FRE7241 Algorithmic Portfolio Management

Lecture#4, Spring 2026

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 10, 2026

NYU | TANDON SCHOOL OF ENGINEERING

Calculating the Trailing Variance of Asset Returns

The variance of asset returns exhibits *heteroskedasticity*, i.e. it changes over time.

The trailing variance of returns is given by:

$$\sigma_t^2 = \frac{1}{k-1}\sum_{j=0}^{k-1}(r_{t-j} - \bar{r}_t)^2$$

$$\bar{r}_t = \frac{1}{k}\sum_{j=0}^{k-1} r_{t-j}$$

Where `k` is the *look-back interval* equal to the number of data points for performing aggregations over the past.

It's also possible to calculate the trailing variance in `R` using vectorized functions, without using an `apply()` loop.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> # Define end points
> endd <- 1:NROW(retp)
> # Start points are multi-period lag of endd
> lookb <- 11
> startp <- c(rep_len(0, lookb-1), endd[1:(nrows-lookb+1)])
> # Calculate trailing variance in sapply() loop - takes long
> varv <- sapply(1:nrows, function(it) {
+   retp <- retp[startp[it]:endd[it]]
+   sum((retp - mean(retp))^2)/lookb
+ })  ##  end sapply
> # Use only vectorized functions
> retc <- cumsum(retp)
> retc <- (retc - c(rep_len(0, lookb), retc[1:(nrows-lookb)]))
> retc2 <- cumsum(retp^2)
> retc2 <- (retc2 - c(rep_len(0, lookb), retc2[1:(nrows-lookb)]))
> var2 <- (retc2 - retc^2/lookb)/lookb
> all.equal(varv[-(1:lookb)], as.numeric(var2)[-(1:lookb)])
> # Or using package rutils
> retc <- rutils::roll_sum(retp, lookb=lookb)
> retc2 <- rutils::roll_sum(retp^2, lookb=lookb)
> var2 <- (retc2 - retc^2/lookb)/lookb
> # Coerce variance into xts
> tail(varv)
> class(varv)
> varv <- xts(varv, order.by=zoo::index(retp))
> colnames(varv) <- "VTI.variance"
> head(varv)
```

Calculating the Trailing Variance Using Package *roll*

The package *roll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()` for the *weighted* trailing sum,
- `roll_var()` for the *weighted* trailing variance,
- `roll_scale()` for the trailing scaling and centering of time series,
- `roll_pcr()` for the trailing principal component regressions of time series.

The *roll* functions are about `1,000` times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled `C++` code, using packages *Rcpp*, *RcppArmadillo*, and *RcppParallel*.

The *roll* functions accept *xts* time series, and they return *xts*.

```
> # Calculate trailing VTI variance using package HighFreq
> varv <- roll::roll_var(retp, width=lookb)
> colnames(varv) <- "Variance"
> head(varv)
> sum(is.na(varv))
> varv[1:(lookb-1)] <- 0
> # Benchmark calculation of trailing variance
> library(microbenchmark)
> summary(microbenchmark(
+   sapply=sapply(1:nrows, function(it) {
+     var(retp[startp[it]:endd[it]])
+   }),
+   roll=roll::roll_var(retp, width=lookb),
+   times=10))[, c(1, 4, 5)]
```

Trailing *EMA* Realized Volatility Estimator

Time-varying volatility can be more accurately estimated using an *Exponential Moving Average* (*EMA*) variance estimator.

If the *time series* has zero *expected* mean, then the *EMA realized* variance estimator can be written approxiamtely as:

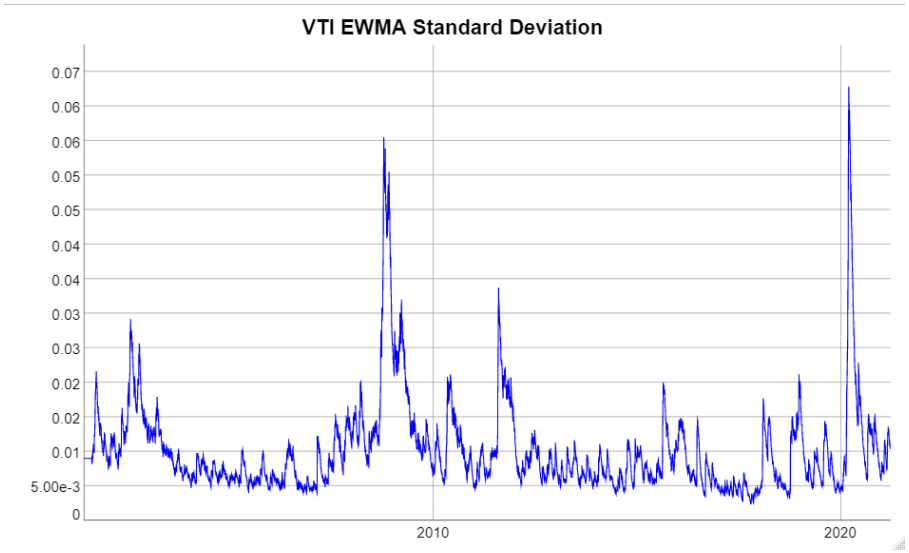
$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2 = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j r_{t-j}^2$$

σ_t^2 is the weighted *realized* variance, equal to the weighted average of the point realized variance for period `i` and the past *realized* variance.

The parameter λ determines the rate of decay of the *EMA* weights, with smaller values of λ producing faster decay, giving more weight to recent realized variance, and vice versa.

The function `stats:::C_cfilter()` calculates the convolution of a vector or a time series with a filter of coefficients (weights).

The function `stats:::C_cfilter()` is very fast because it's compiled `C++` code.




```
> # Calculate EMA VTI variance using compiled C++ function
> lookb <- 51
> weightv <- exp(-0.1*1:lookb)
> weightv <- weightv/sum(weightv)
> varv <- .Call(stats:::C_cfilter, retp^2, filter=weightv, sides=1,
> varv[1:(lookb-1)] <- varv[lookb]
> # Plot EMA volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(retp))
> dygraphs::dygraph(varv, main="VTI EMA Volatility") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
> quantmod::chart_Series(xtsv, name="VTI EMA Volatility")
```

Estimating Trailing Variance Using Package *roll*

If the *time series* has non-zero *expected* mean, then the trailing *EMA* variance is a vector given by the estimator:

$$\sigma_t^2 = \frac{1}{k-1}\sum_{j=0}^{k-1} w_j (r_{t-j} - \bar{r}_t)^2$$

$$\bar{r}_t = \frac{1}{k}\sum_{j=0}^{k-1} w_j r_{t-j}$$

Where w_j is the vector of exponentially decaying weights:

$$w_j = \frac{\lambda^j}{\sum_{j=0}^{k-1} \lambda^j}$$

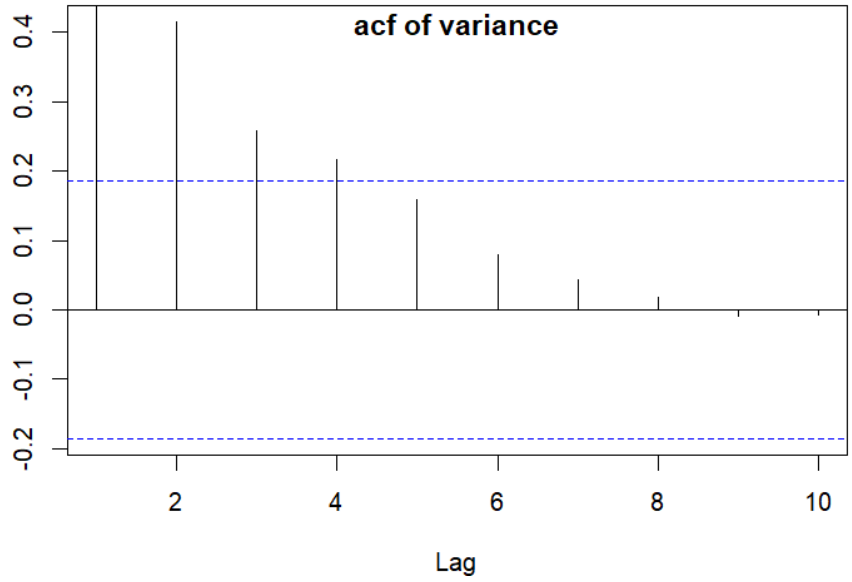
The function `roll_var()` from package *roll* calculates the trailing *EMA* variance.

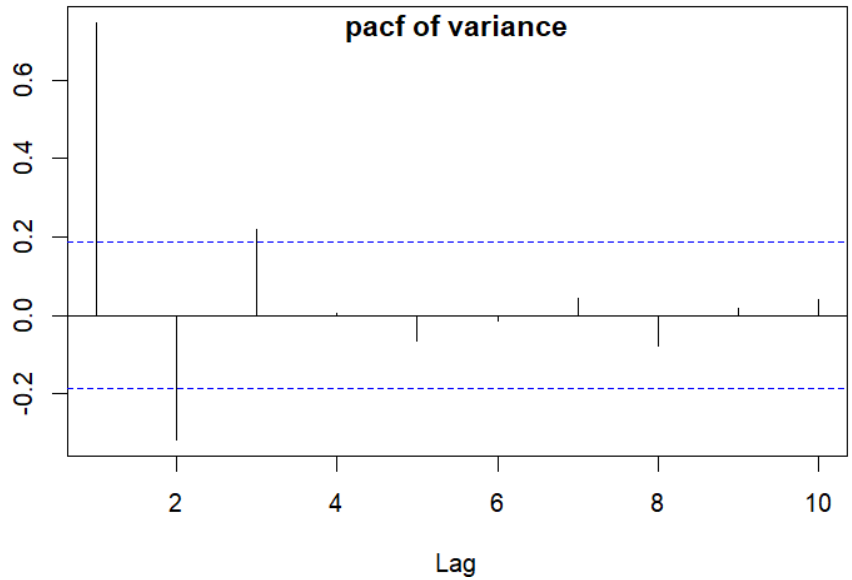
```
> # Calculate trailing VTI variance using package roll
> library(roll)  ##  Load roll
> varv <- roll::roll_var(retp, weights=rev(weightv), width=lookb)
> colnames(varv) <- "VTI.variance"
> class(varv)
> head(varv)
> sum(is.na(varv))
> varv[1:(lookb-1)] <- 0
```

Autocorrelation of Volatility

Variance calculated over non-overlapping intervals has very statistically significant autocorrelations.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> # Calculate VTI rolling variance
> lookb <- 21
> varv <- HighFreq::roll_var(retp, lookb=lookb)
> colnames(varv) <- "Variance"
> # Number of lookbv that fit over returns
> nrows <- NROW(retp)
> nagg <- nrows %/% lookb
> # Define end points with beginning stub
> endd <- c(0, nrows-lookb*nagg + (0:nagg)*lookb)
> nrows <- NROW(endd)
> # Subset variance to end points
> varv <- varv[endd]
> # Plot autocorrelation function
> rutils::plot_acf(varv, lag=10, main="ACF of Variance")
> # Plot partial autocorrelation
> pacf(varv, lag=10, main="PACF of Variance", ylab=NA)
```





Exponential Moving Average *EMA* Volatility Estimator

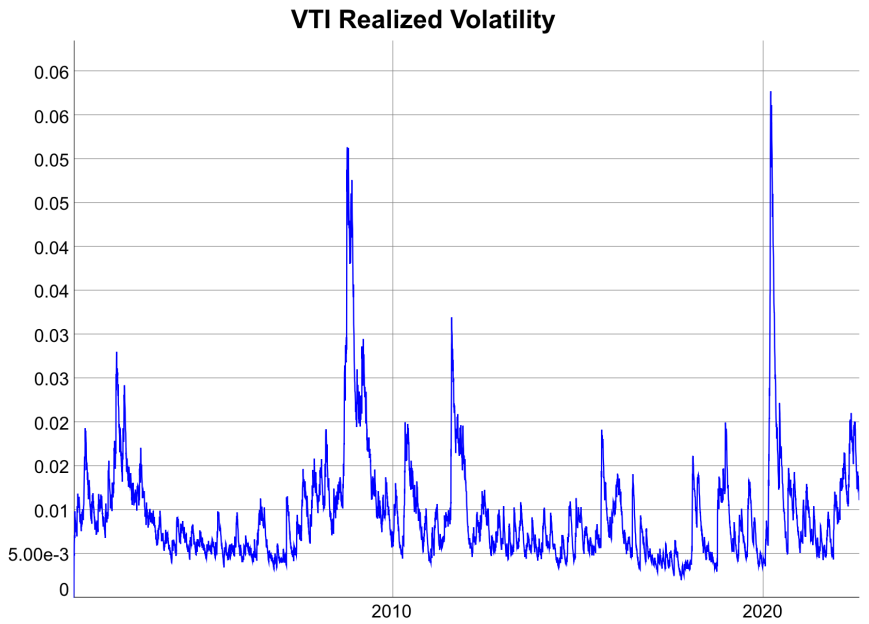
The exponential moving average (*EMA*) variance of the returns r_t can be calculated by recursively weighting the past variance estimates σ^2_{t-1}, with the squared differences of the returns minus their trailing means $(r_t - \bar{r}_t)^2$:

$$\bar{r}_t = \lambda \bar{r}_{t-1} + (1 - \lambda) r_t$$
$$\sigma^2_t = \lambda^2 \sigma^2_{t-1} + (1 - \lambda^2)(r_t - \bar{r}_t)^2$$

Where $\bar{r}_t$ and σ^2_t are the *EMA* mean and variance at time t, and λ is the decay factor.

The decay factor λ determines how quickly the mean and variance estimates are updated, with smaller values of λ producing faster updating, giving more weight to recent prices, and vice versa.

The function `HighFreq::run_var()` calculates the *EMA* mean and variance of the returns r_t.




```
> # Calculate realized variance recursively
> lambdaf <- 0.9
> volv <- HighFreq::run_var(retp, lambda=lambdaf)
> volv <- sqrt(volv[, 2])
> # Plot EMA volatility
> volv <- xts:::xts(volv, order.by=datev)
> dygraphs::dygraph(volv, main="VTI EMA Volatility") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
```

Estimating Daily Volatility From Intraday Returns

The standard *close-to-close* volatility σ depends on the *Close* prices C_i from *OHLC* data:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2$$

$$\bar{r} = \frac{1}{n}\sum_{i=0}^{n} r_i \quad r_i = \log(\frac{C_i}{C_{i-1}})$$

But intraday time series of prices (for example `HighFreq::SPY` prices), can have large overnight jumps which inflate the volatility estimates.

So the overnight returns must be divided by the overnight time interval (in seconds), which produces per second returns.

The per second returns can be multiplied by 60 to scale them back up to per minute returns.

The function `zoo::index()` extracts the time index of a time series.

The function `xts::.index()` extracts the time index expressed in the number of seconds.

```
> library(HighFreq)  ##  Load HighFreq
> # Minutely SPY returns (unit per minute) single day
> # Minutely SPY volatility (unit per minute)
> retspy <- rutils::diffit(log(SPY["2012-02-13", 4]))
> sd(retspy)
> # SPY returns multiple days (includes overnight jumps)
> retspy <- rutils::diffit(log(SPY[, 4]))
> sd(retspy)
> # Table of time intervals - 60 second is most frequent
> indeks <- rutils::diffit(xts:::.index(SPY))
> table(indeks)
> # SPY returns divided by the overnight time intervals (unit per se
> retspy <- retspy/indeks
> retspy[1] <- 0
> # Minutely SPY volatility scaled to unit per minute
> 60*sd(retspy)
```

Range Volatility Estimators of *OHLC* Time Series

Range estimators of return volatility utilize the `high` and `low` prices, and therefore have lower standard errors than the standard *close-to-close* estimator.

The *Garman-Klass* estimator uses the *low-to-high* price range, but it underestimates volatility because it doesn't account for *close-to-open* price jumps:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(0.5\log(\frac{H_i}{L_i})^2 - (2\log 2 - 1)\log(\frac{C_i}{O_i})^2)$$

The *Yang-Zhang* estimator accounts for *close-to-open* price jumps and has the lowest standard error among unbiased estimators:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\log(\frac{O_i}{C_{i-1}}) - \bar{r}_{co})^2 + 0.134(\log(\frac{C_i}{O_i}) - \bar{r}_{oc})^2 + \frac{0.866}{n}\sum_{i=1}^{n}(\log(\frac{H_i}{O_i})\log(\frac{H_i}{C_i}) + \log(\frac{L_i}{O_i})\log(\frac{L_i}{C_i}))$$

The *Yang-Zhang* (*YZ*) and *Garman-Klass-Yang-Zhang* (*GKYZ*) estimators are unbiased and have up to seven times smaller standard errors than the standard close-to-close estimator.

But in practice, prices are not observed continuously, so the price range is underestimated, and so is the variance when using the *YZ* and *GKYZ* range estimators.

Therefore in practice the *YZ* and *GKYZ* range estimators underestimate the volatility, and their standard errors are reduced less than by the theoretical amount, for the same reason.

The *Garman-Klass-Yang-Zhang* estimator is another very efficient and unbiased estimator, and also accounts for *close-to-open* price jumps:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}((\log(\frac{O_i}{C_{i-1}}) - \bar{r})^2 + 0.5\log(\frac{H_i}{L_i})^2 - (2\log 2 - 1)(\log(\frac{C_i}{O_i})^2))$$

Calculating the Trailing Range Variance Using *HighFreq*

The function `HighFreq::calc_var_ohlc()` calculates the *variance* of returns using several different range volatility estimators.

If the logarithms of the *OHLC* prices are passed into `HighFreq::calc_var_ohlc()` then it calculates the variance of percentage returns, and if simple *OHLC* prices are passed then it calculates the variance of dollar returns.

The function `HighFreq::roll_var_ohlc()` calculates the *trailing* variance of returns using several different range volatility estimators.

The functions `HighFreq::calc_var_ohlc()` and `HighFreq::roll_var_ohlc()` are very fast because they are written in C++ code.

The function `TTR::volatility()` calculates the range volatility, but it's significantly slower than `HighFreq::calc_var_ohlc()`.

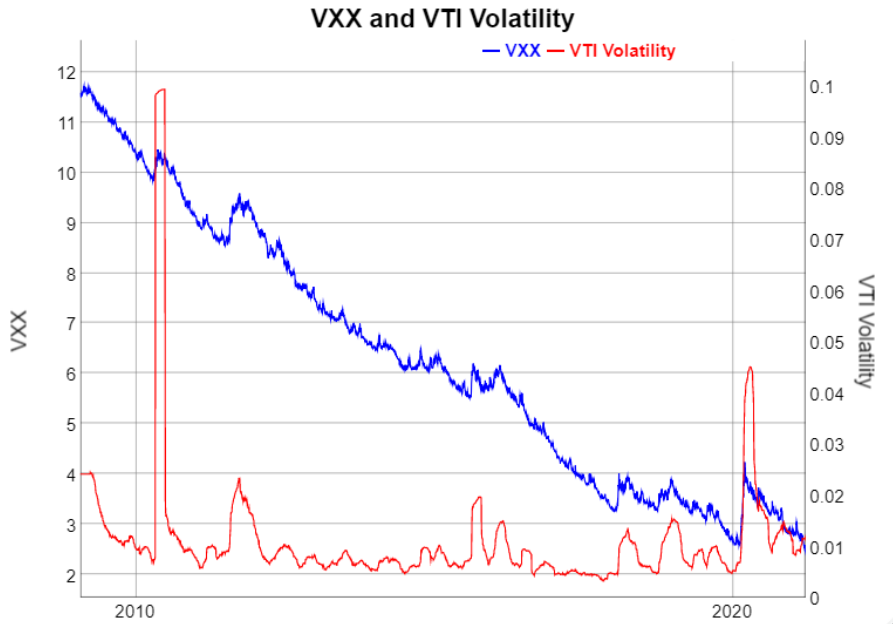
```
> library(HighFreq)  ##  Load HighFreq
> spy <- HighFreq::SPY["2008/2009"]
> # Calculate daily SPY volatility using package HighFreq
> sqrt(6.5*60*HighFreq::calc_var_ohlc(log(spy),
+   method="yang_zhang"))
> # Calculate daily SPY volatility from minutely prices using packag
> sqrt((6.5*60)*mean(na.omit(
+   TTR::volatility(spy, N=1, calc="yang.zhang"))^2))
> # Calculate trailing SPY variance using package HighFreq
> varv <- HighFreq::roll_var_ohlc(log(spy), method="yang_zhang",
+   lookb=lookb)
> # Plot range volatility
> varv <- xts:::xts(sqrt(varv), order.by=zoo::index(spy))
> dygraphs::dygraph(varv["2009-02"], main="SPY Trailing Range Volati
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
> # Benchmark the speed of HighFreq vs TTR
> library(microbenchmark)
> summary(microbenchmark(
+   ttr=TTR::volatility(rutils::etfenv$VTI, N=1, calc="yang.zhang"),
+   HighFreq=HighFreq::calc_var_ohlc(log(rutils::etfenv$VTI), method
+   times=2))[, c(1, 4, 5)]
```

VXX Prices and the Trailing Volatility

The *VXX* ETF invests in *VIX* futures, so its price is tied to the level of the *VIX* index, with higher *VXX* prices corresponding to higher levels of the *VIX* index.

The trailing volatility of past returns moves in sympathy with the implied volatility and *VXX* prices, but with a lag.

But *VXX* prices exhibit a very strong downward trend which makes them hard to compare with the trailing volatility.




```
> # Calculate VXX log prices
> vxx <- na.omit(rutils::etfenv$prices$VXX)
> datev <- zoo::index(vxx)
> lookb <- 41
> vxx <- log(vxx)
> # Calculate trailing VTI volatility
> closep <- get("VTI", rutils::etfenv)[datev]
> closep <- log(closep)
> volv <- sqrt(HighFreq::roll_var_ohlc(ohlc=closep, lookb=lookb, s
> volv[1:lookb] <- volv[lookb+1]
```

```
> # Plot dygraph of VXX and VTI volatility
> datav <- cbind(vxx, volv)
> colnames(datav)[2] <- "VTI Volatility"
> colv <- colnames(datav)
> captiont <- "VXX and VTI Volatility"
> dygraphs::dygraph(datav[, 1:2], main=captiont) %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=1, col="blue") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=1, col="red") %>%
+   dyLegend(show="always", width=300)
```

The *GARCH* Volatility Model

The *GARCH(1,1)* volatility model is defined by two coupled equations:

$$r_t = \sigma_{t-1}\xi_t$$
$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

The time-dependent variance σ_t^2, is equal to the weighted average of the *realized* variance r_t^2 and the past variance σ_{t-1}^2.

The source of uncertainty are the returns r_t, which are proportional to the standard normal innovations ξ_t.

The parameter α is the weight associated with recent realized variance updates, and β is the weight associated with the past variance.

The long-term equi value of the variance is proportional to the parameter ω:

$$\sigma_{eq}^2 = \frac{\omega}{1-\alpha-\beta}$$

So the sum of α plus β should be less than 1, otherwise the volatility is explosive.

The difference between the *GARCH(1,1)* estimator and the *EMA* estimator is that the *GARCH(1,1)* decays to a non-zero equilibrium value σ_{eq}^2, while the *EMA* estimator decays to zero.

```
> # Define GARCH parameters
> alphac <- 0.3; betac <- 0.5;
> omega <- 1e-4*(1 - alphac - betac)
> nrows <- 1000
> # Calculate matrix of standard normal innovations
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  ##  R
> innov <- rnorm(nrows)
> retp <- numeric(nrows)
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> retp[1] <- sqrt(varv[1])*innov[1]
> # Simulate GARCH model
> for (i in 2:nrows) {
+   retp[i] <- sqrt(varv[i-1])*innov[i]
+   varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+ }  ##  end for
> # Simulate the GARCH process using Rcpp
> garchsim <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> all.equal(garchsim, cbind(retp, varv), check.attributes=FALSE)
```

The *GARCH* process is path dependent, so it must be simulated using an explicit loop, and it's better to perform it in C++ instead of R.

NYU Professor Robert Engle was awarded the Nobel Prize in Economics for developing the *GARCH* volatility model.

GARCH Volatility Time Series

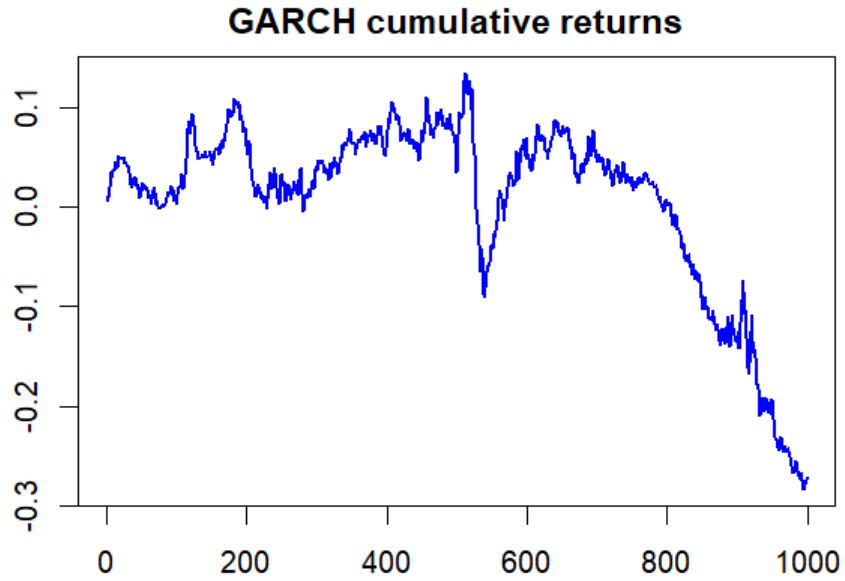
The simulated *GARCH* volatility exhibits spikes of volatility followed by an exponential decay.

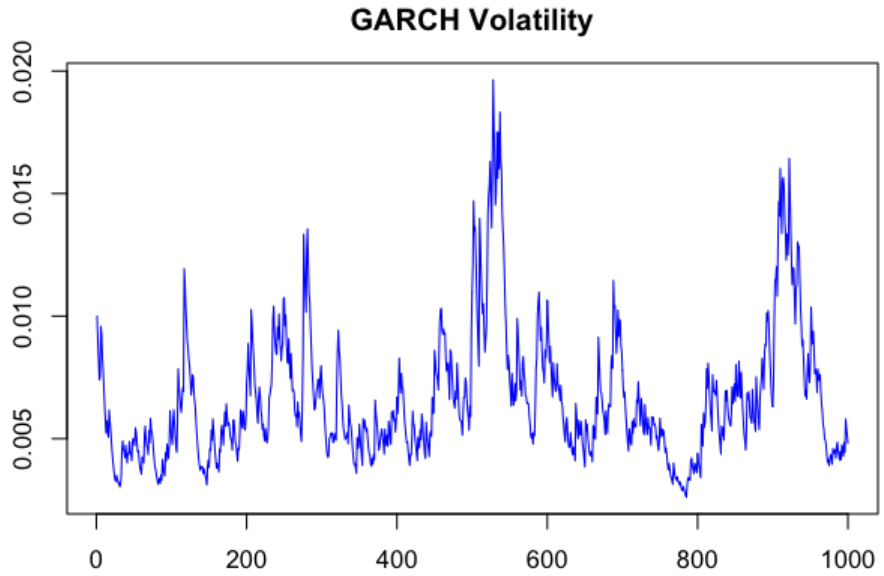
Larger values of α produce a stronger feedback between the simulated returns and variance, which produce larger variance spikes, which produce larger kurtosis.

The parameter α is the weight of the squared realized returns in the variance.

But the decay of the volatility in the *GARCH* model is faster than what is observed in practice.

```
> # Open plot window on Mac
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot GARCH cumulative returns
> plot(cumsum(retp), t="l", col="blue", xlab="", ylab="",
+   main="GARCH Cumulative Returns")
> quartz.save("figure/garch_returns.png", type="png",
+   width=6, height=5)
> # Plot GARCH volatility
> plot(sqrt(varv), t="l", col="blue", xlab="", ylab="",
+   main="GARCH Volatility")
> quartz.save("figure/garch_volat.png", type="png",
+   width=6, height=5)
```





GARCH Returns Distribution

The return process r_t follows a normal distribution, *conditional* on the variance in the previous period σ^2_{t-1}.

$$r_t = \sigma_{t-1}\xi_t$$
$$\sigma^2_t = \omega + \beta\sigma^2_{t-1} + \alpha r^2_t$$

But the *unconditional* distribution of returns is *not* normal, since their standard deviation is time-dependent, so they are *leptokurtic* (fat tailed).

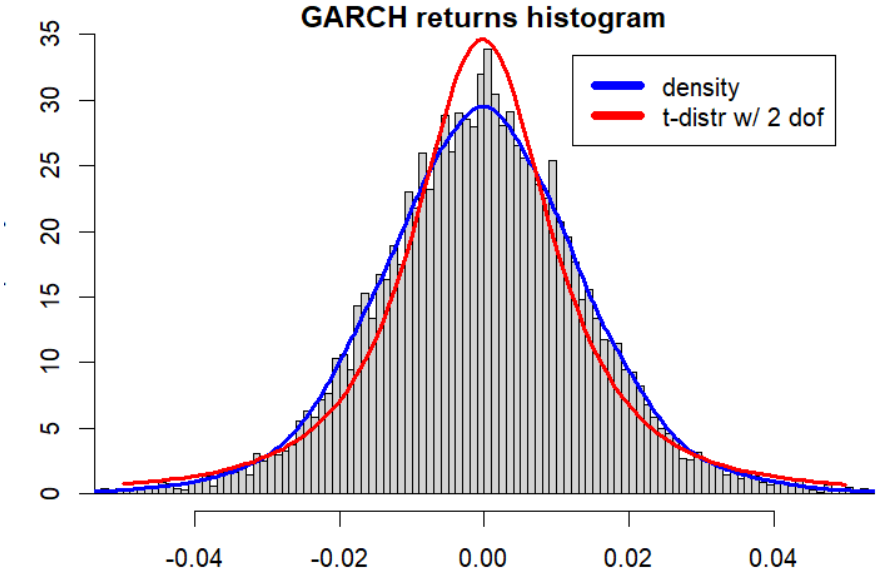
The *GARCH* volatility model produces *leptokurtic* return distribution, with fat tails.

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

Student's *t-distribution* with 3 degrees of freedom is often used to represent asset returns.

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

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




```
> # Calculate kurtosis of GARCH returns
> mean(((retp-mean(retp))/sd(retp))^4)
> # Perform Jarque-Bera test of normality
> tseries::jarque.bera.test(retp)
```

```
> # Fit t-distribution into GARCH returns
> fitobj <- MASS::fitdistr(retp, densfun="t", df=2)
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
> # Plot histogram of GARCH returns
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.03, 0.03),
+   ylab="frequency", freq=FALSE, main="GARCH Returns Histogram")
> lines(density(retp, adjust=1.5), lwd=2, col="blue")
> curve(expr=dt((x-locv)/scalev, df=2)/scalev,
+   type="l", xlab="", ylab="", lwd=2,
+   col="red", add=TRUE)
> legend("topright", inset=-0, bty="n", y.intersp=0.4,
+  leg=c("density", "t-distr w/ 2 dof"),
+  lwd=6, lty=1, col=c("blue", "red"))
> quartz.save("figure/garch_hist.png", type="png", width=6, height=5
```

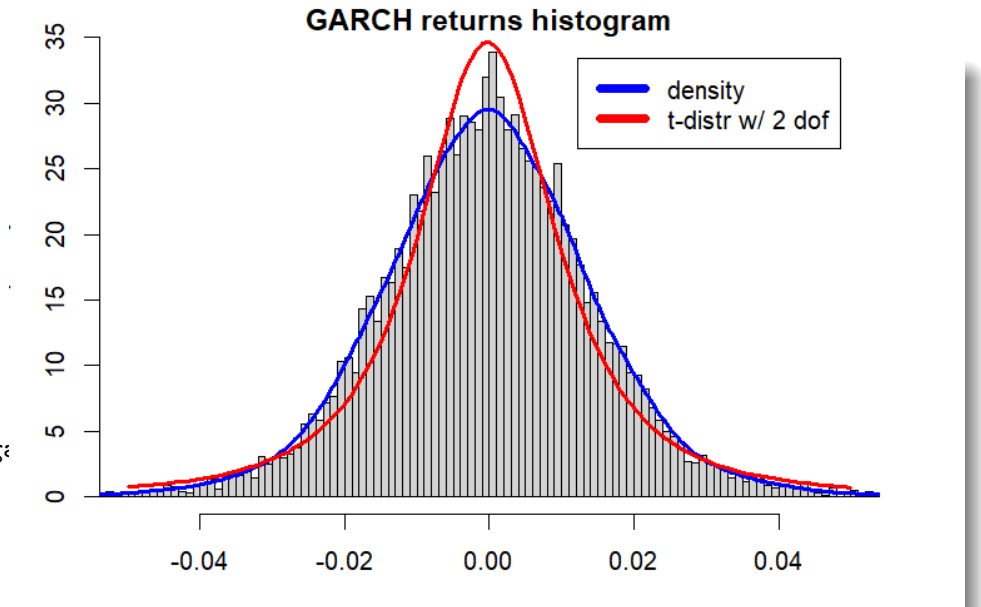
GARCH Model Simulation

The package *fGarch* contains functions for applying *GARCH* models.

The function `fGarch::garchSpec()` specifies a *GARCH* model.

The function `fGarch::garchSim()` simulates a *GARCH* model, but it uses its own random innovations, so its output is not reproducible.

```
> # Specify GARCH model
> garch_spec <- fGarch::garchSpec(model=list(ar=c(0, 0), omega=omega
+   alpha=alphac, beta=betac))
> # Simulate GARCH model
> garch_sim <- fGarch::garchSim(spec=garch_spec, n=nrows)
> retp <- as.numeric(garch_sim)
> # Calculate kurtosis of GARCH returns
> moments::moment(retp, order=4) /
+   moments::moment(retp, order=2)^2
> # Perform Jarque-Bera test of normality
> tseries::jarque.bera.test(retp)
> # Plot histogram of GARCH returns
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.05, 0.05),
+   ylab="frequency", freq=FALSE,
+   main="GARCH Returns Histogram")
> lines(density(retp, adjust=1.5), lwd=3, col="blue")
```




```
> # Fit t-distribution into GARCH returns
> fitobj <- MASS::fitdistr(retp, densfun="t", df=2, lower=c(-1, 1e-7
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
> curve(expr=dt((x-locv)/scalev, df=2)/scalev,
+   type="l", xlab="", ylab="", lwd=3,
+   col="red", add=TRUE)
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+  leg=c("density", "t-distr w/ 2 dof"),
+  lwd=6, lty=1, col=c("blue", "red"))
```

GARCH Returns Kurtosis

The expected value of the variance σ^2 of *GARCH* returns is proportional to the parameter ω:

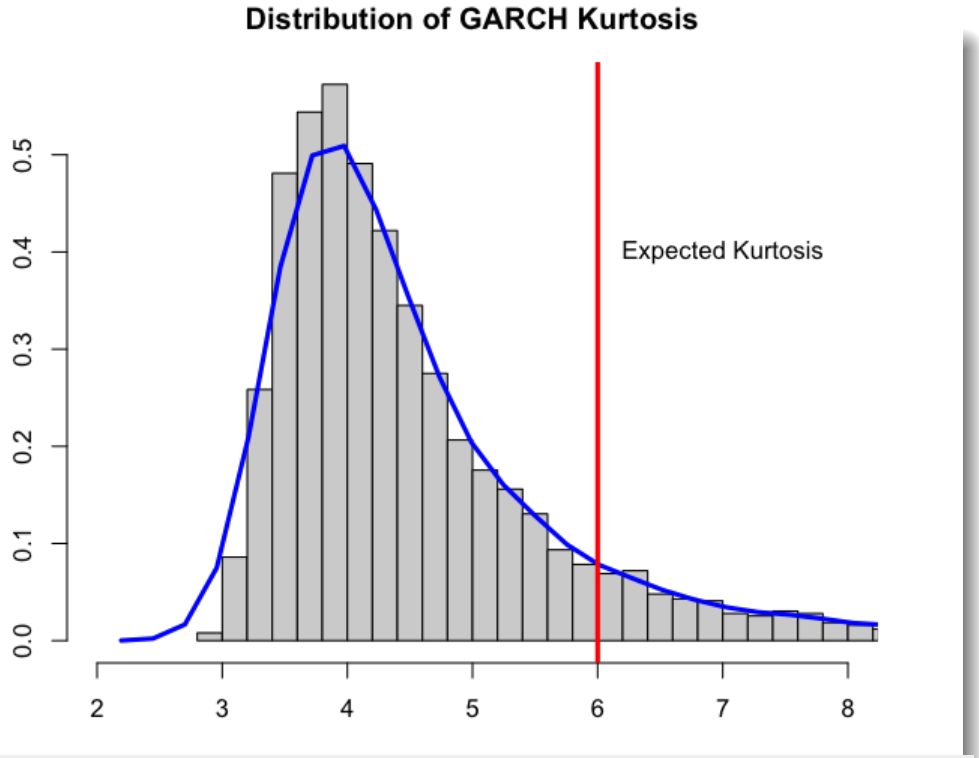
$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

The expected value of the kurtosis κ of *GARCH* returns is equal to:

$$\kappa = 3 + \frac{6\alpha^2}{1 - 2\alpha^2 - (\alpha + \beta)^2}$$

The excess kurtosis $\kappa - 3$ is proportional to α^2 because larger values of the parameter α produce larger variance spikes which produce larger kurtosis.

The distribution of kurtosis is highly positively skewed, especially for short returns samples, so most kurtosis values will be significantly below their expected value.




```
> # Calculate variance of GARCH returns
> var(retp)
> # Calculate expected value of variance
> omega/(1 - alphac - betac)
> # Calculate kurtosis of GARCH returns
> mean(((retp-mean(retp))/sd(retp))^4)
> # Calculate expected value of kurtosis
> 3 + 6*alpha^2/(1-2*alpha^2-(alphac+betac)^2)
```

```
> # Calculate the distribution of GARCH kurtosis
> kurt <- sapply(1:1e4, function(x) {
+   garchsim <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+     beta=betac, innov=matrix(rnorm(nrows)))
+   retp <- garchsim[, 1]
+   c(var(retp), mean(((retp-mean(retp))/sd(retp))^4))
+ })  ##  end sapply
> kurt <- t(kurt)
> apply(kurt, 2, mean)
> # Plot the distribution of GARCH kurtosis
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> histp <- hist(kurt[, 2], breaks=500, col="lightgrey",
+   xlim=c(2, 8), xlab="returns", ylab="frequency", freq=FALSE,
+   main="Distribution of GARCH Kurtosis")
```

GARCH Variance Estimation

The *GARCH* model can be used to estimate the trailing variance of empirical (historical) returns.

If the time series of returns r_t is given, then it can be used in the *GARCH(1,1)* formula to estimate the trailing variance σ_t^2:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

If the simulated returns from the *GARCH(1,1)* model are used in the above formula, then it produces the simulated *GARCH(1,1)* variance.

But to estimate the trailing variance of historical returns, the parameters ω, α, and β must be first estimated through model calibration.

```
> # Simulate the GARCH process using Rcpp
> garchsim <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> # Extract the returns
> retp <- garchsim[, 1]
> # Estimate the trailing variance from the returns
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> for (i in 2:nrows) {
+   varv[i] <- omega + alphac*retp[i]^2 +
+     betac*varv[i-1]
+ }  ##  end for
> all.equal(garchsim[, 2], varv, check.attributes=FALSE)
```

GARCH Model Calibration

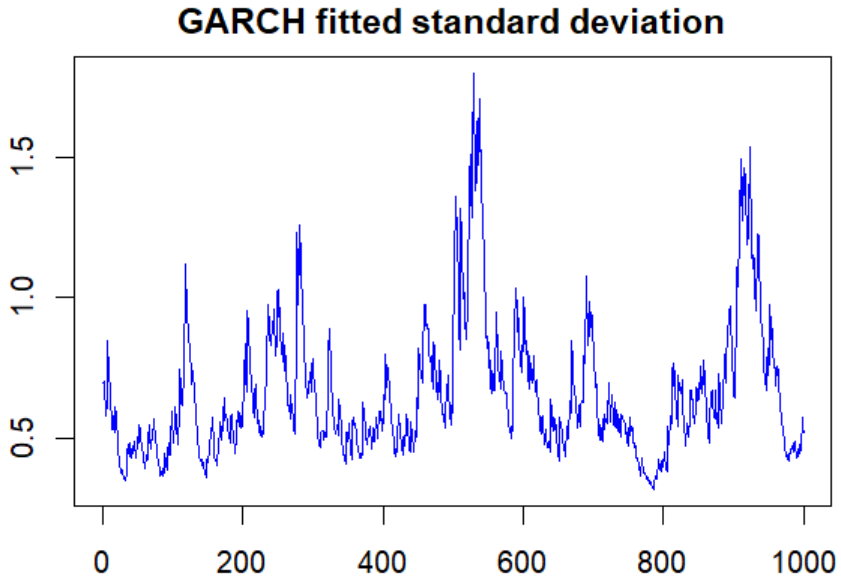
GARCH models can be calibrated from the returns using the *maximum-likelihood* method.

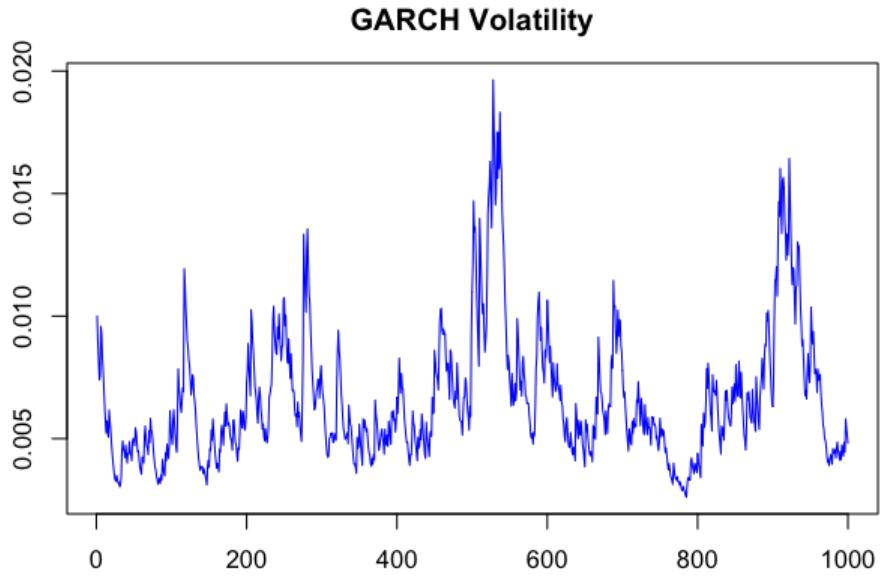
But it's a complex optimization procedure which requires a large amount of data for accurate results.

The function `fGarch::garchFit()` calibrates a *GARCH* model on a time series of returns.

The function `garchFit()` returns an S4 object of class *fGARCH*, with multiple slots containing the *GARCH* model outputs and diagnostic information.

```
> library(fGarch)
> # Fit returns into GARCH
> garchfit <- fGarch::garchFit(data=retp)
> # Fitted GARCH parameters
> garchfit@fit$coef
> # Actual GARCH parameters
> c(mu=mean(retp), omega=omega,alpha=alphac, beta=betac)
> # Plot GARCH fitted volatility
> plot(sqrt(garchfit@fit$series$h), t="l",
+   col="blue", xlab="", ylab="",
+   main="GARCH Fitted Volatility")
> quartz.save("figure/garch_fGarch_fitted.png",
+   type="png", width=6, height=5)
```





GARCH Likelihood Function

Under the *GARCH(1,1)* volatility model, the returns follow the process: $r_t = \sigma_{t-1}\xi_t$. (We can assume that the returns have been centered.)

So the *conditional* distribution of returns is normal with standard deviation equal to σ_{t-1}:

$$\phi(r_t, \sigma_{t-1}) = \frac{e^{-r_t^2/2\sigma_{t-1}^2}}{\sqrt{2\pi}\sigma_{t-1}}$$

The *log-likelihood* function $\mathcal{L}(\omega, \alpha, \beta|r_t)$ for the normally distributed returns is therefore equal to:

$$\mathcal{L}(\omega, \alpha, \beta|r_t) = -\sum_{t=1}^{n}(\frac{r_t^2}{\sigma_{t-1}^2} + \log(\sigma_{t-1}^2))$$

The *log-likelihood* depends on the *GARCH(1,1)* parameters ω, α, and β because the trailing variance σ_t^2 depends on the *GARCH(1,1)* parameters:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

The *GARCH* process must be simulated using an explicit loop, so it's better to perform it in `C++` instead of `R`.

```
> # Define likelihood function
> likefun <- function(omega, alphac, betac) {
+   ##  Estimate the trailing variance from the returns
+   varv <- numeric(nrows)
+   varv[1] <- omega/(1 - alphac - betac)
+   for (i in 2:nrows) {
+     varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+   }  ##  end for
+   varv <- ifelse(varv > 0, varv, 0.000001)
+   ##  Lag the variance
+   varv <- rutils::lagit(varv, pad_zeros=FALSE)
+   ##  Calculate the likelihood
+   -sum(retp^2/varv + log(varv))
+ }  ##  end likefun
> # Calculate the likelihood in R
> likefun(omega, alphac, betac)
> # Calculate the likelihood in Rcpp
> HighFreq::lik_garch(omega=omega, alpha=alphac,
+   beta=betac, returns=matrix(retp))
> # Benchmark speed of likelihood calculations
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode=likefun(omega, alphac, betac),
+   Rcpp=HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac,
+   ), times=10)[, c(1, 4, 5)]
```

GARCH Likelihood Function Matrix

The *GARCH(1,1) log-likelihood* function depends on three parameters $\mathcal{L}(\omega, \alpha, \beta | r_t)$.

The more parameters the harder it is to find their optimal values using optimization.

We can simplify the optimization task by assuming that the equilibrium variance is equal to the realized variance:

$$\sigma^2_{eq} = \frac{\omega}{1 - \alpha - \beta} = \frac{1}{n-1} \sum_{t=1}^{n} (r_t - \bar{r})^2$$

This way the *log-likelihood* becomes a function of only two parameters, say α and β.

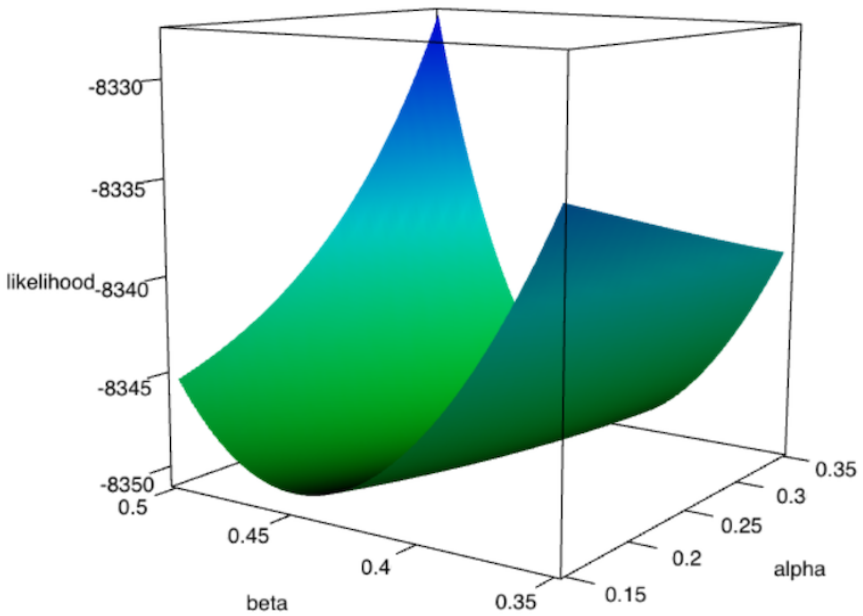
```
> # Calculate the variance of returns
> retp <- garchsim[, 1, drop=FALSE]
> varv <- var(retp)
> retp <- (retp - mean(retp))
> # Calculate likelihood as function of alpha and betac parameters
> likefun <- function(alphac, betac) {
+   omega <- variance*(1 - alpha - betac)
+   -HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac, retu
+ }  ##  end likefun
> # Calculate matrix of likelihood values
> alphas <- seq(from=0.15, to=0.35, len=50)
> betac <- seq(from=0.35, to=0.5, len=50)
> likmat <- sapply(alphacs, function(alphac) sapply(betac,
+   function(betac) likefun(alphac, betac)))
```

GARCH Likelihood Perspective Plot

The perspective plot shows that the *log-likelihood* is much more sensitive to the β parameter than to α.

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a *vectorized* function or a matrix.

The optimal values of α and β can be found approximately using a grid search on the *log-likelihood* matrix.




```
> # Set rgl options and load package rgl
> options(rgl.useNULL=TRUE); library(rgl)
> # Draw and render 3d surface plot of likelihood function
> ncols <- 100
> color <- rainbow(ncols, start=2/6, end=4/6)
> zcols <- cut(likmat, ncols)
> rgl::persp3d(alphacs, betac, likmat, col=color[zcols],
+   xlab="alpha", ylab="beta", zlab="likelihood")
> rgl::rglwidget(elementId="plot3drgl", width=700, height=700)
> # Perform grid search
> coord <- which(likmat == min(likmat), arr.ind=TRUE)
> c(alphacs[coord[2]], betac[coord[1]])
> likmat[coord]
> likefun(alphacs[coord[2]], betac[coord[1]])
> # Optimal and actual parameters
> options(scipen=2)  ##  Use fixed not scientific notation
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+   optimal=c(alphacs[coord[2]], betac[coord[1]], variance*(1 - sum(alphacs[coord[2]], betac[coord[1]]))))
```

GARCH Likelihood Function Optimization

The flat shape of the *GARCH* likelihood function makes it difficult for steepest descent optimizers to find the best parameters.

The function `DEoptim()` from package *DEoptim* performs *global* optimization using the *Differential Evolution* algorithm.

Differential Evolution is a genetic algorithm which evolves a population of solutions over several generations:

https://link.springer.com/content/pdf/10.1023/A:1008202821328.pdf

The first generation of solutions is selected randomly.

Each new generation is obtained by combining the best solutions from the previous generation.

The *Differential Evolution* algorithm is well suited for very large multi-dimensional optimization problems, such as portfolio optimization.

Gradient optimization methods are more efficient than *Differential Evolution* for smooth objective functions with no local minima.

```
> # Define vectorized likelihood function
> likefun <- function(x, retp) {
+   alphac <- x[1]; betac <- x[2]; omega <- x[3]
+   -HighFreq::lik_garch(omega=omega, alpha=alphac, beta=betac, retu
+ }  ##  end likefun
> # Initial parameters
> initp <- c(alphac=0.2, beta=0.4, omega=varv/0.2)
> # Find max likelihood parameters using steepest descent optimizer
> fitobj <- optim(par=initp,
+   fn=likefun, ##  Log-likelihood function
+   method="L-BFGS-B", ##  Quasi-Newton method
+   retp=retp,
+   upper=c(0.35, 0.55, varv), ##  Upper constraint
+   lower=c(0.15, 0.35, varv/100)) ##  Lower constraint
> # Optimal and actual parameters
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+ optimal=c(fitobj$par["alpha"], fitobj$par["beta"], fitobj$par["ome
> # Find max likelihood parameters using DEoptim
> optiml <- DEoptim::DEoptim(fn=likefun,
+   upper=c(0.35, 0.55, varv), ##  Upper constraint
+   lower=c(0.15, 0.35, varv/100), ##  Lower constraint
+   retp=retp,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal and actual parameters
> cbind(actual=c(alphac=alphac, beta=betac, omega=omega),
+ optimal=c(optiml$optim$bestmem[1], optiml$optim$bestmem[2], optiml
```

GARCH Variance of Stock Returns

The *GARCH* model can be used to estimate the trailing variance of empirical (historical) returns.

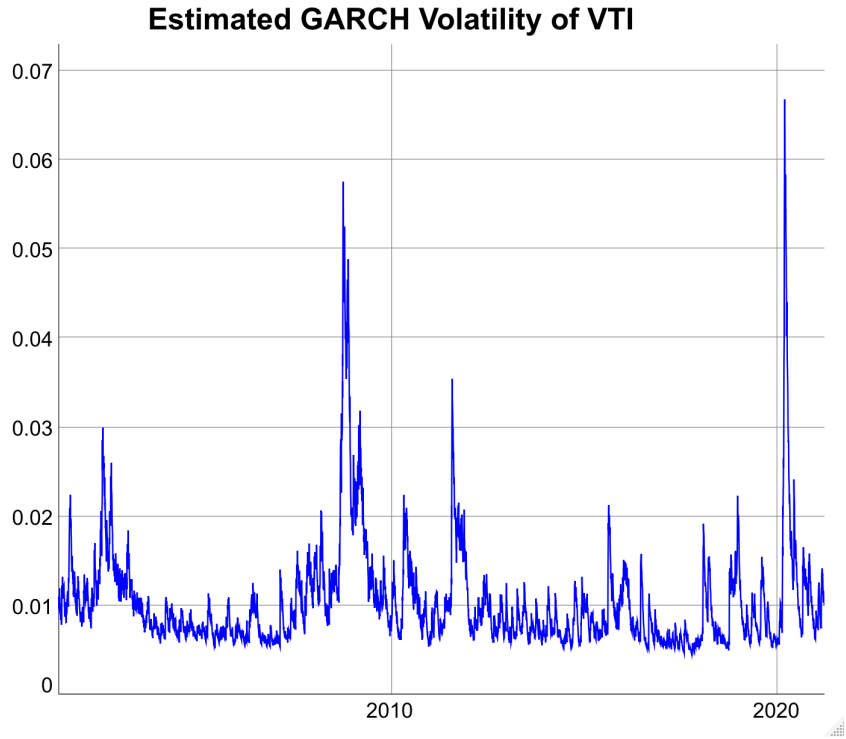
If the time series of returns r_t is given, then it can be used in the *GARCH(1,1)* formula to estimate the trailing variance σ_t^2:

$$\sigma_t^2 = \omega + \beta\sigma_{t-1}^2 + \alpha r_t^2$$

The *GARCH* estimator of the trailing variance is a generalization of the exponential moving average (*EMA*) variance estimator:

$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2$$

The main difference is that the *GARCH* model has a non-zero equilibrium value of the variance σ_{eq}^2, while the *EMA* estimator decays to zero.




```
> # Calculate VTI returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> # Find max likelihood parameters using DEoptim
> optiml <- DEoptim::DEoptim(fn=likefun,
+   upper=c(0.4, 0.9, varv), ##  Upper constraint
+   lower=c(0.1, 0.5, varv/100), ##  Lower constraint
+   retp=retp,
+   control=list(trace=FALSE, itermax=1000, parallelType=1))
> # Optimal parameters
> paramv <- unname(optiml$optim$bestmem)
> alphac <- paramv[1]; betac <- paramv[2]; omega <- paramv[3]
> c(alphac, betac, omega)
> # Equilibrium GARCH variance
> omega/(1 - alphac - betac)
> drop(var(retp))
```

```
> # Estimate the GARCH volatility of VTI returns
> nrows <- NROW(retp)
> varv <- numeric(nrows)
> varv[1] <- omega/(1 - alphac - betac)
> for (i in 2:nrows) {
+   varv[i] <- omega + alphac*retp[i]^2 + betac*varv[i-1]
+ }  ##  end for
> # Estimate the GARCH volatility using Rcpp
> garchsim <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=retp, is_random=FALSE)
> all.equal(garchsim[, 2], varv, check.attributes=FALSE)
> # Plot dygraph of the estimated GARCH volatility
> dygraphs::dygraph(xts::xts(sqrt(varv), zoo::index(retp)),
+   main="Estimated GARCH Volatility of VTI") %>%
+   dyOptions(colors="blue") %>% dyLegend(show="always", width=300)
```

GARCH Variance Forecasts

The *GARCH* model can't forecast the volatility spikes. It only forecasts the exponential decay of the volatility after a spike.

The one-step-ahead forecast of the squared returns is equal to their expected value: $r_{t+1}^2 = \mathbb{E}[(\sigma_t \xi_t)^2] = \sigma_t^2$.

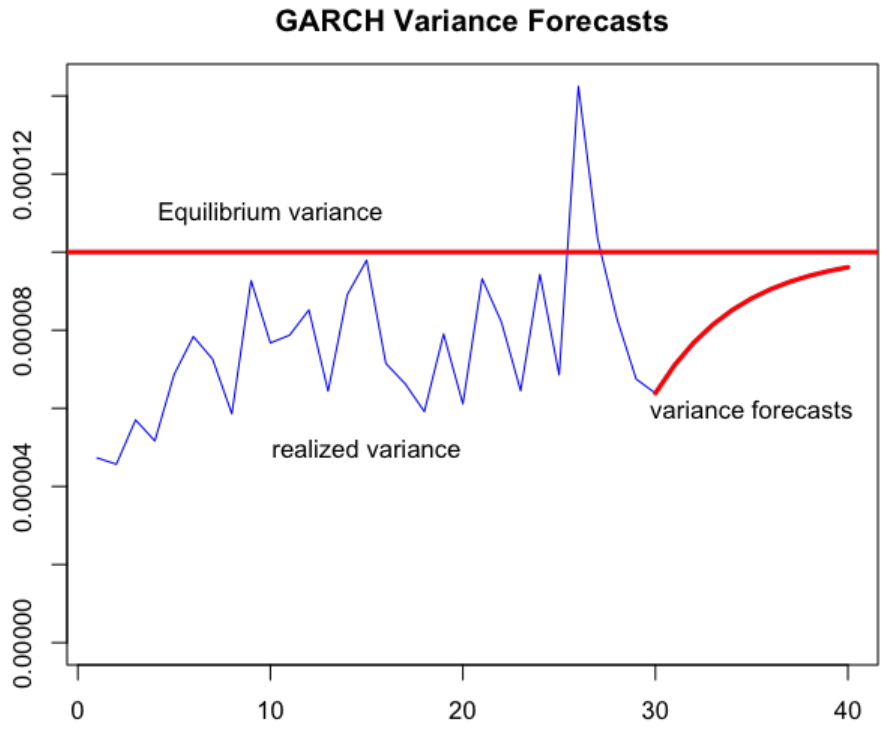
The variance forecasts depend on the previous variance: $\sigma_{t+1}^2 = \mathbb{E}[\omega + \alpha r_{t+1}^2 + \beta \sigma_t^2] = \omega + (\alpha + \beta)\sigma_t^2$

The variance forecasts gradually decay to the equilibrium value σ_{eq}^2, such that the forecast is equal to itself: $\sigma_{eq}^2 = \omega + (\alpha + \beta)\sigma_{eq}^2$.

This gives: $\sigma_{eq}^2 = \frac{\omega}{1-\alpha-\beta}$, which is the long-term expected value of the variance.

So the variance forecasts decay exponentially to their equilibrium value σ^2 at the decay rate equal to $(\alpha + \beta)$:

$$\sigma_{t+1}^2 - \sigma_{eq}^2 = (\alpha + \beta)(\sigma_t^2 - \sigma_{eq}^2)$$




```
> # Simulate GARCH model
> garchsim <- HighFreq::sim_garch(omega=omega, alpha=alphac,
+   beta=betac, innov=matrix(innov))
> varv <- garchsim[, 2]
> # Calculate the equilibrium variance
> vareq <- omega/(1 - alphac - betac)
> # Calculate the variance forecasts
> varf <- numeric(10)
> varf[1] <- vareq + (alphac + betac)*(xts::last(varv) - vareq)
> for (i in 2:10) {
+   varf[i] <- vareq + (alphac + betac)*(varf[i-1] - vareq)
+ }  ##  end for
```

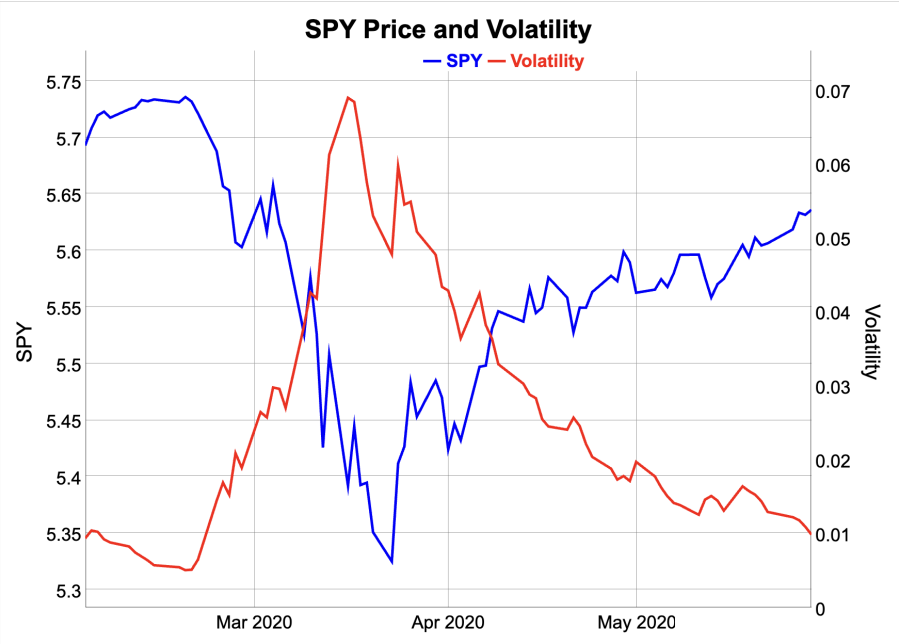
```
> # Open plot window on Mac
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot GARCH variance forecasts
> plot(tail(varv, 30), t="l", col="blue", xlab="", ylab="",
+   xlim=c(1, 40), ylim=c(0, max(tail(varv, 30))),
+   main="GARCH Variance Forecasts")
> text(x=15, y=0.5*vareq, "realized variance")
> lines(x=30:40, y=c(xts::last(varv), varf), col="red", lwd=3)
> text(x=35, y=0.6*vareq, "variance forecasts")
> abline(h=vareq, lwd=3, col="red")
> text(x=10, y=1.1*vareq, "Equilibrium variance")
> quartz.save("figure/garch_forecast.png", type="png", width=6, heig
```

Stock Prices and Volatility

The volatility typically rises when stock prices drop, and it falls when stock prices rise, so the volatility is negatively correlated with the stock returns.

The negative correlation between stock returns and volatility is especially strong for months with very high volatility, which are usually associated with large negative returns.

```
> # Calculate SPY percentage returns
> symboln <- "SPY"
> pricev <- log(na.omit(get(symboln, rutils::etfenv$prices)))
> datev <- zoo::index(pricev)
> retp <- rutils::diffit(pricev)
> # Calculate the EMA volatility
> volma <- sqrt(HighFreq::run_var(retp, lambda=0.9)[, 2])
> datav <- cbind(pricev, volma)
> colnames(datav)[2] <- "Volatility"
```



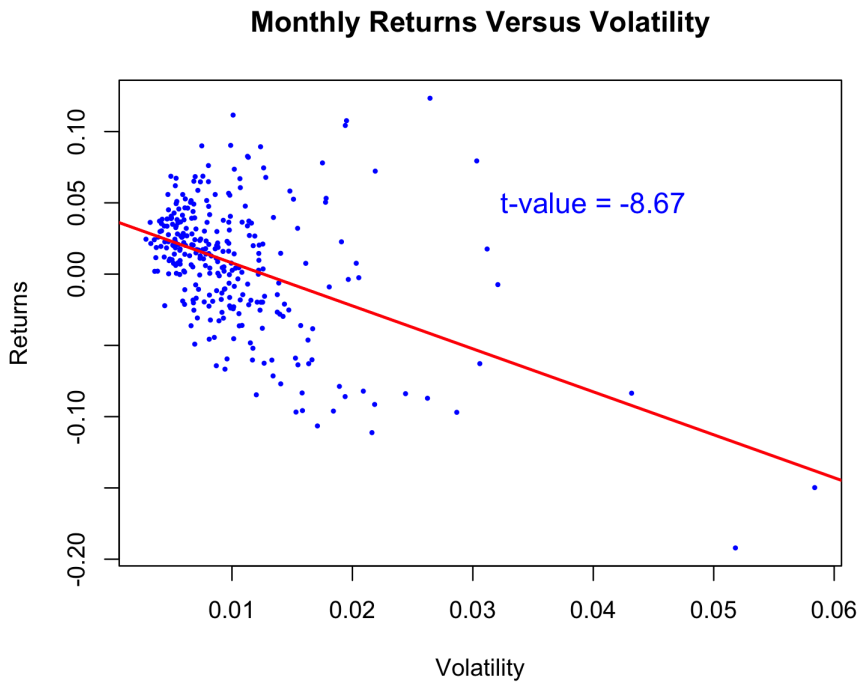

```
> # Plot dygraph of the SPY price and EMA volatility
> colv <- colnames(datav)
> dygraphs::dygraph(datav["2020-02/2020-05"],
+   main="SPY Price and Volatility") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=2, col="blue") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=2, col="red") %>%
+   dyLegend(show="always", width=300)
```

Stock Returns and Volatility

The volatility typically rises when stock prices drop, and it falls when stock prices rise, so the volatility is negatively correlated with the stock returns.

The negative correlation between stock returns and volatility is especially strong for months with very high volatility, which are usually associated with large negative returns.

```
> # Calculate monthly end points
> endd <- rutils::calc_endpoints(retp, interval="months")
> npts <- NROW(endd)
> # Calculate monthly returns and volatilities
> retvol <- sapply(2:npts, function(tday) {
+   retis <- retp[(endd[tday-1]+1):endd[tday]]
+   return(c(ret=sum(retis), vol=sd(retis)))
+ })  # end sapply
> retvol <- t(retvol)
```




```
> # Perform regression of returns versus the volatilities
> formobj <- as.formula(paste(colnames(retvol), collapse=" ~ "))
> regmod <- lm(formobj, data=as.data.frame(retvol))
> summary(regmod)$coefficients
> # Plot the returns versus the volatilities with regression line
> plot(formula=formobj, data=retvol,
+      main="Monthly Returns Versus Volatility",
+      pch=20, cex=0.5, col="blue",
+      xlab="Volatility", ylab="Returns")
> abline(regmod, col="red", lwd=2)
> text(x=0.04, y=0.05, cex=1.2, col="blue",
+   labels=paste("t-value =", round(summary(regmod)$coefficients[2,
```

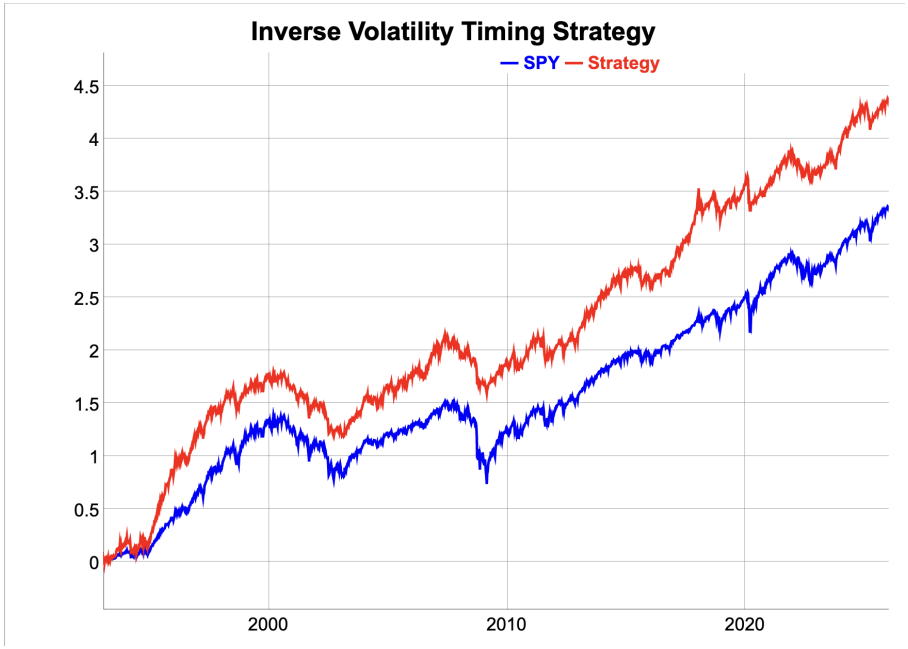
Market Timing Using the Inverse Volatility

Volatility timing strategies use the trailing volatility to change market positions.

Volatility timing strategies increase the risk exposure when the volatility is low or decreasing, and decrease it when the volatility is high or increasing

A simple volatility timing strategy is to set the position size proportional to the inverse of the trailing volatility, so that the strategy has a constant volatility.

```
> # The bid-ask spread for the most liquid ETFs
> bidask <- 0.0001
> # Calculate the EMA volatility
> lambdaf <- 0.99
> volma <- HighFreq::run_var(retp, lambda=lambdaf)
> volma <- sqrt(volma[, 2])
> # The positions are the inverse of the volatility
> posv <- 1/volma
> posv <- rutils::lagit(posv)
> dygraphs::dygraph(xts(posv, datev),
+   main="Positions of Volatility Timing Strategy",
+   ylab="Position Size", xlab="")
> # Calculate the PnLs minus the transaction costs
> pnls <- retp*posv
> colnames(pnls) <- "Vol Timing"
> costv <- 0.5*bidask*abs(rutils::diffit(posv))
> pnls <- (pnls - costv)
> pnls <- pnls*sd(retp[retp<0])/sd(pnls[pnls<0])
```




```
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retp, pnls)
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of the strategy
> endw <- rutils::calc_endpoints(pricev, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Inverse Volatility Timing Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

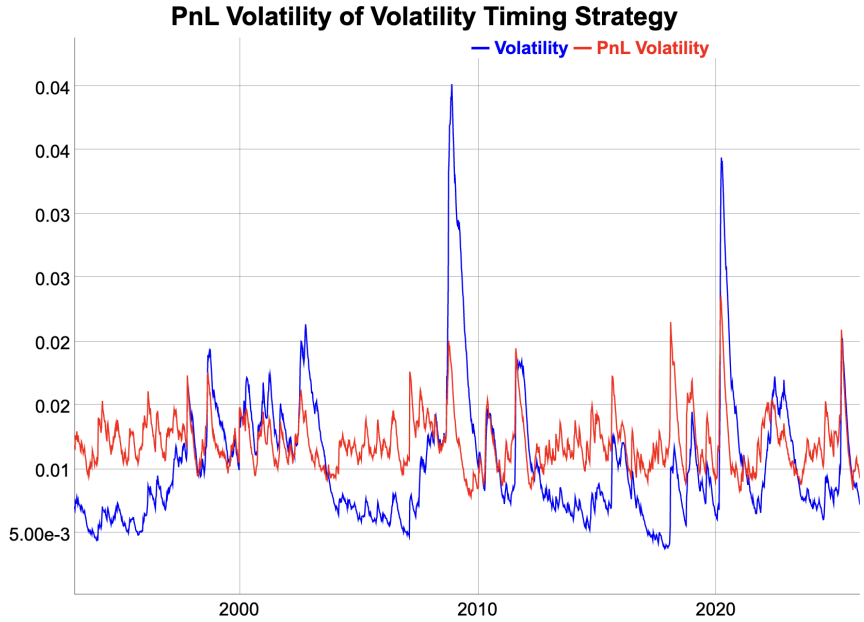
The PnL Variability of Volatility Timing Strategies

The pnl variability of volatility timing strategies is lower than that of the underlying stock, because the strategy reduces the risk exposure when the volatility is higher, which is often accompanied by negative returns.

So volatility timing are similar to volatility targeting, because they both reduce the risk exposure when the volatility is higher.

Volatility timing is also similar to the risk parity strategy, which has weights inversely proportional to the volatility, so it also reduces the risk exposure when the volatility is higher.

The kurtosis of the timing strategy PnLs is smaller than that of the stock returns, but their skewness is more negative.




```
> # Calculate the skewness and kurtosis of the stock returns and tl
> retc <- cbind(retp, pnls)
> apply(retc, 2, function(x) {
+   ##  Standardize the returns
+   x <- (x - mean(x))/sd(x)
+   c(skew=mean(x^3), kurt=mean(x^4))
+ }) ##  end apply
> # Calculate the PnL volatility
> volpnl <- HighFreq::run_var(pnls, lambda=lambdaf)
> volpnl <- sqrt(volpnl[, 2])
> datav <- cbind(volma, volpnl)
> colnames(datav) <- c("Volatility", "PnL Volatility")
> datav <- xts(datav, datev)
```

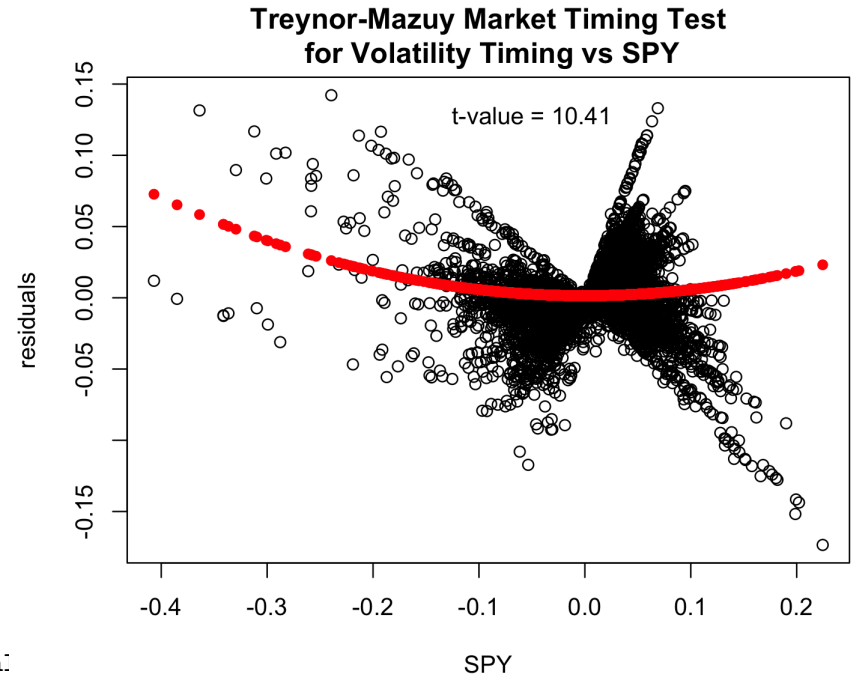
```
> # Plot dygraph of the PnL volatility
> dygraphs::dygraph(datav[endw],
+   main="PnL Volatility of Volatility Timing Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=1) %>%
+   dyLegend(show="always", width=300)
```

Volatility Strategy Market Timing Skill

The volatility timing strategy has significant market timing skill over longer time intervals, because it reduces the risk exposure when the volatility is higher, which is often accompanied by negative returns.

This allows the volatility timing strategy to time the markets better - selling when prices are droppng and buying when prices are rising.

```
> # Test market timing skill of the volatility timing strategy
> desm <- cbind(pnls, retp)
> desm <- HighFreq::roll_sum(desm, lookb=22)
> desm <- cbind(desm, desm[, 2]^2)
> colnames(desm) <- c("voltiming", "SPY", "Treynor")
> regmod <- lm(voltiming ~ SPY + Treynor, data=as.data.frame(desm))
> summary(regmod)
> # Plot residual scatterplot
> resids <- regmod$residuals
> plot.default(x=desm[, "SPY"], y=resids, xlab="SPY", ylab="residual
> title(main="Treynor-Mazuy Market Timing Test\n for Volatility Timi
```




```
> # Plot fitted (predicted) response values
> coefreg <- summary(regmod)$coeff
> fitv <- regmod$fitted.values - coefreg["SPY", "Estimate"]*desm[,
> tvalue <- round(coefreg["Treynor", "t value"], 2)
> points.default(x=desm[, "SPY"], y=fitv, pch=16, col="red")
> text(x=-0.05, y=0.9*max(resids), paste("t-value =", tvalue))
```

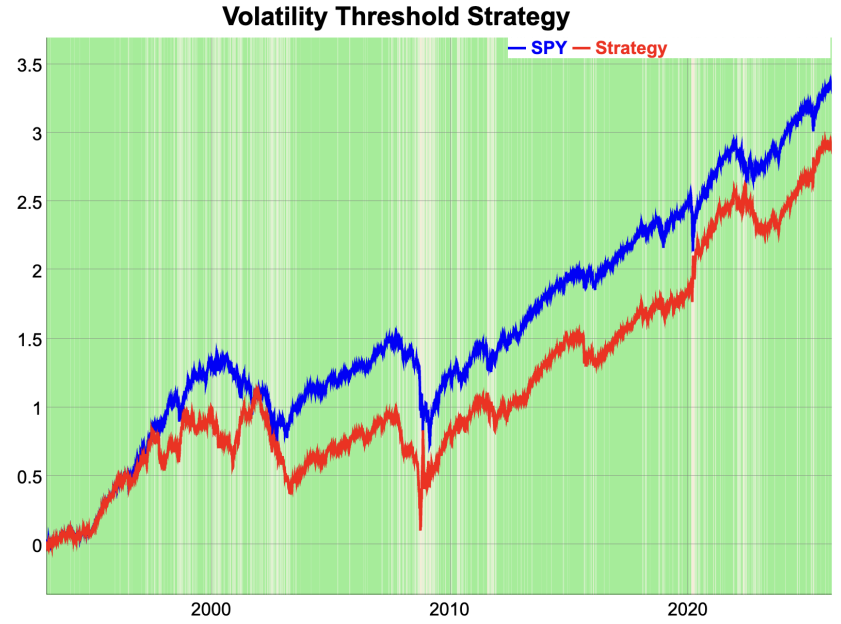
Market Timing Using a Volatility Threshold

The strategy switches between $1 dollar long and -$1 dollar short positions depending on whether the volatility is below or above the threshold.

The strategy performance depends on the value of the lambda decay parameter used to calculate the *EMA* volatility, and on the level of the threshold.

If the lambda decay parameter is too large, the strategy is too slow to react to changes in the volatility (bias), and if the lambda decay parameter is too small, the volatility estimate is too noisy (variance).

If the threshold is too low, the strategy is mostly short the stock, and if the threshold is too high, the strategy is mostly long.




```
> # Calculate the EMA volatility
> volma <- HighFreq::run_var(retp, lambda=0.2)
> volma <- sqrt(volma[, 2])
> # Calculate the positions and PnLs
> threshv <- 3*median(volma) ##  Volatility threshold
> posv <- -rutils::lagit(sign(volma - threshv), lagg=1)
> pnls <- retp*posv
> costv <- 0.5*bidask*abs(rutils::diffit(posv))
> pnls <- (pnls - costv)
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c(symboln, "Strategy")
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
```

```
> # Create colors for background shading
> indeks <- (rutils::diffit(posv) != 0)
> crossdates <- c(datev[indeks], last(datev))
> shadev <- ifelse(posv[indeks] == 1, "lightgreen", "antiquewhite")
> # Plot dygraph of the strategy with background shading
> dyplot <- dygraphs::dygraph(cumsum(wealthv),
+   main="Volatility Threshold Strategy") %>%
+   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], c
+ }  # end for
> # Plot the dygraph object
> dyplot
```

Interest Rate Yield Curve and Stock Returns

The level of interest rates and the shape of the yield curve can be used to forecast stock returns.

The *FRED* database provides historical constant maturity Treasury yields:
https://fred.stlouisfed.org/series/DGS5

```
> # Download time series of bond yields
> # symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20", "DGS30")
> # ratesenv <- new.env()
> # quantmod::getSymbols(symbolv, env=ratesenv, src="FRED")
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RData")
> # Combine rates into single xts series
> ratev <- do.call(cbind, as.list(ratesenv))
> # Sort the columns of rates according bond maturity
> namev <- colnames(ratev)
> namev <- substr(namev, start=4, stop=10)
> namev <- as.numeric(namev)
> indeks <- order(namev)
> ratev <- ratev[, indeks]
> # Align rates dates with VTI prices
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> colnames(closep) <- "VTI"
> datev <- zoo::index(closep)
> ratev <- na.omit(ratev[datev])
> closep <- closep[zoo::index(ratev)]
> datev <- zoo::index(closep)
> nrows <- NROW(closep)
```

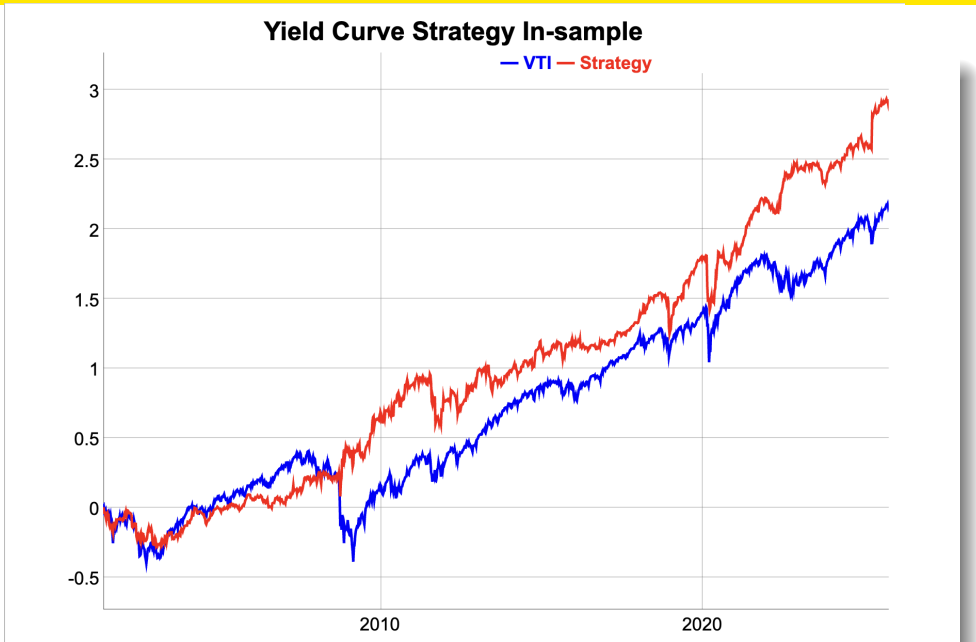
```
> # Calculate VTI returns and IR changes
> retp <- rutils::diffit(closep)
> retr <- rutils::diffit(ratev)
> # Regress VTI returns versus the lagged rate differences
> predm <- rutils::lagit(retr)
> regmod <- lm(retp ~ predm)
> summary(regmod)
> # Regress VTI returns before and after 2010
> summary(lm(retp["/2010"] ~ predm["/2010"]))
> summary(lm(retp["2010/"] ~ predm["2010/"]))
```

Yield Curve Strategy In-Sample

For in-sample forecasts, the training set and the test set are the same. The model is calibrated on the data that is used for forecasting.

Although it's not realistic to achieve the in-sample performance, it's useful because it provides insights into how the model works.

The in-sample strategy performs well in periods of high volatility, but not as well in periods of low volatility.



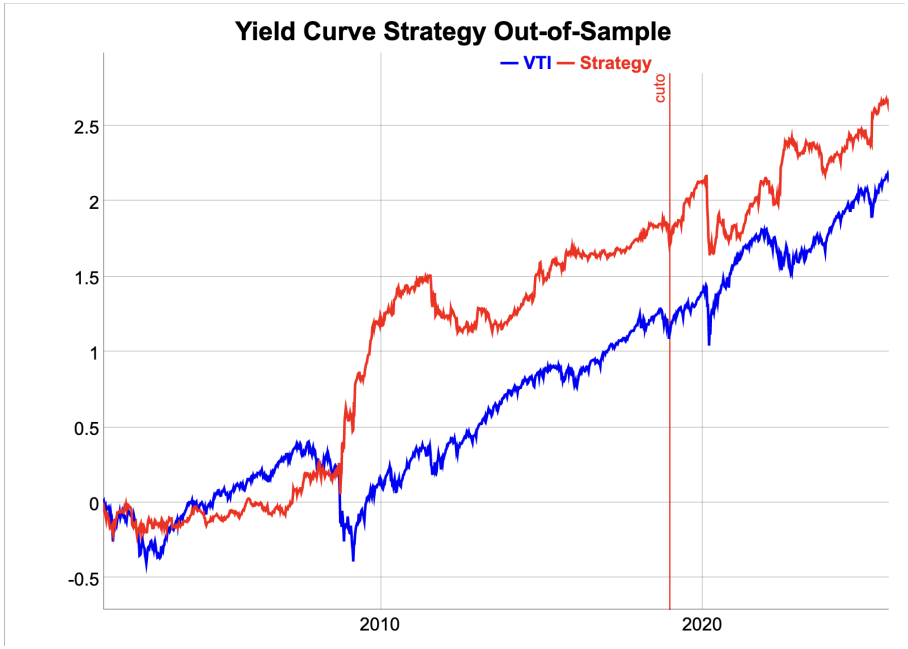

```
> # Define predictor with intercept term
> predm <- rutils::lagit(retr)
> predm <- cbind(rep(1, NROW(predm)), predm)
> colnames(predm)[1] <- "intercept"
> # Calculate inverse of predictor
> invreg <- MASS::ginv(predm)
> # Calculate coefficients from response and inverse of predictor
> respv <- retp
> coeff <- drop(invreg %*% respv)
> # Calculate forecasts and PnLs in-sample
> fcasts <- (predm %*% coeff)
> fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Calculate the in-sample factors
> factv <- lapply(1:NCOL(predm), function(x) predm[, x]*coeff[x])
> factv <- do.call(cbind, factv)
> # Or:
> # foo <- HighFreq::mult_mat(coeff, predm)
> # all.equal(foo, coredata(factv), check.attributes=FALSE)
> # Calculate the factor volatilities
> apply(factv, 2, sd)
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```

```
> # Plot dygraph of in-sample YC strategy
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> cor(wealthv)
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Yield Curve Strategy In-sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Yield Curve Strategy Out-of-Sample

For out-of-sample forecasts, the training set and the test set are separate. The model is calibrated on the training set, and forecasts are calculated using the test set.

```
> # Calculate inverse of predictor in-sample
> invreg <- MASS::ginv(predm["/2018"])
> # Calculate coefficients in-sample
> coeff <- drop(invreg %*% respv["/2018"])
> # Calculate forecasts and PnLs
> fcasts <- (predm %*% coeff)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```

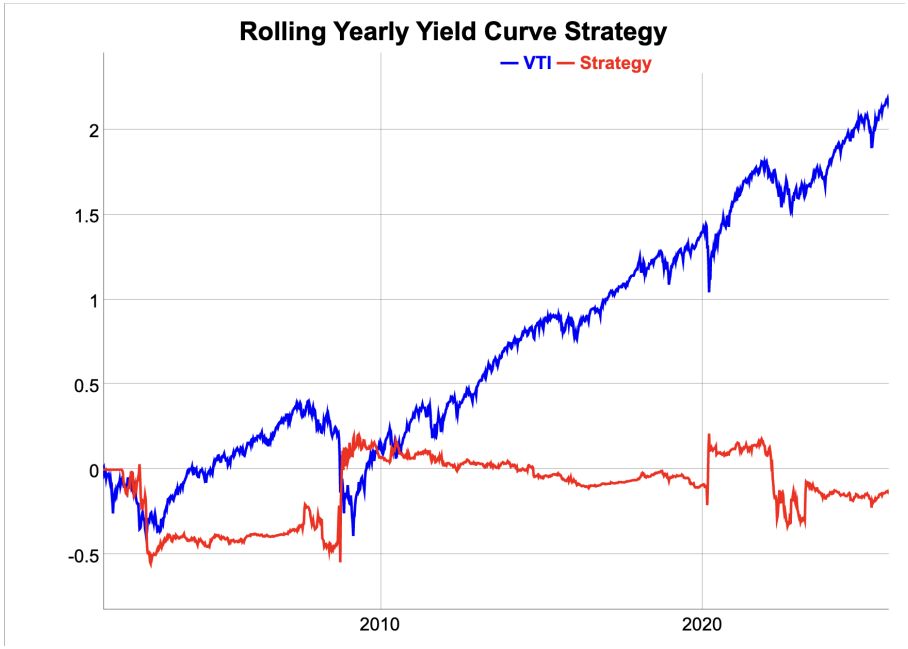



```
> # Plot dygraph of out-of-sample YC strategy
> wealthv <- cbind(retp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Yield Curve Strategy Out-of-Sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyEvent(end(predm["/2018"]), label="cutoff", strokePattern="soli
+   dyLegend(show="always", width=300)
```

Rolling Yearly Yield Curve Strategy

In the rolling yearly yield curve strategy, the model is recalibrated at the end of every year using a training set of data from the past `lookb` days. The coefficients are applied to calculate out-of-sample forecasts in the following year.

```
> # Define yearly dates
> endd <- rutils::calc_endpoints(respv, interval="years")
> # endd <- index(closep)[endd]
> # Perform loop over yearly dates
> library(parallel)  ##  Load package parallel
> ncores <- detectCores() - 1
> lookb <- 500
> fcasts <- mclapply(seq_along(endd)[-1], function(tday) {
+   ##  Define in-sample and out-of-sample intervals
+   insample <- (max(1, endd[tday-1]-lookb):endd[tday-1])
+   ##  insample <- (1:endd[tday-1]) ##  Expanding look-back
+   outsample <- (endd[tday-1]+1):endd[tday]
+   ##  Calculate coefficients in-sample
+   invreg <- MASS::ginv(predm[insample, ])
+   coeff <- drop(invreg %*% respv[insample, ])
+   ##  Calculate forecasts out-of-sample
+   fcasts <- (predm[outsample, ] %*% coeff)
+ }, mc.cores=ncores)  ##  end mclapply
> fcasts <- do.call(rbind, fcasts)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```




```
> # Plot dygraph of rolling yearly YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Rolling Yearly Yield Curve Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

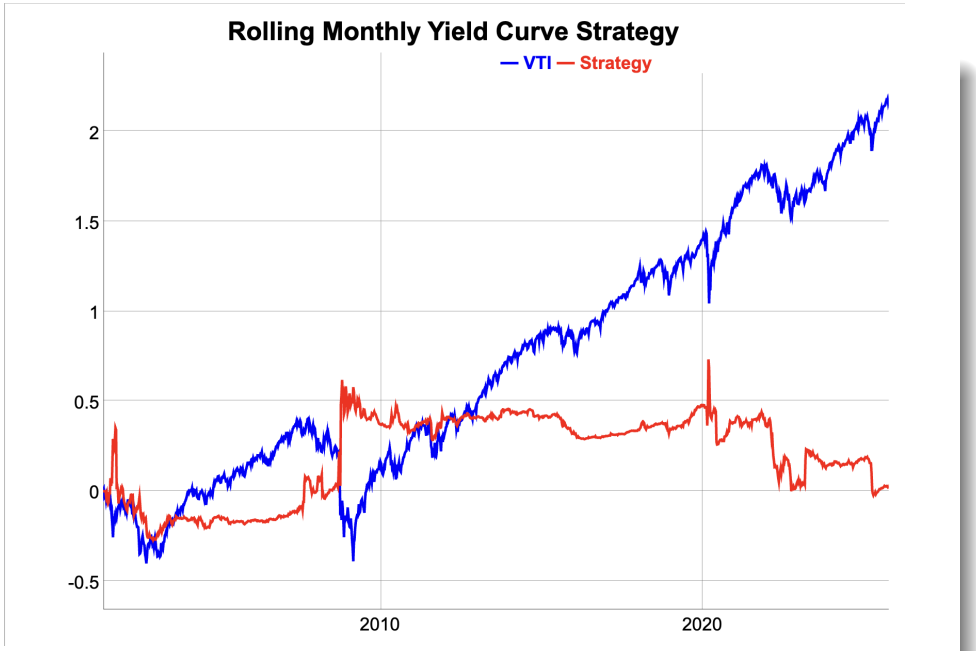
Rolling Monthly Yield Curve Strategy

In the rolling monthly yield curve strategy, the model is recalibrated at the end of every month using a training set of the past `lookb` days. The coefficients are applied to perform out-of-sample forecasts in the following month.

Research shows that looking back roughly a year provides the best out-of-sample forecasts.

The rolling monthly strategy performs better than the yearly strategy, but mostly in periods of high volatility, and otherwise it's flat.

```
> # Define monthly dates
> endd <- rutils::calc_endpoints(respv, interval="month")
> fcasts <- mclapply(seq_along(endd)[-1], function(tday) {
+   ##  Define in-sample and out-of-sample intervals
+   insample <- (max(1, endd[tday-1]-lookb):endd[tday-1])
+   ##  insample <- (1:endd[tday-1]) ##  Expanding look-back
+   outsample <- (endd[tday-1]+1):endd[tday]
+   ##  Calculate coefficients in-sample
+   invreg <- MASS::ginv(predm[insample, ])
+   coeff <- drop(invreg %*% respv[insample, ])
+   ##  Calculate forecasts out-of-sample
+   fcasts <- (predm[outsample, ] %*% coeff)
+ }, mc.cores=ncores)  ##  end mclapply
> fcasts <- do.call(rbind, fcasts)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```



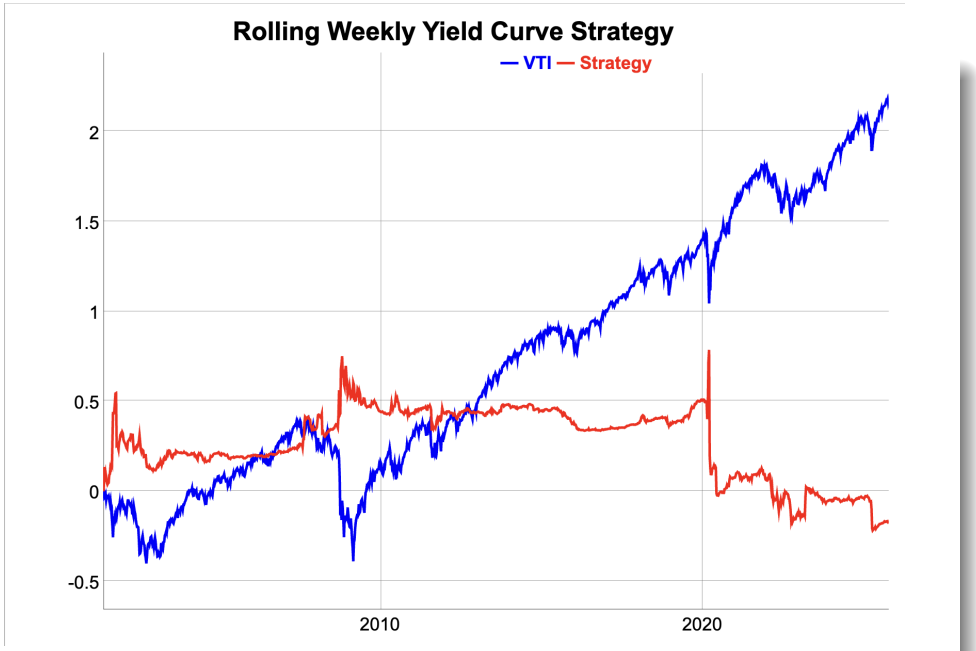

```
> # Plot dygraph of rolling monthly YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Rolling Monthly Yield Curve Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Rolling Weekly Yield Curve Strategy

In the rolling weekly yield curve strategy, the model is recalibrated at the end of every week using a training set of the past `lookb` days. The coefficients are applied to perform out-of-sample forecasts in the following week.

The rolling weekly strategy performs worse than the monthly strategy, because the coefficients have a larger variance and estimation error.

```
> # Define weekly dates
> endd <- rutils::calc_endpoints(respv, interval="weeks")
> fcasts <- mclapply(seq_along(endd)[-1], function(tday) {
+   ##  Define in-sample and out-of-sample intervals
+   insample <- (max(1, endd[tday-1]-lookb):endd[tday-1])
+   ##  insample <- (1:endd[tday-1]) ##  Expanding look-back
+   outsample <- (endd[tday-1]+1):endd[tday]
+   ##  Calculate coefficients in-sample
+   invreg <- MASS::ginv(predm[insample, ])
+   coeff <- drop(invreg %*% respv[insample, ])
+   ##  Calculate forecasts out-of-sample
+   fcasts <- (predm[outsample, ] %*% coeff)
+ }, mc.cores=ncores)  ##  end mclapply
> fcasts <- do.call(rbind, fcasts)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```




```
> # Plot dygraph of rolling weekly YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Rolling Weekly Yield Curve Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

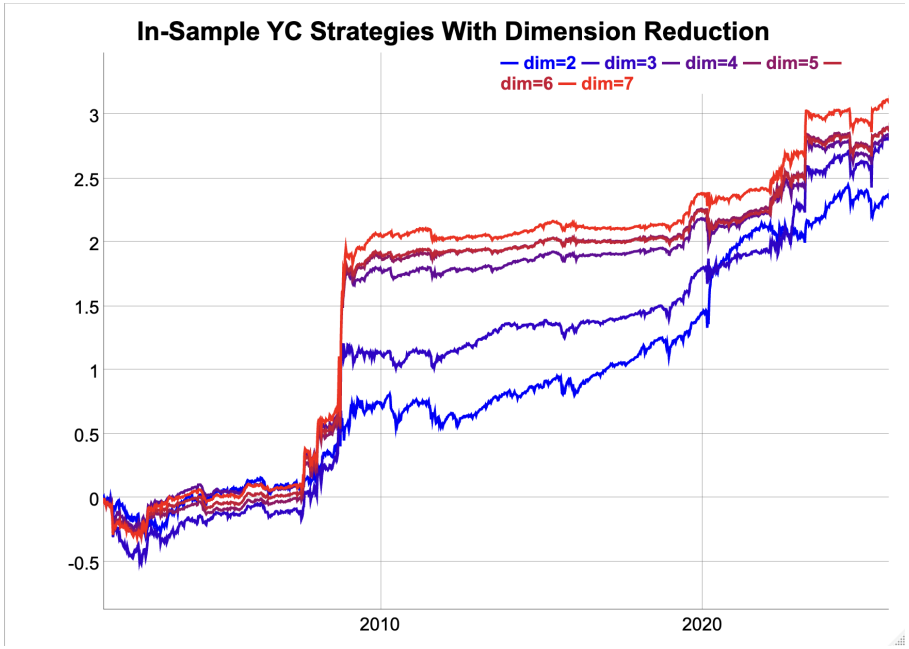
Yield Curve Strategy With Dimension Reduction In-Sample

The technique of *dimension reduction* is used to reduce the number of predictors in a model, for example in portfolio optimization.

Regularization of the inverse predictor matrix improves the in-sample performance of the yield curve strategy.

The best performing in-sample YC strategy are with the intermediate order parameters $dimax = 5, 6$.

Although it's not realistic to achieve the in-sample performance, it's useful because it provides insights into how the model can be improved.




```
> # Calculate in-sample pnls for different dimax values
> dimv <- 2:7
> pnls <- mclapply(dimv, function(dimax) {
+   invred <- HighFreq::calc_invsvd(predm, dimax=dimax)
+   coeff <- drop(invred %*% respv)
+   fcasts <- (predm %*% coeff)
+   # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
+   # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
+   pnls <- fcasts*respv
+   pnls/sd(pnls)
+ }, mc.cores=ncores)  ##  end mclapply
> pnls <- sd(respv)*do.call(cbind, pnls)
> colnames(pnls) <- paste0("dim=", dimv)
```

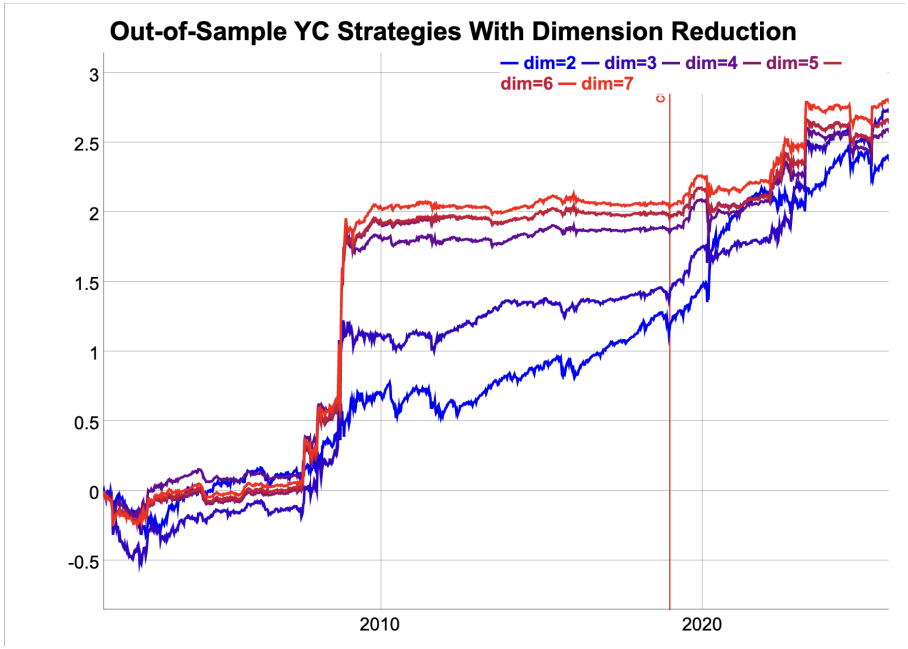
```
> # Plot dygraph of in-sample pnls
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(pnls))
> endw <- rutils::calc_endpoints(pnls, interval="weeks")
> dygraphs::dygraph(cumsum(pnls)[endw],
+   main="In-Sample YC Strategies With Dimension Reduction") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Yield Curve Strategy With Dimension Reduction Out-of-Sample

For out-of-sample forecasts, the training set and the test set are separate. The model is calibrated on the training set, and forecasts are calculated using the test set.

The best performing out-of-sample YC strategy is with the smallest order parameter (strongest dimension reduction) $dimax = 2$.

```
> # Calculate in-sample pnls for different dimax values
> pnls <- mclapply(dimv, function(dimax) {
+   invred <- HighFreq::calc_invsvd(predm["/2018"], dimax=dimax)
+   coeff <- drop(invred %*% respv["/2018"])
+   fcasts <- (predm %*% coeff)
+   # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
+   # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
+   pnls <- fcasts*respv
+   pnls/sd(pnls)
+ }, mc.cores=ncores)  ##  end mclapply
> pnls <- sd(respv)*do.call(cbind, pnls)
> colnames(pnls) <- paste0("dim=", dimv)
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls["2018/"], function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
```



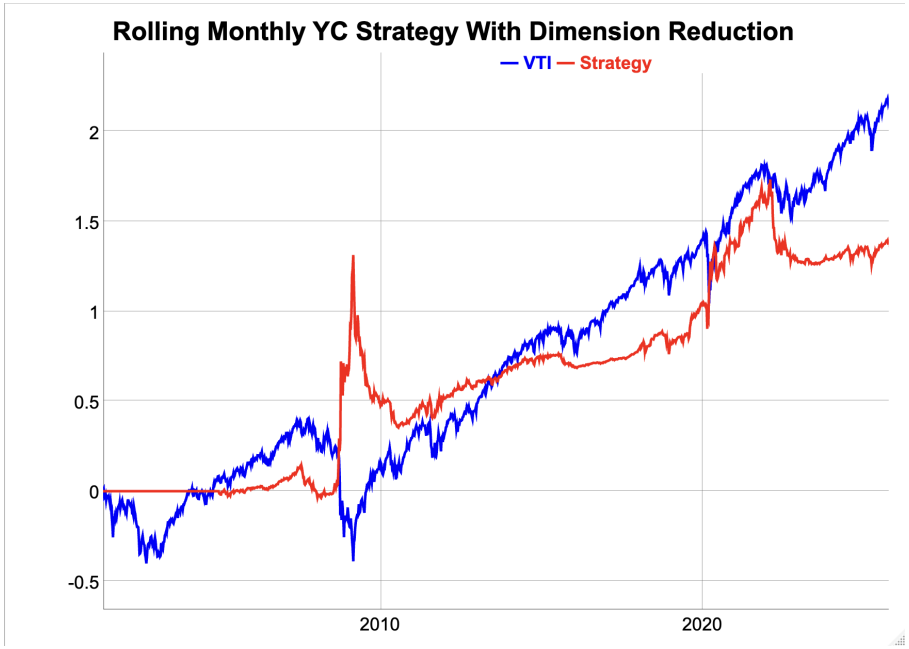

```
> # Plot dygraph of out-of-sample pnls
> colorv <- colorRampPalette(c("blue", "red"))(NCOL(pnls))
> endw <- rutils::calc_endpoints(pnls, interval="weeks")
> dygraphs::dygraph(cumsum(pnls)[endw],
+   main="Out-of-Sample YC Strategies With Dimension Reduction") %>%
+   dyOptions(colors=colorv, strokeWidth=2) %>%
+   dyEvent(end(predm["/2018"]), label="cutoff", strokePattern="soli
+   dyLegend(show="always", width=300)
```

Rolling Monthly Yield Curve Strategy With Dimension Reduction

The rolling monthly strategy with dimension reduction $dimax = 2$ performs better than the standard strategy.

In the rolling monthly yield curve strategy, the model is recalibrated at the end of every month using a training set of the past `lookb` days. The coefficients are applied to perform out-of-sample forecasts in the following month.

```
> # Define monthly dates
> endd <- rutils::calc_endpoints(respv, interval="month")
> enddd <- seq_along(endd)[endd > lookb]
> # Perform loop over monthly dates
> lookb <- 500
> dimax <- 2
> fcasts <- mclapply(enddd, function(tday) {
+   ##  Define in-sample and out-of-sample intervals
+   insample <- (max(1, endd[tday-1]-lookb):endd[tday-1])
+   outsample <- (endd[tday-1]+1):endd[tday]
+   ##  Calculate coefficients in-sample
+   invreg <- HighFreq::calc_invsvd(predm[insample, ], dimax=dimax)
+   coeff <- drop(invreg %*% respv[insample, ])
+   ##  Calculate forecasts out-of-sample
+   fcasts <- (predm[outsample, ] %*% coeff)
+ }, mc.cores=ncores)  ##  end mclapply
> fcasts <- do.call(rbind, fcasts)
> fcasts <- rbind(matrix(rep(0, nrows-NROW(fcasts)), nc=1), fcasts)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```

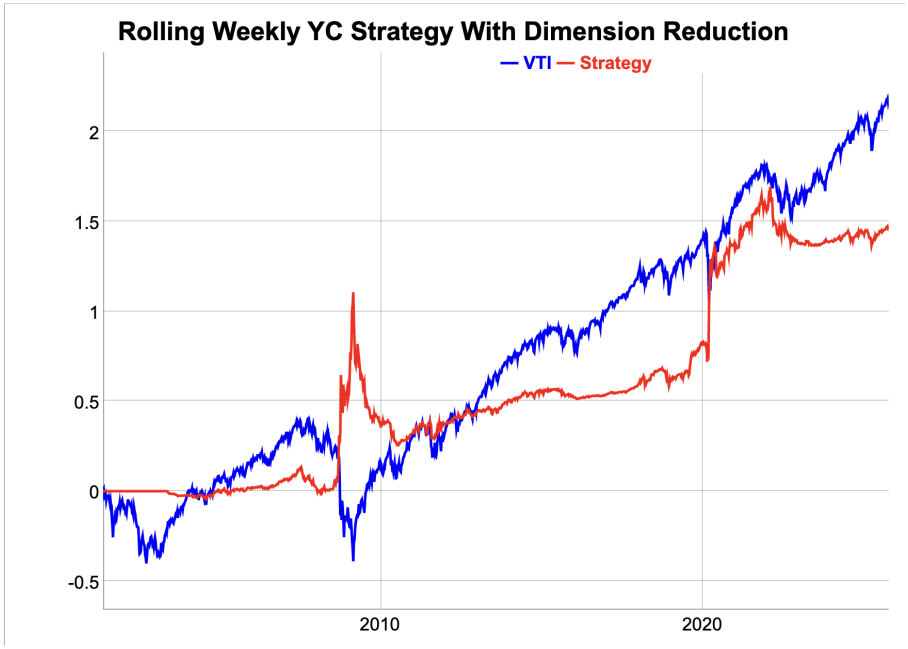



```
> # Plot dygraph of rolling monthly YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Rolling Monthly YC Strategy With Dimension Reduction") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Rolling Weekly Yield Curve Strategy With Regularization

In the rolling weekly yield curve strategy, the model is recalibrated at the end of every week using a training set of the past `lookb` days. The coefficients are applied to perform out-of-sample forecasts in the following week.

```
> # Define weekly dates
> endd <- rutils::calc_endpoints(closep, interval="weeks")
> enddd <- seq_along(endd)[endd > lookb]
> fcasts <- mclapply(enddd, function(tday) {
+   ##  Define in-sample and out-of-sample intervals
+   insample <- (max(1, endd[tday-1]-lookb):endd[tday-1])
+   outsample <- (endd[tday-1]+1):endd[tday]
+   ##  Calculate coefficients in-sample
+   invreg <- HighFreq::calc_invsvd(predm[insample, ], dimax=dimax)
+   coeff <- drop(invreg %*% respv[insample, ])
+   ##  Calculate forecasts out-of-sample
+   fcasts <- (predm[outsample, ] %*% coeff)
+ }, mc.cores=ncores)  ##  end mclapply
> fcasts <- do.call(rbind, fcasts)
> fcasts <- rbind(matrix(rep(0, nrows-NROW(fcasts)), nc=1), fcasts)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv)/sd(pnls)
```



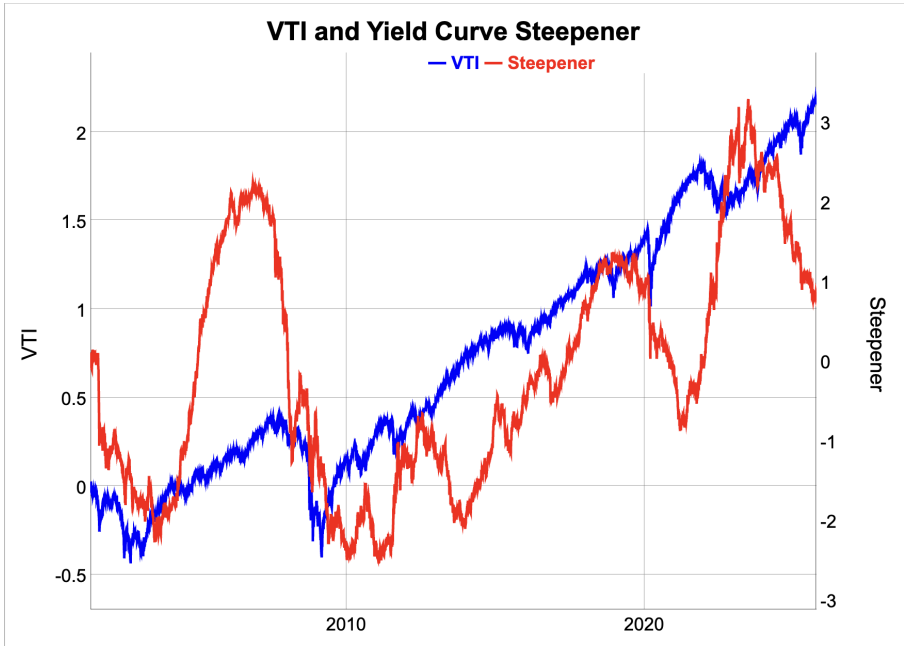

```
> # Plot dygraph of rolling weekly YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="Rolling Weekly YC Strategy With Dimension Reduction") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Yield Curve Principal Components and Stock Returns

The principal components of the interest rate yield curve can also be used as predictors of stock indices.

The second principal component describes the steepening and flattening of the yield curve, and it's an indicator of investor risk appetite. So it's also related to bullish and bearish market periods.

```
> # Calculate PCA of rates from correlation matrix
> eigend <- eigen(cor(retr))
> pcar <- (retr %*% eigend$vectors)
> colnames(pcar) <- paste0("PC", 1:6)
> pcar <- xts::xts(pcar, datev)
> # Define predictor as the YC PCAs
> predm <- rutils::lagit(pcar)
> regmod <- lm(retp ~ predm)
> summary(regmod)
> # After 2010, the PCAs are not good predictors
> regmod <- lm(retp["2010/"] ~ predm["2010/"])
> summary(regmod)
```




```
> # Plot YC steepener principal component with VTI
> datav <- cbind(respv, pcar[, 2])
> colnames(datav) <- c("VTI", "Steepener")
> colv <- colnames(datav)
> dygraphs::dygraph(cumsum(datav),
+   main="VTI and Yield Curve Steepener") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=2, col="blue") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=2, col="red") %>%
+   dyLegend(show="always", width=300)
```

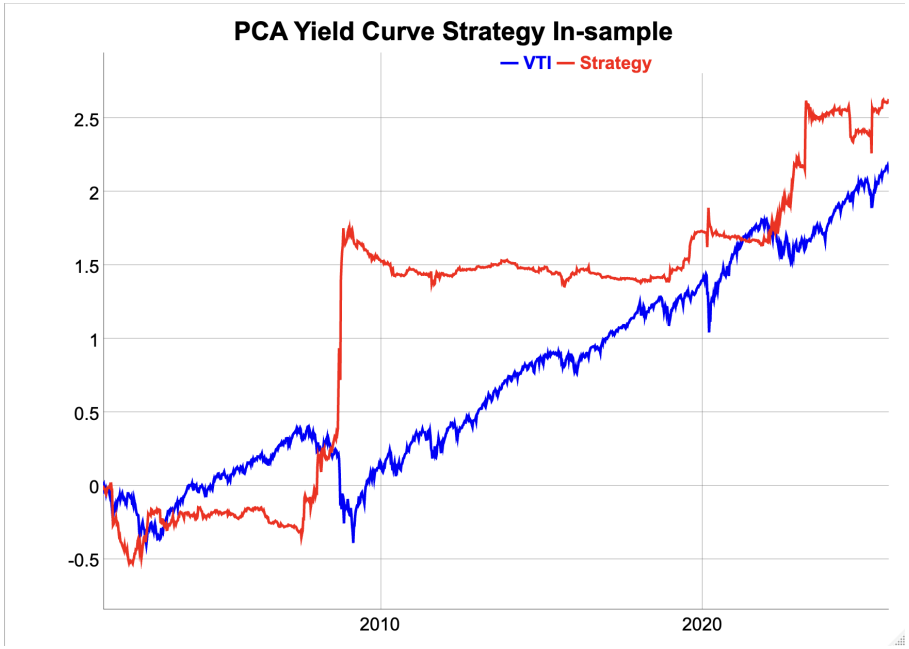
PCA Yield Curve Strategy In-Sample

The best performing PCA yield curve strategy uses only the first two principal components of the yield curve as predictors of stock returns.

For in-sample forecasts, the training set and the test set are the same. The model is calibrated on the data that is used for forecasting.

Although it's not realistic to achieve the in-sample performance, it's useful because it provides insights into how the model works.

The in-sample strategy performs well in periods of high volatility, but otherwise it's flat.




```
> # Define predictor without intercept term
> predm <- rutils::lagit(pcar[, 1:2])
> # Calculate inverse of predictor
> invreg <- MASS::ginv(predm)
> # Calculate coefficients from response and inverse of predictor
> coeff <- drop(invreg %*% respv)
> # Calculate forecasts and PnLs in-sample
> fcasts <- (predm %*% coeff)
> # fcastv <- sqrt(HighFreq::run_var(fcasts, lambda=0.4)[, 2])
> # fcasts <- ifelse(fcastv > mad(fcastv)/20, fcasts/fcastv, 0)
> pnls <- fcasts*respv
> # Scale the PnL volatility to that of VTI
> pnls <- pnls*sd(respv[respv<0])/sd(pnls[pnls<0])
> # Calculate in-sample factors
> factv <- (predm*coeff)
> apply(factv, 2, sd)
```

```
> # Plot dygraph of in-sample YC strategy
> wealthv <- cbind(respv, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv, function(x)
+   c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> colv <- colnames(wealthv)
> endw <- rutils::calc_endpoints(wealthv, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endw],
+   main="PCA Yield Curve Strategy In-sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

Homework Assignment

Required

- Study all the lecture slides in *FRE7241_Lecture_4.pdf*, and run all the code in *FRE7241_Lecture_4.R*

Recommended

- Read about volatility timing and targeting:
  Harvey Portfolio Volatility Targeting.pdf
  Papageorgiou Portfolio Volatility Targeting Strategy Factor Models.pdf
  Fleming Portfolio Volatility Targeting Strategy.pdf