-back intervals
+   pas_t <- t(apply(look_backs, 1, function(ep) sapply(re_turns[ep[1]:ep[2]], perform_ance)))
+   pas_t[is.na(pas_t)] <- 0
+   # Calculate future performance
+   fu_ture <- t(apply(look_fwds, 1, function(ep) sapply(re_turns[ep[1]:ep[2]], sum)))
+   fu_ture[is.na(fu_ture)] <- 0
+   # Scale weight_s so sum of squares is equal to 1
+   weight_s <- pas_t
+   weight_s <- weight_s/sqrt(rowSums(weight_s^2))
+   weight_s[is.na(weight_s)] <- 0  # Set NA values to zero
+   # Calculate momentum profits and losses
+   pnl_s <- rowSums(weight_s*fu_ture)
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*cumprod(1 + pnl_s)*rowSums(abs(rutils::diff_it(weight_s)))
+   pnl_s <- (pnl_s - cost_s)
+   if (with_weights)
+     rutils::lag_it(cbind(pnl_s, weight_s))
+   else
+     rutils::lag_it(pnl_s)
+ }  # end backtest_momentum
```

Optimization of Momentum Strategy Parameters

The performance of the *momentum* strategy depends on the length of the *look-back interval* used for calculating the past performance.

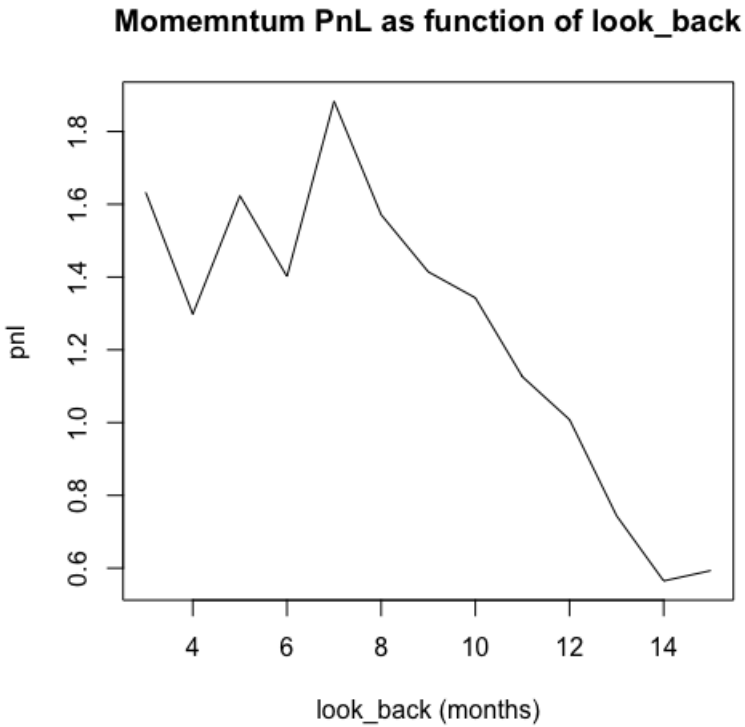
Performing a *backtest* allows finding the optimal *momentum* (trading) strategy parameters, such as the *look-back interval*.

But using a different rebalancing frequency in the *backtest* can produce different values for the optimal trading strategy parameters.

So *backtesting* just redefines the problem of finding (tuning) the optimal trading strategy parameters, into the problem of finding the optimal *backtest* (meta-model) parameters.

But the advantage of using the *backtest* meta-model is that it can reduce the number of parameters that need to be optimized.

Performing many *backtests* on multiple trading strategies risks identifying inherently unprofitable trading strategies as profitable, purely by chance (*p-value hacking*).

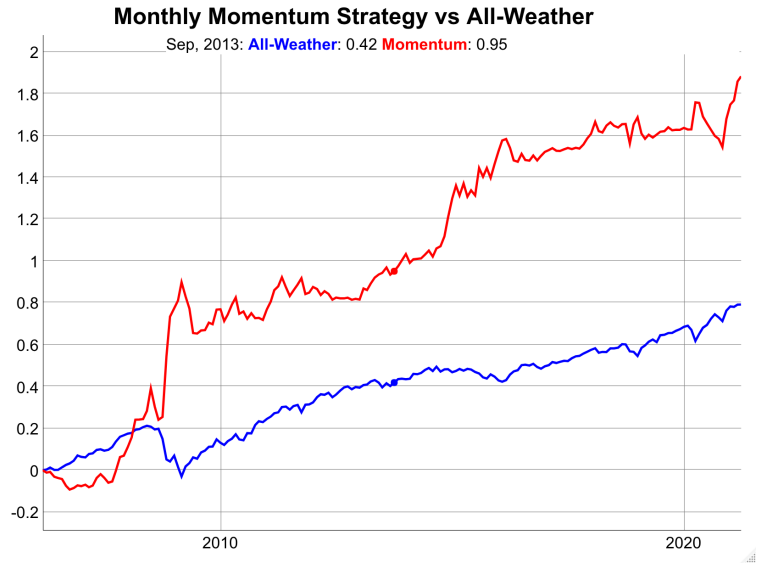



```
> source("/Users/jerzy/Develop/lecture_slides/scripts/back_test.R")
> look_backs <- seq(3, 15, by=1)
> perform_ance <- function(re_turns) sum(re_turns)/sd(re_turns)
> pro_file <- sapply(look_backs, function(look_back) {
+   pnl_s <- backtest_momentum(returns=re_turns, endp=end_p,
+     look_back=look_back, perform_ance=perform_ance)
+   sum(pnl_s)
+ })  # end sapply
> # Plot momemntum PnLs
> x11(width=6, height=5)
> plot(x=look_backs, y=pro_file, t="l",
+   main="Momemntum PnL as function of look_back",
+   xlab="look_back (months)", ylab="pnl")
```

Optimal ETF Momentum Strategy Performance

The hypothetical out-of-sample *momentum* strategy returns can be calculated by multiplying the `fu_ture` returns by the forecast *ETF* portfolio weights.

The *training* data is specified over the *look-back* intervals, and the forecast weights are applied to the future data defined by the *look-forward* intervals.




```
> # Optimal look_back
> look_back <- look_backs[which.max(pro_file)]
> pnl_s <- backtest_momentum(returns=re_turns,
+   look_back=look_back, endp=end_p,
+   perform_ance=perform_ance, with_weights=TRUE)
> tail(pnl_s)
> # Calculate the wealth of momentum returns
> ret_mom <- pnl_s[, 1]
> weal_th <- xts::xts(cbind(all_weather, ret_mom), order.by=date_s
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
```

```
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th), main="Monthly Momentum Strategy
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> quantmod::chart_Series(cumsum(weal_th), theme=plot_theme, lwd=2,
+        name="Momentum PnL")
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

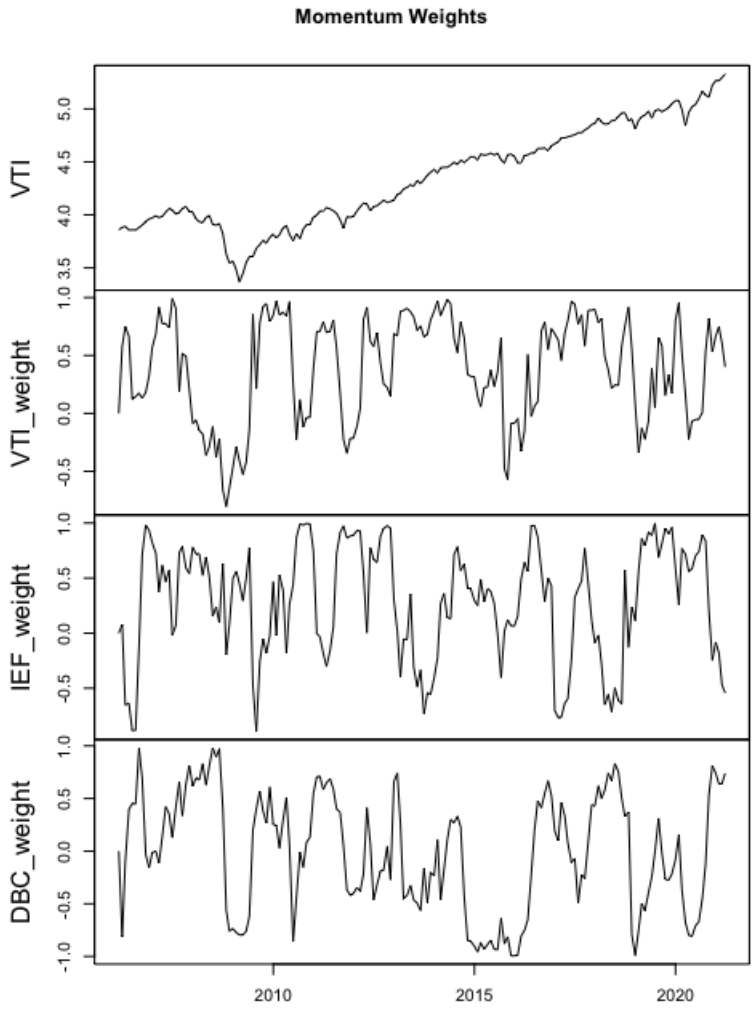
Time Series of Momentum Portfolio Weights

In *momentum* strategies, the portfolio weights are adjusted over time to be proportional to the past performance of the assets.

This way *momentum* strategies switch their weights to the best performing assets.

The weights are scaled to limit the portfolio *leverage* and its market *beta*.

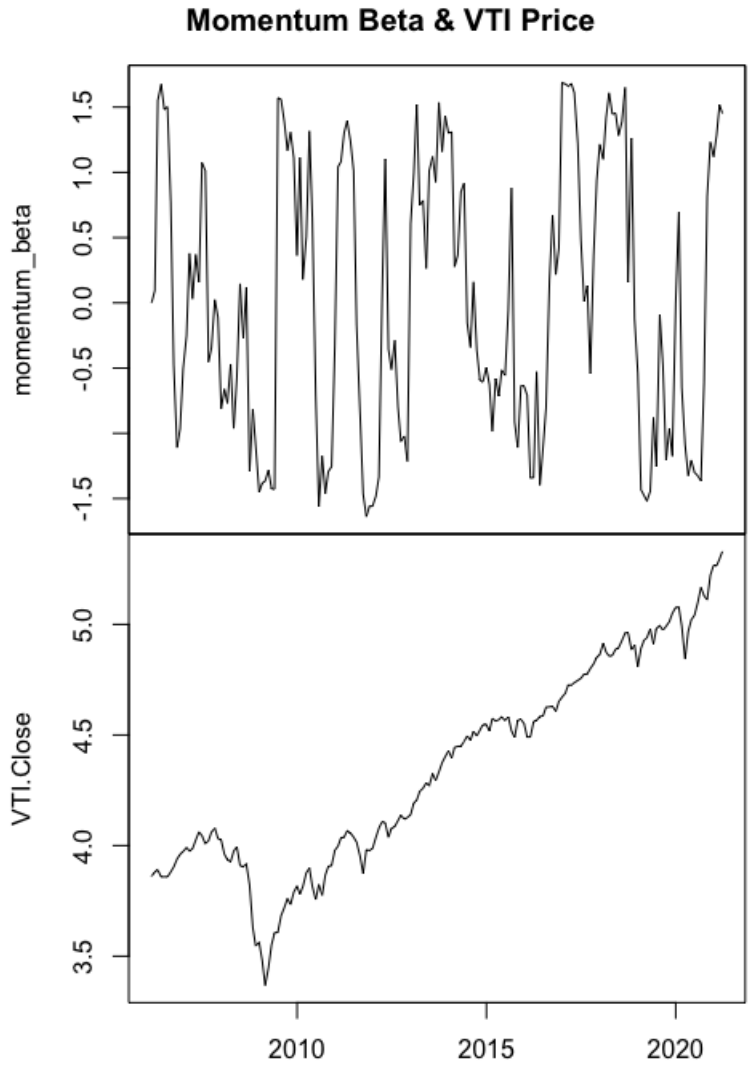
```
> # Plot the momentum portfolio weights
> weight_s <- pnl_s[, -1]
> vt_i <- log(quantmod::Cl(rutils::etf_env$VTI[date_s]))
> colnames(vt_i) <- "VTI"
> da_ta <- cbind(vt_i, weight_s)
> da_ta <- na.omit(da_ta)
> colnames(da_ta)[2:NCOL(pnl_s)] <- paste0(colnames(weight_s), "_we
> zoo::plot.zoo(da_ta, xlab=NULL, main="Momentum Weights")
```



Momentum Strategy Market Beta

The *momentum* strategy market beta can be calculated by multiplying the *ETF* betas by the *ETF* portfolio weights.

```
> # Calculate ETF betas
> betas_etf <- sapply(re_turns, function(x)
+   cov(re_turns$VTI, x)/var(x))
> # Momentum beta is equal weights times ETF betas
> beta_s <- weight_s %*% betas_etf
> beta_s <- xts::xts(beta_s, order.by=date_s)
> colnames(beta_s) <- "momentum_beta"
> da_ta <- cbind(beta_s, vt_i)
> zoo::plot.zoo(da_ta,
+   oma = c(3, 1, 3, 0), mar = c(0, 4, 0, 1),
+   main="Momentum Beta & VTI Price", xlab="")
```

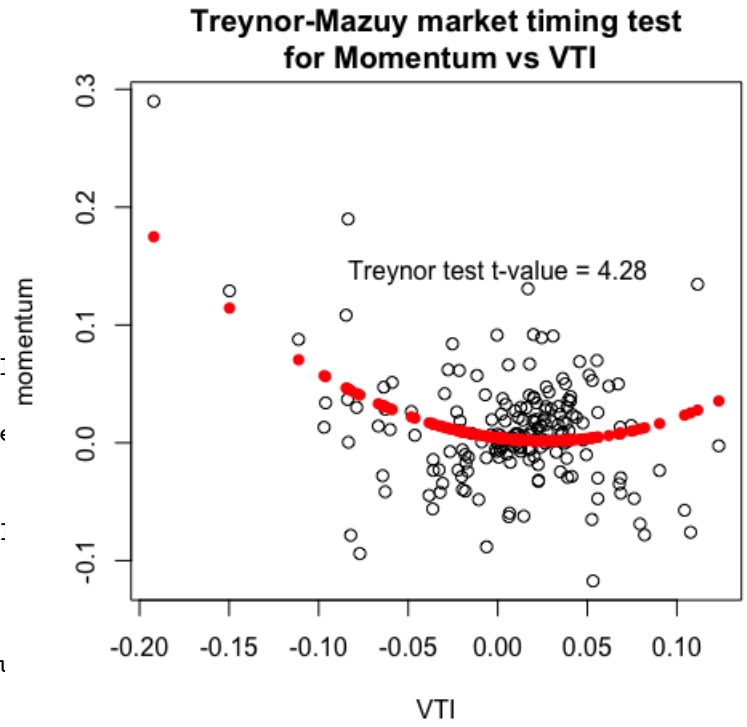


Momentum Strategy Market Timing Skill

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

The *Treynor-Mazuy* test shows that the *momentum* strategy has significant *market timing* skill.

```
> # Merton-Henriksson test
> vt_i <- rutils::diff_it(vt_i)
> de_sign <- cbind(VTI=vt_i, 0.5*(vt_i+abs(vt_i)), vt_i^2)
> colnames(de_sign)[2:3] <- c("merton", "treynor")
> mod_el <- lm(ret_mom ~ VTI + merton, data=de_sign); summary(mod_el
> # Treynor-Mazuy test
> mod_el <- lm(ret_mom ~ VTI + treynor, data=de_sign); summary(mod_e
> # Plot residual scatterplot
> plot.default(x=vt_i, y=ret_mom, xlab="VTI", ylab="momentum")
> title(main="Treynor-Mazuy market timing test\n for Momentum vs VTI
> # Plot fitted (predicted) response values
> points.default(x=vt_i, y=mod_el$fitted.values, pch=16, col="red")
> residual_s <- mod_el$residuals
> text(x=0.0, y=max(residual_s), paste("Treynor test t-value =", rou
```



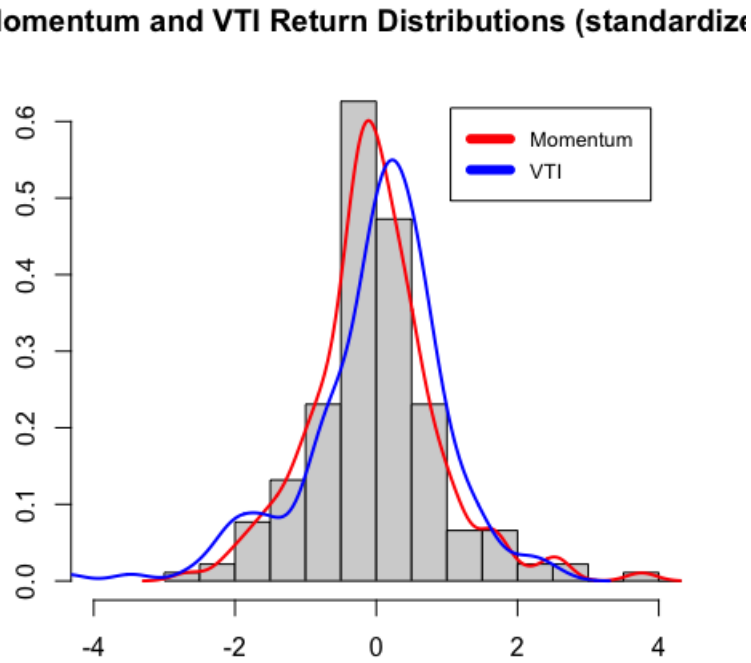
Skewness of Momentum Strategy Returns

Most assets with *positive returns* suffer from *negative skewness*.

The *momentum* strategy returns have more positive skewness compared to the negative skewness of *VTI*.

The *momentum* strategy is a genuine *market anomaly*, because it has both positive returns and positive skewness.

```
> # Standardize the returns
> ret_mom_std <- (ret_mom-mean(ret_mom))/sd(ret_mom)
> vt_i <- (vt_i-mean(vt_i))/sd(vt_i)
> # Calculate skewness and kurtosis
> apply(cbind(ret_mom_std, vt_i), 2, function(x)
+   sapply(c(skew=3, kurt=4),
+     function(e) sum(x^e)))/n_rows
```



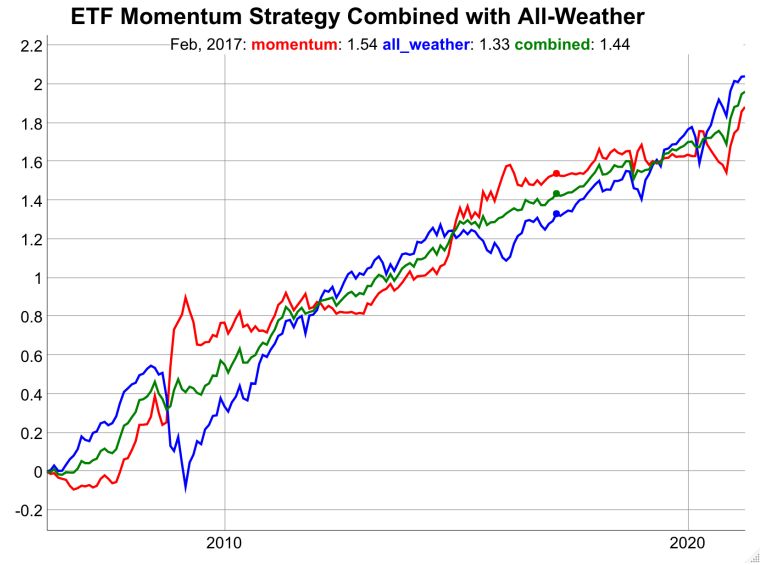

```
> # Plot histogram
> hist(ret_mom_std, breaks=30,
+   main="Momentum and VTI Return Distributions (standardized",
+   xlim=c(-4, 4),
+   xlab="", ylab="", freq=FALSE)
> # Draw kernel density of histogram
> lines(density(ret_mom_std), col='red', lwd=2)
> lines(density(vt_i), col='blue', lwd=2)
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("Momentum", "VTI"),
+  lwd=6, bg="white", col=c("red", "blue"))
```

Combining Momentum with the *All-Weather* Portfolio

The *momentum* strategy has attractive returns compared to a static buy-and-hold strategy.

But the *momentum* strategy suffers from draw-downs called *momentum crashes*, especially after the market rallies from a sharp-sell-off.

This suggests that combining the *momentum* strategy with a static buy-and-hold strategy can achieve significant diversification of risk.




```
> # Combine momentum strategy with all-weather
> all_weather <- sd(ret_mom)*all_weather/sd(all_weather)
> weal_th <- cbind(ret_mom, all_weather, 0.5*(ret_mom + all_weather)
> colnames(weal_th) <- c("momentum", "all_weather", "combined")
> # Calculate strategy annualized Sharpe ratios
> apply(weal_th, MARGIN=2, function(x) {
+   sqrt(12)*sum(x)/sd(x)/NROW(x)
+ })  # end apply
> # Calculate strategy correlations
> cor(weal_th)
> # Calculate cumulative wealth
> weal_th <- xts::xts(weal_th, date_s)
```

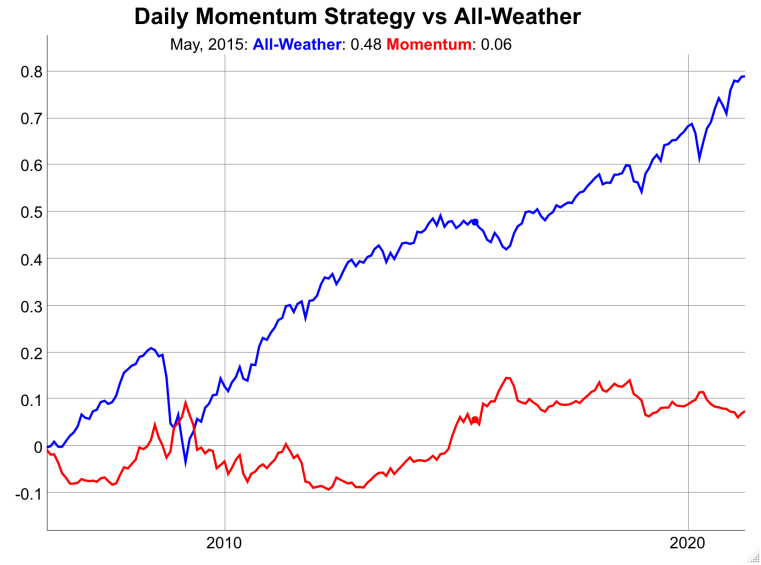
```
> # Plot ETF momentum strategy combined with All-Weather
> dygraphs::dygraph(cumsum(weal_th), main="ETF Momentum Strategy Cor
+   dyOptions(colors=c("red", "blue", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("green", "blue", "red")
> quantmod::chart_Series(weal_th, theme=plot_theme,
+        name="ETF Momentum Strategy Combined with All-Weather")
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Momentum Strategy With Daily Rebalancing

A momentum strategy with *daily* rebalancing can't be practically backtested using `apply()` loops because they are too slow.

The package *roll* contains extremely fast functions for calculating rolling aggregations using compiled `C++` code.

The momentum strategy with *daily* rebalancing performs worse than the strategy with *monthly* rebalancing.




```
> # Calculate rolling variance
> look_back <- 252
> vari_ance <- roll::roll_var(re_turns, width=look_back, min_obs=1)
> vari_ance[1, ] <- 1
> # Calculate rolling Sharpe
> pas_t <- roll::roll_mean(re_turns, width=look_back, min_obs=1)
> weight_s <- pas_t/sqrt(vari_ance)
> weight_s <- weight_s/sqrt(rowSums(weight_s^2))
> weight_s <- rutils::lag_it(weight_s)
> sum(is.na(weight_s))
> # Calculate momentum profits and losses
> pnl_s <- rowMeans(weight_s*re_turns)
```

```
> # Calculate transaction costs
> bid_offer <- 0.001
> cost_s <- 0.5*bid_offer*rowSums(abs(rutils::diff_it(weight_s)))
> pnl_s <- (pnl_s - cost_s)
> # Define all-weather benchmark
> weights_aw <- c(0.30, 0.55, 0.15)
> all_weather <- re_turns %*% weights_aw
> # Calculate the wealth of momentum returns
> weal_th <- xts::xts(cbind(all_weather, pnl_s), order.by=index(re_t
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th)[date_s], main="Daily Momentum St
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Backtesting Functional for Daily Momentum Strategy

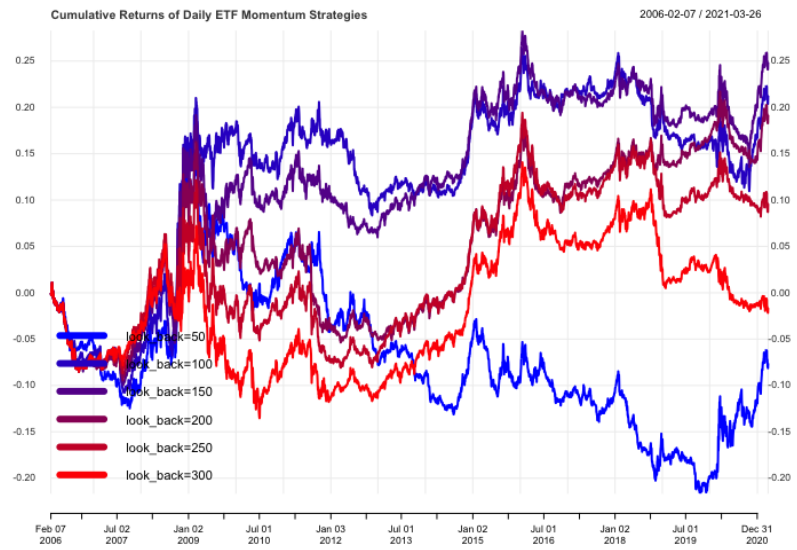
```
> # If trend=(-1) then it backtests a mean reverting strategy
> momentum_daily <- function(returns, look_back=252, bid_offer=0.001, trend=1, ...) {
+   stopifnot("package:quantmod" %in% search() || require("quantmod", quietly=TRUE))
+   # Calculate rolling variance
+   vari_ance <- roll::roll_var(returns, width=look_back, min_obs=1)
+   vari_ance[1, ] <- 1
+   vari_ance[vari_ance <= 0] <- 1
+ # Calculate rolling Sharpe
+   pas_t <- roll::roll_mean(returns, width=look_back, min_obs=1)
+   weight_s <- pas_t/sqrt(vari_ance)
+   weight_s <- weight_s/sqrt(rowSums(weight_s^2))
+   weight_s <- rutils::lag_it(weight_s)
+   # Calculate momentum profits and losses
+   pnl_s <- trend*rowMeans(weight_s*returns)
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*rowSums(abs(rutils::diff_it(weight_s)))
+   (pnl_s - cost_s)
+ }  # end momentum_daily
```

Multiple Daily ETF Momentum Strategies

Multiple daily ETF *momentum* strategies can be backtested by calling the function `momentum_daily()` in a loop over a vector of *look-back* parameters.

The *momentum* strategies do not perform well, especially the ones with a small *look-back* parameter.

```
> # Simulate a daily ETF momentum strategy
> source("/Users/jerzy/Develop/lecture_slides/scripts/back_test.R")
> pnl_s <- momentum_daily(returns=re_turns, look_back=252,
+   bid_offer=bid_offer)
> # Perform sapply loop over look_backs
> look_backs <- seq(50, 300, by=50)
> pnl_s <- sapply(look_backs, momentum_daily,
+   returns=re_turns, bid_offer=bid_offer)
> colnames(pnl_s) <- paste0("look_back=", look_backs)
> pnl_s <- xts::xts(pnl_s, index(re_turns))
> tail(pnl_s)
```




```
> # Plot dygraph of daily ETF momentum strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s), main="Daily ETF Momentum Strategi
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> quantmod::chart_Series(cumsum(pnl_s),
+   theme=plot_theme, name="Cumulative Returns of Daily ETF Momentu
> legend("bottomleft", legend=colnames(pnl_s),
+   inset=0.02, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

Daily Momentum Strategy with Holding Period

The daily ETF *momentum* strategy can be improved by introducing a holding period for the portfolio.

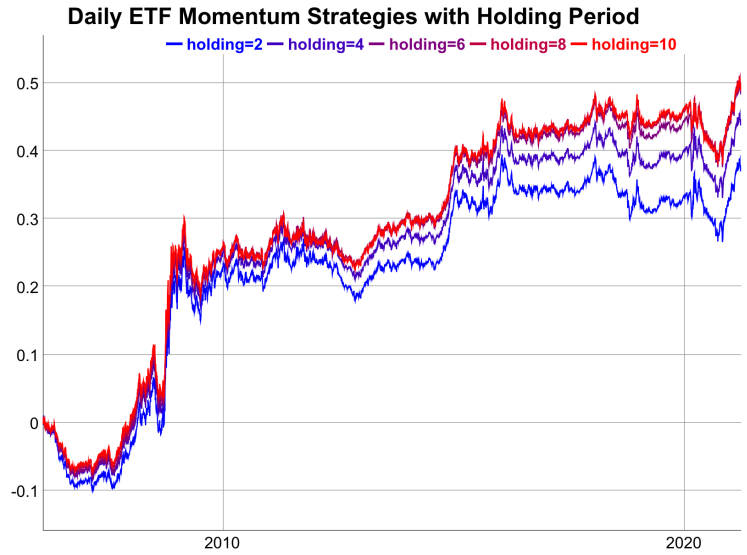
```
> # If trend=(-1) then it backtests a mean reverting strategy
> momentum_daily <- function(returns, look_back=252, hold_period=5, bid_offer=0.001, trend=1, ...) {
+   stopifnot("package:quantmod" %in% search() || require("quantmod", quietly=TRUE))
+   # Calculate rolling variance
+   vari_ance <- roll::roll_var(returns, width=look_back, min_obs=1)
+   vari_ance[1, ] <- 1
+   vari_ance[vari_ance <= 0] <- 1
+   # Calculate rolling Sharpe
+   pas_t <- roll::roll_mean(returns, width=look_back, min_obs=1)
+   weight_s <- pas_t/sqrt(vari_ance)
+   weight_s <- weight_s/sqrt(rowSums(weight_s^2))
+   weight_s <- rutils::lag_it(weight_s)
+   # Average the weights over holding period
+   weight_s <- roll::roll_mean(weight_s, width=hold_period, min_obs=1)
+   # Calculate momentum profits and losses
+   pnl_s <- trend*rowMeans(weight_s*returns)
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*rowSums(abs(rutils::diff_it(weight_s)))
+   (pnl_s - cost_s)
+ }  # end momentum_daily
```

Multiple Daily ETF Momentum Strategies

Multiple daily ETF *momentum* strategies can be backtested by calling the function `momentum_daily()` in a loop over a vector of holding periods.

The daily *momentum* strategies with a holding period perform perform much better.

```
> # Perform sapply loop over holding periods
> hold_periods <- seq(2, 11, by=2)
> pnl_s <- sapply(hold_periods, momentum_daily, look_back=120,
+             returns=re_turns, bid_offer=bid_offer)
> colnames(pnl_s) <- paste0("holding=", hold_periods)
> pnl_s <- xts::xts(pnl_s, index(re_turns))
```



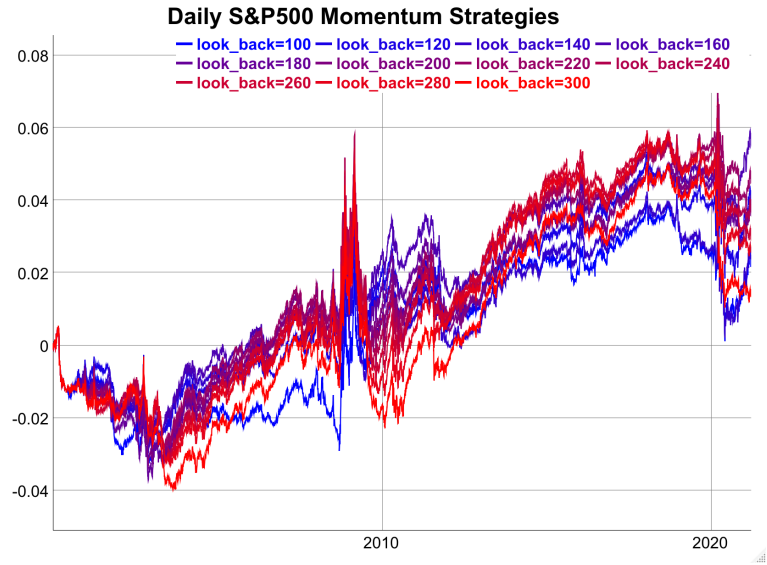

```
> # Plot dygraph of daily ETF momentum strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s), main="Daily ETF Momentum Strategi
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> quantmod::chart_Series(cumsum(pnl_s),
+   theme=plot_theme, name="Cumulative Returns of Daily ETF Momentu
> legend("bottomleft", legend=colnames(pnl_s),
+   inset=0.02, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

Backtesting Multiple S&P500 Momentum Strategies

Multiple *S&P500 momentum* strategies can be backtested by calling the function `momentum_daily()` in a loop over a vector of *look-back* parameters.

The *momentum* strategies do not perform well, especially the ones with a small *look-back* parameter.

```
> # Load daily S&P500 percentage stock returns.
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns
> # Overwrite NA values in returns_100
> returns_100 <- returns_100["2000/"]
> returns_100[1, is.na(returns_100[1, ])] <- 0
> returns_100 <- zoo::na.locf(returns_100, na.rm=FALSE)
> # Simulate a daily S&P500 momentum strategy.
> # Perform sapply loop over look_backs
> look_backs <- seq(100, 300, by=20)
> pnl_s <- sapply(look_backs, momentum_daily,
+   hold_period=5, returns=returns_100, bid_offer=0)
> colnames(pnl_s) <- paste0("look_back=", look_backs)
> pnl_s <- xts::xts(pnl_s, index(returns_100))
```



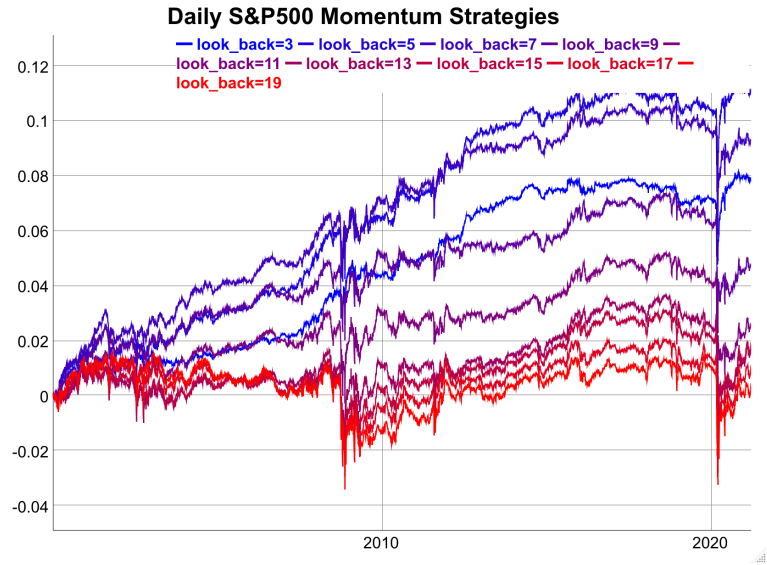

```
> # Plot dygraph of daily ETF momentum strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s), main="Daily S&P500 Momentum Strat
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot daily S&P500 momentum strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorRampPalette(c("blue", "red"))(NCOL
> quantmod::chart_Series(cumsum(pnl_s),
+   theme=plot_theme, name="Daily S&P500 Momentum Strategies")
> legend("bottomleft", legend=colnames(pnl_s),
+   inset=0.02, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

Backtesting Multiple S&P500 *Mean Reverting* Strategies

Multiple *S&P500 mean reverting* strategies can be backtested by calling the function `momentum_daily()` in a loop over a vector of *look-back* parameters.

The *mean reverting* strategies for the *S&P500* constituents perform the best for short *look-back* parameters.

The *mean reverting* strategies had their best performance prior to the 2008 financial crisis.



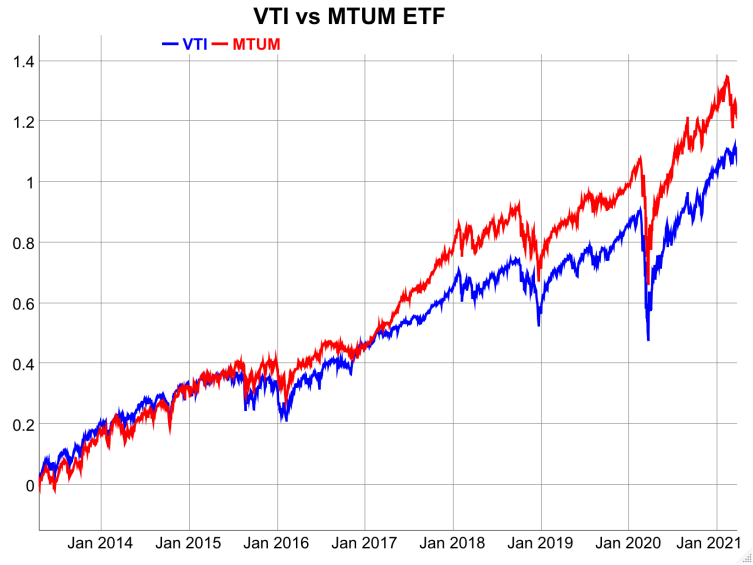

```
> # Perform sapply loop over look_backs
> look_backs <- seq(3, 20, by=2)
> pnl_s <- sapply(look_backs, momentum_daily,
+   hold_period=5, returns=returns_100, bid_offer=0, trend=(-1))
> colnames(pnl_s) <- paste0("look_back=", look_backs)
> pnl_s <- xts::xts(pnl_s, index(returns_100))
```

```
> # Plot dygraph of daily ETF momentum strategies
> col_ors <- colorRampPalette(c("blue", "red"))(NCOL(pnl_s))
> dygraphs::dygraph(cumsum(pnl_s), main="Daily S&P500 Momentum Strat
+   dyOptions(colors=col_ors, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorRampPalette(c("blue", "red"))(NCOL
> quantmod::chart_Series(cumsum(pnl_s),
+   theme=plot_theme, name="Cumulative Returns of S&P500 Mean Revert
> legend("topleft", legend=colnames(pnl_s),
+   inset=0.05, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

The MTUM Momentum ETF

The *MTUM* ETF is an actively managed ETF following a momentum strategy.

```
> # Plot cumulative returns of VTI vs MTUM ETF
> weal_th <- log(na.omit(rutils::etf_env$price_s[, c("VTI", "MTUM")]))
> colnames(weal_th) <- c("VTI", "MTUM")
> weal_th <- rutils::diff_it(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="VTI vs MTUM ETF") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(width=500)
```



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

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

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

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

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

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

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

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

http://www.rcpp.org/

http://gallery.rcpp.org/

```
> # Verify that rtools are working properly:
> devtools::find_rtools()
> devtools::has_devel()
>
> # Load package Rcpp
> library(Rcpp)
> # Get documentation for package Rcpp
> # Get short description
> packageDescription("Rcpp")
> # Load help page
> help(package="Rcpp")
> # List all datasets in "Rcpp"
> data(package="Rcpp")
> # List all objects in "Rcpp"
> ls("package:Rcpp")
> # Remove Rcpp from search path
> detach("package:Rcpp")
```

Function cppFunction() for Compiling C++ code

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

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

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

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

Performing Loops in *Rcpp Sugar*

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

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

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

```
> # Define R function with loop
> inner_mult_r <- function(x, y) {
+     to_tal <- 0
+     for(i in 1:NROW(x)) {
+ to_tal <- to_tal + x[i] * y[i]
+     }
+     to_tal
+ }  # end inner_mult_r
> # Run R function
> inner_mult_r(1:3, 6:4)
> inner_mult_r(1:3, 6:3)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=inner_mult_r(1:10000, 1:10000),
+   inner_r=1:10000 %*% 1:10000,
+   r_cpp=inner_mult(1:10000, 1:10000),
+   r_cpp_sugar=inner_mult_sugar(1:10000, 1:10000),
+   times=10))[, c(1, 4, 5)]
```

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

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

```
> # Define Ornstein-Uhlenbeck function in R
> sim_ou <- function(n_rows=1000, eq_price=5.0,
+                    vol_at=0.01, theta=0.01) {
+   re_turns <- numeric(n_rows)
+   price_s <- numeric(n_rows)
+   price_s[1] <- eq_price
+   for (i in 2:n_rows) {
+     re_turns[i] <- the_ta*(eq_price - price_s[i-1]) + vol_at*rno
+     price_s[i] <- price_s[i-1] + re_turns[i]
+   }  # end for
+   price_s
+ }  # end sim_ou
> # Simulate Ornstein-Uhlenbeck process in R
> eq_price <- 5.0; sig_ma <- 0.01
> the_ta <- 0.01; n_rows <- 1000
> set.seed(1121)  # Reset random numbers
> ou_sim <- sim_ou(n_rows=n_rows, eq_price=eq_price, vol_at=sig_ma
```

```
> # Define Ornstein-Uhlenbeck function in Rcpp
> Rcpp::cppFunction("
+ NumericVector sim_ou_rcpp(double eq_price,
+                  double vol_at,
+                  double the_ta,
+                  NumericVector in_nov) {
+   int n_rows = in_nov.size();
+   NumericVector price_s(n_rows);
+   NumericVector re_turns(n_rows);
+   price_s[0] = eq_price;
+   for (int it = 1; it < n_rows; it++) {
+     re_turns[it] = the_ta*(eq_price - price_s[it-1]) + vol_at*in_
+     price_s[it] = price_s[it-1] + re_turns[it];
+   }  // end for
+   return price_s;
+ }")  # end cppFunction
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121)  # Reset random numbers
> ou_sim_rcpp <- sim_ou_rcpp(eq_price=eq_price,
+   vol_at=sig_ma,
+   theta=the_ta,
+   innov=rnorm(n_rows))
> all.equal(ou_sim, ou_sim_rcpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=sim_ou(n_rows=n_rows, eq_price=eq_price, vol_at=sig_ma, t
+   r_cpp=sim_ou_rcpp(eq_price=eq_price, vol_at=sig_ma, theta=the_ta
+   times=10))[, c(1, 4, 5)]
```

Rcpp Attributes

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

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

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

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

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

```
> # Source Rcpp function for Ornstein-Uhlenbeck process from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121)  # Reset random numbers
> ou_sim_rcpp <- sim_ou_rcpp(eq_price=eq_price,
+   vol_at=sig_ma,
+   theta=the_ta,
+   innov=rnorm(n_rows))
> all.equal(ou_sim, ou_sim_rcpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=sim_ou(n_rows=n_rows, eq_price=eq_price, vol_at=sig_ma, theta=the_ta),
+   r_cpp=sim_ou_rcpp(eq_price=eq_price, vol_at=sig_ma, theta=the_ta, innov=rnorm(n_rows)),
+   times=10))[, c(1, 4, 5)]
```

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

// The function sim_ou_rcpp() simulates an Ornstein-Uhle
// export the function roll_maxmin() to R
// [[Rcpp::export]]
NumericVector sim_ou_rcpp(double eq_price,
                          double vol_at,
                          double the_ta,
                          NumericVector in_nov) {
  int n_rows = in_nov.size();
  NumericVector price_s(n_rows);
  NumericVector re_turns(n_rows);
  price_s[0] = eq_price;
  for (int it = 1; it < n_rows; it++) {
    re_turns[it] = the_ta*(eq_price - price_s[it-1]) + v
    price_s[it] = price_s[it-1] + re_turns[it];
  }  // end for
  return price_s;
}  // end sim_ou_rcpp
```

Generating Random Numbers Using Logistic Map in *Rcpp*

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

```
> # Calculate uniformly distributed pseudo-random sequence
> uni_form <- function(see_d, n_rows=10) {
+   out_put <- numeric(n_rows)
+   out_put[1] <- see_d
+   for (i in 2:n_rows) {
+     out_put[i] <- 4*out_put[i-1]*(1-out_put[i-1])
+   }  # end for
+   acos(1-2*out_put)/pi
+ }  # end uni_form
>
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts,
> # Microbenchmark Rcpp code
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=runif(1e5),
+   r_loop=uni_form(0.3, 1e5),
+   r_cpp=uniform_rcpp(0.3, 1e5),
+   times=10))[, c(1, 4, 5)]
```

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

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

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

Package *RcppArmadillo* for Fast Linear Algebra

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

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

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

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

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

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

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

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses RcppArmadillo.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vec_tor2, const arma::
  return arma::as_scalar(trans(vec_tor2) * (mat_rix * ve
}  // end inner_mat
```

```
> # Microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   vec_in = vec_in(vec1, vec2),
+   r_code = (vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Microbenchmark shows:
> # vec_in() is several times faster than %*%, especially for longer
> #     expr     mean   median
> # 1 vec_in 110.7067 110.4530
> # 2 r_code 585.5127 591.3575
```

Simulating *ARIMA* Processes Using *RcppArmadillo*

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

```
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Define AR(2) coefficients
> co_eff <- c(0.9, 0.09)
> n_rows <- 1e4
> set.seed(1121)
> in_nov <- rnorm(n_rows)
> # Simulate ARIMA using filter()
> arima_filter <- filter(x=in_nov,
+   filter=co_eff, method="recursive")
> # Simulate ARIMA using sim_arima()
> ari_ma <- sim_arima(in_nov, rev(co_eff))
> all.equal(drop(ari_ma),
+   as.numeric(arima_filter))
> # Microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   sim_arima = sim_arima(in_nov, rev(co_eff)),
+   filter = filter(x=in_nov, filter=co_eff, method="recursive"),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

//' @export
// [[Rcpp::export]]
arma::vec sim_arima(const arma::vec& in_nov, const arma:
  uword n_rows = in_nov.n_elem;
  uword look_back = co_eff.n_elem;
  arma::vec ari_ma(n_rows);

  // startup period
  ari_ma(0) = in_nov(0);
  ari_ma(1) = in_nov(1) + co_eff(look_back-1) * ari_ma(0
  for (uword it = 2; it < look_back-1; it++) {
    ari_ma(it) = in_nov(it) + arma::dot(co_eff.subvec(lo
  }  // end for

  // remaining periods
  for (uword it = look_back; it < n_rows; it++) {
    ari_ma(it) = in_nov(it) + arma::dot(co_eff, ari_ma.s
  }  // end for

  return ari_ma;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

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

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

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> mat_rix <- matrix(runif(1e5), nc=1e3)
> # De-mean using apply()
> new_mat <- apply(mat_rix, 2, function(x) (x-mean(x)))
> # De-mean using demean_mat()
> demean_mat(mat_rix)
> all.equal(new_mat, mat_rix)
> # Microbenchmark RcppArmadillo code
> library(microbenchmark)
> summary(microbenchmark(
+   apply = (apply(mat_rix, 2, mean)),
+   demean_mat = demean_mat(mat_rix),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Perform matrix inversion
> # Create random positive semi-definite matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- t(mat_rix) %*% mat_rix
> # Invert the matrix
> matrix_inv <- solve(mat_rix)
> inv_mat(mat_rix)
> all.equal(matrix_inv, mat_rix)
> # Microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   solve = solve(mat_rix),
+   inv_mat = inv_mat(mat_rix),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

// Examples of RcppArmadillo functions below

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

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

Fast Correlation Matrix Inverse Using *RcppArmadillo*

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

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("/Users/jerzy/Develop/lecture_slides/scripts/calc_
> # Calculate matrix of random returns
> mat_rix <- matrix(rnorm(300), nc=5)
> # Regularized inverse of correlation matrix
> eigen_max <- 4
> cor_mat <- cor(mat_rix)
> ei_gen <- eigen(cor_mat)
> in_verse <- ei_gen$vectors[, 1:eigen_max] %*%
+   (t(ei_gen$vectors[, 1:eigen_max]) / ei_gen$values[1:eigen_max])
> # Regularized inverse using RcppArmadillo
> inverse_arma <- calc_inv(cor_mat, eigen_max=eigen_max)
> all.equal(in_verse, inverse_arma)
> # Microbenchmark RcppArmadillo code
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode = {ei_gen <- eigen(cor_mat)
+ ei_gen$vectors[, 1:eigen_max] %*% (t(ei_gen$vectors[, 1:eigen_max
+   Rcpp = calc_inv(cor_mat, eigen_max=eigen_max),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& tseries,
                   double eigen_thresh = 0.001,
                   arma::uword eigen_max = 0) {

  if (eigen_max == 0) {
    // Calculate the inverse using arma::pinv()
    return arma::pinv(tseries, eigen_thresh);
  } else {
    // Calculate the regularized inverse using SVD decom
    // Allocate SVD
    arma::vec svd_val;
    arma::mat svd_u, svd_v;
    // Calculate the SVD
    arma::svd(svd_u, svd_val, svd_v, tseries);
    // Subset the SVD
    eigen_max = eigen_max - 1;
    svd_u = svd_u.cols(0, eigen_max);
    svd_v = svd_v.cols(0, eigen_max);
    svd_val = svd_val.subvec(0, eigen_max);
    // Calculate the inverse from the SVD
    return svd_v*arma::diagmat(1/svd_val)*svd_u.t();
  }  // end if
}  // end calc_inv
```

Portfolio Optimization Using *RcppArmadillo*

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

```
// Fast portfolio optimization using matrix algebra and RcppArmadillo
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>  // include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::vec calc_weights(const arma::mat& returns, // Portfolio returns
                       std::string method = "rank_sharpe",
                       double eigen_thresh = 0.001,
                       arma::uword eigen_max = 0,
                       double con_fi = 0.1,
                       double alpha = 0.0,
                       bool scale = true,
                       double vol_target = 0.01) {
  // Initialize
  arma::vec weights(returns.n_cols, fill::zeros);
  if (eigen_max == 0)  eigen_max = returns.n_cols;

  // Switch for the different methods for weights
  switch(calc_method(method)) {
  case meth_od::rank_sharpe: {
    // Mean returns by columns
    arma::vec mean_cols = arma::trans(arma::mean(returns, 0));
    // Standard deviation by columns
    arma::vec sd_cols = arma::trans(arma::stddev(returns, 0));
    sd_cols.replace(0, 1);
    mean_cols = mean_cols/sd_cols;
    // Weights equal to ranks of Sharpe
    weights = conv_to<vec>::from(arma::sort_index(arma::sort_index(mean_cols)));
    weights = (weights - arma::mean(weights));
    break;
```

Strategy Backtesting Using *RcppArmadillo*

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

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

//' @export
// [[Rcpp::export]]
arma::mat back_test(const arma::mat& excess, // Portfolio excess returns
                    const arma::mat& returns, // Portfolio returns
                    arma::uvec startp,
                    arma::uvec endp,
                    std::string method = "rank_sharpe",
                    double eigen_thresh = 0.001,
                    arma::uword eigen_max = 0,
                    double con_fi = 0.1,
                    double alpha = 0.0,
                    bool scale = true,
                    double vol_target = 0.01,
                    double coeff = 1.0,
                    double bid_offer = 0.0) {

  arma::vec weights(returns.n_cols, fill::zeros);
  arma::vec weights_past = zeros(returns.n_cols);
  arma::mat pnl_s = zeros(returns.n_rows, 1);

  // Perform loop over the end points
  for (arma::uword it = 1; it < endp.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate portfolio weights
    weights = coeff*calc_weights(excess.rows(startp(it-1), endp(it-1)), method, eigen_thresh, eigen_max, con_fi
    // Calculate out-of-sample returns
    pnl_s.rows(endp(it-1)+1, endp(it)) = returns.rows(endp(it-1)+1, endp(it))*weights;
    // Add transaction costs
    pnl_s.row(endp(it-1)+1) -= bid_offer*sum(abs(weights - weights_past))/2;
```

Portfolio Optimization Strategy

The *portfolio optimization* strategy invests in the best performing portfolio in the past *in-sample* interval, expecting that it will continue performing well *out-of-sample*.

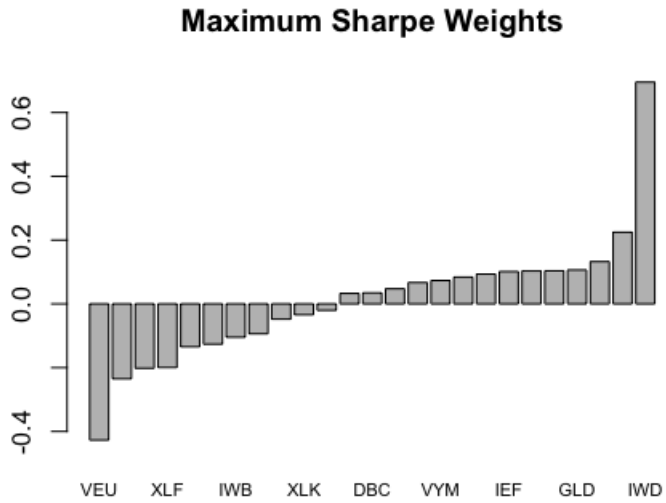
The *portfolio optimization* strategy consists of:

1. Calculating the maximum Sharpe ratio portfolio weights in the *in-sample* interval,
2. Applying the weights and calculating the portfolio returns in the *out-of-sample* interval.

The optimal portfolio weights $\mathbf{w}$ are calculated using the past in-sample excess returns $\mu = \mathbf{r} - r_f$ (in excess of the risk-free rate r_f):

$$\mathbf{w} = \mathbb{C}^{-1}\mu$$

This strategy has performed well for *ETF* portfolios because of the consistent performance of bond ETFs, like *TLT* and *VYM*.



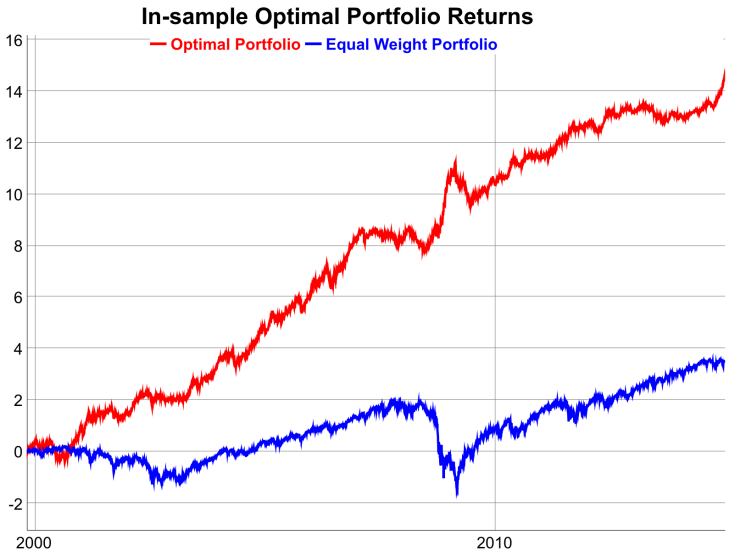

```
> # Select all the ETF symbols except "VXX", "SVXY" and "MTUM"
> sym_bols <- colnames(rutils::etf_env$re_turns)
> sym_bols <- sym_bols[!(sym_bols %in% c("VXX", "SVXY", "MTUM", "QU
> # Extract columns of rutils::etf_env$re_turns and overwrite NA va
> re_turns <- rutils::etf_env$re_turns[, sym_bols]
> n_assets <- NCOL(re_turns)
> # re_turns <- na.omit(re_turns)
> re_turns[1, is.na(re_turns[1, ])] <- 0
> re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
> # Returns in excess of risk-free rate
> risk_free <- 0.03/252
> ex_cess <- (re_turns - risk_free)
```

```
> # Maximum Sharpe weights in-sample interval
> rets_is <- re_turns["/2014"]
> in_verse <- MASS::ginv(cov(rets_is))
> weight_s <- in_verse %*% colMeans(ex_cess["/2014"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Plot portfolio weights
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> barplot(sort(weight_s), main="Maximum Sharpe Weights", cex.names=0
```

Portfolio Optimization Strategy In-Sample

The in-sample performance of the optimal portfolio is much better than the equal weight portfolio.

```
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> in_dex <- xts::xts(rowSums(rets_is)/sqrt(n_assets), index(rets_is)
> portf_is <- portf_is*sd(in_dex)/sd(portf_is)
```



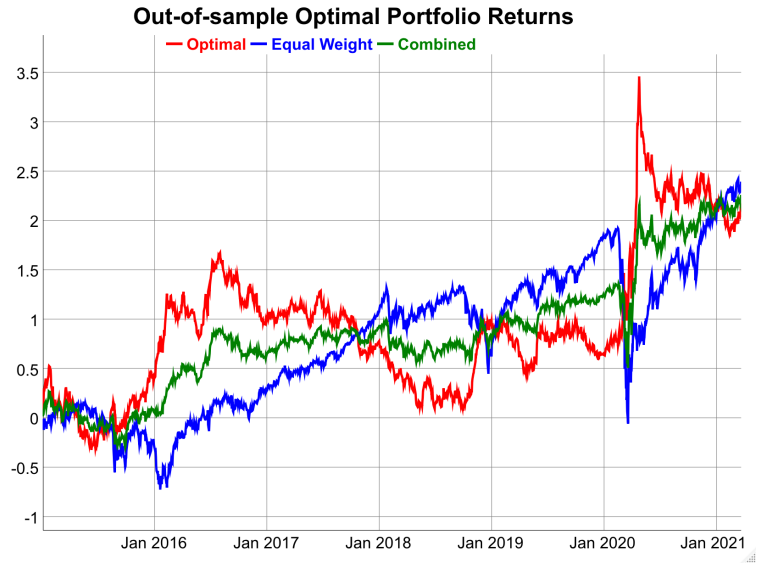

```
> # Plot cumulative portfolio returns
> pnl_s <- cumsum(cbind(portf_is, in_dex))
> colnames(pnl_s) <- c("Optimal Portfolio", "Equal Weight Portfolio"
> dygraphs::dygraph(pnl_s, main="In-sample Optimal Portfolio Returns
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Portfolio Optimization Strategy Out-of-Sample

The out-of-sample performance of the optimal portfolio is not nearly as good as in-sample.

Combining the optimal portfolio with the equal weight portfolio produces and even better performing portfolio.

```
> # Out-of-sample portfolio returns
> rets_os <- re_turns["2015/"]
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(rets_os)/sqrt(n_assets), index(rets_os)
> portf_os <- portf_os*sd(in_dex)/sd(portf_os)
> pnl_s <- cbind(portf_os, in_dex, (portf_os + in_dex)/2)
> colnames(pnl_s) <- c("Optimal", "Equal Weight", "Combined")
> sapply(pnl_s, function(x) mean(x)/sd(x))
```

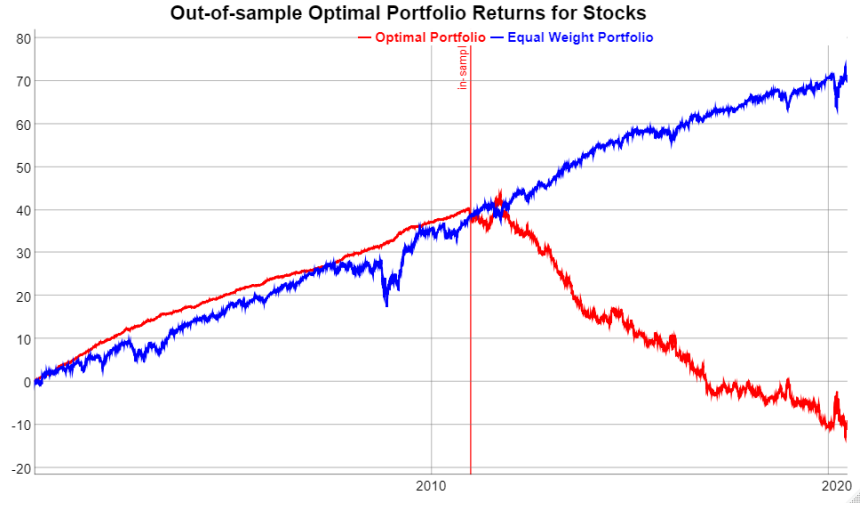



```
> # Plot cumulative portfolio returns
> dygraphs::dygraph(cumsum(pnl_s), main="Out-of-sample Optimal Portf
+   dyOptions(colors=c("red", "blue", "green"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Portfolio Optimization Strategy for Stocks

The *portfolio optimization* strategy for stocks is *overfitted* in the *in-sample* interval.

Therefore the strategy completely fails in the *out-of-sample* interval.




```
> load("/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData
> # Overwrite NA values in re_turns
> re_turns <- re_turns["2000/"]
> n_assets <- NCOL(re_turns)
> re_turns[1, is.na(re_turns[1, ])] <- 0
> re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
> risk_free <- 0.03/252
> ex_cess <- (re_turns - risk_free)
> rets_is <- re_turns["/2010"]
> rets_os <- re_turns["2011/"]
> # Maximum Sharpe weights in-sample interval
> cov_mat <- cov(rets_is)
> in_verse <- MASS::ginv(cov_mat)
> weight_s <- in_verse %*% colMeans(ex_cess["/2010"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turns))
```

```
> # Plot cumulative portfolio returns
> pnl_s <- rbind(portf_is, portf_os)
> pnl_s <- pnl_s*sd(in_dex)/sd(pnl_s)
> pnl_s <- cumsum(cbind(pnl_s, in_dex))
> colnames(pnl_s) <- c("Optimal Portfolio", "Equal Weight Portfolio"
> dygraphs::dygraph(pnl_s, main="Out-of-sample Optimal Portfolio Ret
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Regularized Inverse of Singular Covariance Matrices

The *regularization* technique allows calculating the inverse of *singular* covariance matrices while reducing the effects of statistical noise.

If the number of time periods of returns is less than the number of assets (columns), then the covariance matrix of returns is *singular*, and some of its *eigenvalues* are zero, so it doesn't have an inverse.

The *regularized* inverse $\mathbb{C}_n^{-1}$ is calculated by removing the higher order eigenvalues that are almost zero, and keeping only the first *n eigenvalues*:

$$\mathbb{C}_n^{-1} = \mathbb{O}_n \, \mathbb{D}_n^{-1} \, \mathbb{O}_n^T$$

Where $\mathbb{D}_n$ and $\mathbb{O}_n$ are matrices with the higher order eigenvalues and eigenvectors removed.

The function `MASS::ginv()` calculates the *regularized* inverse of a matrix.

```
> # Create rectangular matrix with collinear columns
> ran_dom <- matrix(rnorm(10*8), nc=10)
> # Calculate covariance matrix
> cov_mat <- cov(ran_dom)
> # Calculate inverse of cov_mat - error
> in_verse <- solve(cov_mat)
> # Perform eigen decomposition
> ei_gen <- eigen(cov_mat)
> eigen_vec <- ei_gen$vectors
> eigen_val <- ei_gen$values
> # Set tolerance for determining zero singular values
> to_l <- sqrt(.Machine$double.eps)
> # Calculate regularized inverse matrix
> not_zero <- (eigen_val > (to_l * eigen_val[1]))
> inv_reg <- eigen_vec[, not_zero] %*%
+   (t(eigen_vec[, not_zero]) / eigen_val[not_zero])
> # Verify inverse property of inv_reg
> all.equal(cov_mat, cov_mat %*% inv_reg %*% cov_mat)
> # Calculate regularized inverse of cov_mat
> in_verse <- MASS::ginv(cov_mat)
> # Verify inverse property of mat_rix
> all.equal(in_verse, inv_reg)
```

Shrinkage Inverse of the Covariance Matrix

For a portfolio of *S&P500* stocks, the number return columns is very large, which may make the covariance matrix of returns simgular.

Removing the very small higher order eigenvalues also reduces the propagation of statistical noise and improves the signal-to-noise ratio.

But removing a larger number of eigenvalues increases the bias of the covariance matrix, which is an example of the *bias-variance tradeoff*.

Even though the *shrinkage inverse* $\mathbb{C}_n^{-1}$ does not satisfy the matrix inverse property, its out-of-sample forecasts may be more accurate than those using the actual inverse matrix.

The parameter `max_eigen` specifies the number of eigenvalues used for calculating the *regularized* inverse of the covariance matrix of returns.

The optimal value of the parameter `max_eigen` can be determined using *backtesting* (*cross-validation*).

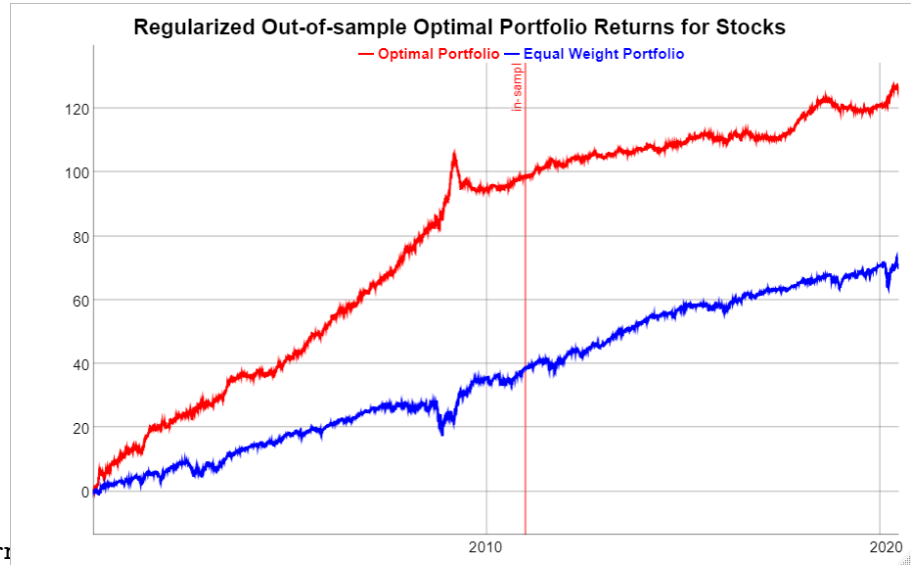
```
> # Calculate in-sample covariance matrix
> cov_mat <- cov(rets_is)
> ei_gen <- eigen(cov_mat)
> eigen_vec <- ei_gen$vectors
> eigen_val <- ei_gen$values
> # Calculate shrinkage inverse of covariance matrix
> eigen_max <- 21
> in_verse <- eigen_vec[, 1:eigen_max] %*%
+   (t(eigen_vec[, 1:eigen_max]) / ei_gen$values[1:eigen_max])
```

Portfolio Optimization Strategy with Shrinkage

The *out-of-sample* performance of the *portfolio optimization* strategy is greatly improved by shrinking the inverse of the covariance matrix.

The *in-sample* performance is worse because shrinkage reduces *overfitting*.

```
> # Calculate portfolio weights
> weight_s <- in_verse %*% colMeans(ex_cess["/2010"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turn
```




```
> # Plot cumulative portfolio returns
> pnl_s <- rbind(portf_is, portf_os)
> pnl_s <- pnl_s*sd(in_dex)/sd(pnl_s)
> pnl_s <- cumsum(cbind(pnl_s, in_dex))
> colnames(pnl_s) <- c("Optimal Portfolio", "Equal Weight Portfolio"
> dygraphs::dygraph(pnl_s, main="Regularized Out-of-sample Optimal P
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Optimal Portfolio Weights With Return Shrinkage

To further reduce the statistical noise, the individual returns r_i can be *shrunk* to the average portfolio returns $\bar{r}$:

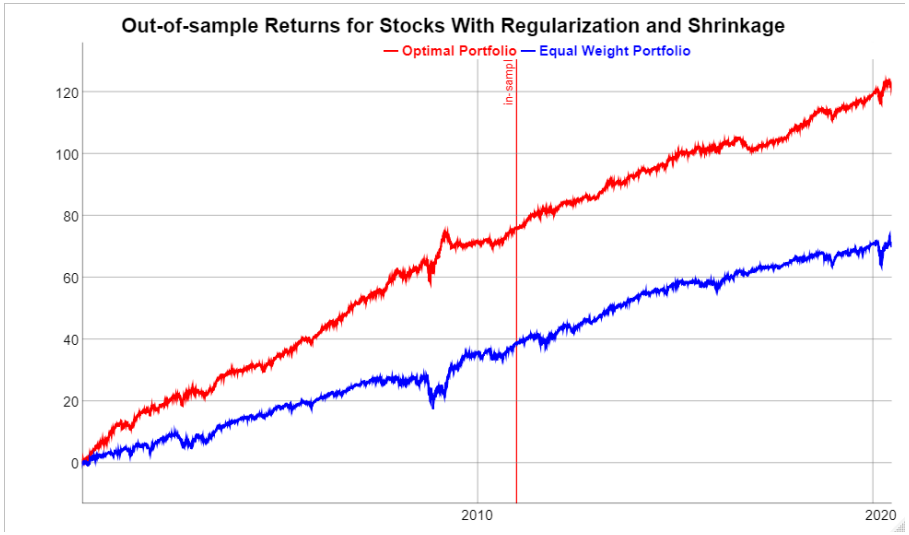
$$r_i = (1 - \alpha)\, r_i + \alpha\, \bar{r}$$

The parameter α is the *shrinkage* intensity, and it determines the strength of the *shrinkage* of individual returns to their mean.

If $\alpha = 0$ then there is no *shrinkage*, while if $\alpha = 1$ then all the returns are *shrunk* to their common mean: $r_i = \bar{r}$.

The optimal value of the *shrinkage* intensity α can be determined using *backtesting* (*cross-validation*).

```
> # Shrink the in-sample returns to their mean
> rets_mean <- colMeans(rets_is) - risk_free
> al_pha <- 0.7
> rets_mean <- (1 - al_pha)*rets_mean + al_pha*mean(rets_mean)
```




```
> # Calculate portfolio weights
> weight_s <- in_verse %*% rets_mean
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> # Plot cumulative portfolio returns
> pnl_s <- rbind(portf_is, portf_os)
> pnl_s <- pnl_s*sd(in_dex)/sd(pnl_s)
> pnl_s <- cumsum(cbind(pnl_s, in_dex))
> colnames(pnl_s) <- c("Optimal Portfolio", "Equal Weight Portfolio"
> dygraphs::dygraph(pnl_s, main="Out-of-sample Returns for Stocks Wi
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Fast Covariance Matrix Inverse Using *RcppArmadillo*

RcppArmadillo can be used to quickly calculate the regularized inverse of a covariance matrix.

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

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& mat_rix, const arma::uword& eigen_max) {
  arma::mat eigen_vec;
  arma::vec eigen_val;

  arma::eig_sym(eigen_val, eigen_vec, cov(mat_rix));
  eigen_vec = eigen_vec.cols(eigen_vec.n_cols-eigen_max, eigen_vec.n_cols-1);
  eigen_val = 1/eigen_val.subvec(eigen_val.n_elem-eigen_max, eigen_val.n_elem-1);

  return eigen_vec*diagmat(eigen_val)*eigen_vec.t();

}  // end calc_inv
```

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("/Users/jerzy/Develop/lecture_slides/scripts/calc_weights.cpp")
> # Create random matrix of returns
> mat_rix <- matrix(rnorm(300), nc=5)
> # Regularized inverse of covariance matrix
> eigen_max <- 4
> ei_gen <- eigen(cov(mat_rix))
> cov_inv <- ei_gen$vectors[, 1:eigen_max] %*%
+   (t(ei_gen$vectors[, 1:eigen_max]) / ei_gen$values[1:eigen_max])
> # Regularized inverse using RcppArmadillo
> cov_inv_arma <- calc_inv(mat_rix, eigen_max)
> all.equal(cov_inv, cov_inv_arma)
> # Microbenchmark RcppArmadillo code
```

Portfolio Optimization Using *RcppArmadillo*

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

```
// Fast portfolio optimization using matrix algebra and RcppArmadillo
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>  // include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::vec calc_weights(const arma::mat& re_turns,
                       const std::string& typ_e = "max_sharpe",
                       int eigen_max = 1,
                       const double& pro_b = 0.1,
                       const double& alpha = 0.0,
                       const bool scal_e = true) {
  // Initialize
  arma::vec weight_s(re_turns.n_cols);
  if (eigen_max == 1)  eigen_max = re_turns.n_cols;

  // Calculate weights depending on typ_e
  if (typ_e == "max_sharpe") {
    // Mean returns by columns
    arma::vec mean_cols = arma::trans(arma::mean(re_turns, 0));
    // Shrink mean_cols to the mean of re_turns
    mean_cols = ((1-al_pha)*mean_cols + al_pha*arma::mean(mean_cols));
    // Apply regularized inverse
    weight_s = calc_inv(re_turns, eigen_max)*mean_cols;
  } else if (typ_e == "max_sharpe_median") {
    // Mean returns by columns
    arma::vec mean_cols = arma::trans(arma::median(re_turns, 0));
    // Shrink mean_cols to the mean of re_turns
    mean_cols = ((1-al_pha)*mean_cols + al_pha*arma::median(mean_cols));
    // Apply regularized inverse
    weight_s = calc_inv(re_turns, eigen_max)*mean_cols;
```

Strategy Backtesting Using *RcppArmadillo*

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

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

arma::mat back_test(const arma::mat& ex_cess, // Portfolio excess returns
                    const arma::mat& re_turns, // Portfolio returns
                    const arma::uvec& start_p,
                    const arma::uvec& end_p,
                    const std::string& typ_e = "max_sharpe",
                    const arma::uword& eigen_max = 1,
                    const double& pro_b = 0.1,
                    const double& alpha = 0,
                    const bool& scal_e = true,
                    const double& co_eff = 1.0,
                    const double& bid_offer = 0.0) {
  arma::vec pnl_s = zeros(re_turns.n_rows);
  arma::vec weights_past = zeros(re_turns.n_cols);
  arma::vec weight_s(re_turns.n_cols);

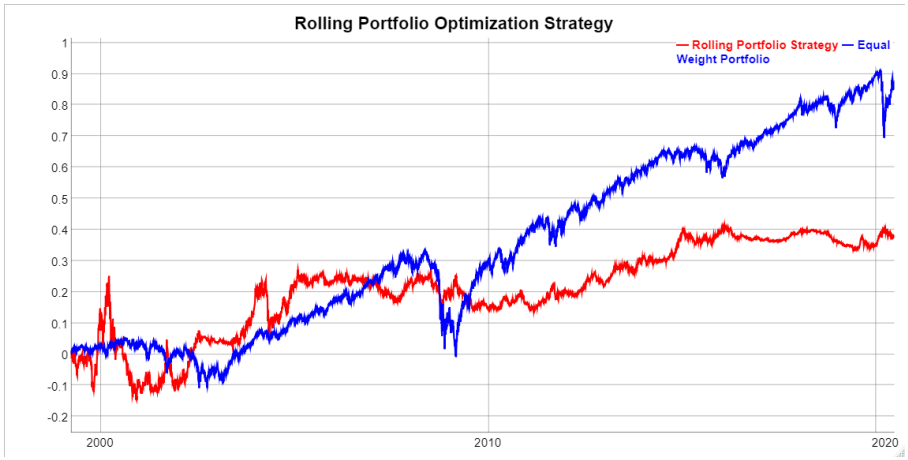
  // Perform loop over the end_p
  for (arma::uword it=1; it < end_p.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate portfolio weights
    weight_s = co_eff*calc_weights(ex_cess.rows(start_p(it-1), end_p(it-1)), typ_e, eigen_max, pro_b, al_pha, s
    // Calculate out-of-sample returns
    pnl_s.subvec(end_p(it-1)+1, end_p(it)) = re_turns.rows(end_p(it-1)+1, end_p(it))*weight_s;
    // Add transaction costs
    pnl_s.row(end_p(it-1)+1) -= bid_offer*sum(abs(weight_s - weights_past))/2;
    weights_past = weight_s;
  }  // end for
  // Return the strategy returns
  return pnl_s;
}  // end back_test
```

Rolling Portfolio Optimization Strategy

A *rolling portfolio optimization* strategy consists of rebalancing a portfolio over the end points:

1. Calculate the maximum Sharpe ratio portfolio weights at each end point,
2. Apply the weights in the next interval and calculate the out-of-sample portfolio returns.

The parameters of this strategy are: the rebalancing frequency (annual, monthly, etc.), and the length of look-back interval.




```
> # Calculate vector of monthly end points and start points
> end_p <- rutils::calc_endpoints(re_turns, inter_val="months")
> end_p <- end_p[end_p > 2*NCOL(re_turns)]
> n_rows <- NROW(end_p)
> look_back <- 24
> start_p <- c(rep_len(0, look_back-1),
+        end_p[1:(n_rows-look_back+1)])
```

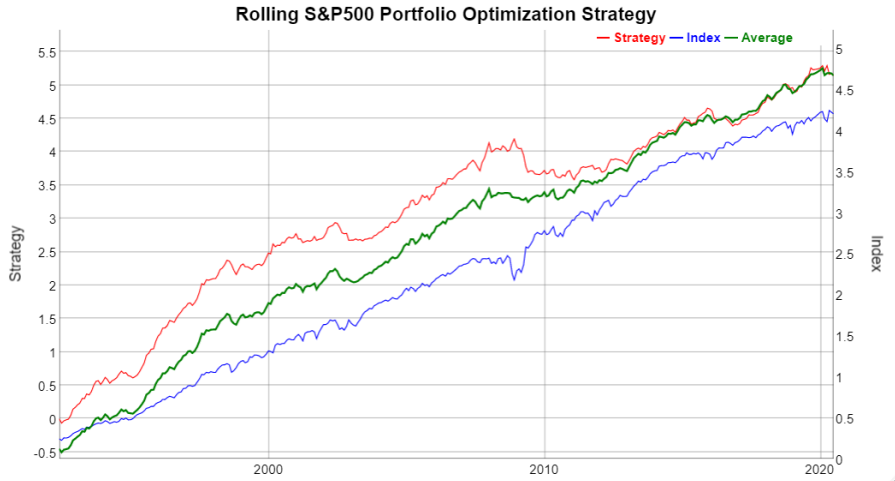
```
> # Perform loop over end points
> rets_portf <- lapply(2:n_rows, function(i) {
+     # Subset the ex_cess returns
+     ex_cess <- ex_cess[start_p[i-1]:end_p[i-1], ]
+     in_verse <- MASS::ginv(cov(ex_cess))
+     # Calculate the maximum Sharpe ratio portfolio weights
+     weight_s <- in_verse %*% colMeans(ex_cess)
+     weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
+     # Calculate the out-of-sample portfolio returns
+     re_turns <- re_turns[(end_p[i-1]+1):end_p[i], ]
+     xts::xts(re_turns %*% weight_s, index(re_turns))
+ })  # end lapply
> rets_portf <- rutils::do_call(rbind, rets_portf)
> # Plot cumulative strategy returns
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turn
> pnl_s <- cumsum(na.omit(cbind(rets_portf, in_dex*sd(rets_portf)/sd
> colnames(pnl_s) <- c("Rolling Portfolio Strategy", "Equal Weight P
> dygraphs::dygraph(pnl_s, main="Rolling Portfolio Optimization Stra
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Rolling Portfolio Optimization Strategy for S&P500

In the rolling portfolio optimization strategy the portfolio weights are adjusted to their optimal values at every end point.

A portfolio optimization is performed using past data, and the optimal portfolio weights are applied out-of-sample in the next interval.

The weights are scaled to match the volatility of the equally weighted portfolio, and are kept constant until the next end point.



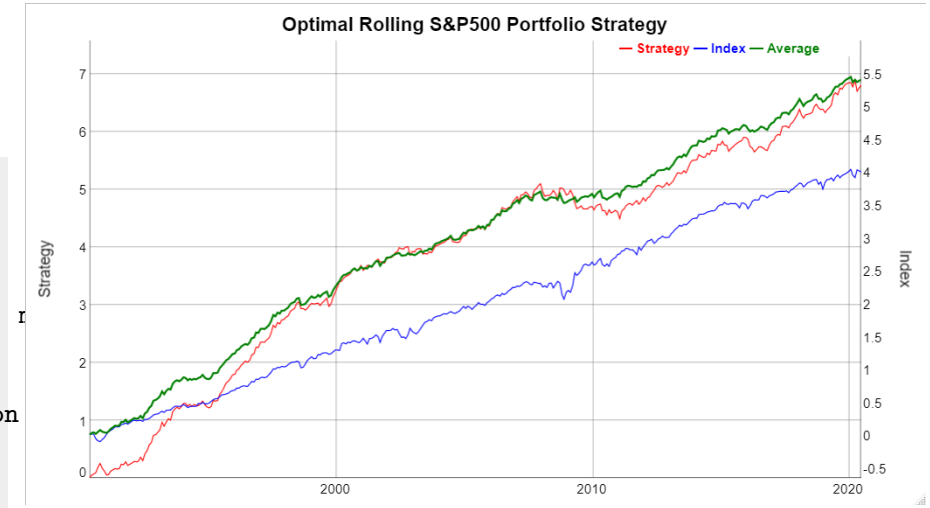

```
> load("/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData")
> # Overwrite NA values in re_turns
> returns_100[1, is.na(returns_100[1, ])] <- 0
> returns_100 <- zoo::na.locf(returns_100, na.rm=FALSE)
> n_cols <- NCOL(returns_100) ; date_s <- index(returns_100)
> # Define monthly end points
> end_p <- rutils::calc_endpoints(returns_100, inter_val="months")
> end_p <- end_p[end_p > (n_cols+1)]
> n_rows <- NROW(end_p) ; look_back <- 12
> start_p <- c(rep_len(0, look_back-1), end_p[1:(n_rows-look_back+1
> end_p <- (end_p - 1)
> start_p <- (start_p - 1)
> start_p[start_p < 0] <- 0
> al_pha <- 0.7 ; eigen_max <- 21
```

```
> # Perform backtest in Rcpp
> pnl_s <- HighFreq::back_test(excess=returns_100, returns=returns_1
+   startp=start_p, endp=end_p, alpha=al_pha, eigen_max=eigen_max, r
> # Calculate returns on equal weight portfolio
> in_dex <- xts::xts(rowMeans(returns_100), index(returns_100))
> # Plot cumulative strategy returns
> pnl_s <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> pnl_s <- cumsum(na.omit(pnl_s))
> col_names <- c("Strategy", "Index", "Average")
> colnames(pnl_s) <- col_names
> dygraphs::dygraph(pnl_s[end_p], main="Rolling S&P500 Portfolio Opt
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=1
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Determining Strategy Parameters Using Backtesting

The optimal values of the parameters `max_eigen` and α can be determined using *backtesting*.

```
> # Perform backtest over alphas
> alpha_s <- seq(from=0.01, to=0.91, by=0.1)
> pnl_s <- lapply(alpha_s, function(al_pha) {
+   HighFreq::back_test(excess=returns_100, returns=returns_100,
+   startp=start_p, endp=end_p, alpha=al_pha, eigen_max=eigen_max, r
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=alpha_s, y=pro_file, t="l", main="Strategy PnL as Function
+   xlab="Shrinkage Intensity Alpha", ylab="pnl")
> al_pha <- alpha_s[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
> # Perform backtest over eigen_maxs
> eigen_maxs <- seq(from=3, to=40, by=2)
> pnl_s <- lapply(eigen_maxs, function(eigen_max) {
+   HighFreq::back_test(excess=returns_100, returns=returns_100,
+     startp=start_p, endp=end_p, alpha=al_pha, eigen_max=eigen_max
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=eigen_maxs, y=pro_file, t="l", main="Strategy PnL as Funct
+   xlab="eigen_max", ylab="pnl")
> eigen_max <- eigen_maxs[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
```

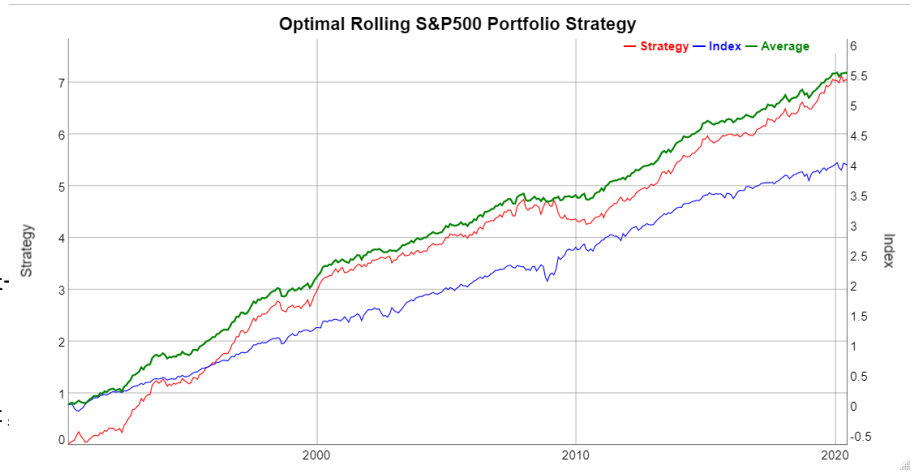



```
> # Plot cumulative strategy returns
> pnl_s <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> pnl_s <- cumsum(na.omit(pnl_s))
> col_names <- c("Strategy", "Index", "Average")
> colnames(pnl_s) <- col_names
> dygraphs::dygraph(pnl_s[end_p], main="Optimal Rolling S&P500 Portf
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=1
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Determining Look-back Interval Using Backtesting

The optimal value of the look-back interval can be determined using *backtesting*.

```
> # Perform backtest over look-backs
> look_backs <- seq(from=3, to=24, by=1)
> pnl_s <- lapply(look_backs, function(look_back) {
+   start_p <- c(rep_len(0, look_back-1), end_p[1:(n_rows-look_back-
+   start_p <- (start_p - 1)
+   start_p[start_p < 0] <- 0
+   HighFreq::back_test(excess=returns_100, returns=returns_100,
+     startp=start_p, endp=end_p, alpha=al_pha, eigen_max=eigen_max
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=look_backs, y=pro_file, t="l", main="Strategy PnL as Funct
+   xlab="Look-back Interval", ylab="pnl")
> look_back <- look_backs[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
```




```
> # Plot cumulative strategy returns
> pnl_s <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> pnl_s <- cumsum(na.omit(pnl_s))
> col_names <- c("Strategy", "Index", "Average")
> colnames(pnl_s) <- col_names
> dygraphs::dygraph(pnl_s[end_p], main="Optimal Rolling S&P500 Portf
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=1
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Dynamic Documents Using *R markdown*

markdown is a simple markup language designed for creating documents in different formats, including *pdf* and *html*.

R Markdown is a modified version of *markdown*, which allows creating documents containing *math formulas* and `R` code embedded in them.

An *R Markdown* document (with extension `.Rmd`) contains:

- A *YAML* header,
- Text in *R Markdown* code format,
- Math formulas (equations), delimited using either single "$" symbols (for inline formulas), or double "$$" symbols (for display formulas),
- `R` code chunks, delimited using either single "`" backtick symbols (for inline code), or triple "```" backtick symbols (for display code).

The packages *rmarkdown* and *knitr* compile `R` documents into either *pdf*, *html*, or *MS Word* documents.

````
---
title: "My First R Markdown Document"
author: Jerzy Pawlowski
date: ''`r format(Sys.time(), "%m/%d/%Y")`''
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)

# install package quantmod if it can't be loaded success
if (!require("quantmod"))
  install.packages("quantmod")
```

### R Markdown
This is an *R Markdown* document. Markdown is a simple f

One of the advantages of writing documents *R Markdown*

You can read more about publishing documents using *R* h
https://algoquant.github.io/r,/markdown/2016/07/02/Publi

You can read more about using *R* to create *HTML* docum
https://algoquant.github.io/2016/07/05/Interactive-Plots

Clicking the **Knit** button in *RStudio*, compiles the

Example of an *R* code chunk:
```{r cars}
summary(cars)
```

### Plots in *R Markdown* documents

Plots can also be embedded, for example:
```{r pressure, echo=FALSE}
plot(pressure)
````
````

Package *shiny* for Creating Interactive Applications

The package *shiny* creates interactive applications running in R, with their outputs presented as live visualizations.

Shiny allows changing the model parameters, recalculating the model, and displaying the resulting outputs as plots and charts.

A *shiny app* is a file with *shiny* commands and R code.

The *shiny* code consists of a *shiny interface* and a *shiny server*.

The *shiny interface* contains widgets for data input and an area for plotting.

The *shiny server* contains the R model code and the plotting code.

The function `shiny::fluidPage()` creates a GUI layout for the user inputs of model parameters and an area for plots and charts.

The function `shiny::renderPlot()` renders a plot from the outputs of a live model.

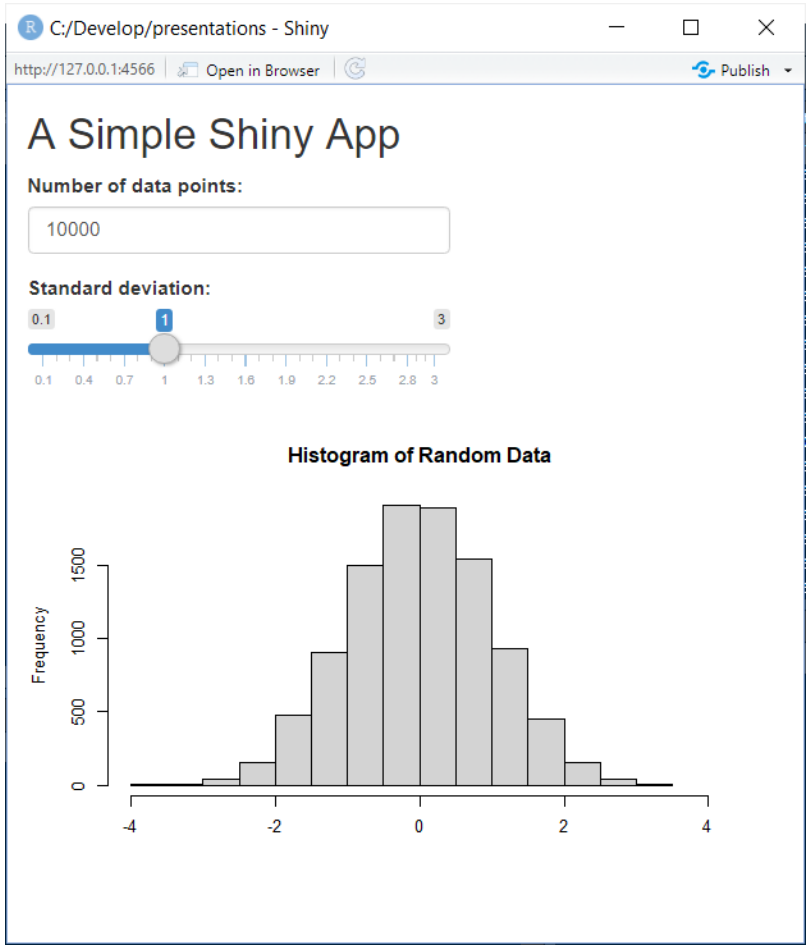
```
> ## App setup code that runs only once at startup.
> n_data <- 1e4
> std_dev <- 1.0
>
> ## Define the user interface
> inter_face <- shiny::fluidPage(
+   # Create numeric input for the number of data points.
+   numericInput('n_data', "Number of data points:", value=n_data),
+   # Create slider input for the standard deviation parameter.
+   sliderInput("std_dev", label="Standard deviation:",
+         min=0.1, max=3.0, value=std_dev, step=0.1),
+   # Render plot in a panel.
+   plotOutput("plo_t", height=300, width=500)
+ )  # end user interface
>
> ## Define the server function
> ser_ver <- function(input, output) {
+   output$plo_t <- shiny::renderPlot({
+     # Simulate the data
+     da_ta <- rnorm(input$n_data, sd=input$std_dev)
+     # Plot the data
+     par(mar=c(2, 4, 4, 0), oma=c(0, 0, 0, 0))
+     hist(da_ta, xlim=c(-4, 4), main="Histogram of Random Data")
+   })  # end renderPlot
+ }  # end ser_ver
>
> # Return a Shiny app object
> shiny::shinyApp(ui=inter_face, server=ser_ver)
```

The function `shiny::shinyApp()` creates a shiny app from a *shiny interface* and a *shiny server*.

Running Shiny Apps in *RStudio*

A *shiny app* can be run by pressing the "Run App" button in *RStudio*.

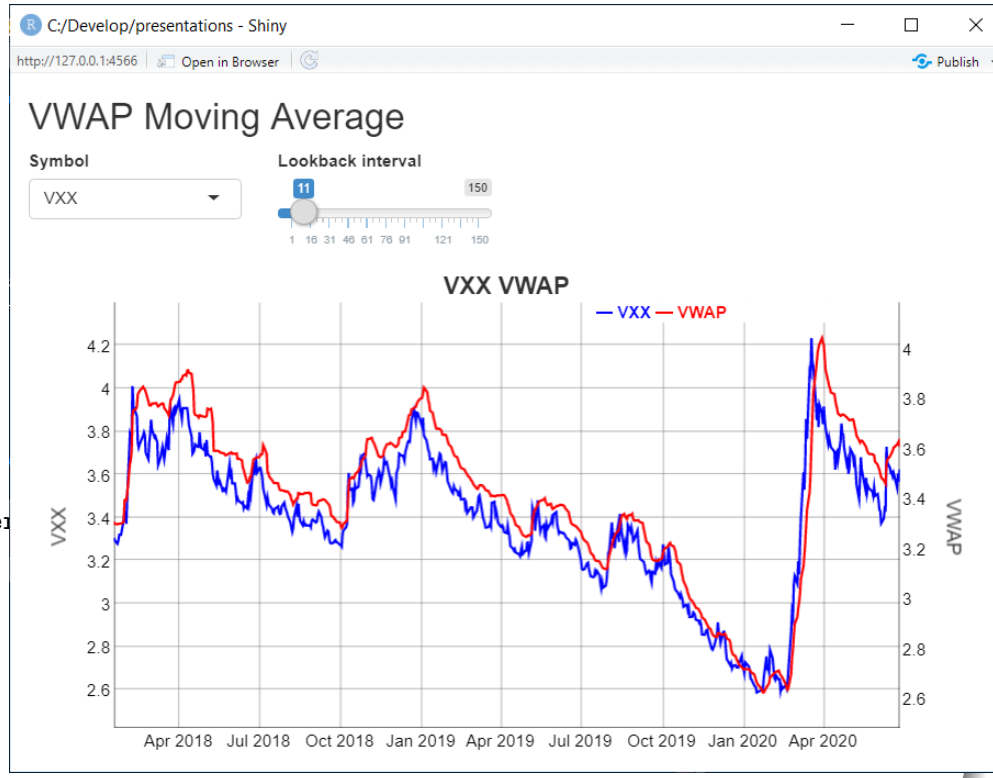
When the *shiny app* is run, the *shiny* commands are translated into *JavaScript* code, which creates a graphical user interface (GUI) with buttons, sliders, and boxes for data input, and also with the output plots and charts.



Positioning and Sizing Widgets Within the Shiny GUI

The functions `shiny::fluidRow()` and `shiny::column()` allow positioning and sizing widgets within the *shiny* GUI.

```
> ## Create elements of the user interface
> inter_face <- shiny::fluidPage(
+   titlePanel("VWAP Moving Average"),
+   # Create single row of widgets with two slider inputs
+   fluidRow(
+     # Input stock symbol
+     column(width=3, selectInput("sym_bol", label="Symbol",
+                           choices=sym_bols, selected=sym_bol)),
+     # Input look-back interval
+     column(width=3, sliderInput("look_back", label="Lookback inter
+                           min=1, max=150, value=11, step=1))
+   ),  # end fluidRow
+   # Create output plot panel
+   mainPanel(dygraphs::dygraphOutput("dy_graph"), width=12)
+ )  # end fluidPage interface
```



Shiny Apps With Reactive Expressions

The package *shiny* allows specifying reactive expressions which are evaluated only when their input data is updated.

Reactive expressions avoid performing unnecessary calculations.

If the reactive expression is invalidated (recalculated), then other expressions that depend on its output are also recalculated.

This way calculations cascade through the expressions that depend on each other.

The function `shiny::reactive()` transforms an expression into a reactive expression.

```
> ## Define the server function
> ser_ver <- shiny::shinyServer(function(input, output) {
+   # Get the close and volume data in a reactive environment
+   clos_e <- shiny::reactive({
+     # Get the data
+     oh_lc <- get(input$sym_bol, data_env)
+     clos_e <- log(quantmod::Cl(oh_lc))
+     vol_ume <- quantmod::Vo(oh_lc)
+     # Return the data
+     cbind(clos_e, vol_ume)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   v_wap <- shiny::reactive({
+     # Get model parameters from input argument
+     look_back <- input$look_back
+     # Calculate the VWAP indicator
+     clos_e <- clos_e()[, 1]
+     vol_ume <- clos_e()[, 2]
+     v_wap <- HighFreq::roll_sum(se_ries=clos_e*vol_ume, look_back=
+     volume_rolling <- HighFreq::roll_sum(se_ries=vol_ume, look_bac
+     v_wap <- v_wap/volume_rolling
+     v_wap[is.na(v_wap)] <- 0
+     # Return the plot data
+     da_ta <- cbind(clos_e, v_wap)
+     colnames(da_ta) <- c(input$sym_bol, "VWAP")
+     da_ta
+   })  # end reactive code
+
+   # Return the dygraph plot to output argument
+   output$dy_graph <- dygraphs::renderDygraph({
+     col_names <- colnames(v_wap())
+     dygraphs::dygraph(v_wap(), main=paste(col_names[1], "VWAP")) %
+ dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+ dySeries(name=col_names[1], axis="y", label=col_names[1], strokeW
+ dySeries(name=col_names[2], axis="y2", label=col_names[2], stroke
+   })  # end output plot
+ })  # end server code
```

Reactive Event Handlers

Event handlers are functions which evaluate expressions when an event occurs (like a button press).

The functions `shiny::observeEvent()` and `shiny::eventReactive()` are event handlers.

The function `shiny::eventReactive()` returns a value, while `shiny::observeEvent()` produces a side-effect, without returning a value.

The function `shiny::reactiveValues()` creates a list for storing reactive values, which can be updated by event handlers.

```
> ## Define the server function
> ser_ver <- shiny::shinyServer(function(input, output) {
+
+   # Create an empty list of reactive values.
+   value_s <- reactiveValues()
+
+   # Get input parameters from the user interface.
+   n_rows <- reactive({
+     # Add n_rows to list of reactive values.
+     value_s$n_rows <- input$n_rows
+     input$n_rows
+   })  # end reactive code
+
+   # Broadcast a message to the console when the button is pressed.
+   observeEvent(eventExpr=input$but_ton, handlerExpr={
+     cat("Input button pressed\n")
+   })  # end observeEvent
+
+   # Send the data when the button is pressed.
+   da_ta <- eventReactive(eventExpr=input$but_ton, valueExpr={
+     # eventReactive() executes on input$but_ton, but not on n_rows
+     cat("Sending", n_rows(), "rows of data\n")
+     da_ta <- head(mtcars, input$n_rows)
+     value_s$mpg <- mean(da_ta$mpg)
+     da_ta
+   })  # end eventReactive
+   #   da_ta
+
+   # Draw table of the data when the button is pressed.
+   observeEvent(eventExpr=input$but_ton, handlerExpr={
+     da_ta <- da_ta()
+     cat("Received", value_s$n_rows, "rows of data\n")
+     cat("Average mpg = ", value_s$mpg, "\n")
+     cat("Drawing table\n")
+     output$tabl_e <- renderTable(da_ta)
+   })  # end observeEvent
+
+ })  # end server code
>
```

Trade and Quote (*TAQ*) Data

High frequency data is typically formatted as either Trade and Quote (*TAQ*) data, or *Open-High-Low-Close* (*OHLC*) data.

Trade and Quote (*TAQ*) data contains intraday *trades* and *quotes* on exchange-traded stocks and futures.

TAQ data is often called *tick data*, with a *tick* being a row of data containing new *trades* or *quotes*.

The *TAQ* data is spaced irregularly in time, with data recorded each time a new trade or quote arrives.

Each row of *TAQ* data may contain the quote and trade prices, and the corresponding quote size or trade volume: *Bid.Price, Bid.Size, Ask.Price, Ask.Size, Trade.Price, Volume*.

TAQ data is often split into *trade* data and *quote* data.

```
> # Load package HighFreq
> library(HighFreq)
> # Or load the high frequency data file directly:
> symbol_s <- load("C:/Develop/R/HighFreq/data/hf_data.RData")

Error in readChar(con, 5L, useBytes = TRUE): cannot open the
connection

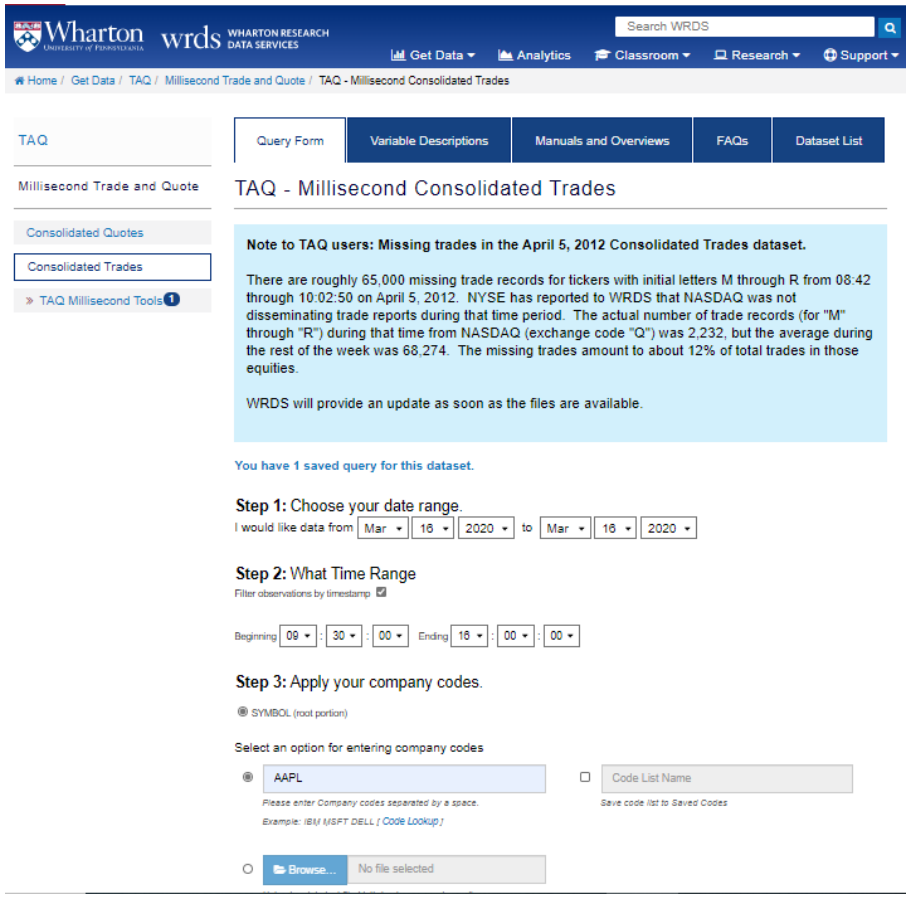
> head(SPY_TAQ)
                    Bid.Price Bid.Size Ask.Price Ask.Size Trade.Pric
2014-05-02 00:00:01       188        1       189       15          N
2014-05-02 08:00:01       188        1       189        5         18
2014-05-02 08:00:02       189        1       189        5          N
2014-05-02 08:01:13       188        1       189        5          N
2014-05-02 08:01:29       188        1       189        5          N
2014-05-02 08:01:52       189        2       189        5          N
> head(SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2008-01-02 09:31:00      147      147     147       147     591203
2008-01-02 09:32:00      147      147     147       147     385457
2008-01-02 09:33:00      147      147     147       147     343700
2008-01-02 09:34:00      147      147     147       147     863418
2008-01-02 09:35:00      147      147     147       147     457500
2008-01-02 09:36:00      147      147     147       147     416708
> tail(SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2014-05-19 15:56:00      189      189     189       189     321321
2014-05-19 15:57:00      189      189     189       189     299474
2014-05-19 15:58:00      189      189     189       189     261357
2014-05-19 15:59:00      189      189     189       189     435886
2014-05-19 16:00:00      189      189     189       189    1824185
2014-05-19 16:01:00      189      189     189       189      63920
```

Downloading *TAQ* Data From *WRDS*

TAQ data can be downloaded from the *WRDS TAQ* web page.

The *TAQ* data are at millisecond frequency, and are *consolidated* (combined) from the New York Stock Exchange *NYSE* and other exchanges.

The *WRDS TAQ* web page provides separately *trades* data and separately *quotes* data.



Reading *TAQ* Data From .csv Files

TAQ data stored in `.csv` files can be very large, so it's better to read it using the function `data.table::fread()` which is much faster than the function `read.csv()`.

Each *trade* or *quote* contributes a *tick* (row) of data, and the number of ticks can be very large (hundred of thousands per day, or more).

The function `strptime()` coerces `character` strings representing the date and time into `POSIXlt` *date-time* objects.

The argument `format="%H:%M:%OS"` allows the parsing of fractional seconds, for example `"15:59:59.989847074"`.

The function `as.POSIXct()` coerces objects into `POSIXct` *date-time* objects, with a `numeric` value representing the *moment of time* in seconds.

```
> library(rutils)
> # Read TAQ trade data from csv file
> ta_q <- data.table::fread(file="C:/Develop/data/xlk_tick_trades_20
> # Inspect the TAQ data
> ta_q
> class(ta_q)
> colnames(ta_q)
> sapply(ta_q, class)
> sym_bol <- ta_q$SYM_ROOT[1]
> # Create date-time index
> date_s <- paste(ta_q$DATE, ta_q$TIME_M)
> # Coerce date-time index to POSIXlt
> date_s <- strptime(date_s, "%Y%m%d %H:%M:%OS")
> class(date_s)
> # Display more significant digits
> # options("digits")
> options(digits=20, digits.secs=10)
> last(date_s)
> unclass(last(date_s))
> as.numeric(last(date_s))
> # Coerce date-time index to POSIXct
> date_s <- as.POSIXct(date_s)
> class(date_s)
> last(date_s)
> unclass(last(date_s))
> as.numeric(last(date_s))
> # Calculate the number of ticks per second
> n_secs <- as.numeric(last(date_s)) - as.numeric(first(date_s))
> NROW(ta_q)/(6.5*3600)
> # Select TAQ data columns
> ta_q <- ta_q[, .(price=PRICE, volume=SIZE)]
> # Add date-time index
> ta_q <- cbind(index=date_s, ta_q)
```

Microstructure Noise in High Frequency Data

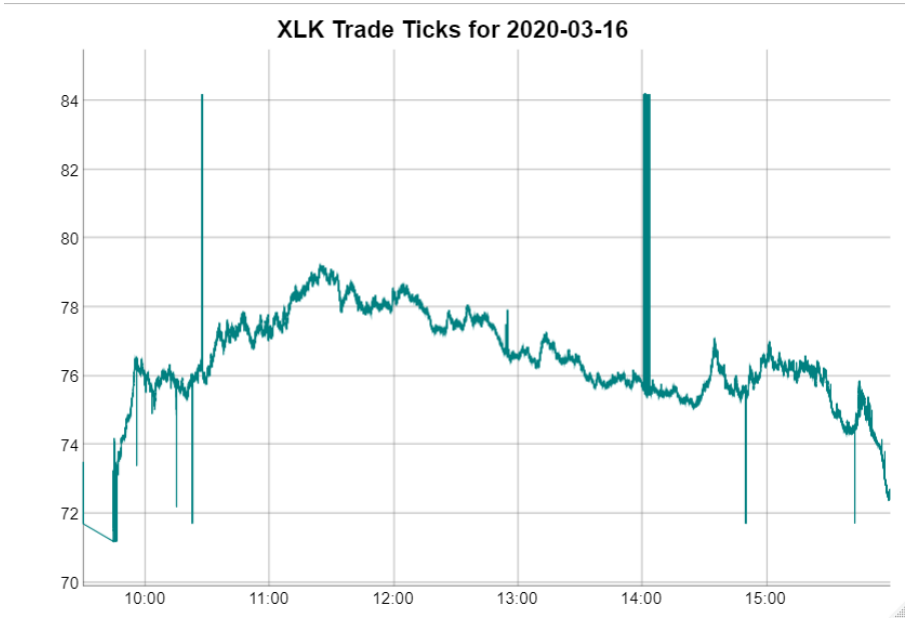
High frequency data contains *microstructure noise* in the form of *price jumps* and the *bid-ask bounce*.

Price jumps are single ticks with prices far away from the average.

Price jumps are often caused by data collection errors, but sometimes they represent actual very large lot trades.

The *bid-ask bounce* is the bouncing of traded prices between the bid and ask prices.

The *bid-ask bounce* creates an illusion of rapidly changing prices, while in fact the mid price is constant.




```
> # Coerce trade ticks to xts series
> x_ts <- xts::xts(ta_q[, .(price, volume)], ta_q$index)
> colnames(x_ts) <- paste(sym_bol, c("Close", "Volume"), sep=".")
> save(x_ts, file="C:/Develop/data/xlk_tick_trades_2020_03_16.RData"
> # Plot dygraph
> dygraphs::dygraph(x_ts$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16")
> # Plot in x11 window
> x11(width=6, height=5)
> quantmod::chart_Series(x=x_ts$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16")
```

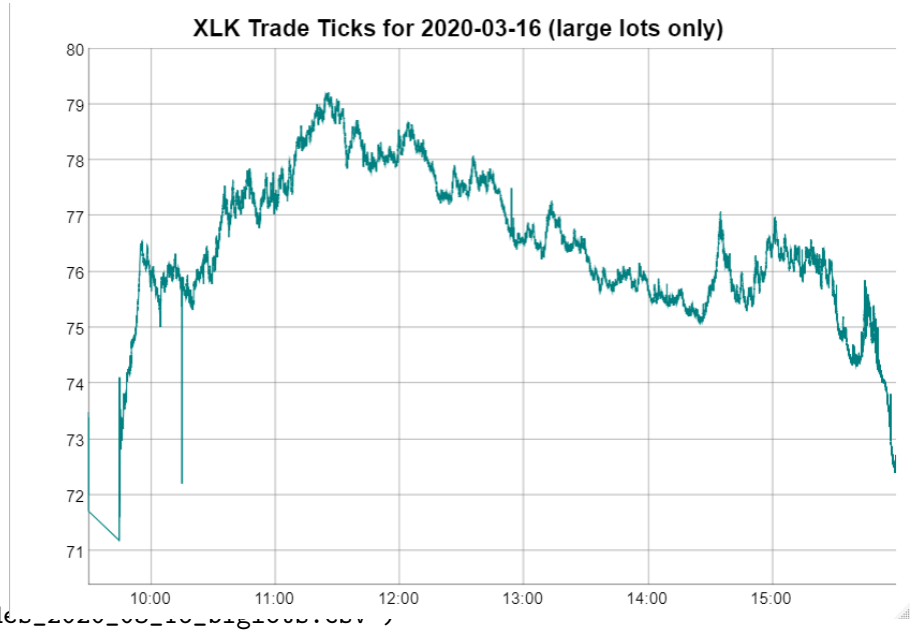
Removing Odd Lot Trades From *TAQ* Data

Most of the trade ticks are *odd lots* with a small volume of less than 100 shares.

The *odd lot* ticks are often removed to reduce the size of the *TAQ* data.

Selecting only the large lot trades reduces microstructure noise (price jumps, bid-ask bounce) in high frequency data.

```
> # Select the large lots greater than 100
> dim(ta_q)
> big_ticks <- ta_q[ta_q$volume > 100]
> dim(big_ticks)
> # Number of large lot ticks per second
> NROW(big_ticks)/(6.5*3600)
> # Save trade ticks with large lots
> data.table::fwrite(big_ticks, file="C:/Develop/data/xlk_tick_trades_2020_03_16_big_lots.csv")
> # Coerce trade prices to xts
> x_ts <- xts::xts(big_ticks[, .(price, volume)], big_ticks$index)
> colnames(x_ts) <- c("XLK.Close", "XLK.Volume")
```




```
> # Plot dygraph of the large lots
> dygraphs::dygraph(x_ts$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16 (large lots only)")
> # Plot the large lots
> x11(width=6, height=5)
> quantmod::chart_Series(x=x_ts$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16 (large lots only)")
```

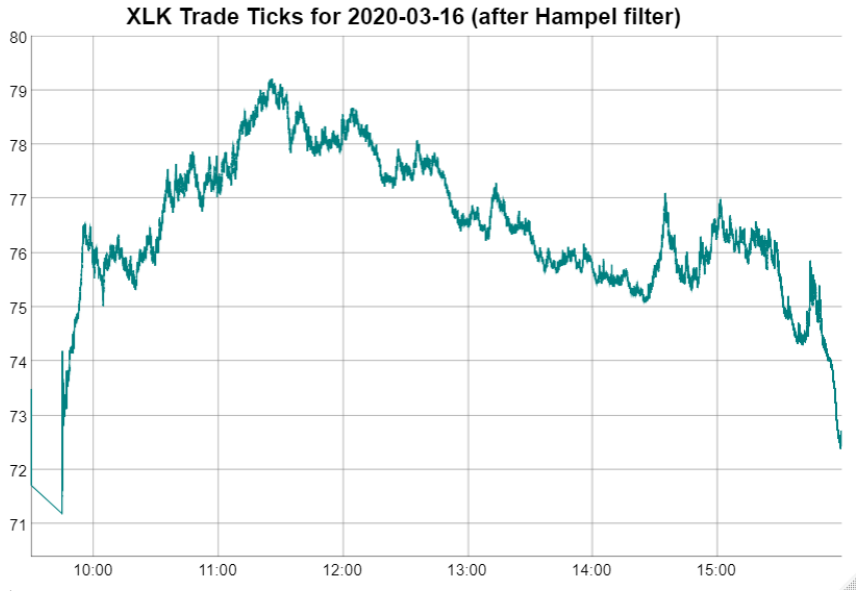
Removing Microstructure Noise From High Frequency Data

Microstructure noise can be removed from high frequency data by using a *Hampel filter*.

The *z-scores* are equal to the prices minus the median prices, divided by the median absolute deviation (*MAD*) of prices:

$$z_i = (p_i - \text{median}(\mathbf{p}))/\,\text{MAD}$$

If the *z-score* exceeds the *threshold value* then it's classified as an *outlier* (jump in prices).




```
> # Apply centered Hampel filter to remove price jumps
> win_dow <- 111
> half_window <- win_dow %/% 2
> medi_an <- TTR::runMedian(ta_q$price, n=win_dow)
> medi_an <- rutils::lag_it(medi_an, lagg=-half_window, pad_zeros=FA
> ma_d <- TTR::runMAD(ta_q$price, n=win_dow)
> ma_d <- rutils::lag_it(ma_d, lagg=-half_window, pad_zeros=FALSE)
> ma_d[1:half_window] <- 1
> ma_d[ma_d == 0] <- 1
> # Calculate Z-scores
> z_scores <- (ta_q$price - medi_an)/ma_d
> z_scores[is.na(z_scores)] <- 0
> z_scores[!is.finite(z_scores)] <- 0
> sum(is.na(z_scores))
> sum(!is.finite(z_scores))
> range(z_scores)
> mad(z_scores)
> hist(z_scores, breaks=2000, xlim=c(-5*mad(z_scores), 5*mad(z_scor
```

```
> # Remove price jumps with large z-scores
> thresh_old <- 3
> bad_ticks <- (abs(z_scores) > thresh_old)
> good_ticks <- ta_q[!bad_ticks]
> # Calculate number of price jumps
> sum(bad_ticks)/NROW(z_scores)
> # Coerce trade prices to xts
> x_ts <- xts::xts(good_ticks[, .(price, volume)], good_ticks$index)
> colnames(x_ts) <- c("XLK.Close", "XLK.Volume")
> # Plot dygraph of the clean lots
> dygraphs::dygraph(x_ts$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
> # Plot the large lots
> x11(width=6, height=5)
> quantmod::chart_Series(x=x_ts$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
```

Aggregating *TAQ* Data to *OHLC*

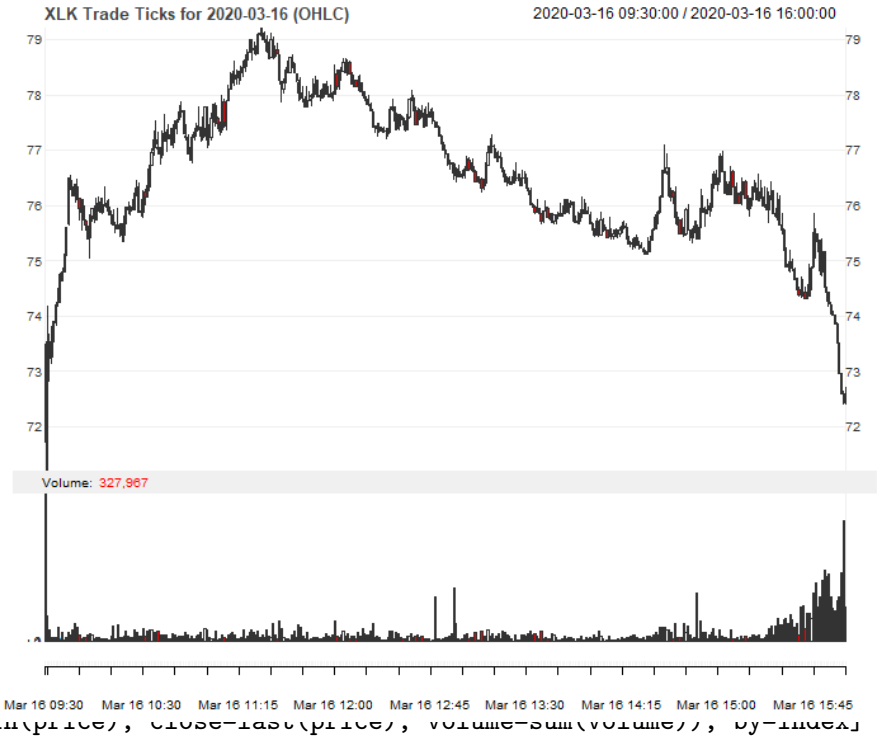
The *data table* columns can be *aggregated* over categories (factors) defined by one or more columns passed to the "by" argument.

Multiple *data table* columns can be referenced by passing a list of names specified by the dot `.()` operator.

The function `round.POSIXt()` rounds date-time objects to seconds, minutes, hours, days, months or years.

The function `as.POSIXct()` coerces objects to class `POSIXct`.

```
> # Round time index to seconds
> good_ticks[, index := as.POSIXct(round.POSIXt(index, "secs"))]
> # Aggregate to OHLC by seconds
> oh_lc <- good_ticks[, .(open=first(price), high=max(price), low=m
> # Round time index to minutes
> good_ticks[, index := as.POSIXct(round.POSIXt(index, "mins"))]
> # Aggregate to OHLC by minutes
> oh_lc <- good_ticks[, .(open=first(price), high=max(price), low=min(price), close=last(price), volume=sum(volume)), by=index]
```




```
> # Coerce OHLC prices to xts
> x_ts <- xts::xts(oh_lc[, -"index"], oh_lc$index)
> # Plot dygraph of the OHLC prices
> dygraphs::dygraph(x_ts[, -5], main="XLK Trade Ticks for 2020-03-16
+   dyCandlestick()
> # Plot the OHLC prices
> x11(width=6, height=5)
> quantmod::chart_Series(x=x_ts, TA="add_Vo()",
+   name="XLK Trade Ticks for 2020-03-16 (OHLC)")
```

Open-High-Low-Close (*OHLC*) Data

Open-High-Low-Close (*OHLC*) data contains intraday trade prices and trade volumes.

OHLC data is evenly spaced in time, with each row containing the *Open, High, Low, Close* prices, and the trade *Volume*, recorded over the past time interval (called a *bar* of data).

The *Open* and *Close* prices are the first and last trade prices recorded in the time bar.

The *High* and *Low* prices are the highest and lowest trade prices recorded in the time bar.

The *Volume* is the total trading volume recorded in the time bar.

The *OHLC* data format provides a way of efficiently compressing *TAQ* data, while preserving information about price levels, volatility (range), and trading volumes.

In addition, evenly spaced *OHLC* data allows for easier analysis of multiple time series, since the prices for different assets are given at the same moments in time.

```
> # Load package HighFreq
> library(HighFreq)
> head(SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2008-01-02 09:31:00      147      147     147       147     591203
2008-01-02 09:32:00      147      147     147       147     385457
2008-01-02 09:33:00      147      147     147       147     343700
2008-01-02 09:34:00      147      147     147       147     863418
2008-01-02 09:35:00      147      147     147       147     457500
2008-01-02 09:36:00      147      147     147       147     416708
```

Plotting High Frequency *OHLC* Data

Aggregating high frequency *TAQ* data into *OHLC* format with lower periodicity allows for data compression while maintaining some information about volatility.

```
> # Load package HighFreq
> library(HighFreq)
> # Define sym_bol
> sym_bol <- "SPY"
> # Load OHLC data
> output_dir <- "C:/Develop/data/hfreq/scrub/"
> sym_bol <- load(
+   file.path(output_dir,
+       paste0(sym_bol, ".RData")))
> inter_val <-
+   "2013-11-11 09:30:00/2013-11-11 10:30:00"
> chart_Series(SPY[inter_val], name=sym_bol)
```

The package *HighFreq* contains both *TAQ* data and *Open-High-Low-Close* (*OHLC*) data.

If you are not able to install package *HighFreq* then download the file `hf_data.RData` from NYU Classes and load it.

Package *HighFreq* for Managing High Frequency Data

The package *HighFreq* contains functions for managing high frequency time series data, such as:

- converting *TAQ* data to *OHLC* format,
- chaining and joining time series,
- scrubbing bad data,
- managing time zones and alligning time indices,
- aggregating data to lower frequency (periodicity),
- calculating rolling aggregations (VWAP, Hurst exponent, etc.),
- calculating seasonality aggregations,
- estimating volatility, skewness, and higher moments,

```
> # Install package HighFreq from github
> devtools::install_github(repo="algoquant/HighFreq")
> # Load package HighFreq
> library(HighFreq)
> # Get documentation for package HighFreq
> # Get short description
> packageDescription(HighFreq)
> # Load help page
> help(package=HighFreq)
> # List all datasets in HighFreq
> data(package=HighFreq)
> # List all objects in HighFreq
> ls("package:HighFreq")
> # Remove HighFreq from search path
> detach("package:HighFreq")
```

Datasets in Package *HighFreq*

The package *HighFreq* contains several high frequency time series, in *xts* format, stored in a file called `hf_data.RData`:

- a time series called `SPY_TAQ`, containing a single day of *TAQ* data for the *SPY* ETF.
- three time series called `SPY`, `TLT`, and `VXX`, containing intraday 1-minute *OHLC* data for the *SPY*, *TLT*, and *VXX* ETFs.

Even after the *HighFreq* package is loaded, its datasets aren't loaded into the workspace, so they aren't listed in the workspace.

That's because the datasets in package *HighFreq* are set up for *lazy loading*, which means they can be called as if they were loaded, even though they're not loaded into the workspace.

The datasets in package *HighFreq* can be loaded into the workspace using the function `data()`.

The data is set up for *lazy loading*, so it doesn't require calling `data(hf_data)` to load it into the workspace before calling it.

```
> # Load package HighFreq
> library(HighFreq)
> # You can see SPY when listing objects in HighFreq
> ls("package:HighFreq")
> # You can see SPY when listing datasets in HighFreq
> data(package=HighFreq)
> # But the SPY dataset isn't listed in the workspace
> ls()
> # HighFreq datasets are lazy loaded and available when needed
> head(SPY)
> # Load all the datasets in package HighFreq
> data(hf_data)
> # HighFreq datasets are now loaded and in the workspace
> head(SPY)
```

Distribution of High Frequency Returns

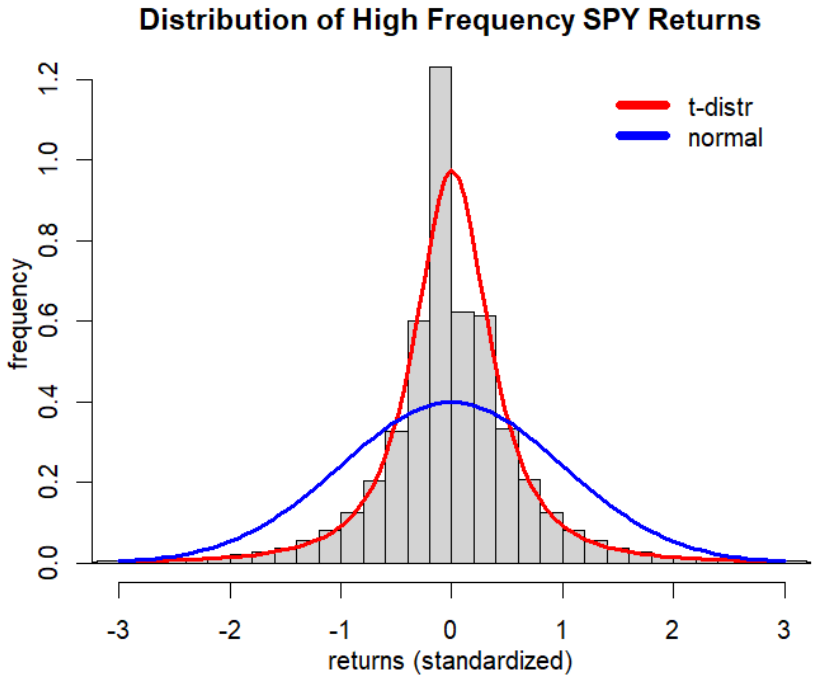
High frequency returns exhibit *large negative skewness* and *very large kurtosis* (leptokurtosis), or fat tails.

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

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

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

```
> # SPY percentage returns
> oh_lc <- HighFreq::SPY
> n_rows <- NROW(oh_lc)
> clos_e <- log(quantmod::Cl(oh_lc))
> re_turns <- rutils::diff_it(clos_e)
> colnames(re_turns) <- "SPY"
> # Standardize raw returns to make later comparisons
> re_turns <- (re_turns - mean(re_turns))/sd(re_turns)
> # Calculate moments and perform normality test
> sapply(c(var=2, skew=3, kurt=4),
+   function(x) sum(re_turns^x)/n_rows)
> tseries::jarque.bera.test(re_turns)
> # Fit SPY returns using MASS::fitdistr()
> optim_fit <- MASS::fitdistr(re_turns, densfun="t", df=2)
> lo_cation <- optim_fit$estimate[1]
> scal_e <- optim_fit$estimate[2]
```




```
> # Plot histogram of SPY returns
> histo_gram <- hist(re_turns, col="lightgrey", mgp=c(2, 1, 0),
+   xlab="returns (standardized)", ylab="frequency", xlim=c(-3, 3),
+   breaks=1e3, freq=FALSE, main="Distribution of High Frequency SPY
> # lines(density(re_turns, bw=0.2), lwd=3, col="blue")
> # Plot t-distribution function
> curve(expr=dt((x-lo_cation)/scal_e, df=2)/scal_e,
+ type="l", lwd=3, col="red", add=TRUE)
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(re_turns),
+   sd=sd(re_turns)), add=TRUE, lwd=3, col="blue")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("t-distr", "normal"),
+   lwd=6, lty=1, col=c("red", "blue"))
```

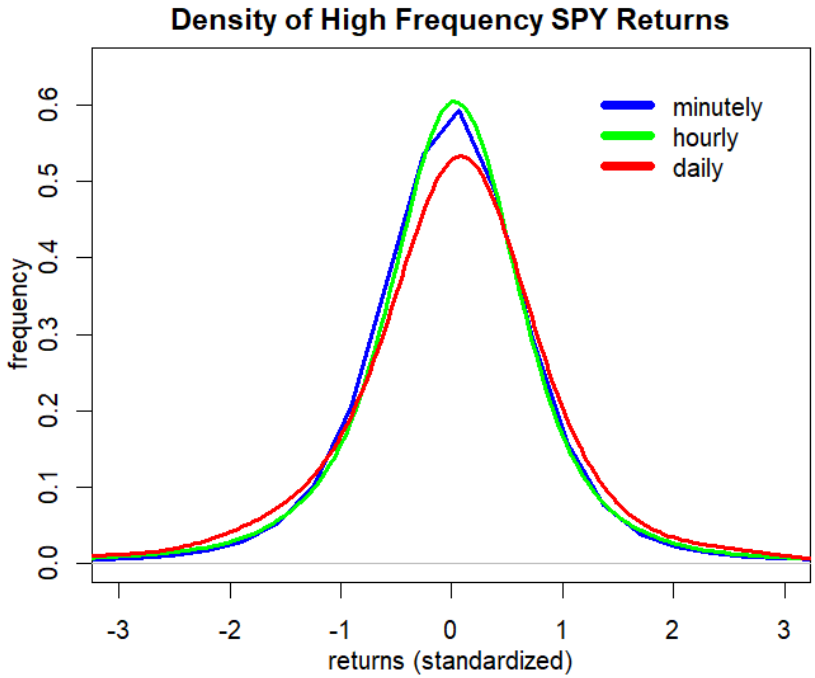
Distribution of Aggregated High Frequency Returns

The distribution of returns depends on the sampling frequency.

High frequency returns aggregated to a lower periodicity become less negatively skewed and less fat tailed, and closer to the normal distribution.

The function `xts::to.period()` converts a time series to a lower periodicity (for example from hourly to daily periodicity).

```
> # Hourly SPY percentage returns
> clos_e <- log(Cl(xts::to.period(x=oh_lc, period="hours")))
> hour_ly <- rutils::diff_it(clos_e)
> hour_ly <- (hour_ly - mean(hour_ly))/sd(hour_ly)
> # Daily SPY percentage returns
> clos_e <- log(Cl(xts::to.period(x=oh_lc, period="days")))
> dai_ly <- rutils::diff_it(clos_e)
> dai_ly <- (dai_ly - mean(dai_ly))/sd(dai_ly)
> # Calculate moments
> sapply(list(minutely=re_turns, hourly=hour_ly, daily=dai_ly),
+  function(rets) {
+    sapply(c(var=2, skew=3, kurt=4),
+           function(x) mean(rets^x))
+ })  # end sapply
```



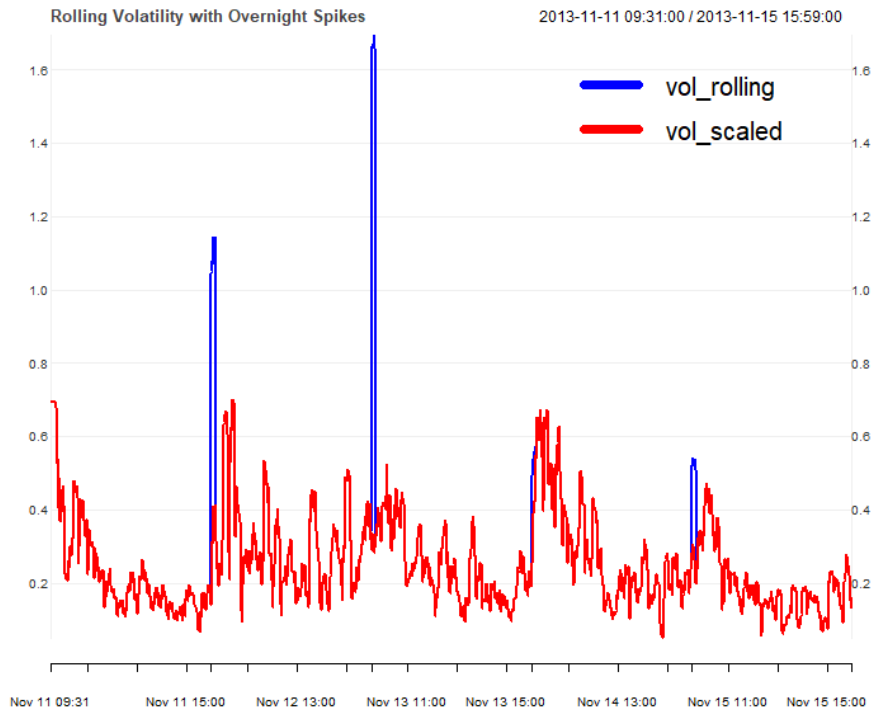

```
> # Plot densities of SPY returns
> plot(density(re_turns, bw=0.4), xlim=c(-3, 3),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of High Frequency SPY Returns")
> lines(density(hour_ly, bw=0.4), lwd=3, col="green")
> lines(density(dai_ly, bw=0.4), lwd=3, col="red")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("minutely", "hourly", "daily"),
+   lwd=6, lty=1, col=c("blue", "green", "red"))
```

Estimating Rolling Volatility of High Frequency Returns

The volatility of high frequency returns can be inflated by large overnight returns.

The large overnight returns can be scaled down by dividing them by the overnight time interval.

```
> # Calculate rolling volatility of SPY returns
> ret_2013 <- re_turns["2013-11-11/2013-11-15"]
> # Calculate rolling volatility
> look_back <- 11
> end_p <- seq_along(ret_2013)
> start_p <- c(rep_len(1, look_back-1),
+   end_p[1:(NROW(end_p)-look_back+1)])
> end_p[end_p < look_back] <- look_back
> vol_rolling <- sapply(seq_along(end_p),
+   function(it) sd(ret_2013[start_p[it]:end_p[it]]))
> vol_rolling <- xts::xts(vol_rolling, index(ret_2013))
> # Extract time intervals of SPY returns
> in_dex <- c(60, diff(xts::.index(ret_2013)))
> head(in_dex)
> table(in_dex)
> # Scale SPY returns by time intervals
> ret_2013 <- 60*ret_2013/in_dex
> # Calculate scaled rolling volatility
> vol_scaled <- sapply(seq_along(end_p),
+   function(it) sd(ret_2013[start_p[it]:end_p[it]]))
> vol_rolling <- cbind(vol_rolling, vol_scaled)
> vol_rolling <- na.omit(vol_rolling)
> sum(is.na(vol_rolling))
> sapply(vol_rolling, range)
```




```
> # Plot rolling volatility
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue", "red")
> chart_Series(vol_rolling, theme=plot_theme,
+       name="Rolling Volatility with Overnight Spikes")
> legend("topright", legend=colnames(vol_rolling),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

The Bid-ask Bounce of High Frequency Prices

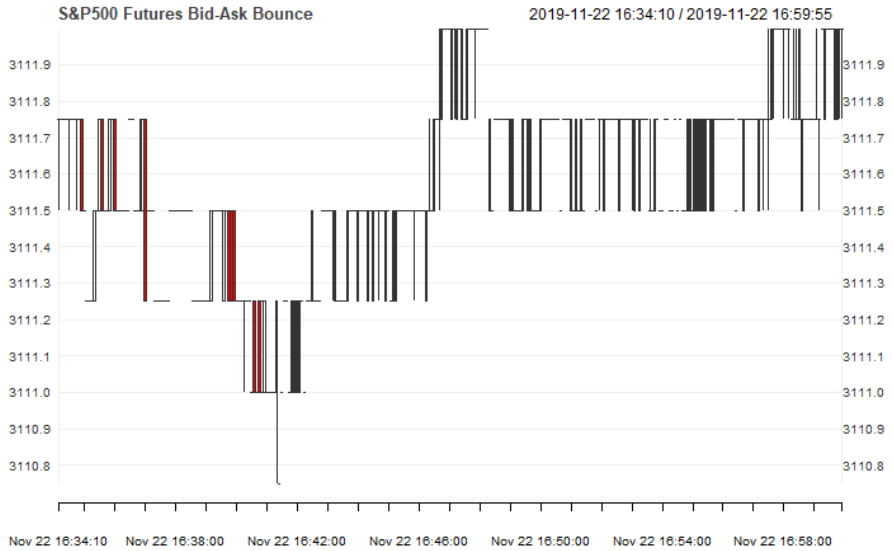
The *bid-ask bounce* is the bouncing of traded prices between the bid and ask prices.

The *bid-ask bounce* is prominent at very high frequency time scales or in periods of low volatility.

The *bid-ask bounce* creates an illusion of rapidly changing prices, while in fact the mid price is constant.

The *bid-ask bounce* produces very high realized volatility and the appearance of mean reversion (negative autocorrelation), that isn't tradeable for most traders.

```
> price_s <- read.zoo(file="C:/Develop/lecture_slides/data/bid_ask_t
+   header=TRUE, sep=",")
> price_s <- as.xts(price_s)
> x11(width=6, height=4)
> par(mar=c(2, 2, 0, 0), oma=c(1, 1, 0, 0))
> chart_Series(x=price_s, name="S&P500 Futures Bid-Ask Bounce")
```



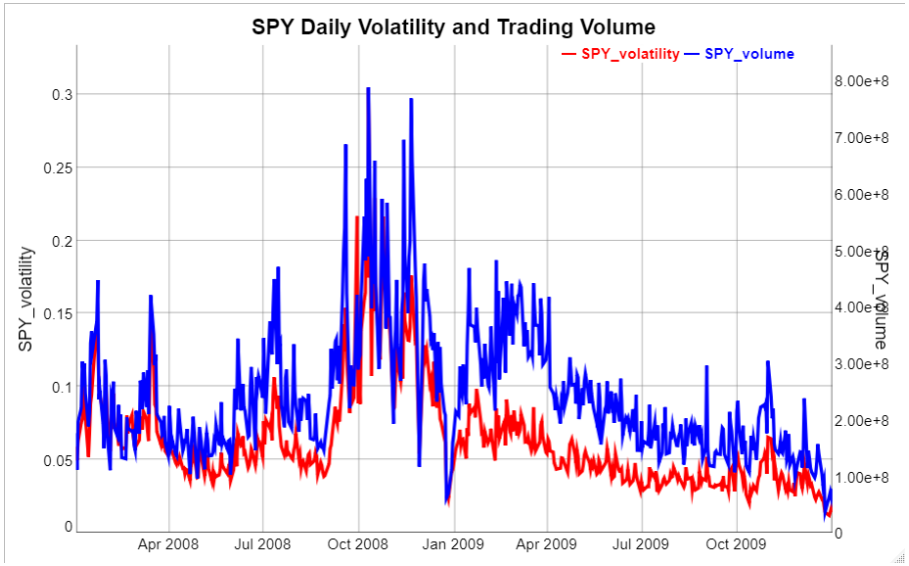
Daily Volume and Volatility of High Frequency Returns

Trading volumes typically rise together with market price volatility.

The function `apply.daily()` from package `xts` applies functions to time series over daily periods.

The function `calc_var_ohlc()` from package `HighFreq` calculates the variance of an *OHLC* time series using range estimators.

```
> # Volatility of SPY
> sqrt(HighFreq::calc_var_ohlc(oh_lc))
> # Daily SPY volatility and volume
> vol_daily <- sqrt(xts::apply.daily(oh_lc, FUN=calc_var_ohlc))
> colnames(vol_daily) <- ("SPY_volatility")
> vol_ume <- quantmod::Vo(oh_lc)
> volume_daily <- xts::apply.daily(vol_ume, FUN=sum)
> colnames(volume_daily) <- ("SPY_volume")
> # Plot SPY volatility and volume
> da_ta <- cbind(vol_daily, volume_daily)["2008/2009"]
> col_names <- colnames(da_ta)
> dygraphs::dygraph(da_ta,
+   main="SPY Daily Volatility and Trading Volume") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=3) %>%
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=3)
```

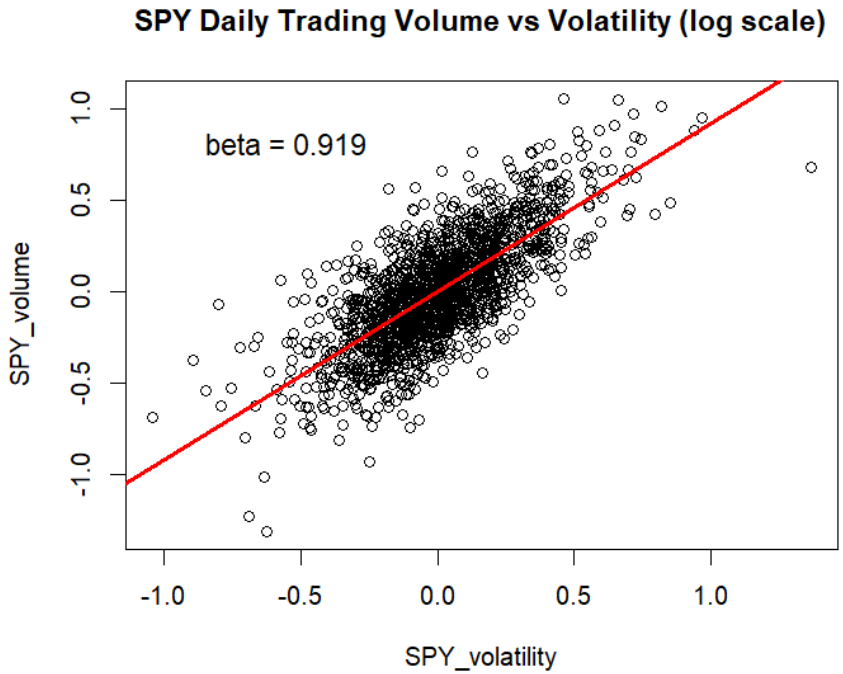


Beta of Volume vs Volatility of High Frequency Returns

As a general empirical rule, the *trading volume* υ in a given time period is roughly proportional to the *volatility* of the returns σ: $\upsilon \propto \sigma$.

The regression of the *log trading volume* versus the *log volatility* fails the *Durbin-Watson test* for the autocorrelation of residuals.

But the regression of the *differences* passes the *Durbin-Watson test*.

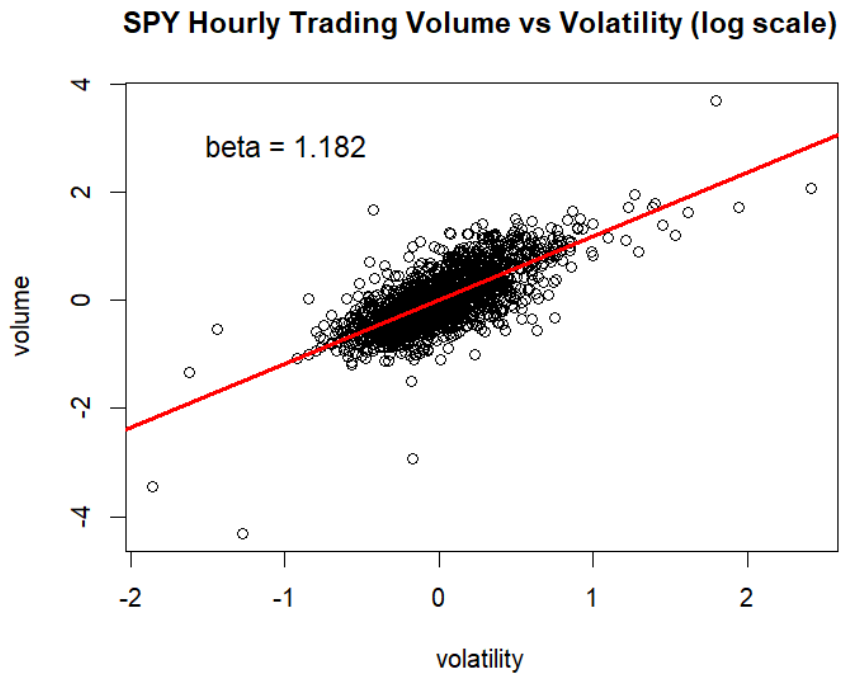



```
> # Regress log of daily volume vs volatility
> da_ta <- log(cbind(volume_daily, vol_daily))
> col_names <- colnames(da_ta)
> data_frame <- as.data.frame(da_ta)
> for_mula <- as.formula(paste(col_names, collapse="~"))
> mod_el <- lm(for_mula, data=data_frame)
> # Durbin-Watson test for autocorrelation of residuals
> lmtest::dwtest(mod_el)
> # Regress diff log of daily volume vs volatility
> data_frame <- as.data.frame(rutils::diff_it(da_ta))
> mod_el <- lm(for_mula, data=data_frame)
> lmtest::dwtest(mod_el)
> summary(mod_el)
> plot(for_mula, data=data_frame, main="SPY Daily Trading Volume vs Volatility (log scale)")
> abline(mod_el, lwd=3, col="red")
> mtext(paste("beta =", round(coef(mod_el)[2], 3)), cex=1.2, lwd=3, side=2, las=2, adj=(-0.5), padj=(-7))
```

Beta of Hourly Trading Volume vs Volatility

Hourly aggregations of high frequency data also support the rule that the *trading volume* is roughly proportional to the *volatility* of the returns: $\upsilon \propto \sigma$.

```
> # 60 minutes of data in look_back interval
> look_back <- 60
> vol_2013 <- vol_ume["2013"]
> ret_2013 <- re_turns["2013"]
> # Define end points with beginning stub
> n_rows <- NROW(ret_2013)
> n_agg <- n_rows %/% look_back
> end_p <- n_rows-look_back*n_agg + (0:n_agg)*look_back
> start_p <- c(1, end_p[1:(NROW(end_p)-1)])
> # Calculate SPY volatility and volume
> da_ta <- sapply(seq_along(end_p), function(it) {
+   point_s <- start_p[it]:end_p[it]
+   c(volume=sum(vol_2013[point_s]),
+     volatility=sd(ret_2013[point_s]))
+ })  # end sapply
> da_ta <- t(da_ta)
> da_ta <- rutils::diff_it(log(da_ta))
> data_frame <- as.data.frame(da_ta)
```




```
> for_mula <- as.formula(paste(colnames(da_ta), collapse="~"))
> mod_el <- lm(for_mula, data=data_frame)
> lmtest::dwtest(mod_el)
> summary(mod_el)
> plot(for_mula, data=data_frame,
+      main="SPY Hourly Trading Volume vs Volatility (log scale)")
> abline(mod_el, lwd=3, col="red")
> mtext(paste("beta =", round(coef(mod_el)[2], 3)), cex=1.2, lwd=3,
```

High Frequency Returns in Trading Time

The *trading time* (volume clock) is the time measured by the level of *trading volume*, with the *volume clock* running faster in periods of higher *trading volume*.

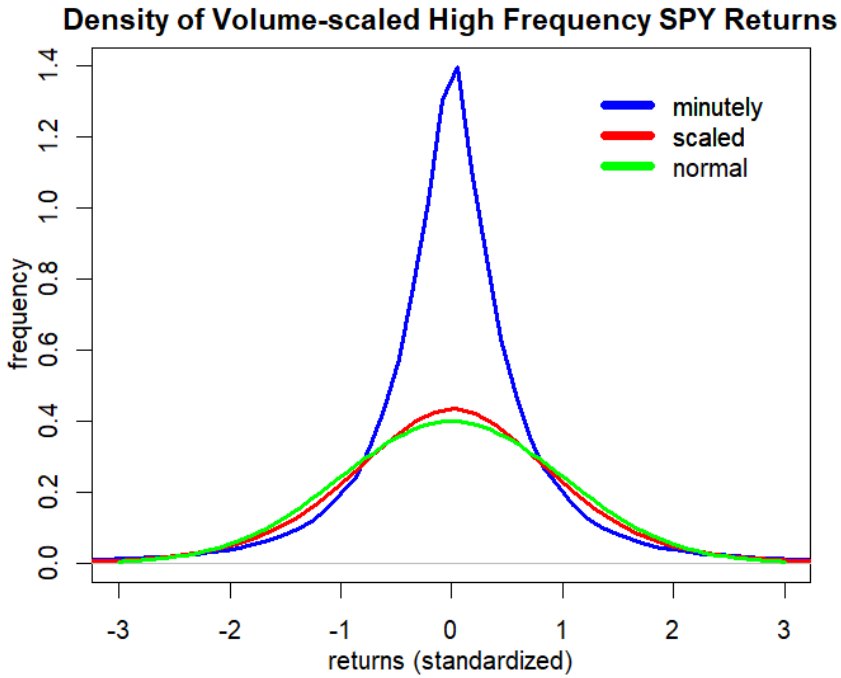
The time-dependent volatility of high frequency returns (*heteroskedasticity*) produces their *leptokurtosis* (large kurtosis, or fat tails).

The returns can be divided by the *trading volume* to obtain scaled returns over equal trading volumes.

But the returns should not be divided by very small volumes below a certain threshold.

The scaled returns have a smaller *skewness* and *kurtosis*, and they also have even higher autocorrelations than unscaled returns.

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




```
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> # Plot densities of SPY returns
> plot(density(re_turns), xlim=c(-3, 3),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of Volume-scaled High Frequency SPY Returns")
> lines(density(rets_scaled, bw=0.4), lwd=3, col="red")
> curve(expr=dnorm, add=TRUE, lwd=3, col="green")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("minutely", "scaled", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Autocorrelations of High Frequency Returns

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 *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.

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

For *scaled minutely SPY* returns, the *Ljung-Box* statistic is even larger, so its autocorrelations are even more statistically significant.

SPY returns aggregated to longer time intervals are less autocorrelated.

```
> # Ljung-Box test for minutely SPY returns
> Box.test(re_turns, lag=10, type="Ljung")
> # Ljung-Box test for daily SPY returns
> Box.test(dai_ly, lag=10, type="Ljung")
> # Ljung-Box test statistics for scaled SPY returns
> sapply(list(re_turns=re_turns, rets_scaled=rets_scaled),
+   function(rets) {
+     Box.test(rets, lag=10, type="Ljung")$statistic
+ })  # end sapply
> # Ljung-Box test statistics for aggregated SPY returns
> sapply(list(minutely=re_turns, hourly=hour_ly, daily=dai_ly),
+   function(rets) {
+     Box.test(rets, lag=10, type="Ljung")$statistic
+ })  # end sapply
```

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

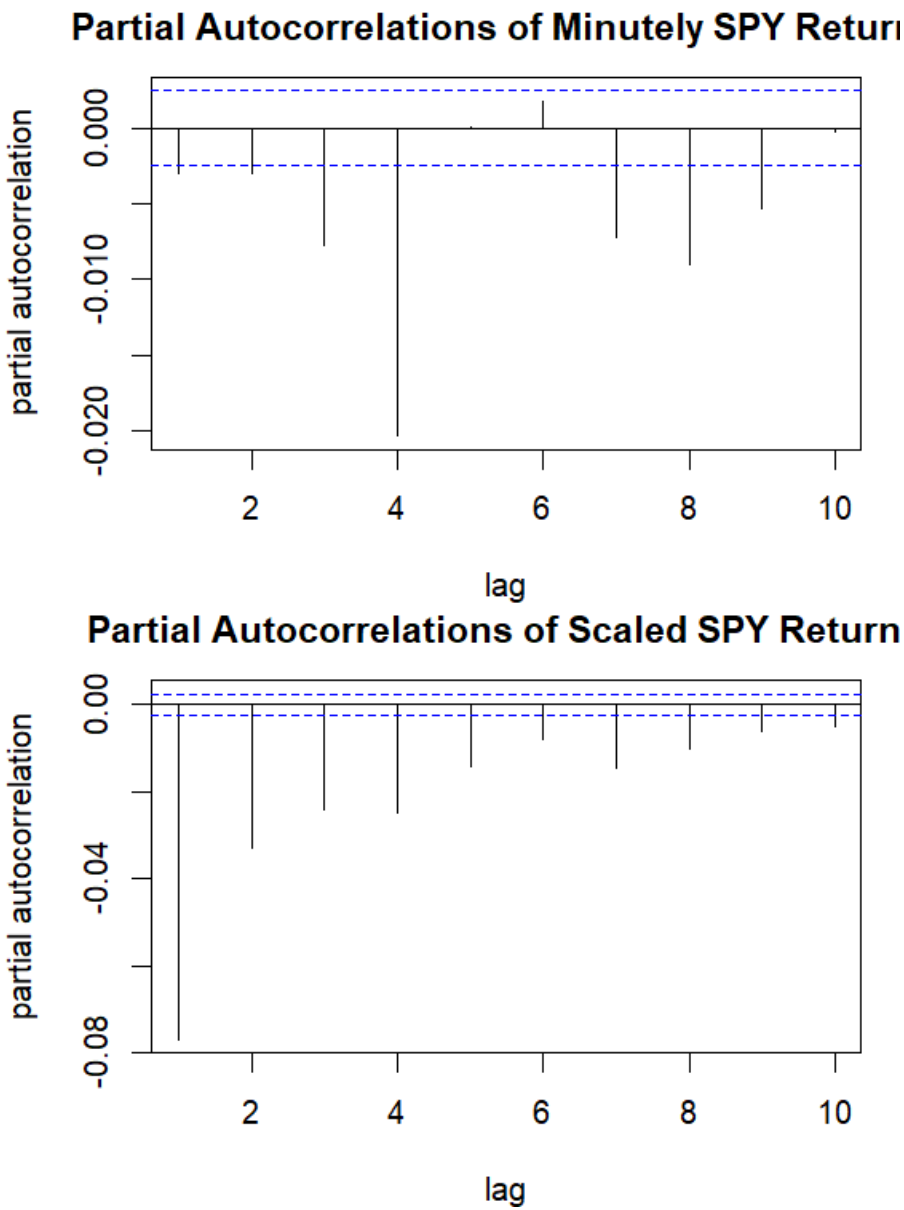
As the returns are aggregated to a lower periodicity, they become less autocorrelated, with daily returns having almost insignificant autocorrelations.

Partial Autocorrelations of High Frequency Returns

High frequency minutely *SPY* returns have statistically significant negative autocorrelations.

SPY returns *scaled* by the trading volumes have even more significant negative autocorrelations.

```
> # Set plot parameters
> x11(width=6, height=8)
> par(mar=c(4, 4, 2, 1), oma=c(0, 0, 0, 0))
> layout(matrix(c(1, 2), ncol=1), widths=c(6, 6), heights=c(4, 4))
> # Plot the partial autocorrelations of minutely SPY returns
> pa_cf <- pacf(as.numeric(re_turns), lag=10,
+       xlab="lag", ylab="partial autocorrelation", main="")
> title("Partial Autocorrelations of Minutely SPY Returns", line=1)
> # Plot the partial autocorrelations of scaled SPY returns
> pacf_scaled <- pacf(as.numeric(rets_scaled), lag=10,
+       xlab="lag", ylab="partial autocorrelation", main="")
> title("Partial Autocorrelations of Scaled SPY Returns", line=1)
> # Calculate the sums of partial autocorrelations
> sum(pa_cf$acf)
> sum(pacf_scaled$acf)
```



Market Liquidity, Trading Volume and Volatility

Market illiquidity is defined as the market price impact resulting from supply-demand imbalance.

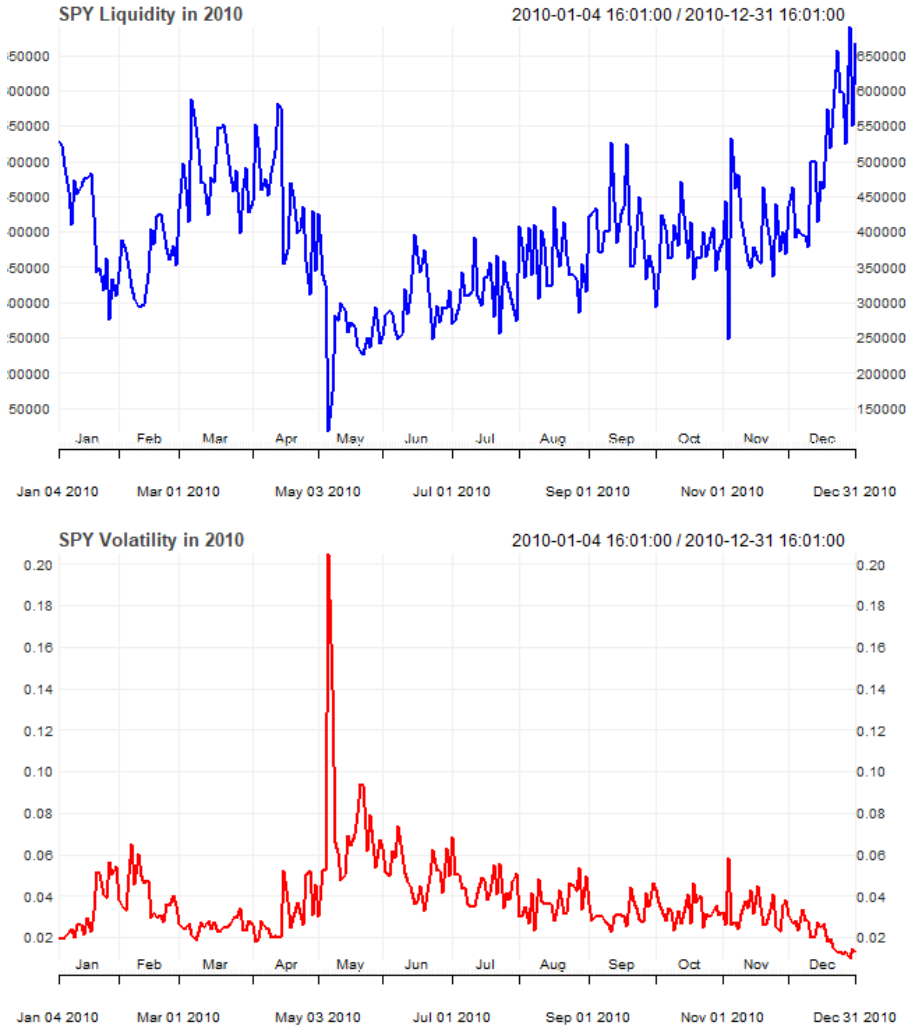
Market liquidity $\mathcal{L}$ is proportional to the square root of the *trading volume* υ divided by the price volatility σ:

$$\mathcal{L} \sim \frac{\sqrt{\upsilon}}{\sigma}$$

Market illiquidity spiked during the May 6, 2010 *flash crash*.

Research suggests that market crashes are caused by declining market liquidity:
Donier et al., Why Do Markets Crash?

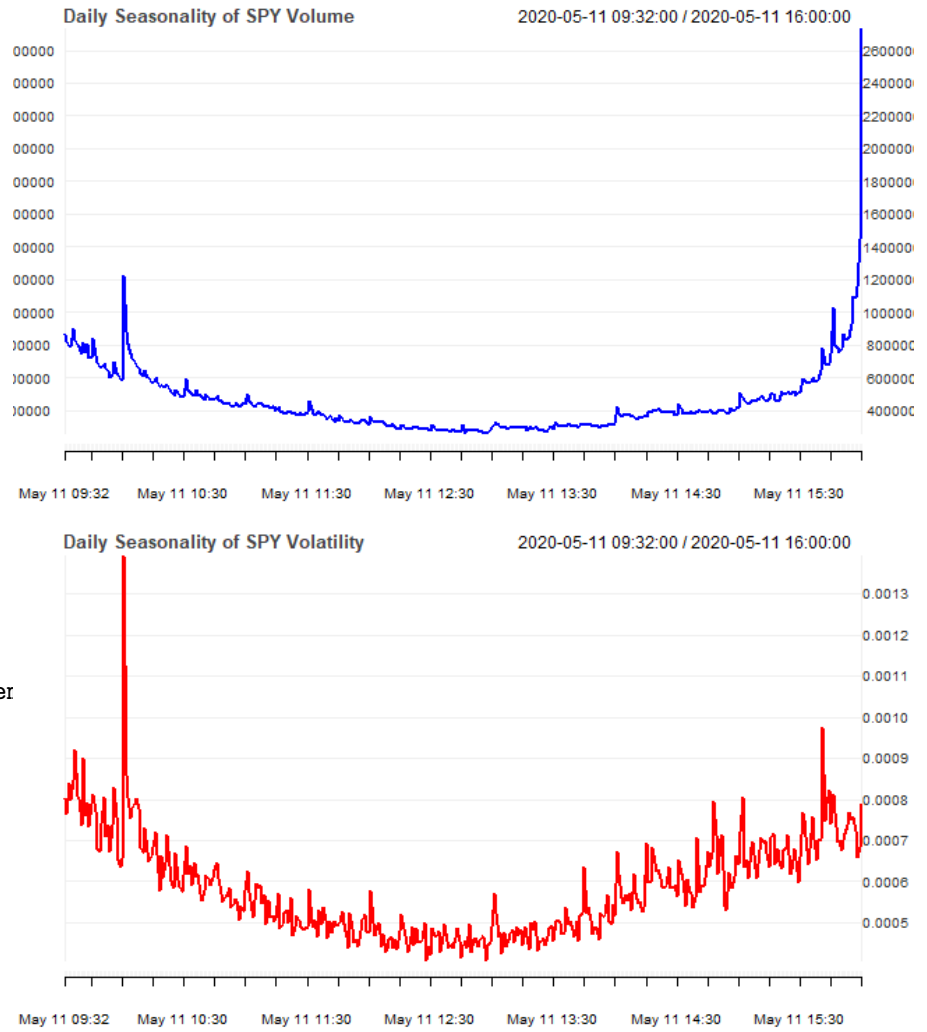
```
> # Calculate market illiquidity
> liquidi_ty <- sqrt(volume_daily)/vol_daily
> # Plot market illiquidity
> x11(width=6, height=7) ; par(mfrow=c(2, 1))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> chart_Series(liquidi_ty["2010"], theme=plot_theme,
+   name="SPY Liquidity in 2010", plot=FALSE)
> plot_theme$col$line.col <- c("red")
> chart_Series(vol_daily["2010"],
+   theme=plot_theme, name="SPY Volatility in 2010")
```



Daily Seasonality of Volume and Volatility

The volatility and trading volumes are typically higher at the beginning and end of the trading sessions.

```
> # Calculate intraday time index with hours and minutes
> date_s <- format(zoo::index(re_turns), "%H:%M")
> # Aggregate the mean volume
> volume_agg <- tapply(X=vol_ume, INDEX=date_s, FUN=mean)
> volume_agg <- drop(volume_agg)
> # Aggregate the mean volatility
> vol_agg <- tapply(X=re_turns^2, INDEX=date_s, FUN=mean)
> vol_agg <- sqrt(drop(vol_agg))
> # Coerce to xts
> intra_day <- as.POSIXct(paste(Sys.Date(), names(volume_agg)))
> volume_agg <- xts::xts(volume_agg, intra_day)
> vol_agg <- xts::xts(vol_agg, intra_day)
> # Plot seasonality of volume and volatility
> x11(width=6, height=7) ; par(mfrow=c(2, 1))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> chart_Series(volume_agg[c(-1, -NROW(volume_agg))], theme=plot_ther
+   name="Daily Seasonality of SPY Volume", plot=FALSE)
> plot_theme$col$line.col <- c("red")
> chart_Series(vol_agg[c(-1, -NROW(vol_agg))], theme=plot_theme,
+   name="Daily Seasonality of SPY Volatility")
```

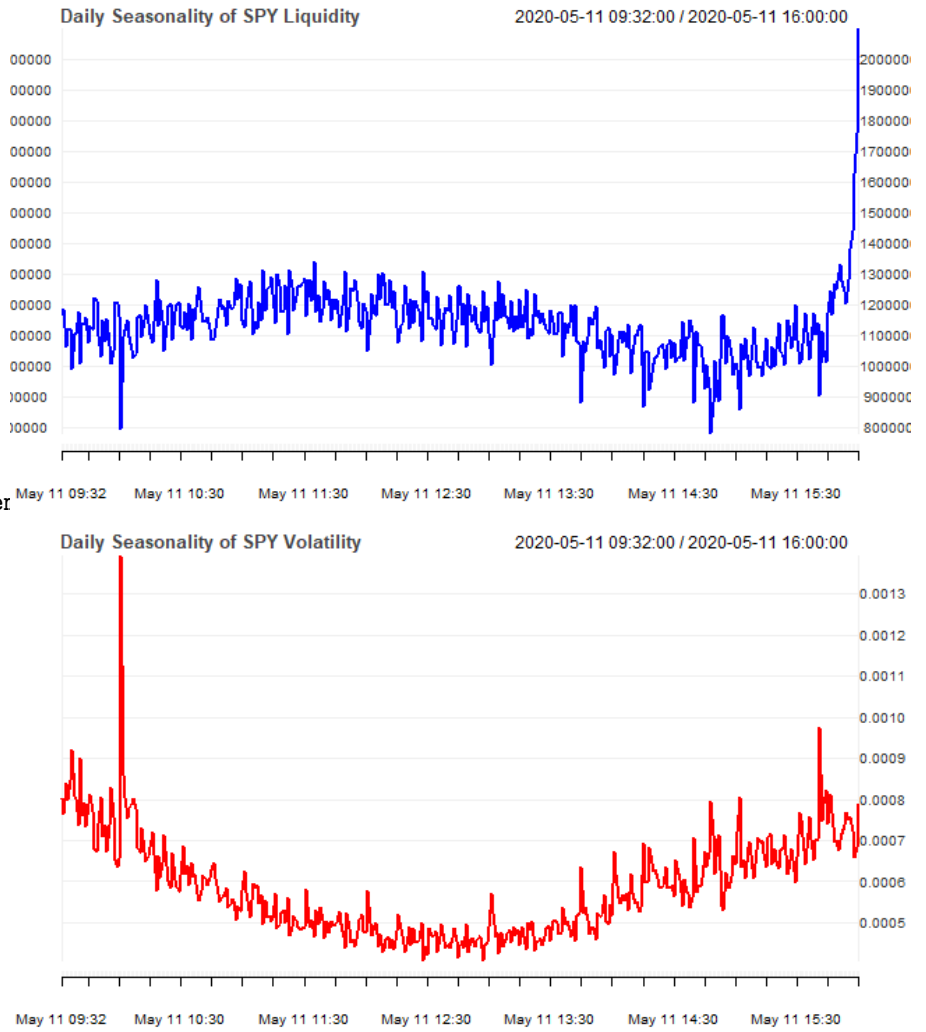


Daily Seasonality of Liquidity and Volatility

Market liquidity is typically the highest at the end of the trading session, and the lowest at the beginning.

The end of day spike in trading volumes and liquidity is driven by computer-driven investors liquidating their positions.

```
> # Calculate market liquidity
> liquidi_ty <- sqrt(volume_agg)/vol_agg
> # Plot daily seasonality of market liquidity
> x11(width=6, height=7) ; par(mfrow=c(2, 1))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> chart_Series(liquidi_ty[c(-1, -NROW(liquidi_ty))], theme=plot_ther
+   name="Daily Seasonality of SPY Liquidity", plot=FALSE)
> plot_theme$col$line.col <- c("red")
> chart_Series(vol_agg[c(-1, -NROW(vol_agg))], theme=plot_theme,
+   name="Daily Seasonality of SPY Volatility")
```



Financial and Commodity Futures Contracts

The underlying assets delivered in *commodity futures* contracts are commodities, such as grains (corn, wheat), or raw materials and metals (oil, aluminum).

The underlying assets delivered in *financial futures* contracts are financial assets, such as stocks, bonds, and currencies.

Many futures contracts use cash settlement instead of physical delivery of the asset.

Futures contracts on different underlying assets can have quarterly, monthly, or even weekly expiration dates.

The front month futures contract is the contract with the closest expiration date to the current date.

Symbols of futures contracts are obtained by combining the contract code with the month code and the year.

For example, *ESM9* is the symbol for the *S&P500* index E-mini futures expiring in June 2019.

Futures contract	Code
S&P500 index	ES
10yr Treasury	ZN
VIX index	VX
Gold	GC
Oil	CL
Euro FX	EC
Swiss franc	SF
Japanese Yen	JY

Month	Code
January	F
February	G
March	H
April	J
May	K
June	M
July	N
August	Q
September	U
October	V
November	X
December	Z

Interactive Brokers provides more information about futures contracts:

IB Contract and Symbol Database

IB Traded Products

List of Popular Futures Contracts.

E-mini Futures Contracts

E-mini futures are contracts with smaller notionals and tick values, which are more suitable for retail investors.

For example, the *QM* E-mini oil future notional is 500 barrels, while the standard *CL* oil future notional is 1,000 barrels.

The tick value is the change in the dollar value of the futures contract due to a one tick change in the underlying price.

For example, the tick value of the *ES* E-mini *S&P500* future is $12.50, and one tick is 0.25.

So if the *S&P500* index changes by one tick (0.25), then the value of a single *ES* E-mini contract changes by $12.50, while the standard *SP* contract value changes by $62.5.

The *ES* E-mini *S&P500* futures trade almost continuously 24 hours per day, from 6:00 PM Eastern Time (ET) on Sunday night to 5:00 PM Friday night (with a trading halt between 4:15 and 4:30 PM ET each day).

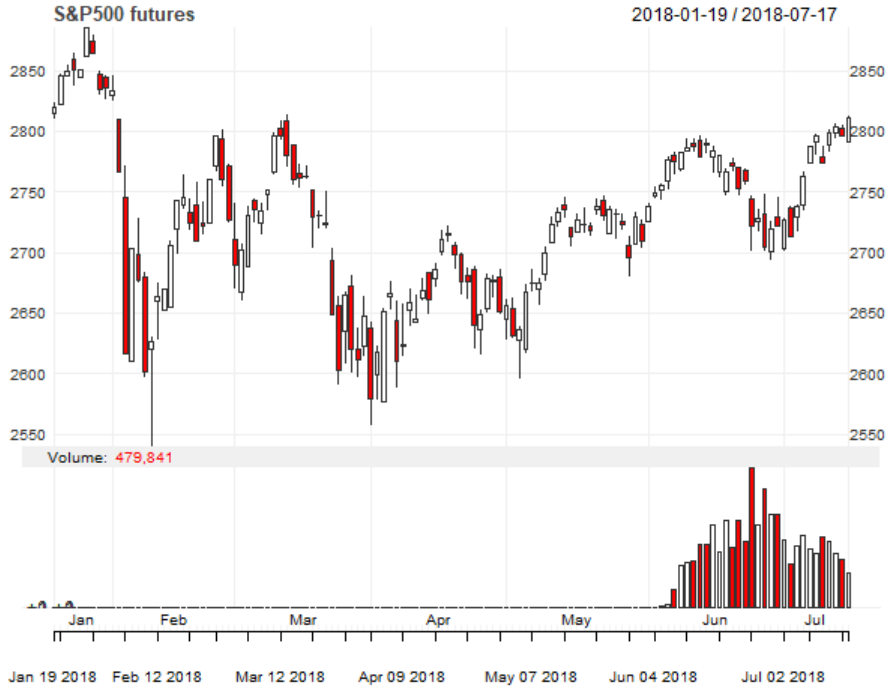
Futures contract	Standard	E-mini
S&P500 index	SP	ES
10yr Treasury	ZN	ZN
VIX index	VX	delisted
Gold	GC	YG
Oil	CL	QM
Euro FX	EC	E7
Swiss franc	SF	MSF
Japanese Yen	JY	J7

Plotting *S&P500* Futures Data

The function `data.table::fread()` reads `.csv` files over five times faster than the function `read.csv()`!

The function `as.POSIXct.numeric()` coerces a `numeric` value representing the *moment of time* into a `POSIXct` *date-time*, equal to the *clock time* in the local *time zone*.

```
> # Load data for S&P Emini futures June 2019 contract
> dir_name <- "C:/Develop/data/ib_data"
> file_name <- file.path(dir_name, "ES_ohlc.csv")
> # Read a data table from CSV file
> price_s <- data.table::fread(file_name)
> class(price_s)
> # Coerce first column from string to date-time
> unlist(sapply(price_s, class))
> tail(price_s)
> price_s$Index <- as.POSIXct(price_s$Index,
+   tz="America/New_York", origin="1970-01-01")
> # Coerce price_s into xts series
> price_s <- data.table::as.xts.data.table(price_s)
> class(price_s)
> tail(price_s)
> colnames(price_s)[1:5] <- c("Open", "High", "Low", "Close", "Volu
> tail(price_s)
```



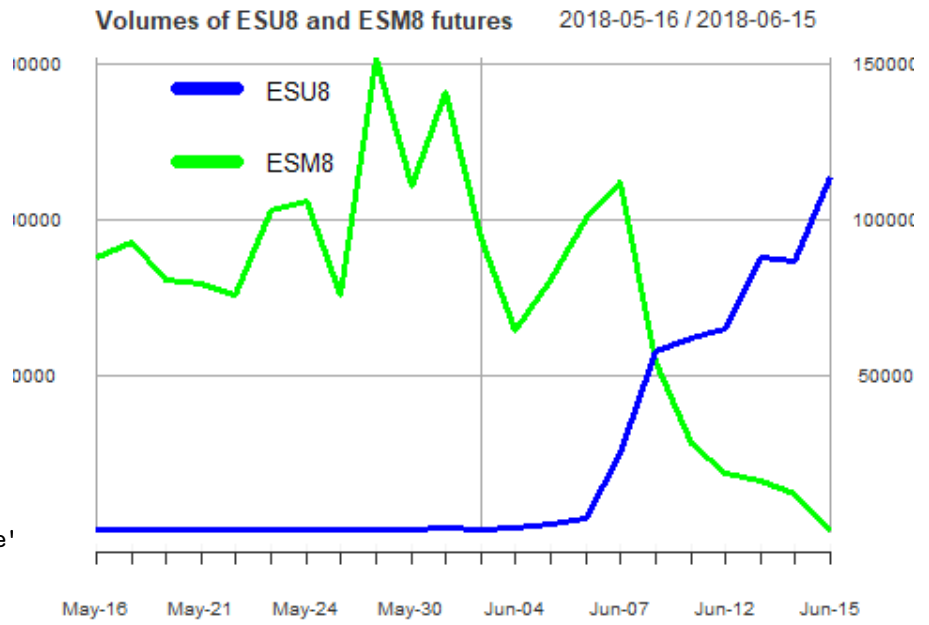

```
> # Plot OHLC data in x11 window
> x11(width=5, height=4)  # Open x11 for plotting
> par(mar=c(5, 5, 2, 1), oma=c(0, 0, 0, 0))
> chart_Series(x=price_s, TA="add_Vo()",
+   name="S&P500 futures")
> # Plot dygraph
> dygraphs::dygraph(price_s[, 1:4], main="OHLC prices") %>%
+   dyCandlestick()
```

Consecutive Contract Futures Volumes

The trading volumes of a futures contract drop significantly shortly before its expiration, and the successive contract volumes increase.

The contract with the highest trading volume is usually considered the most liquid contract.

```
> # Load ESU8 data
> dir_name <- "C:/Develop/data/ib_data"
> file_name <- file.path(dir_name, "ESU8.csv")
> ES_U8 <- data.table::fread(file_name)
> # Coerce ES_U8 into xts series
> ES_U8$V1 <- as.Date(as.POSIXct.numeric(ES_U8$V1,
+     tz="America/New_York", origin="1970-01-01"))
> ES_U8 <- data.table::as.xts.data.table(ES_U8)
> colnames(ES_U8)[1:5] <- c("Open", "High", "Low", "Close", "Volume"
> # Load ESM8 data
> file_name <- file.path(dir_name, "ESM8.csv")
> ES_M8 <- data.table::fread(file_name)
> # Coerce ES_M8 into xts series
> ES_M8$V1 <- as.Date(as.POSIXct.numeric(ES_M8$V1,
+     tz="America/New_York", origin="1970-01-01"))
> ES_M8 <- data.table::as.xts.data.table(ES_M8)
> colnames(ES_M8)[1:5] <- c("Open", "High", "Low", "Close", "Volume
```




```
> # Plot last month of ESU8 and ESM8 volume data
> en_d <- end(ES_M8)
> star_t <- (en_d - 30)
> vol_ume <- cbind(Vo(ES_U8),
+   Vo(ES_M8))[paste0(star_t, "/", en_d)]
> colnames(vol_ume) <- c("ESU8", "ESM8")
> col_ors <- c("blue", "green")
> plot(vol_ume, col=col_ors, lwd=3, major.ticks="days",
+      format.labels="%b-%d", observation.based=TRUE,
+      main="Volumes of ESU8 and ESM8 futures")
> legend("topleft", legend=colnames(vol_ume), col=col_ors,
+  title=NULL, bty="n", lty=1, lwd=6, inset=0.1, cex=0.7)
```

Chaining Together Futures Prices

Chaining futures means splicing together prices from several consecutive futures contracts.

A continuous futures contract is a time series of prices obtained by chaining together prices from consecutive futures contracts.

The price of the continuous contract is equal to the most liquid contract times a scaling factor.

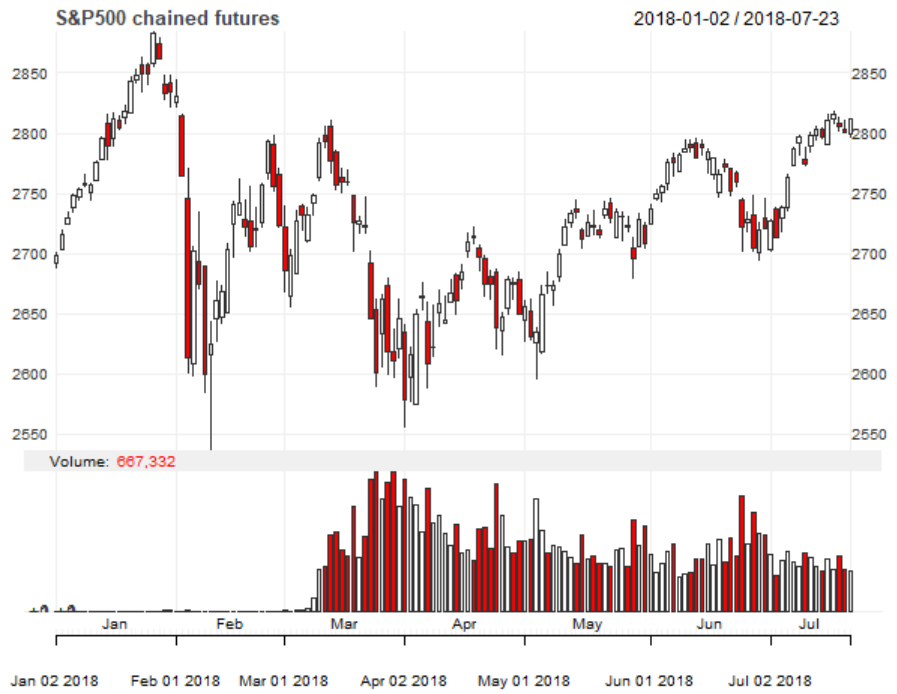
When the next contract becomes more liquid, then the continuous contract price is rolled over to that contract.

Futures contracts with different maturities (expiration dates) trade at different prices because of the futures curve, which causes price jumps between consecutive futures contracts.

The old contract price is multiplied by a scaling factor after that contract is rolled, to remove price jumps.

So the continuous contract prices are not equal to the past futures prices.

Interactive Brokers provides information about Continuous Contract Futures market data:
Continuous Contract Futures Data




```
> # Find date when ESU8 volume exceeds ESM8
> exceed_s <- (vol_ume[, "ESU8"] > vol_ume[, "ESM8"])
> in_dex <- match(TRUE, exceed_s)
> # in_dex <- min(which(exceed_s))
> # Scale the ES_M8 prices
> in_dex <- index(exceed_s[in_dex])
> fac_tor <- as.numeric(Cl(ES_U8[in_dex])/Cl(ES_M8[in_dex]))
> ES_M8[, 1:4] <- fac_tor*ES_M8[, 1:4]
> # Calculate continuous contract prices
> chain_ed <- rbind(ES_M8[index(ES_M8) < in_dex],
+             ES_U8[index(ES_U8) >= in_dex])
> # Or
> # Chain_ed <- rbind(ES_M8[paste0("/", in_dex-1)],
> #                   ES_U8[paste0(in_dex, "/")])
> # Plot continuous contract prices
> chart_Series(x=chain_ed["2018"], TA="add_Vo()",
```

VIX Volatility Index

The *VIX* Volatility Index is an estimate of expected stock market volatility, calculated from the implied volatilities of options on the *S&P500* Index (SPX).

The *VIX* index is not a directly tradable asset, but it can be traded using *VIX* futures.

The CBOE provides daily historical data for the *VIX* index.

```
> # Download VIX index data from CBOE
> vix_index <- data.table::fread("http://www.cboe.com/publish/schedu
> class(vix_index)
> dim(vix_index)
> tail(vix_index)
> sapply(vix_index, class)
> vix_index <- xts(vix_index[, -1],
+   order.by=as.Date(vix_index$Date, format="%m/%d/%Y"))
> colnames(vix_index) <- c("Open", "High", "Low", "Close")
> # Save the VIX data to binary file
> load(file="C:/Develop/data/ib_data/vix_cboe.RData")
> ls(vix_env)
> vix_env$vix_index <- vix_index
> ls(vix_env)
> save(vix_env, file="C:/Develop/data/ib_data/vix_cboe.RData")
> # Plot OHLC data in x11 window
> chart_Series(x=vix_index["2018"], name="VIX Index")
> # Plot dygraph
> dygraphs::dygraph(vix_index, main="VIX Index") %>%
+   dyCandlestick()
```

VIX Futures Contracts

VIX futures are cash-settled futures contracts on the *VIX* Index.

The most liquid *VIX* futures are with monthly expiration dates (CBOE Expiration Calendar), but weekly *VIX* futures are also traded.

These are the VIX Futures Monthly Expiration Dates from 2004 to 2019.

VIX futures are traded on the CFE (CBOE Futures Exchange):

http://cfe.cboe.com/

http://www.cboe.com/vix

VIX Contract Specifications:

VIX Contract Specifications

VIX Expiration Calendar

Standard and Poor's explains the methodology of the *VIX* Futures Indices

```
> # Read CBOE monthly futures expiration dates
> date_s <- read.csv(
+   file="C:/Develop/data/vix_data/vix_dates.csv")
> date_s <- as.Date(date_s[, 1])
> year_s <- format(date_s, format="%Y")
> year_s <- substring(year_s, 4)
> # Monthly futures contract codes
> month_codes <-
+   c("F", "G", "H", "J", "K", "M",
+     "N", "Q", "U", "V", "X", "Z")
> sym_bols <- paste0("VX", month_codes, year_s)
> date_s <- as.data.frame(date_s)
> colnames(date_s) <- "exp_dates"
> rownames(date_s) <- sym_bols
> # Write dates to CSV file, with row names
> write.csv(date_s, row.names=TRUE,
+   file="C:/Develop/data/vix_data/vix_futures.csv")
> # Read back CBOE futures expiration dates
> date_s <- read.csv(file="C:/Develop/data/vix_data/vix_futures.csv"
+   row.names=1)
> date_s[, 1] <- as.Date(date_s[, 1])
```

VIX Futures Curve

Futures contracts with different expiration dates trade at different prices, known as the *futures curve* (or *term structure*).

The *VIX* futures curve is similar to the interest rate *yield curve*, which displays yields at different bond maturities.

The *VIX* futures curve is not the same as the *VIX* index term structure.

More information about the *VIX* Index and the *VIX* futures curve:

- VIX Futures
- VIX Futures Data
- VIX Futures Curve
- VIX Index Term Structure

```
> # Load VIX futures data from binary file
> load(file="C:/Develop/data/vix_data/vix_cboe.RData")
> # Get all VIX futures for 2018 except January
> sym_bols <- ls(vix_env)
> sym_bols <- sym_bols[grep("*8", sym_bols)]
> sym_bols <- sym_bols[2:9]
> # Specify dates for curves
> low_vol <- as.Date("2018-01-11")
> hi_vol <- as.Date("2018-02-05")
> # Extract all VIX futures prices on the dates
> curve_s <- lapply(sym_bols, function(sym_bol) {
+   x_ts <- get(x=sym_bol, envir=vix_env)
+   Cl(x_ts[c(low_vol, hi_vol)])
+ })  # end lapply
> curve_s <- rutils::do_call(cbind, curve_s)
> colnames(curve_s) <- sym_bols
> curve_s <- t(coredata(curve_s))
> colnames(curve_s) <- c("Contango 01/11/2018",
+                        "Backwardation 02/05/2018")
```

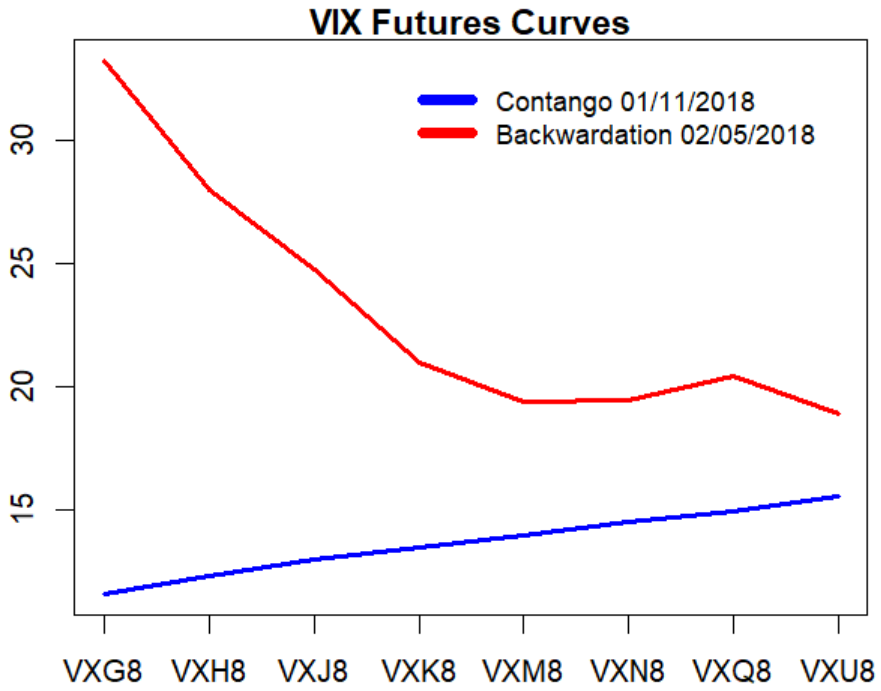
Contango and *Backwardation* of *VIX* Futures Curve

When prices are *low* then the futures curve is usually *upward sloping*, known as *contango*.

Futures prices are in *contango* most of the time.

When prices are *high* then the curve is usually *downward sloping*, known as *backwardation*.

```
> x11(width=7, height=5)
> par(mar=c(3, 2, 1, 1), oma=c(0, 0, 0, 0))
> plot(curve_s[, 1], type="l", lty=1, col="blue", lwd=3,
+      xaxt="n", xlab="", ylab="", ylim=range(curve_s),
+      main="VIX Futures Curves")
> axis(1, at=(1:NROW(curve_s)), labels=rownames(curve_s))
> lines(curve_s[, 2], lty=1, lwd=3, col="red")
> legend(x="topright", legend=colnames(curve_s),
+  inset=0.05, cex=1.0, bty="n",
+  col=c("blue", "red"), lwd=6, lty=1)
```



Futures Prices at Constant Maturity

A constant maturity futures price is the price of a hypothetical futures contract with an expiration date at a fixed number of days in the future.

Futures prices at a constant maturity can be calculated by interpolating the prices of contracts with neighboring expiration dates.

```
> # Load VIX futures data from binary file
> load(file="C:/Develop/data/vix_data/vix_cboe.RData")
> # Read CBOE futures expiration dates
> date_s <- read.csv(file="C:/Develop/data/vix_data/vix_futures.csv"
+   row.names=1)
> sym_bols <- rownames(date_s)
> date_s <- as.Date(date_s[, 1])
> to_day <- as.Date("2018-05-07")
> maturi_ty <- (to_day + 30)
> # Find neighboring futures contracts
> in_dex <- match(TRUE, date_s > maturi_ty)
> front_date <- date_s[in_dex-1]
> back_date <- date_s[in_dex]
> front_symbol <- sym_bols[in_dex-1]
> back_symbol <- sym_bols[in_dex]
> front_price <- get(x=front_symbol, envir=vix_env)
> front_price <- as.numeric(Cl(front_price[to_day]))
> back_price <- get(x=back_symbol, envir=vix_env)
> back_price <- as.numeric(Cl(back_price[to_day]))
> # Calculate the constant maturity 30-day futures price
> ra_tio <- as.numeric(maturi_ty - front_date) /
+   as.numeric(back_date - front_date)
> pric_e <- (ra_tio*back_price + (1-ra_tio)*front_price)
```

VIX Futures Investing

The volatility index moves in the opposite direction to the underlying asset price.

An increase in the *VIX* index coincides with a drop in stock prices, and vice versa.

Taking a *long* position in *VIX* futures is similar to a *short* position in stocks, and vice versa.

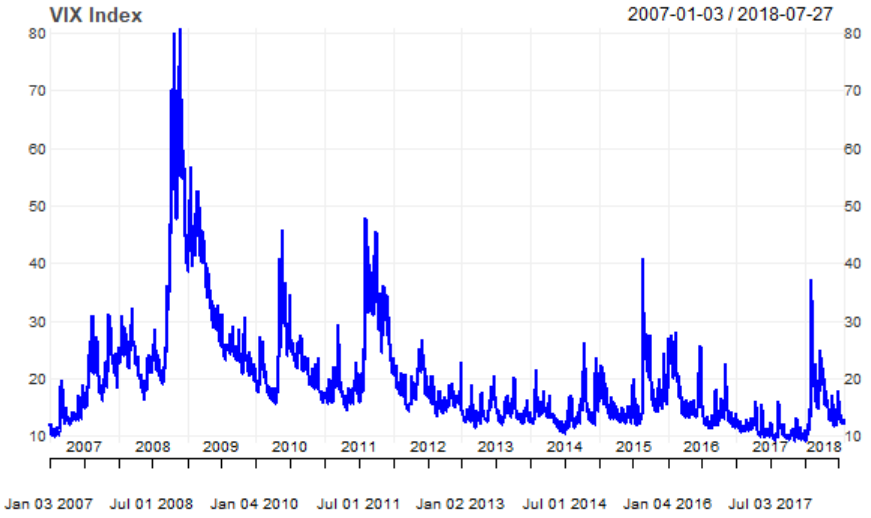
There are several exchange-traded funds (*ETFs*) and exchange traded notes (*ETNs*) which are linked to *VIX* futures.

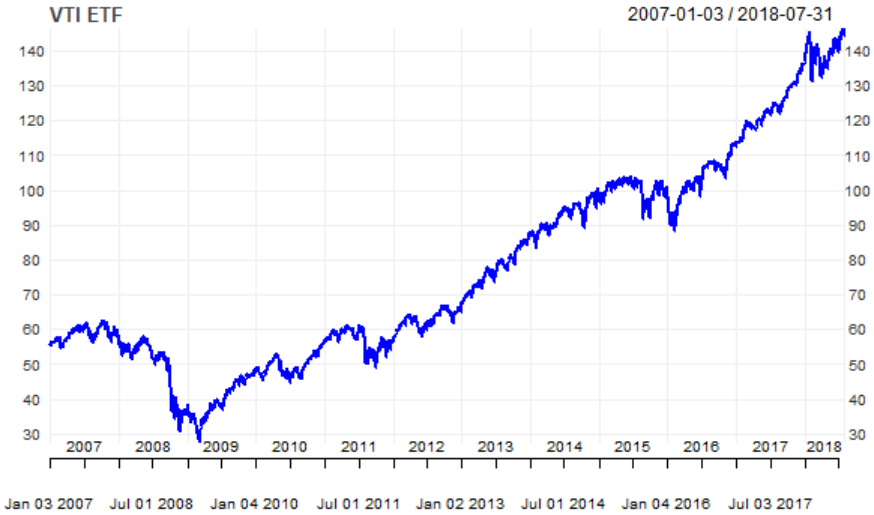
VXX is an *ETN* providing the total return of a *long* *VIX* futures contract (short market risk).

SVXY is an *ETF* providing the total return of a *short* *VIX* futures contract (long market risk).

Standard and Poor's explains the calculation of the Total Return on VIX Futures Indices.

```
> # Plot VIX and SVXY data in x11 window
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- "blue"
> chart_Series(x=Cl(vix_env$vix_index["2007/"]),
+        theme=plot_theme, name="VIX Index")
> chart_Series(x=Cl(rutils::etf_env$VTI["2007/"]),
+        theme=plot_theme, name="VTI ETF")
```





VIX Crash on February 5th 2018

The *SVXY* and *XIV* ETFs rallied strongly after the financial crisis of 2008, so they became very popular with individual investors, and became very "crowded trades".

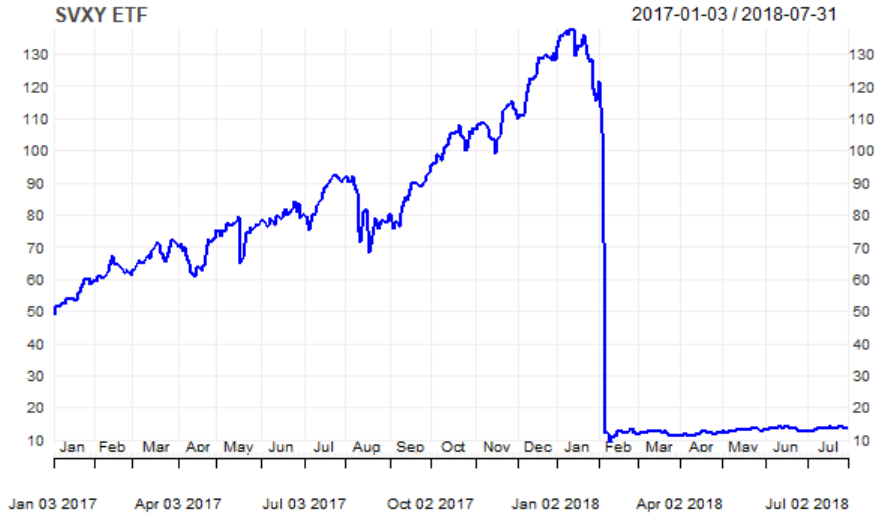
The *SVXY* and *XIV* ETFs had $3.6 billion of assets at the beginning of 2018.

On February 5th 2018 the U.S. stock markets experienced a mini-crash, which was exacerbated by *VIX* futures short sellers.

As a result, the *XIV* ETF hit its termination event and its value dropped to zero:

Volatility Caused Stock Market Crash
XIV ETF Termination Event

```
> chart_Series(x=Cl(vix_env$vix_index["2017/2018"]),
+        theme=plot_theme, name="VIX Index")
> chart_Series(x=Cl(rutils::etf_env$SVXY["2017/2018"]),
+        theme=plot_theme, name="SVXY ETF")
```



Pursuing a Career in Portfolio Management

Becoming a portfolio manager (PM) is very difficult:

- Portfolio management jobs are more difficult than they appear because of survivorship bias among portfolio managers. There are very few successful portfolio managers, and most of the others get fired. But you never hear about those.
- Becoming a portfolio manager (PM) is very difficult because it requires experience and a track record.
- Funds only want to hire experienced PMs with a track record. But how to gain experience and a track record without working as a PM?
- The best way to start is by getting a job as a quant supporting portfolio managers.
- Strong programming skills will drastically improve your chances.
- The best way to get a good job is through networking (personal contacts). Personal contacts are the most valuable resource in pursuing a career in portfolio management.
- Networking techniques: Create a Github account with your projects (it's your calling card), Collaborate on open-source projects.

Homework Assignment

No homework!