FRE7241 Algorithmic Portfolio Management
Lecture#3, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 21, 2021

Combining the Returns of Multiple Assets

There are several ways of combining the returns of multiple assets.

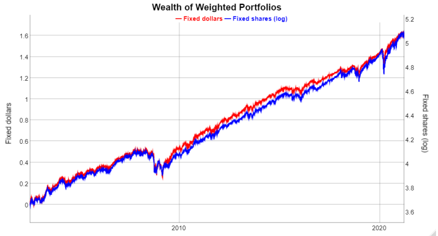
Adding the weighted simple (dollar) returns is equivalent to buying a *fixed number of shares* (aka *Fixed Share Amount* or FSA) proportional to the weights.

Adding the weighted percentage returns is equivalent to investing in *fixed dollar amounts of stock* (aka *Fixed Dollar Amount* or FDA) proportional to the weights.

The portfolio allocations must be periodically rebalanced to keep the dollar amounts of the stocks proportional to the weights.

This *rebalancing* acts as a mean reverting strategy - selling shares when their price goes up, and vice versa.

```
> # Calculate VTI and IEF dollar returns
> price_s <- rutils::etf_env$price_s[, c("VTI", "IEF")]
> price_s <- na.omit(price_s)
> date_s <- index(price_s)
> rets_dollar <- rutils::diff_it(price_s)
> # Calculate VTI and IEF percentage returns
> rets_percent <- rets_dollar/
+   rutils::lag_it(price_s, lagg=1, pad_zeros=FALSE)
```




```
> # Wealth of fixed shares (without rebalancing)
> weight_s <- c(0.5, 0.5)
> rets_dollar[1, ] <- price_s[1, ]
> wealth_fsa <- cumsum(rets_dollar %*% weight_s)
> # Wealth of fixed dollars (with rebalancing)
> wealth_fda <- cumsum(rets_percent %*% weight_s)
> # Plot log wealth
> weal_th <- cbind(wealth_fda, log(wealth_fsa))
> weal_th <- xts::xts(weal_th, index(price_s))
> colnames(weal_th) <- c("Fixed dollars", "Fixed shares (log)")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(weal_th, main="Wealth of Weighted Portfolios") %
+   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=2)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=2
+   dyLegend(show="always", width=500)
```

Transaction Costs of Weighted Portfolio Rebalancing

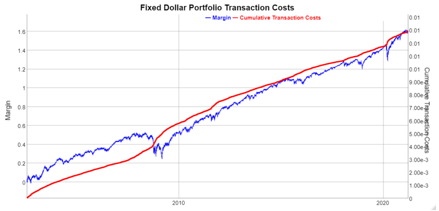
Maintaining a *fixed dollar amount* of stock requires periodic *rebalancing*, selling shares when their price goes up, and vice versa.

Adding the weighted percentage returns is equivalent to investing in *fixed dollar amounts of stock* proportional to the weights.

The dollar amount of stock that must be traded in a given period is equal to the weighted sum of the absolute percentage returns: $w_1 \left| r_t^1 \right| + w_2 \left| r_t^2 \right|$.

The *transaction costs* c_t^r due to rebalancing are equal to half the *bid-offer spread* δ times the dollar amount of the traded stock: $c_t^r = \frac{\delta}{2}(w_1 \left| r_t^1 \right| + w_2 \left| r_t^2 \right|)$.

The *cumulative transaction costs* $\sum_{i=1}^{t} c_i^r$ must be subtracted from the *margin account* m_t: $m_t - \sum_{i=1}^{t} c_i^r$.




```
> # Margin account for fixed dollars (with rebalancing)
> mar_gin <- cumsum(rets_percent %*% weight_s)
> # Cumulative transaction costs
> cost_s <- bid_offer*cumsum(abs(rets_percent) %*% weight_s)/2
> # Subtract transaction costs from margin account
> mar_gin <- (mar_gin - cost_s)
> # dygraph plot of margin and transaction costs
> da_ta <- cbind(mar_gin, cost_s)
> da_ta <- xts::xts(da_ta, index(price_s))
> col_names <- c("Margin", "Cumulative Transaction Costs")
> colnames(da_ta) <- col_names
> dygraphs::dygraph(da_ta, main="Fixed Dollar Portfolio Transaction
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue") %>%
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=3)
+   dyLegend(show="always", width=500)
```

Portfolio With Fixed Ratios of Dollar Amounts

Consider a portfolio with *fixed ratios of dollar amounts* (aka *Constant Dollar Allocations* or CDA), not fixed dollar amounts.

The total wealth is not fixed and is equal to the portfolio market value, so there's no margin account.

Let r_i be the percentage returns, ω_i be the portfolio weights, and $\bar{r}_t = \sum_{i=1}^{n} \omega_i r_i$ be the weighted percentage returns at time t.

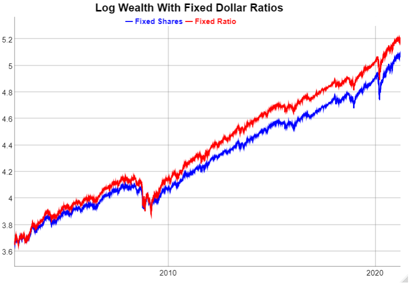
The total portfolio wealth at time t is equal to the wealth at time $t-1$ multiplied by the weighted returns: $w_t = w_{t-1}(1 + \bar{r}_t)$.

The dollar amount of stock i at time t increases by $\omega_i r_i$ so it's equal to $\omega_i w_{t-1}(1 + r_i)$, while the target amount is $\omega_i w_t = \omega_i w_{t-1}(1 + \bar{r}_t)$

The dollar amount of stock i needed to trade to rebalance back to the target weight is equal to:

$$\begin{aligned} \varepsilon_i &= |\omega_i w_{t-1}(1 + \bar{r}_t) - \omega_i w_{t-1}(1 + r_i)| \\ &= \omega_i w_{t-1} \, |\bar{r}_t - r_i| \end{aligned}$$

If $\bar{r}_t > r_i$ then an amount ε_i of the stock i needs to be bought, and if $\bar{r}_t < r_i$ then it needs to be sold.




```
> # Wealth of fixed shares (without rebalancing)
> wealth_fsa <- cumsum(rets_dollar %*% weight_s)
> # Calculate weighted percentage returns
> rets_weighted <- rets_percent %*% weight_s
> # Wealth of fixed ratio of dollar amounts (with rebalancing)
> wealth_cda <- cumprod(1 + rets_weighted)
> wealth_cda <- wealth_fsa[1]*wealth_cda
> # Plot log wealth
> weal_th <- log(cbind(wealth_fsa, wealth_cda))
> weal_th <- xts::xts(weal_th, index(price_s))
> colnames(weal_th) <- c("Fixed Shares", "Fixed Ratio")
> dygraphs::dygraph(weal_th, main="Log Wealth of Fixed Dollar Ratios
+   dyOptions(colors=c("blue","red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Transaction Costs With Constant Dollar Allocations

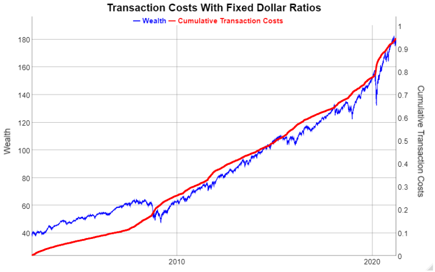
In each period the stocks must be rebalanced to maintain the constant dollar allocations.

The total dollar amount of stocks that need to be traded to rebalance back to the target weight is equal to: $\sum_{i=1}^{n} \varepsilon_i = w_{t-1} \sum_{i=1}^{n} \omega_i \, |\bar{r}_t - r_i|$

The *transaction costs* c_t^r are equal to half the *bid-offer spread* δ times the dollar amount of the traded stock: $c_t^r = \frac{\delta}{2} \sum_{i=1}^{n} \varepsilon_i$.

The *cumulative transaction costs* $\sum_{i=1}^{t} c_i^r$ must be subtracted from the *wealth* w_t: $w_t - \sum_{i=1}^{t} c_i^r$.

```
> # Returns in excess of weighted returns
> ex_cess <- lapply(rets_percent, function(x) (rets_weighted - x))
> ex_cess <- do.call(cbind, ex_cess)
> sum(ex_cess %*% weight_s)
> # Calculate weighted sum of absolute excess returns
> ex_cess <- abs(ex_cess) %*% weight_s
> # Total dollar amount of stocks that need to be traded
> ex_cess <- ex_cess*rutils::lag_it(wealth_cda)
> # Cumulative transaction costs
> cost_s <- bid_offer*cumsum(ex_cess)/2
> # Subtract transaction costs from wealth
> wealth_cda <- (wealth_cda - cost_s)
```




```
> # dygraph plot of wealth and transaction costs
> weal_th <- cbind(wealth_cda, cost_s)
> weal_th <- xts::xts(weal_th, index(price_s))
> col_names <- c("Wealth", "Cumulative Transaction Costs")
> colnames(weal_th) <- col_names
> dygraphs::dygraph(weal_th, main="Transaction Costs With Fixed Doll
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue") %>%
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=3)
+   dyLegend(show="always", width=500)
```

Stock and Bond Portfolio With Constant Dollar Allocations

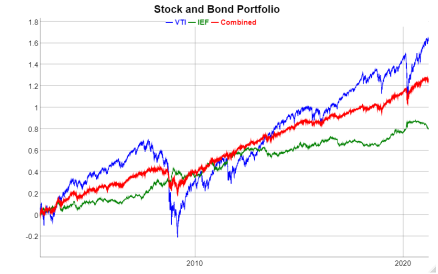
Portfolios combining stocks and bonds can provide a much better risk versus return tradeoff than either of the assets separately, because the returns of stocks and bonds are usually negatively correlated, so they are natural hedges of each other.

The fixed portfolio weights represent the percentage dollar allocations to stocks and bonds, while the portfolio wealth grows over time.

The weights depend on the investment horizon, with a greater allocation to bonds for a shorter investment horizon.

Active investment strategies are expected to outperform static stock and bond portfolios.

```
> # Calculate stock and bond returns
> re_turns <- na.omit(rutils::etf_env$re_turns[, c("VTI", "IEF")])
> weight_s <- c(0.4, 0.6)
> re_turns <- cbind(re_turns, re_turns %*% weight_s)
> colnames(re_turns)[3] <- "Combined"
> # Calculate correlations
> cor(re_turns)
> # Calculate Sharpe ratios
> sqrt(252)*sapply(re_turns, function(x) mean(x)/sd(x))
> # Calculate standard deviation, skewness, and kurtosis
> sapply(re_turns, function(x) {
+   # Calculate standard deviation
+   stddev <- sd(x)
+   # Standardize the returns
+   x <- (x - mean(x))/stddev
+   c(stddev=stddev, skew=mean(x^3), kurt=mean(x^4))
+ })  # end sapply
```




```
> # Wealth of fixed ratio of dollar amounts
> weal_th <- cumprod(1 + re_turns)
> # Plot cumulative wealth
> dygraphs::dygraph(log(weal_th), main="Stock and Bond Portfolio") %
+   dyOptions(colors=c("blue","green","blue","red")) %>%
+   dySeries("Combined", color="red", strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

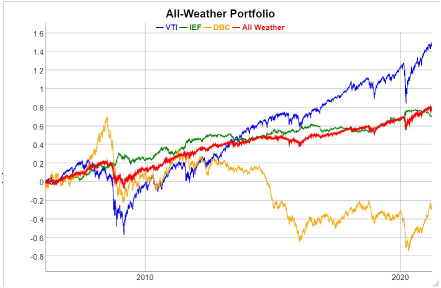
The *All-Weather* Portfolio

The *All-Weather* portfolio is a static portfolio of stocks (30%), bonds (55%), and commodities and precious metals (15%) (approximately), and was designed by Bridgewater Associates, the largest hedge fund in the world:
https://www.bridgewater.com/research-library/the-all-weather-strategy/
http://www.nasdaq.com/article/remember-the-allweather-portfolio-its-having-a-killer-year-cm6855

The three different asset classes (stocks, bonds, commodities) provide positive returns under different economic conditions (recession, expansion, inflation).

The combination of bonds, stocks, and commodities in the *All-Weather* portfolio is designed to provide positive returns under most economic conditions, without the costs of trading.

```
> # Extract ETF returns
> sym_bols <- c("VTI", "IEF", "DBC")
> re_turns <- na.omit(rutils::etf_env$re_turns[, sym_bols])
> # Calculate all-weather portfolio wealth
> weights_aw <- c(0.30, 0.55, 0.15)
> re_turns <- cbind(re_turns, re_turns %*% weights_aw)
> colnames(re_turns)[4] <- "All Weather"
```




```
> # Calculate cumulative wealth from returns
> weal_th <- cumsum(re_turns)
> # dygraph all-weather wealth
> dygraphs::dygraph(weal_th, main="All-Weather Portfolio") %>%
+   dyOptions(colors=c("blue", "green", "orange", "red")) %>%
+   dySeries("All Weather", color="red", strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Plot all-weather wealth
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue", "green", "red")
> quantmod::chart_Series(weal_th, theme=plot_theme, lwd=c(2, 2, 2, 4
+        name="All-Weather Portfolio")
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Constant Proportion Portfolio Insurance Strategy

In the *Constant Proportion Portfolio Insurance* (CPPI) strategy the portfolio is rebalanced between stocks and zero-coupon bonds, to protect against the loss of principal.

A zero-coupon bond pays no coupon, but it's bought at a discount to par (100%), and pays par at maturity. The investor receives capital appreciation instead of coupons.

Let P be the investor principal amount (total initial invested dollar amount), and let F be the zero-coupon *bond floor*. The zero-coupon bond floor F is set so that its value at maturity is equal to the principal P. This guarantees that the investor is paid back at least the full principal P.

The stock investment is levered by the *multiplier* C. The initial dollar amount invested in stocks is equal to the *cushion* $(P - F)$ times the *multiplier* C: $C * (P - F)$. The remaining amount of the principal is invested in zero-coupon bonds and is equal to: $P - C * (P - F)$.

```
> # Calculate VTI returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI["2008/2009"])
> date_s <- index(re_turns)
> n_rows <- NROW(re_turns)
> re_turns <- drop(zoo::coredata(re_turns))
> bfloor <- 60  # bond floor
> co_eff <- 2  # multiplier
> portf_value <- numeric(n_rows)
> portf_value[1] <- 100  # principal
> stock_value <- numeric(n_rows)
> stock_value[1] <- co_eff*(portf_value[1] - bfloor)
> bond_value <- numeric(n_rows)
> bond_value[1] <- (portf_value[1] - stock_value[1])
```

CPPI Strategy Dynamics

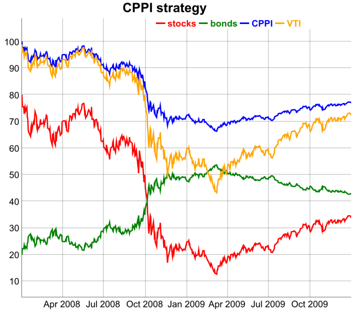
If the stock price changes and the portfolio value becomes P_t, then the dollar amount invested in stocks must be adjusted to: $C * (P_t - F)$. The amount invested in stocks changes both because the stock price changes and because of rebalancing with the zero-coupon bonds.

The amount invested in zero-coupon bonds is then equal to: $P_t - C * (P_t - F)$. If the portfolio value drops to the *bond floor* $P_t = F$, then all the stocks must be sold, with only the zero-coupon bonds remaining. But if the stock price rises, more stocks must be purchased, and vice versa.

Therefore the *CPPI* strategy is a *trend-following* strategy, buying stocks when their prices are rising, and selling when their prices are dropping.

The *CPPI* strategy can be considered a dynamic replication of a portfolio with a zero-coupon bond and a stock call option.

The *CPPI* strategy is exposed to *gap risk*, if stock prices drop suddenly by a large amount. The *gap risk* is exacerbated by high leverage, when the *multiplier* C is large, say greater than 5.




```
> # Simulate CPPI strategy
> for (t in 2:n_rows) {
+   portf_value[t] <- portf_value[t-1] + stock_value[t-1]*re_turns[t
+   stock_value[t] <- co_eff*(portf_value[t] - bfloor)
+   bond_value[t] <- (portf_value[t] - stock_value[t])
+ }  # end for
> # dygraph plot of CPPI strategy
> vt_i <- 100*cumprod(1+re_turns)
> da_ta <- xts::xts(cbind(stock_value, bond_value, portf_value, vt_i
> colnames(da_ta) <- c("stocks", "bonds", "CPPI", "VTI")
> dygraphs::dygraph(da_ta, main="CPPI strategy") %>%
+   dyOptions(colors=c("red", "green","blue","orange"), strokeWidth=
+   dyLegend(show="always", width=300)
```

Risk Parity Strategy

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

This means that the allocations a_i are proportional to the *standardized prices* ($\frac{p_i}{\sigma_i^d}$ - the dollar amounts of stocks with unit dollar volatilities): $a_i \propto \frac{p_i}{\sigma_i^d}$, where σ_i^d is the dollar volatility.

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

In general, the dollar allocations a_i may be set proportional to some target weights ω_i:

$$a_i \propto \frac{\omega_i}{\sigma_i}$$

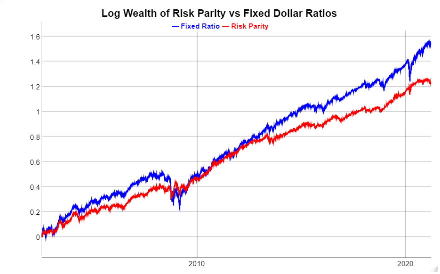
The risk parity strategy is also called the equal risk contributions (ERC) strategy.

```
> # Calculate dollar and percentage returns for VTI and IEF.
> price_s <- rutils::etf_env$price_s[, c("VTI", "IEF")]
> price_s <- na.omit(price_s)
> rets_dollar <- rutils::diff_it(price_s)
> rets_percent <- rets_dollar/rutils::lag_it(price_s, lagg=1, pad_ze
> # Calculate wealth of fixed ratio of dollar amounts.
> weight_s <- c(0.5, 0.5)
> rets_weighted <- rets_percent %*% weight_s
> wealth_cda <- cumprod(1 + rets_weighted)
> # Calculate rolling percentage volatility.
> look_back <- 21
> vo_l <- roll::roll_sd(rets_percent, width=look_back)
> vo_l <- zoo::na.locf(vo_l, na.rm=FALSE)
> vo_l <- zoo::na.locf(vo_l, fromLast=TRUE)
> # Calculate the risk parity portfolio allocations.
> allocation_s <- lapply(1:NCOL(price_s),
+   function(x) weight_s[x]/vo_l[, x])
> allocation_s <- do.call(cbind, allocation_s)
> # Scale allocations to 1 dollar total.
> allocation_s <- allocation_s/rowSums(allocation_s)
> # Lag the allocations
> allocation_s <- rutils::lag_it(allocation_s)
> # Calculate wealth of risk parity.
> rets_weighted <- rowSums(rets_percent*allocation_s)
> wealth_risk_parity <- cumprod(1 + rets_weighted)
```

Risk Parity Strategy Performance

The risk parity strategy for *VTI* and *IEF* has a higher *Sharpe ratio* than the fixed ratio strategy because it's more overweight bonds, which is also why it has lower absolute returns.

Risk parity works better for assets with low correlations and very different volatilities, like stocks and bonds.



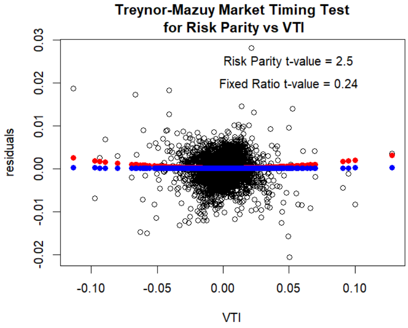

```
> # Calculate the log wealths.
> weal_th <- log(cbind(wealth_cda, wealth_risk_parity))
> weal_th <- xts::xts(weal_th, index(price_s))
> colnames(weal_th) <- c("Fixed Ratio", "Risk Parity")
> # Calculate the Sharpe ratios.
> sqrt(252)*sapply(rutils::diff_it(weal_th), function (x) mean(x)/sd
> # Plot a dygraph of the log wealths.
> dygraphs::dygraph(weal_th, main="Log Wealth of Risk Parity vs Fixe
+   dyOptions(colors=c("blue","red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Risk Parity Strategy Market Timing Skill

The risk parity strategy reduces allocations to assets with rising volatilities, which is often accompanied by negative returns.

This allows the risk parity strategy to better time the markets - selling when prices are about to drop and buying when prices are rising.

The t-value of the *Treynor-Mazuy* test is slightly significant, indicating some market timing skill of the risk parity strategy for *VTI* and *IEF*.




```
> # Test risk parity market timing of VTI using Treynor-Mazuy test
> re_turns <- rutils::diff_it(weal_th)
> vt_i <- rets_percent$VTI
> de_sign <- cbind(re_turns, vt_i, vt_i^2)
> de_sign <- na.omit(de_sign)
> colnames(de_sign)[1:2] <- c("fixed","risk_parity")
> colnames(de_sign)[4] <- "treynor"
> mod_el <- lm(risk_parity ~ VTI + treynor, data=de_sign)
> summary(mod_el)
> # Plot residual scatterplot
> residual_s <- (de_sign$risk_parity - mod_el$coeff[2]*de_sign$VTI)
> residual_s <- mod_el$residuals
> x11(width=6, height=5)
> plot.default(x=de_sign$VTI, y=residual_s, xlab="VTI", ylab="resi
> title(main="Treynor-Mazuy Market Timing Test\n for Risk Parity v
> # Plot fitted (predicted) response values
> fit_ted <- (mod_el$coeff["(Intercept)"] +
+         mod_el$coeff["treynor"]*vt_i^2)
> points.default(x=de_sign$VTI, y=fit_ted, pch=16, col="red")
> text(x=0.05, y=0.025, paste("Risk Parity t-value =", round(summary(mod_el)$coeff["treynor", "t value"], 2)))
```

```
> # Test for fixed ratio market timing of VTI using Treynor-Mazuy te
> mod_el <- lm(fixed ~ VTI + treynor, data=de_sign)
> summary(mod_el)
> # Plot fitted (predicted) response values
> fit_ted <- (mod_el$coeff["(Intercept)"] + mod_el$coeff["treynor"]*
> points.default(x=de_sign$VTI, y=fit_ted, pch=16, col="blue")
> text(x=0.05, y=0.02, paste("Fixed Ratio t-value =", round(summary(
```

Sell in May Calendar Strategy

Sell in May is a *market timing calendar strategy*, in which stocks are sold at the beginning of May, and then bought back at the beginning of November.

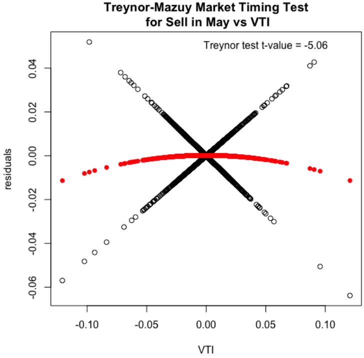
```
> # Calculate positions
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> position_s <- rep(NA_integer_, NROW(vt_i))
> date_s <- index(vt_i)
> date_s <- format(date_s, "%m-%d")
> position_s[date_s == "05-01"] <- 0
> position_s[date_s == "05-03"] <- 0
> position_s[date_s == "11-01"] <- 1
> position_s[date_s == "11-03"] <- 1
> # Carry forward and backward non-NA position_s
> position_s <- zoo::na.locf(position_s, na.rm=FALSE)
> position_s <- zoo::na.locf(position_s, fromLast=TRUE)
> # Calculate strategy returns
> sell_inmay <- position_s*vt_i
> weal_th <- cbind(vt_i, sell_inmay)
> colnames(weal_th) <- c("VTI", "sell_in_may")
> # Calculate Sharpe and Sortino ratios
> sqrt(252)*sapply(weal_th,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])
```


```
> # Plot wealth of Sell in May strategy
> dygraphs::dygraph(cumsum(weal_th), main="Sell in May Strategy") %>
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # OR: Open x11 for plotting
> x11(width=6, height=5)
> par(mar=c(4, 4, 3, 1), oma=c(0, 0, 0, 0))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue", "red")
> quantmod::chart_Series(weal_th, theme=plot_theme, name="Sell in Ma
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Sell in May Strategy Market Timing

The *Sell in May* strategy doesn't demonstrate any ability of *timing* the *VTI* ETF.

```
> # Test if Sell in May strategy can time VTI
> de_sign <- cbind(vt_i, 0.5*(vt_i+abs(vt_i)), vt_i^2)
> colnames(de_sign) <- c("VTI", "merton", "treynor")
> # Perform Merton-Henriksson test
> mod_el <- lm(sell_inmay ~ VTI + merton, data=de_sign)
> summary(mod_el)
> # Perform Treynor-Mazuy test
> mod_el <- lm(sell_inmay ~ VTI + treynor, data=de_sign)
> summary(mod_el)
> # Plot Treynor-Mazuy residual scatterplot
> residual_s <- (sell_inmay - mod_el$coeff[2]*vt_i)
> plot.default(x=vt_i, y=residual_s, xlab="VTI", ylab="residuals")
> title(main="Treynor-Mazuy Market Timing Test\n for Sell in May vs
> # Plot fitted (predicted) response values
> fit_ted <- (mod_el$coeff["(Intercept)"] +
+         mod_el$coeff["treynor"]*vt_i^2)
> points.default(x=vt_i, y=fit_ted, pch=16, col="red")
> text(x=0.05, y=0.05, paste("Treynor test t-value =", round(summary
```



Seasonal Overnight Market Anomaly

The *Overnight Market Anomaly* is the consistent outperformance of overnight returns relative to the daytime returns.

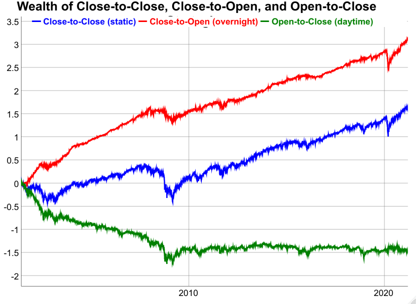
The Overnight Strategy consists of holding a long position only overnight (buying at the close and selling at the open the next day).

The Daytime Strategy consists of holding a long position only during the daytime (buying at the open and selling at the close the same day).

The *Overnight Market Anomaly* has been observed for many decades for most stock market indices, but not always for all stock sectors.

The *Overnight Market Anomaly* has mostly disappeared after the 2008–2009 financial crisis.

```
> # Calculate the log of OHLC VTI prices
> oh_lc <- log(rutils::etf_env$VTI)
> op_en <- quantmod::Op(oh_lc)
> hi_gh <- quantmod::Hi(oh_lc)
> lo_w <- quantmod::Lo(oh_lc)
> clos_e <- quantmod::Cl(oh_lc)
> # Calculate the close-to-close log returns, the intraday
> # open-to-close returns and the overnight close-to-open returns.
> close_close <- rutils::diff_it(clos_e)
> colnames(close_close) <- "close_close"
> open_close <- (clos_e - op_en)
> colnames(open_close) <- "open_close"
> close_open <- (op_en - rutils::lag_it(clos_e, lagg=1, pad_zeros=FALSE))
> colnames(close_open) <- "close_open"
```




```
> # Calculate Sharpe and Sortino ratios
> weal_th <- cbind(close_close, close_open, open_close)
> sqrt(252)*sapply(weal_th,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot log wealth
> dygraphs::dygraph(cumsum(weal_th),
+   main="Wealth of Close-to-Close, Close-to-Open, and Open-to-Close
+   dySeries(name="close_close", label="Close-to-Close (static)", st
+   dySeries(name="close_open", label="Close-to-Open (overnight)", s
+   dySeries(name="open_close", label="Open-to-Close (daytime)", str
+   dyLegend(width=600)
```

Turn of the Month Effect

The *Turn of the Month* (TOM) effect is the outperformance of stocks on the last trading day of the month and on the first three days of the following month.

The *TOM* effect was observed for the period from 1928 to 1975, but it has been less pronounced since the year 2000.

The *TOM* effect has been attributed to the investment of funds deposited at the end of the month.

This would explain why the *TOM* effect has been more pronounced for less liquid small-cap stocks.

```
> # Calculate the VTI returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- zoo::index(vt_i)
> # Calculate first business day of every month
> day_s <- as.numeric(format(date_s, "%d"))
> indeks <- which(rutils::diff_it(day_s) < 0)
> date_s[head(indeks)]
> # Calculate Turn of the Month dates
> indeks <- lapply((-1):2, function(x) indeks + x)
> indeks <- do.call(c, indeks)
> sum(indeks > NROW(date_s))
> indeks <- sort(indeks)
> date_s[head(indeks, 11)]
> # Calculate Turn of the Month pnls
> pnl_s <- numeric(NROW(vt_i))
> pnl_s[indeks] <- vt_i[indeks, ]
```

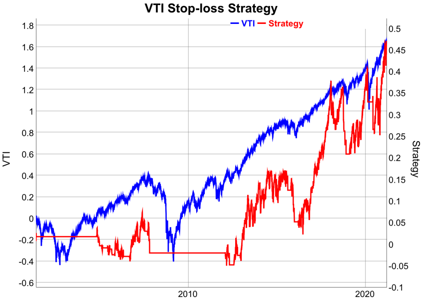

```
> # Combine data
> weal_th <- cbind(vt_i, pnl_s)
> col_names <- c("VTI", "Strategy")
> colnames(weal_th) <- col_names
> # Calculate Sharpe and Sortino ratios
> sqrt(252)*sapply(weal_th,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # dygraph plot VTI Turn of the Month strategy
> dygraphs::dygraph(cumsum(weal_th), main="Turn of the Month Strateg
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", strokeWidth=2, col="blue")
+   dySeries(name=col_names[2], axis="y2", strokeWidth=2, col="red")
```

Stop-loss Rules

Stop-loss rules are used to reduce losses in case of a significant drawdown in returns.

For example, a simple stop-loss rule is to sell the stock if its price drops by 5% below the recent maximum price, and buy it back when the price recovers.

```
> # Calculate the VTI returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- zoo::index(vt_i)
> vt_i <- drop(coredata(vt_i))
> n_rows <- NROW(vt_i)
> # Simulate stop-loss strategy
> sto_p <- 0.05
> ma_x <- 0.0
> cum_ret <- 0.0
> pnl_s <- vt_i
> for (i in 1:n_rows) {
+ # Calculate drawdown
+   cum_ret <- cum_ret + vt_i[i]
+   ma_x <- max(ma_x, cum_ret)
+   dd <- (cum_ret - ma_x)
+ # Check for stop-loss
+   if (dd < -sto_p*ma_x)
+     pnl_s[i+1] <- 0
+ }  # end for
> # Same but without using explicit loops
> cum_sum <- cumsum(vt_i)
> cum_max <- cummax(cumsum(vt_i))
> dd <- (cum_sum - cum_max)
> pnls2 <- vt_i
> is_dd <- rutils::lag_it(dd < -sto_p*cum_max)
> pnls2 <- ifelse(is_dd, 0, pnls2)
> all.equal(pnl_s, pnls2)
```

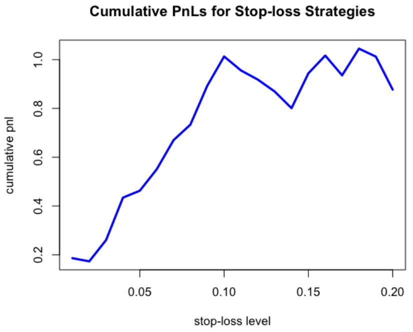



```
> # Combine data
> weal_th <- xts::xts(cbind(vt_i, pnl_s), date_s)
> col_names <- c("VTI", "Strategy")
> colnames(weal_th) <- col_names
> # Calculate Sharpe and Sortino ratios
> sqrt(252)*sapply(weal_th,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # dygraph plot VTI stop-loss strategy
> dygraphs::dygraph(cumsum(weal_th), main="VTI Stop-loss Strategy")
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", strokeWidth=2, col="blue")
+   dySeries(name=col_names[2], axis="y2", strokeWidth=2, col="red")
```

Optimal Stop-loss Rules

Stop-loss rules can reduce the largest losses but they also tend to reduce cumulative returns.

```
> # Simulate multiple stop-loss strategies
> cum_sum <- cumsum(vt_i)
> cum_max <- cummax(cumsum(vt_i))
> dd <- (cum_sum - cum_max)
> cum_pnls <- sapply(0.01*(1:20), function(sto_p) {
+   pnl_s <- vt_i
+   is_dd <- rutils::lag_it(dd < -sto_p*cum_max)
+   pnl_s <- ifelse(is_dd, 0, pnl_s)
+   sum(pnl_s)
+ })  # end sapply
```




```
> # Plot cumulative pnls for stop-loss strategies
> plot(x=0.01*(1:20), y=cum_pnls,
+      main="Cumulative PnLs for Stop-loss Strategies",
+      xlab="stop-loss level", ylab="cumulative pnl",
+      t="l", lwd=3, col="blue")
```

Convolution Filtering of Time Series

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

The function `filter()` with the argument `method="convolution"` calculates the *convolution* of the vector r_i with the filter φ_i:

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

Where f_i is the filtered output vector, and φ_i are the filter coefficients.

`filter()` is very fast because it calculates the filter by calling compiled C++ functions.

`filter()` with `method="convolution"` calls the function `stats:::C_cfilter()` to calculate the *convolution*.

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

The function `roll::roll_sum()` calculates the *weighted* rolling sum (convolution) even faster than `stats:::C_cfilter()`.

```
> # Extract time series of VTI log prices
> clos_e <- log(na.omit(rutils::etf_env$price_s$VTI))
> # Inspect the R code of the function filter()
> filter
> # Calculate EWMA weight_s
> look_back <- 21
> weight_s <- exp(-0.1*1:look_back)
> weight_s <- weight_s/sum(weight_s)
> # Calculate convolution using filter()
> filter_ed <- filter(clos_e, filter=weight_s,
+               method="convolution", sides=1)
> # filter() returns time series of class "ts"
> class(filter_ed)
> # Get information about C_cfilter()
> getAnywhere(C_cfilter)
> # Filter using C_cfilter() over past values (sides=1).
> filter_fast <- .Call(stats:::C_cfilter, clos_e, filter=weight_s,
+                sides=1, circular=FALSE)
> all.equal(as.numeric(filter_ed), filter_fast, check.attributes=FAI
> # Calculate EWMA prices using roll::roll_sum()
> weights_rev <- rev(weight_s)
> roll_ed <- roll::roll_sum(clos_e, width=look_back, weights=weights
> all.equal(filter_fast[-(1:look_back)], as.numeric(roll_ed)[-(1:loo
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(clos_e, filter=weight_s, method="convolution", sic
+   filter_fast=.Call(stats:::C_cfilter, clos_e, filter=weight_s, si
+   roll=roll::roll_sum(clos_e, width=look_back, weights=weights_rev
+   ), times=10)[, c(1, 4, 5)]
```

Recursive Filtering of Time Series

The function `filter()` with `method="recursive"` calls the function `stats:::C_rfilter()` to calculate the *recursive filter* as follows:

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

Where r_i is the filtered output vector, φ_i are the filter coefficients, and ξ_i are standard normal *innovations*.

The *recursive* filter describes an *AR(p)* process, which is a special case of an *ARIMA* process.

The function `HighFreq::sim_arima()` is very fast because it's written using the C++ *Armadillo* numerical library.

```
> # Simulate AR process using filter()
> n_rows <- NROW(clos_e)
> # Calculate ARIMA coefficients and innovations
> co_eff <- weight_s/4
> n_coeff <- NROW(co_eff)
> in_nov <- rnorm(n_rows)
> ari_ma <- filter(x=in_nov, filter=co_eff, method="recursive")
> # Get information about C_rfilter()
> getAnywhere(C_rfilter)
> # Filter using C_rfilter() compiled C++ function directly
> arima_fast <- .Call(stats:::C_rfilter, in_nov, co_eff,
+                 double(n_coeff + n_rows))
> all.equal(as.numeric(ari_ma), arima_fast[-(1:n_coeff)],
+     check.attributes=FALSE)
> # Filter using C++ code
> arima_fastest <- HighFreq::sim_arima(in_nov, rev(co_eff))
> all.equal(arima_fast[-(1:n_coeff)], drop(arima_fastest))
> # Benchmark speed of the three methods
> summary(microbenchmark(
+   filter=filter(x=in_nov, filter=co_eff, method="recursive"),
+   filter_fast=.Call(stats:::C_rfilter, in_nov, co_eff, double(n_co
+   Rcpp=HighFreq::sim_arima(in_nov, rev(co_eff))
+   ), times=10)[, c(1, 4, 5)]
```

Data Smoothing and The Bias-Variance Tradeoff

Filtering through an averaging filter produces data *smoothing*.

Smoothing real-time data with a trailing filter reduces its *variance* but it increases its *bias* because it introduces a time lag.

Smoothing historical data with a centered filter reduces its *variance* but it introduces *data snooping*.

In engineering, smoothing is called a *low-pass filter*, since it eliminates high frequency signals, and it passes through low frequency signals.

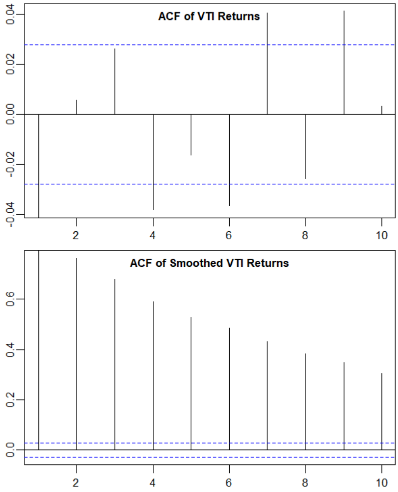

```
> # Calculate trailing EWMA prices using roll::roll_sum()
> look_back <- 21
> weight_s <- exp(-0.1*1:look_back)
> weight_s <- weight_s/sum(weight_s)
> weights_rev <- rev(weight_s)
> filter_ed <- roll::roll_sum(clos_e, width=NROW(weight_s), weight
> # Copy warmup period
> filter_ed[1:look_back] <- clos_e[1:look_back]
> # Combine prices with smoothed prices
> price_s <- cbind(clos_e, filter_ed)
> colnames(price_s)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diff_it(price_s), sd)
> # Plot dygraph
> dygraphs::dygraph(price_s["2009"], main="VTI Prices and Trailing
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

```
> # Calculate centered EWMA prices using roll::roll_sum()
> weight_s <- c(weights_rev, weight_s[-1])
> weight_s <- weight_s/sum(weight_s)
> filter_ed <- roll::roll_sum(clos_e, width=NROW(weight_s), weights=
> # Copy warmup period
> filter_ed[1:(2*look_back)] <- clos_e[1:(2*look_back)]
> # Center the data
> filter_ed <- rutils::lag_it(filter_ed, -(look_back-1), pad_zeros=
> # Combine prices with smoothed prices
> price_s <- cbind(clos_e, filter_ed)
> colnames(price_s)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diff_it(price_s), sd)
> # Plot dygraph
> dygraphs::dygraph(price_s["2009"], main="VTI Prices and Centered S
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

Autocorrelations of Smoothed Time Series

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

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



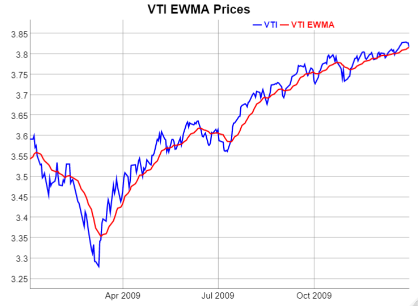
EWMA Price Technical Indicator

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

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

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

```
> # Extract log VTI prices
> clos_e <- log(na.omit(rutils::etf_env$price_s$VTI))
> n_rows <- NROW(clos_e)
> # Calculate EWMA weights
> look_back <- 21
> lamb_da <- 0.1
> weight_s <- exp(lamb_da*1:look_back)
> weight_s <- weight_s/sum(weight_s)
> # Calculate EWMA prices
> ew_ma <- roll::roll_sum(clos_e, width=look_back, weights=weight_s
> # Copy over NA values
> ew_ma <- zoo::na.locf(ew_ma, fromLast=TRUE)
> price_s <- cbind(clos_e, ew_ma)
> colnames(price_s) <- c("VTI", "VTI EWMA")
```




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

Volume-Weighted Average Price Indicator

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

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

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

```
> # Calculate log OHLC prices and volumes
> sym_bol <- "VTI"
> oh_lc <- rutils::etf_env$VTI
> n_rows <- NROW(oh_lc)
> clos_e <- log(quantmod::Cl(oh_lc))
> vol_ume <- quantmod::Vo(oh_lc)
> # Calculate the VWAP prices
> look_back <- 21
> v_wap <- roll::roll_sum(clos_e*vol_ume, width=look_back, min_obs=
> volume_roll <- roll::roll_sum(vol_ume, width=look_back, min_obs=1
> v_wap <- v_wap/volume_roll
> v_wap <- zoo::na.locf(v_wap, fromLast=TRUE)
> price_s <- cbind(clos_e, v_wap)
> colnames(price_s) <- c(sym_bol, paste(sym_bol, "VWAP"))
```


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

Smooth Asset Returns

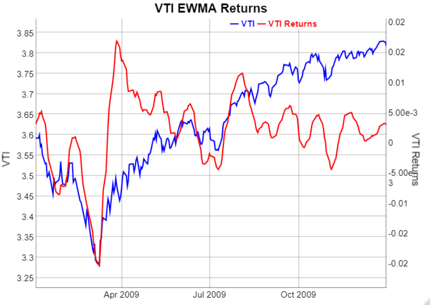
Asset returns are calculated by filtering prices through a *differencing* filter.

The simplest *differencing* filter is the filter with coefficients $(1, -1)$: $r_i = p_i - p_{i-1}$.

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

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

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




```
> # Calculate two EWMA prices
> look_back <- 21
> lamb_da <- 0.1
> weight_s <- exp(lamb_da*1:look_back)
> weight_s <- weight_s/sum(weight_s)
> ewma_fast <- roll::roll_sum(clos_e, width=look_back, weights=wei
> lamb_da <- 0.05
> weight_s <- exp(lamb_da*1:look_back)
> weight_s <- weight_s/sum(weight_s)
> ewma_slow <- roll::roll_sum(clos_e, width=look_back, weights=weight_s, min_obs=1)
```

```
> # Calculate VTI returns
> re_turns <- (ewma_fast - ewma_slow)
> price_s <- cbind(clos_e, re_turns)
> colnames(price_s) <- c(sym_bol, paste(sym_bol, "Returns"))
> # Plot dygraph of VTI Returns
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s["2009"], main=paste(sym_bol, "EWMA Retu
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

Fractional Asset Returns

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

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

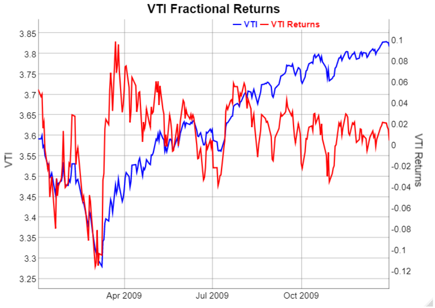
The simple returns can be generalized to the fractional returns by raising the returns operator to some power $\delta < 1$:

$$r_i = (1 - L)^\delta p_i =$$

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

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

The fractional returns provide a tradeoff between simple returns (which are range-bound but with no memory) and prices (which have memory but are not range-bound).




```
> # Calculate fractional weights
> del_ta <- 0.1
> weight_s <- (del_ta - 0:(look_back-2)) / 1:(look_back-1)
> weight_s <- (-1)^(1:(look_back-1))*cumprod(weight_s)
> weight_s <- c(1, weight_s)
> weight_s <- (weight_s - mean(weight_s))
> weight_s <- rev(weight_s)
> # Calculate fractional VTI returns
> re_turns <- roll::roll_sum(clos_e, width=look_back, weights=weight
> price_s <- cbind(clos_e, re_turns)
> colnames(price_s) <- c(sym_bol, paste(sym_bol, "Returns"))
> # Plot dygraph of VTI Returns
> col_names <- colnames(price_s)
> dygraphs::dygraph(price_s["2009"], main=paste(sym_bol, "Fractional
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

Augmented Dickey-Fuller Test for Asset Returns

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

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

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

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

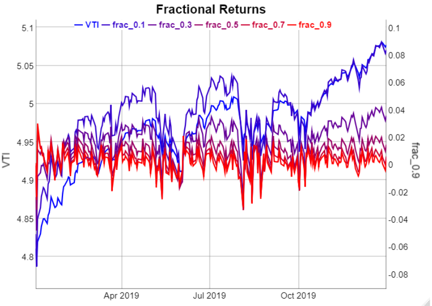
The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

```
> # Calculate VTI log returns
> clos_e <- log(quantmod::Cl(rutils::etf_env$VTI))
> re_turns <- rutils::diff_it(clos_e)
> # Perform ADF test for prices
> tseries::adf.test(clos_e)
> # Perform ADF test for returns
> tseries::adf.test(re_turns)
```

Augmented Dickey-Fuller Test for Fractional Returns

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

```
> # Calculate fractional VTI returns
> delta_s <- 0.1*c(1, 3, 5, 7, 9)
> re_turns <- lapply(delta_s, function(del_ta) {
+   weight_s <- (del_ta - 0:(look_back-2)) / 1:(look_back-1)
+   weight_s <- c(1, (-1)^(1:(look_back-1))*cumprod(weight_s))
+   weight_s <- rev(weight_s - mean(weight_s))
+   roll::roll_sum(clos_e, width=look_back, weights=weight_s, min_ol
+ })  # end lapply
> re_turns <- do.call(cbind, re_turns)
> re_turns <- cbind(clos_e, re_turns)
> colnames(re_turns) <- c("VTI", paste0("frac_", delta_s))
> # Calculate ADF test statistics
> adf_stats <- sapply(re_turns, function(x)
+   suppressWarnings(tseries::adf.test(x)$statistic)
+ )  # end sapply
> names(adf_stats) <- colnames(re_turns)
```




```
> # Plot dygraph of fractional VTI returns
> color_s <- colorRampPalette(c("blue", "red"))(NCOL(re_turns))
> col_names <- colnames(re_turns)
> dy_graph <- dygraphs::dygraph(re_turns["2019"], main="Fractional R
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
> for (i in 2:NROW(col_names))
+   dy_graph <- dy_graph %>%
+   dyAxis("y2", label=col_names[i], independentTicks=TRUE) %>%
+   dySeries(name=col_names[i], axis="y2", label=col_names[i], strok
> dy_graph <- dy_graph %>% dyLegend(width=500)
> dy_graph
```

Trading Volume Z-Scores

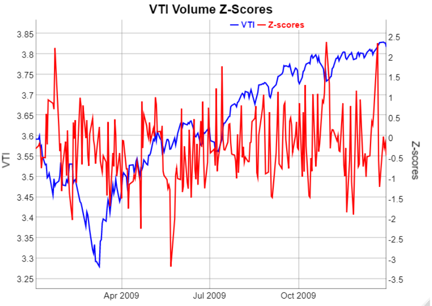
The trailing *volume z-score* is equal to the volume v_i minus the trailing average volumes $\bar{v}_i$ divided by the volatility of the volumes σ_i:

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

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

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

```
> # Calculate volume z-scores
> vol_ume <- quantmod::Vo(rutils::etf_env$VTI)
> look_back <- 21
> volume_mean <- roll::roll_mean(vol_ume, width=look_back, min_obs=
> volume_sd <- roll::roll_sd(rutils::diff_it(vol_ume), width=look_l
> volume_scores <- (vol_ume - volume_mean)/volume_sd
> # Plot histogram of volume z-scores
> x11(width=6, height=5)
> hist(volume_scores, breaks=1e2)
```




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

Volatility Z-Scores

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

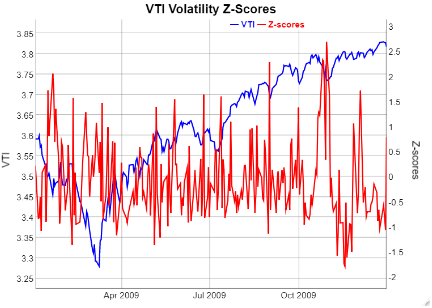
The *volatility z-score* is equal to the spot volatility v_i minus the trailing average volatility $\bar{v}_i$ divided by the standard deviation of the volatility σ_i:

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

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

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

```
> # Extract VTI log OHLC prices
> oh_lc <- log(rutils::etf_env$VTI)
> # Calculate volatility z-scores
> vol_at <- quantmod::Hi(oh_lc)-quantmod::Lo(oh_lc)
> look_back <- 21
> volat_mean <- roll::roll_mean(vol_at, width=look_back, min_obs=1)
> volat_sd <- roll::roll_sd(rutils::diff_it(vol_at), width=look_bac
> volat_scores <- ifelse(is.na(volat_sd), 0, (vol_at - volat_mean),
> # Plot histogram of volatility z-scores
> x11(width=6, height=5)
> hist(volat_scores, breaks=1e2)
> # Plot scatterplot of volume and volatility z-scores
> plot(as.numeric(volat_scores), as.numeric(volume_scores),
+      xlab="volatility z-score", ylab="volume z-score")
```




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

Centered Price Z-scores

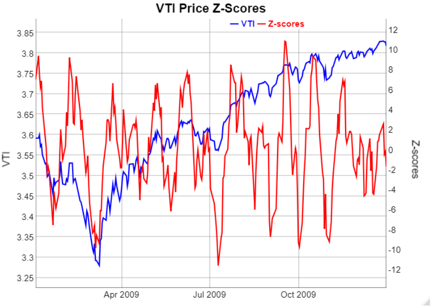
An extreme local price is a price which differs significantly from neighboring prices.

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

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

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

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




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

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

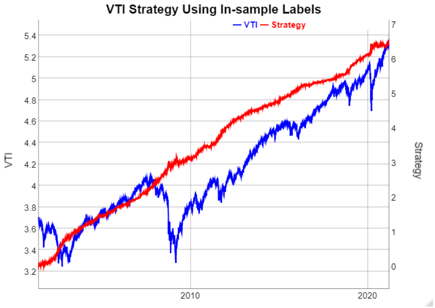
Labeling the Tops and Bottoms of Prices

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

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

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

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




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

Regression Z-Scores

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

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

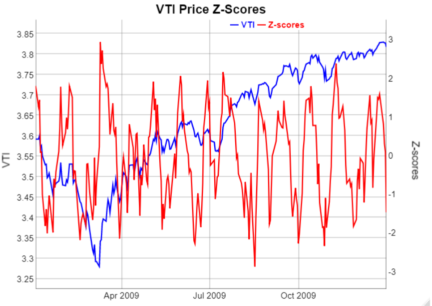
Where α and β are the *regression coefficients*, and σ_i is the standard deviation of the residuals.

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

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

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

```
> # Calculate trailing price z-scores
> date_s <- matrix(as.numeric(zoo::index(clos_e)))
> look_back <- 21
> price_scores <- drop(HighFreq::roll_zscores(res_ponse=clos_e, de
> price_scores[1:look_back] <- 0
```




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

Hampel Filter for Outlier Detection

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

$$\mathrm{MAD} = \mathrm{median}(\mathrm{abs}(p_i - \mathrm{median}(\mathbf{p})))$$

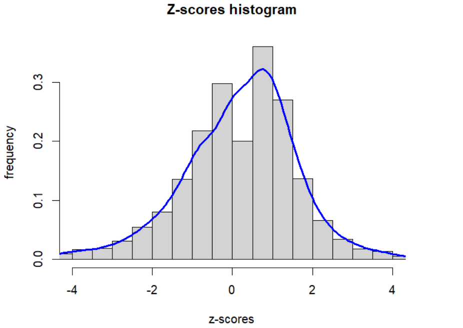
The *Hampel filter* uses the *MAD* dispersion measure to detect outliers in data.

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

$$z_i = \frac{p_i - \mathrm{median}(\mathbf{p})}{\mathrm{MAD}}$$

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

```
> # Extract time series of VTI log prices
> clos_e <- log(na.omit(rutils::etf_env$price_s$VTI))
> # Define look-back window and a half window
> look_back <- 11
> # Calculate time series of medians
> medi_an <- roll::roll_median(clos_e, width=look_back)
> # medi_an <- TTR::runMedian(clos_e, n=look_back)
> # Calculate time series of MAD
> ma_d <- HighFreq::roll_var(clos_e, look_back=look_back, method="
> # ma_d <- TTR::runMAD(clos_e, n=look_back)
> # Calculate time series of z-scores
> z_scores <- (clos_e - medi_an)/ma_d
> z_scores[1:look_back, ] <- 0
> tail(z_scores, look_back)
> range(z_scores)
```




```
> # Plot prices and medians
> dygraphs::dygraph(cbind(clos_e, medi_an), main="VTI median") %>%
+   dyOptions(colors=c("black", "red"))
> # Plot histogram of z-scores
> histo_gram <- hist(z_scores, col="lightgrey",
+   xlab="z-scores", breaks=50, xlim=c(-4, 4),
+   ylab="frequency", freq=FALSE, main="Hampel Z-scores histogram")
> lines(density(z_scores, adjust=1.5), lwd=3, col="blue")
```

One-sided and Two-sided Data Filters

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

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

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

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

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

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

```
> # Calculate one-sided Hampel z-scores
> medi_an <- roll::roll_median(clos_e, width=look_back)
> # medi_an <- TTR::runMedian(clos_e, n=look_back)
> ma_d <- HighFreq::roll_var(clos_e, look_back=look_back, method="qu
> # ma_d <- TTR::runMAD(clos_e, n=look_back)
> z_scores <- (clos_e - medi_an)/ma_d
> z_scores[1:look_back, ] <- 0
> tail(z_scores, look_back)
> range(z_scores)
> # Calculate two-sided Hampel z-scores
> half_back <- look_back %/% 2
> medi_an <- rutils::lag_it(medi_an, lagg=-half_back)
> ma_d <- rutils::lag_it(ma_d, lagg=-half_back)
> z_scores <- (clos_e - medi_an)/ma_d
> z_scores[1:look_back, ] <- 0
> tail(z_scores, look_back)
> range(z_scores)
```

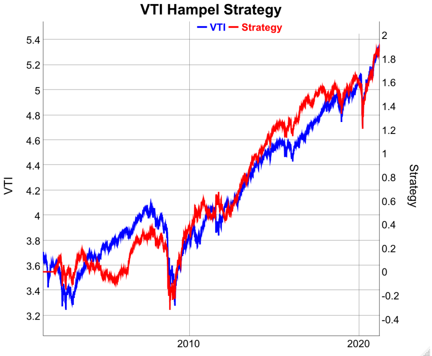
Hampel Filter Strategy

The Hampel filter strategy is a contrarian strategy that uses Hampel z-scores to establish long and short positions.

The Hampel strategy has two meta-parameters: the look-back interval and the threshold level.

The best choice of the meta-parameters can be determined through backtesting.

```
> # Calculate VTI percentage returns
> re_turns <- rutils::diff_it(clos_e)
> # Define threshold value
> thresh_old <- sum(abs(range(z_scores)))/8
> # Backtest VTI strategy
> position_s <- rep(NA_integer_, NROW(clos_e))
> position_s[1] <- 0
> position_s[z_scores < -thresh_old] <- 1
> position_s[z_scores > thresh_old] <- (-1)
> position_s <- zoo::na.locf(position_s)
> position_s <- rutils::lag_it(position_s)
> pnl_s <- cumsum(re_turns*position_s)
```




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

Reading *TAQ* Data From .csv Files

Trade and Quote (*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(HighFreq)
> # Read TAQ trade data from csv file
> ta_q <- data.table::fread(file="/Volumes/external/Develop/data/xlk
> # 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

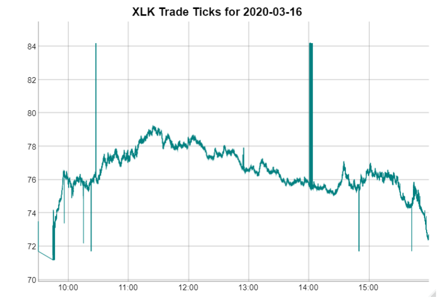
High frequency data contains *microstructure noise* in the form of *price jumps* and the *bid-ask bounce*.

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 Microstructure Noise From High Frequency Data

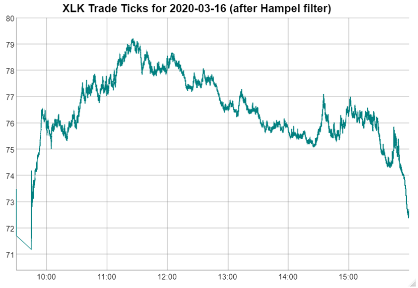
Microstructure noise can be removed from high frequency data by using a *Hampel filter*.

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

$$z_i = \frac{p_i - \mathrm{median}(\mathbf{p})}{\mathrm{MAD}}$$

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

```
> # Calculate centered Hampel filter to remove price jumps
> look_back <- 111
> half_back <- look_back %/% 2
> medi_an <- roll::roll_median(ta_q$price, width=look_back)
> # medi_an <- TTR::runMedian(ta_q$price, n=look_back)
> medi_an <- rutils::lag_it(medi_an, lagg=-half_back, pad_zeros=FAl
> ma_d <- HighFreq::roll_var(matrix(ta_q$price), look_back=look_bac
> # ma_d <- TTR::runMAD(ta_q$price, n=look_back)
> ma_d <- rutils::lag_it(ma_d, lagg=-half_back, pad_zeros=FALSE)
> # 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
```




```
> # Define discrimination threshold value
> thresh_old <- 6*mad(z_scores)
> # Remove price jumps with large z-scores
> 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)")
```

Classifying Data Outliers Using the Hampel Filter

The data points whose absolute *z-scores* exceed a *threshold value* are classified as outliers.

This procedure is a *classifier*, which classifies the prices as either good or bad data points.

If the bad data points are not labeled, then we can add jumps to the data to test the performance of the classifier.

Let the *null hypothesis* be that the given price is a good data point.

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

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

A *type I* error is the incorrect rejection of a TRUE *null hypothesis* (i.e. a "false positive"), when good data is classified as bad.

A *type II* error is the incorrect acceptance of a FALSE *null hypothesis* (i.e. a "false negative"), when bad data is classified as good.

```
> # Define discrimination threshold value
> thresh_old <- 6*mad(z_scores)
> # Calculate number of prices classified as bad data
> is_bad <- (abs(z_scores) > thresh_old)
> sum(is_bad)
> # Add 200 random price jumps into price_s
> set.seed(1121)
> n_bad <- 200
> is_jump <- logical(NROW(clos_e))
> is_jump[sample(x=NROW(is_jump), size=n_bad)] <- TRUE
> clos_e[is_jump] <- clos_e[is_jump]*
+   sample(c(0.95, 1.05), size=n_bad, replace=TRUE)
> # Plot prices and medians
> dygraphs::dygraph(cbind(clos_e, medi_an), main="VTI median") %>%
+   dyOptions(colors=c("black", "red"))
> # Calculate time series of z-scores
> medi_an <- roll::roll_median(clos_e, width=look_back)
> # medi_an <- TTR::runMedian(clos_e, n=look_back)
> ma_d <- HighFreq::roll_var(clos_e, look_back=look_back, method="qu
> # ma_d <- TTR::runMAD(clos_e, n=look_back)
> z_scores <- (clos_e - medi_an)/ma_d
> z_scores[1:look_back, ] <- 0
> # Calculate number of prices classified as bad data
> is_bad <- (abs(z_scores) > thresh_old)
> sum(is_bad)
```

Confusion Matrix of a Binary Classification Model

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

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

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

```
> # Calculate confusion matrix
> table(actual=!is_jump, forecast=!is_bad)
> sum(is_bad)
> # FALSE positive (type I error)
> sum(!is_jump & is_bad)
> # FALSE negative (type II error)
> sum(is_jump & !is_bad)
```

Let the *null hypothesis* be that the given price is a good data point.

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

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

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

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

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

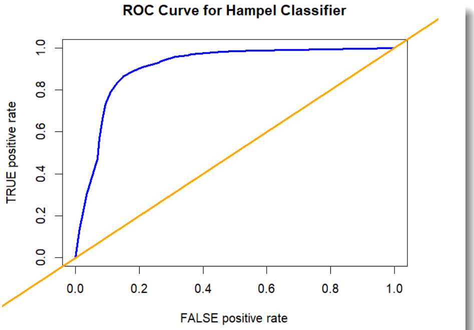
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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

The *area under the ROC curve* (AUC) is a measure of the performance of a binary classification model.

```
> # Confusion matrix as function of thresh_old
> con_fuse <- function(actu_al, z_scores, thresh_old) {
+     confu_sion <- table(!actu_al, !(abs(z_scores) > thresh_old))
+     confu_sion <- confu_sion / rowSums(confu_sion)
+     c(typeI=confu_sion[2, 1], typeII=confu_sion[1, 2])
+   }  # end con_fuse
> con_fuse(is_jump, z_scores, thresh_old=thresh_old)
> # Define vector of discrimination thresholds
> threshold_s <- seq(from=0.2, to=5.0, by=0.2)
> # Calculate error rates
> error_rates <- sapply(threshold_s, con_fuse,
+   actu_al=is_jump, z_scores=z_scores)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshold_s
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> # Calculate area under ROC curve (AUC)
> true_pos <- (1 - error_rates[, "typeII"])
> true_pos <- (true_pos + rutils::lag_it(true_pos))/2
> false_pos <- rutils::diff_it(error_rates[, "typeI"])
> abs(sum(true_pos*false_pos))
```



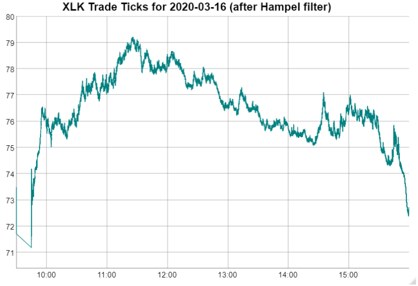

```
> # Plot ROC curve for Hampel classifier
> x11(width=6, height=5)
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Hampel Classifier",
+      type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

Improved Microstructure Noise Filtering

The filtering of microstructure noise can be improved by calculating the median prices over an interval of only 3 data points.

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

```
> # Calculate centered Hampel filter over 3 data points
> medi_an <- roll::roll_median(ta_q$price, width=3)
> medi_an[1:2] <- ta_q$price[1:2]
> medi_an <- rutils::lag_it(medi_an, lagg=-1, pad_zeros=FALSE)
> ma_d <- HighFreq::roll_var(matrix(ta_q$price), look_back=look_bacl
> ma_d <- rutils::lag_it(ma_d, lagg=-1, pad_zeros=FALSE)
> # Calculate Z-scores
> z_scores <- ifelse(ma_d > 0, (ta_q$price - medi_an)/ma_d, 0)
> range(z_scores); mad(z_scores)
> ma_d <- mad(z_scores[abs(z_scores)>0])
> hist(z_scores, breaks=2000, xlim=c(-5*ma_d, 5*ma_d))
```




```
> # Define discrimination threshold value
> thresh_old <- 6*ma_d
> 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)")
```

Homework Assignment

Required

- Study all the lecture slides in *FRE7241_Lecture_3.pdf*, and run all the code in *FRE7241_Lecture_3.R*