FRE7241 Algorithmic Portfolio Management

Lecture#3, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 20, 2022

NYU | TANDON SCHOOL OF ENGINEERING

Tests for Market Timing Skill

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

The *market timing* skill can be measured by performing a *linear regression* of a strategy's returns against a strategy with perfect *market timing* skill.

The *Merton-Henriksson* market timing test uses a linear *market timing* term:

$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma \max(0, R_m - R_f) + \varepsilon$$

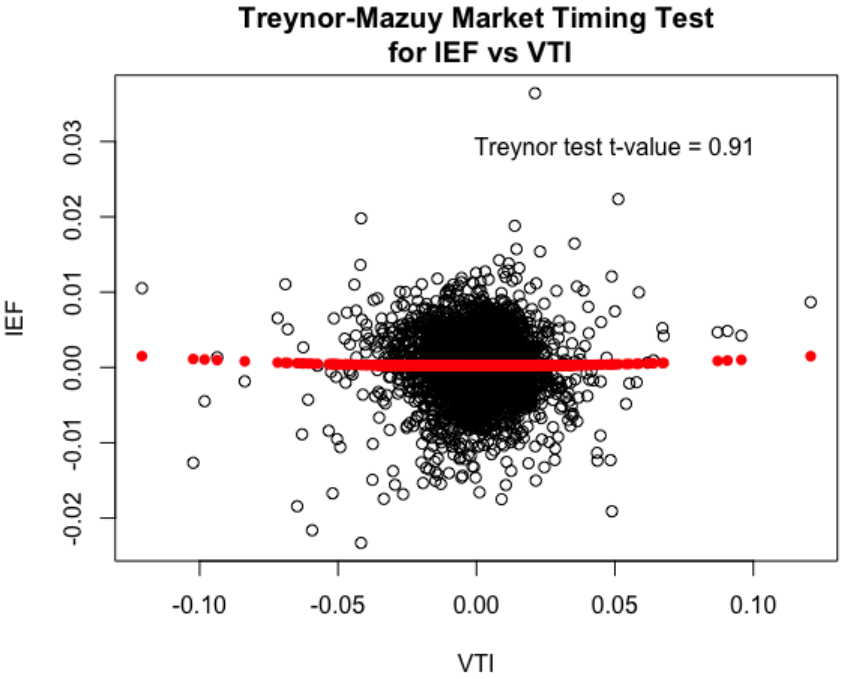
Where R are the strategy returns, R_m are the market returns, and R_f are the risk-free returns.

If the coefficient γ is statistically significant, then it's very likely due to *market timing* skill.

The *market timing* regression is a generalization of the *Capital Asset Pricing Model*.

The *Treynor-Mazuy* test uses a quadratic term, which makes it more sensitive to the magnitude of returns:

$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma(R_m - R_f)^2 + \varepsilon$$




```
> # Test if IEF can time VTI
> retsp <- na.omit(rutils::etfenv$returns[, c("IEF", "VTI")])
> vti <- retsp$VTI
> design <- cbind(retsp, 0.5*(vti+abs(vti)), vti^2)
> colnames(design)[3:4] <- c("merton", "treynor")
```

```
> # Merton-Henriksson test
> model <- lm(IEF ~ VTI + merton, data=design); summary(model)
> # Treynor-Mazuy test
> model <- lm(IEF ~ VTI + treynor, data=design); summary(model)
> # Plot residual scatterplot
> x11(width=6, height=5)
> residuals <- (design$IEF - model$coeff["VTI"]*vti)
> plot.default(x=vti, y=residuals, xlab="VTI", ylab="IEF")
> title(main="Treynor-Mazuy Market Timing Test\n for IEF vs VTI", li
> # Plot fitted (predicted) response values
> fittedv <- (model$coeff["(Intercept)"] + model$coeff["treynor"]*vt
> points.default(x=vti, y=fittedv, pch=16, col="red")
> text(x=0.05, y=0.8*max(residuals), paste("Treynor test t-value ="
```

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*.

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
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Inspect the R code of the function filter()
> filter
> # Calculate EWMA weights
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate convolution using filter()
> filtered <- filter(closep, filter=weightv,
+                    method="convolution", sides=1)
> # filter() returns time series of class "ts"
> class(filtered)
> # Get information about C_cfilter()
> getAnywhere(C_cfilter)
> # Filter using C_cfilter() over past values (sides=1).
> filterfast <- .Call(stats:::C_cfilter, closep, filter=weightv,
+                     sides=1, circular=FALSE)
> all.equal(as.numeric(filtered), filterfast, check.attributes=FALSE
> # Calculate EWMA prices using roll::roll_sum()
> weightrev <- rev(weightv)
> filtercpp <- roll::roll_sum(closep, width=look_back, weights=weigh
> all.equal(filterfast[-(1:look_back)], as.numeric(filtercpp)[-(1:lo
> # Benchmark speed of rolling calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   filterfast=.Call(stats:::C_cfilter, closep, filter=weightv, side
+   roll=roll::roll_sum(closep, width=look_back, weights=weightrev)
+   ), 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()
> nrows <- NROW(closep)
> # Calculate ARIMA coefficients and innovations
> coeff <- matrix(weightv)/4
> ncoeff <- NROW(coeff)
> innov <- matrix(rnorm(nrows))
> arimav <- filter(x=innov, filter=coeff, method="recursive")
> # Get information about C_rfilter()
> getAnywhere(C_rfilter)
> # Filter using C_rfilter() compiled C++ function directly
> arimafast <- .Call(stats:::C_rfilter, innov, coeff,
+                double(ncoeff + nrows))
> all.equal(as.numeric(arimav), arimafast[-(1:ncoeff)],
+     check.attributes=FALSE)
> # Filter using C++ code
> arimacpp <- HighFreq::sim_ar(coeff, innov)
> all.equal(arimafast[-(1:ncoeff)], drop(arimacpp))
> # Benchmark speed of the three methods
> summary(microbenchmark(
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   filterfast=.Call(stats:::C_rfilter, innov, coeff, double(ncoeff
+   Rcpp=HighFreq::sim_ar(coeff, innov)
+   ), 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.

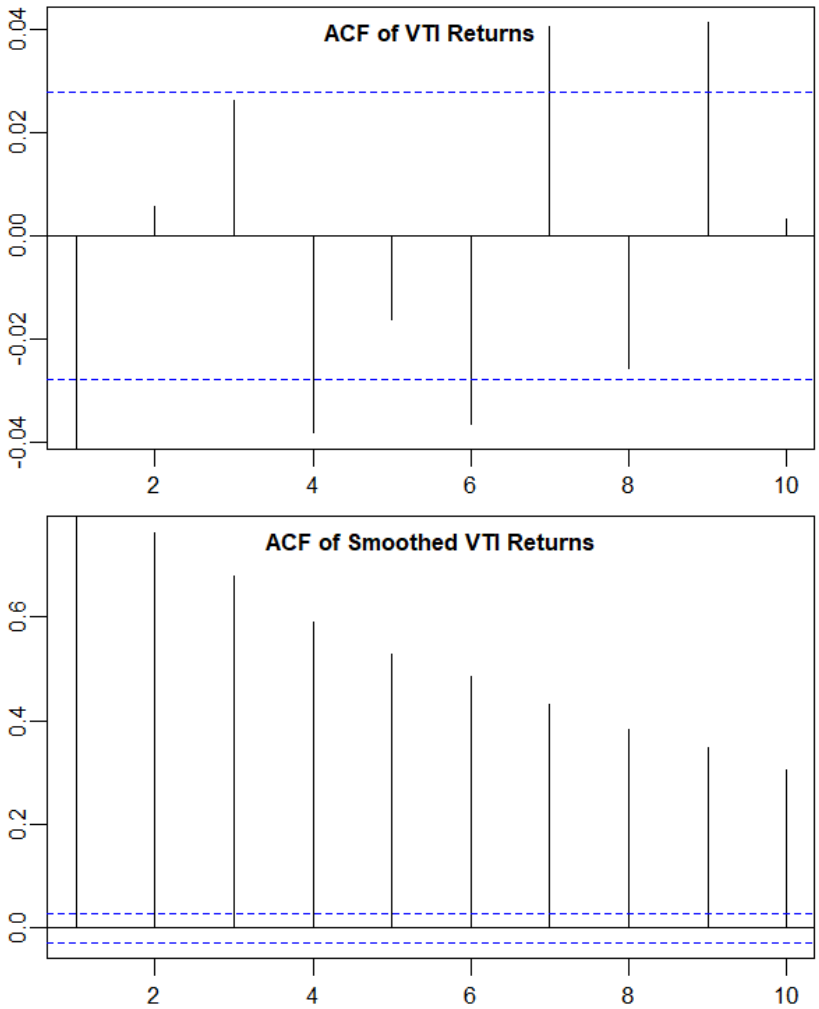

```
> # Calculate trailing EWMA prices using roll::roll_sum()
> look_back <- 21
> weightv <- exp(-0.1*1:look_back)
> weightv <- weightv/sum(weightv)
> weightrev <- rev(weightv)
> filtered <- roll::roll_sum(closep, width=NROW(weightv), weights=
> # Copy warmup period
> filtered[1:look_back] <- closep[1:look_back]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

```
> # Calculate centered EWMA prices using roll::roll_sum()
> weightv <- c(weightrev, weightv[-1])
> weightv <- weightv/sum(weightv)
> filtered <- roll::roll_sum(closep, width=NROW(weightv), weights=we
> # Copy warmup period
> filtered[1:(2*look_back)] <- closep[1:(2*look_back)]
> # Center the data
> filtered <- rutils::lagit(filtered, -(look_back-1), pad_zeros=FALS
> # Combine prices with smoothed prices
> prices <- cbind(closep, filtered)
> colnames(prices)[2] <- "VTI Smooth"
> # Calculate standard deviations of returns
> sapply(rutils::diffit(prices), sd)
> # Plot dygraph
> dygraphs::dygraph(prices["2009"], main="VTI Prices and Centered Sm
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
```

Autocorrelations of Smoothed Time Series

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

```
> # Calculate VTI log returns
> retsp <- rutils::diffit(prices)
> # 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(retsp[, 1], lag=10, xlab="")
> title(main="ACF of VTI Returns", line=-1)
> # Plot ACF of smoothed VTI returns
> rutils::plot_acf(retsp[, 2], lag=10, xlab="")
> title(main="ACF of Smoothed VTI Returns", line=-1)
```



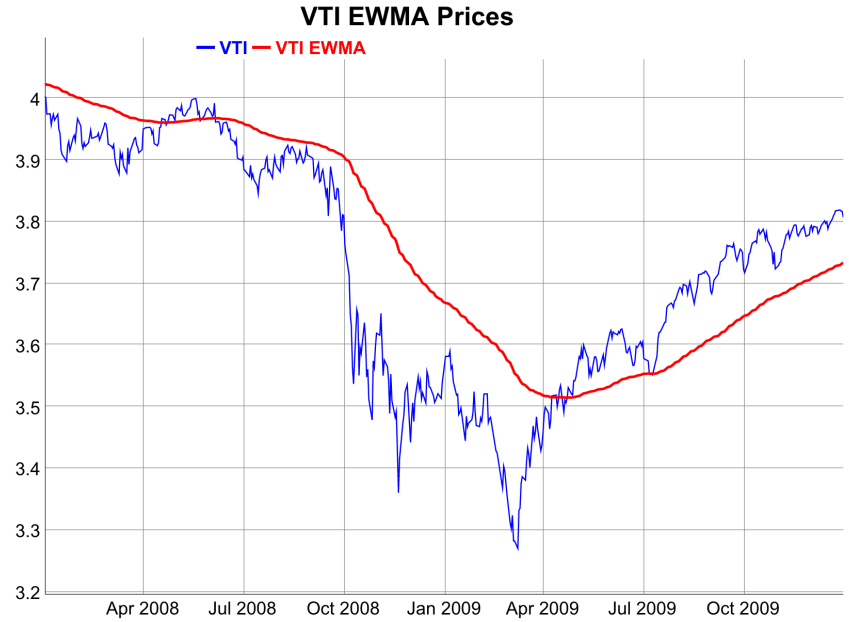
EWMA Price Technical Indicator

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

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

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

The function `HighFreq::roll_wsum()` calculates the convolution of a time series with a vector of weights.




```
> # Extract log VTI prices
> ohlc <- log(rutils::etfenv$VTI)
> closep <- quantmod::Cl(ohlc)
> colnames(closep) <- "VTI"
> nrows <- NROW(closep)
> # Calculate EWMA weights
> look_back <- 333
> lambda <- 0.9
> weightv <- lambda^(1:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate EWMA prices as the convolution
> ewmacpp <- HighFreq::roll_wsum(closep, weights=weightv)
> prices <- cbind(closep, ewmacpp)
> colnames(prices) <- c("VTI", "VTI EWMA")
```

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

Recursive EWMA Price Indicator

The *EWMA* prices can be calculated recursively as follows:

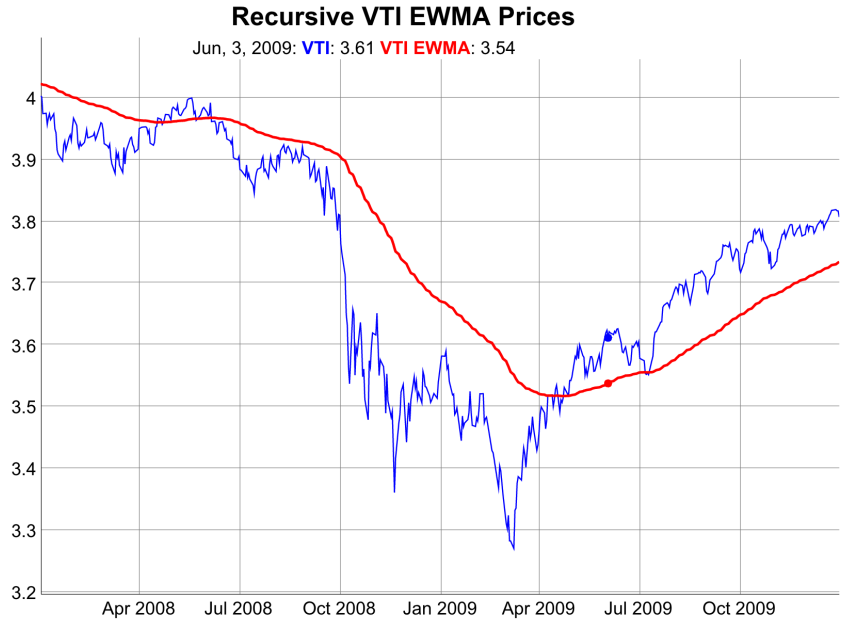
$$p_i^{EWMA} = (1 - \lambda)p_i + \lambda p_{i-1}^{EWMA}$$

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

The recursive *EWMA* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

The compiled C++ function `stats:::C_rfilter()` calculates the exponentially weighted moving average prices recursively.

The function `HighFreq::run_mean()` calculates the exponentially weighted moving average prices recursively.




```
> # Calculate EWMA prices recursively using C++ code
> ewmar <- .Call(stats:::C_rfilter, closep, lambda, c(as.numeric(cl
> # Or R code
> # ewmar <- filter(closep, filter=lambda, init=as.numeric(closep[1
> ewmar <- (1-lambda)*ewmar
> # Calculate EWMA prices recursively using RcppArmadillo
> ewmacpp <- HighFreq::run_mean(closep, lambda=lambda, weights=0)
> all.equal(drop(ewmacpp), ewmar)
> # Compare the speed of C++ code with RcppArmadillo
> library(microbenchmark)
> summary(microbenchmark(
+   filtercpp=HighFreq::run_mean(closep, lambda=lambda, weights=0),
+   rfilter=.Call(stats:::C_rfilter, closep, lambda, c(as.numeric(c
+   times=10))[, c(1, 4, 5)]
```

```
> # Dygraphs plot with custom line colors
> prices <- cbind(closep, ewmacpp)
> colnames(prices) <- c("VTI", "VTI EWMA")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="Recursive VTI EWMA Prices"
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dyLegend(show="always", width=500)
> # Standard plot of  EWMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> colorv <- c("blue", "red")
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(prices["2009"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(prices),
+  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

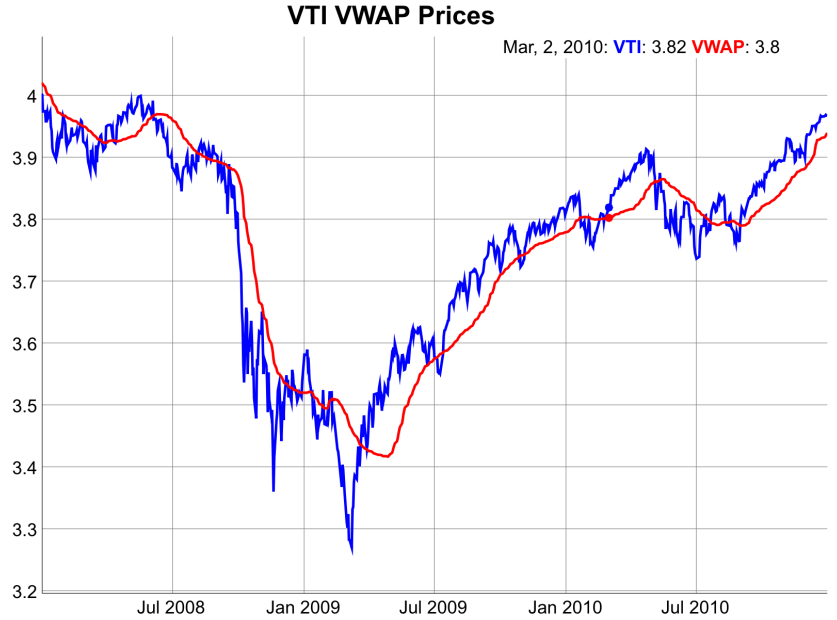
The Volume-Weighted Average Price (*VWAP*) is defined as the sum of prices multiplied by trading volumes, divided by the sum of volumes:

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

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

The drawback of the *VWAP* indicator is that it applies large weights to prices far in the past.

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




```
> # Calculate log OHLC prices and volumes
> ohlc <- rutils::etfenv$VTI
> closep <- log(quantmod::Cl(ohlc))
> colnames(closep) <- "VTI"
> volumes <- quantmod::Vo(ohlc)
> colnames(volumes) <- "Volume"
> nrows <- NROW(closep)
> # Calculate the VWAP prices
> look_back <- 21
> vwap <- roll::roll_sum(closep*volumes, width=look_back, min_obs=
> volume_roll <- roll::roll_sum(volumes, width=look_back, min_obs=
> vwap <- vwap/volume_roll
> colnames(vwap) <- "VWAP"
> prices <- cbind(closep, vwap)
```

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

Recursive VWAP Price Indicator

The *VWAP* prices p^{VWAP} can also be calculated as the ratio of the volume weighted prices μ^{pv} divided by the mean trading volumes μ^{v}:

$$p^{VWAP} = \frac{\mu^{pv}}{\mu^{v}}$$

The volume weighted prices μ^{pv} and the mean trading volumes μ^{v} are both calculated recursively:

$$\mu_i^{v} = (1 - \lambda)v_i + \lambda\mu_{i-1}^{v}$$

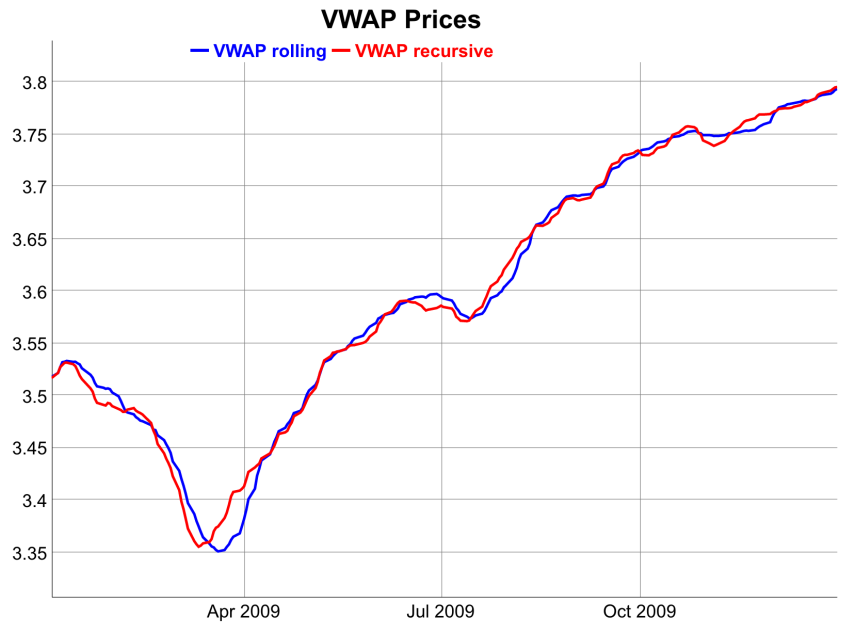
$$\mu_i^{pv} = (1 - \lambda)v_i p_i + \lambda\mu_{i-1}^{pv}$$

The recursive *VWAP* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

The advantage of the recursive *VWAP* indicator is that it gradually "forgets" about large trading volumes far in the past.

The compiled C++ function `stats:::C_rfilter()` calculates the weighted running values recursively.

The function `HighFreq::run_mean()` also calculates the weighted running values recursively.




```
> # Calculate VWAP prices recursively using C++ code
> lambda <- 0.98
> volumer <- .Call(stats:::C_rfilter, volumes, lambda, c(as.numeric(
> pricer <- .Call(stats:::C_rfilter, volumes*closep, lambda, c(as.nu
> vwapr <- pricer/volumer
> # Calculate VWAP prices recursively using RcppArmadillo
> vwapcpp <- HighFreq::run_mean(closep, lambda=lambda, weights=volum
> all.equal(vwapr, drop(vwapcpp))
> # Dygraphs plot the VWAP prices
> prices <- xts(cbind(vwap, vwapr), zoo::index(ohlc))
> colnames(prices) <- c("VWAP rolling", "VWAP recursive")
> dygraphs::dygraph(prices["2009"], main="VWAP Prices") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Smooth Asset Returns

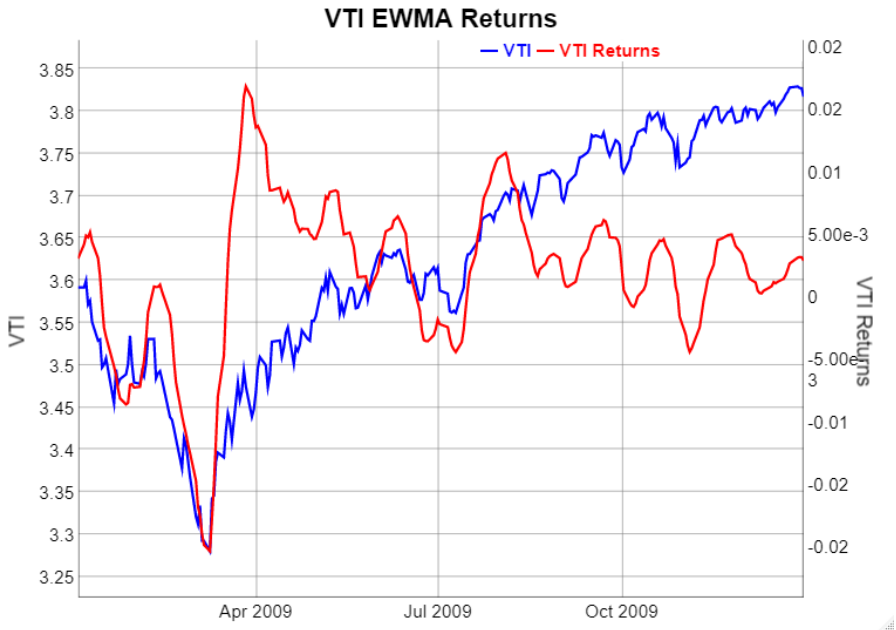
Asset returns are calculated by filtering prices through a *differencing* filter.

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
> lambda <- 0.1
> weightv <- exp(lambda*1:look_back)
> weightv <- weightv/sum(weightv)
> ewma_fast <- roll::roll_sum(closep, width=look_back, weights=wei
> lambda <- 0.05
> weightv <- exp(lambda*1:look_back)
> weightv <- weightv/sum(weightv)
> ewma_slow <- roll::roll_sum(closep, width=look_back, weights=wei
```

```
> # Calculate VTI prices
> ewmadiff <- (ewma_fast - ewma_slow)
> prices <- cbind(closep, ewmadiff)
> symbol <- "VTI"
> colnames(prices) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main=paste(symbol, "EWMA Returns
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
```

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: $r_i = (1 - L)p_i$.

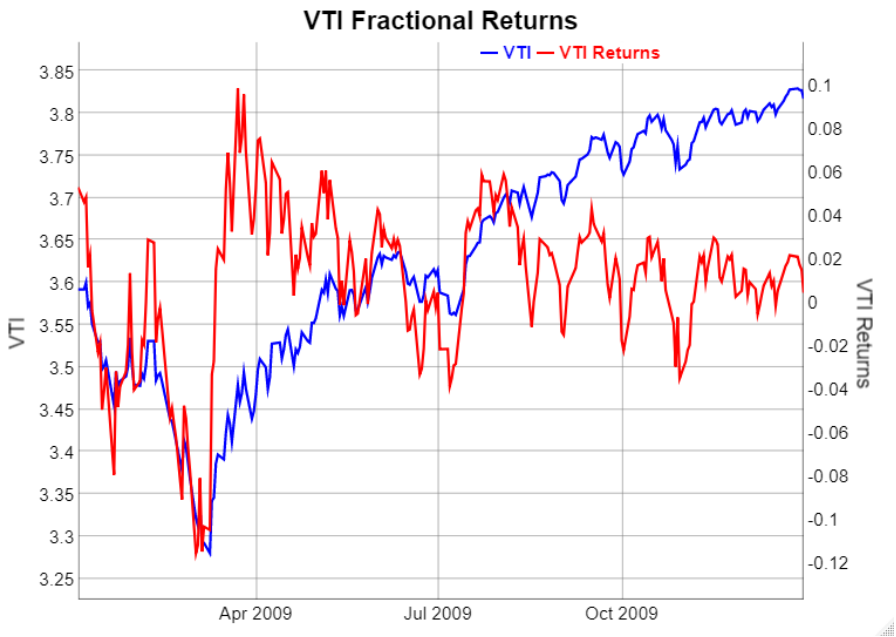
The simple returns can be generalized to the fractional returns by raising the returns operator to some power $\delta < 1$:

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

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

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

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
> deltav <- 0.1
> weightv <- (deltav - 0:(look_back-2)) / 1:(look_back-1)
> weightv <- (-1)^(1:(look_back-1))*cumprod(weightv)
> weightv <- c(1, weightv)
> weightv <- (weightv - mean(weightv))
> weightv <- rev(weightv)
> # Calculate fractional VTI returns
> retsp <- roll::roll_sum(closep, width=look_back, weights=weightv,
> prices <- cbind(closep, retsp)
> symbol <- "VTI"
> colnames(prices) <- c(symbol, paste(symbol, "Returns"))
> # Plot dygraph of VTI Returns
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main=paste(symbol, "Fractional
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
```

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).

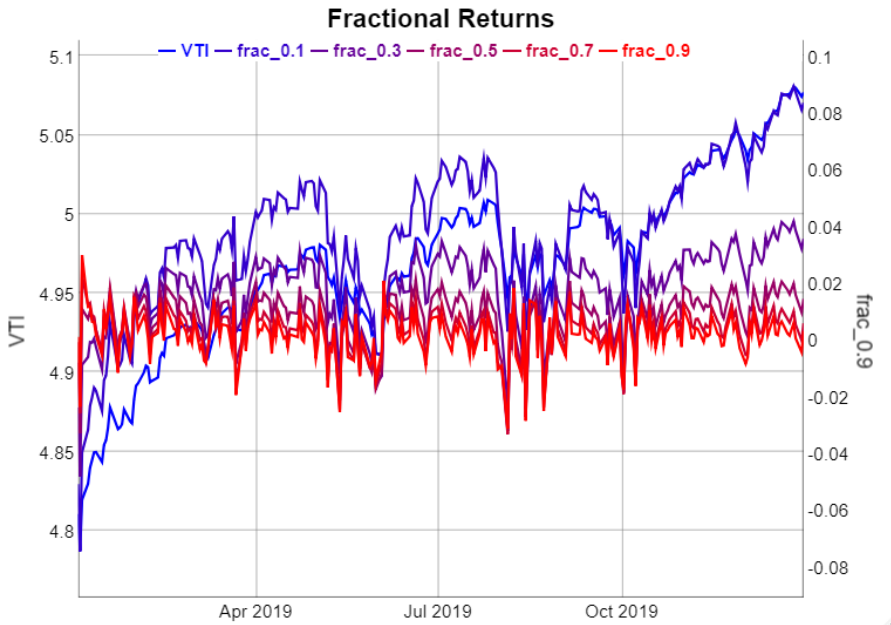
The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

```
> # Calculate VTI log returns
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> retsp <- rutils::diffit(closep)
> # Perform ADF test for prices
> tseries::adf.test(closep)
> # Perform ADF test for returns
> tseries::adf.test(retsp)
```

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
> deltav <- 0.1*c(1, 3, 5, 7, 9)
> retsp <- lapply(deltav, function(deltav) {
+   weightv <- (deltav - 0:(look_back-2)) / 1:(look_back-1)
+   weightv <- c(1, (-1)^(1:(look_back-1))*cumprod(weightv))
+   weightv <- rev(weightv - mean(weightv))
+   roll::roll_sum(closep, width=look_back, weights=weightv, min_obs
+ })  # end lapply
> retsp <- do.call(cbind, retsp)
> retsp <- cbind(closep, retsp)
> colnames(retsp) <- c("VTI", paste0("frac_", deltav))
> # Calculate ADF test statistics
> adfstats <- sapply(retsp, function(x)
+   suppressWarnings(tseries::adf.test(x)$statistic)
+ )  # end sapply
> names(adfstats) <- colnames(retsp)
```




```
> # Plot dygraph of fractional VTI returns
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(retsp))
> colnamev <- colnames(retsp)
> dyplot <- dygraphs::dygraph(retsp["2019"], main="Fractional Return
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
> for (i in 2:NROW(colnamev))
+   dyplot <- dyplot %>%
+   dyAxis("y2", label=colnamev[i], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[i], axis="y2", label=colnamev[i], strokeW
> dyplot <- dyplot %>% dyLegend(width=500)
> dyplot
```

Trading Volume Z-Scores

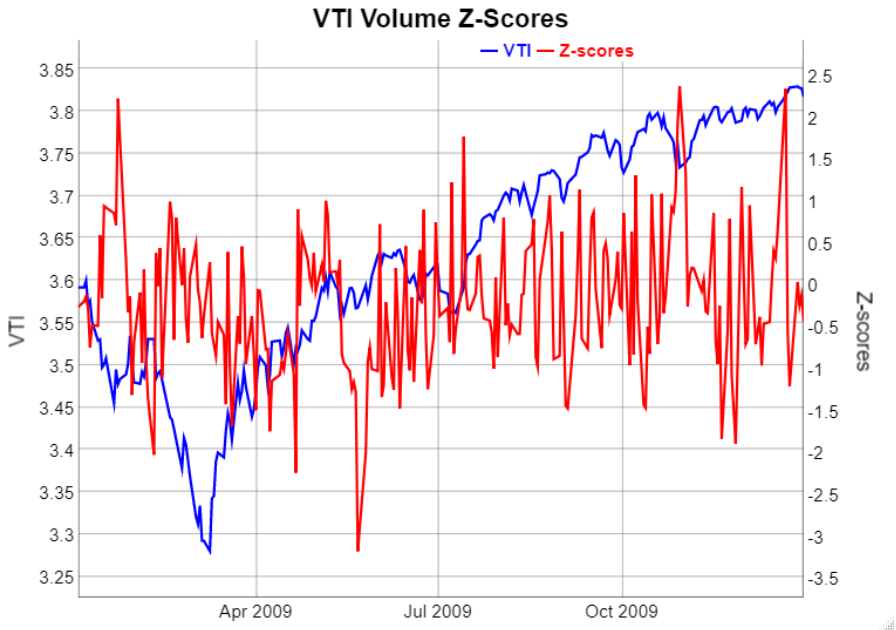
The trailing *volume z-score* is equal to the volume v_i minus the trailing average volumes $\bar{v}_i$ divided by the volatility of the volumes σ_i:

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

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
> volumes <- quantmod::Vo(rutils::etfenv$VTI)
> look_back <- 21
> volumem <- roll::roll_mean(volumes, width=look_back, min_obs=1)
> volumesd <- roll::roll_sd(rutils::diffit(volumes), width=look_ba
> volumesd[1] <- 0
> volumez <- ifelse(volumesd > 0, (volumes - volumem)/volumesd, 0)
> # Plot histogram of volume z-scores
> x11(width=6, height=5)
> hist(volumez, breaks=1e2)
```




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

Volatility Z-Scores

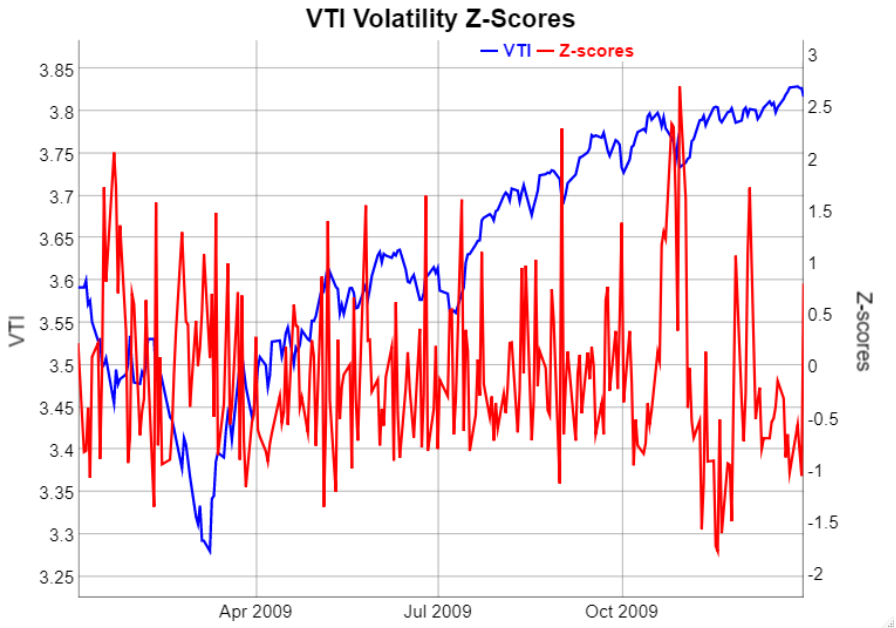
The difference between high and low prices is a proxy for the spot volatility in a bar of data.

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
> ohlc <- log(rutils::etfenv$VTI)
> # Calculate volatility z-scores
> volat <- quantmod::Hi(ohlc)-quantmod::Lo(ohlc)
> look_back <- 21
> volatm <- roll::roll_mean(volat, width=look_back, min_obs=1)
> volatsd <- roll::roll_sd(rutils::diffit(volat), width=look_back,
> volatsd[1] <- 0
> volatz <- ifelse(volatsd > 0, (volat - volatm)/volatsd, 0)
> # Plot histogram of volatility z-scores
> x11(width=6, height=5)
> hist(volatz, breaks=1e2)
> # Plot scatterplot of volume and volatility z-scores
> plot(as.numeric(volatz), as.numeric(volumez),
+      xlab="volatility z-score", ylab="volume z-score")
> regmod <- lm(volatz ~ volumez)
> abline(regmod, col="red", lwd=3)
```

```
> # Plot dygraph of VTI volatility z-scores
> closep <- quantmod::Cl(ohlc)
> prices <- cbind(closep, volatz)
> colnames(prices) <- c("VTI", "Z-scores")
> colnamev <- colnames(prices)
> dygraphs::dygraph(prices["2009"], main="VTI Volatility Z-Scores")
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
```

Centered Price Z-scores

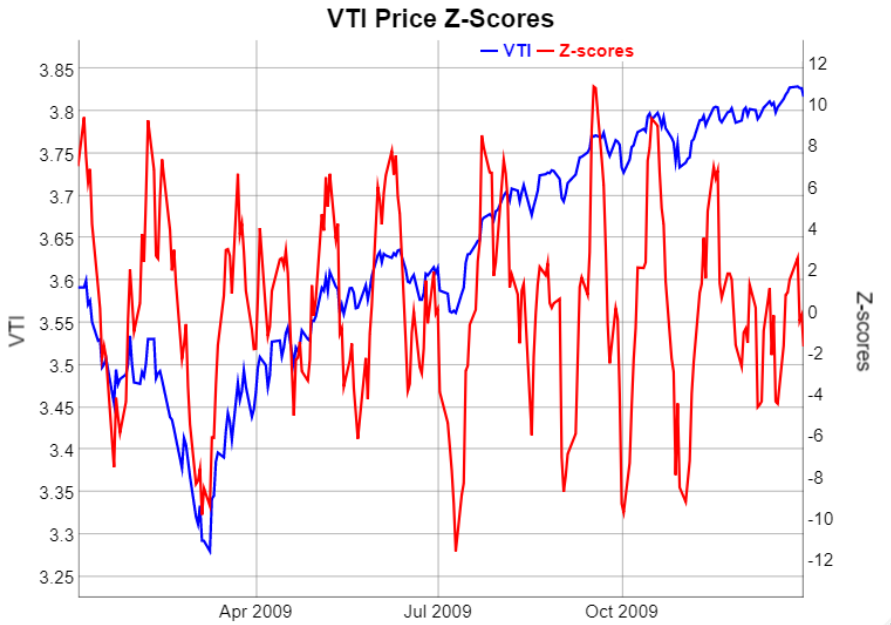
An extreme local price is a price which differs significantly from neighboring prices.

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

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

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

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




```
> # Calculate the centered volatility
> look_back <- 21
> half_back <- look_back %/% 2
> retsp <- rutils::diffit(closep)
> volat <- roll::roll_sd(retsp, width=look_back, min_obs=1)
> volat <- rutils::lagit(volat, lagg=(-half_back))
> # Calculate the z-scores of prices
> pricez <- (2*closep -
+   rutils::lagit(closep, half_back, pad_zeros=FALSE) -
+   rutils::lagit(closep, -half_back, pad_zeros=FALSE))
> pricez <- ifelse(volat > 0, pricez/volat, 0)
```

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

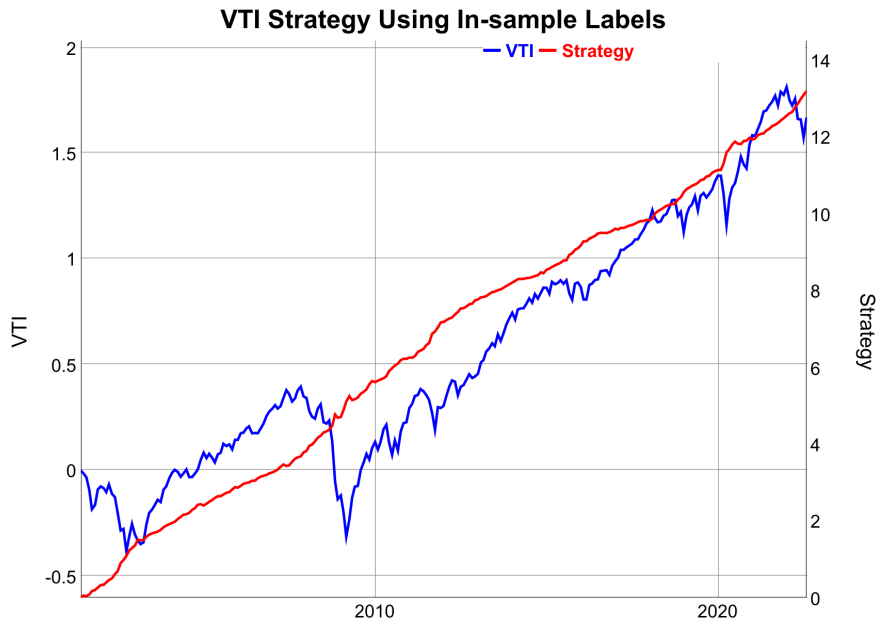
Labeling the Tops and Bottoms of Prices

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

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

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

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




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

```
> # Plot dygraph of in-sample VTI strategy
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="VTI Strategy Using In-sample Labels") %>%
+   dyAxis("y", label="VTI", independentTicks=TRUE) %>%
+   dyAxis("y2", label="Strategy", independentTicks=TRUE) %>%
+   dySeries(name="VTI", axis="y", label="VTI", strokeWidth=2, col="
+   dySeries(name="Strategy", axis="y2", label="Strategy", strokeWi
```

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:

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

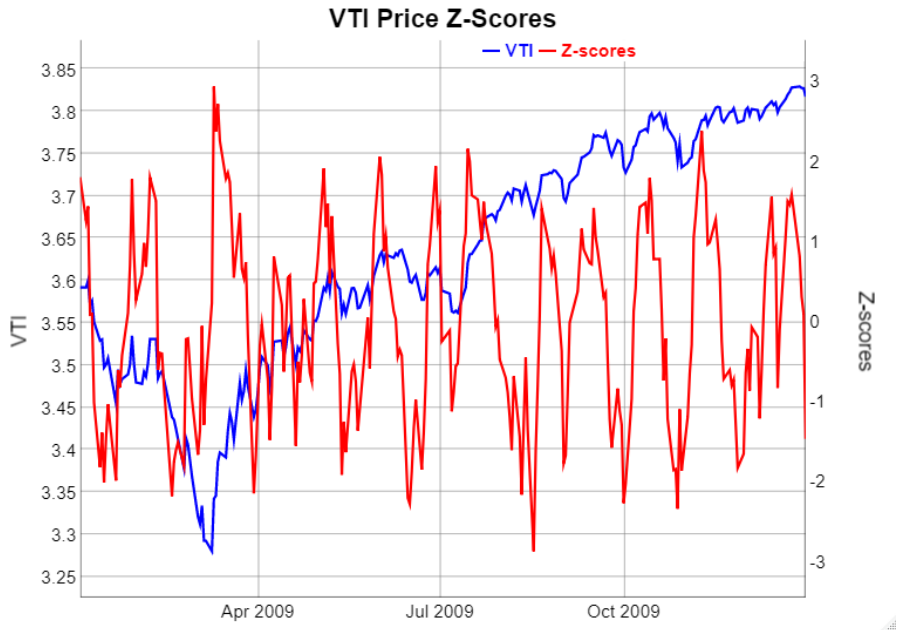
Where α and β are the *regression coefficients*, and σ_i is the standard deviation of the residuals.

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
> dates <- matrix(as.numeric(zoo::index(closep)))
> look_back <- 21
> pricez <- drop(HighFreq::roll_zscores(response=closep, design=dat
> pricez[1:look_back] <- 0
```




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

Hampel Filter for Outlier Detection

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

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

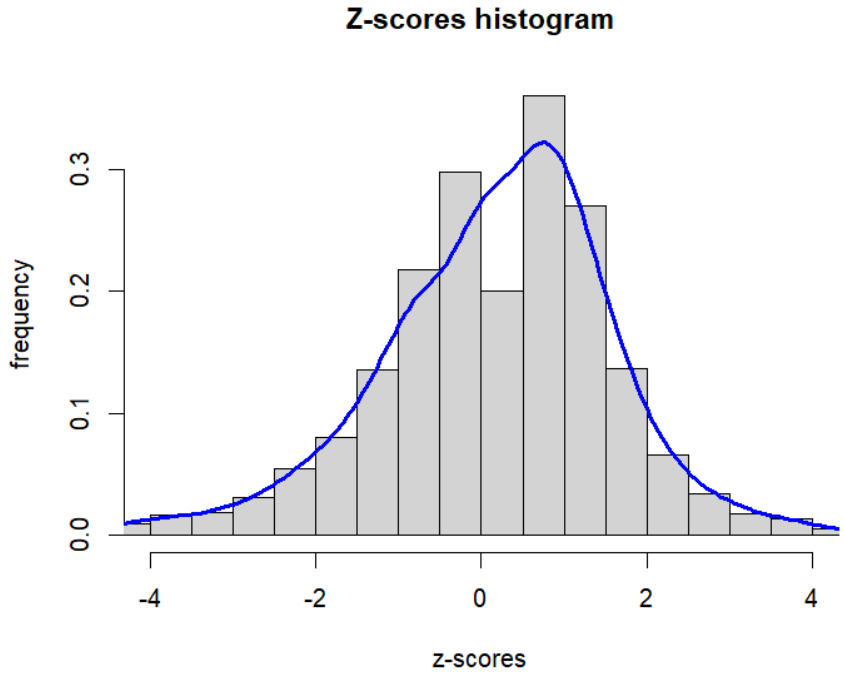
The *Hampel filter* uses the *MAD* dispersion measure to detect outliers in data.

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

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

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
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Define look-back window
> look_back <- 11
> # Calculate time series of medians
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> # Calculate time series of MAD
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="
> # madv <- TTR::runMAD(closep, n=look_back)
> # Calculate time series of z-scores
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
```




```
> # Plot prices and medians
> dygraphs::dygraph(cbind(closep, medianv), main="VTI median") %>%
+   dyOptions(colors=c("black", "red"))
> # Plot histogram of z-scores
> histp <- hist(zscores, col="lightgrey",
+   xlab="z-scores", breaks=50, xlim=c(-4, 4),
+   ylab="frequency", freq=FALSE, main="Hampel Z-scores histogram")
> lines(density(zscores, 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="nonparametric"` 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
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="no
> # madv <- TTR::runMAD(closep, n=look_back)
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
> # Calculate two-sided Hampel z-scores
> half_back <- look_back %/% 2
> medianv <- rutils::lagit(medianv, lagg=-half_back)
> madv <- rutils::lagit(madv, lagg=-half_back)
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
```

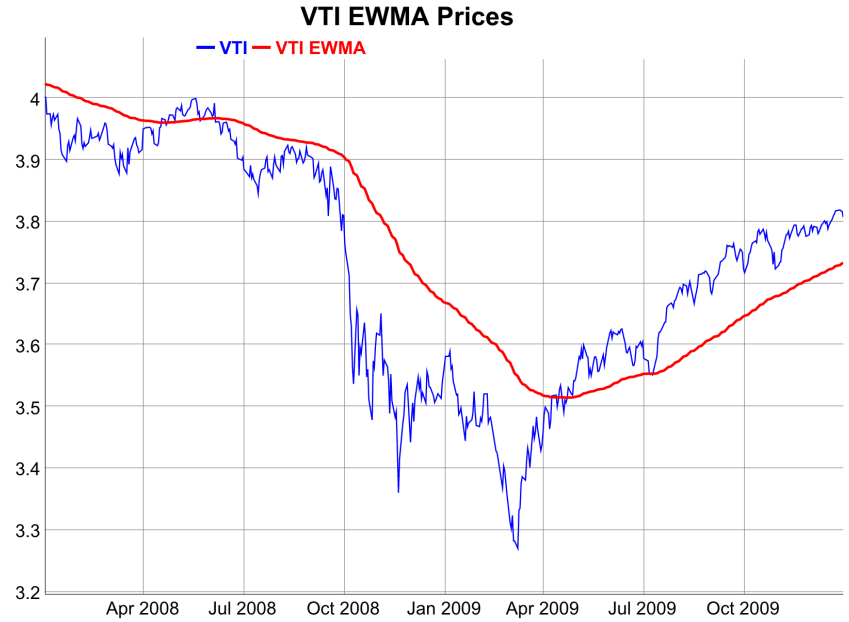
EWMA Price Technical Indicator

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

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

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

The function `HighFreq::roll_wsum()` calculates the convolution of a time series with a vector of weights.




```
> # Extract log VTI prices
> ohlc <- log(rutils::etfenv$VTI)
> closep <- quantmod::Cl(ohlc)
> colnames(closep) <- "VTI"
> nrows <- NROW(closep)
> # Calculate EWMA weights
> look_back <- 333
> lambda <- 0.984
> weightv <- lambda^(0:look_back)
> weightv <- weightv/sum(weightv)
> # Calculate EWMA prices as the convolution
> ewmacpp <- HighFreq::roll_wsum(closep, weights=weightv)
> pricets <- cbind(closep, ewmacpp)
> colnames(pricets) <- c("VTI", "VTI EWMA")
```

```
> # Dygraphs plot with custom line colors
> colnamev <- colnames(pricets)
> dygraphs::dygraph(pricets["2008/2009"], main="VTI EWMA Prices") %>
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dyLegend(show="always", width=500)
> # Standard plot of  EWMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> colorv <- c("blue", "red")
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(pricets["2009"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(pricets),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

Recursive EWMA Price Indicator

The *EWMA* prices can be calculated recursively as follows:

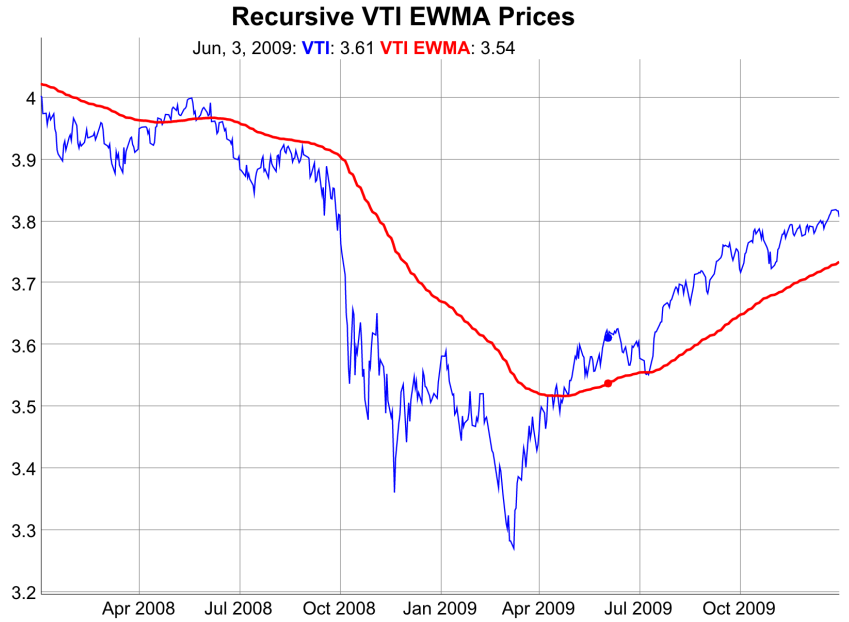
$$p_i^{EWMA} = (1 - \lambda)p_i + \lambda p_{i-1}^{EWMA}$$

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

The recursive *EWMA* prices are slightly different from those calculated as a convolution, because the convolution uses a fixed look-back interval.

The compiled `C++` function `stats:::C_rfilter()` calculates the exponentially weighted moving average prices recursively.

The function `HighFreq::run_mean()` calculates the exponentially weighted moving average prices recursively.




```
> # Calculate EWMA prices recursively using C++ code
> ewmar <- .Call(stats:::C_rfilter, closep, lambda, c(as.numeric(c
> # Or R code
> # ewmar <- filter(closep, filter=lambda, init=as.numeric(closep[
> ewmar <- (1-lambda)*ewmar
> # Calculate EWMA prices recursively using RcppArmadillo
> ewmacpp <- HighFreq::run_mean(closep, lambda=lambda, weights=0)
> all.equal(drop(ewmacpp), ewmar)
> # Compare the speed of C++ code with RcppArmadillo
> library(microbenchmark)
> summary(microbenchmark(
+   run_mean=HighFreq::run_mean(closep, lambda=lambda, weights=0),
+   rfilter=.Call(stats:::C_rfilter, closep, lambda, c(as.numeric(
+   times=10))[, c(1, 4, 5)]
```

```
> # Dygraphs plot with custom line colors
> pricets <- cbind(closep, ewmacpp)
> colnames(pricets) <- c("VTI", "VTI EWMA")
> colnamev <- colnames(pricets)
> dygraphs::dygraph(pricets["2008/2009"], main="Recursive VTI EWMA P
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dyLegend(show="always", width=500)
> # Standard plot of  EWMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> colorv <- c("blue", "red")
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(pricets["2009"], theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> legend("topleft", legend=colnames(pricets),
+  inset=0.1, bg="white", lty=1, lwd=6, cex=0.8,
+  col=plot_theme$col$line.col, bty="n")
```

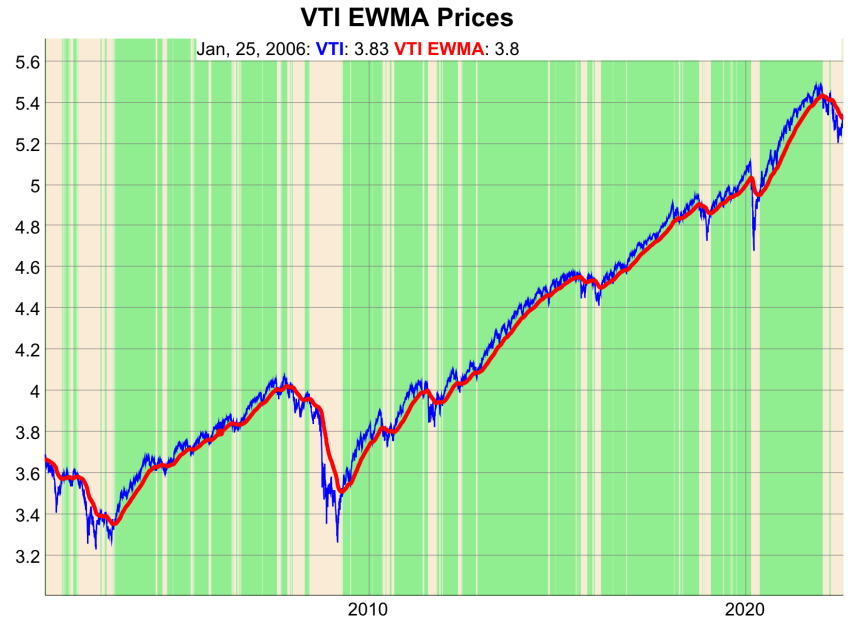
Simulating the EWMA Crossover Strategy

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

If the stock price is above the *EWMA* price, then the strategy switches to long \$1 dollar risk position, and if it is below, to short risk \$1 dollar.

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

The strategy is therefore always either long \$1 dollar risk, or short \$1 dollar risk.




```
> # Calculate positions, either: -1, 0, or 1
> indic <- sign(closep - ewmacpp)
> posit <- rutils::lagit(indic, lagg=1)
> # Create colors for background shading
> crossd <- (rutils::diffit(posit) != 0)
> shadev <- posit[crossd]
> crossd <- c(zoo::index(shadev), end(posit))
> shadev <- ifelse(drop(zoo::coredata(shadev)) == 1, "lightgreen",
> # Create dygraph object without plotting it
> dyplot <- dygraphs::dygraph(pricets, main="VTI EWMA Prices") %>%
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, c
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=3, c
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>% dyShading(from=crossd[i], to=crossd[i+1], color=shadev[i])
+ }  # end for
> # Plot the dygraph object
> dyplot
```

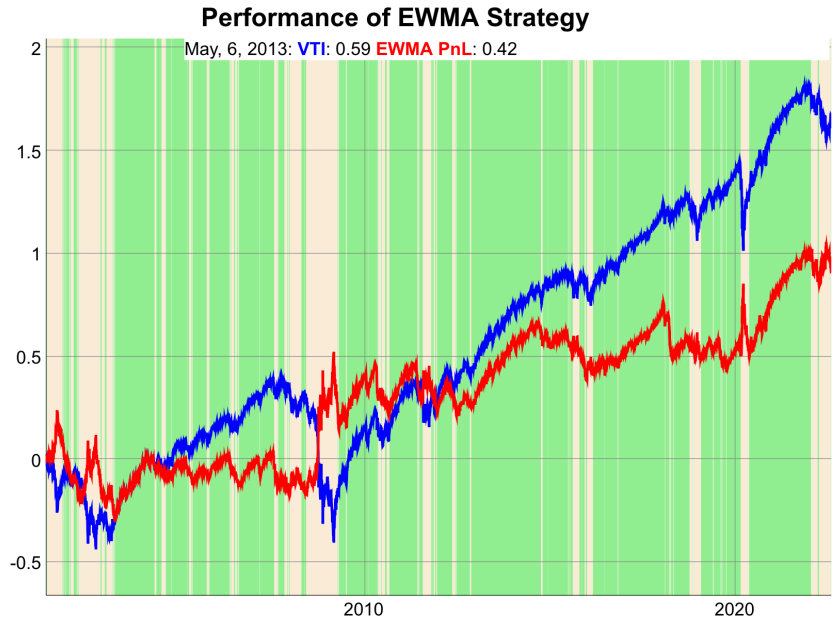
```
> # Standard plot of EWMA prices with position shading
> x11(width=6, height=5)
> quantmod::chart_Series(pricets, theme=plot_theme,
+        lwd=2, name="VTI EWMA Prices")
> add_TA(posit > 0, on=-1, col="lightgreen", border="lightgreen")
> add_TA(posit < 0, on=-1, col="lightgrey", border="lightgrey")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

Performance of the EWMA Crossover Strategy

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

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

```
> # Calculate daily profits and losses of EWMA strategy
> retsp <- rutils::diffit(closep)  # VTI returns
> pnls <- retsp*posit
> colnames(pnls) <- "EWMA"
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "EWMA PnL")
> # Annualized Sharpe ratio of EWMA strategy
> sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(wealthv)[1, 2]
> # Plot dygraph of EWMA strategy wealth
> # Create dygraph object without plotting it
> colorv <- c("blue", "red")
> dyplot <- dygraphs::dygraph(cumsum(wealthv), main="Performance o
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>%
+ dyShading(from=crossd[i], to=crossd[i+1], color=shadev[i])
+ }  # end for
> # Plot the dygraph object
> dyplot
```



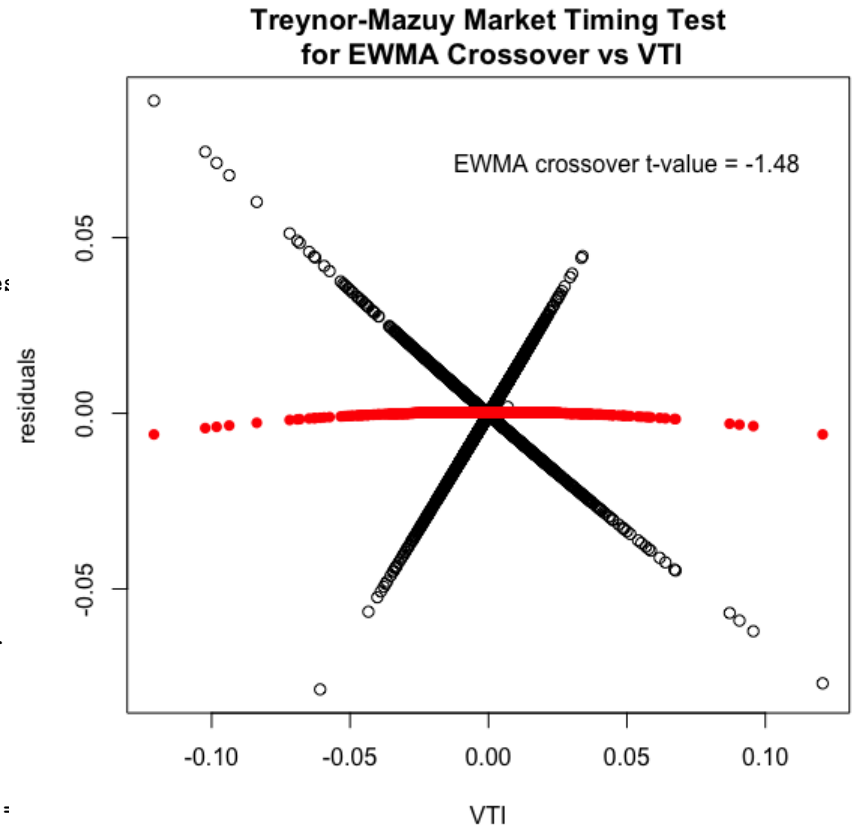

```
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(wealthv), theme=plot_theme,
+        name="Performance of EWMA Strategy")
> add_TA(posit > 0, on=-1, col="lightgreen", border="lightgreen")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

EWMA Crossover Strategy Market Timing Skill

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

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

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



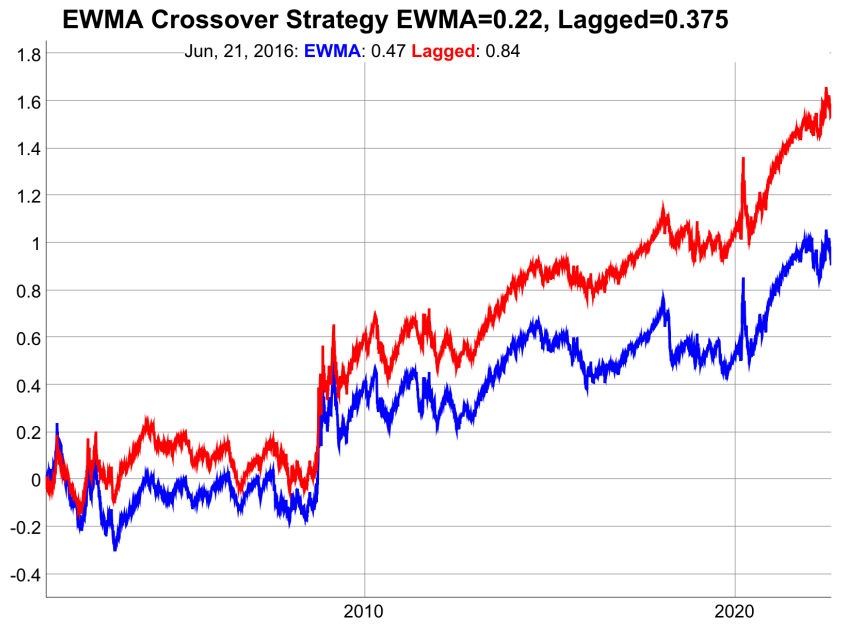
EWMA Crossover Strategy With Lag

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

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

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

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




```
> wealthv <- cbind(pnls, pnlslag)
> colnames(wealthv) <- c("EWMA", "Lagged")
> # Annualized Sharpe ratios of EWMA strategies
> sharper <- sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
> # Plot both strategies
> dygraphs::dygraph(cumsum(wealthv), main=paste("EWMA Crossover Stra
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

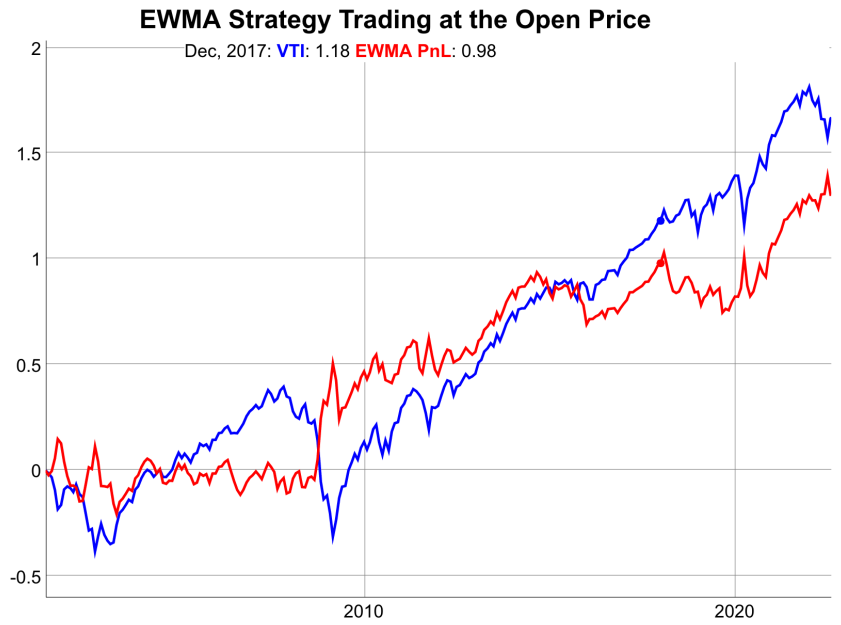
EWMA Strategy Trading at the Open Price

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

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

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

```
> # Calculate positions, either: -1, 0, or 1
> indic <- sign(closep - ewmacpp)
> posit <- rutils::lagit(indic, lagg=1)
> # Calculate daily pnl for days without trades
> pnls <- retsp*posit
> # Determine trade dates right after EWMA has crossed prices
> crossd <- which(rutils::diffit(posit) != 0)
> # Calculate realized pnl for days with trades
> openp <- quantmod::Op(ohlc)
> closelag <- rutils::lagit(closep)
> poslag <- rutils::lagit(posit)
> pnls[crossd] <- poslag[crossd]*(openp[crossd] - closelag[crossd])
> # Calculate unrealized pnl for days with trades
> pnls[crossd] <- pnls[crossd] +
+   posit[crossd]*(closep[crossd] - openp[crossd])
> # Calculate the wealth
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "EWMA PnL")
> # Annualized Sharpe ratio of EWMA strategy
> sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(wealthv)[1, 2]
```




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

EWMA Crossover Strategy With Transaction Costs

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

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

Let n_t be the number of shares of the stock owned at time t, and let p_t be their price.

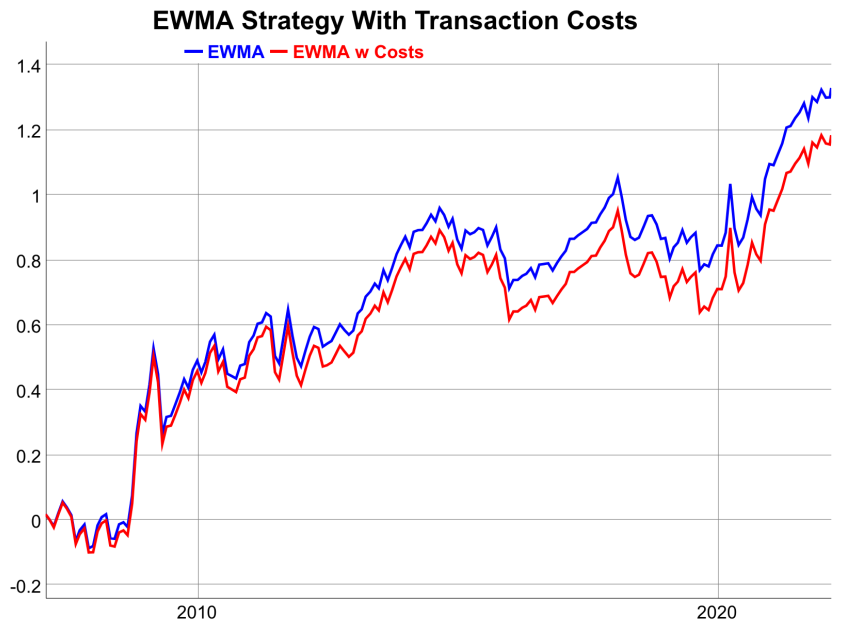
Then the traded dollar amount of the stock is equal to the change in the number of shares times the stock price: $\Delta n_t p_t$.

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

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

If d_t is the dollar amount of the stock owned at time t then the *transaction costs* c^r are equal to:

$$c^r = \frac{\delta}{2} |\Delta d_t|$$




```
> # bid_offer equal to 10 bps for liquid ETFs
> bid_offer <- 0.001
> # Calculate transaction costs
> costs <- 0.5*bid_offer*abs(poslag - posit)
> # Plot strategy with transaction costs
> wealthv <- cbind(pnls, pnls - costs)
> colnames(wealthv) <- c("EWMA", "EWMA w Costs")
> colorv <- c("blue", "red")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="EWMA Strategy With
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Simulation Function for EWMA Crossover Strategy

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

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

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

```
> sim_ewma <- function(ohlc, lambda=0.9, look_back=333, bid_offer=0.
+                trend=1, lagg=1) {
+   closep <- quantmod::Cl(ohlc)
+   retsp <- rutils::diffit(closep)
+   nrows <- NROW(ohlc)
+   # Calculate EWMA prices
+   ewmacpp <- HighFreq::run_mean(closep, lambda=lambda, weights=0)
+   # Calculate the indicator
+   indic <- trend*sign(closep - ewmacpp)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   # Calculate positions, either: -1, 0, or 1
+   posit <- rep(NA_integer_, nrows)
+   posit[1] <- 0
+   posit <- ifelse(indic == lagg, 1, posit)
+   posit <- ifelse(indic == (-lagg), -1, posit)
+   posit <- zoo::na.locf(posit, na.rm=FALSE)
+   posit <- xts::xts(posit, order.by=zoo::index(closep))
+   # Lag the positions to trade on next day
+   posit <- rutils::lagit(posit, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- retsp*posit
+   costs <- 0.5*bid_offer*abs(rutils::diffit(posit))
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(posit, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end sim_ewma
```

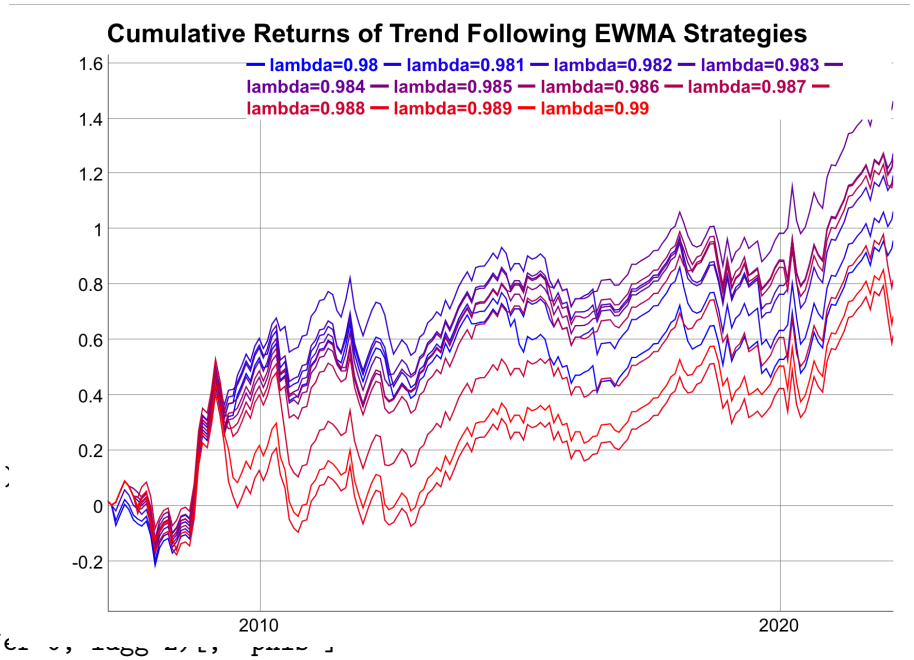
Simulating Multiple Trend Following EWMA Strategies

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

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

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

```
> source("/Users/jerzy/Develop/lecture_slides/scripts/ewma_model.R")
> lambdas <- seq(from=0.98, to=0.99, by=0.001)
> # Perform lapply() loop over lambdas
> pnltrend <- lapply(lambdas, function(lambda) {
+   # Simulate EWMA strategy and calculate returns
+   sim_ewma(ohlc=ohlc, lambda=lambda, look_back=look_back, bid_offe
+ })  # end lapply
> pnltrend <- do.call(cbind, pnltrend)
> colnames(pnltrend) <- paste0("lambda=", lambdas)
```




```
> # Plot dygraph of multiple EWMA strategies
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(pnltrend))
> dygraphs::dygraph(cumsum(pnltrend)[endd], main="Cumulative Returns
+   dyOptions(colors=colorv, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(pnltrend), theme=plot_theme,
+   name="Cumulative Returns of EWMA Strategies")
> legend("topleft", legend=colnames(pnltrend), inset=0.1,
+   bg="white", cex=0.8, lwd=rep(6, NCOL(pnltrend)),
+   col=plot_theme$col$line.col, bty="n")
```

Simulating EWMA Strategies Using Parallel Computing

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

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

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

```
> # Initialize compute cluster under Windows
> library(parallel)
> cluster <- makeCluster(detectCores()-1)
> clusterExport(cluster,
+   varlist=c("ohlc", "look_back", "sim_ewma"))
> # Perform parallel loop over lambdas under Windows
> pnltrend <- parLapply(cluster, lambdas, function(lambda) {
+   library(quantmod)
+   # Simulate EWMA strategy and calculate returns
+   sim_ewma(ohlc=ohlc, lambda=lambda, look_back=look_back)[, "pnls"
+ })  # end parLapply
> stopCluster(cluster)  # Stop R processes over cluster under Windo
> # Perform parallel loop over lambdas under Mac-OSX or Linux
> pnltrend <- mclapply(lambdas, function(lambda) {
+   library(quantmod)
+   # Simulate EWMA strategy and calculate returns
+   sim_ewma(ohlc=ohlc, lambda=lambda, look_back=look_back)[, "pnls"
+ })  # end mclapply
> pnltrend <- do.call(cbind, pnltrend)
> colnames(pnltrend) <- paste0("lambda=", lambdas)
```

Optimal Decay Parameter of Trend Following EWMA Strategies

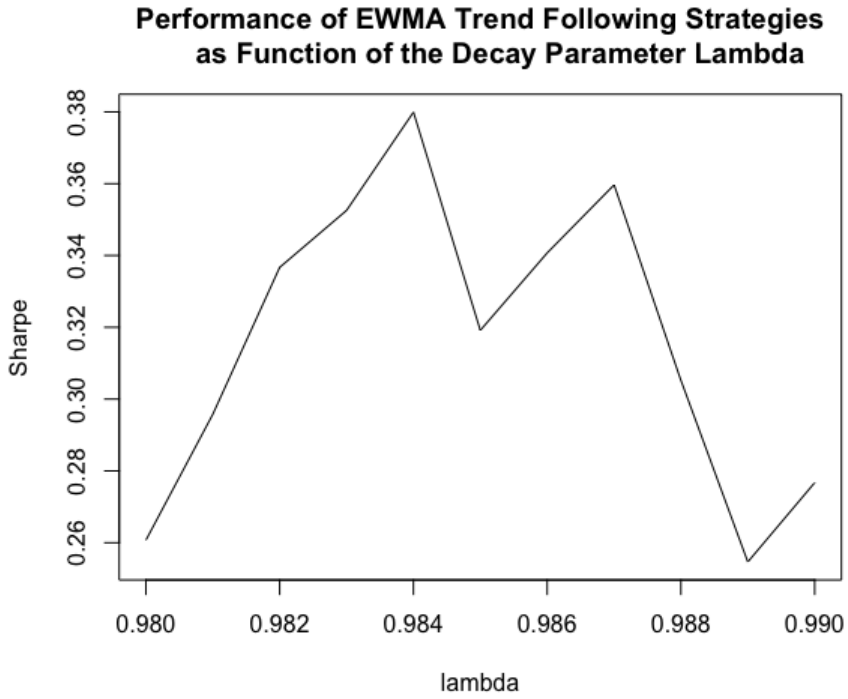
The performance of trend following *EWMA* strategies depends on the λ decay parameter, with smaller λ parameters performing better than larger ones.

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

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

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

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

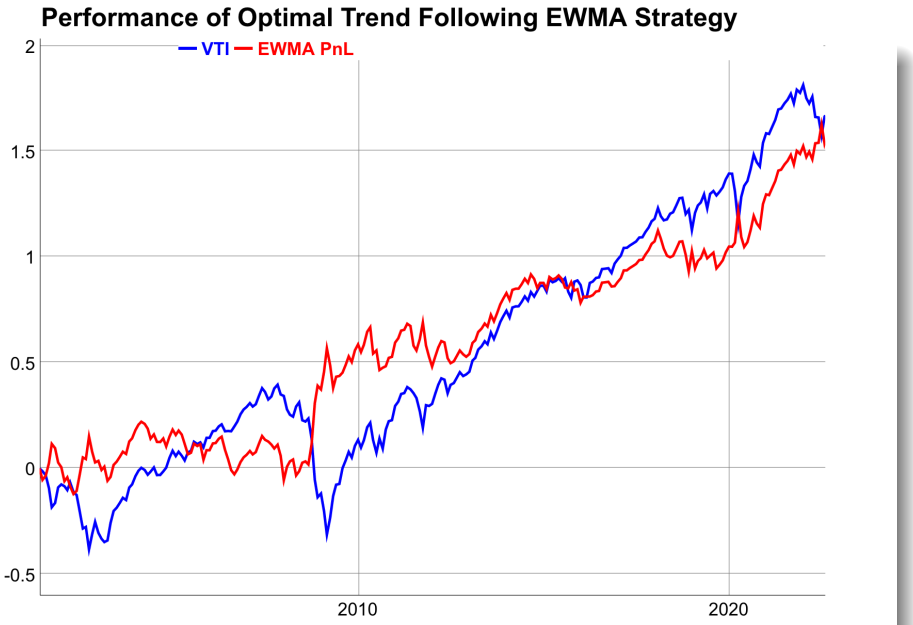



```
> # Calculate annualized Sharpe ratios of strategy returns
> sharpetrend <- sqrt(252)*sapply(pnltrend, function(xtsv) {
+   mean(xtsv)/sd(xtsv)
+ })  # end sapply
> # Plot Sharpe ratios
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> plot(x=lambdas, y=sharpetrend, t="l",
+      xlab="lambda", ylab="Sharpe",
+      main="Performance of EWMA Trend Following Strategies
+      as Function of the Decay Parameter Lambda")
```

Optimal Trend Following EWMA Strategy

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

```
> # Calculate optimal lambda
> lambda <- lambdas[which.max(sharpetrend)]
> # Simulate best performing strategy
> ewmatrend <- sim_ewma(ohlc=ohlc, lambda=lambda, bid_offer=0, lagg=
> posit <- ewmatrend[, "positions"]
> trendopt <- ewmatrend[, "pnls"]
> wealthv <- cbind(retsp, trendopt)
> colnames(wealthv) <- c("VTI", "EWMA PnL")
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> cor(wealthv)[1, 2]
> # Plot dygraph of EWMA strategy wealth
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Performance of Opt
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```



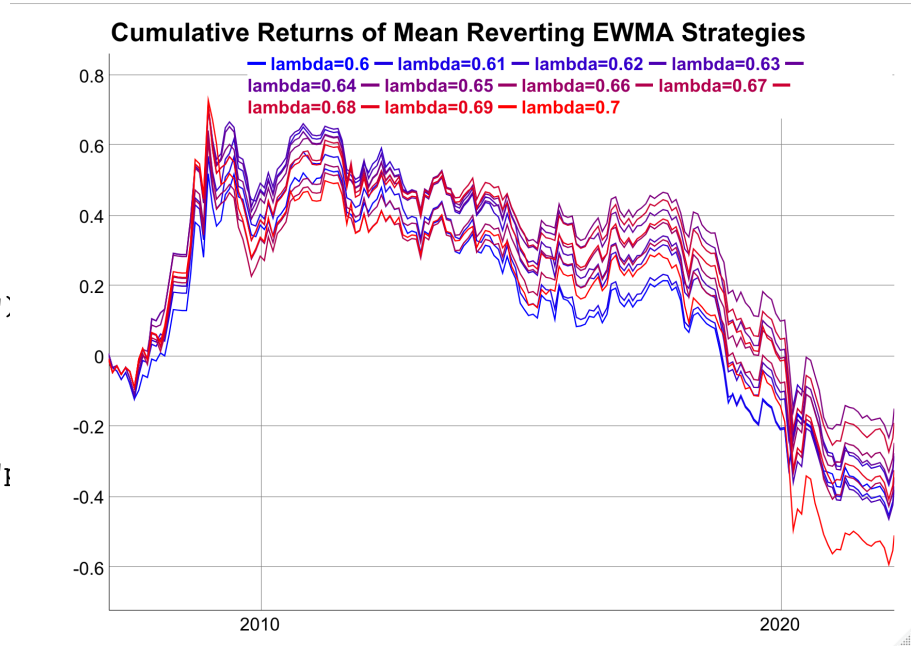

```
> # Plot EWMA PnL with position shading
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(wealthv), theme=plot_theme,
+        name="Performance of EWMA Strategy")
> add_TA(posit > 0,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

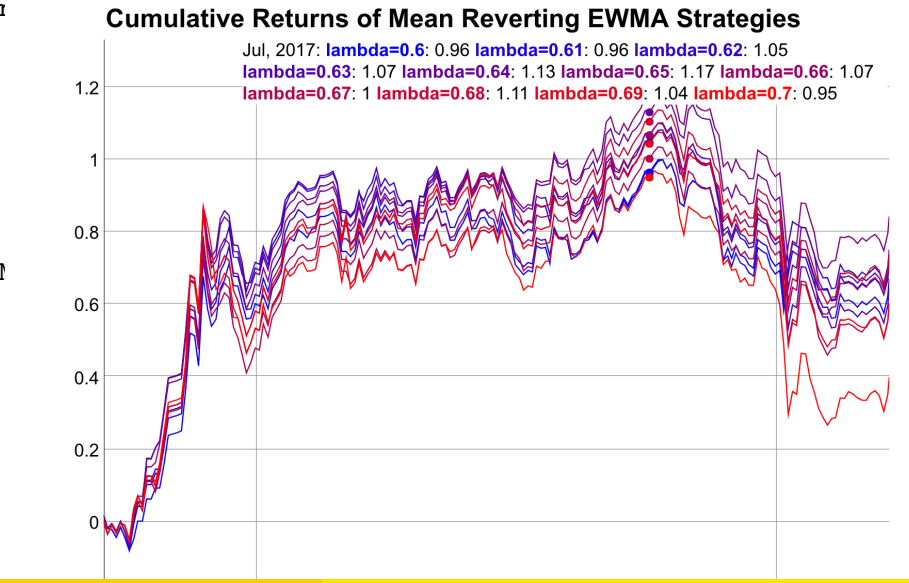
Mean Reverting EWMA Crossover Strategies

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

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

```
> source("/Users/jerzy/Develop/lecture_slides/scripts/ewma_model.R")
> lambdas <- seq(0.6, 0.7, 0.01)
> # Perform lapply() loop over lambdas
> pnlrevert <- lapply(lambdas, function(lambda) {
+   # Simulate EWMA strategy and calculate returns
+   sim_ewma(ohlc=ohlc, lambda=lambda, bid_offer=0, trend=(-1))[, "I
+ })  # end lapply
> pnlrevert <- do.call(cbind, pnlrevert)
> colnames(pnlrevert) <- paste0("lambda=", lambdas)
> # Plot dygraph of mean reverting EWMA strategies
> colorv <- colorRampPalette(c("blue", "red"))(NROW(lambdas))
> dygraphs::dygraph(cumsum(pnlrevert)[endd], main="Cumulative Returr
+   dyOptions(colors=colorv, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot EWMA strategies with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(pnlrevert,
+   theme=plot_theme, name="Cumulative Returns of Mean Reverting EWI
> legend("topleft", legend=colnames(pnlrevert),
+   inset=0.1, bg="white", cex=0.8, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```





Performance of Mean Reverting EWMA Strategies

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

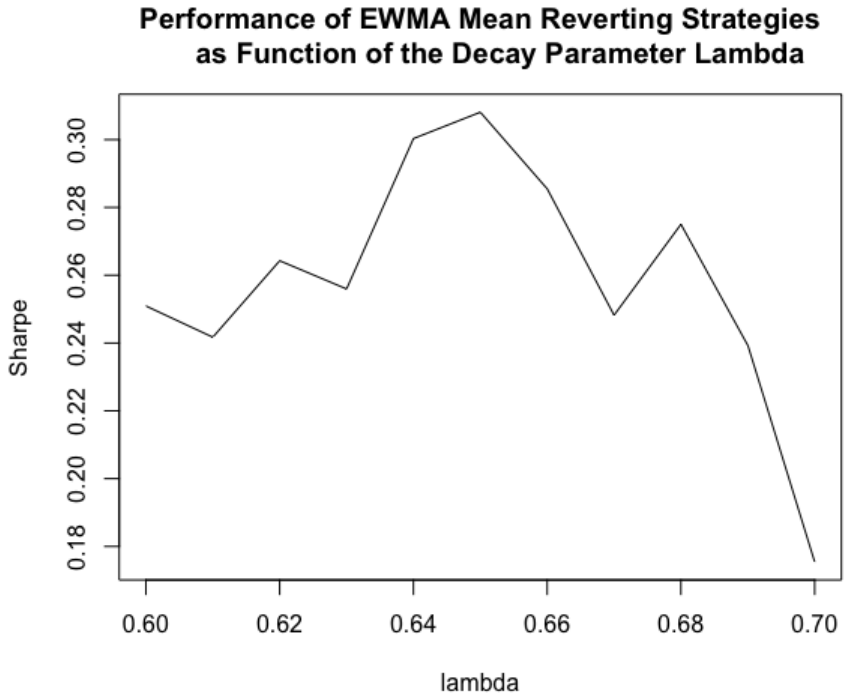
`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns.

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

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

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

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



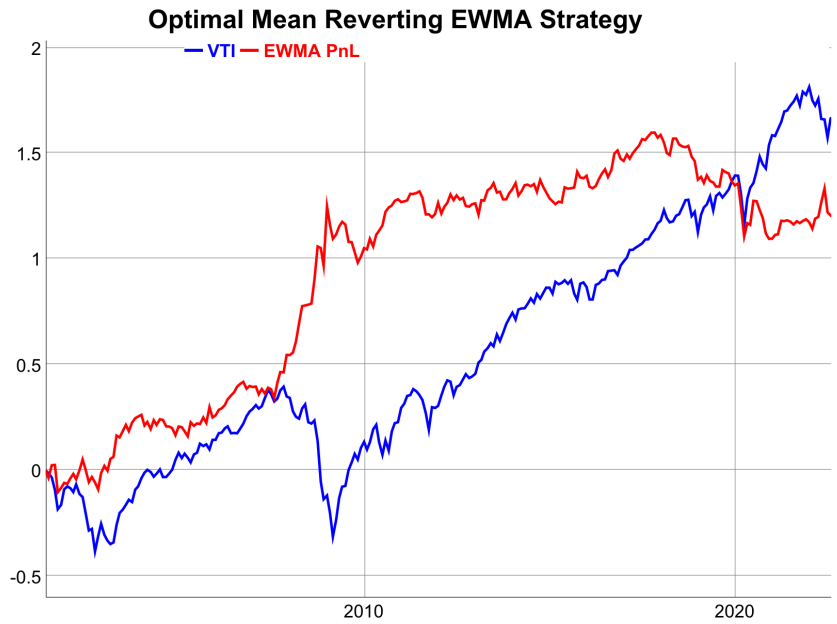

```
> # Calculate Sharpe ratios of strategy returns
> sharperevert <- sqrt(252)*sapply(pnlrevert, function(xtsv) {
+   mean(xtsv)/sd(xtsv)
+ })  # end sapply
> plot(x=lambdas, y=sharperevert, t="l",
+      xlab="lambda", ylab="Sharpe",
+      main="Performance of EWMA Mean Reverting Strategies
+    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

Optimal Mean Reverting EWMA Strategy

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

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

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



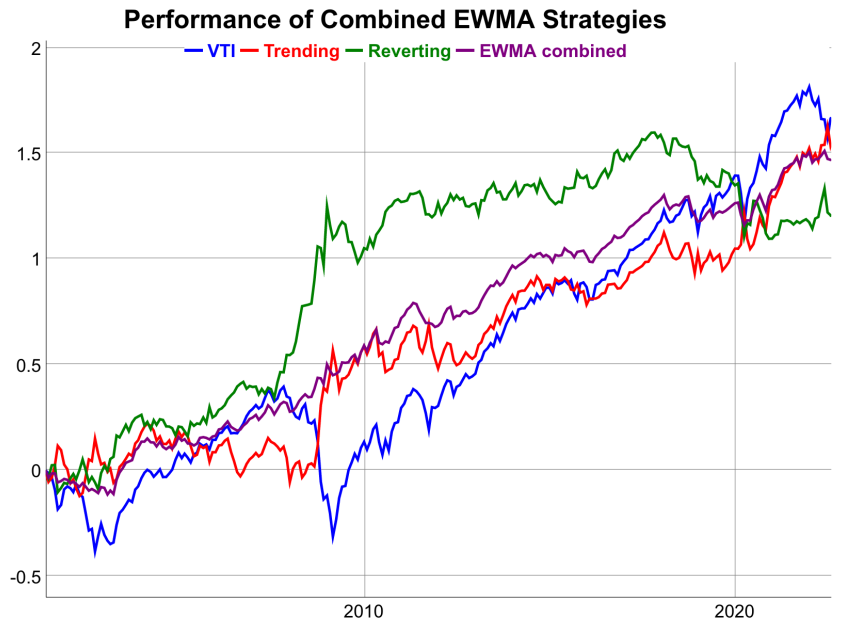

```
> # Calculate optimal lambda
> lambda <- lambdas[which.max(sharperevert)]
> # Simulate best performing strategy
> ewmarevert <- sim_ewma(ohlc=ohlc, bid_offer=0.0,
+   lambda=lambda, trend=(-1))
> posit <- ewmarevert[, "positions"]
> revertopt <- ewmarevert[, "pnls"]
> wealthv <- cbind(retsp, revertopt)
> colnames(wealthv) <- c("VTI", "EWMA PnL")
> # Plot dygraph of EWMA strategy wealth
> colorv <- c("blue", "red")
> dygraphs::dygraph(cumsum(wealthv)[endd], main="Optimal Mean Reve
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

```
> # Standard plot of EWMA strategy wealth
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(wealthv), theme=plot_theme,
+        name="Optimal Mean Reverting EWMA Strategy")
> add_TA(posit > 0, on=-1, col="lightgreen", border="lightgreen")
> add_TA(posit < 0, on=-1, col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(wealthv),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Combining Trend Following and Mean Reverting Strategies

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

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




```
> # Calculate correlation between trend following and mean revertin
> trendopt <- ewmatrend[, "pnls"]
> colnames(trendopt) <- "trend"
> revertopt <- ewmarevert[, "pnls"]
> colnames(revertopt) <- "revert"
> cor(cbind(retsp, trendopt, revertopt))
> # Calculate combined strategy
> combstrat <- (retsp + trendopt + revertopt)/3
> colnames(combstrat) <- "combined"
> # Calculate annualized Sharpe ratio of strategy returns
> retsp <- cbind(retsp, trendopt, revertopt, combstrat)
> colnames(retsp) <- c("VTI", "Trending", "Reverting", "EWMA combi
> sqrt(252)*sapply(retsp, function(xtsv) mean(xtsv)/sd(xtsv))
```

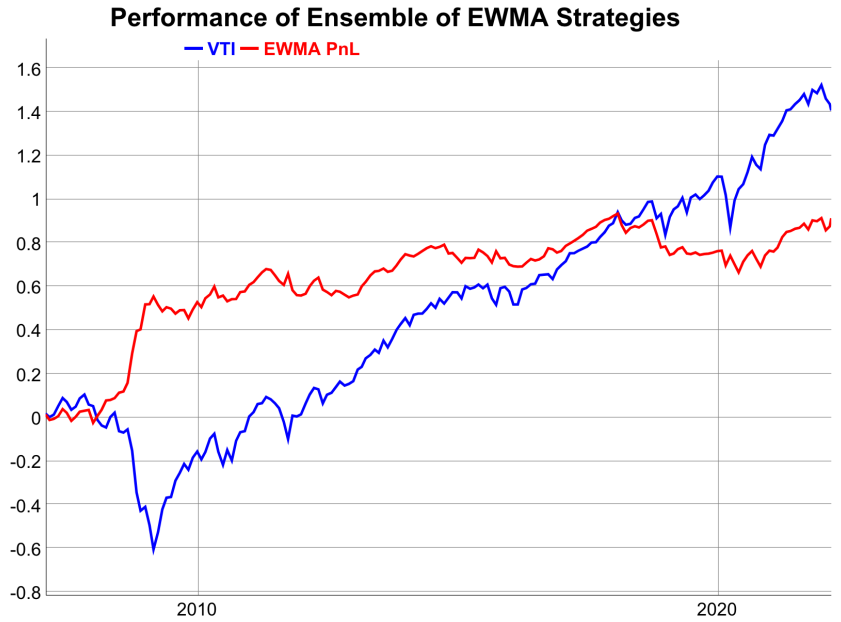
```
> # Plot dygraph of EWMA strategy wealth
> colorv <- c("blue", "red", "green", "purple")
> dygraphs::dygraph(cumsum(retsp)[endd], main="Performance of Combi
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Standard plot of EWMA strategy wealth
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(pnls, theme=plot_theme,
+        name="Performance of Combined EWMA Strategies")
> legend("topleft", legend=colnames(pnls),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Ensemble of EWMA Strategies

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

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

```
> # Calculate weights proportional to Sharpe ratios
> weightv <- c(sharpetrend, sharperevert)
> weightv[weightv<0] <- 0
> weightv <- weightv/sum(weightv)
> retsp <- cbind(pnltrend, pnlrevert)
> retsp <- retsp %*% weightv
> retsp <- xts::xts(retsp, order.by=zoo::index(retsp))
> retsp <- cbind(retsp, retsp)
> colnames(retsp) <- c("VTI", "EWMA PnL")
> # Plot dygraph of EWMA strategy wealth
> colorv <- c("blue", "red")
> dygraphs::dygraph(cumsum(retsp)[endd], main="Performance of Ensemble of EWMA Strategies") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Standard plot of EWMA strategy wealth
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(retsp), theme=plot_theme,
+        name="Performance of Ensemble of EWMA Strategies")
> legend("topleft", legend=colnames(pnls),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

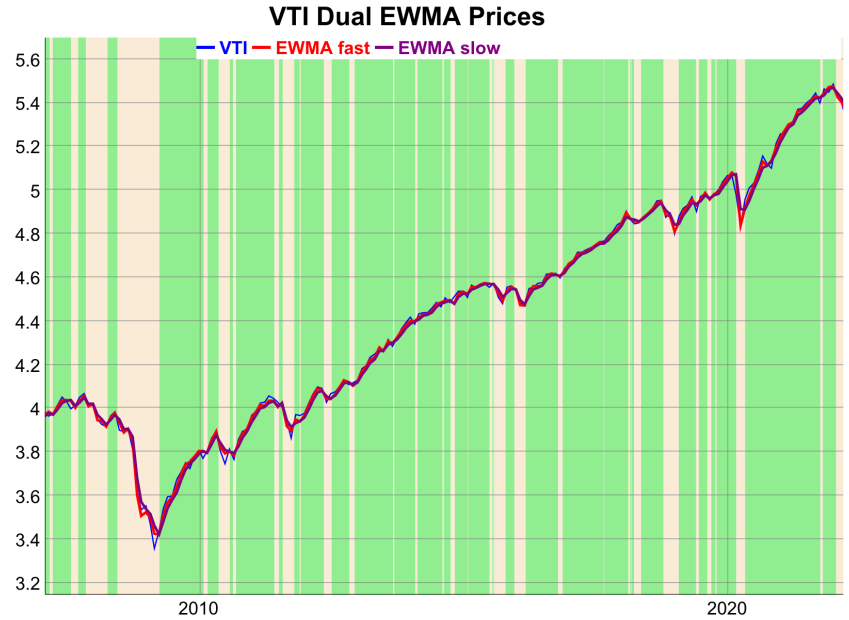


Simulating the Dual EWMA Crossover Strategy

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

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

```
> # Calculate fast and slow EWMAs
> look_back <- 333
> lambda1 <- 0.89
> lambda2 <- 0.95
> # Calculate EWMA prices
> ewma1 <- HighFreq::run_mean(closep, lambda=lambda1, weights=0)
> ewma2 <- HighFreq::run_mean(closep, lambda=lambda2, weights=0)
> # Calculate EWMA prices
> pricets <- cbind(closep, ewma1, ewma2)
> colnames(pricets) <- c("VTI", "EWMA fast", "EWMA slow")
> # Calculate positions, either: -1, 0, or 1
> indic <- sign(ewma1 - ewma2)
> lagg <- 2
> indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
> posit <- rep(NA_integer_, nrows)
> posit[1] <- 0
> posit <- ifelse(indic == lagg, 1, posit)
> posit <- ifelse(indic == (-lagg), -1, posit)
> posit <- zoo::na.locf(posit, na.rm=FALSE)
> posit <- xts::xts(posit, order.by=zoo::index(closep))
> posit <- rutils::lagit(posit, lagg=1)
```



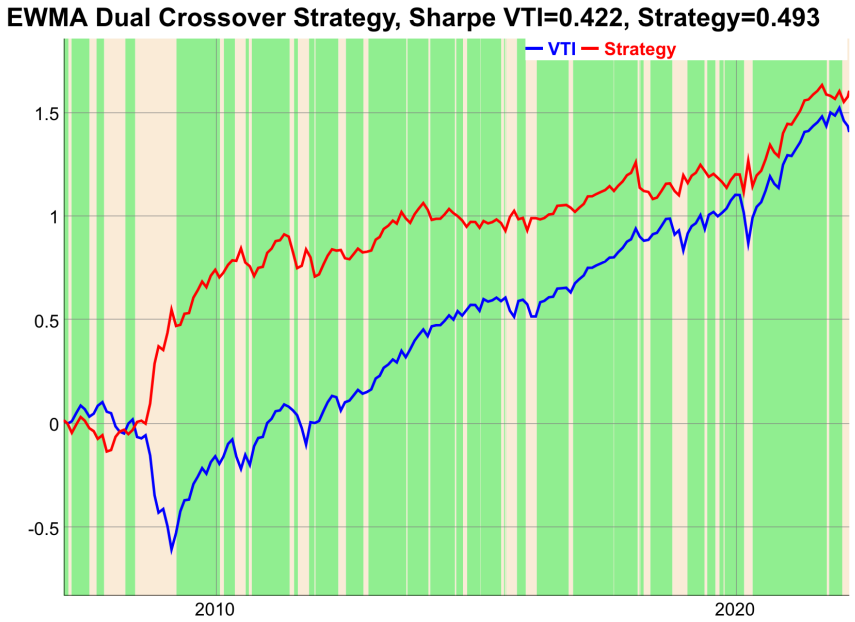

```
> # Create colors for background shading
> crossd <- (rutils::diffit(posit) != 0)
> shadev <- posit[crossd]
> crossd <- c(zoo::index(shadev), end(posit))
> shadev <- ifelse(drop(zoo::coredata(shadev)) == 1, "lightgreen", "
> # Plot dygraph
> colnamev <- colnames(pricets)
> dyplot <- dygraphs::dygraph(pricets[endd], main="VTI Dual EWMA Pri
+   dySeries(name=colnamev[1], label=colnamev[1], strokeWidth=1, col
+   dySeries(name=colnamev[2], label=colnamev[2], strokeWidth=2, col
+   dySeries(name=colnamev[3], label=colnamev[3], strokeWidth=2, col
+   dyLegend(show="always", width=500)
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>% dyShading(from=crossd[i], to=crossd[i+1],
+ }  # end for
> dyplot
```

Performance of the Dual EWMA Crossover Strategy

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

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

```
> # Calculate daily profits and losses of strategy
> pnls <- retsp*posit
> colnames(pnls) <- "Strategy"
> wealthv <- cbind(retsp, pnls)
> # Annualized Sharpe ratio of Dual EWMA strategy
> sharper <- sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(wealthv)[1, 2]
```




```
> # Plot Dual EWMA strategy
> dyplot <- dygraphs::dygraph(cumsum(wealthv), main=paste("EWMA Dual
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
> # Add shading to dygraph object
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>% dyShading(from=crossd[i], to=crossd[i+1],
+ }  # end for
> # Plot the dygraph object
> dyplot
```

Simulation Function for the Dual EWMA Crossover Strategy

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

The function `sim_ewma2()` performs a simulation of the *Dual EWMA* strategy, given an *OHLC* time series of prices, and two decay parameters λ_1 and λ_2.

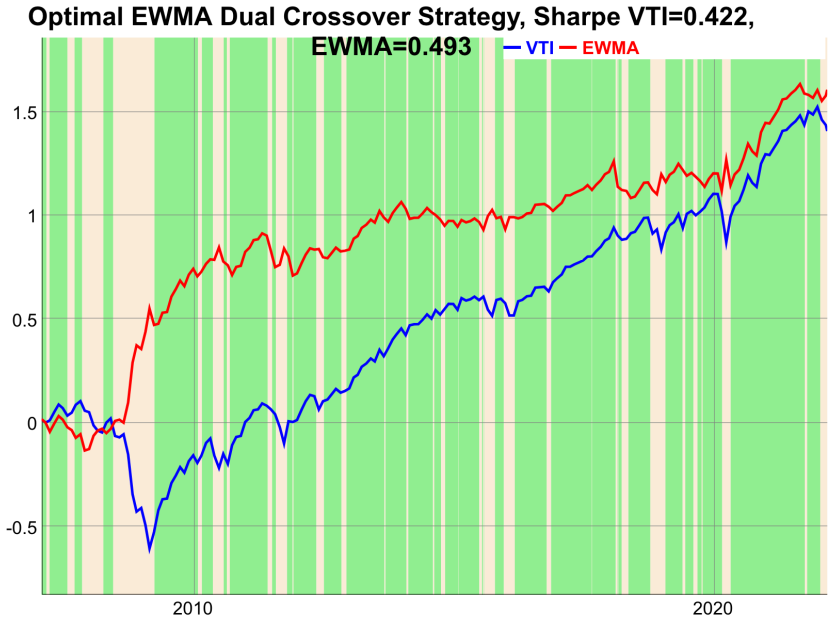
The function `sim_ewma2()` returns the *EWMA* strategy positions and returns, in a two-column *xts* time series.

```
> sim_ewma2 <- function(ohlc, lambda1=0.1, lambda2=0.01, look_back=3
+                bid_offer=0.001, trend=1, lagg=1) {
+   closep <- quantmod::Cl(ohlc)
+   retsp <- rutils::diffit(closep)
+   nrows <- NROW(ohlc)
+   # Calculate EWMA prices
+   ewma1 <- HighFreq::run_mean(closep, lambda=lambda1, weights=0)
+   ewma2 <- HighFreq::run_mean(closep, lambda=lambda2, weights=0)
+   # Calculate positions, either: -1, 0, or 1
+   indic <- sign(ewma1 - ewma2)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   posit <- rep(NA_integer_, nrows)
+   posit[1] <- 0
+   posit <- ifelse(indic == lagg, 1, posit)
+   posit <- ifelse(indic == (-lagg), -1, posit)
+   posit <- zoo::na.locf(posit, na.rm=FALSE)
+   posit <- xts::xts(posit, order.by=zoo::index(closep))
+   # Lag the positions to trade on next day
+   posit <- rutils::lagit(posit, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- retsp*posit
+   costs <- 0.5*bid_offer*abs(rutils::diffit(posit))
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(posit, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end sim_ewma2
```

Optimal Dual EWMA Strategy

Multiple *Dual EWMA* strategies can be simulated by calling the function `sim_ewma2()` in two loops over the vectors of λ parameters.

The best *Dual EWMA* strategy performs better than the best *single EWMA* strategy, because it has an extra parameter that can be adjusted to improve in-sample performance. But this doesn't guarantee better out-of-sample performance.




```
> source("/Users/jerzy/Develop/lecture_slides/scripts/ewma_model.R")
> lambdas1 <- seq(from=0.85, to=0.99, by=0.01)
> lambdas2 <- seq(from=0.85, to=0.99, by=0.01)
> # Perform sapply() loops over lambdas
> sharpem <- sapply(lambdas1, function(lambda1) {
+   sapply(lambdas2, function(lambda2) {
+     if (lambda2 > lambda1) {
+ # Simulate Dual EWMA strategy
+ pnls <- sim_ewma2(ohlc=ohlc, lambda1=lambda1, lambda2=lambda2,
+                   look_back=look_back, bid_offer=0.0, trend=1,
+ sqrt(252)*mean(pnls)/sd(pnls)
+     } else NA
+   })  # end sapply
+ })  # end sapply
> colnames(sharpem) <- lambdas1
> rownames(sharpem) <- lambdas2
> # Calculate the PnLs for the optimal strategy
> whichv <- which(sharpem == max(sharpem, na.rm=TRUE), arr.ind=TRUE
> lambda1 <- lambdas1[whichv[2]]
> lambda2 <- lambdas2[whichv[1]]
> pnls <- sim_ewma2(ohlc=ohlc, lambda1=lambda1, lambda2=lambda2,
+              look_back=look_back, bid_offer=0.0, trend=1, lagg=2)[, "pnls"]
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv)[2] <- "EWMA"
```

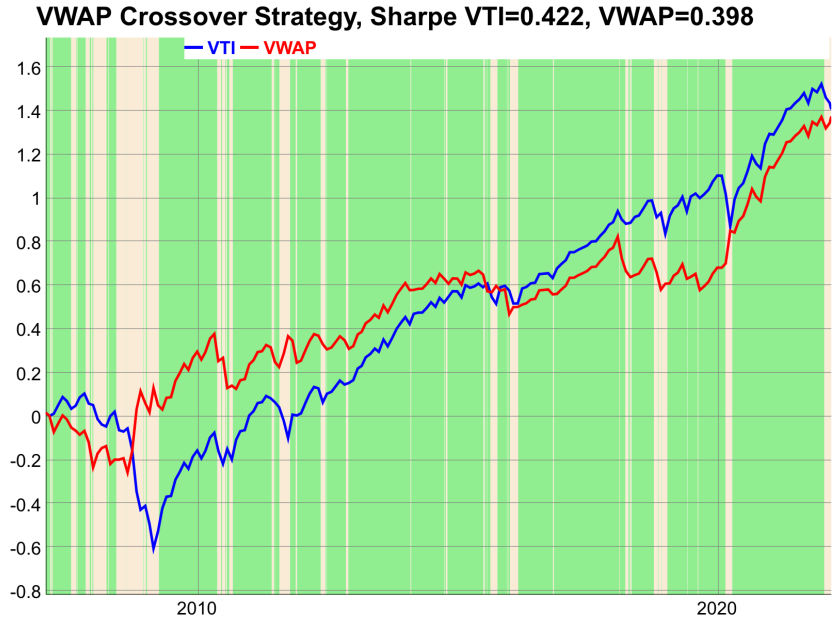
```
> # Annualized Sharpe ratio of Dual EWMA strategy
> sharper <- sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
> # The crossover strategy has a negative correlation to VTI
> cor(wealthv)[1, 2]
> # Plot Optimal Dual EWMA strategy
> dyplot <- dygraphs::dygraph(cumsum(wealthv), main=paste("Optimal
+   dyOptions(colors=c("blue", "red"), strokeWidth=2)
> # Add shading to dygraph object
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>% dyShading(from=crossd[i], to=crossd[i+1],
+ }  # end for
> # Plot the dygraph object
> dyplot
```

Simulating the VWAP Crossover Strategy

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

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

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




```
> # Calculate positions from lagged indicator
> indic <- sign(closep - vwapcpp)
> lagg <- 2
> indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
> # Calculate positions, either: -1, 0, or 1
> posit <- rep(NA_integer_, nrows)
> posit[1] <- 0
> posit <- ifelse(indic == lagg, 1, posit)
> posit <- ifelse(indic == (-lagg), -1, posit)
> posit <- zoo::na.locf(posit, na.rm=FALSE)
> posit <- xts::xts(posit, order.by=zoo::index(closep))
> # Lag the positions to trade in next period
> posit <- rutils::lagit(posit, lagg=1)
> # Calculate PnLs of VWAP strategy
> retsp <- rutils::diffit(closep)  # VTI returns
> pnls <- retsp*posit
> colnames(pnls) <- "VWAP"
> wealthv <- cbind(retsp, pnls)
> colnamev <- colnames(wealthv)
> # Annualized Sharpe ratios of VTI and VWAP strategy
> sharper <- sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
```

```
> # Create colors for background shading
> crossd <- (rutils::diffit(posit) != 0)
> shadev <- posit[crossd]
> crossd <- c(zoo::index(shadev), end(posit))
> shadev <- ifelse(drop(zoo::coredata(shadev)) == 1, "lightgreen", "
> # Plot dygraph of VWAP strategy
> # Create dygraph object without plotting it
> dyplot <- dygraphs::dygraph(cumsum(wealthv), main=paste("VWAP Cros
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Add shading to dygraph object
> for (i in 1:NROW(shadev)) {
+     dyplot <- dyplot %>% dyShading(from=crossd[i], to=crossd[i+1],
+ }  # end for
> # Plot the dygraph object
> dyplot
```

Combining VWAP Crossover Strategy with Stocks

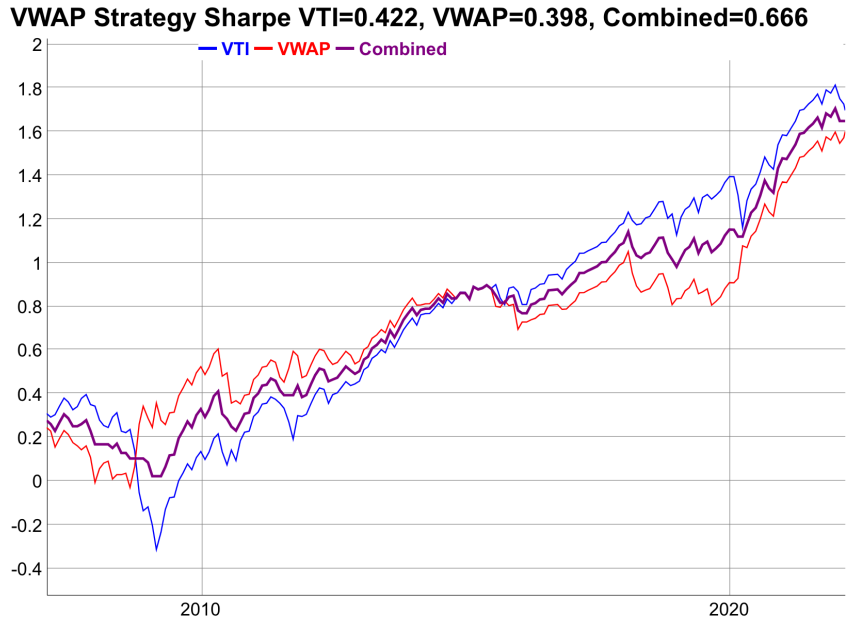
Even though the *VWAP* strategy doesn't perform as well as a static buy-and-hold strategy, it can provide risk reduction when combined with it.

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

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

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

```
> # Calculate correlation of VWAP strategy with VTI
> cor(retsp, pnls)
> # Combine VWAP strategy with VTI
> wealthv <- cbind(retsp, pnls, 0.5*(retsp+pnls))
> colnames(wealthv) <- c("VTI", "VWAP", "Combined")
> sharper <- sqrt(252)*sapply(wealthv, function (x) mean(x)/sd(x))
```

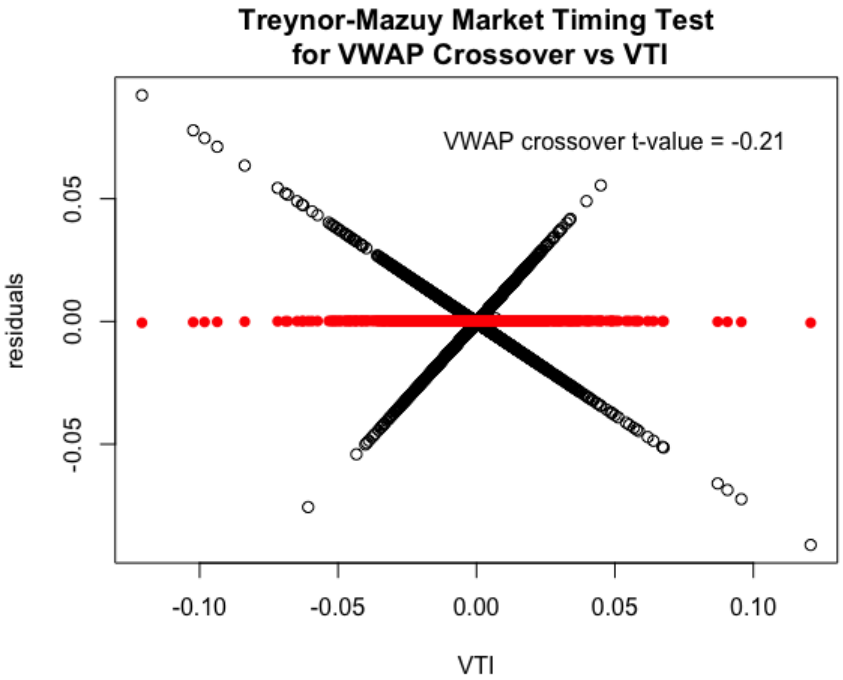



```
> # Plot dygraph of VWAP strategy combined with VTI
> colnamev <- colnames(wealthv)
> dygraphs::dygraph(cumsum(wealthv)[endd], paste("VWAP Strategy Shar
+   dySeries(name=colnamev[1], label=colnamev[1], col="blue", stroke
+   dySeries(name=colnamev[2], label=colnamev[2], col="red", strokeW
+   dySeries(name=colnamev[3], label=colnamev[3], col="purple", stro
+   dyLegend(show="always", width=500)
> # Or
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main=paste("VWAP Strategy Sharpe", paste(paste(names(sharper),
+   dyOptions(colors=c("blue", "red", "purple"), strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
```

VWAP Crossover Strategy Market Timing Skill

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

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




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

Simulation Function for VWAP Crossover Strategy

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

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

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

```
> sim_vwap <- function(ohlc, lambda=0.9, bid_offer=0.001, trend=1, l
+   closep <- log(quantmod::Cl(ohlc))
+   volumes <- quantmod::Vo(ohlc)
+   retsp <- rutils::diffit(closep)
+   nrows <- NROW(ohlc)
+   # Calculate VWAP prices
+   vwap <- HighFreq::run_mean(closep, lambda=lambda, weights=volume
+   # Calculate the indicator
+   indic <- trend*sign(closep - vwap)
+   if (lagg > 1) {
+     indic <- roll::roll_sum(indic, width=lagg, min_obs=1)
+     indic[1:lagg] <- 0
+   }  # end if
+   # Calculate positions, either: -1, 0, or 1
+   posit <- rep(NA_integer_, nrows)
+   posit[1] <- 0
+   posit <- ifelse(indic == lagg, 1, posit)
+   posit <- ifelse(indic == (-lagg), -1, posit)
+   posit <- zoo::na.locf(posit, na.rm=FALSE)
+   posit <- xts::xts(posit, order.by=zoo::index(closep))
+   # Lag the positions to trade on next day
+   posit <- rutils::lagit(posit, lagg=1)
+   # Calculate PnLs of strategy
+   pnls <- retsp*posit
+   costs <- 0.5*bid_offer*abs(rutils::diffit(posit))
+   pnls <- (pnls - costs)
+   # Calculate strategy returns
+   pnls <- cbind(posit, pnls)
+   colnames(pnls) <- c("positions", "pnls")
+   pnls
+ }  # end sim_vwap
```

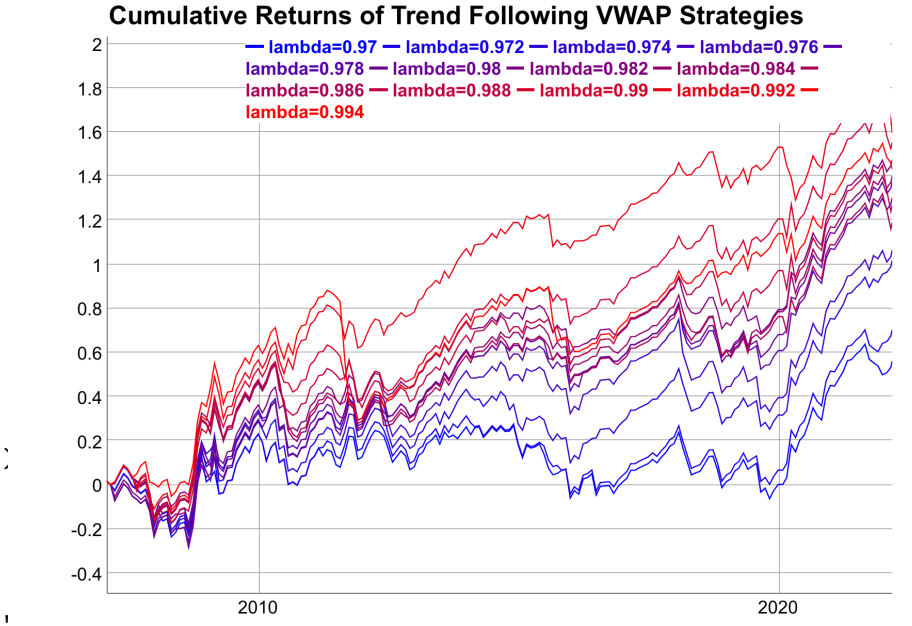
Simulating Multiple Trend Following VWAP Strategies

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

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

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

```
> source("/Users/jerzy/Develop/lecture_slides/scripts/ewma_model.R")
> lambdas <- seq(from=0.97, to=0.995, by=0.002)
> # Perform lapply() loop over lambdas
> pnls <- lapply(lambdas, function(lambda) {
+   # Simulate VWAP strategy and calculate returns
+   sim_vwap(ohlc=ohlc, lambda=lambda, bid_offer=0, lagg=2)[, "pnls"]
+ })  # end lapply
> pnls <- do.call(cbind, pnls)
> colnames(pnls) <- paste0("lambda=", lambdas)
```




```
> # Plot dygraph of multiple VWAP strategies
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(pnls))
> dygraphs::dygraph(cumsum(pnls)[endd], main="Cumulative Returns of
+   dyOptions(colors=colorv, strokeWidth=1) %>%
+   dyLegend(show="always", width=500)
> # Plot VWAP strategies with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorv
> quantmod::chart_Series(cumsum(pnls), theme=plot_theme,
+   name="Cumulative Returns of VWAP Strategies")
> legend("topleft", legend=colnames(pnls), inset=0.1,
+   bg="white", cex=0.8, lwd=rep(6, NCOL(pnls)),
+   col=plot_theme$col$line.col, bty="n")
```

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.

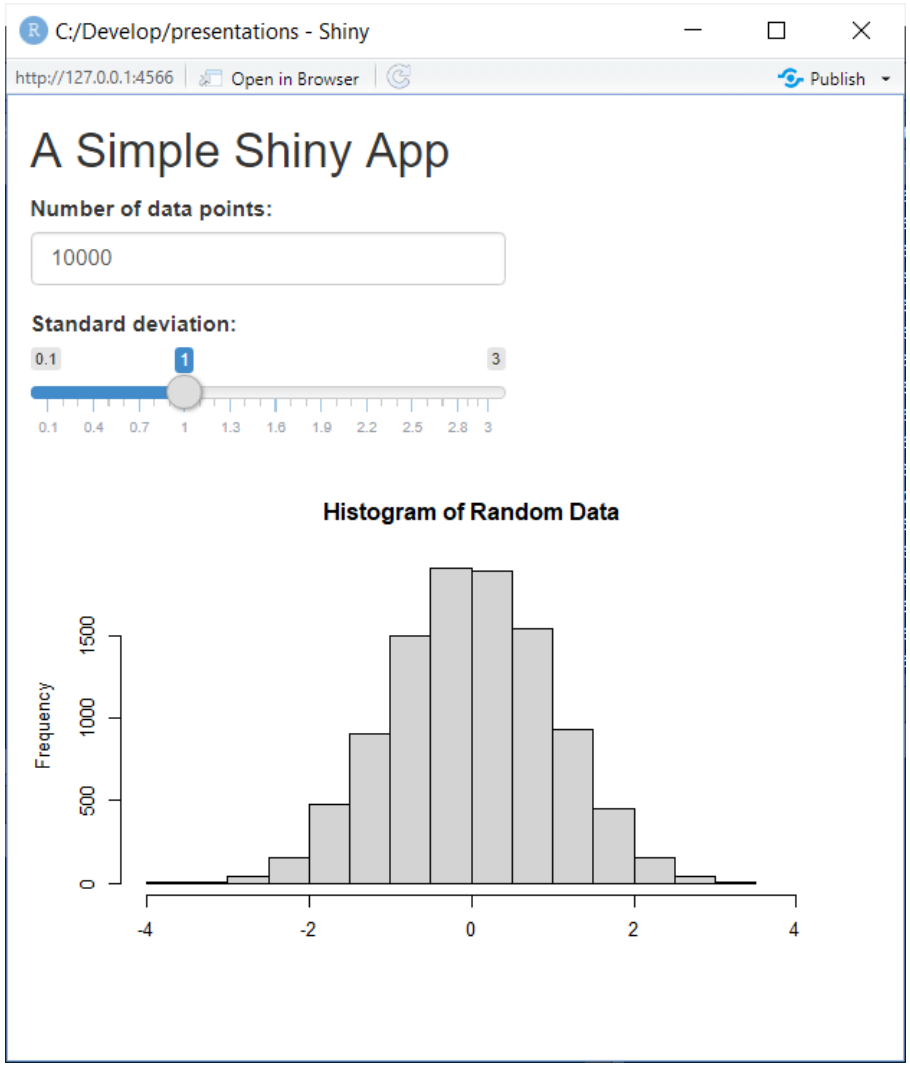
The function `shiny::shinyApp()` creates a `shiny` app from a *shiny interface* and a *shiny server*.

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

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
> uiface <- shiny::fluidPage(
+   titlePanel("VWAP Moving Average"),
+   # Create single row of widgets with two slider inputs
+   fluidRow(
+     # Input stock symbol
+     column(width=3, selectInput("symbol", label="Symbol",
+                           choices=symbolv, selected=symbol)),
+     # 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("dyplot"), 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
> servfun <- shiny::shinyServer(function(input, output) {
+   # Get the close and volume data in a reactive environment
+   closep <- shiny::reactive({
+     # Get the data
+     ohlc <- get(input$symbol, data_env)
+     closep <- log(quantmod::Cl(ohlc))
+     volumes <- quantmod::Vo(ohlc)
+     # Return the data
+     cbind(closep, volumes)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   vwapv <- shiny::reactive({
+     # Get model parameters from input argument
+     look_back <- input$look_back
+     # Calculate the VWAP indicator
+     closep <- closep()[, 1]
+     volumes <- closep()[, 2]
+     vwapv <- HighFreq::roll_sum(se_ries=closep*volumes, look_back=
+     volume_rolling <- HighFreq::roll_sum(se_ries=volumes, look_bac
+     vwapv <- vwapv/volume_rolling
+     vwapv[is.na(vwapv)] <- 0
+     # Return the plot data
+     datav <- cbind(closep, vwapv)
+     colnames(datav) <- c(input$symbol, "VWAP")
+     datav
+   })  # end reactive code
+
+   # Return the dygraph plot to output argument
+   output$dyplot <- dygraphs::renderDygraph({
+     colnamev <- colnames(vwapv())
+     dygraphs::dygraph(vwapv(), main=paste(colnamev[1], "VWAP")) %>
+ dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+ dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWidt
+ dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeWi
+   })  # 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
> servfun <- shiny::shinyServer(function(input, output) {
+
+   # Create an empty list of reactive values.
+   value_s <- reactiveValues()
+
+   # Get input parameters from the user interface.
+   nrows <- reactive({
+     # Add nrows to list of reactive values.
+     value_s*nrows <- input$nrows
+     input$nrows
+   })  # end reactive code
+
+   # Broadcast a message to the console when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     cat("Input button pressed\n")
+   })  # end observeEvent
+
+   # Send the data when the button is pressed.
+   datav <- eventReactive(eventExpr=input$button, valueExpr={
+     # eventReactive() executes on input$button, but not on nrows()
+     cat("Sending", nrows(), "rows of data\n")
+     datav <- head(mtcars, input$nrows)
+     value_s$mpg <- mean(datav$mpg)
+     datav
+   })  # end eventReactive
+   #   datav
+
+   # Draw table of the data when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     datav <- datav()
+     cat("Received", value_s*nrows, "rows of data\n")
+     cat("Average mpg = ", value_s$mpg, "\n")
+     cat("Drawing table\n")
+     output$tablev <- renderTable(datav)
+   })  # end observeEvent
+
+ })  # end server code
>
```

Homework Assignment

Required

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