Markets and Trading

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 30, 2022

NYU | TANDON SCHOOL OF ENGINEERING

Downloading Treasury Bond Rates from *FRED*

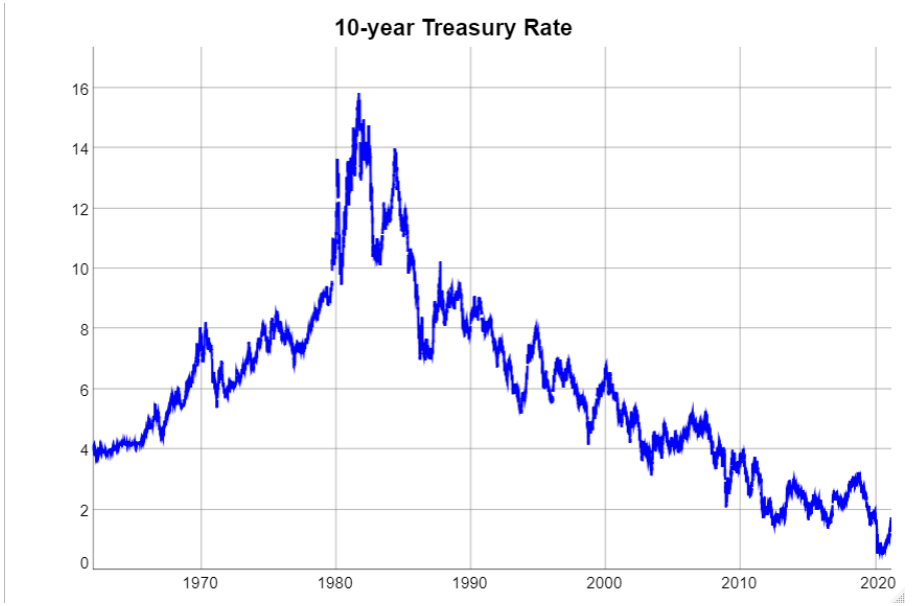
The constant maturity Treasury rates are yields of hypothetical fixed-maturity bonds, interpolated from the market yields of actual Treasury bonds.

The *FRED* database contains current and historical constant maturity Treasury rates,

https://fred.stlouisfed.org/series/DGS5

`quantmod::getSymbols()` creates objects in the specified *environment* from the input strings (names).

It then assigns the data to those objects, without returning them as a function value, as a *side effect*.




```
> # Symbols for constant maturity Treasury rates
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20", "DGS30")
> # Create new environment for time series
> ratesenv <- new.env()
> # Download time series for symbolv into ratesenv
> quantmod::getSymbols(symbolv, env=ratesenv, src="FRED")
> # List files in ratesenv
> ls(ratesenv)
> # Get class of all objects in ratesenv
> sapply(ratesenv, class)
> # Get class of all objects in R workspace
> sapply(ls(), function(name) class(get(name)))
> # Save the time series environment into a binary .RData file
> save(ratesenv, file="/Users/jerzy/Develop/lecture_slides/data/rat
```

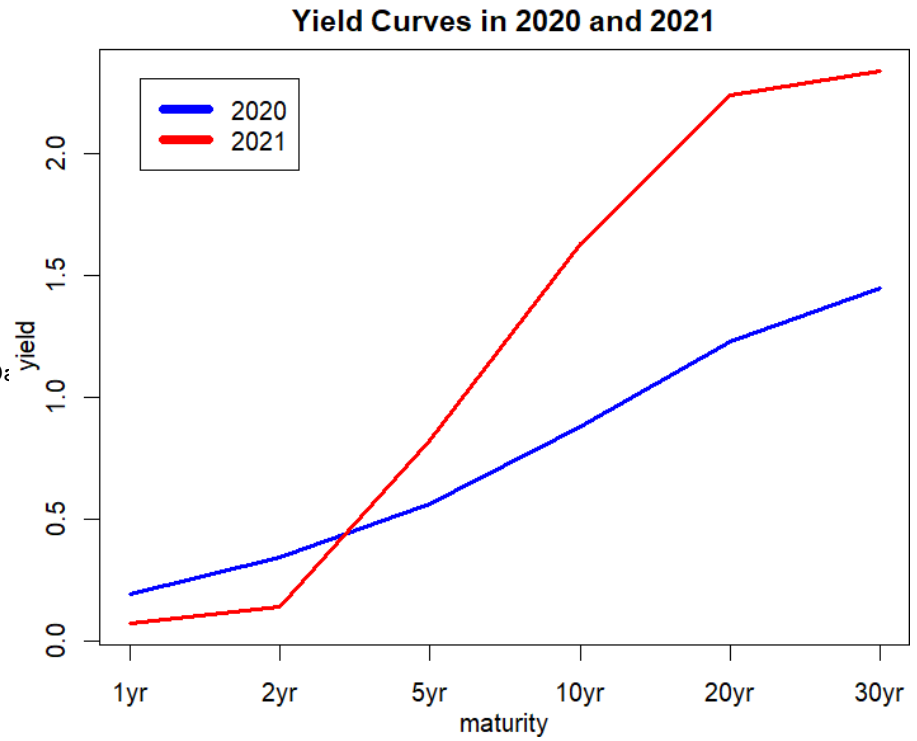
```
> # Get class of time series object DGS10
> class(get(x="DGS10", envir=ratesenv))
> # Another way
> class(ratesenv$DGS10)
> # Get first 6 rows of time series
> head(ratesenv$DGS10)
> # Plot dygraphs of 10-year Treasury rate
> dygraphs::dygraph(ratesenv$DGS10, main="10-year Treasury Rate") %>
+   dyOptions(colors="blue", strokeWidth=2)
> # Plot 10-year constant maturity Treasury rate
> x11(width=6, height=5)
> par(mar=c(2, 2, 0, 0), oma=c(0, 0, 0, 0))
> chart_Series(ratesenv$DGS10["1990/"], name="10-year Treasury Rate"
```

Treasury Yield Curve

The *yield curve* is a vector of interest rates at different maturities, on a given date.

The *yield curve* shape changes depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get most recent yield curve
> ycnow <- eapply(ratesenv, xts::last)
> class(ycnow)
> ycnow <- do.call(cbind, ycnow)
> # Check if 2020-03-25 is not a holiday
> date2020 <- as.Date("2020-03-25")
> weekdays(date2020)
> # Get yield curve from 2020-03-25
> yc2020 <- eapply(ratesenv, function(x) x[date2020])
> yc2020 <- do.call(cbind, yc2020)
> # Combine the yield curves
> rates <- c(yc2020, ycnow)
> # Rename columns and rows, sort columns, and transpose into matr
> colnames(rates) <- substr(colnames(rates), start=4, stop=11)
> rates <- rates[, order(as.numeric(colnames(rates)))]
> colnames(rates) <- paste0(colnames(rates), "yr")
> rates <- t(rates)
> colnames(rates) <- substr(colnames(rates), start=1, stop=4)
```

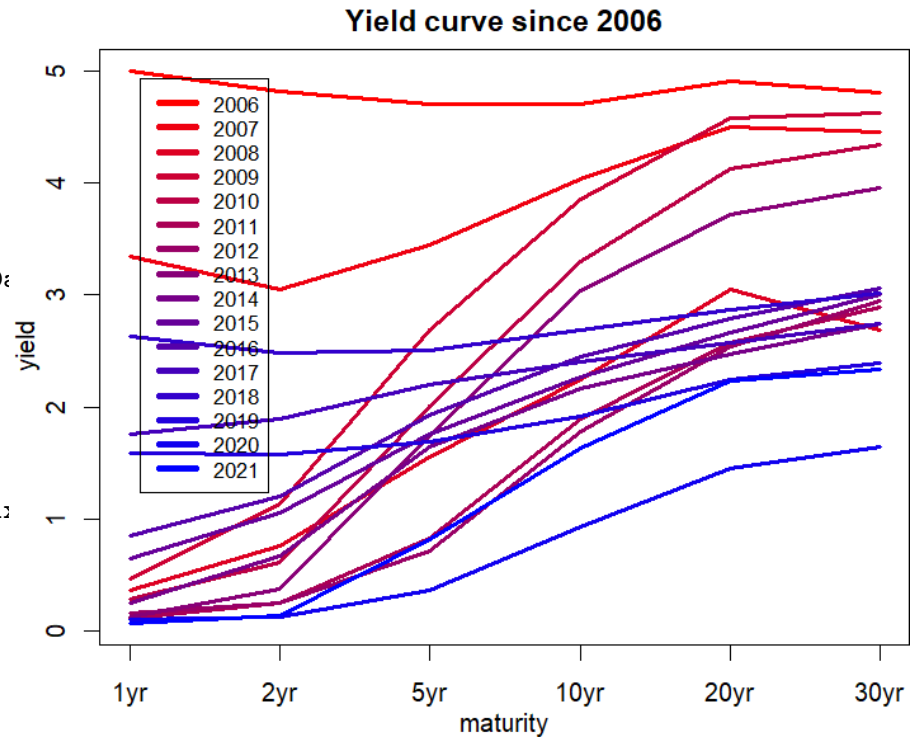



```
> # Plot using matplot()
> colorv <- c("blue", "red")
> matplot(rates, main="Yield Curves in 2020 and 2021", xaxt="n", lwd
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates), y.intersp=0.5,
+  bty="n", col=colorv, lty=1, lwd=6, inset=0.05, cex=1.0)
```

Treasury Yield Curve Over Time

The *yield curve* has changed shape dramatically depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get end-of-year dates since 2006
> dates <- xts::endpoints(ratesenv$DGS1["2006/"], on="years")
> dates <- zoo::index(ratesenv$DGS1["2006/"][dates])
> # Create time series of end-of-year rates
> rates <- eapply(ratesenv, function(ratev) ratev[dates])
> rates <- rutils::do_call(cbind, rates)
> # Rename columns and rows, sort columns, and transpose into matri
> colnames(rates) <- substr(colnames(rates), start=4, stop=11)
> rates <- rates[, order(as.numeric(colnames(rates)))]
> colnames(rates) <- paste0(colnames(rates), "yr")
> rates <- t(rates)
> colnames(rates) <- substr(colnames(rates), start=1, stop=4)
> # Plot matrix using plot.zoo()
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(rates))
> plot.zoo(rates, main="Yield curve since 2006", lwd=3, xaxt="n",
+    plot.type="single", xlab="maturity", ylab="yield", col=colorv
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates), y.intersp=0.5,
+  bty="n", col=colorv, lty=1, lwd=4, inset=0.05, cex=0.8)
```



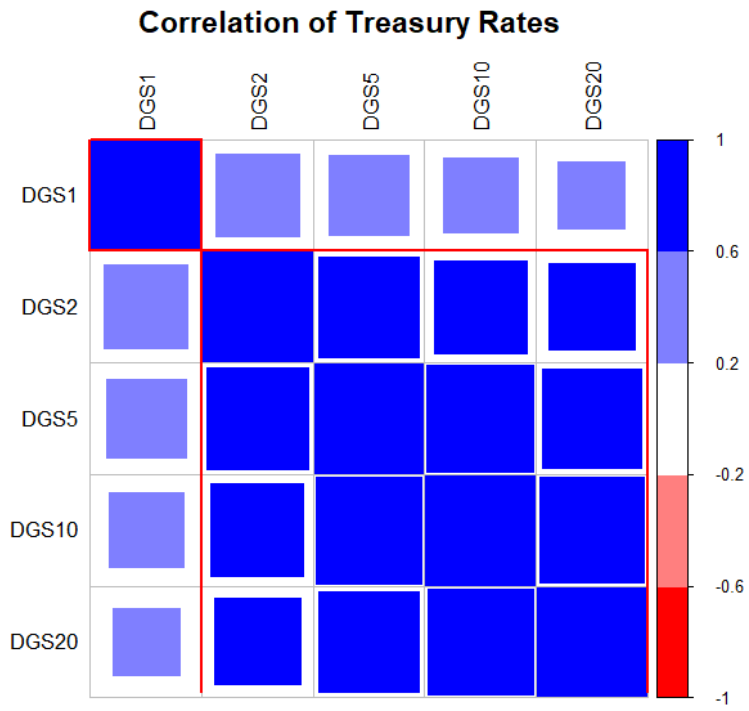

```
> # Alternative plot using matplot()
> matplot(rates, main="Yield curve since 2006", xaxt="n", lwd=3, lty
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates), y.intersp=0.5,
+  bty="n", col=colorv, lty=1, lwd=4, inset=0.05, cex=0.8)
```

Covariance Matrix of Interest Rates

The covariance matrix $\mathbb{C}$, of the interest rate matrix $\mathbf{r}$ is given by:

$$\mathbb{C} = \frac{(\mathbf{r} - \bar{\mathbf{r}})^T (\mathbf{r} - \bar{\mathbf{r}})}{n - 1}$$

```
> # Extract rates from ratesenv
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20")
> rates <- mget(symbolv, envir=ratesenv)
> rates <- rutils::do_call(cbind, rates)
> rates <- zoo::na.locf(rates, na.rm=FALSE)
> rates <- zoo::na.locf(rates, fromLast=TRUE)
> # Calculate daily percentage rates changes
> retsp <- rutils::diffit(log(rates))
> # De-mean the returns
> retsp <- lapply(retsp, function(x) {x - mean(x)})
> retsp <- rutils::do_call(cbind, retsp)
> sapply(retsp, mean)
> # Covariance and Correlation matrices of Treasury rates
> covmat <- cov(retsp)
> cormat <- cor(retsp)
> # Reorder correlation matrix based on clusters
> library(corrplot)
> ordern <- corrMatOrder(cormat, order="hclust",
+   hclust.method="complete")
> cormat <- cormat[ordern, ordern]
```




```
> # Plot the correlation matrix
> x11(width=6, height=6)
> colorv <- colorRampPalette(c("red", "white", "blue"))
> corrplot(cormat, title=NA, tl.col="black",
+     method="square", col=colorv(NCOL(cormat)), tl.cex=0.8,
+     cl.offset=0.75, cl.cex=0.7, cl.align.text="l", cl.ratio=0.25)
> title("Correlation of Treasury Rates", line=1)
> # Draw rectangles on the correlation matrix plot
> corrRect.hclust(cormat, k=NROW(cormat) %/% 2,
+   method="complete", col="red")
```

Principal Component Vectors

Principal components are linear combinations of the k return vectors $\mathbf{r}_i$:

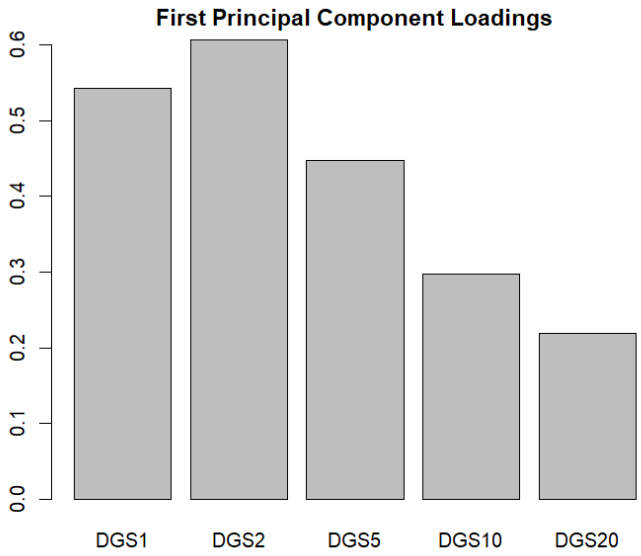
$$\mathbf{pc}_j = \sum_{i=1}^{k} w_{ij}\, \mathbf{r}_i$$

Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* j, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

The weights $\mathbf{w}_j$ are chosen to maximize the variance of the *principal components*, under the condition that they are orthogonal:

$$\mathbf{w}_j = \arg\max \left\{ \mathbf{pc}_j^T\, \mathbf{pc}_j \right\}$$

$$\mathbf{pc}_i^T\, \mathbf{pc}_j = 0 \; (i \neq j)$$




```
> # Create initial vector of portfolio weights
> nweights <- NROW(symbolv)
> weights <- rep(1/sqrt(nweights), nweights)
> names(weights) <- symbolv
> # Objective function equal to minus portfolio variance
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -1e7*var(retsp) + 1e7*(1 - sum(weights*weights))^2
+ }  # end objfun
> # Objective function for equal weight portfolio
> objfun(weights, retsp)
> # Compare speed of vector multiplication methods
> library(microbenchmark)
> summary(microbenchmark(
+   transp=t(retsp) %*% retsp,
+   sumv=sum(retsp*retsp),
```

```
> # Find weights with maximum variance
> optim1 <- optim(par=weights,
+   fn=objfun,
+   retsp=retsp,
+   method="L-BFGS-B",
+   upper=rep(5.0, nweights),
+   lower=rep(-5.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optim1$par
> objfun(weights1, retsp)
> # Plot first principal component loadings
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> barplot(weights1, names.arg=names(weights1),
+   xlab="", ylab="", main="First Principal Component Loadings")
```

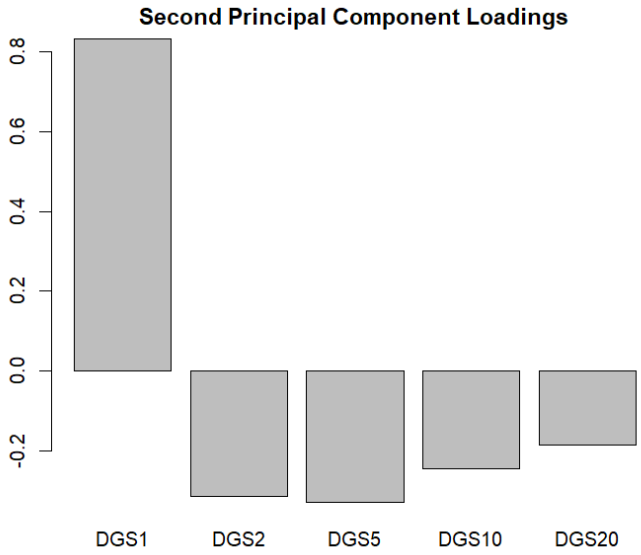
Higher Order Principal Components

The *second principal component* can be calculated by maximizing its variance, under the constraint that it must be orthogonal to the *first principal component*.

Similarly, higher order *principal components* can be calculated by maximizing their variances, under the constraint that they must be orthogonal to all the previous *principal components*.

The number of principal components is equal to the dimension of the covariance matrix.

```
> # pc1 weights and returns
> pc1 <- drop(retsp %*% weights1)
> # Redefine objective function
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -1e7*var(retsp) + 1e7*(1 - sum(weights^2))^2 +
+     1e7*sum(weights1*weights)^2
+ }  # end objfun
> # Find second principal component weights
> optiml <- optim(par=weights,
+              fn=objfun,
+              retsp=retsp,
+              method="L-BFGS-B",
+              upper=rep(5.0, nweights),
+              lower=rep(-5.0, nweights))
```




```
> # pc2 weights and returns
> weights2 <- optiml$par
> pc2 <- drop(retsp %*% weights2)
> sum(pc1*pc2)
> # Plot second principal component loadings
> barplot(weights2, names.arg=names(weights2),
+   xlab="", ylab="", main="Second Principal Component Loadings")
```

Eigenvalues of the Covariance Matrix

The portfolio variance: $\mathbf{w}^T \mathbb{C} \mathbf{w}$ can be maximized under the *quadratic* weights constraint $\mathbf{w}^T \mathbf{w} = 1$, by maximizing the *Lagrangian* $\mathcal{L}$:

$$\mathcal{L} = \mathbf{w}^T \mathbb{C} \mathbf{w} - \lambda (\mathbf{w}^T \mathbf{w} - 1)$$

Where λ is a *Lagrange multiplier*.

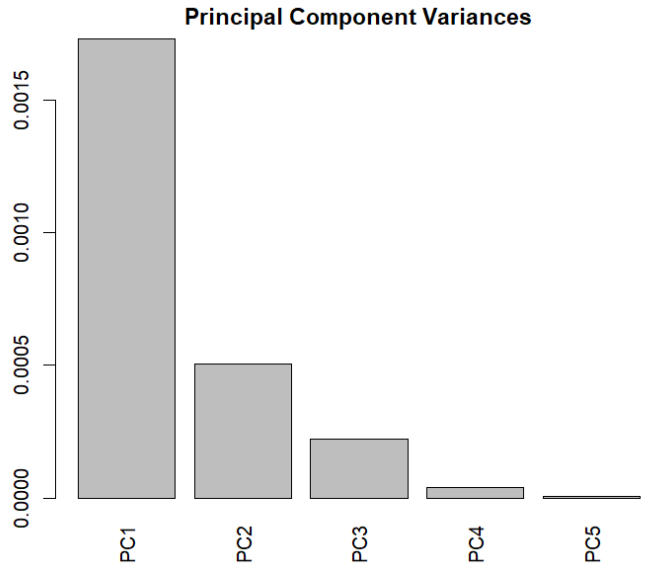
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

$$\mathbb{C} \mathbf{w} = \lambda \mathbf{w}$$

The above is the *eigenvalue* equation of the covariance matrix $\mathbb{C}$, with the optimal weights $\mathbf{w}$ forming an *eigenvector*, and λ is the *eigenvalue* corresponding to the *eigenvector* $\mathbf{w}$.

The *eigenvalues* are the variances of the *eigenvectors*, and their sum is equal to the sum of the return variances:

$$\sum_{i=1}^{k} \lambda_i = \frac{1}{1-k} \sum_{i=1}^{k} \mathbf{r}_i^T \mathbf{r}_i$$




```
> eigend <- eigen(covmat)
> eigend$vectors
> # Compare with optimization
> all.equal(sum(diag(covmat)), sum(eigend$values))
> all.equal(abs(eigend$vectors[, 1]), abs(weights1), check.attribute
> all.equal(abs(eigend$vectors[, 2]), abs(weights2), check.attribute
> all.equal(eigend$values[1], var(pc1), check.attributes=FALSE)
> all.equal(eigend$values[2], var(pc2), check.attributes=FALSE)
> # Eigenvalue equations are satisfied approximately
> (covmat %*% weights1) / weights1 / var(pc1)
> (covmat %*% weights2) / weights2 / var(pc2)
> # Plot eigenvalues
> barplot(eigend$values, names.arg=paste0("PC", 1:nweights),
+   las=3, xlab="", ylab="", main="Principal Component Variances")
```

Principal Component Analysis Versus *Eigen Decomposition*

Principal Component Analysis (*PCA*) is equivalent to the *eigen decomposition* of either the correlation or the covariance matrix.

If the input time series *are* scaled, then *PCA* is equivalent to the eigen decomposition of the *correlation matrix*.

If the input time series *are not* scaled, then *PCA* is equivalent to the eigen decomposition of the *covariance matrix*.

Scaling the input time series improves the accuracy of the *PCA dimension reduction*, allowing a smaller number of *principal components* to more accurately capture the data contained in the input time series.

The function `prcomp()` performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class `prcomp`.

The `prcomp()` argument `scale=TRUE` specifies that the input time series should be scaled by their standard deviations.

```
> # Eigen decomposition of correlation matrix
> eigend <- eigen(cormat)
> # Perform PCA with scaling
> pcad <- prcomp(retsp, scale=TRUE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
> # Eigen decomposition of covariance matrix
> eigend <- eigen(covmat)
> # Perform PCA without scaling
> pcad <- prcomp(retsp, scale=FALSE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
```

Principal Component Analysis of the Yield Curve

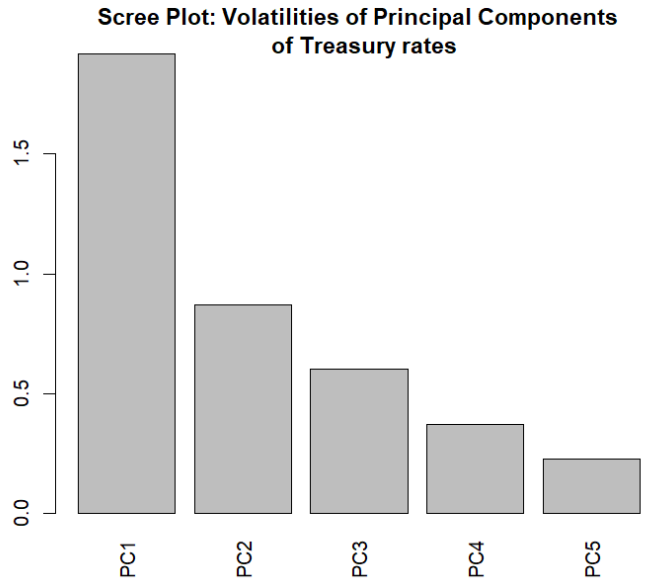
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

The input time series are often scaled by their standard deviations, to improve the accuracy of *PCA dimension reduction*, so that more information is retained by the first few *principal component* time series.

If the input time series are not scaled, then *PCA* analysis is equvalent to the *eigen decomposition* of the covariance matrix, and if they are scaled, then *PCA* analysis is equvalent to the *eigen decomposition* of the correlation matrix.

The function `prcomp()` performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class `prcomp`.

The `prcomp()` argument `scale=TRUE` specifies that the input time series should be scaled by their standard deviations.



A *scree plot* is a bar plot of the volatilities of the *principal components*.

```
> # Perform principal component analysis PCA
> pcad <- prcomp(retsp, scale=TRUE)
> # Plot standard deviations
> barplot(pcad$sdev, names.arg=colnames(pcad$rotation),
+   las=3, xlab="", ylab="",
+   main="Scree Plot: Volatilities of Principal Components
+   of Treasury rates")
```

Yield Curve Principal Component Loadings (Weights)

Principal component loadings are the weights of portfolios which have mutually orthogonal returns.

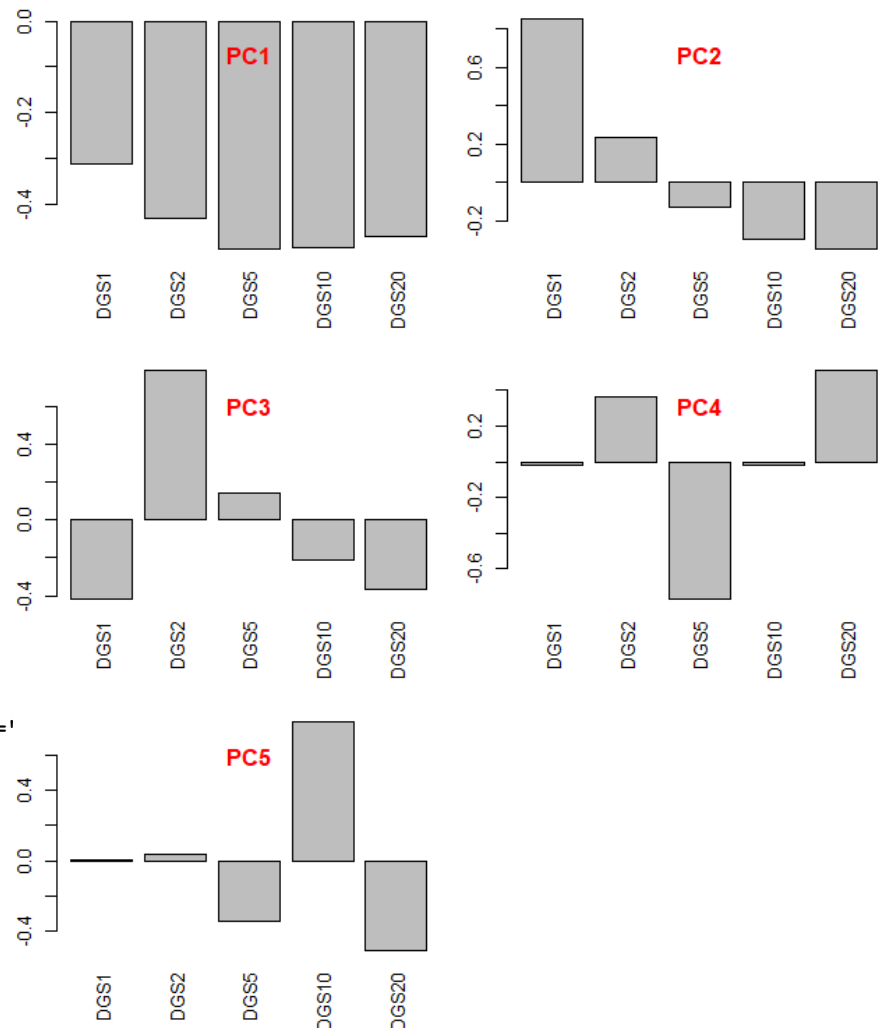
The *principal component* portfolios represent the different orthogonal modes of the data variance.

The first *principal component* of the *yield curve* is the correlated movement of all rates up and down.

The second *principal component* is *yield curve* steepening and flattening.

The third *principal component* is the *yield curve* butterfly movement.

```
> # Calculate principal component loadings (weights)
> pcad$rotation
> # Plot loading barplots in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(3.5, 2, 2, 1), oma=c(0, 0, 0, 0))
> for (ordern in 1:NCOL(pcad$rotation)) {
+   barplot(pcad$rotation[, ordern], las=3, xlab="", ylab="", main='
+   title(paste0("PC", ordern), line=-2.0, col.main="red")
+ }  # end for
```



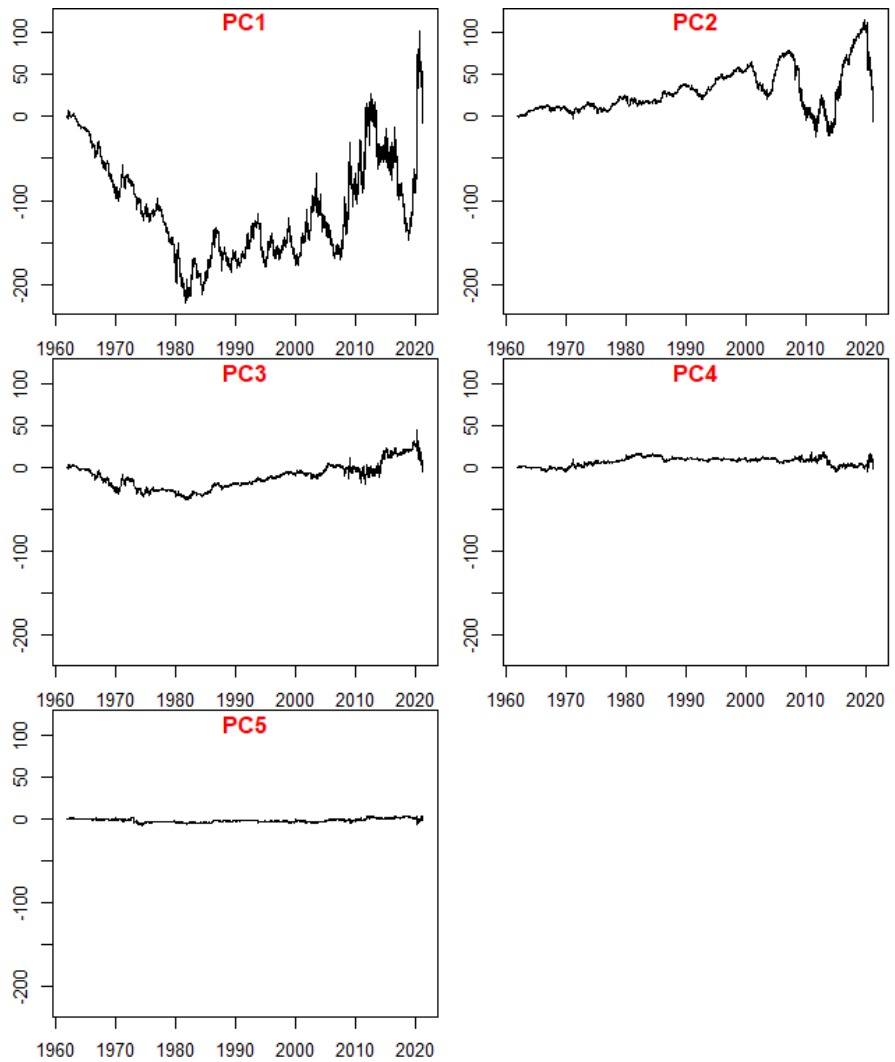
Yield Curve Principal Component Time Series

The time series of the *principal components* can be calculated by multiplying the loadings (weights) times the original data.

The *principal component* time series have mutually orthogonal returns.

Higher order *principal components* are gradually less volatile.

```
> # Standardize (de-mean and scale) the returns
> retsp <- lapply(retsp, function(x) {(x - mean(x))/sd(x)})
> retsp <- rutils::do_call(cbind, retsp)
> sapply(retsp, mean)
> sapply(retsp, sd)
> # Calculate principal component time series
> pcacum <- retsp %*% pcad$rotation
> all.equal(pcad$x, pcacum, check.attributes=FALSE)
> # Calculate products of principal component time series
> round(t(pcacum) %*% pcacum, 2)
> # Coerce to xts time series
> pcacum <- xts(pcacum, order.by=zoo::index(retsp))
> pcacum <- cumsum(pcacum)
> # Plot principal component time series in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> rangev <- range(pcacum)
> for (ordern in 1:NCOL(pcacum)) {
+   plot.zoo(pcacum[, ordern], ylim=rangev, xlab="", ylab="")
+   title(paste0("PC", ordern), line=-1, col.main="red")
+ }  # end for
```



Inverting Principal Component Analysis

The original time series can be calculated *exactly* from the time series of all the *principal components*, by inverting the loadings matrix.

The function `solve()` solves systems of linear equations, and also inverts square matrices.

```
> # Invert all the principal component time series
> retspca <- retsp %*% pcad$rotation
> solved <- retspca %*% solve(pcad$rotation)
> all.equal(coredata(retsp), solved)
```

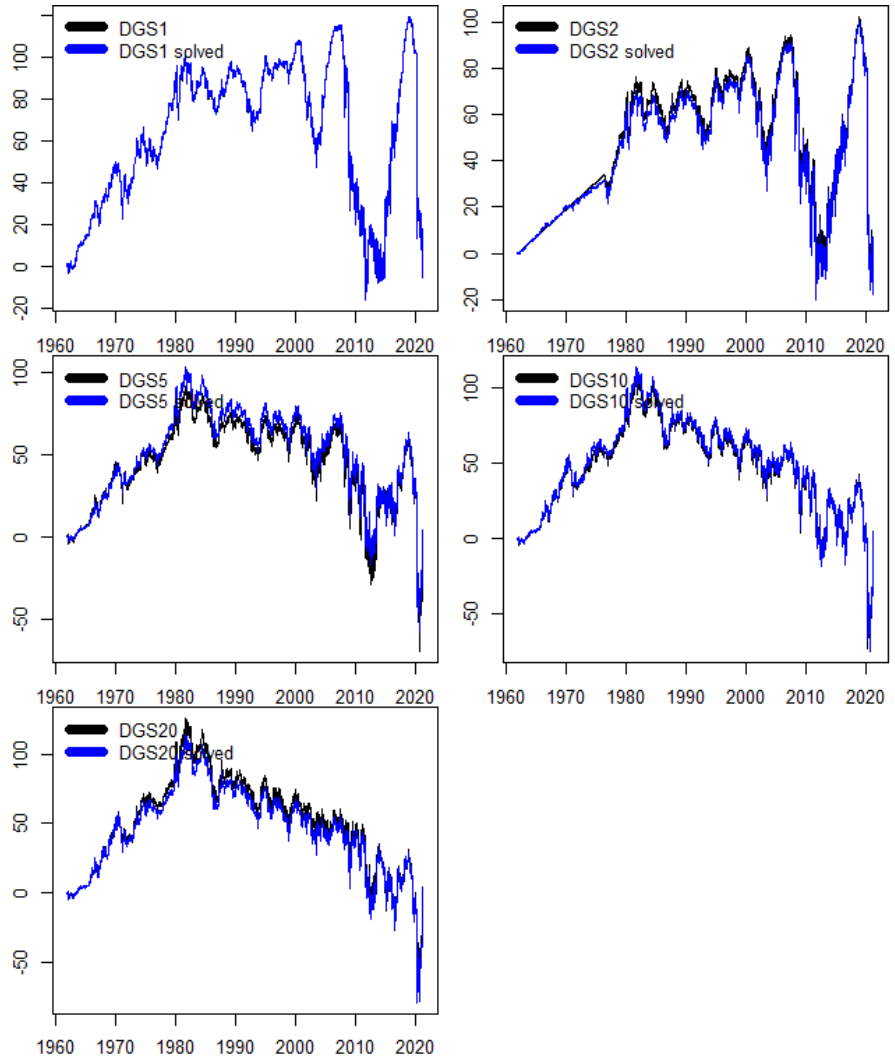
Dimension Reduction Using Principal Component Analysis

The original time series can be calculated *approximately* from just the first few *principal components*, which demonstrates that *PCA* is a form of *dimension reduction*.

A popular rule of thumb is to use the *principal components* with the largest variances, which sum up to 80% of the total variance of returns.

The *Kaiser-Guttman* rule uses only *principal components* with variance greater than 1.

```
> # Invert first 3 principal component time series
> solved <- retspca[, 1:3] %*% solve(pcad$rotation)[1:3, ]
> solved <- xts::xts(solved, zoo::index(retsp))
> solved <- cumsum(solved)
> retc <- cumsum(retsp)
> # Plot the solved returns
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> for (symbol in symbolv) {
+   plot.zoo(cbind(retc[, symbol], solved[, symbol]),
+     plot.type="single", col=c("black", "blue"), xlab="", ylab="")
+   legend(x="topleft", bty="n", y.intersp=0.5,
+    legend=paste0(symbol, c("", " solved")),
+    title=NULL, inset=0.0, cex=1.0, lwd=6,
+    lty=1, col=c("black", "blue"))
+ }  # end for
```



Calibrating Yield Curve Using Package *RQuantLib*

The package *RQuantLib* is an interface to the *QuantLib* open source C/C++ library for quantitative finance, mostly designed for pricing fixed-income instruments and options.

The function `DiscountCurve()` calibrates a *zero coupon yield curve* from *money market* rates, *Eurodollar* futures, and *swap* rates.

The function `DiscountCurve()` interpolates the *zero coupon* rates into a vector of dates specified by the `times` argument.

```
> library(RQuantLib)  # Load RQuantLib
> # Specify curve parameters
> curve_params <- list(tradeDate=as.Date("2018-01-17"),
+                 settleDate=as.Date("2018-01-19"),
+                 dt=0.25,
+                 interpWhat="discount",
+                 interpHow="loglinear")
> # Specify market data: prices of FI instruments
> market_data <- list(d3m=0.0363,
+                fut1=96.2875,
+                fut2=96.7875,
+                fut3=96.9875,
+                fut4=96.6875,
+                s5y=0.0443,
+                s10y=0.05165,
+                s15y=0.055175)
> # Specify dates for calculating the zero rates
> disc_dates <- seq(0, 10, 0.25)
> # Specify the evaluation (as of) date
> setEvaluationDate(as.Date("2018-01-17"))
> # Calculate the zero rates
> disc_curves <- DiscountCurve(params=curve_params,
+                         tsQuotes=market_data,
+                         times=disc_dates)
> # Plot the zero rates
> x11()
> plot(x=disc_curves$zerorates, t="l", main="zerorates")
```

Financial and Commodity Futures Contracts

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

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

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

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

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

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

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

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

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

Interactive Brokers provides more information about futures contracts:

IB Contract and Symbol Database

IB Traded Products

List of Popular Futures Contracts.

E-mini Futures Contracts

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

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

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

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

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

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

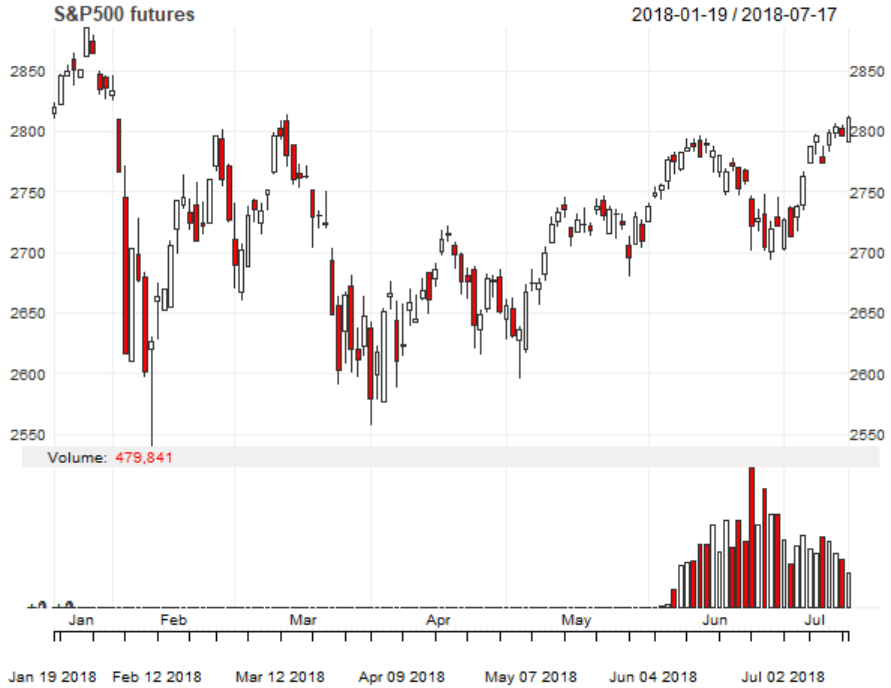
Futures contract	Standard	E-mini
S&P500 index	SP	ES
10yr Treasury	ZN	ZN
VIX index	VX	delisted
Gold	GC	YG
Oil	CL	QM
Euro FX	EC	E7
Swiss franc	SF	MSF
Japanese Yen	JY	J7

Plotting *S&P500* Futures Data

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

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

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



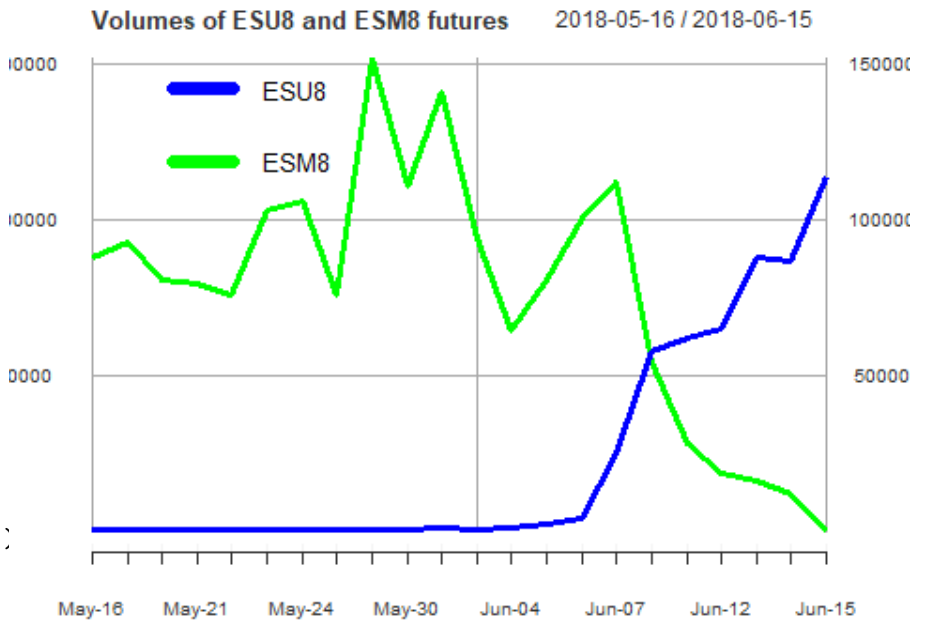

```
> # Plot OHLC data in x11 window
> x11(width=5, height=4)  # Open x11 for plotting
> par(mar=c(5, 5, 2, 1), oma=c(0, 0, 0, 0))
> chart_Series(x=pricev, TA="add_Vo()",
+   name="S&P500 futures")
> # Plot dygraph
> dygraphs::dygraph(pricev[, 1:4], main="OHLC prices") %>%
+   dyCandlestick()
```

Consecutive Contract Futures Volumes

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

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

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




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

Chaining Together Futures Prices

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

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

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

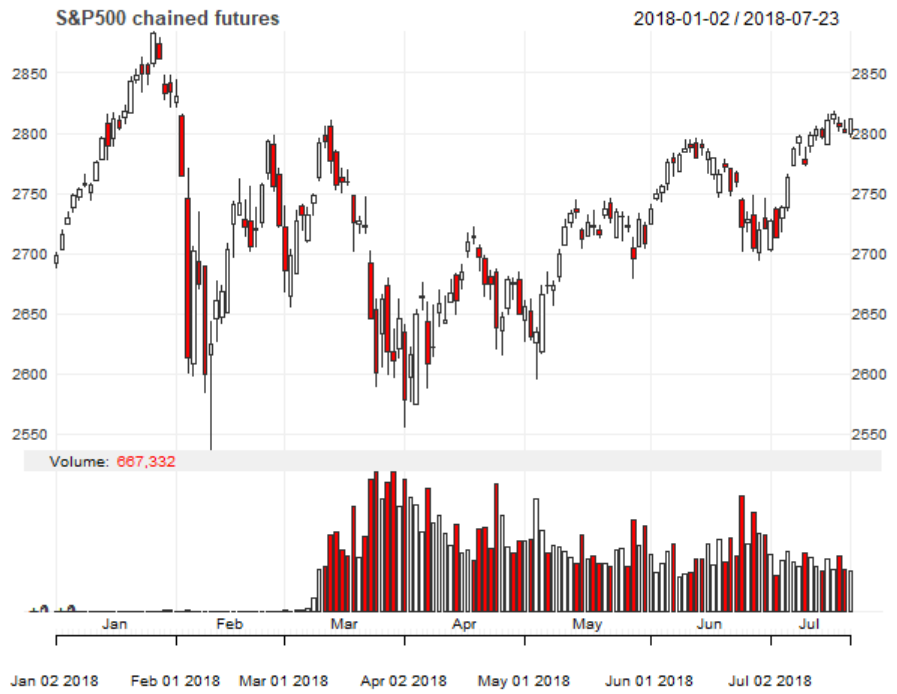
When the next contract becomes more liquid, then the continuous contract price is rolled over to that contract.

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

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

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

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




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

VIX Volatility Index

The *VIX* Volatility Index is an average of the implied volatilities of options on the *S&P500* Index (SPX).

The *VIX* index is an estimate of the *future* stock market volatility that is expected (anticipated) by investors.

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

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

```
> # Download VIX index data from CBOE
> vix_index <- data.table::fread("http://www.cboe.com/publish/schedu
> class(vix_index)
> dim(vix_index)
> tail(vix_index)
> sapply(vix_index, class)
> vix_index <- xts(vix_index[, -1],
+   order.by=as.Date(vix_index$Date, format="%m/%d/%Y"))
> colnames(vix_index) <- c("Open", "High", "Low", "Close")
> # Save the VIX data to binary file
> load(file="/Users/jerzy/Develop/data/ib_data/vix_cboe.RData")
> ls(vix_env)
> vix_env$vix_index <- vix_index
> save(vix_env, file="/Users/jerzy/Develop/data/ib_data/vix_cboe.RData")
```


```
> # Plot VIX OHLC data in x11 window
> chart_Series(x=vix_index["2018"], name="VIX Index")
> # Plot dygraph
> dygraphs::dygraph(vix_index, main="VIX Index") %>%
+   dyCandlestick()
```

VIX Futures Contracts

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

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

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

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

http://cfe.cboe.com/

http://www.cboe.com/vix

VIX Contract Specifications:

VIX Contract Specifications

VIX Expiration Calendar

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

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

VIX Futures Curve

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

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

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

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

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

```
> # Load VIX futures data from binary file
> load(file="/Users/jerzy/Develop/data/vix_data/vix_cboe.RData")
> # Get all VIX futures for 2018 except January
> symbolv <- ls(vix_env)
> symbolv <- symbolv[grep(glob2rx("*8"), symbolv)]
> symbolv <- symbolv[2:9]
> # Specify dates for curves
> startd <- as.Date("2018-01-11")
> endd <- as.Date("2018-02-05")
> # Extract all VIX futures prices on the dates
> futcurves <- lapply(symbolv, function(symbol) {
+   xtsv <- get(x=symbol, envir=vix_env)
+   Cl(xtsv[c(startd, endd)])
+ })  # end lapply
> futcurves <- rutils::do_call(cbind, futcurves)
> colnames(futcurves) <- symbolv
> futcurves <- t(coredata(futcurves))
> colnames(futcurves) <- c("Contango 01/11/2018",
+                          "Backwardation 02/05/2018")
```

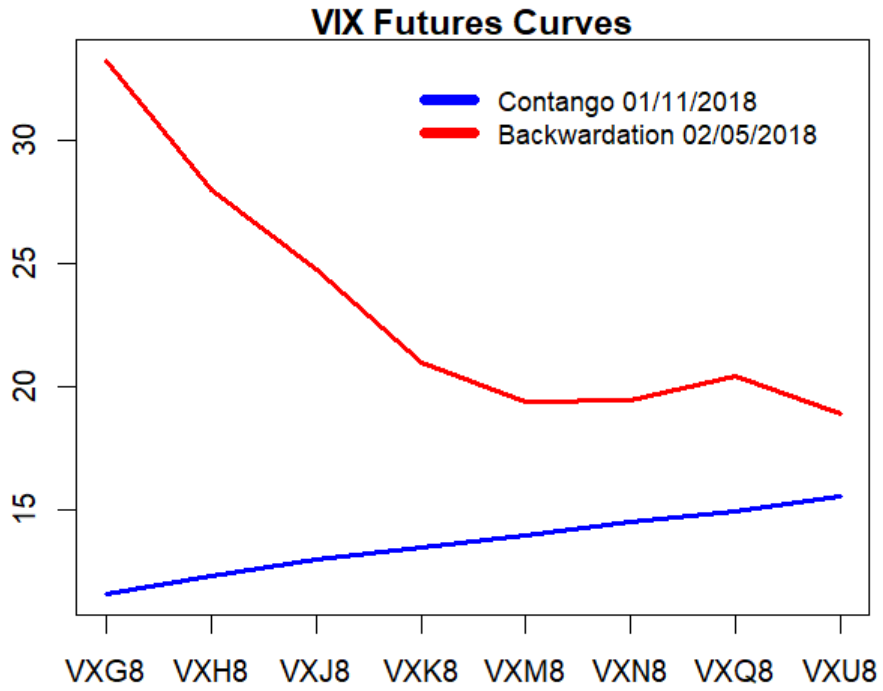
Contango and *Backwardation* of *VIX* Futures Curve

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

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

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

```
> x11(width=7, height=5)
> par(mar=c(3, 2, 1, 1), oma=c(0, 0, 0, 0))
> plot(futcurves[, 1], type="l", lty=1, col="blue", lwd=3,
+      xaxt="n", xlab="", ylab="", ylim=range(futcurves),
+      main="VIX Futures Curves")
> axis(1, at=(1:NROW(futcurves)), labels=rownames(futcurves))
> lines(futcurves[, 2], lty=1, lwd=3, col="red")
> legend(x="topright", legend=colnames(futcurves),
+  inset=0.05, cex=1.0, bty="n", y.intersp=0.5,
+  col=c("blue", "red"), lwd=6, lty=1)
```



Futures Prices at Constant Maturity

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

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

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

VIX Futures Investing

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

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

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

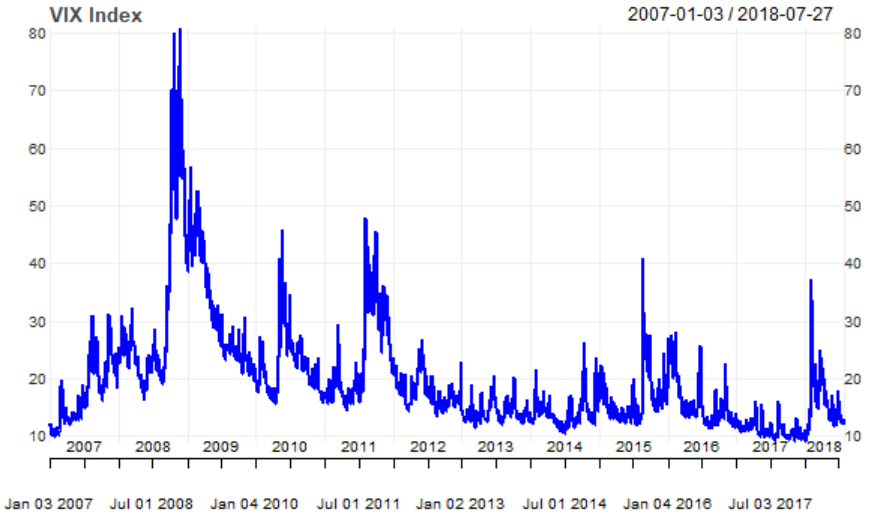
There are several exchange-traded funds (*ETFs*) and exchange traded notes (*ETNs*) which are linked to *VIX* futures.

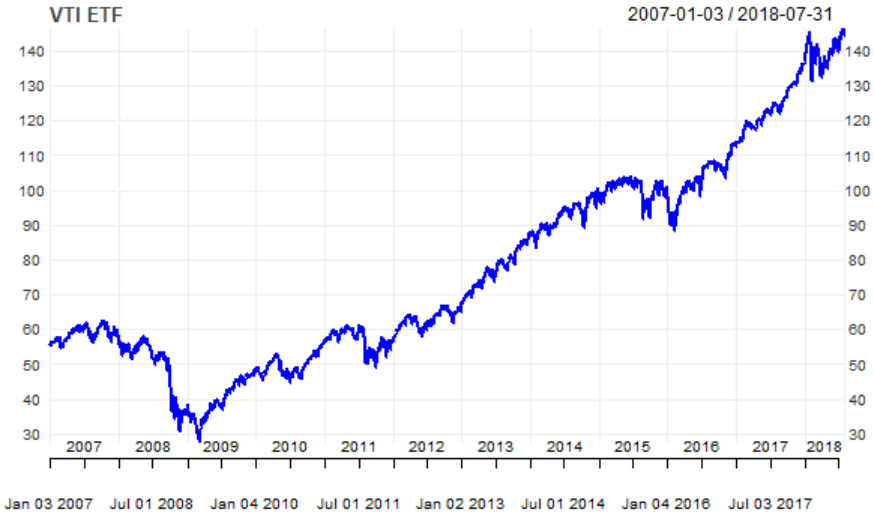
VXX is an *ETN* providing the total return of a *long* *VIX* futures contract (short market risk).

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

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

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





VIX Crash on February 5th 2018

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

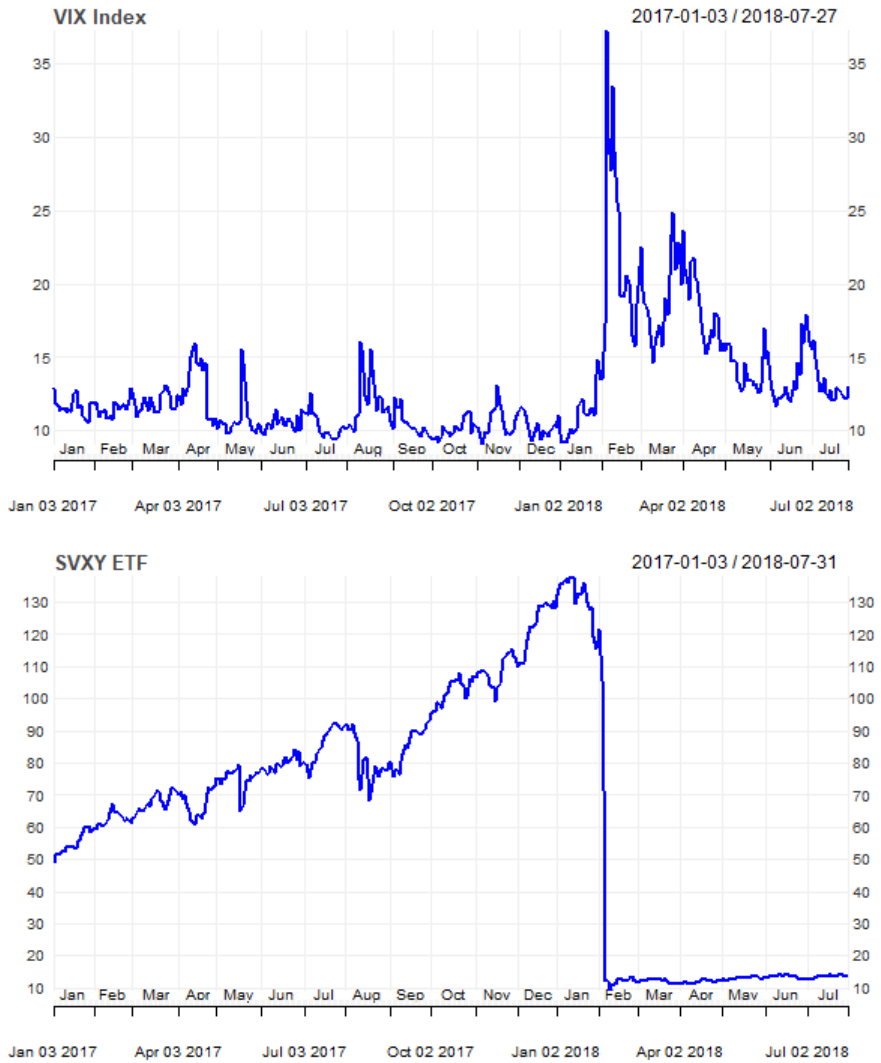
The *SVXY* and *XIV* ETFs had $3.6 billion of assets at the beginning of 2018.

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

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

Volatility Caused Stock Market Crash
XIV ETF Termination Event

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



draft: Types of Market Data

Fundamental company data summarizes the financial, economic, and regulatory state of a company.

Fundamental company data can be divided into several different types:

- Balance Sheet (assets and liabilities),
- Income Statement (profits and losses),
- Cash Flow Statement (current operating cash flows),
- Financial Ratios (performance and risk measures),

Financial ratios summarize the performance and risk measures of a company, and can be used for investment decisions, like value investing.

```
> # Read table of fundamental data into data frame
> fundamental_data <-
+   read.csv(file="/Users/jerzy/Develop/lecture_slides/data/fundamer
```

Balance.Sheet	Income.Statement	Cash.Flow.Statement	Financial.Ratios
Cash & Equivalents	Total Revenue	Cash from Operations	Earnings per Share
Investments	Cost of Goods Sold	Cash from Investments	Price/Earnings Ratio
Receivables	Sales, G and A	Cash from Financing	EBIT
Property & Equipment	Research & Development	FX Adjustment	EBITDA
Goodwill & Intangibles	Depreciation & Amortization	Capital Expenditure	Dividend Yield
Total Assets	Operating Expenses	Starting Cash	Gross Margin
Accounts Payable	Operating Income	Ending Cash	Profit Margin
Long-Term Debt	Investment Income		Free Cash Flow
Total Liabilities	Non-Operating Income		Asset Turnover
Common Stock	Pre-Tax Income		Inventory Turnover
			Receivable Turnover

Types of Fundamental Company Data

Fundamental company data summarizes the financial, economic, and regulatory state of a company.

Fundamental company data can be divided into several different types:

- Balance Sheet (assets and liabilities),
- Income Statement (profits and losses),
- Cash Flow Statement (operating cash flows),
- Financial Ratios (performance and risk measures),

Financial ratios summarize the performance and risk measures of a company, and can be used for investment decisions, like value investing.

```
> # Read table of fundamental data into data frame
> fundamental_data <-
+   read.csv(file="/Users/jerzy/Develop/lecture_slides/data/fundamer
```

Balance.Sheet	Income.Statement	Cash.Flow.Statement	Financial.Ratios
Cash & Equivalents	Total Revenue	Cash from Operations	Earnings per Share
Investments	Cost of Goods Sold	Cash from Investments	Price/Earnings Ratio
Receivables	Sales, G and A	Cash from Financing	EBIT
Property & Equipment	Research & Development	FX Adjustment	EBITDA
Goodwill & Intangibles	Depreciation & Amortization	Capital Expenditure	Dividend Yield
Total Assets	Operating Expenses	Starting Cash	Gross Margin
Accounts Payable	Operating Income	Ending Cash	Profit Margin
Long-Term Debt	Investment Income		Free Cash Flow
Total Liabilities	Non-Operating Income		Asset Turnover
Common Stock	Pre-Tax Income		Inventory Turnover
			Receivable Turnover

Sources of Fundamental Company Data

Fundamental company data is obtained from financial statements provided to government agencies and from earnings statements provided to investors.

U.S. companies are required to provide quarterly *10Q* reports and annual *10K* reports to the *Securities and Exchange Commission* (*SEC*).

The *10Q* and *10K* reports are publicly available on the *SEC EDGAR* website.

The data provided by *EDGAR* only goes back to the year 2009, because the *SEC* did not mandate electronic filing before then.

The data from the *10Q* and *10K* reports is compiled by vendors such as *Compustat* and *FactSet*, into premium databases.

There are also free databases of fundamental company data, from providers such as *SimFin*, *usfundamentals*, and *Quandl*.

WRDS redistributes fundamental company data from *Compustat*, *S&P Capital IQ*, *Thomson Reuters*, *FactSet*, *Markit*, etc.

There are occasional gaps in the quarterly data, because companies are not required to file quarterly reports on dates when they're also filing annual reports.

The report harmonization procedure fails when companies change their accounting treatment from year to year.

Quandl has written a review of both premium and free sources of fundamental company data.

Identifiers of Company Data

Identifiers are strings that correspond to *corporate entities* and *securities*.

Headers are current identifiers, while *historical identifiers* have date ranges when they were valid.

Some *universal identifiers* are company namesv, tickers, CUSIP, SEDOL, ISIN, and CIK.

Most *universal identifiers*, such as company namesv, tickers, CUSIPs, etc. change over time due to corporate events such as mergers, bankruptices, etc.

Tickers are short and recognizable strings, but are not unique, and can change over time and can be reused by a different company.

CUSIPs change infrequently and are never reused, and identify securities in the U.S. and Canada.

SEDOL identify securities in the UK.

ISIN identify international securities in the UK.

CIK identify corporations and individuals who have filed disclosures with the SEC.

Data vendors such as *Compustat*, *Capital IQ*, *IBES*, *Markit*, and *FactSet* have introduced *proprietary identifiers* that are *permanent*.

CRSP *PERMCO* and *PERMNO* are identifiers of companies and securities.

Compustat *GVKEYs* and *IIDs* are identifiers of companies and securities.

The *PERMCO* and *PERMNO* are *permanent identifiers* which never change, even if other identifiers such as tickers do change.

So analysts use *permanent identifiers*, such as *PERMCO*, *PERMNO*, and *GVKEY*.

Capital IQ *Company IDs* are identifiers of companies.

IBES *tickers* are identifiers of securities.

Markit *RED codes* are identifiers of companies.

Factset *Entity IDs* and *Perm Sec IDs* are identifiers of companies and securities.

Wharton Research Data Services *WRDS*

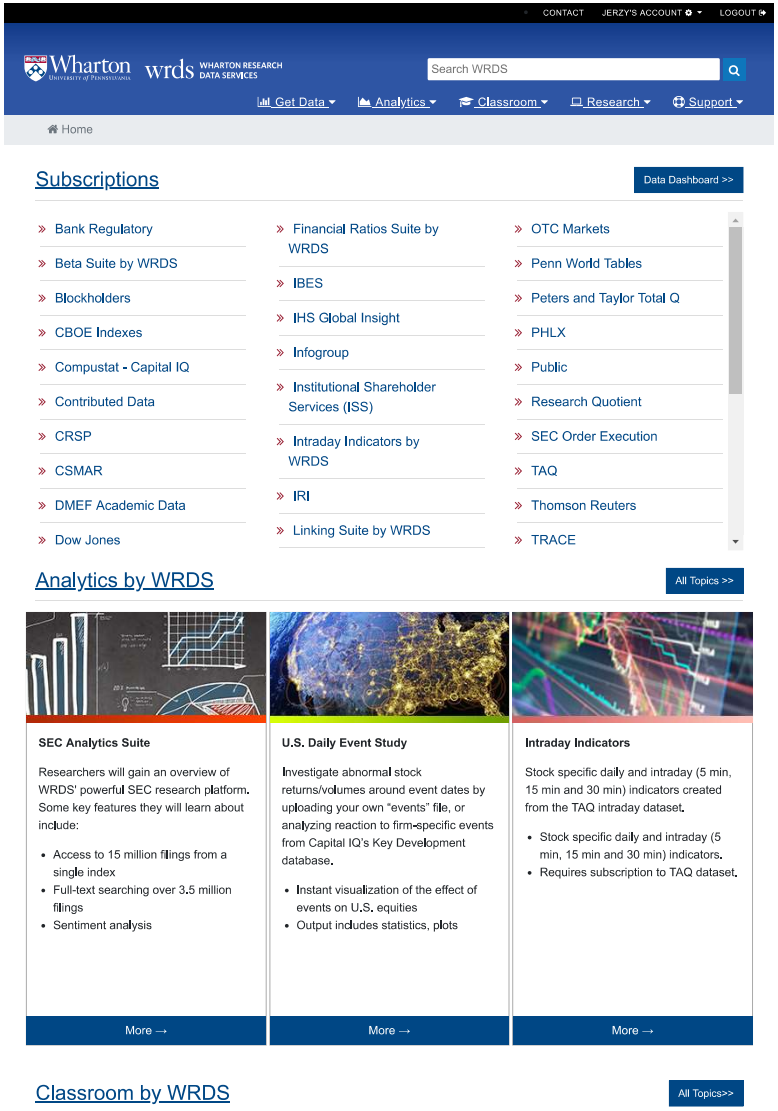
Wharton Research Data Services (*WRDS*) is a distributor of premium third party data for the academic and research communities.

WRDS provides time series of security prices and fundamental company data, and other financial, econometric, and social datasets.

WRDS provides stock prices, options and implied volatilities, stock fundamentals, financial ratios, zoo::indexes, earnings estimates, analyst ratings, etc.

WRDS redistributes fundamental company data from *Compustat*, *S&P Capital IQ*, *Thomson Reuters*, *FactSet*, *Hedge Fund Research*, *Markit*, etc.

NYU students can obtain user accounts for *WRDS* data.

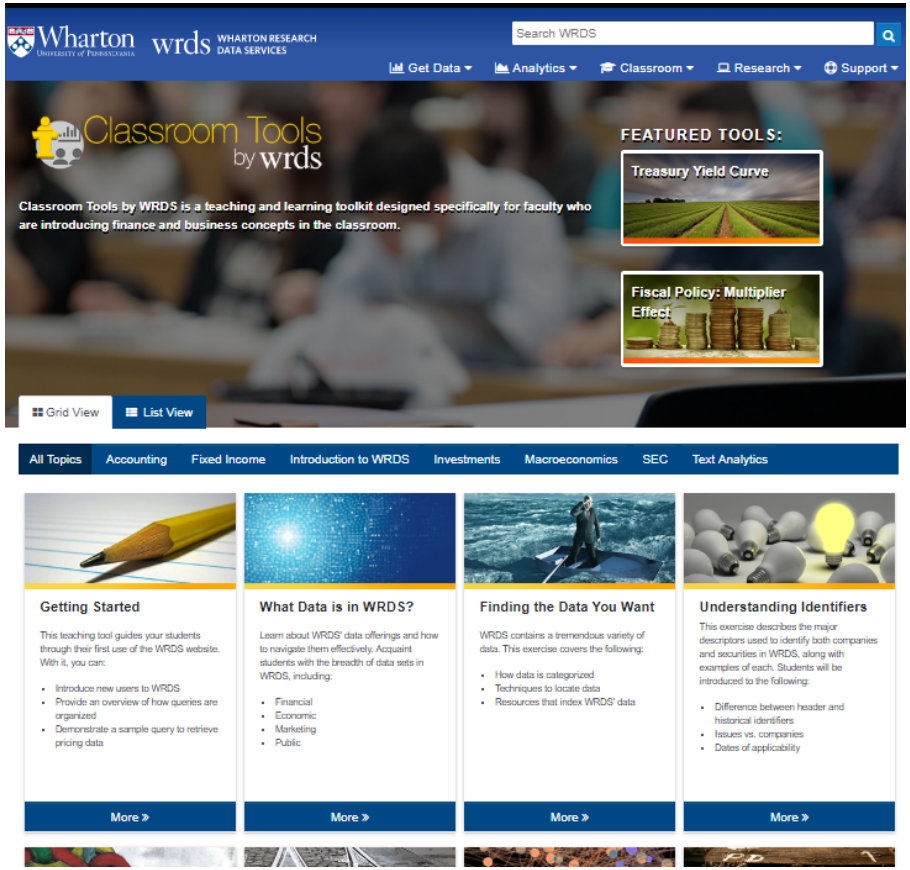


WRDS Help and Documentation

WRDS provides extensive *Learning Resources and Documentation*.

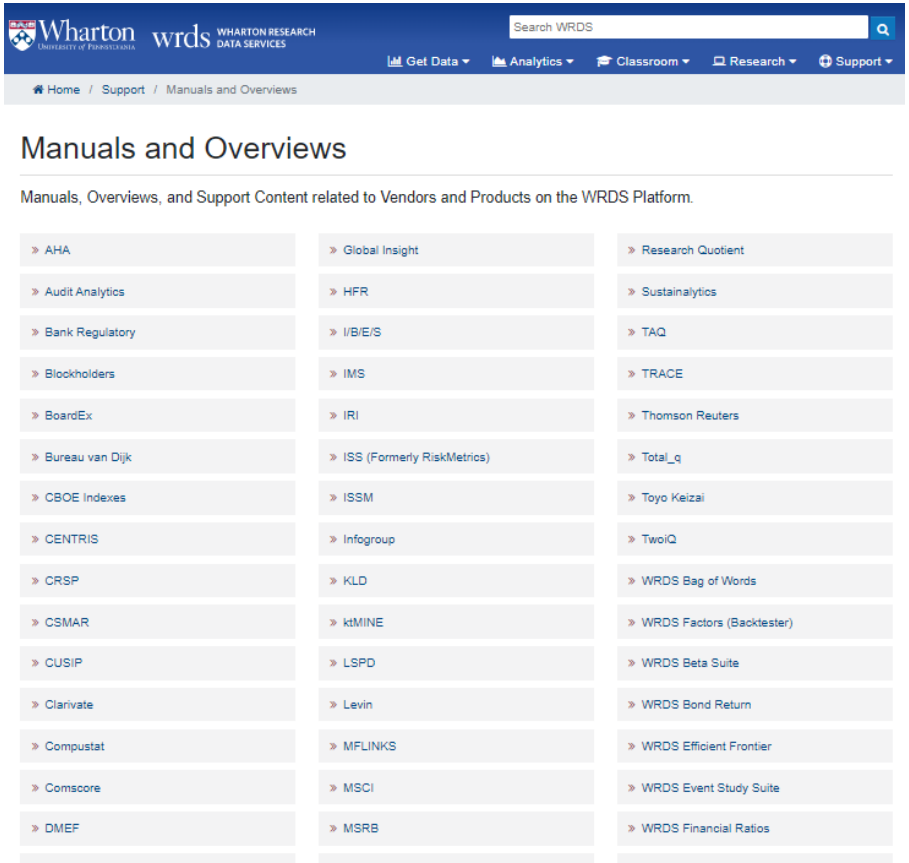
WRDS provides online help and a guide to its datasets:

- *Getting Started*
- *WRDS Data*
- *Finding Data*
- *Understanding Identifiers*
- *Company Identifiers*
- *Futures Contracts*
- *Using SAS Studio*



WRDS Vendor and Database Manuals

WRDS provides many *Vendor and Database Manuals*.

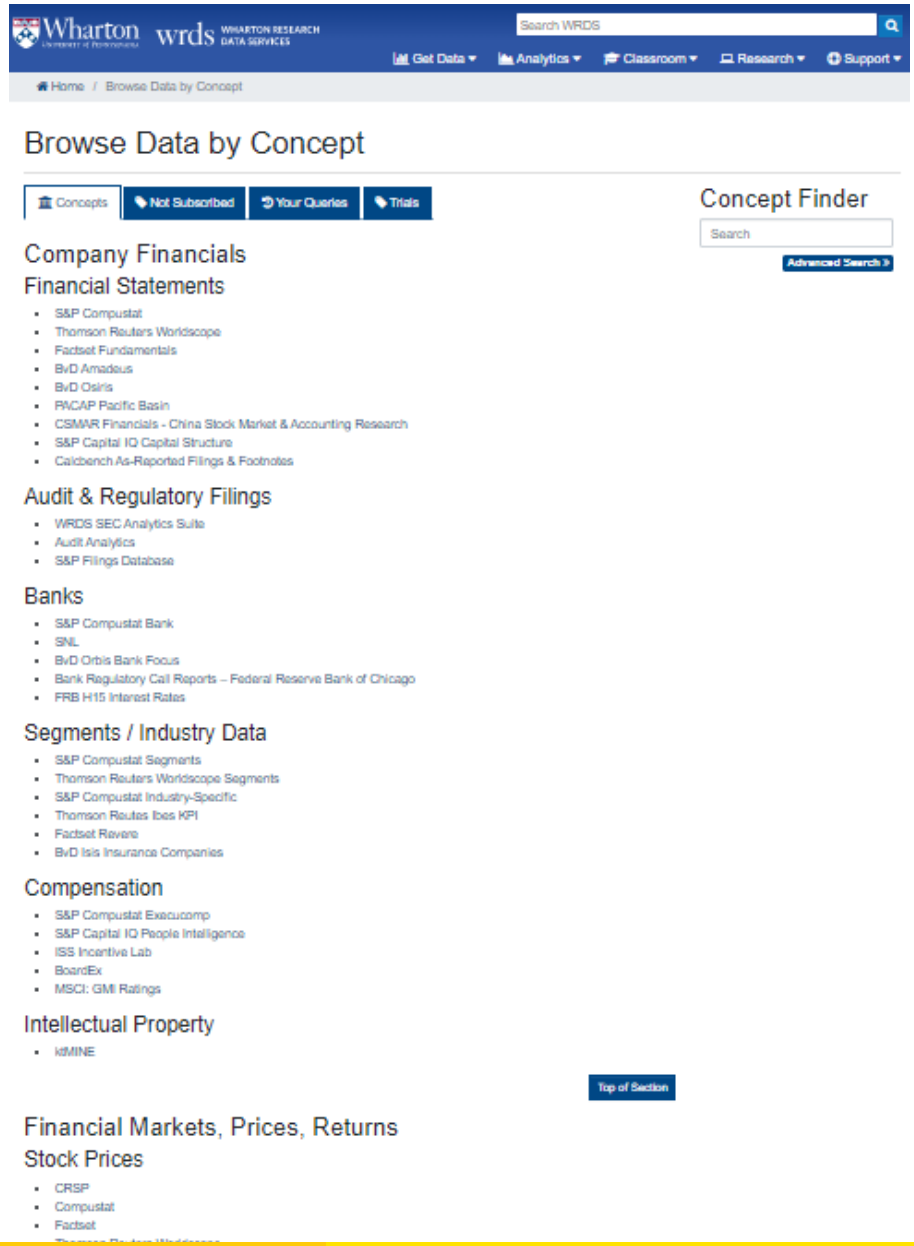


Downloading Data From *WRDS*

The two main databases on *WRDS* are the *Compustat* database of fundamental company data and security prices, and the *CRSP* database of security prices.

The *WRDS* data can be accessed through vendor pages, which are conveniently organized by *concept*.

The easest way to download data from *WRDS* into `R`, is by first downloading it into a `.csv` file, and then reading it into `R`.



The *Compustat* Database on *WRDS*

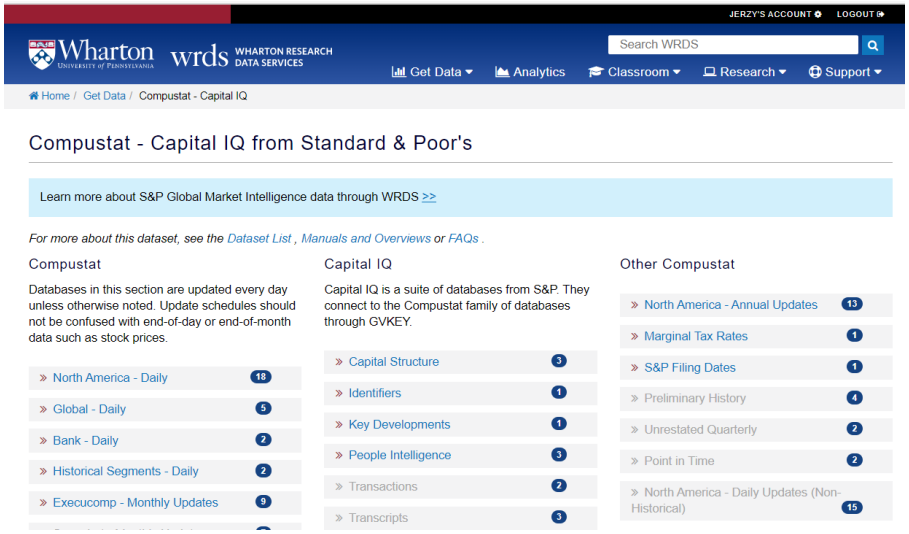
Compustat is a provider of fundamental company data, and also security prices, and other data such as credit ratings.

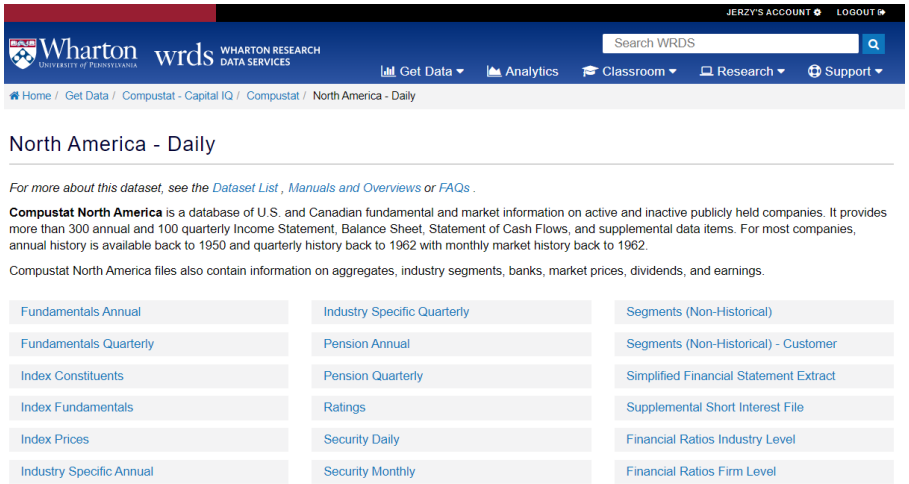
Daily North American security prices can be downloaded from the *Compustat Daily Prices* database, which can be reached by navigating from the *WRDS* home page to *North America - Daily* and then to *Security Daily*.

Compustat is part of *S&P Capital IQ*.

Compustat uses the *GVKEY* and *IID* identifiers of companies and securities.

Compustat provides only recent time series of data starting in 1983.





Package *rwrds* for Downloading Data From *WRDS*

The package *rwrds* contains functions for downloading data from *WRDS*.

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

Downloading Names Table From *Compustat*

The *Compustat* names table is a database containing various company *identifiers* (company namesv, tickers, CUSIPs, GVKEYs).

The names table can be used to cross-reference the *identifiers*, for example to find the *GVKEYs* from the company *tickers*.

The function `rwrds::compustat_names()` downloads the names table from *Compustat*.

```
> library(rwrds)
> library(dplyr)
> # Establish connection to WRDS
> wrds_con <- rwrds::wrds_connect(username="jp3900", password="Ripva
> # Download Compustat names table as dplyr object
> namesv_table <- rwrds::compustat_names(wrds=wrds_con, subset=FALSE
> dim(namesv_table)
> # Save names table as csv file
> write.csv(namesv_table, file="/Users/jerzy/Develop/lecture_slides/
> # rm(namesv_table)
> # Read names table from csv file
> namesv_table <- read.csv(file="/Users/jerzy/Develop/lecture_slides
> # symbol <- "VTI"
> # match(symbol, namesv_table$tic)
> # Create ETF symbols (tickers)
> symbolv <- c("VTI", "VEU", "EEM")
> # Get cusips of symbolv
> indeks <- match(symbolv, namesv_table$tic)
> names(indeks) <- symbolv
> etf_cusips <- namesv_table$cusip[indeks]
> names(etf_cusips) <- symbolv
> # Save cusips into text file
> cat(etf_cusips, file="/Users/jerzy/Develop/lecture_slides/data/etf
> # Save gvkeys into text file
> etf_gvkeys <- namesv_table$gvkey[indeks]
> names(etf_gvkeys) <- symbolv
> cat(etf_gvkeys, file="/Users/jerzy/Develop/lecture_slides/data/etf
```

draft: Downloading Index Constituents From *Compustat*

The names and tickers of the constituents of a given index can be downloaded from the *WRDS Compustat web page*.

The constituents can be downloaded for a specific date, or for a range of dates, to obtain the names and tickers of all the companies which belonged to the index during that time.

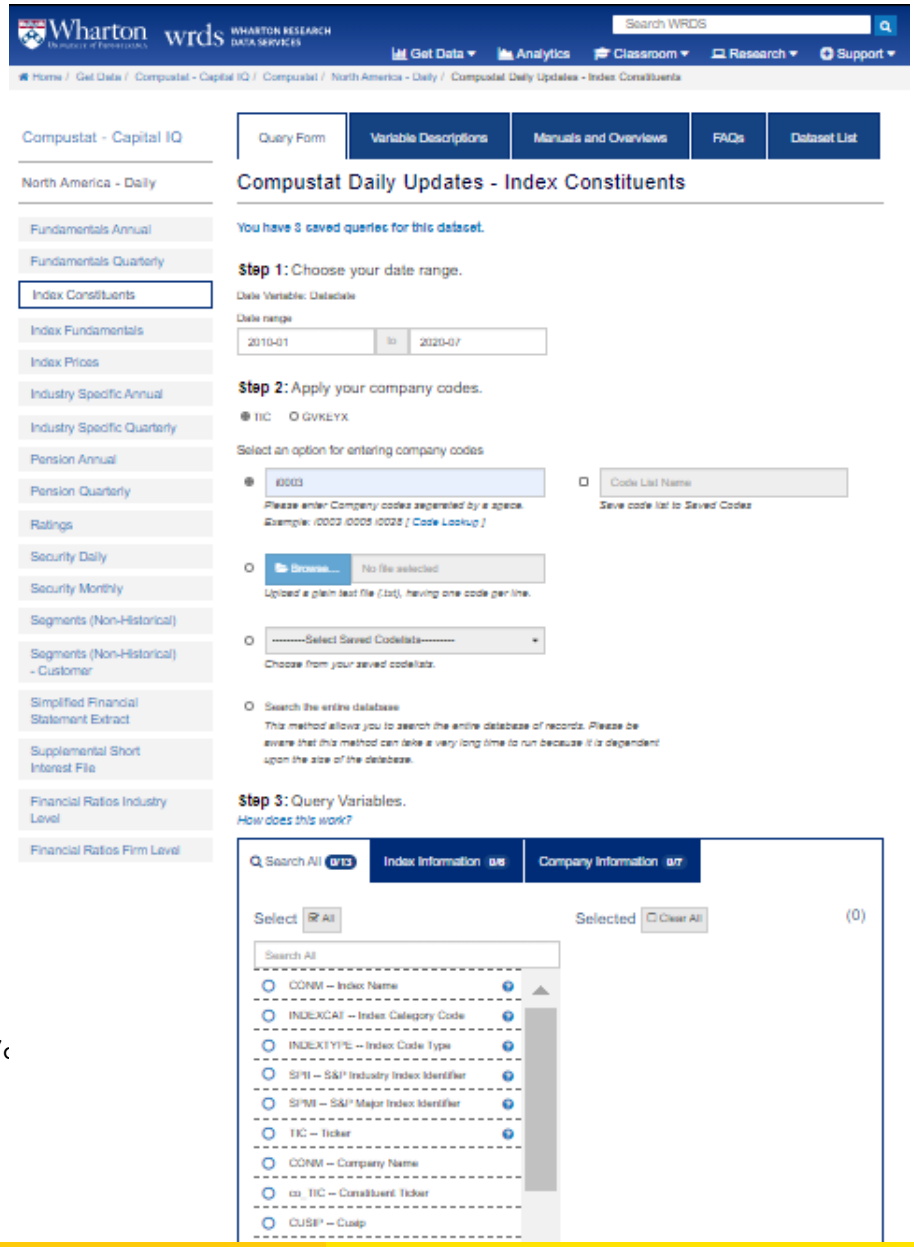
The *S&P500 Index* ticker is equal to *i0003*, and its *GVKEY* is equal to 3.

WRDS provides *help about how to download index constituents*. The table with the index constituents can be read into `R` a `.csv` from the file downloaded from the *WRDS Compustat web page*.

The table of constituents is a data frame containing various company *identifiers* (company namesv, tickers, CUSIPs, GVKEYs).

Some *GVKEYs* are duplicates because some companies entered the index more than once.

The *GVKEYs* and *CUSIPs* can be saved into text files for use in *WRDS* data queries.




```
> # Read .csv file with S&P500 constituents
> sp500table <- read.csv(file="/Users/jerzy/Develop/lecture_slides/c
> class(sp500table)
> dim(sp500table)
> head(sp500table)
> # Select unique sp500 tickers and save them into text file
> sp500_tickers <- unique(sp500table$co_tic)
```

Downloading the *S&P500* Index Constituents

As of July 2020, the *S&P500* stock index constituents data was removed from *Compustat*, so it's no longer available on *WRDS*.

As an alternative, users can download the *SPY ETF* constituents from *State Street*.

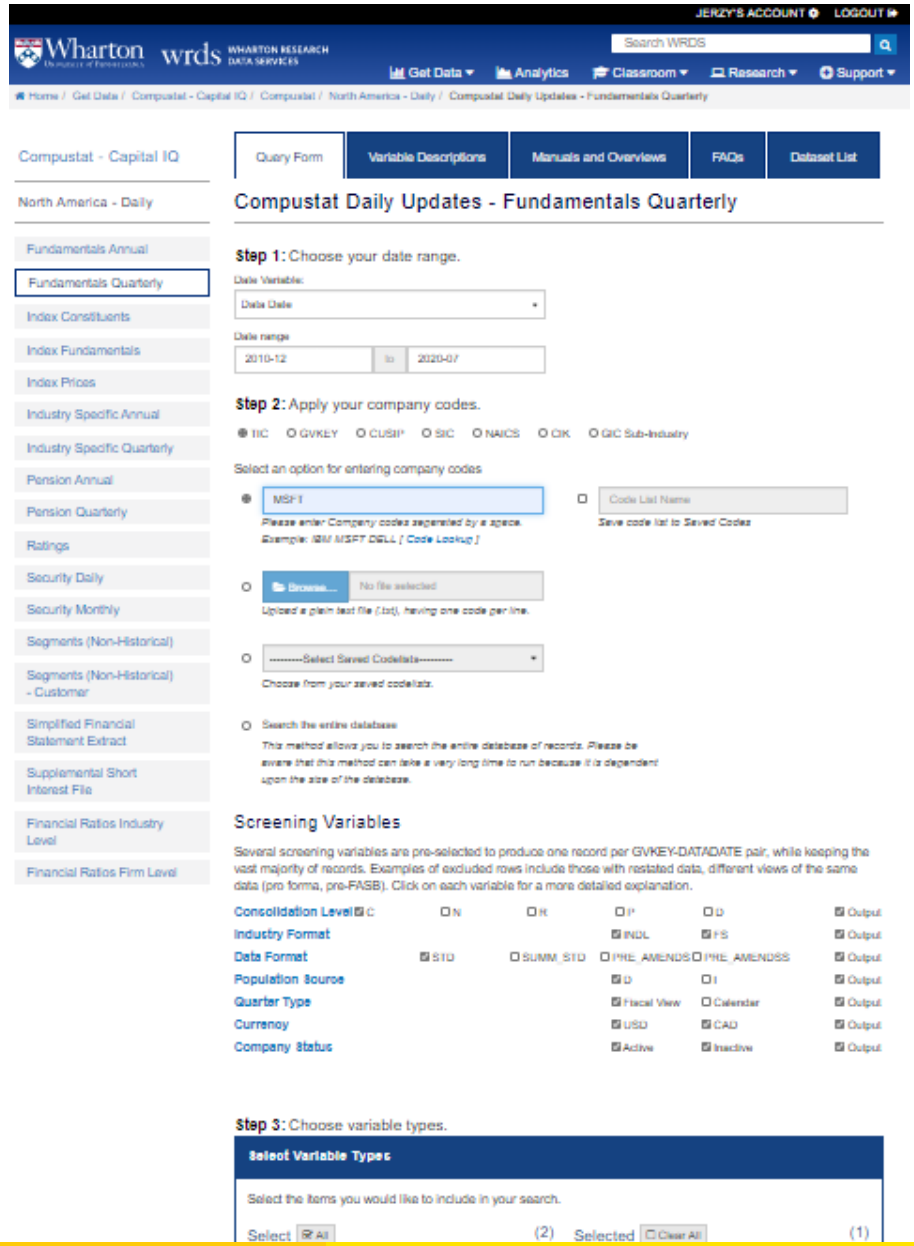
The file `sp500_constituents.csv` contains the symbols (tickers) for almost 800 stocks belonging to the *S&P500* stock index, either now or in the past.

```
> # Read .csv file with S&P500 constituents
> sp500table <- read.csv(file="/Users/jerzy/Develop/lecture_slides/d
> class(sp500table)
> dim(sp500table)
> head(sp500table)
```

Downloading Fundamental Company Data

Fundamental company data can be downloaded from the *Compustat Fundamentals* database.

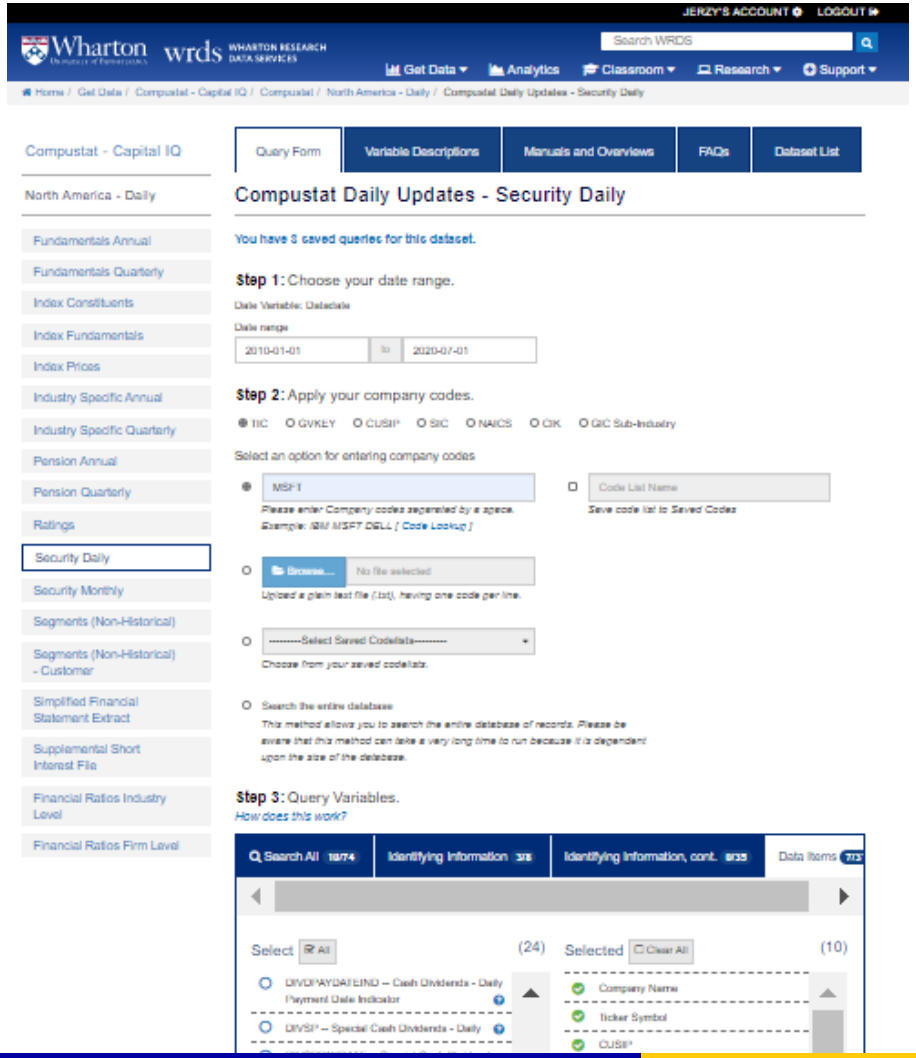
The user can download data for either a single identifier, or for many identifiers supplied in a text file.



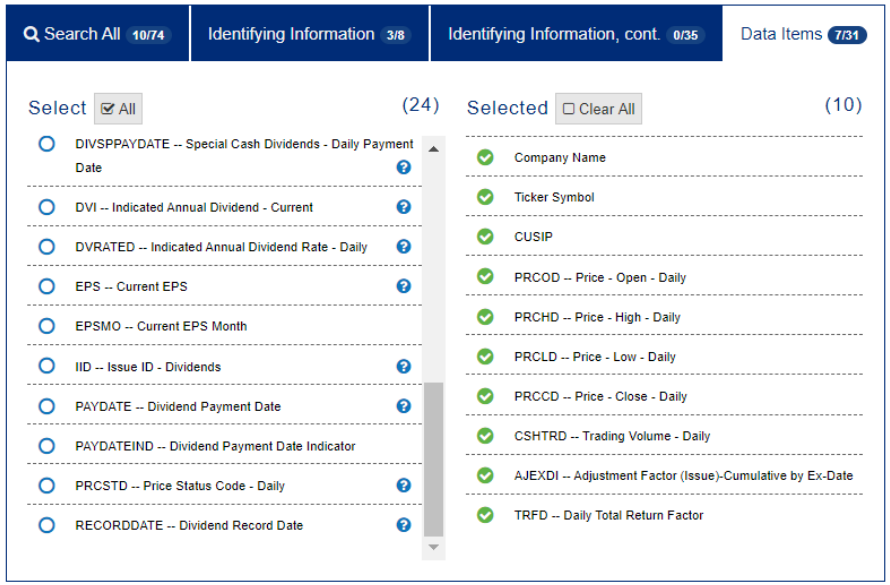
Downloading Daily Prices From *Compustat*

Daily security prices can be downloaded from the *Compustat Daily Prices* database.

The user can download data for either a single identifier, or for many identifiers in a text file.



The query for *OHLC* prices should also include the *split adjustment factor* and the *total return factor*.



Reading Daily Prices Into R

The time series data downloaded into a `.csv` file can then be read into R.

But the data downloaded from the *Compustat Daily Prices* database may contain data for several related securities, so it must be properly selected (pruned).

The *OHLC* prices can be adjusted by dividing them by the *split adjustment factor* and multiplying them by the *total return factor*.

The *OHLC* prices are also divided by the final *total return factor* so that the most recent adjusted prices are equal to the most recent market prices.

```
> # Read .csv file with TAP OHLC prices
> ohlc <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/TA
> # ohlc contains cusips not in sp500_cusips
> cusips <- unique(ohlc$cusip)
> cusips %in% sp500_cusips
> # Select data only for sp500_cusips
> ohlc <- ohlc[ohlc$cusip %in% sp500_cusips, ]
> # ohlc contains tickers not in sp500_tickers
> tickers <- unique(ohlc$tic)
> tickers %in% sp500_tickers
> # Select data only for sp500_tickers
> ohlc <- ohlc[ohlc$tic %in% sp500_tickers, ]
> # Select ticker from sp500table
> symbol <- sp500table$co_tic[match(ohlc$gvkey[1], sp500table$gvkey)
> # Adjustment factor and total return factor
> factadj <- drop(ohlc[, "ajexdi"])
> factr <- drop(ohlc[, "trfd"])
> # Extract index of dates
> dates <- drop(ohlc[, "datadate"])
> dates <- lubridate::ymd(dates)
> # Select only OHLCV data columns
> ohlc <- ohlc[, c("prcod", "prchd", "prcld", "prccd", "cshtrd")]
> colnames(ohlc) <- paste(symbol, c("Open", "High", "Low", "Close",
> # Coerce to xts series
> ohlc <- xts::xts(ohlc, dates)
> # Fill the missing (NA) Open prices
> isna <- is.na(ohlc[, 1])
> ohlc[isna, 1] <- (ohlc[isna, 2] + ohlc[isna, 3])/2
> sum(is.na(ohlc))
> # Adjust all the prices
> ohlc[, 1:4] <- factr*ohlc[, 1:4]/factadj/factr[NROW(factr)]
> ohlc <- na.omit(ohlc)
> plot(quantmod::Cl(ohlc), main="TAP Stock")
```

Function for Reading Daily Prices Into R

The function `format_ohlc()` formats raw data downloaded from *Compustat* into a time series of *OHLC* prices.

```
> # Define formatting function for OHLC prices
> format_ohlc <- function(ohlc, environ_ment) {
+   # Select ticker from sp500table
+   symbol <- sp500table$co_tic[match(ohlc$gvkey[1], sp500table$gvke
+   # Split adjustment and total return factors
+   factadj <- drop(ohlc[, c("ajexdi")])
+   factr <- drop(ohlc[, "trfd"])
+   factr <- ifelse(is.na(factr), 1, factr)
+   # Extract dates index
+   dates <- drop(ohlc[, "datadate"])
+   dates <- lubridate::ymd(dates)
+   # Select only OHLCV data
+   ohlc <- ohlc[, c("prcod", "prchd", "prcld", "prccd", "cshtrd")]
+   colnames(ohlc) <- paste(symbol, c("Open", "High", "Low", "Close"
+   # Coerce to xts series
+   ohlc <- xts::xts(ohlc, dates)
+   # Fill NA prices
+   isna <- is.na(ohlc[, 1])
+   ohlc[isna, 1] <- (ohlc[isna, 2] + ohlc[isna, 3])/2
+   # Adjust the prices
+   ohlc[, 1:4] <- factr*ohlc[, 1:4]/factadj/factr[NROW(factr)]
+   # Copy the OHLCV data to environ_ment
+   ohlc <- na.omit(ohlc)
+   assign(x=symbol, value=ohlc, envir=environ_ment)
+   symbol
+ }  # end format_ohlc
```

Reading Index Constituent Prices Into R

The prices for all the index constituents are first downloaded from *Compustat* into a single `.csv` file, and then they are read into R and formatted into a time series of *OHLC* prices.

But the prices downloaded from the *Compustat Daily Prices* database often contain prices for several related securities from the same company, so they must be properly selected (pruned) to match the index constituents.

```
> # Load OHLC prices from .csv file downloaded from WRDS by cusip
> sp500_prices <- read.csv(file="/Users/jerzy/Develop/lecture_slides
> # sp500_prices contains cusips not in sp500_cusips
> cusips <- unique(sp500_prices$cusip)
> NROW(sp500_cusips); NROW(cusips)
> # Select data only for sp500_cusips
> sp500_prices <- sp500_prices[sp500_prices$cusip %in% sp500_cusips,
> # sp500_prices contains tickers not in sp500_tickers
> tickers <- unique(sp500_prices$tic)
> NROW(sp500_tickers); NROW(tickers)
> # Select data only for sp500_tickers
> sp500_prices <- sp500_prices[sp500_prices$tic %in% sp500_tickers,
> # Create new data environment
> sp500env <- new.env()
> # Perform OHLC aggregations by cusip column
> sp500_prices <- split(sp500_prices, sp500_prices$cusip)
> process_ed <- lapply(sp500_prices, format_ohlc,
+                environ_ment=sp500env)
```

Managing Exceptions in Index Constituent Prices

Some securities of past index constituents may no longer be trading because of corporate events (mergers, bankruptcies, etc.), so they should be removed from the data.

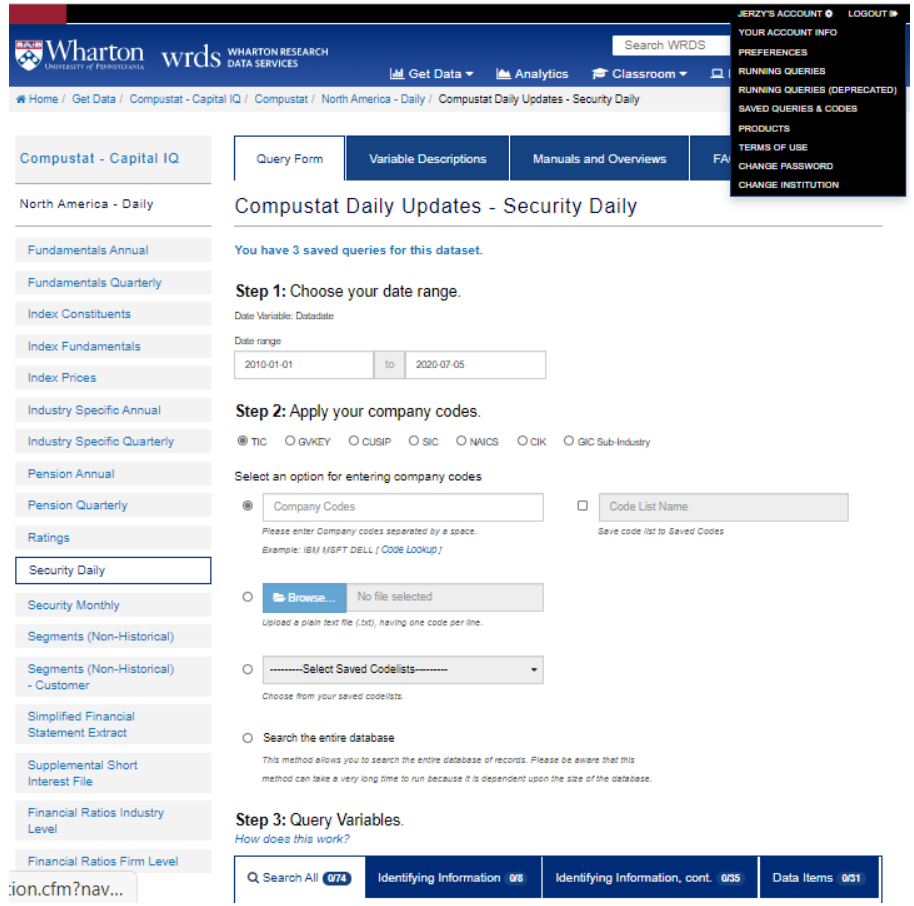
Tickers which contain a dot in their name (like "BRK.B") are not valid symbols in `R`, so they must be renamed.

The column names for symbol `"LOW"` (Lowe's company) must be renamed for the extractor function `quantmod::Lo()` to work properly.

```
> # Get end dates of series in sp500env
> endds <- eapply(sp500env, end)
> endds <- unlist(endds)
> endds <- as.Date(endds)
> # Remove elements with short end dates
> se_lect <- (endds < max(endds))
> rm(list=names(sp500env)[se_lect], envir=sp500env)
> # Rename element "BRK.B" to "BRKB"
> sp500env$BRKB <- sp500env$BRK.B
> rm(BRK.B, envir=sp500env)
> names(sp500env$BRKB) <- paste("BRKB",
+   c("Open", "High", "Low", "Close", "Volume"), sep=".")
> # Rename element "LOW" to "LOWES"
> sp500env$LOWES <- sp500env$LOW
> names(sp500env$LOWES) <- paste("LO_WES",
+   c("Open", "High", "Low", "Close", "Volume"), sep=".")
> rm(LOW, envir=sp500env)
> # Rename element "BF.B" to "BFB"
> sp500env$BFB <- sp500env$BF.B
> names(sp500env$BFB) <- paste("BFB",
+   c("Open", "High", "Low", "Close", "Volume"), sep=".")
> rm(BF.B, envir=sp500env)
> # Save OHLC prices to .RData file
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp5
> plot(quantmod::Cl(sp500env$MSFT))
```

Saving *WRDS* Queries

WRDS queries can be saved and reused again.



draft: Downloading Fundamental Company Data From *Quandl*

SEC is a free database of stock fundamentals extracted from SEC and (but not harmonized),
https://www.quandl.com/data/SEC

RAYMOND is a free database of harmonized stock fundamentals, based on the *SEC* database,
https://www.quandl.com/data/RAYMOND
https://www.quandl.com/data/RAYMOND?keyword=aapl

Wharton Research Data Services (*WRDS*) is a distributor of third party data for the academic and research communities.

WRDS provides time series of stock prices and fundamental company data, and other financial, econometric, and social datasets.

WRDS redistributes fundamental company data from *Compustat*, *S&P Capital IQ*, *Thomson Reuters*, *FactSet*, *Hedge Fund Research*, *Markit*, etc.

The Center for Research in Security Prices (*CRSP* is a provider of historical security prices for the academic and research communities.

Much of the *WRDS* data is free, while premium data can be obtained under a temporary license.

WRDS provides online help and a guide to its datasets:
https://wrds-www.wharton.upenn.edu

WRDS provides stock prices, stock fundamentals, financial ratios, zoo::indexes, options and volatility,

```
> library(Quandl)  # Load package Quandl
> # Register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
>
> # Quandl stock market data
> # https://blog.quandl.com/stock-market-data-ultimate-guide-part-1
> # https://blog.quandl.com/stock-market-data-the-ultimate-guide-par
>
> # Download RAYMOND metadata
> # https://www.quandl.com/data/RAYMOND-Raymond/documentation/metada
>
> # Download S&P500 Index constituents
> # https://s3.amazonaws.com/static.quandl.com/tickers/SP500.csv
>
> # Download AAPL gross profits from RAYMOND
> profitaapl <- Quandl("RAYMOND/AAPL_GROSS_PROFIT_Q", type="xts")
> chart_Series(profitaapl, name="AAPL gross profits")
>
> # Download multiple time series
> pricev <- Quandl(code=c("NSE/OIL", "WIKI/AAPL"),
+    startd="2013-01-01", type="xts")
>
> # Download datasets for AAPL
> # https://www.quandl.com/api/v3/datasets/WIKI/AAPL.json
>
> # Download metadata for AAPL
> pricev <- Quandl(code=c("NSE/OIL", "WIKI/AAPL"),
+    startd="2013-01-01", type="xts")
> # https://www.quandl.com/api/v3/datasets/WIKI/AAPL/metadata.json
>
> # scrape fundamental data from Google using quantmod - doesn't wo
> funda_mentals <- getFinancials("HPQ", src="google", auto.assign=FA
> # view quarterly fundamentals
> viewFinancials(funda_mentals,  period="Q")
> viewFinancials(funda_mentals)
>
> # scrape fundamental data from Yahoo using quantmod
```

The *CRSP* Database on *WRDS*

The Center for Research in Security Prices (*CRSP*) is a provider of historical security prices, and is an affiliate of the University of Chicago Booth School of Business.

CRSP data provides daily prices, but it's updated only monthly, quarterly, and annually.

CRSP provides very long time series of data starting in 1926.

CRSP data is indexed using the *PERMCO* company identifier and the *PERMNO* security identifier.

Since a single company can have many securities, therefore a single *PERMCO* may be linked to multiple *PERMNOs*.

WRDS provides a tool to translate tickers to PERMCO/PERMNO:

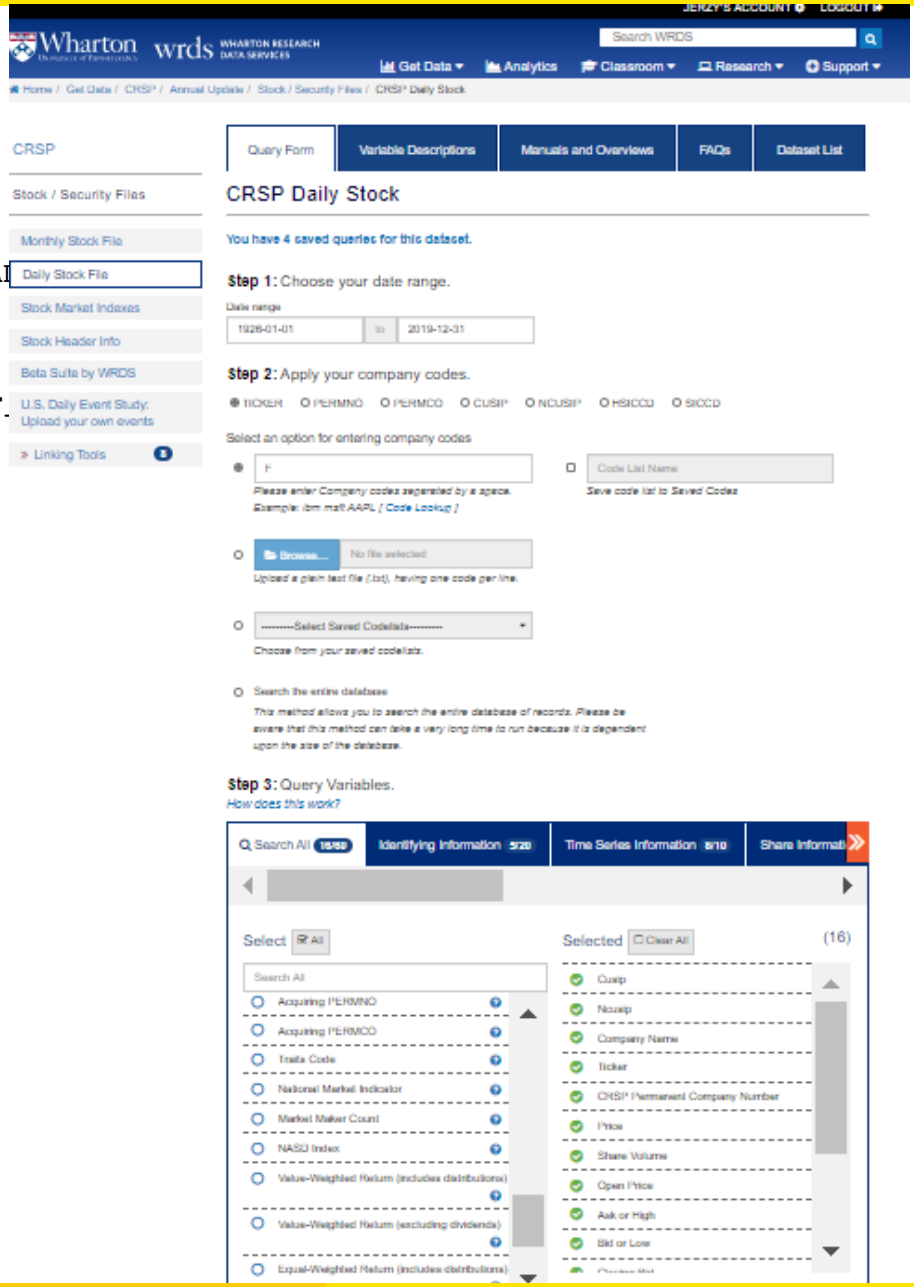
https://wrds-web.wharton.upenn.edu/wrds/ds/crsp/tools_a/dse/translate/index.cfm

Downloading Daily Prices From *CRSP*

Daily North American security prices can be downloaded from the *CRSP daily* database.

The time series data can be downloaded into a `.csv` file, and then read into `R`.

```
> # Specify class for column "TICKER" so that "F" doesn't become FAI
> col_class <- "character"
> names(col_class) <- "TICKER"
> # Read .csv file with Ford OHLC prices
> ohlc <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/F_
+   colClasses=col_class)
> symbol <- ohlc[1, "TICKER"]
> # Adjustment factor
> factadj <- drop(ohlc[, "CFACPR"])
> # Extract dates index
> dates <- drop(ohlc[, "date"])
> dates <- lubridate::ymd(dates)
> # Select only OHLCV data
> ohlc <- ohlc[, c("OPENPRC", "ASKHI", "BIDLO", "PRC", "VOL")]
> colnames(ohlc) <- paste(symbol, c("Open", "High", "Low", "Close",
> # Coerce to xts series
> ohlc <- xts::xts(ohlc, dates)
> # Fill missing Open NA prices
> isna <- is.na(ohlc[, 1])
> ohlc[isna, 1] <- (ohlc[isna, 2] + ohlc[isna, 3])/2
> # Adjust all the prices
> ohlc[, 1:4] <- ohlc[, 1:4]/factadj/factr[NROW(factr)]
> plot(quantmod::Cl(ohlc), main="Ford Stock")
```



draft: The Merged *CRSP/Compustat* Database

Links:
http://kaichen.work/?p=138
http://www.crsp.org/products/documentation/crsp-link
https://mingze-gao.com/posts/merge-compustat-and-crsp/
http://www.crsp.org/products/documentation/crspcompu
merged-database-guide

The Center for Research in Security Prices (*CRSP* is a provider of historical security prices.

CRSP data is indexed using the *PERMCO* company identifier and the *PERMNO* security identifier.

Therefore a given *PERMCO* may be linked to multiple *PERMNOs*.

Compustat provides financial statement data of a company. The micro unit on Compustat is each and every company. However, it is less known that Compustat also provides security data. In addition, because the coverage of Compustat is more extensive than that of *CRSP*, Compustat contains addtional security data that are unavailable on *CRSP*.

Compustat uses *IID* and *GVKEY* to identify all securities. A marker item, PRIMISS, indicates whether a security is primary or secondary.

Similar to *PERMCO* on *CRSP*, one *GVKEY* may have multiple *IIDs* assigned to it.

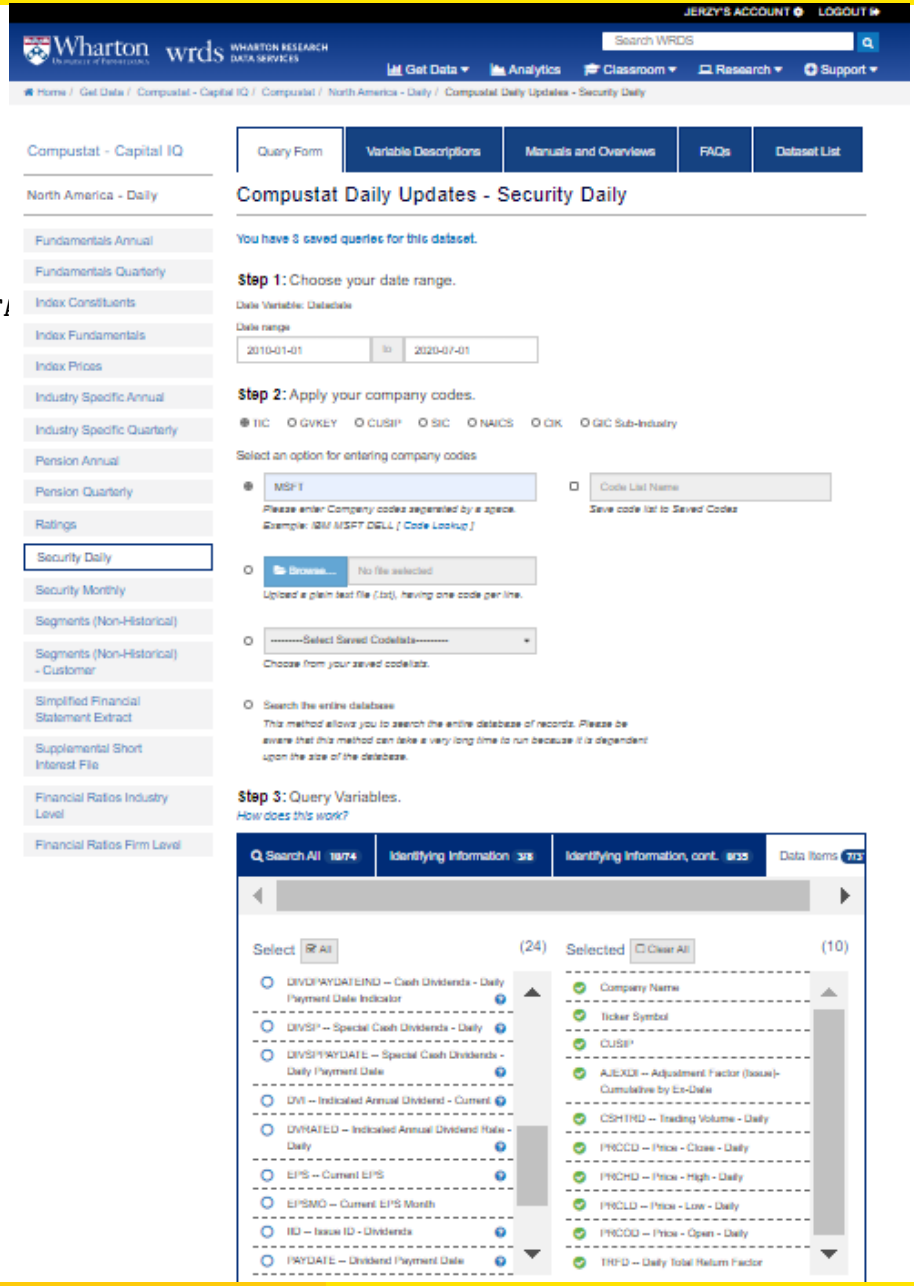
Database	Identifier	Description
CRSP	PERMCO	Permanent company identifier.
CRSP	PERMNO	Permanent security identifier. One PERMNO belongs to only one PERMCO. One PERMCO may have one or more PERMNOs.
Compustat	GVKEY	Permanent company identifier.
Compustat	IID	Permanent security identifier. One GVKEY may have multiple IIDs. Both are needed to identify a Compustat security.
Compustat	PRIMISS	Indicator for primary (P) or secondary (J) securities.

draft: Downloading Daily Prices From Merged *CRSP/Compustat* Database

Daily North American security prices can be downloaded from the *CRSP/Compustat daily* database.

The time series data can be downloaded into a `.csv` file, and then read into R.

```
> # Read .csv file with TAP OHLC prices
> ohlc <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/TA
> symbol <- ohlc[1, "tic"]
> # Adjustment factor and total return factor
> factadj <- drop(ohlc[, "ajexdi"])
> factr <- drop(ohlc[, "trfd"])
> # Extract dates index
> dates <- drop(ohlc[, "datadate"])
> dates <- lubridate::ymd(dates)
> # Select only OHLCV data
> ohlc <- ohlc[, c("prcod", "prchd", "prcld", "prccd", "cshtrd")]
> colnames(ohlc) <- paste(symbol, c("Open", "High", "Low", "Close",
> # Coerce to xts series
> ohlc <- xts::xts(ohlc, dates)
> # Fill missing Open NA prices
> isna <- is.na(ohlc[, 1])
> ohlc[isna, 1] <- (ohlc[isna, 2] + ohlc[isna, 3])/2
> sum(is.na(ohlc))
> # Adjust all the prices
> ohlc[, 1:4] <- factr*ohlc[, 1:4]/factadj/factr[NROW(factr)]
> plot(quantmod::Cl(ohlc), main="TAP Stock")
```



Downloading Fama-French Factors from *Quandl*

The Fama/French factors are constructed using six value-weight portfolios formed on size and book-to-market,
https://www.quandl.com/data/KFRENCH/FACTORS_D

`Mkt-RF` is the excess return on the market (value-weighted NYSE, AMEX, and NASDAQ stocks minus the one-month Treasury bill rate).

`SMB` (Small Minus Big) is the return on the three small-cap portfolios minus the three big-cap portfolios.

`HML` (High Minus Low) is the return on the two value portfolios minus the two growth portfolios.

```
> # Download Fama-French factors from KFRENCH database
> factors <- Quandl(code="KFRENCH/FACTORS_D",
+   startd="2001-01-01", type="xts")
> dim(factors)
> head(factors)
> tail(factors)
> chart_Series(cumsum(factors["2001/", 1]/100),
+   name="Fama-French factors")
```

Trade and Quote (*TAQ*) Data

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

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

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

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

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

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

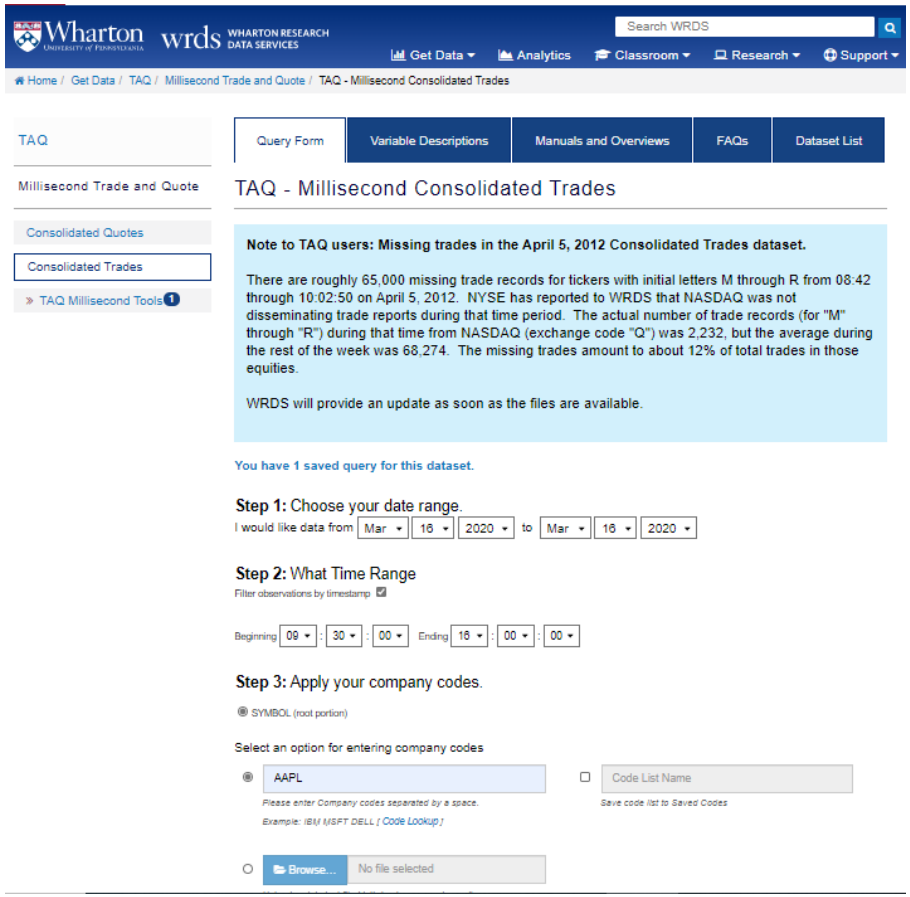
```
> # Load package HighFreq
> library(HighFreq)
> # Or load the high frequency data file directly:
> # symbolv <- load("/Users/jerzy/Develop/R/HighFreq/data/hf_data.RD
> head(HighFreq::SPY_TAQ)
                    Bid.Price Bid.Size Ask.Price Ask.Size Trade.Pric
2014-05-02 00:00:01       188        1       189       15          N
2014-05-02 08:00:01       188        1       189        5         18
2014-05-02 08:00:02       189        1       189        5          N
2014-05-02 08:01:13       188        1       189        5          N
2014-05-02 08:01:29       188        1       189        5          N
2014-05-02 08:01:52       189        2       189        5          N
> head(HighFreq::SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2008-01-02 09:31:00      147      147     147       147     591203
2008-01-02 09:32:00      147      147     147       147     385457
2008-01-02 09:33:00      147      147     147       147     343700
2008-01-02 09:34:00      147      147     147       147     863418
2008-01-02 09:35:00      147      147     147       147     457500
2008-01-02 09:36:00      147      147     147       147     416708
> tail(HighFreq::SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2014-05-19 15:56:00      189      189     189       189     321321
2014-05-19 15:57:00      189      189     189       189     299474
2014-05-19 15:58:00      189      189     189       189     261357
2014-05-19 15:59:00      189      189     189       189     435886
2014-05-19 16:00:00      189      189     189       189    1824185
2014-05-19 16:01:00      189      189     189       189      63920
```

Downloading *TAQ* Data From *WRDS*

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

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

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



Reading *TAQ* Data From .csv Files

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

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

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

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

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

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

Microstructure Noise in High Frequency Data

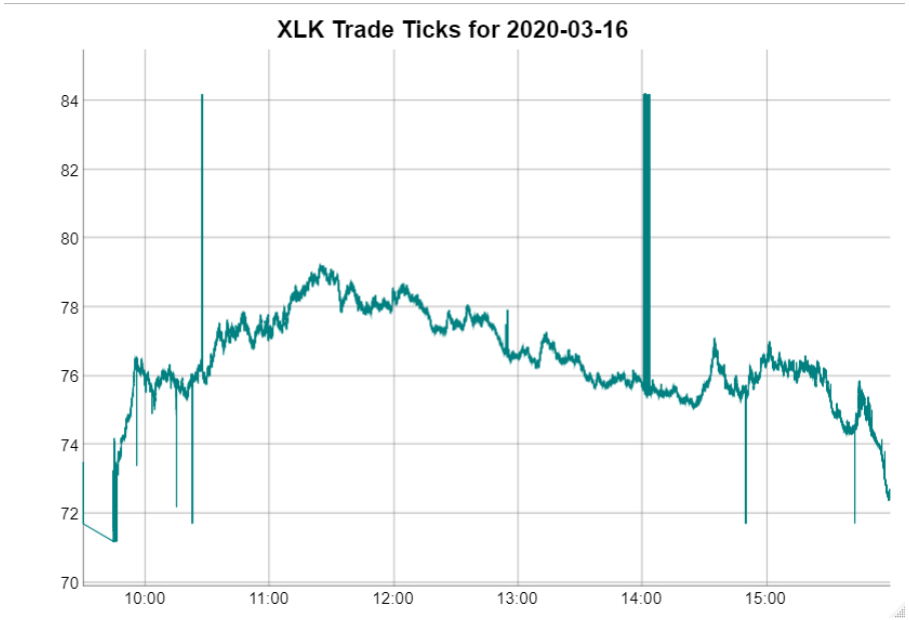
High frequency data contains *microstructure noise* in the form of *price jumps* and the *bid-ask bounce*.

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

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

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

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




```
> # Coerce trade ticks to xts series
> xtsv <- xts::xts(taq[, .(price, volume)], taq$index)
> colnames(xtsv) <- paste(symbol, c("Close", "Volume"), sep=".")
> save(xtsv, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_03
> # save(xtsv, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_(
> # Plot dygraph
> dygraphs::dygraph(xtsv$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16")
> # Plot in x11 window
> x11(width=6, height=5)
> quantmod::chart_Series(x=xtsv$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16")
```

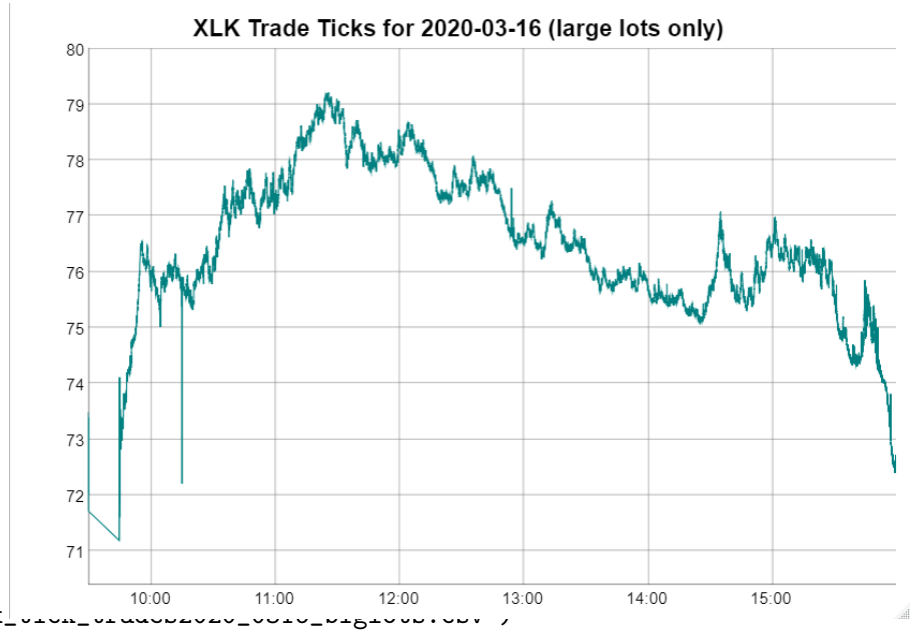
Removing Odd Lot Trades From *TAQ* Data

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

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

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

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




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

Aggregating *TAQ* Data to *OHLC*

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

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

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

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

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


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

draft: Trade and Quote (*TAQ*) Data

Trade and Quote (*TAQ*) Data
The NYSE TAQ set is 'consolidated' meaning that the trades and quotes come from many exchanges, including the NASDAQ system. TAQ has all trades and quotes from the Consolidated Tape / Consolidated Quote, which includes data from the regional exchanges. (See https://www.tradearca.com/marketdata/intermarket.asp for a brief discussion of CTA/CQ and OTC/UTP.) https://wrds-www.wharton.upenn.edupages/support/research-wrds/research-guides/wrds-taq-faqs/

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

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

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

Each row of *TAQ* data contains both the quote and trade prices, and the corresponding quote size or trade volume.

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

```
> # Load package HighFreq
> library(HighFreq)
> # Or load the high frequency data file directly:
> # symbolv <- load("/Users/jerzy/Develop/R/HighFreq/data/hf_data.RD
> head(HighFreq::SPY_TAQ)
                    Bid.Price Bid.Size Ask.Price Ask.Size Trade.Pric
2014-05-02 00:00:01       188        1       189       15          N
2014-05-02 08:00:01       188        1       189        5         18
2014-05-02 08:00:02       189        1       189        5          N
2014-05-02 08:01:13       188        1       189        5          N
2014-05-02 08:01:29       188        1       189        5          N
2014-05-02 08:01:52       189        2       189        5          N
> head(HighFreq::SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2008-01-02 09:31:00      147      147     147       147     591203
2008-01-02 09:32:00      147      147     147       147     385457
2008-01-02 09:33:00      147      147     147       147     343700
2008-01-02 09:34:00      147      147     147       147     863418
2008-01-02 09:35:00      147      147     147       147     457500
2008-01-02 09:36:00      147      147     147       147     416708
> tail(HighFreq::SPY)
                    SPY.Open SPY.High SPY.Low SPY.Close SPY.Volume
2014-05-19 15:56:00      189      189     189       189     321321
2014-05-19 15:57:00      189      189     189       189     299474
2014-05-19 15:58:00      189      189     189       189     261357
2014-05-19 15:59:00      189      189     189       189     435886
2014-05-19 16:00:00      189      189     189       189    1824185
2014-05-19 16:01:00      189      189     189       189      63920
```

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

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

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

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

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

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

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

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

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

Plotting High Frequency *OHLC* Data

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

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

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

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

Package *HighFreq* for Managing High Frequency Data

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

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

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

draft: Package HighFreq for Managing High Frequency Data

Package HighFreq contains functions for managing high frequency *TAQ* and *OHLC* market data:

- reading and writing data from files,
- managing time zones and alligning indices,
- chaining and joining time series,
- scrubbing bad data points,
- converting *TAQ* data to *OHLC* format,
- aggregating data to lower frequency,

HighFreq is inspired by the package highfrequency, and follows many of its conventions.

HighFreq depends on packages xts, quantmod, lubridate, and caTools.

The function scrub_agg() scrubs a single day of *TAQ* data, aggregates it, and converts it to *OHLC* format.

The function save_scrub_agg() loads, scrubs, aggregates, and binds multiple days of *TAQ* data for a single symbol, and saves the *OHLC* time series to a single *.RData file.

```
> # install package HighFreq from github
> install.packages("devtools")
> library(devtools)
> install_github(repo="algoquant/HighFreq")
> # Load package HighFreq
> library(HighFreq)
> # set data directories
> data_dir <- "/Users/jerzy/Develop/data/hfreq/src/"
> output_dir <- "/Users/jerzy/Develop/data/hfreq/scrub/"
> # Define symbol
> symbol <- "SPY"
> # Load a single day of TAQ data
> symbol <- load(file.path(data_dir, paste0(symbol, "/2014.05.02.",
> # scrub, aggregate single day of TAQ data to OHLC
> ohlc_data <- scrub_agg(taq_data=get(symbol))
> # Aggregate TAQ data for symbol, save to file
> save_scrub_agg(symbol,
+          data_dir=data_dir,
+          output_dir=output_dir,
+          period="minutes")
```

Datasets in Package *HighFreq*

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

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

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

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

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

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

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

Distribution of High Frequency Returns

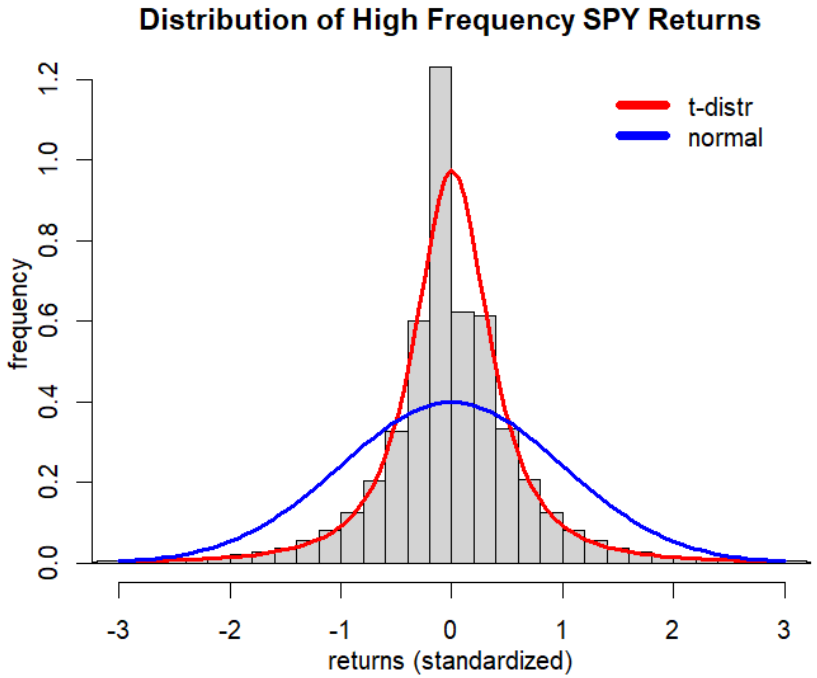
High frequency returns exhibit *large negative skewness* and *very large kurtosis* (leptokurtosis), or fat tails.

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

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

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

```
> # Calculate SPY percentage returns
> ohlc <- HighFreq::SPY
> nrows <- NROW(ohlc)
> closep <- log(quantmod::Cl(ohlc))
> retsp <- rutils::diffit(closep)
> colnames(retsp) <- "SPY"
> # Standardize raw returns to make later comparisons
> retsp <- (retsp - mean(retsp))/sd(retsp)
> # Calculate moments and perform normality test
> sapply(c(var=2, skew=3, kurt=4), function(x) sum(retsp^x)/nrows)
> tseries::jarque.bera.test(retsp)
> # Fit SPY returns using MASS::fitdistr()
> optiml <- MASS::fitdistr(retsp, densfun="t", df=2)
> loc <- optiml$estimate[1]
> scalev <- optiml$estimate[2]
```




```
> # Plot histogram of SPY returns
> histp <- hist(retsp, col="lightgrey", mgp=c(2, 1, 0),
+   xlab="returns (standardized)", ylab="frequency", xlim=c(-3, 3),
+   breaks=1e3, freq=FALSE, main="Distribution of High Frequency SPY
> # lines(density(retsp, bw=0.2), lwd=3, col="blue")
> # Plot t-distribution function
> curve(expr=dt((x-loc)/scalev, df=2)/scalev,
+ type="l", lwd=3, col="red", add=TRUE)
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(retsp),
+   sd=sd(retsp)), add=TRUE, lwd=3, col="blue")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("t-distr", "normal"), y.intersp=0.5,
+   lwd=6, lty=1, col=c("red", "blue"))
```

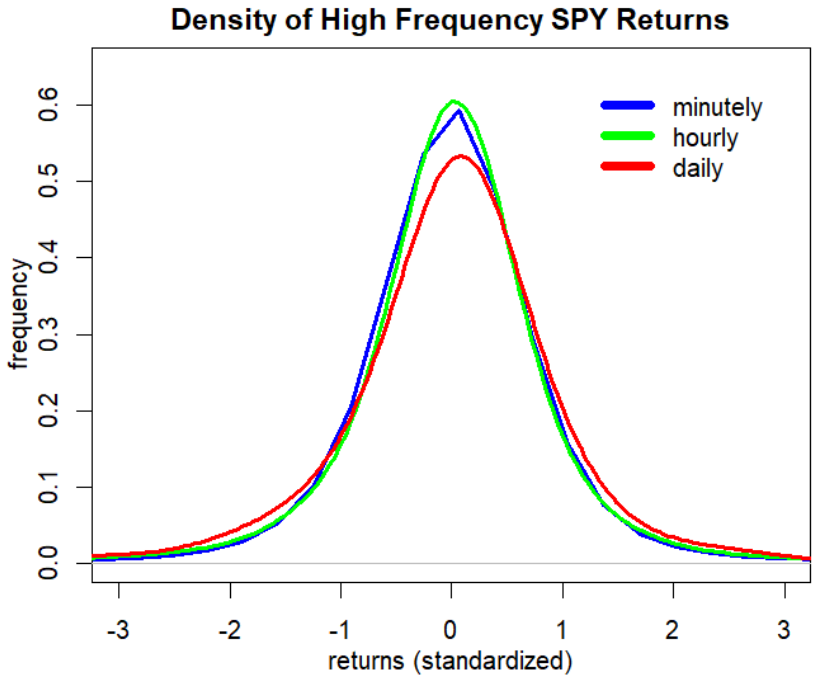
Distribution of Aggregated High Frequency Returns

The distribution of returns depends on the sampling frequency.

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

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

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



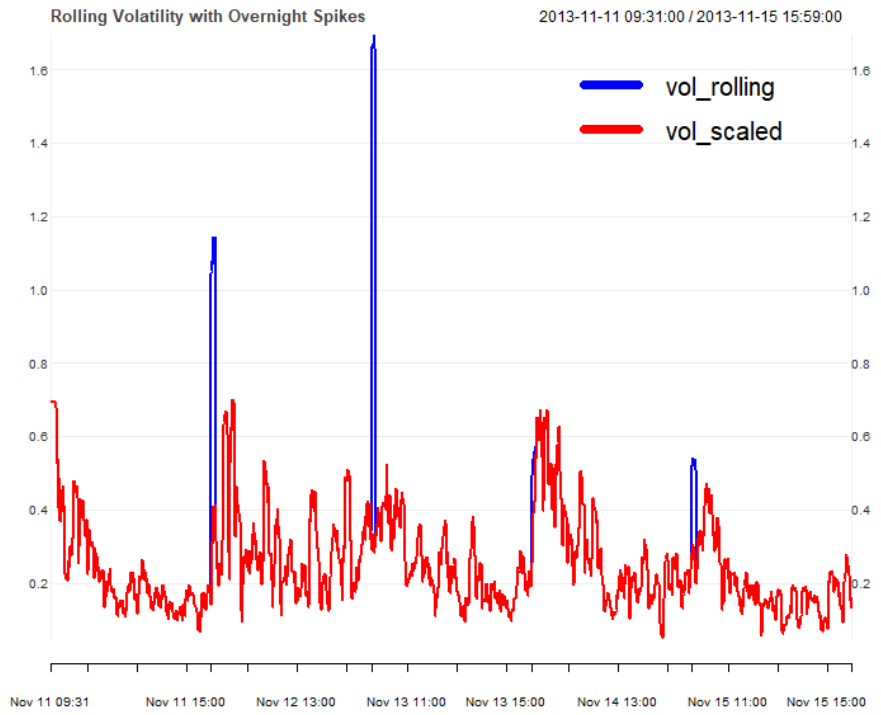

```
> # Plot densities of SPY returns
> plot(density(retsp, bw=0.4), xlim=c(-3, 3),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of High Frequency SPY Returns")
> lines(density(retsh, bw=0.4), lwd=3, col="green")
> lines(density(retsd, bw=0.4), lwd=3, col="red")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("minutely", "hourly", "daily"), y.intersp=0.5,
+   lwd=6, lty=1, col=c("blue", "green", "red"))
```

Estimating Rolling Volatility of High Frequency Returns

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

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

```
> # Calculate rolling volatility of SPY returns
> ret2013 <- retsp["2013-11-11/2013-11-15"]
> # Calculate rolling volatility
> look_back <- 11
> endp <- seq_along(ret2013)
> startp <- c(rep_len(1, look_back),
+   endp[1:(NROW(endp)-look_back)])
> endp[endp < look_back] <- look_back
> vol_rolling <- sapply(seq_along(endp),
+   function(it) sd(ret2013[startp[it]:endp[it]]))
> vol_rolling <- xts::xts(vol_rolling, zoo::index(ret2013))
> # Extract time intervals of SPY returns
> indeks <- c(60, diff(xts::.index(ret2013)))
> head(indeks)
> table(indeks)
> # Scale SPY returns by time intervals
> ret2013 <- 60*ret2013/indeks
> # Calculate scaled rolling volatility
> vol_scaled <- sapply(seq_along(endp),
+   function(it) sd(ret2013[startp[it]:endp[it]]))
> vol_rolling <- cbind(vol_rolling, vol_scaled)
> vol_rolling <- na.omit(vol_rolling)
> sum(is.na(vol_rolling))
> sapply(vol_rolling, range)
```




```
> # Plot rolling volatility
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue", "red")
> chart_Series(vol_rolling, theme=plot_theme,
+       name="Rolling Volatility with Overnight Spikes")
> legend("topright", legend=colnames(vol_rolling),
+   inset=0.1, bg="white", lty=1, lwd=6, y.intersp=0.5,
+   col=plot_theme$col$line.col, bty="n")
```

The Bid-ask Bounce of High Frequency Prices

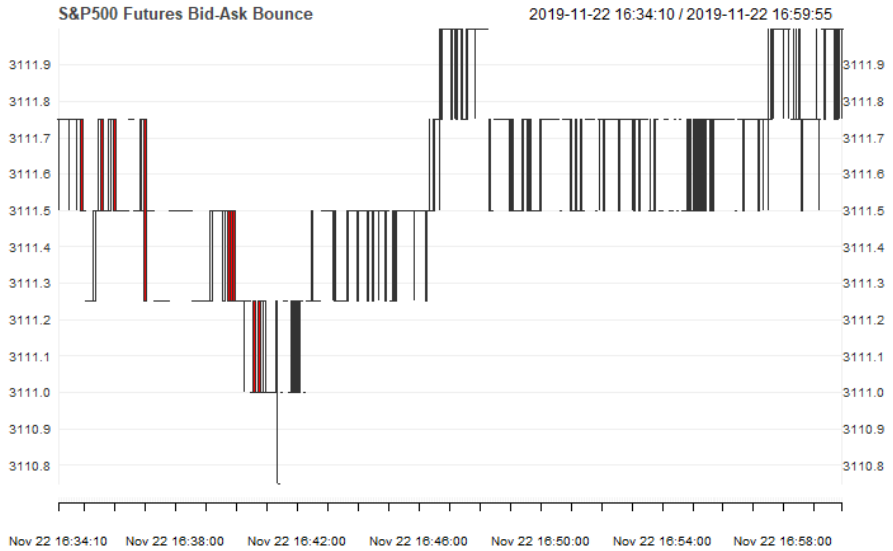
The *bid-ask bounce* is the bouncing of traded prices between the bid and ask prices.

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

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

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

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



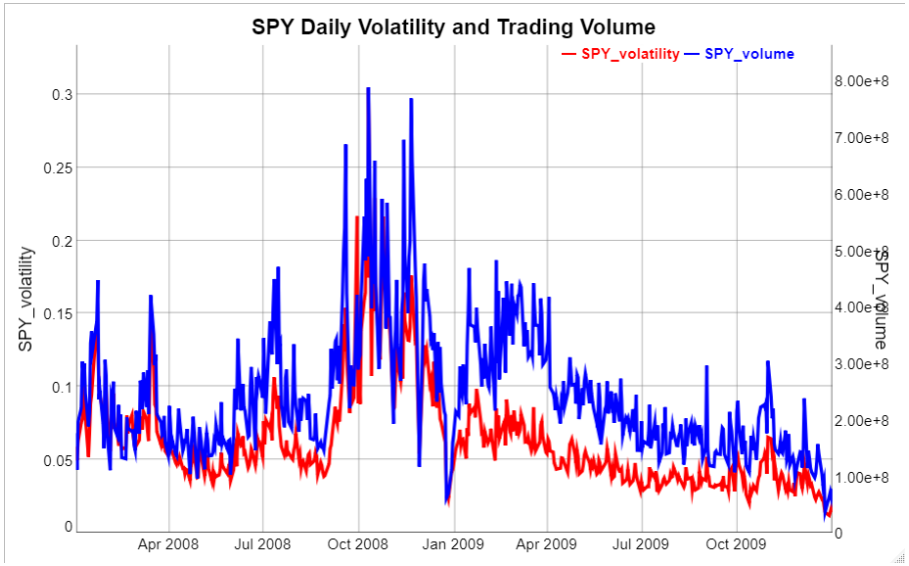
Daily Volume and Volatility of High Frequency Returns

Trading volumes typically rise together with market price volatility.

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

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

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

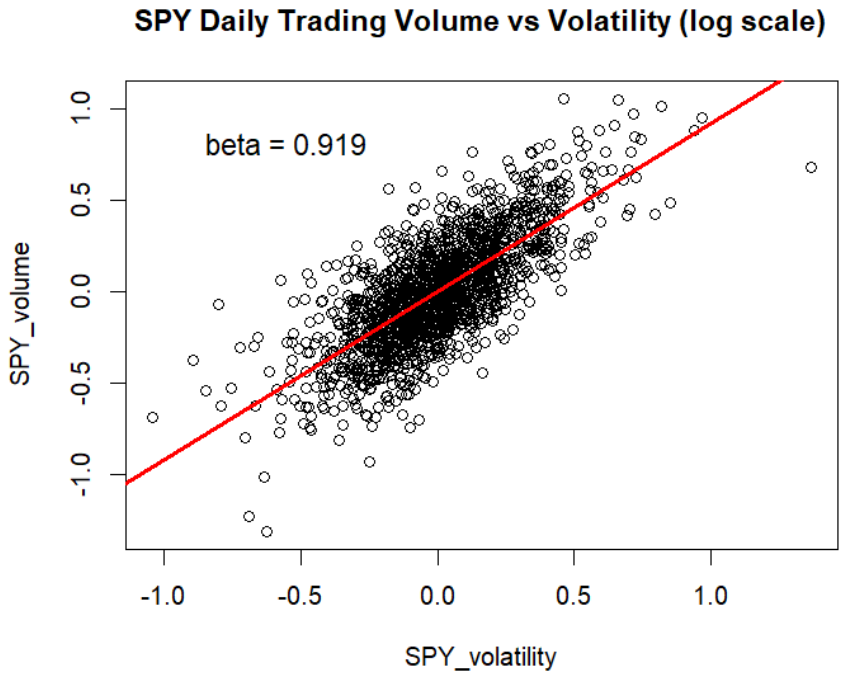


Beta of Volume vs Volatility of High Frequency Returns

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

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

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

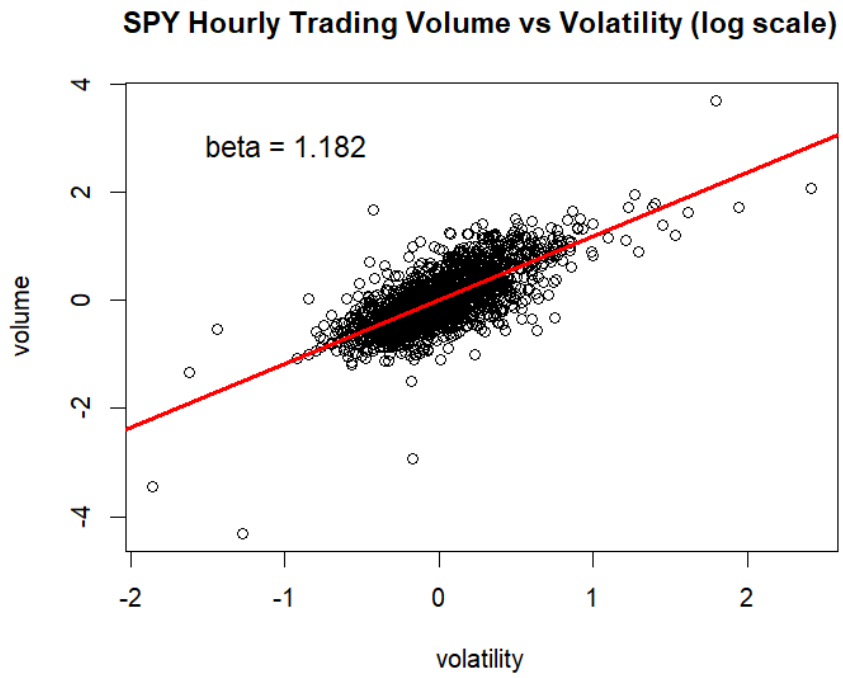



```
> # Regress log of daily volume vs volatility
> datav <- log(cbind(volume_daily, vol_daily))
> colnamev <- colnames(datav)
> dframe <- as.data.frame(datav)
> formulav <- as.formula(paste(colnamev, collapse="~"))
> model <- lm(formulav, data=dframe)
> # Durbin-Watson test for autocorrelation of residuals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

Beta of Hourly Trading Volume vs Volatility

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

```
> # 60 minutes of data in look_back interval
> look_back <- 60
> vol2013 <- volumes["2013"]
> ret2013 <- retsp["2013"]
> # Define end points with beginning stub
> nrows <- NROW(ret2013)
> nagg <- nrows %/% look_back
> endp <- nrows-look_back*nagg + (0:nagg)*look_back
> startp <- c(1, endp[1:(NROW(endp)-1)])
> # Calculate SPY volatility and volume
> datav <- sapply(seq_along(endp), function(it) {
+   point_s <- startp[it]:endp[it]
+   c(volume=sum(vol2013[point_s]),
+     volatility=sd(ret2013[point_s]))
+ })  # end sapply
> datav <- t(datav)
> datav <- rutils::diffit(log(datav))
> dframe <- as.data.frame(datav)
```




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

High Frequency Returns in Trading Time

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

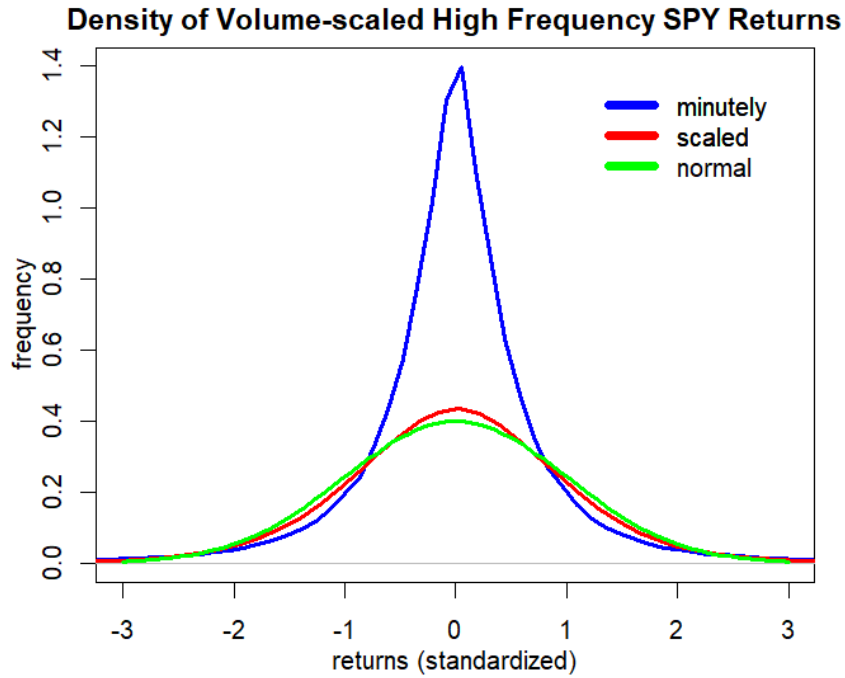
The time-dependent volatility of high frequency returns (*heteroskedasticity*) produces their *leptokurtosis* (large kurtosis, or fat tails).

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

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

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

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




```
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> # Plot densities of SPY returns
> plot(density(retsp), xlim=c(-3, 3),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of Volume-scaled High Frequency SPY Returns")
> lines(density(rets_scaled, bw=0.4), lwd=3, col="red")
> curve(expr=dnorm, add=TRUE, lwd=3, col="green")
> # Add legend
> legend("topright", inset=0.05, bty="n", y.intersp=0.5,
+   leg=c("minutely", "scaled", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Autocorrelations of High Frequency Returns

The *Ljung-Box* test, tests if the autocorrelations of a time series are *statistically significant*.

The *null hypothesis* of the *Ljung-Box* test is that the autocorrelations are equal to zero.

The *Ljung-Box* statistic is small for time series that have *statistically insignificant* autocorrelations.

The function `Box.test()` calculates the *Ljung-Box* test and returns the test statistic and its p-value.

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

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

SPY returns aggregated to longer time intervals are less autocorrelated.

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

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

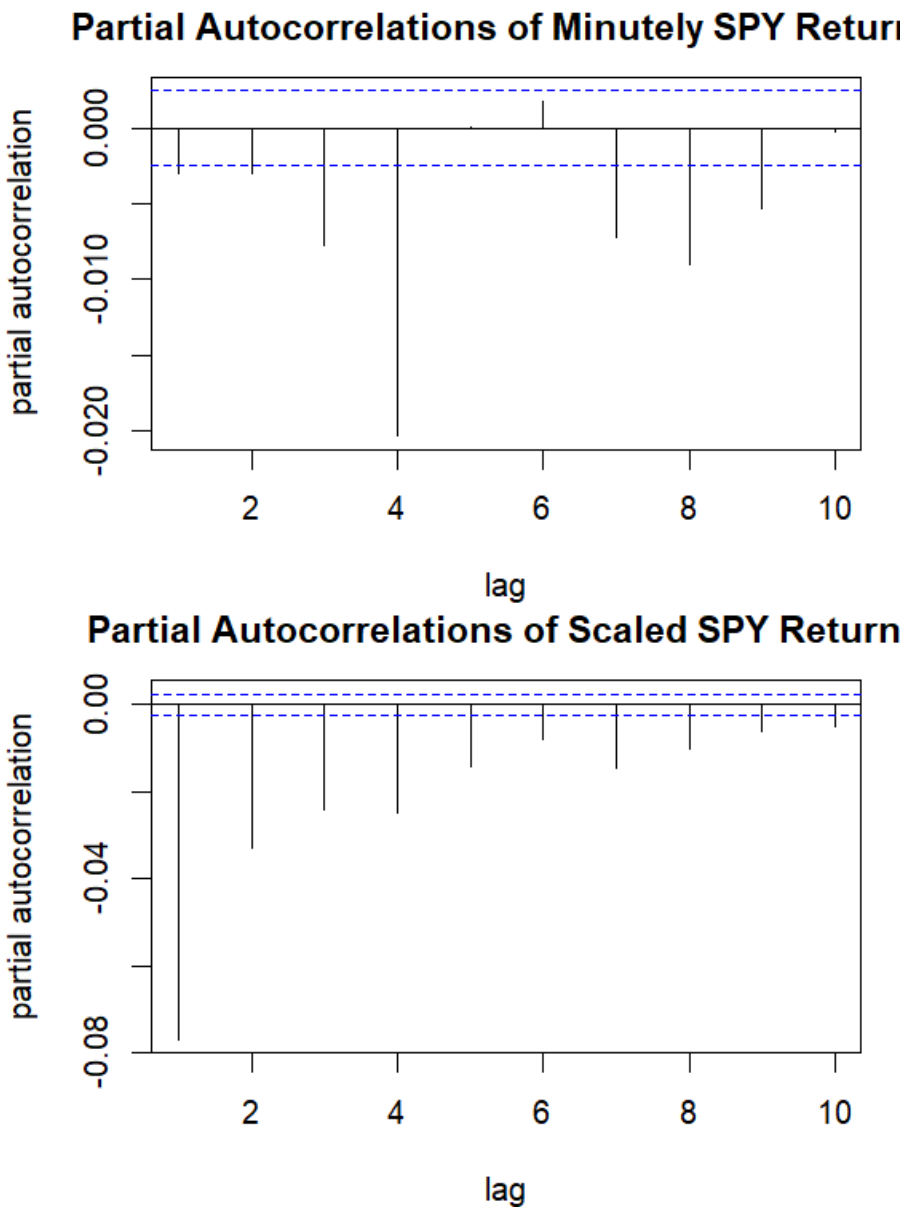
As the returns are aggregated to a lower periodicity, they become less autocorrelated, with daily returns having almost insignificant autocorrelations.

Partial Autocorrelations of High Frequency Returns

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

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

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



Market Liquidity, Trading Volume and Volatility

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

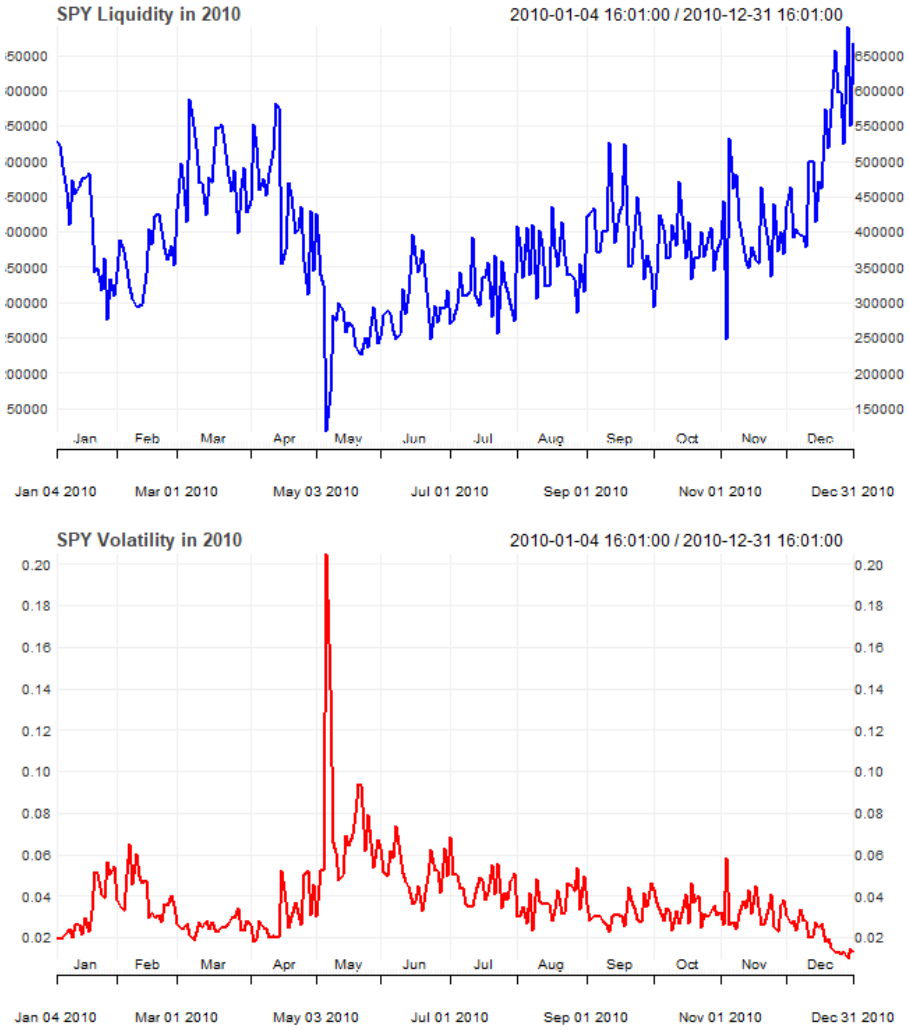
Market liquidity $\mathcal{L}$ is proportional to the square root of the *trading volume* v divided by the price volatility σ:

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

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

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

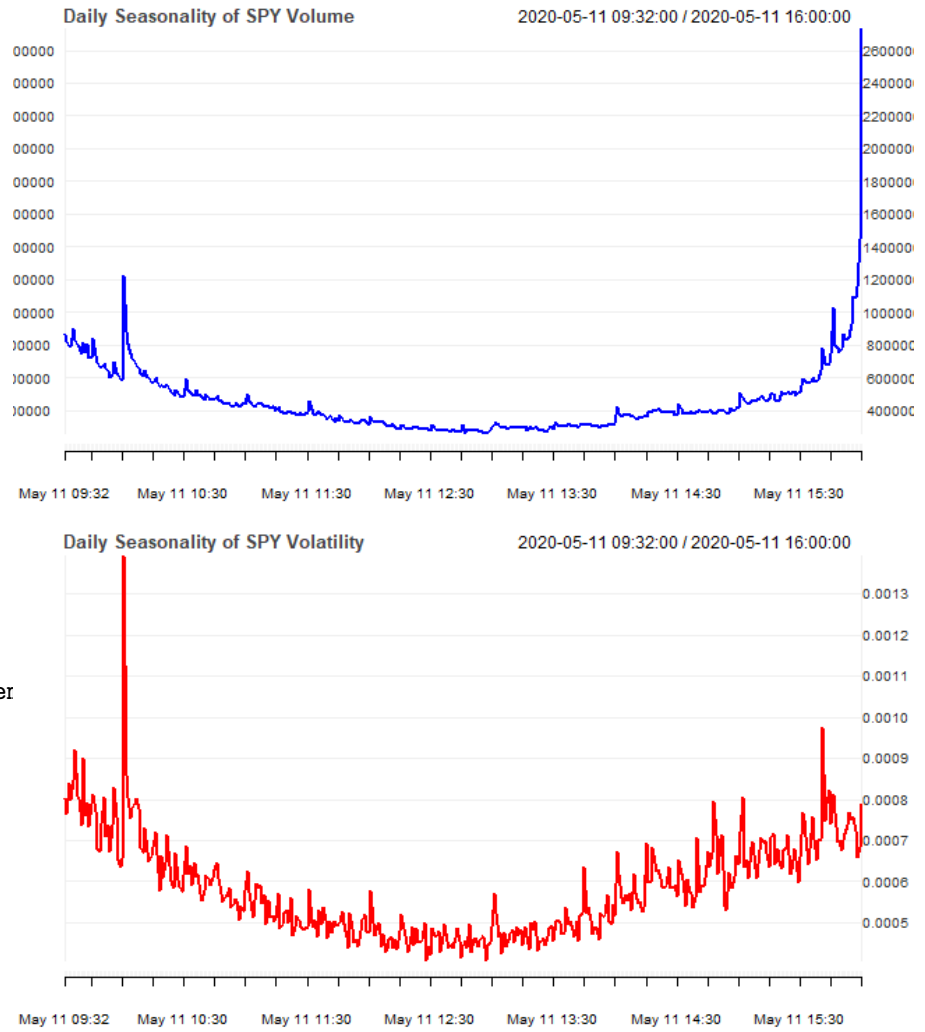
```
> # Calculate market illiquidity
> liquidi_ty <- sqrt(volume_daily)/vol_daily
> # Plot market illiquidity
> x11(width=6, height=7) ; par(mfrow=c(2, 1))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> chart_Series(liquidi_ty["2010"], theme=plot_theme,
+   name="SPY Liquidity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



Daily Seasonality of Volume and Volatility

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

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

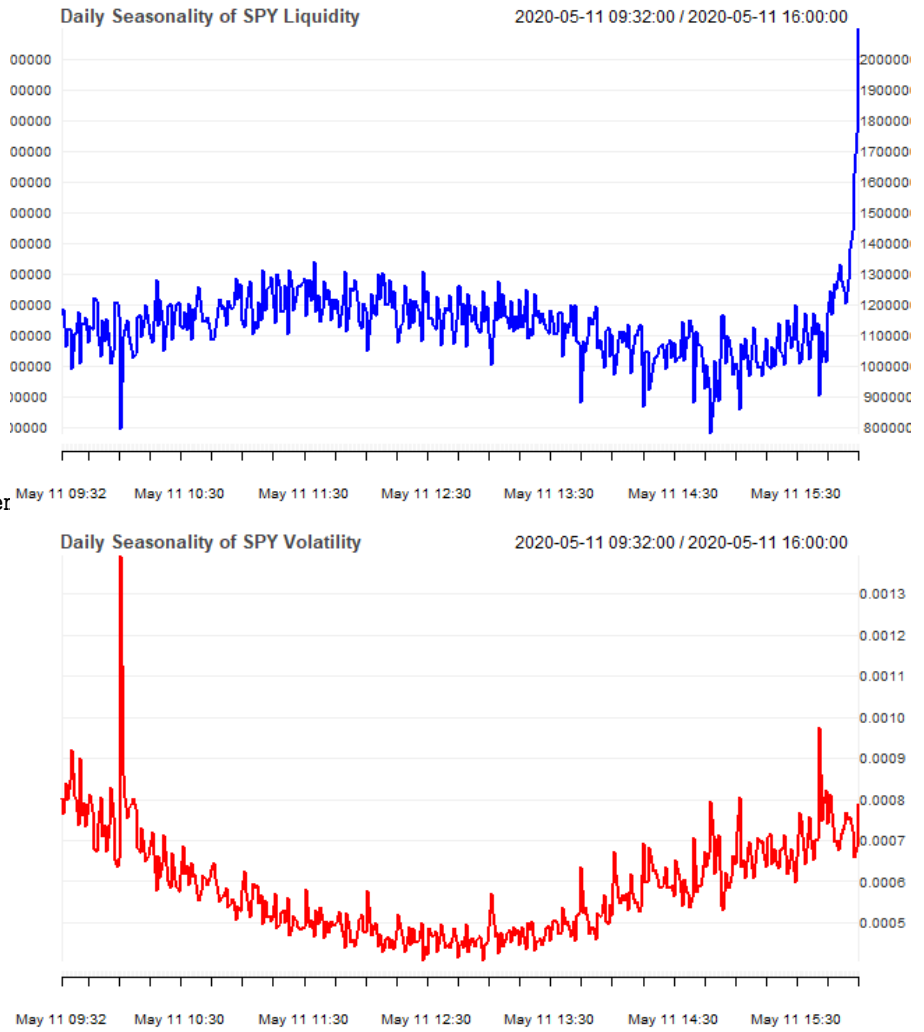


Daily Seasonality of Liquidity and Volatility

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

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

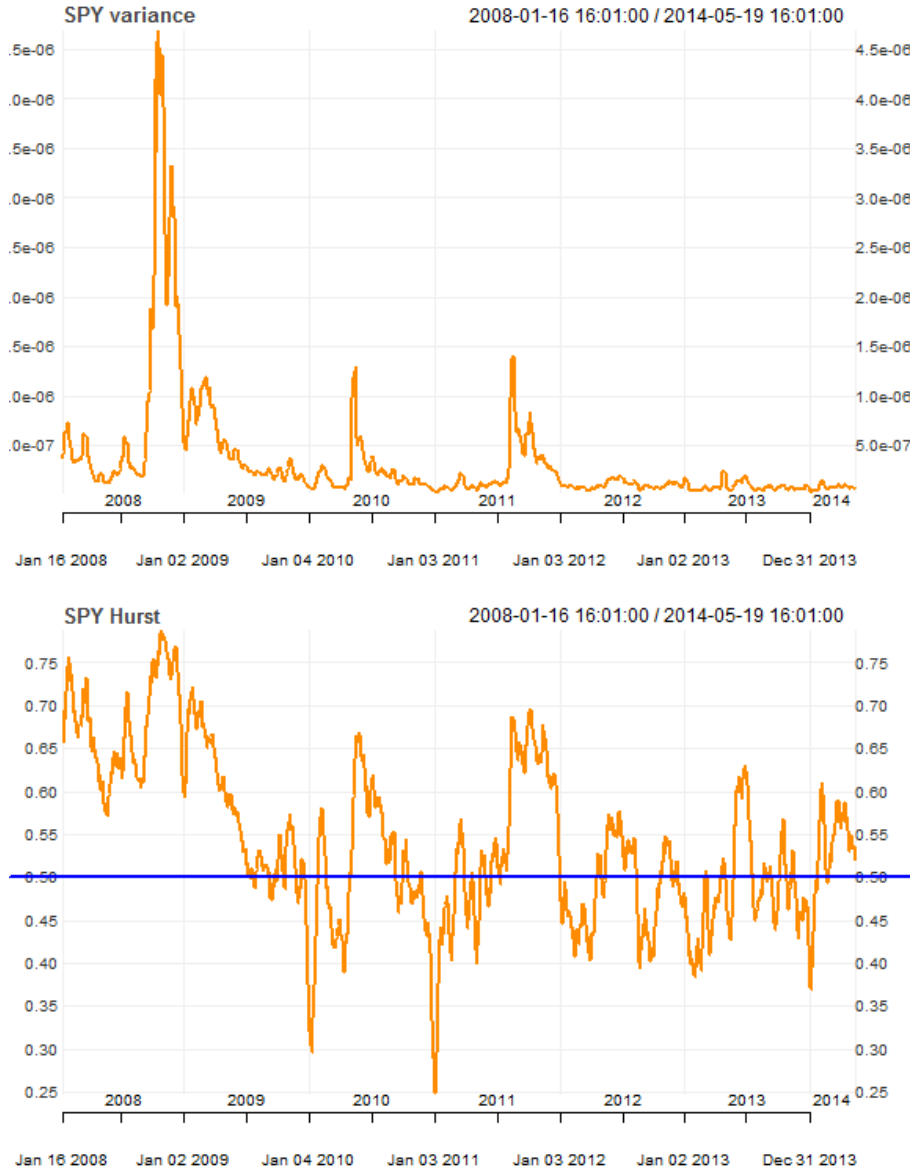
```
> # Calculate market liquidity
> liquidi_ty <- sqrt(volume_agg)/vol_agg
> # Plot daily seasonality of market liquidity
> x11(width=6, height=7) ; par(mfrow=c(2, 1))
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> chart_Series(liquidi_ty[c(-1, -NROW(liquidi_ty))], theme=plot_ther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



draft: Daily Volatility and Hurst Exponent

The Hurst exponent typically moves higher with higher market price volatility, and is above 0.5 with high volatility.

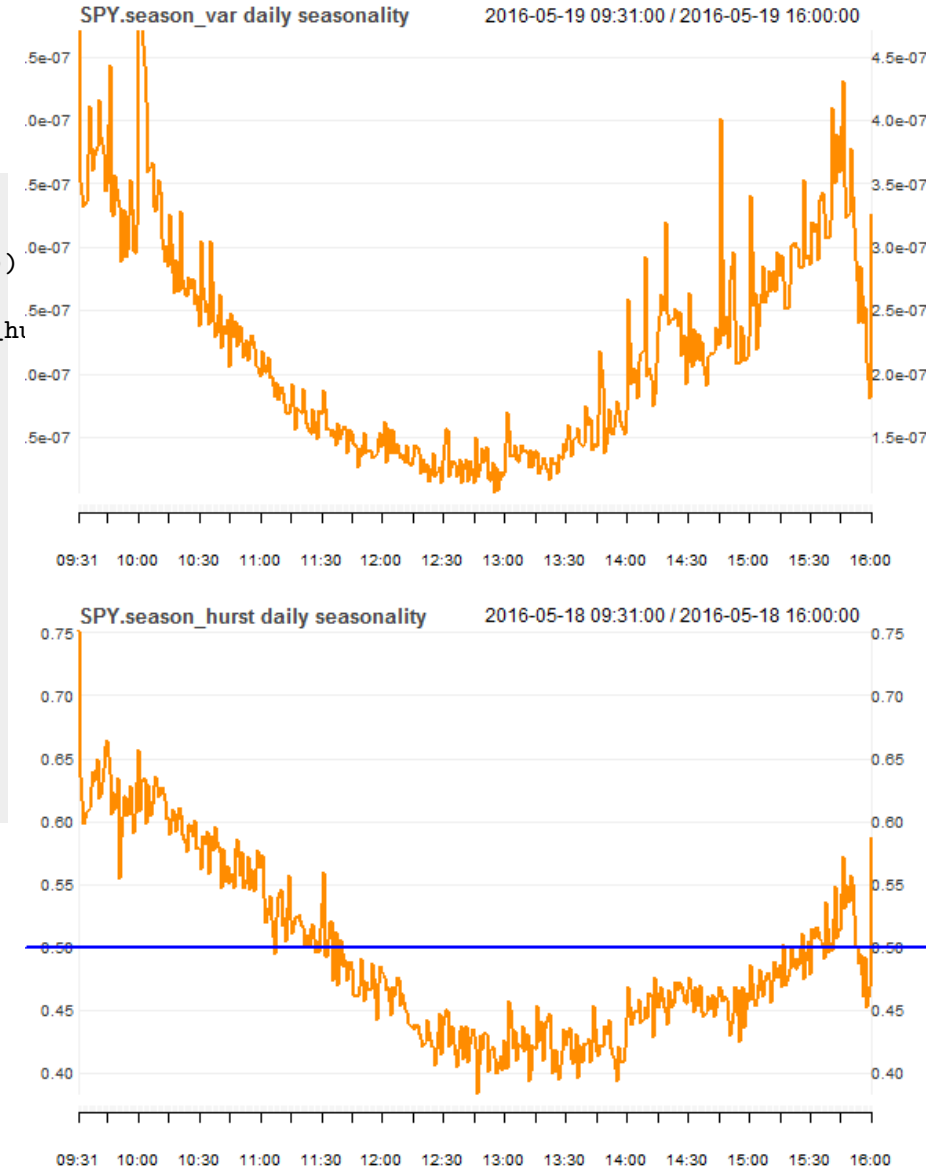
```
> chart_Series(roll_sum(vol_daily, 10)[-(1:10)]/10,
+        name=paste(symbol, "variance"))
> chart_Series(roll_sum(hurst_daily, 10)[-(1:10)]/10,
+        name=paste(symbol, "Hurst"))
> abline(h=0.5, col="blue", lwd=2)
```



draft: Daily Seasonality of Hurst Exponent and Volatility

The Hurst exponent typically moves higher with higher market price volatility, and is above 0.5 with high volatility.

```
> # Daily seasonality of Hurst exponent
> interval <- "2013"
> season_hurst <- season_ality(hurst_ohlc(ohlc=SPY[interval, 1:4]))
> season_hurst <- season_hurst[-(nrow(season_hurst))]
> colnames(season_hurst) <- paste0(colname(get(symbol)), ".season_hu
> plot_theme <- chart_theme()
> plot_theme$format.labels <- "%H:%M"
> chobj <- chart_Series(x=season_hurst,
+   name=paste(colnames(season_hurst),
+   "daily seasonality"), theme=plot_theme,
+   plot=FALSE)
> ylim <- chobj$get_ylim()
> ylim[[2]] <- structure(c(ylim[[2]][1],
+         ylim[[2]][2]), fixed=TRUE)
> chobj$set_ylim(ylim)
> plot(chobj)
> abline(h=0.5, col="blue", lwd=2)
> # Daily seasonality of volatility
> season_var <- season_ality(vol_ohlc(ohlc=SPY))
```



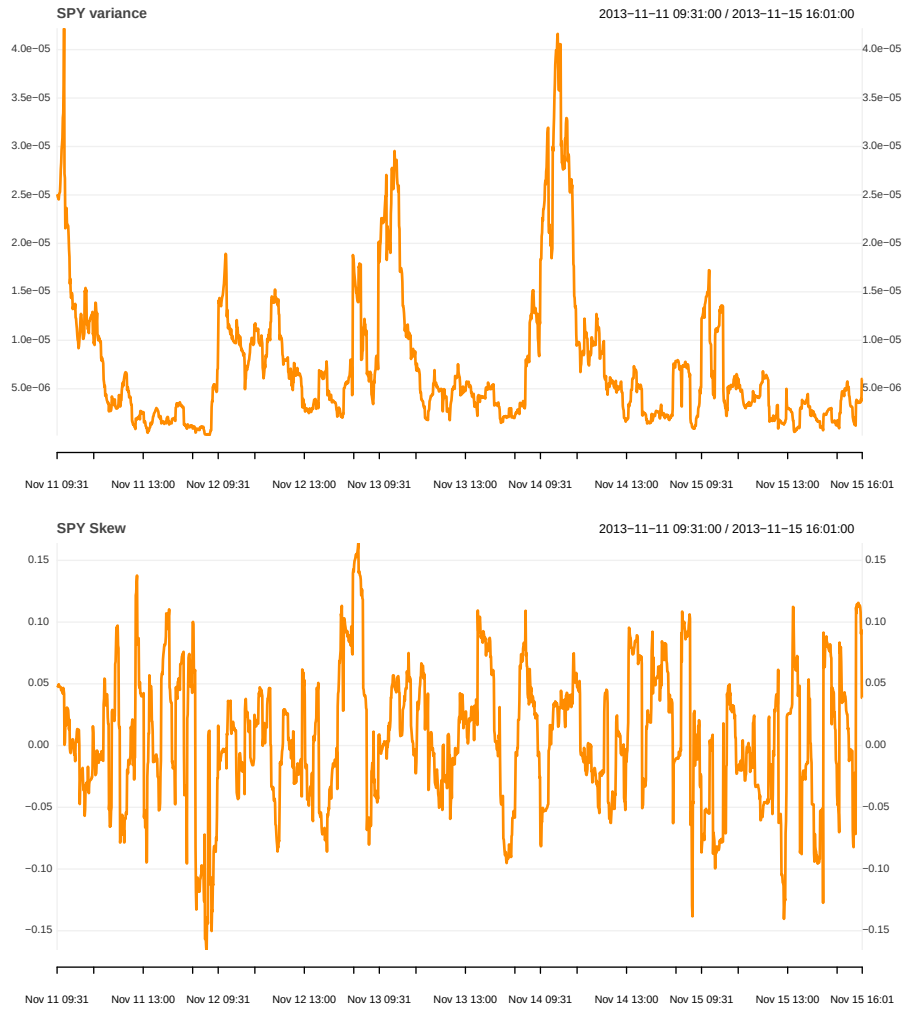
draft: Estimating Skew From *OHLC* Data

The function `skew_ohlc()` from package `HighFreq` calculates a skew-like indicator:

$$s^3 = \frac{1}{n}\sum_{i=1}^{n}((H_i - O_i)(H_i - C_i)(H_i - 0.5(O_i + C_i)) + (L_i - O_i)(L_i - C_i)(L_i - 0.5(O_i + C_i))$$

The function `roll_agg_ohlc()` aggregates rolling, volume weighted moment estimators.

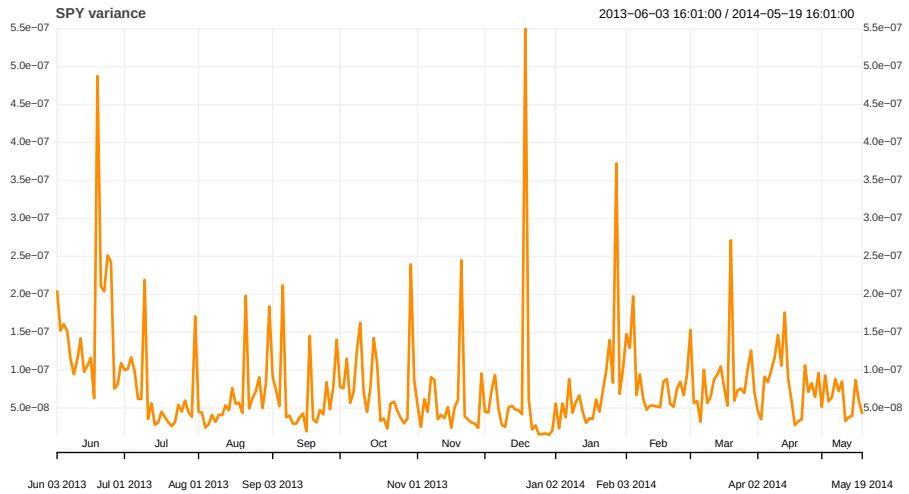
```
> library(rutils)  # Load package rutils
> # Rolling variance
> var_iance <-
+   roll_agg_ohlc(ohlc=SPY, agg_fun="vol_ohlc")
> # Rolling skew
> skew <-
+   roll_agg_ohlc(ohlc=SPY, agg_fun="skew_ohlc")
> skew <- skew/(var_iance)^(1.5)
> skew[1, ] <- 0
> skew <- na.locf(skew)
> interval <- "2013-11-11/2013-11-15"
> chart_Series(var_iance[interval],
+       name=paste(symbol, "variance"))
> chart_Series(skew[interval],
+       name=paste(symbol, "Skew"),
+       ylim=c(-1, 1))
```

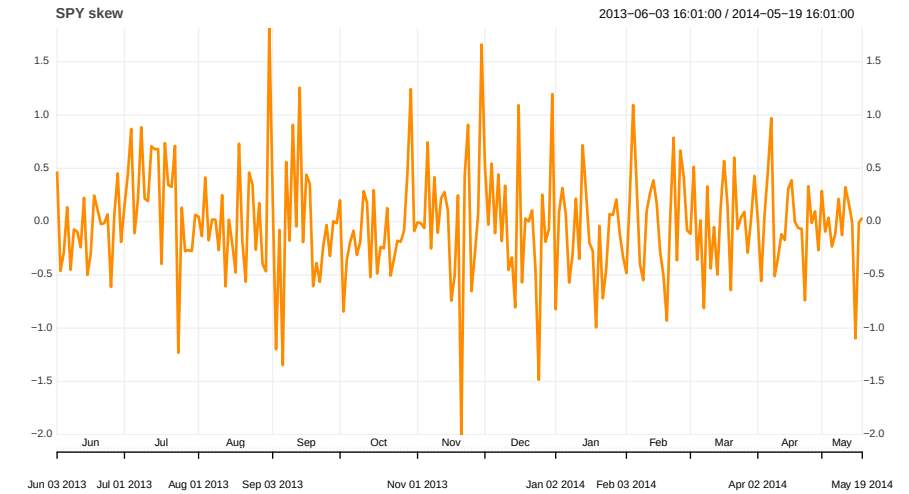


draft: Daily Volatility and Skew From *OHLC* Data

The function `agg_ohlc()` calculates a statistical estimator over an *OHLC* time series.

```
> # Daily variance and skew
> vol_daily <- xts::apply.daily(x=HighFreq::SPY, FUN=agg_ohlc,
+                   agg_fun="vol_ohlc")
> colnames(vol_daily) <- paste0(symbol, ".var")
> daily_skew <- xts::apply.daily(x=HighFreq::SPY, FUN=agg_ohlc,
+                   agg_fun="skew_ohlc")
> daily_skew <- daily_skew/(vol_daily)^(1.5)
> colnames(daily_skew) <- paste0(symbol, ".skew")
> interval <- "2013-06-01/"
> chart_Series(vol_daily[interval],
+        name=paste(symbol, "variance"))
> chart_Series(daily_skew[interval],
+        name=paste(symbol, "skew"))
```

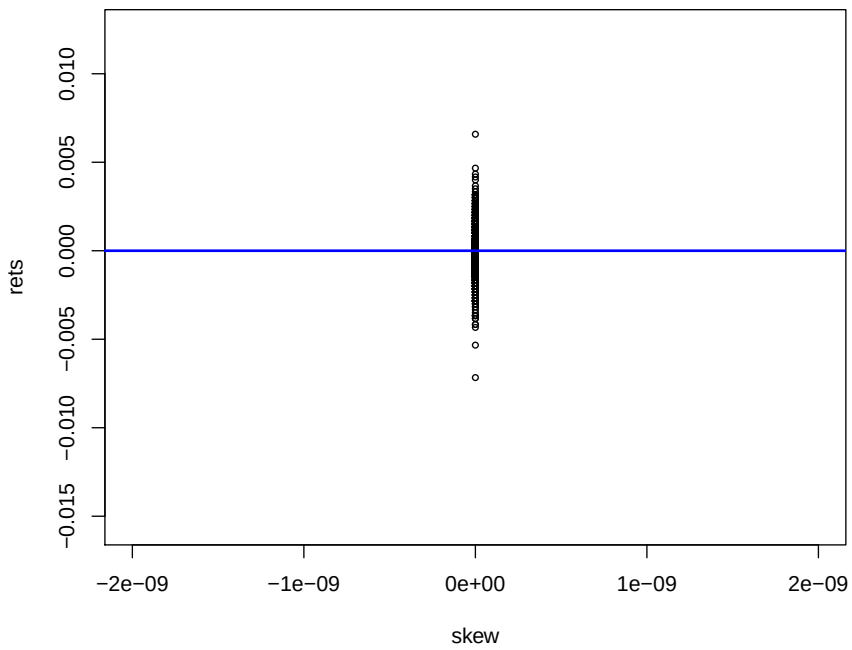




draft: Regression of Skews Versus Returns

A regression of lagged skews versus returns appears to be statistically significant, especially in periods of high volatility during the financial crisis of 2008-09.

```
> retsp <- calc_rets(xts_data=SPY)
> skew <- skew_ohlc(log_ohlc=log(SPY[, -5]))
> colnames(skew) <- paste0(symbol, ".skew")
> lag_skew <- rutils::lag_it(skew)
> lag_skew[1, ] <- 0
> datav <- cbind(retsp[, 1], sign(lag_skew))
> formulav <- as.formula(paste(colnames(datav)[1],
+     paste(paste(colnames(datav)[-1],
+       collapse=" + "), "- 1"), sep="~"))
> formulav
> model <- lm(formulav, data=datav)
> summary(model)$coef
> summary(lm(formulav, data=datav["/2011-01-01"]))$coef
> summary(lm(formulav, data=datav["2011-01-01/"]))$coef
```




```
> interval <- "2013-12-01/"
> plot(formulav, data=datav[interval],
+      xlim=c(-2e-09, 2e-09),
+      cex=0.6, xlab="skew", ylab="rets")
> abline(model, col="blue", lwd=2)
```

draft: Contrarian Strategy Using Skew Oscillator

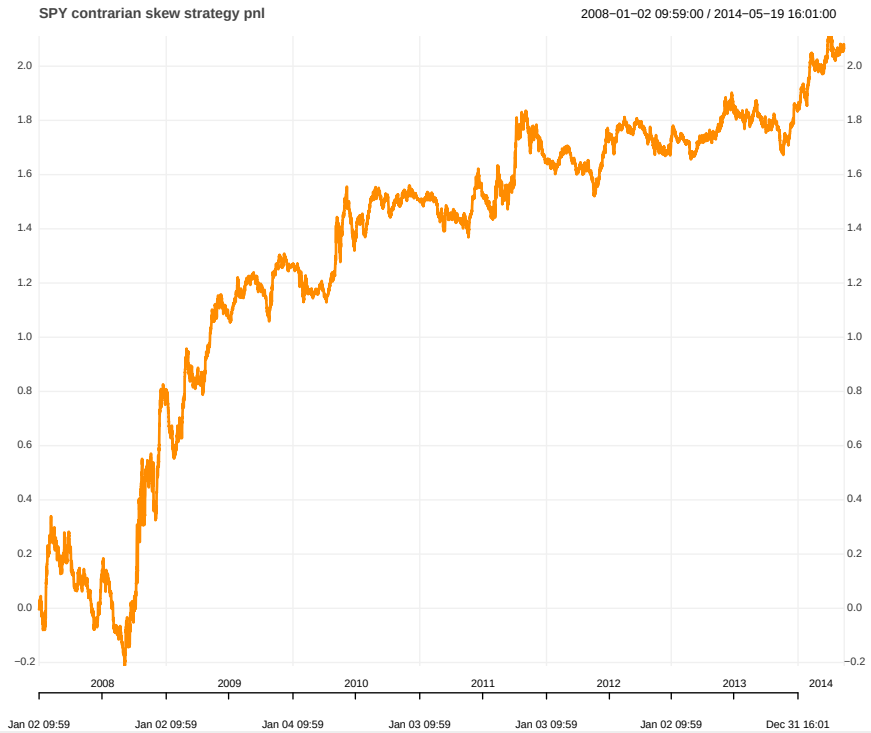
The contrarian skew trading strategy involves long or short positions in a single unit of stock, that is opposite to the sign of the skew.

Skew is calculated over one-minute bars, and trades are executed in the following period.

The contrarian strategy shows good hypothetical performance before transaction costs, and since it's a liquidity providing strategy, should have very low transaction costs.

The contrarian strategy is hyperactive, trading almost 46% of the time in each period.

```
> # Lag the skew to get positions
> posit <- -sign(lag_skew)
> posit[1, ] <- 0
> # Cumulative PnL
> pnl <- cumsum(posit*retsp[, 1])
> # Calculate frequency of trades
> 50*sum(abs(sign(skew)-sign(lag_skew)))/nrow(skew)
> # Calculate transaction costs
> bid_offer <- 0.001  # 10 bps for liquid ETFs
> bid_offer*sum(abs(sign(skew)-sign(lag_skew)))
```




```
> chart_Series(pnl[endpoints(pnl, on="hours"), ],
+   name=paste(symbol, "contrarian skew strategy pnl"))
```

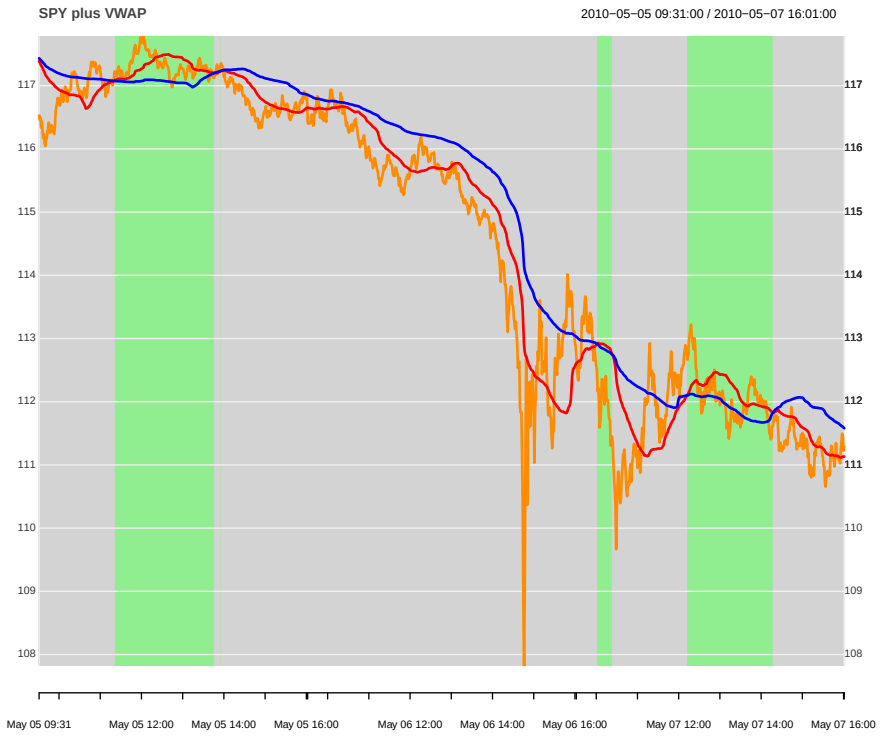
draft: Volume-Weighted Average Price Indicator

The Volume-Weighted Average Price (*VWAP*) is an indicator used for trend following strategies.

The fast-moving *VWAP* is calculated over a short look-back interval, while the slow-moving *VWAP* is calculated over a longer interval.

The trend following reverses direction when the fast-moving *VWAP* crosses the slow-moving one.

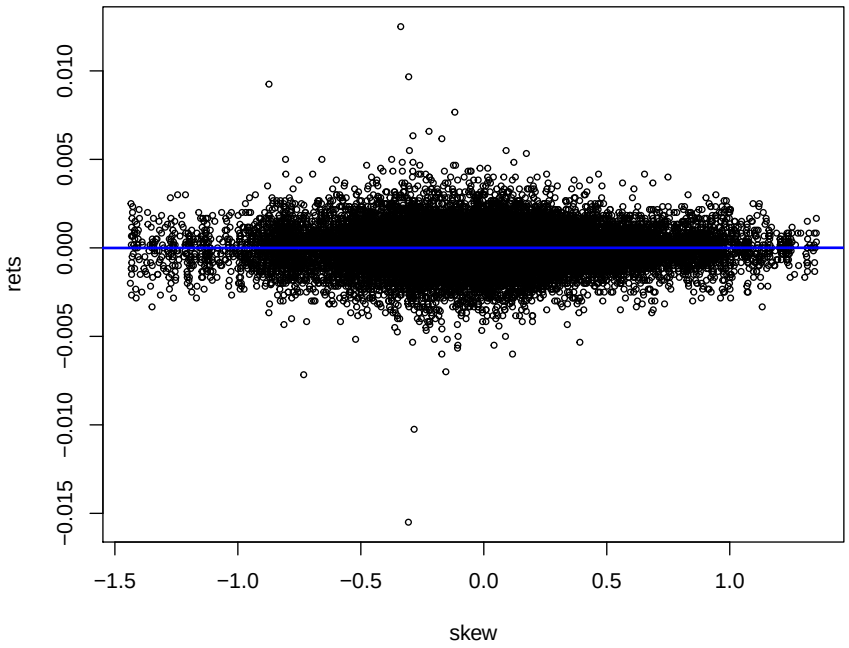
```
> vwap_short <- vwapv(xtsv=SPY, look_back=70)
> vwap_long <- vwapv(xtsv=SPY, look_back=225)
> vwap_diff <- vwap_short - vwap_long
> colnames(vwap_diff) <- paste0(symbol, ".vwap")
> interval <- "2010-05-05/2010-05-07"
> invisible(chart_Series(x=Cl(SPY[interval]),
+          name=paste(symbol, "plus VWAP")))
> invisible(add_TA(vwap_short[interval],
+    on=1, col="red", lwd=2))
> invisible(add_TA(vwap_long[interval],
+    on=1, col="blue", lwd=2))
> invisible(add_TA(vwap_diff[interval] > 0, on=-1,
+    col="lightgreen", border="lightgreen"))
> add_TA(vwap_diff[interval] < 0, on=-1,
+  col="lightgrey", border="lightgrey")
```



draft: Regression of *VWAP* Versus Returns

A regression of the *VWAP* indicator versus returns appears to be statistically significant.

```
> lag_vwap <- rutils::lag_it(vwap_diff)
> lag_vwap[1, ] <- 0
> datav <- cbind(retsp[, 1], sign(lag_vwap))
> formulav <- as.formula(paste(colnames(datav)[1],
+     paste(paste(colnames(datav)[-1],
+       collapse=" + "), "- 1"), sep="~"))
> formulav
> model <- lm(formulav, data=datav)
> summary(model)$coef
> summary(lm(formulav, data=datav["/2011-01-01"]))$coef
> summary(lm(formulav, data=datav["2011-01-01/"]))$coef
```




```
> interval <- "2013-12-01/"
> plot(formulav, data=cbind(retsp[, 1], lag_vwap)[interval],
+      cex=0.6, xlab="skew", ylab="rets")
> abline(model, col="blue", lwd=2)
```

draft: Trend Following Strategy Using *VWAP*

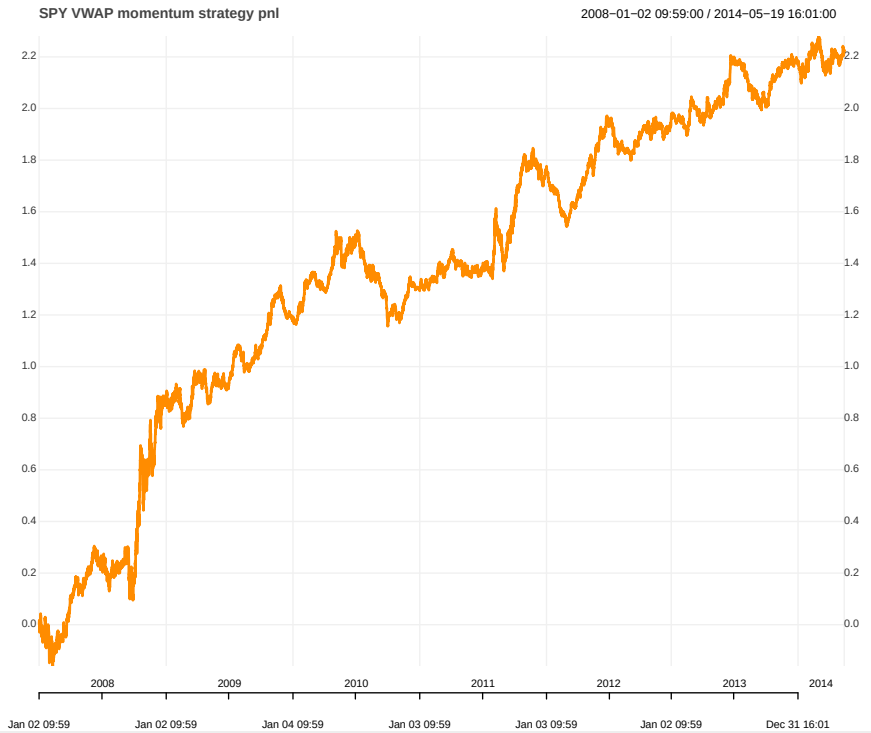
The trend following trading strategy involves long or short positions in a single unit of stock, that is equal to the sign of the *VWAP* indicator.

The *VWAP* indicator is calculated over one-minute bars, and trades are executed in the following period.

The trend following strategy shows good hypothetical performance before transaction costs.

The trend following strategy is infrequent, trading only 0.56% of the time in each period.

```
> # Cumulative PnL
> pnl <- cumsum(sign(lag_vwap)*retsp[, 1])
> # Calculate frequency of trades
> 50*sum(abs(sign(vwap_diff)-sign(lag_vwap)))/nrow(vwap_diff)
> # Calculate transaction costs
> bid_offer <- 0.001  # 10 bps for liquid ETFs
> bid_offer*sum(abs(sign(vwap_diff)-sign(lag_vwap)))
```




```
> chart_Series(
+   pnl[endpoints(pnl, on="hours"), ],
+   name=paste(symbol, "VWAP Trend Following Strategy PnL"))
```

draft: Estimating Hurst Exponent From *OHLC* Data

The Hurst exponent is a measure of long-term memory of a time series, and is related to its autocorrelation:

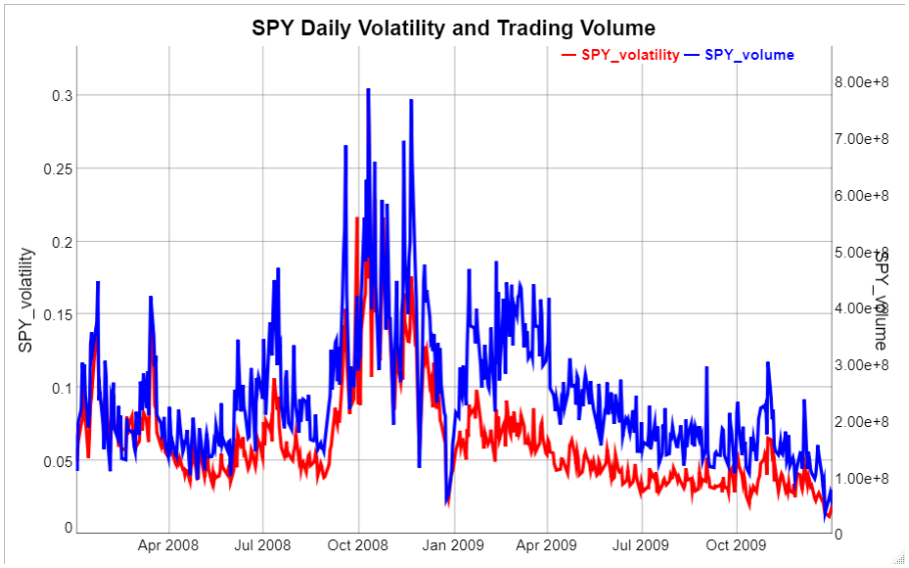
$$\mathbb{E}[\frac{(max(p) - min(p))}{\hat{\sigma}}] = t^H$$

$H = 0.5$ for Brownian motion (no autocorrelations),
$H > 0.5$ for positive autocorrelations,
$H < 0.5$ for negative autocorrelations.

The function `hurst_ohlc()` from package `HighFreq` calculates a Hurst-like indicator:

$$H = \frac{1}{n}\sum_{i=1}^{n} log(\frac{H_i - L_i}{abs(C_i - O_i)})$$

The function `agg_ohlc()` calculates a statistical estimator over an *OHLC* time series.




```
> # Daily Hurst exponents
> hurst_daily <- xts::apply.daily(x=HighFreq::SPY,
+                       FUN=agg_ohlc,
+                       agg_fun="hurst_ohlc")
> colnames(hurst_daily) <-
+   paste(colname(get(symbol)), ".Hurst")
> chart_Series(roll_sum(hurst_daily, 10)[-(1:10)]/10,
+        name=paste(symbol, "Hurst"))
> abline(h=0.5, col="blue", lwd=2)
```

draft: Conclusion

Open questions:

- is there an interaction between volatility, volume and skew?
- does kurtosis also have predictive value for market direction?
- how can higher moments (skew, kurtosis) help predict market crashes?
- what is the persistence of market anomalies over time?
- what is relationship between returns and cross-section of skew?
- This presentation is available on GitHub: https://github.com/algoquant/presentations

Acknowledgements:

- *Snowfall Systems* provides the *PortfolioEffect* system for:

  - real-time high frequency market data aggregations and risk metrics,
  - real-time portfolio analytics and optimization,
  - portfolio hosting,
  - https://www.portfolioeffect.com/
- Brian Peterson for Thomson Reuters tick data,
- Brian Peterson, Joshua Ulrich, and Jeffrey Ryan for packages *xts*, *quantmod*, *PerformanceAnalytics*, and *TTR*,

Package *IBrokers* for Using Interactive Brokers

Interactive Brokers (IB) is a brokerage company which provides an API for computer-driven trading, and it also provides extensive documentation:

- Interactive Brokers Main Page
- Interactive Brokers Documentation
- Interactive Brokers Course

Disclosure: I do have a personal account with Interactive Brokers.
But I do not have any other relationship with Interactive Brokers, and I do not endorse or recommend them.
Warning: Active trading is extremely risky, and most people lose money.
I advise not to trade with your own capital!

The package *IBrokers* contains R functions for executing IB commands using the Interactive Brokers API.

The package *IBrokers* has extensive documentation.

```
> # Install package IBrokers
> install.packages("IBrokers")
> # Load package IBrokers
> library(IBrokers)
> # Get documentation for package IBrokers
> # Get short description
> packageDescription("IBrokers")
> # Load help page
> help(package="IBrokers")
> # List all datasets in "IBrokers"
> data(package="IBrokers")
> # List all objects in "IBrokers"
> ls("package:IBrokers")
> # Remove IBrokers from search path
> detach("package:IBrokers")
> # Install package IBrokers2
> devtools::install_github(repo="algoquant/IBrokers2")
```

The package *IBrokers2* is derived from package *IBrokers*, and contains additional functions for executing real time trading strategies via the Interactive Brokers API.

Configuring the IB Trader Workstation (*TWS*)

Connecting to Interactive Brokers via the API requires being logged into the IB Trader Workstation (*TWS*) or the IB Gateway (*IBG*).

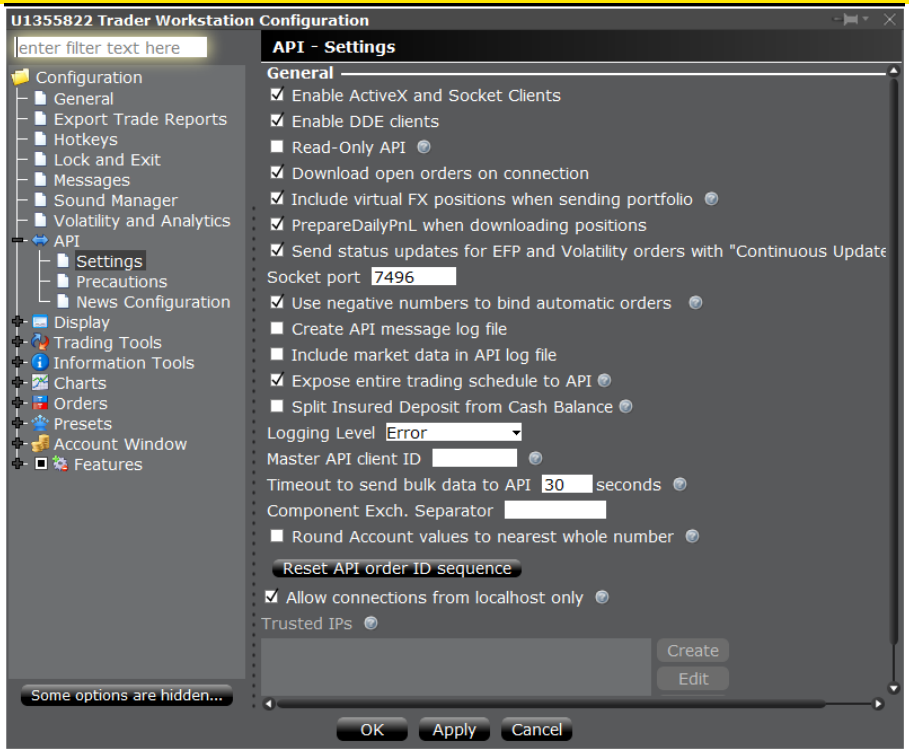
You should Download the TWS and install it.

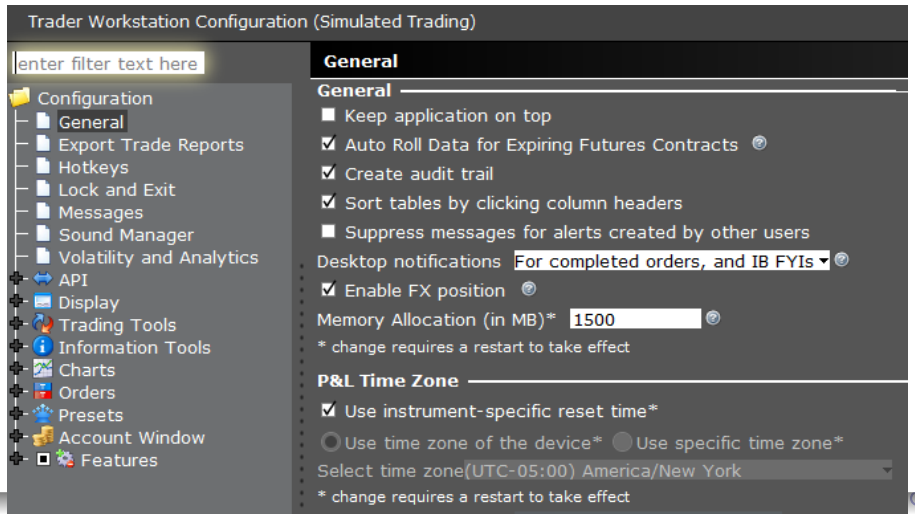
Read about the *TWS* initial setup.

The *TWS* settings must be configured to enable the API: *File* → *Global Configuration* → *API*

The *TWS* Java heap size should be increased to `1.5` GB: *File* → *Global Configuration* → *General*

Read more about the required TWS memory allocation and about how to change the TWS heap size.





Connecting to Interactive Brokers via the API

Connecting to Interactive Brokers via the API requires being logged into the IB Trader Workstation (*TWS*) or the IB Gateway (*IBG*).

Interactive Brokers provides extensive API Documentation.

The functions `twsConnect()` and `ibgConnect()` open a connection to the Interactive Brokers API, via either the *TWS* or the *IBG*.

The parameter `port` should be assigned to the value of the *socket port* displayed in *TWS* or *IBG*.

The function `twsDisconnect()` closes the Interactive Brokers API connection.

```
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Or connect to IB Gateway
> # Ib_connect <- ibgConnect(port=4002)
> # Check connection
> IBrokers::isConnected(ib_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Downloading Account Information from Interactive Brokers

The function `reqAccountUpdates()` returns a list with the account information (requires an account code).

The first element of the list contains the account dollar balances, while the remaining elements contain contract information.

The function `twsPortfolioValue()` returns a data frame with commonly used account fields, such as the contract names, net positions, and realized and unrealized profits and losses (pnl's).

```
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Or connect to IB Gateway
> # Ib_connect <- ibgConnect(port=4002)
> # Download account information from IB
> ac_count <- "DU1215081"
> ib_account <- IBrokers::reqAccountUpdates(conn=ib_connect,
+                                           acctCode=ac_count)
> # Extract account balances
> balance_s <- ib_account[[1]]
> balance_s$AvailableFunds
> # Extract contract names, net positions, and profits and losses
> IBrokers::twsPortfolioValue(ib_account)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Defining Contracts Using Package *IBrokers*

The function `twsEquity()` defines a stock contract (IB contract object).

The functions `twsFuture()` and `twsCurrency()` define futures and currency contracts.

The function `reqContractDetails()` returns a list with information on the IB instrument.

The package *twsInstrument* contains utility functions for enhancing the package *IBrokers*.

To define an IB contract, first look it up by keyword in the online IB Contract and Symbol Database, and find the *Conid* for that instrument.

Enter the *Conid* into the function `twsInstrument::getContract()`, which will return the IB contract object for that instrument.

Interactive Brokers provides more information about financial contracts here: IB Traded Products

```
> # Define AAPL stock contract (object)
> contractobj <- IBrokers::twsEquity("AAPL", primary="SMART")
> # Define CHF currency contract
> contractobj <- IBrokers::twsCurrency("CHF", currency="USD")
> # Define S&P Emini future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ES",
+   exch="GLOBEX", expiry="201906")
> # Define 10yr Treasury future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ZN",
+   exch="ECBOT", expiry="201906")
> # Define euro currency future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="EUR",
+   exch="GLOBEX", expiry="201906")
> # Define Gold future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="GC",
+   exch="NYMEX", expiry="201906")
> # Define Oil future January 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="QM",
+   exch="NYMEX", expiry="201901")
> # Test if contract object is correct
> IBrokers::is.twsContract(contractobj)
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
> # Install the package twsInstrument
> install.packages("twsInstrument", repos="http://r-forge.r-project.
> # Define euro future using getContract() and Conid
> contractobj <- twsInstrument::getContract("317631411")
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
```

Defining *VIX* Futures Contracts

VIX futures have both monthly and weekly contracts, with the monthly contracts being the most liquid.

In order to uniquely specify a *VIX* futures contract, the parameter `"local"` should be passed into the function `twsFuture()`, with the local security name.

For example, *VXV8* is the local security name (symbol) for the monthly *VIX* futures contract expiring on October 17th, 2018.

VX40V8 is the local security name (symbol) for the weekly *VIX* futures contract expiring on October 3rd, 2018.

The function `reqContractDetails()` returns a list with information on the IB instrument.

VIX futures are traded on the *CFE* (CBOE Futures Exchange): http://cfe.cboe.com/

```
> # Define VIX monthly and weekly futures June 2019 contract
> symbol <- "VIX"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   exch="CFE", expiry="201906")
> # Define VIX monthly futures June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   local="VXV8", exch="CFE", expiry="201906")
> # Define VIX weekly futures October 3rd 2018 contract
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   local="VX40V8", exch="CFE", expiry="201906")
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect,
+   Contract=contractobj)
```

Downloading Historical Daily Data from Interactive Brokers

The function `reqHistoricalData()` downloads historical data to a `.csv` file.

The historical *daily* bar data fields are `"Open"`, `"High"`, `"Low"`, `"Close"`, `"Volume"`, `"WAP"`, `"Count"`. (`"WAP"` is the weighted average price.)

Interactive Brokers provides more information about historical market data:

- IB Historical Market Data
- IB Historical Bar Data Fields

```
> # Define S&P Emini futures June 2019 contract
> symbol <- "ES"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   exch="GLOBEX", expiry="201906")
> # Open file for data download
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> dir.create(dir_name)
> file_name <- file.path(dir_name, paste0(symbol, "201906.csv"))
> file_connect <- file(file_name, open="w")
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Write header to file
> cat(paste(paste(symbol, c("Index", "Open", "High", "Low", "Close",
> # Download historical data to file
> IBrokers::reqHistoricalData(conn=ib_connect,
+   Contract=contractobj,
+   barSize="1 day", duration="6 M",
+   file=file_connect)
> # Close data file
> close(file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Downloading Historical Intraday Data for a Portfolio

Historical data for a portfolio of symbols can be downloaded in a loop.

The historical *intraday* bar data fields are `"Open"`, `"High"`, `"Low"`, `"Close"`, `"Volume"`, `"WAP"`, `"XTRA"`, `"Count"`. (`"WAP"` is the weighted average price.)

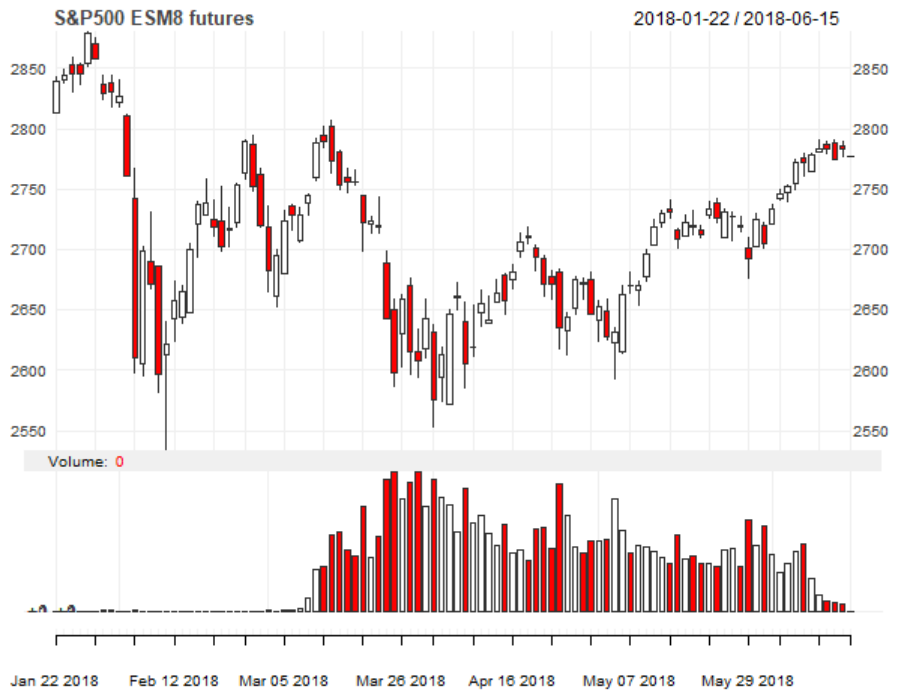
```
> # Define IB contract objects for stock symbols
> symbolv <- c("AAPL", "F", "MSFT")
> contractv <- lapply(symbolv, IBrokers::twsEquity, primary="SMART")
> names(contractv) <- symbolv
> # Open file connections for data download
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> file_names <- file.path(dir_name, paste0(symbolv, format(Sys.time
> file_connects <- lapply(file_names, function(file_name) file(file_
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download historical 1-minute bar data to files
> for (it in 1:NROW(symbolv)) {
+   symbol <- symbolv[it]
+   file_connect <- file_connects[[it]]
+   contractobj <- contractv[[it]]
+   cat("Downloading data for: ", symbol, "\n")
+   # Write header to file
+   cat(paste(paste(symbol, c("Index", "Open", "High", "Low", "Close
+   IBrokers::reqHistoricalData(conn=ib_connect,
+                               Contract=contractobj,
+                               barSize="1 min", duration="2 D",
+                               file=file_connect)
+   Sys.sleep(10) # 10s pause to avoid IB pacing violation
+ }  # end for
> # Close data files
> for (file_connect in file_connects) close(file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Downloading Historical Data for Expired Futures Contracts

Historical data for expired futures contracts can be downloaded using `reqHistoricalData()` with the appropriate `expiry` date, and the parameter `include_expired="1"`.

For example, *ESM8* is the symbol for the *S&P500* emini futures expiring in June 2018.

```
> # Define S&P Emini futures June 2018 contract
> symbol <- "ES"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   include_expired="1",
+   exch="GLOBEX", expiry="201806")
> # Open file connection for ESM8 data download
> file_name <- file.path(dir_name, paste0(symbol, "M8.csv"))
> file_connect <- file(file_name, open="w")
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download historical data to file
> IBrokers::reqHistoricalData(conn=ib_connect,
+   Contract=contractobj,
+   barSize="1 day", duration="2 Y",
+   file=file_connect)
> # Close data file
> close(file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```




```
> # Load OHLC data and coerce it into xts series
> pricev <- data.table::fread(file_name)
> data.table::setDF(pricev)
> pricev <- xts::xts(pricev[, 2:6],
+   order.by=as.Date(as.POSIXct.numeric(pricev[, 1],
+     tz="America/New_York", origin="1970-01-01")))
> colnames(pricev) <- c("Open", "High", "Low", "Close", "Volume")
> # Plot OHLC data in x11 window
> chart_Series(x=pricev, TA="add_Vo()",
+   name="S&P500 ESM8 futures")
> # Plot dygraph
> dygraphs::dygraph(pricev[, 1:4], main="S&P500 ESM8 futures") %>%
+   dyCandlestick()
```

draft: Downloading Continuous Contract Futures Data

Note: Continuous futures require passing the parameter: contract.secType = "CONTFUT", which isn't currently included in package IBrokers.

http://interactivebrokers.github.io/tws-api/basic_contracts.html

A continuous futures contract is a synthetic time series of prices, created by splicing together prices from several futures contracts with different expiration dates.

At any point in time, the price of the continuous contract is equal to the most liquid contract times a normalization factor.

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

The continuous price is multiplied by a normalization factor when the contract is rolled, to remove jumps caused by the shape of the futures curve.

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

Futures contracts trade at different prices (because of the futures convenience yield).

This cause price jumps between the currently expiring futures contract and the next futures contract. A continuous futures contract adjusts the prices to remove these jumps and time differences to create an artificial price series.

```
> # Define S&P Emini futures June 2018 contract
> symbol <- "ES"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   include_expired="1",
+   exch="GLOBEX", expiry="201806")
> # Open file connection for data download
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> dir.create(dir_name)
> file_name <- file.path(dir_name, paste0(symbol, ".csv"))
> file_connect <- file(file_name, open="w")
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download historical data to file
> IBrokers::reqHistoricalData(conn=ib_connect,
+   Contract=contractobj,
+   barSize="1 day", duration="6 M",
+   file=file_connect)
> # Close data file
> close(file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Downloading Live *TAQ* Data from Interactive Brokers

The function `reqMktData()` downloads live (real-time) trades and quotes (*TAQ*) data from Interactive Brokers.

The function `eWrapper()` formats the real-time market events (trades and quotes), so they can be displayed or saved to a file.

The method `eWrapper.MktData.CSV()` formats the real-time *TAQ* data so it can be saved to a `.csv` file.

The real-time *TAQ* data fields are *BidSize, BidPrice, AskPrice, AskSize, Last, LastSize, Volume*.

BidPrice is the quoted bid price, *AskPrice* is the quoted offer price, and *Last* is the most recent traded price.

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

```
> # Define S&P Emini futures June 2019 contract
> symbol <- "ES"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   exch="GLOBEX", expiry="201906")
> # Open file connection for data download
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> # Dir.create(dir_name)
> file_name <- file.path(dir_name, paste0(symbol, "_taq_live.csv"))
> file_connect <- file(file_name, open="w")
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download live data to file
> IBrokers::reqMktData(conn=ib_connect,
+      Contract=contractobj,
+      eventWrapper=eWrapper.MktData.CSV(1),
+      file=file_connect)
> # Close data file
> close(file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Downloading Live *OHLC* Data from Interactive Brokers

The function `reqRealTimeBars()` downloads live (real-time) *OHLC* market data from Interactive Brokers.

The method `eWrapper.RealTimeBars.CSV()` formats the real-time *OHLC* data so it can be saved to a `.csv` file.

Interactive Brokers by default only provides 5-second bars of real-time prices (but it also provides historical data at other frequencies).

```
> # Define S&P Emini futures June 2019 contract
> symbol <- "ES"
> contractobj <- IBrokers::twsFuture(symbol=symbol,
+   exch="GLOBEX", expiry="201906")
> # Open file connection for data download
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> # Dir.create(dir_name)
> file_name <- file.path(dir_name, paste0(symbol, "_ohlc_live.csv"))
> file_connect <- file(file_name, open="w")
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download live data to file
> IBrokers::reqRealTimeBars(conn=ib_connect,
+      Contract=contractobj, barSize="1",
+      eventWrapper=eWrapper.RealTimeBars.CSV(1),
+      file=file_connect)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
> # Close data file
> close(file_connect)
> # Load OHLC data and coerce it into xts series
> library(data.table)
> pricev <- data.table::fread(file_name)
> pricev <- xts::xts(pricev[, paste0("V", 2:6)],
+   as.POSIXct.numeric(as.numeric(pricev[, V1]), tz="America/New_Yo
> colnames(pricev) <- c("Open", "High", "Low", "Close", "Volume")
> # Plot OHLC data in x11 window
> x11()
> chart_Series(x=pricev, TA="add_Vo()",
+        name="S&P500 ESM9 futures")
> # Plot dygraph
> library(dygraphs)
> dygraphs::dygraph(pricev[, 1:4], main="S&P500 ESM9 futures") %>%
+   dyCandlestick()
```

Downloading Live *OHLC* Data For Multiple Contracts

Live *OHLC* data can be downloaded for multiple instruments simultaneously.

This requires passing a *list* of contracts and file connections to `reqRealTimeBars()`, and also passing the number of contracts to `eWrapper.RealTimeBars.CSV()`.

The bar prices for each contract are written into a separate file.

```
> library(IBrokers)
> # Define list of S&P futures and 10yr Treasury contracts
> contractv <- list(ES=IBrokers::twsFuture(symbol="ES", exch="GLOBEX
+             ZN=IBrokers::twsFuture(symbol="ZN", exch="ECBOT", exp
> # Open the file connection for storing the bar data
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> file_names <- file.path(dir_name, paste0(c("ES", "ZN_"), format(Sy
> file_connects <- lapply(file_names, function(file_name) file(file_
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> # Download live data to file
> IBrokers::reqRealTimeBars(conn=ib_connect,
+                     Contract=contractv,
+                     barSize="1", useRTH=FALSE,
+                     eventWrapper=eWrapper.RealTimeBars.CSV(NROW(co
+                     file=file_connects)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
> # Close data files
> for (file_connect in file_connects)
+   close(file_connect)
> library(data.table)
> # Load ES futures June 2019 contract and coerce it into xts series
> pricev <- data.table::fread(file_names[1])
> pricev <- xts::xts(pricev[, paste0("V", 2:6)],
+   as.POSIXct.numeric(as.numeric(pricev[, V1]), tz="America/New_Yor
> colnames(pricev) <- c("Open", "High", "Low", "Close", "Volume")
> # Plot dygraph
> library(dygraphs)
> dygraphs::dygraph(pricev[, 1:4], main="S&P500 ESM9 futures") %>%
+   dyCandlestick()
> # Load ZN 10yr Treasury futures June 2019 contract
> pricev <- data.table::fread(file_names[2])
> pricev <- xts::xts(pricev[, paste0("V", 2:6)],
+   as.POSIXct.numeric(as.numeric(pricev[, V1]), tz="America/New_Yor
> colnames(pricev) <- c("Open", "High", "Low", "Close", "Volume")
> # Plot dygraph
```

Event Processing Using `eWrapper()` and `twsCALLBACK()`

Functions which process real-time market events (like `reqMktData()` and `reqRealTimeBars()`) rely on the functions `eWrapper()` and `twsCALLBACK`.

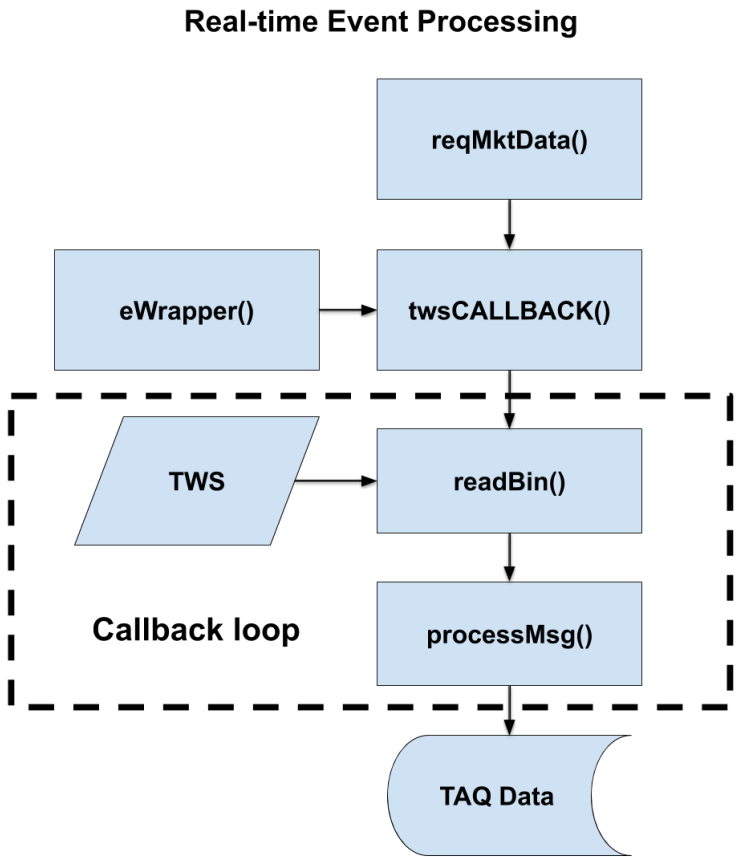
The function `eWrapper()` creates an *eWrapper* object, consisting of a data environment and handler (methods) for formatting and adding new data to it.

The function `reqMktData()` accepts an *eWrapper* object and passes it into `twsCALLBACK()`.

The function `twsCALLBACK()` processes market events by calling functions `readBin()` and `processMsg()` in a `while()` loop.

The *TWS* broadcasts real-time market events in the form of `character` strings, which are captured by `readBin()`.

The `character` strings are then parsed by `processMsg()`, and copied to the data environment of the *eWrapper* object.



Placing Market Trade Orders on *TWS*

The function `reqIds()` requests a trade order ID from Interactive Brokers *TWS*.

The function `twsOrder()` creates a trade order object.

The parameter `orderType` specifies the type of trade order: market order (`MKT`), limit order (`LMT`), etc.

Each trade order requires its own ID generated by `reqIds()`.

The function `placeOrder()` places a trade order.

```
> # Define Japanese yen currency futures June 2019 contract 6JZ8
> contractobj <- IBrokers::twsFuture(symbol="JPY", exch="GLOBEX", ex
> # Connect to Interactive Brokers TWS
> ib_connect <- IBrokers::twsConnect(port=7497)
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
> # Request trade order ID
> order_id <- IBrokers::reqIds(ib_connect)
> # Create buy market order object
> ib_order <- IBrokers::twsOrder(order_id,
+   orderType="MKT", action="BUY", totalQuantity=1)
> # Place trade order
> IBrokers::placeOrder(ib_connect, contractobj, ib_order)
> # Execute sell market order
> order_id <- IBrokers::reqIds(ib_connect)
> ib_order <- IBrokers::twsOrder(order_id,
+   orderType="MKT", action="SELL", totalQuantity=1)
> IBrokers::placeOrder(ib_connect, contractobj, ib_order)
> # Execute buy market order
> order_id <- IBrokers::reqIds(ib_connect)
> ib_order <- IBrokers::twsOrder(order_id,
+   orderType="MKT", action="BUY", totalQuantity=1)
> IBrokers::placeOrder(ib_connect, contractobj, ib_order)
```

Placing and Cancelling Limit Trade Orders on *TWS*

The function `twsOrder()` with `orderType=LMT` creates a limit trade order object.

The function `cancelOrder()` with the parameter `orderId=order_id` cancels a trade order with the ID equal to `order_id`.

```
> # Request trade order ID
> order_id <- IBrokers::reqIds(ib_connect)
> # Create buy limit order object
> ib_order <- IBrokers::twsOrder(order_id, orderType="LMT",
+   lmtPrice="1.1511", action="BUY", totalQuantity=1)
> # Place trade order
> IBrokers::placeOrder(ib_connect, contractobj, ib_order)
> # Cancel trade order
> IBrokers::cancelOrder(ib_connect, order_id)
> # Execute sell limit order
> order_id <- IBrokers::reqIds(ib_connect)
> ib_order <- IBrokers::twsOrder(order_id, orderType="LMT",
+   lmtPrice="1.1512", action="SELL", totalQuantity=1)
> IBrokers::placeOrder(ib_connect, contractobj, ib_order)
> # Cancel trade order
> IBrokers::cancelOrder(ib_connect, order_id)
> # Close the Interactive Brokers API connection
> IBrokers::twsDisconnect(ib_connect)
```

Placing Limit Orders in a Programmatic Loop

The function `reqRealTimeBars()` creates data objects and then calls `twsCALLBACK()` and passes the objects to `twsCALLBACK()`.

`twsCALLBACK()` calls `processMsg()` in a loop, and passes the `eWrapper()` to it.

`processMsg()` performs if-else statements and calls eWrapper methods.

`eWrapper()` creates an environment for data, and methods (handlers) for formatting the data.

`eWrapper.RealTimeBars.CSV()` calls `eWrapper()` to create a closure (function) and returns it.

Closures allow creating mutable states for persistent data objects.

Limit trade orders can be placed inside a modified `eWrapper()` function.

```
> eWrapper_realtimebars <- function(n = 1) {
+   eW <- eWrapper_new(NULL)
+   # eW <- IBrokers::eWrapper(NULL)
+   eW$assign.Data("data", rep(list(structure(.xts(matrix(rep(NA_rea
+   eW$realtimeBars <- function(curMsg, msg, timestamp, file, ...) {
+     id <- as.numeric(msg[2])
+     file <- file[[id]]
+     data <- eW$get.Data("data")
+     attr(data[[id]], "index") <- as.numeric(msg[3])
+     nr.data <- NROW(data[[id]])
+     # Write to file
+     cat(paste(msg[3], msg[4], msg[5], msg[6], msg[7], msg[8], msg
+     # Write to console
+     # eW$count_er <- eW$count_er + 1
+     eW$assign.Data("count_er", eW$get.Data("count_er")+1)
+     cat(paste0("count_er=", eW$get.Data("count_er"), "\tOpen=", ms
+     # cat(paste0("Open=", msg[4], "\tHigh=", msg[5], "\tLow=", msg
+     ### Trade
+     # Cancel previous trade orders
+     buy_id <- eW$get.Data("buy_id")
+     sell_id <- eW$get.Data("sell_id")
+     if (buy_id>0) IBrokers::cancelOrder(ib_connect, buy_id)
+     if (sell_id>0) IBrokers::cancelOrder(ib_connect, sell_id)
+     # Execute buy limit order
+     buy_id <- IBrokers::reqIds(ib_connect)
+     buy_order <- IBrokers::twsOrder(buy_id, orderType="LMT",
+                                     lmtPrice=msg[6]-0.25, action="BUY",
+     IBrokers::placeOrder(ib_connect, contractobj, buy_order)
+     # Execute sell limit order
+     sell_id <- IBrokers::reqIds(ib_connect)
+     sell_order <- IBrokers::twsOrder(sell_id, orderType="LMT",
+                                      lmtPrice=msg[5]+0.25, action="SELL"
+     IBrokers::placeOrder(ib_connect, contractobj, sell_order)
+     # Copy new trade orders
+     eW$assign.Data("buy_id", buy_id)
+     eW$assign.Data("sell_id", sell_id)
+     ### Trade finished
+     data[[id]][nr.data, 1:7] <- as.numeric(msg[4:10])
```

draft: Advanced Order Types and Execution Algos

Interactive Brokers provides many different Advanced Order Types and Execution Algos.

The function `reqRealTimeBars()` creates data objects and then calls `twsCALLBACK()` and passes the objects to `twsCALLBACK()`.

`twsCALLBACK()` calls `processMsg()` in a loop, and passes the `eWrapper()` to it.

`processMsg()` performs if-else statements and calls eWrapper methods.

`eWrapper()` creates an environment for data, and methods (functions) for handling (formatting) the data.

`eWrapper.RealTimeBars.CSV()` calls `eWrapper()` to create a closure (function) and returns it.

Closures allow creating mutable states for persistent data objects.

Limit trade orders can be placed inside a modified `eWrapper()` function.

```
> eWrapper_realtimebars <- function(n = 1) {
+   eW <- eWrapper_new(NULL)
+   # eW <- IBrokers::eWrapper(NULL)
+   eW$assign.Data("data", rep(list(structure(.xts(matrix(rep(NA_rea
+   eW$realtimeBars <- function(curMsg, msg, timestamp, file, ...) {
+     id <- as.numeric(msg[2])
+     file <- file[[id]]
+     data <- eW$get.Data("data")
+     attr(data[[id]], "index") <- as.numeric(msg[3])
+     nr.data <- NROW(data[[id]])
+     # Write to file
+     cat(paste(msg[3], msg[4], msg[5], msg[6], msg[7], msg[8], msg
+     # Write to console
+     # eW$count_er <- eW$count_er + 1
+     eW$assign.Data("count_er", eW$get.Data("count_er")+1)
+     cat(paste0("count_er=", eW$get.Data("count_er"), "\tOpen=", ms
+     # cat(paste0("Open=", msg[4], "\tHigh=", msg[5], "\tLow=", msg
+     ### Trade
+     # Cancel previous trade orders
+     buy_id <- eW$get.Data("buy_id")
+     sell_id <- eW$get.Data("sell_id")
+     if (buy_id>0) IBrokers::cancelOrder(ib_connect, buy_id)
+     if (sell_id>0) IBrokers::cancelOrder(ib_connect, sell_id)
+     # Execute buy limit order
+     buy_id <- IBrokers::reqIds(ib_connect)
+     buy_order <- IBrokers::twsOrder(buy_id, orderType="LMT",
+                                     lmtPrice=msg[6]-0.25, action="BUY",
+     IBrokers::placeOrder(ib_connect, contractobj, buy_order)
+     # Execute sell limit order
+     sell_id <- IBrokers::reqIds(ib_connect)
+     sell_order <- IBrokers::twsOrder(sell_id, orderType="LMT",
+                                      lmtPrice=msg[5]+0.25, action="SELL"
+     IBrokers::placeOrder(ib_connect, contractobj, sell_order)
+     # Copy new trade orders
+     eW$assign.Data("buy_id", buy_id)
+     eW$assign.Data("sell_id", sell_id)
+     ### Trade finished
+     data[[id]][nr.data, 1:7] <- as.numeric(msg[4:10])
```

draft: Downloading Live *TAQ* Data and Replaying It

Note: The script downloads the raw data, but doesn't replay the bar data properly.
https://offerm.wordpress.com/2015/05/21/market-data-recording-and-playback-with-ibrokers-and-r-2/

The function `reqMktData()` downloads live (real-time) trades and quotes (*TAQ*) data from Interactive Brokers.

The function `eWrapper()` formats the real-time market events (trades and quotes), so they can be displayed or saved to a file.

The method `eWrapper.MktData.CSV()` formats the real-time *TAQ* data so it can be saved to a `.csv` file.

The real-time *TAQ* data fields are *BidSize, BidPrice, AskPrice, AskSize, Last, LastSize, Volume*.

BidPrice is the quoted bid price, *AskPrice* is the quoted offer price, and *Last* is the most recent traded price.

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

```
> # Define S&P Emini futures June 2019 contract
> snp_contract <- IBrokers::twsFuture(symbol="ES",
+   exch="GLOBEX", expiry="201906")
> # Define VIX futures June 2019 contract
> vix_contract <- IBrokers::twsFuture(symbol="VIX",
+   local="VXZ8", exch="CFE", expiry="201906")
> # Define 10yr Treasury futures June 2019 contract
> trs_contract <- IBrokers::twsFuture(symbol="ZN",
+   exch="ECBOT", expiry="201906")
> # Define Emini gold futures June 2019 contract
> gold_contract <- IBrokers::twsFuture(symbol="YG",
+   exch="NYSELIFFE", expiry="201906")
> # Define euro currency future June 2019 contract
> euro_contract <- IBrokers::twsFuture(symbol="EUR",
+   exch="GLOBEX", expiry="201906")
> IBrokers::reqContractDetails(conn=ib_connect, Contract=euro_contra
>
> # Define data directory
> dir_name <- "/Users/jerzy/Develop/data/ib_data"
> # Dir.create(dir_name)
>
> # Open file for error messages
> file_root <- "replay"
> file_name <- file.path(dir_name, paste0(file_root, "_error.csv"))
> error_connect <- file(file_name, open="w")
>
> # Open file for raw data
> file_name <- file.path(dir_name, paste0(file_root, "_raw.csv"))
> raw_connect <- file(file_name, open="w")
>
> # Create empty eWrapper to redirect error messages to error file
> error_ewrapper <- eWrapper(debug=NULL, errfile=error_connect)
>
> # Create eWrapper for raw data
> raw_ewrapper <- eWrapper(debug=TRUE)
>
> # Redirect error messages to error eWrapper (error_ewrapper),
> # by replacing handler function errorMessage() in raw_ewrapper
```

draft: Defining Instrument Pairs in *TWS*

Users can Define Pairs of Instruments in the *TWS*, using the *Combo Selection* window.

To open the *Combo Selection* window, right-click on a blank contract field and select *Generic Combo*.

Or enter the symbol for one of the instruments and select *Combinations* followed by *Option Combos* from the drop down menu, and then click the tab *Pair or Leg-by-leg*.

The pairs can be traded as a single instrument, but the execution is not guaranteed, and the bid-offer spread may be very large.

The tab *Multiple* allows selecting a group of combination quotes on the same underlying for comparison.

First access the *Combo Selection* window, enter the symbol for one of the instruments and select *Combinations* followed by *Option Combos* from the drop down menu, and then click the tab *Pair or Leg-by-leg*.

Interactive Brokers provides more information about financial contracts here: IB Traded Products

```
> # Define AAPL stock contract (object)
> contractobj <- IBrokers::twsEquity("AAPL", primary="SMART")
> # Define CHF currency contract
> contractobj <- IBrokers::twsCurrency("CHF", currency="USD")
> # Define S&P Emini future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ES",
+   exch="GLOBEX", expiry="201906")
> # Define 10yr Treasury future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ZN",
+   exch="ECBOT", expiry="201906")
> # Define euro currency future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="EUR",
+   exch="GLOBEX", expiry="201906")
> # Define Gold future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="GC",
+   exch="NYMEX", expiry="201906")
> # Test if contract object is correct
> IBrokers::is.twsContract(contractobj)
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
> # Install the package twsInstrument
> install.packages("twsInstrument", repos="http://r-forge.r-project.
> # Define euro future using getContract() and Conid
> contractobj <- twsInstrument::getContract("317631411")
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
```

draft: Defining Investment Strategies in *TWS*

Users can Define Investment Strategies in the *TWS*, using the *Portfolio Builder* window.

To open the *Portfolio Builder* window, left-click on the New Window dropdown on the upper-left of *TWS*, and select *Portfolio Builder*.

Click "Create New Strategy" to open and edit Investment Rules in the sidecar populated with sample data based on the last sorting option you selected. The main Portfolio Builder reflects changes made in the sidecar in real time.

Or enter the symbol for one of the instruments and select *Combinations* followed by *Option Combos* from the drop down menu, and then click the tab *Pair or Leg-by-leg*.

The pairs can be traded as a single instrument, but the execution is not guaranteed, and the bid-offer spread may be very large.

The tab *Multiple* allows selecting a group of combination quotes on the same underlying for comparison.

First access the *Combo Selection* window, enter the symbol for one of the instruments and select *Combinations* followed by *Option Combos* from the drop down menu, and then click the tab *Pair or Leg-by-leg*.

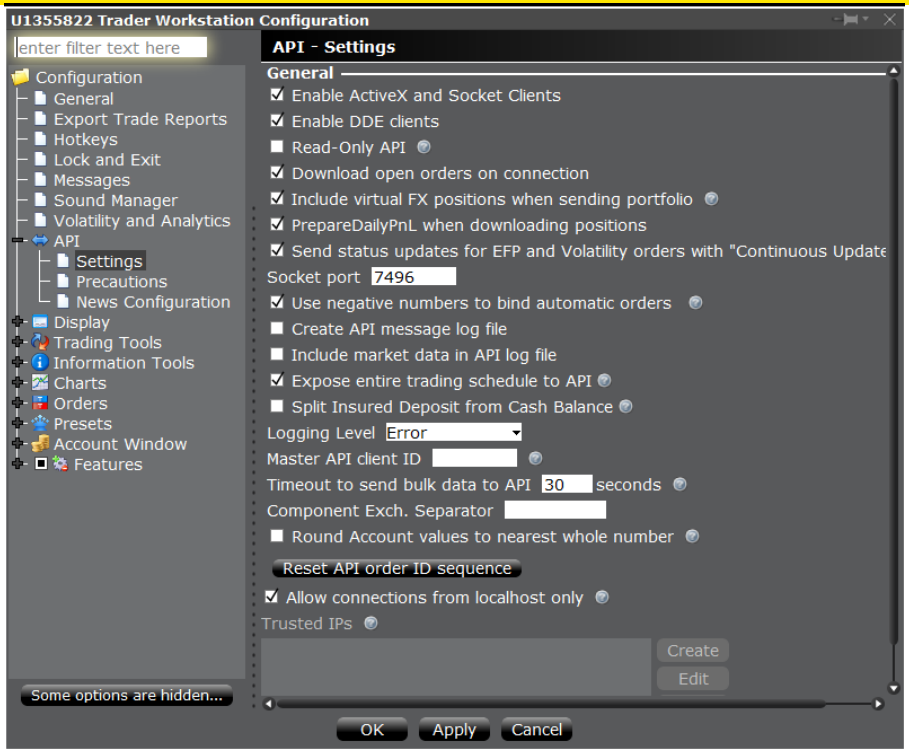
Interactive Brokers provides more information about

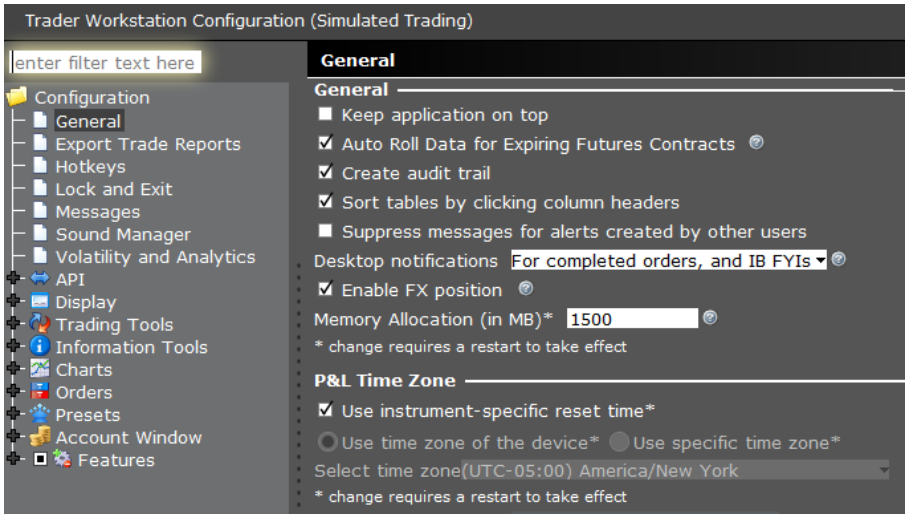
```
> # Define AAPL stock contract (object)
> contractobj <- IBrokers::twsEquity("AAPL", primary="SMART")
> # Define CHF currency contract
> contractobj <- IBrokers::twsCurrency("CHF", currency="USD")
> # Define S&P Emini future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ES",
+   exch="GLOBEX", expiry="201906")
> # Define 10yr Treasury future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="ZN",
+   exch="ECBOT", expiry="201906")
> # Define euro currency future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="EUR",
+   exch="GLOBEX", expiry="201906")
> # Define Gold future June 2019 contract
> contractobj <- IBrokers::twsFuture(symbol="GC",
+   exch="NYMEX", expiry="201906")
> # Test if contract object is correct
> IBrokers::is.twsContract(contractobj)
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
> # Install the package twsInstrument
> install.packages("twsInstrument", repos="http://r-forge.r-project
> # Define euro future using getContract() and Conid
> contractobj <- twsInstrument::getContract("317631411")
> # Get list with instrument information
> IBrokers::reqContractDetails(conn=ib_connect, Contract=contractobj
```

draft: Linux

Advanced Package Tool, or apt, is a free software user interface that works with core libraries to handle the installation and removal of software on Debian, Ubuntu and other Linux distributions. Debian uses the dpkg packaging system apt is a more friendly way to handle packaging than dpkg. apt-get is similar to apt. apt consists some of the most widely used features from apt-get and apt-cache leaving aside obscure and seldom used features. apt provides the most common used commands from apt-get and apt-cache.
https://itsfoss.com/apt-command-guide/
https://itsfoss.com/apt-vs-apt-get-difference/

Get Debian version and release cat /etc/issue cat /etc/os-release
Install GNOME Debian Desktop Environment
https://www.gnome.org/
https://wiki.debian.org/Gnome
https://wiki.debian.org/DesktopEnvironment Install VNC server https://medium.com/google-cloud/graphical-user-interface-gui-for-google-compute-engine-instance-78fccda09e5c sudo apt-get install vnc4server vncserver
Install Windows RDP to Debian remote desktop
http://blog.technotesdesk.com/how-to-use-rdp-from-windows-to-connect-to-debian-or-ubuntu-machine
apt-get install xrdp Run Windows RDP to Static IP Address https://cloud.google.com/compute/docs/ip-addresses/reserve-static-external-ip-address

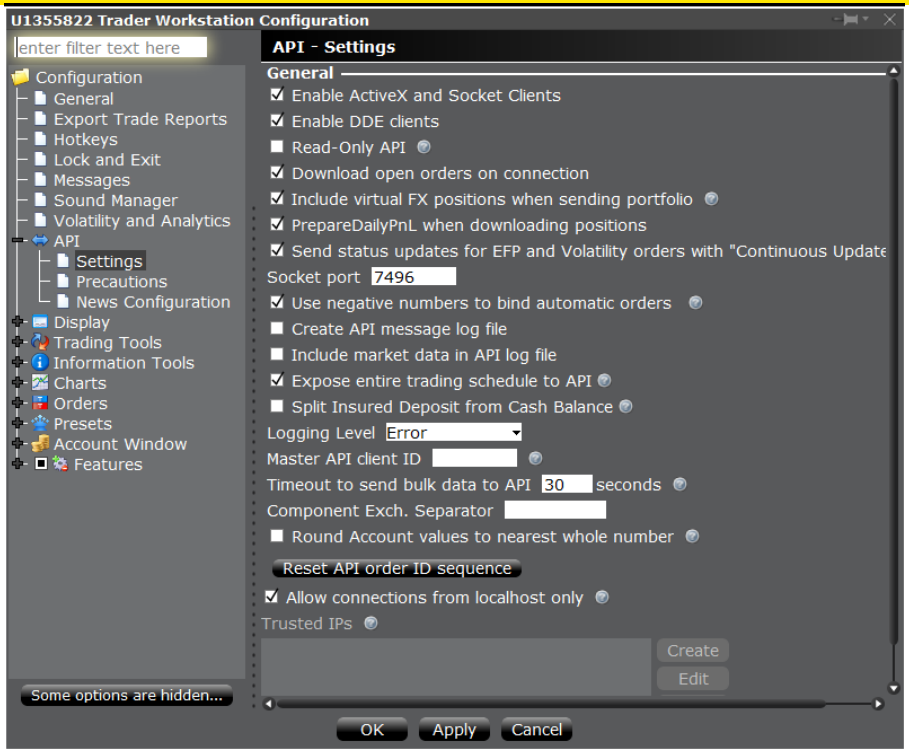


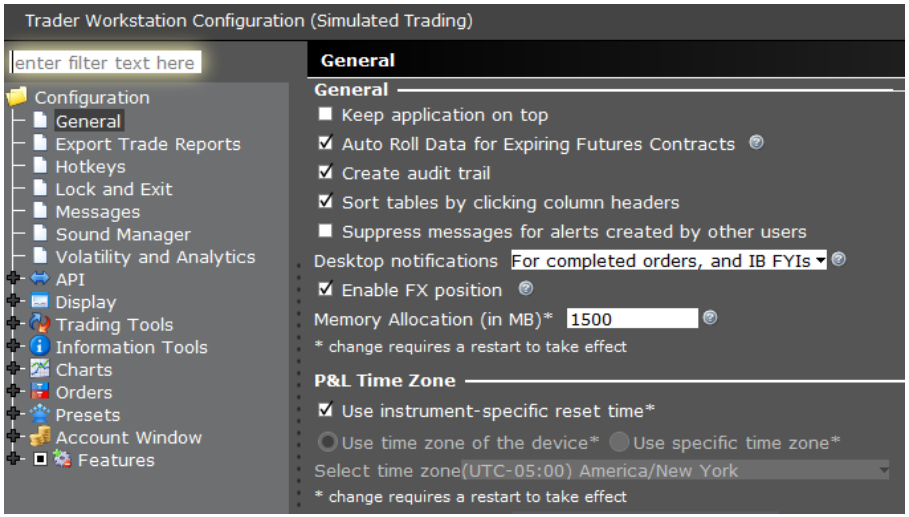


draft: Creating Google Cloud Project

Install R
https://www.rstudio.com/products/rstudio/download-server/ https://cran.rstudio.com/bin/linux/debian/
sudo apt-get update sudo apt-get install r-base r-base-dev sudo apt-get install libopenblas-base sudo apt-get install libssl-dev sudo apt-get install libgit2-dev sudo apt-get install libssh2-1-dev sudo apt-get install libcurl4-openssl-dev
Change R lib permission cd /usr/local/lib/R sudo chmod o+w site-library
Type "R" In R install.packages("openssl") install.packages("git2r") install.packages("curl") install.packages("httr") install.packages("usethis") install.packages("devtools")
Upgrade R sudo apt-get remove r-base cat /etc/apt/sources.list sudo nano /etc/apt/sources.list deb http://cran.rstudio.com/bin/linux/debian stretch-cran35/ sudo apt-get install dirmngr gpg –keyserver keyserver.ubuntu.com –recv-key FCAE2A0E115C3D8A gpg -a –export FCAE2A0E115C3D8A — sudo apt-key add - sudo apt-get update sudo apt-get install r-base sudo apt-get install r-base-dev
sudo apt update sudo apt upgrade
https://www.rstudio.com/products/rstudio/download-server/

Connecting to Interactive Brokers via the API requires being logged into the IB Trader Workstation (TWS) or





draft: Pursuing a Career in Portfolio Management

Becoming a portfolio manager (PM) is very difficult:

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