FRE6871 R in Finance

Lecture#7, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

May 6, 2024

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.

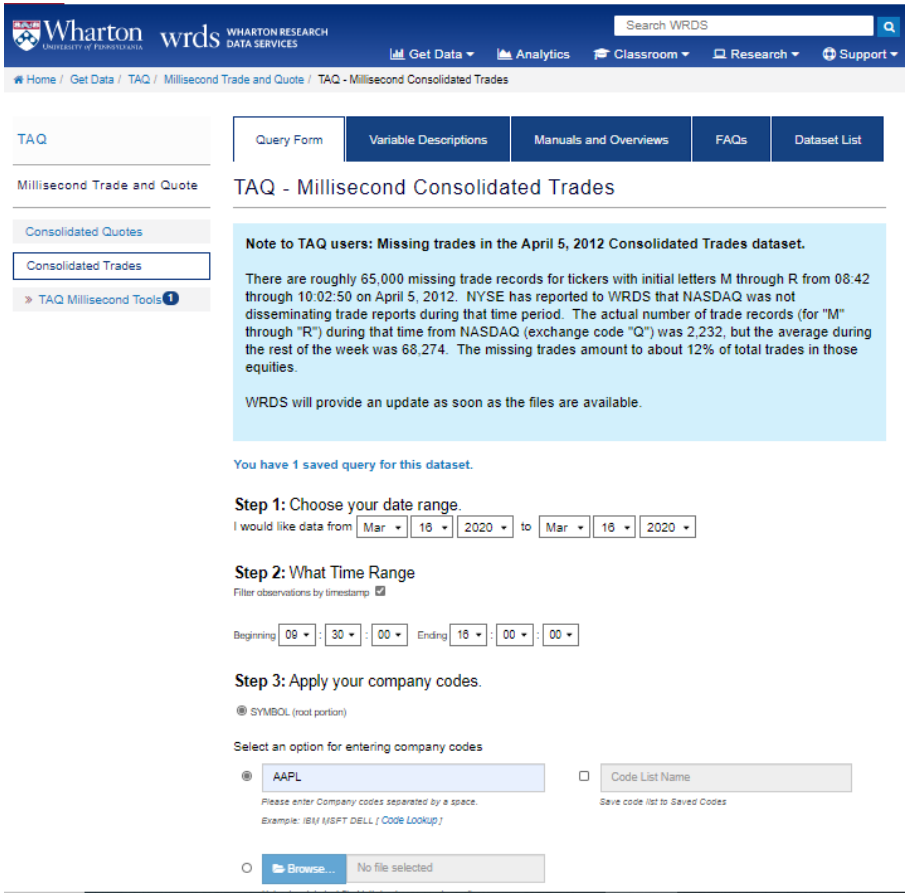
```
> # Load package HighFreq
> library(HighFreq)
> # Or load the high frequency data file directly:
> # symbolv <- load("/Users/jerzy/Develop/R/HighFreq/data/hf_data.RI
> head(HighFreq::SPY_TAQ)
> head(HighFreq::SPY)
> tail(HighFreq::SPY)
```

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

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

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

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

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

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

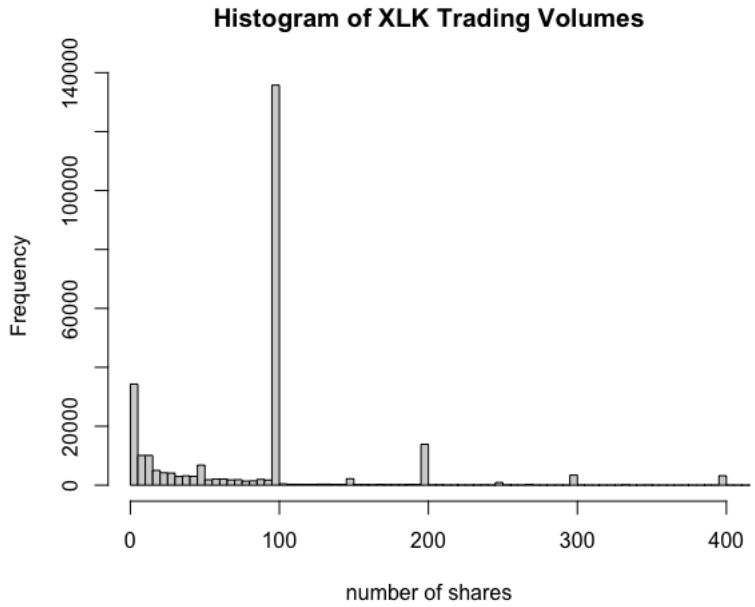
```
> library(HighFreq)
> # Read TAQ trade data from csv file
> taq <- data.table::fread(file="/Users/jerzy/Develop/lecture_slides
> # Inspect the TAQ data in data.table format
> taq
> class(taq)
> colnames(taq)
> sapply(taq, class)
> symboln <- taq$SYM_ROOT[1]
> # Create date-time index
> datev <- paste(taq$DATE,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

Trading Volumes in High Frequency Data

The trading volumes represent the number of shares traded at a given price.

The histogram of the trading volumes shows that the highest frequencies of trades are for 100 shares and for round lots (trades that are multiples of 100 shares.)

There are also significant frequencies for *odd lots*, with small volumes of less than 100 shares.




```
> # Coerce trade ticks to xts series
> xlk <- xts::xts(taq[, .(price, volume)], datev)
> colnames(xlk) <- c("price", "volume")
> save(xlk, file="/Users/jerzy/Develop/data/xlk_tick_trades_2020031
> # Plot histogram of the trading volumes
> hist(xlk$volume, main="Histogram of XLK Trading Volumes",
+      breaks=1e5, xlim=c(1, 400), xlab="number of shares")
```

Microstructure Noise in High Frequency Data

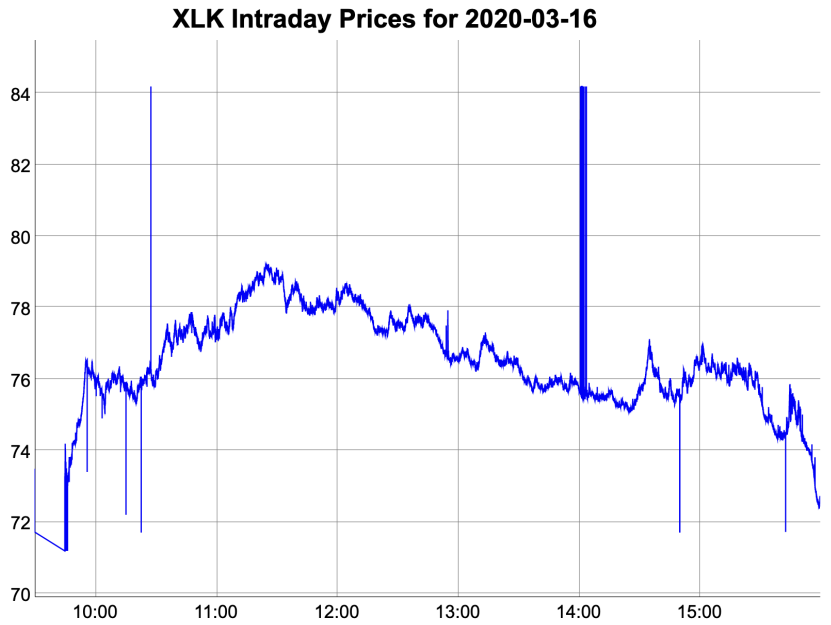
High frequency data contains *microstructure noise* in the form of *price spikes* and the *bid-ask bounce*.

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

Price spikes are often caused by data collection errors, but sometimes they represent actual trades with very large lot (trade) sizes.

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 reality the mid price is unchanged.




```
> # Plot dygraph
> dygraphs::dygraph(xlk$price, main="XLK Intraday Prices for 2020-03
+   dyOptions(colors="blue", strokeWidth=1)
> # Plot in x11 window
> x11(width=6, height=5)
> quantmod::chart_Series(x=xlk$price, name="XLK Intraday Prices for
```

The Bid-ask Bounce of High Frequency Prices

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

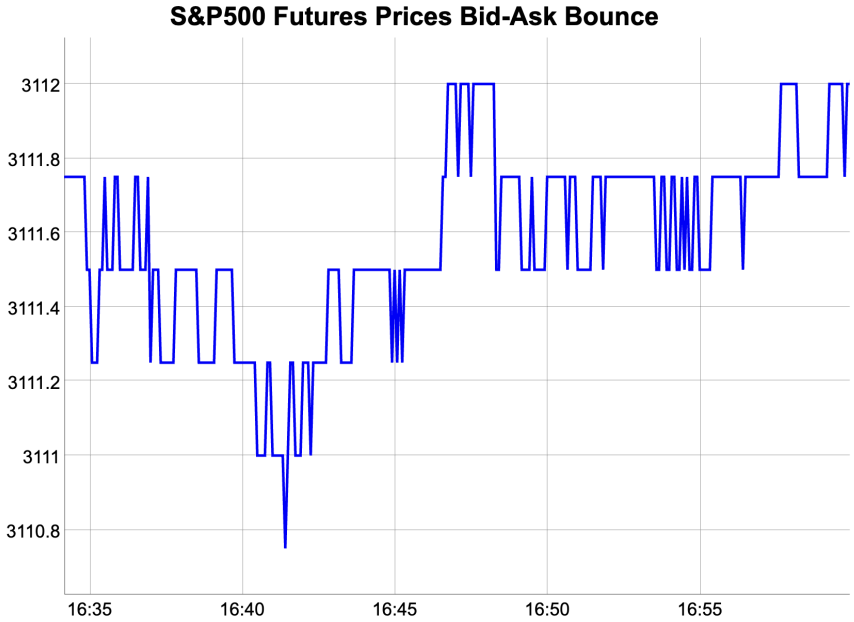
The *bid-ask bounce* is prominent at very high frequency time scales or in periods of low volatility.

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

The *bid-ask bounce* inflates the estimates of realized volatility, above the actual volatility.

The *bid-ask bounce* creates 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)
> dygraphs::dygraph(pricev$Close,
+   main="S&P500 Futures Prices Bid-Ask Bounce") %>%
+   dyOptions(colors="blue", strokeWidth=2)
```



Price Spikes And Trading Volumes in High Frequency Data

The number of the *price spikes* depends on the level of trading volumes, with the number decreasing with higher trading volumes.

The number of price spikes is lower for trade prices with larger trading volumes.


```
> # Plot dygraph of trade prices of at least 100 shares
> dygraphs::dygraph(xlk$price[xlk$volume >= 100, ],
+   main="XLK Prices for Trades of At Least 100 Shares") %>%
+   dyOptions(colors="blue", strokeWidth=1)
```

Removing Odd Lot Trades From *TAQ* Data

The trading volumes represent the number of shares traded at a given price.

The histogram of the trading volumes shows that the highest frequencies are for 100 shares and for round lots (trades that are multiples of 100 shares.)

There are also significant frequencies for *odd lots*, with small volumes 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 spikes, bid-ask bounce) in high frequency data.


```
> # Select the large trade lots of at least 100 shares
> dim(taq)
> tickb <- taq[taq$volume >= 100]
> dim(tickb)
> # Number of large lot ticks per second
> NROW(tickb)/(6.5*3600)
> # Plot histogram of the trading volumes
> hist(tickb$volume, main="Histogram of XLK Trading Volumes",
+      breaks=100000, xlim=c(1, 400), xlab="number of shares")
> # Save trade ticks with large lots
> data.table::fwrite(tickb, file="/Users/jerzy/Develop/data/xlk_tick_trades_20200316_biglots.csv")
> # Coerce trade prices to xts
> xlkb <- xts::xts(tickb[, .(price, volume)], tickb$index)
> colnames(xlkb) <- c("price", "volume")
```

```
> # Plot dygraph of the large lots
> dygraphs::dygraph(xlkb$price,
+   main="XLK Prices for Trades of At Least 100 Shares") %>%
+   dyOptions(colors="blue", strokeWidth=1)
> # Plot the large lots
> x11(width=6, height=5)
> quantmod::chart_Series(x=xlk$price,
+   name="XLK Trade Ticks for 2020-03-16 (large lots only)")
```

The Hampel Filter For Filtering Price Spikes

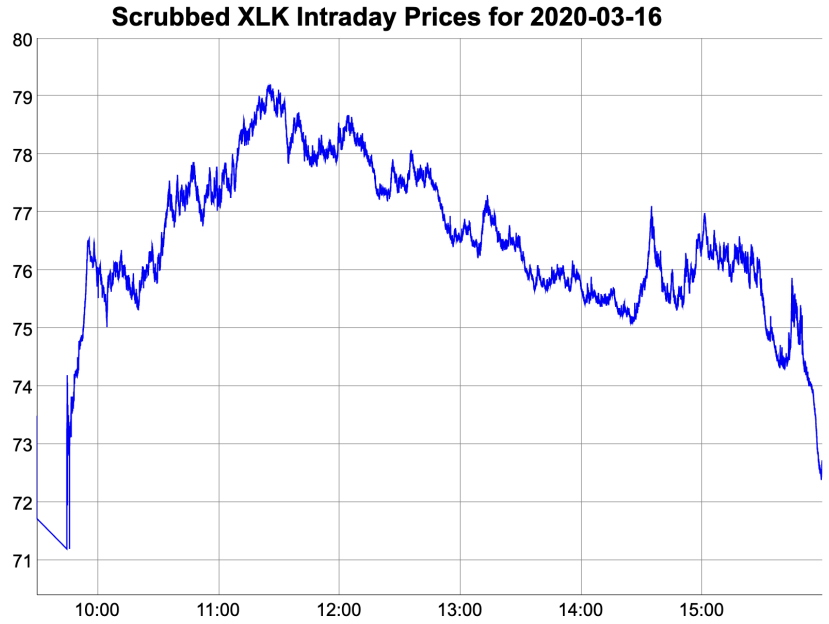
Price spikes in high frequency data can be identified using a *Hampel filter*.

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

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

If the absolute value of the *z-score* exceeds the *threshold value* then it's classified as *bad data*, and it can be removed or replaced.

```
> # Calculate the centered Hampel filter to remove bad prices
> lookb <- 71 # Look-back interval
> halfb <- lookb %/% 2 # Half-back interval
> pricev <- xlk$price
> # Calculate the trailing median and MAD
> medianv <- HighFreq::roll_mean(pricev, lookb=lookb, method="nonpa
> colnames(medianv) <- c("median")
> madv <- HighFreq::roll_var(pricev, lookb=lookb, method="nonparame
> # madv <- TTR::runMAD(pricev, n=lookb)
> # Center the median and the MAD
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> # Calculate the Z-scores
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=50000, xlim=c(-2*madz, 2*madz))
```




```
> # Define discrimination threshold value
> threshv <- 6*madz
> # Identify good prices with small z-scores
> isgood <- (abs(zscores) < threshv)
> # Calculate the number of bad prices
> sum(!isgood)
> # Scrub bad prices by replacing them with previous good prices
> priceg <- pricev
> priceg[!isgood] <- NA
> priceg <- zoo::na.locf(priceg)
> # Plot dygraph of the scrubbed prices
> dygraphs::dygraph(priceg, main="Scrubbed XLK Intraday Prices") %>%
+   dyOptions(colors="blue", strokeWidth=1)
> # Plot using chart_Series()
> x11(width=6, height=5)
> quantmod::chart_Series(x=priceg,
+   name="Clean XLK Intraday Prices for 2020-03-16")
```

Classifying Data Outliers Using the Hampel Filter

The Hampel filter is a *classifier* which classifies the prices as either good or bad data points.

In order to measure the performance of the Hampel filter, we add price spikes to the clean prices, to see how accurately they're classified.

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

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

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

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

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

```
> # Add 200 random price spikes to the clean prices
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nspikes <- 200
> nrows <- NROW(priceg)
> ispike <- logical(nrows)
> ispike[sample(x=nrows, size=nspikes)] <- TRUE
> priceb <- priceg
> priceb[ispike] <- priceb[ispike]*
+   sample(c(0.999, 1.001), size=nspikes, replace=TRUE)
> # Plot the bad prices and their medians
> medianv <- HighFreq::roll_mean(priceb, lookb=lookb, method="nonpar
> pricem <- cbind(priceb, medianv)
> colnames(pricem) <- c("prices with spikes", "median")
> dygraphs::dygraph(pricem, main="XLK Prices With Spikes") %>%
+   dyOptions(colors=c("red", "blue"))
> # Calculate the z-scores
> madv <- HighFreq::roll_var(priceb, lookb=lookb, method="nonparamet
> zscores <- ifelse(madv > 0, (priceb - medianv)/madv, 0)
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=10000, xlim=c(-4*madz, 4*madz))
> # Identify good prices with small z-scores
> threshv <- 3*madz
> isgood <- (abs(zscores) < threshv)
> # Calculate the number of bad prices
> sum(!isgood)
```

Confusion Matrix of a Binary Classification Model

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

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

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

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

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

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

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

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

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

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

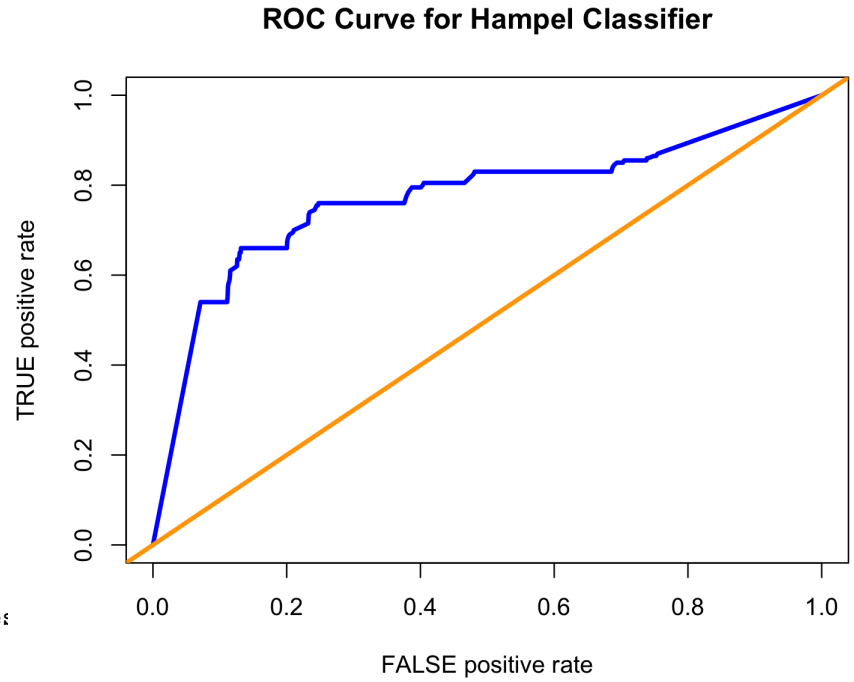
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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

The area under the *ROC curve* (AUC) measures the classification ability of a binary classifier.

```
> # Confusion matrix as function of threshold
> confun <- function(actualv, zscores, threshv) {
+     confmat <- table(actualv, (abs(zscores) < threshv))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+ }  # end confun
> confun(!ispike, zscores, threshv=threshv)
> # Define vector of discrimination thresholds
> threshv <- madz*seq(from=0.1, to=3.0, by=0.05)/2
> # Calculate the error rates
> errorr <- sapply(threshv, confun, actualv=!ispike, zscores=zscores
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
> # Calculate the area under the ROC curve (AUC)
> truepos <- (1 - errorr[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(errorr[, "typeI"])
> abs(sum(truepos*falsepos))
```



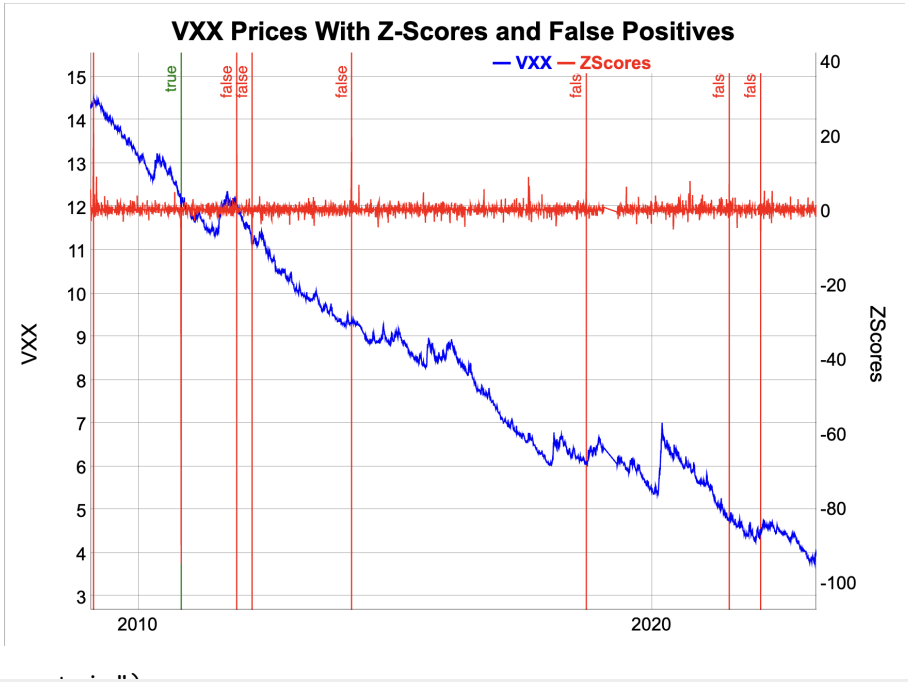

```
> # Plot ROC curve for Hampel classifier
> plot(x=errorr[, "typeI"], y=1-errorr[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Hampel Classifier",
+      type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3,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

Filtering Bad Data From Daily Stock Prices

Daily stock prices can also contain bad data points consisting of mostly single, isolated spikes in prices.

The number of false positives may be too high, so the Hampel filter parameters (the look-back interval and the threshold) need adjustment.

For example, the *VXX* has only one bad price (on 2010-11-08), but the Hampel filter identifies many more than that (which are false positives).




```
> # Load log VXX prices
> load("/Users/jerzy/Develop/lecture_slides/data/pricevxx.RData")
> nrows <- NROW(pricev)
> # Calculate the centered Hampel filter for VXX
> lookb <- 7 # Look-back interval
> halfb <- lookb %/% 2 # Half-back interval
> medianv <- HighFreq::roll_mean(pricev, lookb=lookb, method="nonpa
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(pricev, lookb=lookb, method="nonparame
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=100, xlim=c(-3*madz, 3*madz))
> # Define discrimination threshold value
> threshv <- 9*madz
> # Calculate the good prices
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> # Dates of the bad prices
> zoo::index(pricev[!isgood])
```

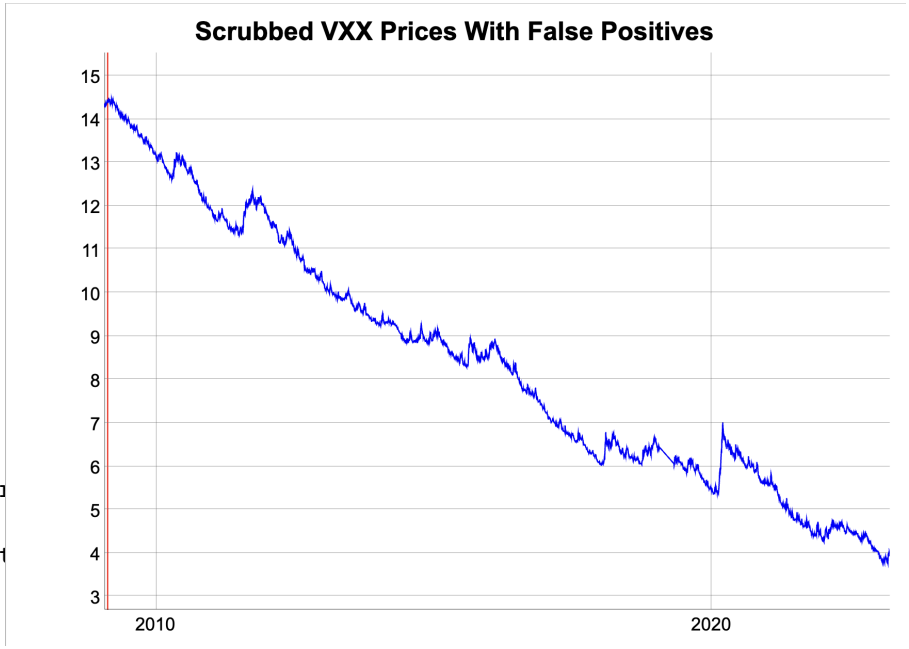
```
> # Calculate the false positives
> falsep <- !isgood
> falsep[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALS
> # Plot dygraph of the prices with bad prices
> datam <- cbind(pricev, zscores)
> colnames(datam)[2] <- "ZScores"
> colnamev <- colnames(datam)
> dygraphs::dygraph(datam, main="VXX Prices With Z-Scores and False
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeW
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], stroke
+   dyEvent(zoo::index(pricev[falsep]), label=rep("false", sum(fals
+   dyEvent(zoo::index(pricev["2010-11-08"]), label="true", strokeP
```

Scrubbing Bad Stock Prices

Bad stock prices can be scrubbed (replaced) with the previous good price.

But it's incorrect to replace bad prices with the average of the previous good price and the next good price, since that would cause data snooping.

```
> # Replace bad stock prices with the previous good prices
> priceg <- pricev
> priceg[!isgood] <- NA
> priceg <- zoo::na.locf(priceg)
> # Calculate the Z-scores
> medianv <- HighFreq::roll_mean(priceg, lookb=lookb, method="nonpar
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(priceg, lookb=lookb, method="nonparamet
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (priceg - medianv)/madv, 0)
> madz <- mad(zscores[abs(zscores) > 0])
> # Calculate the number of bad prices
> threshv <- 9*madz
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> zoo::index(priceg[!isgood])
```



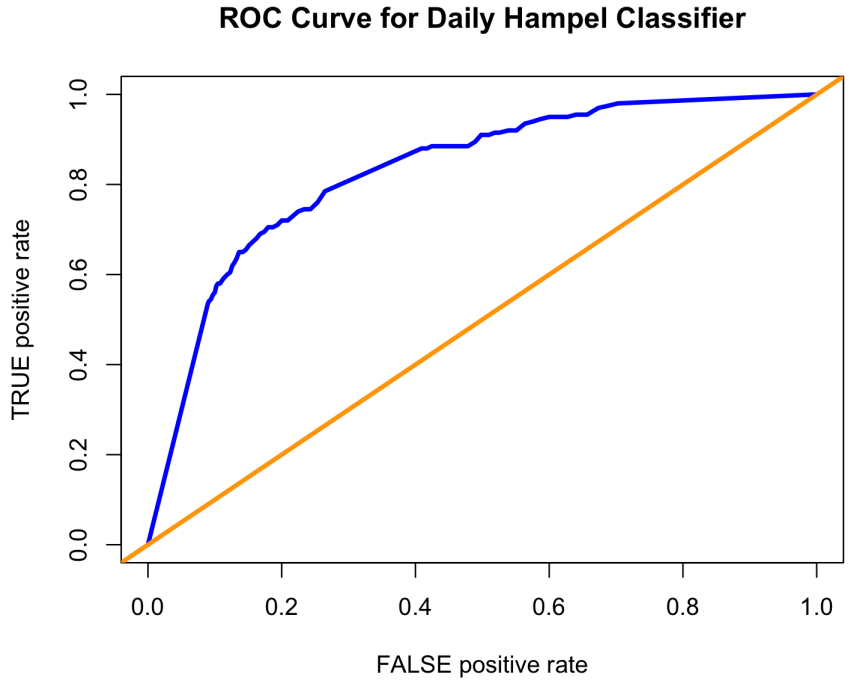

```
> # Calculate the false positives
> falsep <- !isgood
> falsep[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALS
> # Plot dygraph of the prices with bad prices
> dygraphs::dygraph(priceg, main="Scrubbed VXX Prices With False Pos
+   dyEvent(zoo::index(priceg[falsep]), label=rep("false", sum(false
+   dyOptions(colors="blue", strokeWidth=1)
```

ROC Curve for Daily Hampel Classifier

In order to measure the performance of the Hampel filter, we add price spikes to the clean prices, to see how accurately they're classified.

The performance of the Hampel noise classification model depends on the length of the look-back time interval.

The optimal *look-back interval* and *threshold value* can be determined using *cross-validation*.




```
> # Add 200 random price spikes to the clean prices
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nspikes <- 200
> ispike <- logical(nrows)
> ispike[sample(x=nrows, size=nspikes)] <- TRUE
> priceb <- priceg
> priceb[ispike] <- priceb[ispike]*
+   sample(c(0.99, 1.01), size=nspikes, replace=TRUE)
> # Calculate the Z-scores
> medianv <- HighFreq::roll_mean(priceb, lookb=lookb, method="nonparametric")
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(priceb, lookb=lookb, method="nonparame
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (priceb - medianv)/madv, 0)
> madz <- mad(zscores[abs(zscores) > 0])
> # Define vector of discrimination thresholds
> threshv <- madz*seq(from=0.1, to=3.0, by=0.05)/2
> # Calculate the error rates
> errorr <- sapply(threshv, confun, actualv=!ispike, zscores=zscor
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
```

```
> # Calculate the area under the ROC curve (AUC)
> truepos <- (1 - errorr[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(errorr[, "typeI"])
> abs(sum(truepos*falsepos))
> # Plot ROC curve for Hampel classifier
> plot(x=errorr[, "typeI"], y=1-errorr[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Daily Hampel Classifier",
+      type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

Package *data.table* for High Performance Data Management

The package *data.table* is designed for high performance data management.

The package *data.table* implements *data table* objects, which are a special type of *data frame*, and an extension of the *data frame* class.

Data tables are faster and more convenient to work with than *data frames*.

data.table functions are optimized for high performance (speed), because they are written in C++ and they perform operations by reference (in place), without copying data in memory.

Some of the attractive features of package *data.table* are:

- Syntax is analogous to SQL,
- Very fast writing and reading from files,
- Very fast sorting and merging operations,
- Subsetting using multiple logical clauses,
- Columns of type `character` are never converted to factors,

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

The package *data.table* has extensive documentation:
https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
https://github.com/Rdatatable/data.table/wiki

Data Table Objects

Data table objects are a special type of *data frame*, and are derived from the class `data.frame`.

Data table objects resemble databases, with columns of different types of data, and rows of records containing individual observations.

The function `data.table::data.table()` creates a *data table* object.

Data table columns can be referenced directly by their names (without quotes), and their rows can be referenced without a following comma.

When a *data table* is printed (by typing its name) then only the top 5 and bottom 5 rows are displayed (unless `getOption("datatable.print.nrows")` is less than 100).

The operator `.N` returns the number of observations (rows) in the *data table*.

Data table computations are usually much faster than equivalent `R` computations, but not always.

```
> # Create a data table
> library(data.table)
> dtable <- data.table::data.table(
+   col1=sample(7), col2=sample(7), col3=sample(7))
> # Print dtable
> class(dtable); dtable
> # Column referenced without quotes
> dtable[, col2]
> # Row referenced without a following comma
> dtable[2]
> # Print option "datatable.print.nrows"
> getOption("datatable.print.nrows")
> options(datatable.print.nrows=10)
> getOption("datatable.print.nrows")
> # Number of rows in dtable
> NROW(dtable)
> # Or
> dtable[, NROW(col1)]
> # Or
> dtable[, .N]
> # microbenchmark speed of data.table syntax
> library(microbenchmark)
> summary(microbenchmark(
+   dt=dtable[, .N],
+   rcode=NROW(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Writing and Reading Data Using Package *data.table*

The easiest way to share data between `R` and `Excel` is through `.csv` files.

The function `data.table::fread()` reads from `.csv` files and returns a *data table* object of class `data.table`.

Data table objects are a special type of *data frame*, and are derived from the class `data.frame`.

The function `data.table::fread()` is over 6 times faster than `read.csv()`!

The function `data.table::fwrite()` writes to `.csv` files over 12 times faster than the function `write.csv()`, and 300 times faster than function `cat()`!

```
> # Read a data table from CSV file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> filen <- file.path(dirn, "weather_delays14.csv")
> dtable <- data.table::fread(filen)
> class(dtable); dim(dtable)
> dtable
> # fread() reads the same data as read.csv()
> all.equal(read.csv(filen),
+     setDF(data.table::fread(filen)))
> # fread() is much faster than read.csv()
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=read.csv(filen),
+   fread=setDF(data.table::fread(filen)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Write data table to file in different ways
> data.table::fwrite(dtable, file="dtable.csv")
> write.csv(dtable, file="dtable2.csv")
> cat(unlist(dtable), file="dtable3.csv")
> # microbenchmark speed of data.table::fwrite()
> summary(microbenchmark(
+   fwrite=data.table::fwrite(dtable, file="dtable.csv"),
+   write_csv=write.csv(dtable, file="dtable2.csv"),
+   cat=cat(unlist(dtable), file="dtable3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Subsetting *Data Table* Objects

The square braces (brackets) "[]" operator subsets (references) the rows and columns of *data tables*.

Data table rows can be subset without a following comma.

Data table columns can be referenced directly by their names (without quotes, as if they were variables), after a comma.

Multiple *data table* columns can be referenced by passing a list of names.

The brackets "[]" operator is a *data.table* function, and all the commands inside the brackets "[]" are executed using code from the package *data.table*.

The dot `.()` operator is equivalent to the list function `list()`.

```
> # Select first five rows of dtable
> dtable[1:5]
> # Select rows with JFK flights
> jfkf <- dtable[origin=="JFK"]
> # Select rows JFK flights in June
> jfkf <- dtable[origin=="JFK" & month==6]
> # Select rows without JFK flights
> jfkf <- dtable[!(origin=="JFK")]
> # Select flights with carrier_delay
> dtable[carrier_delay > 0]
> # Select column of dtable and return a vector
> head(dtable[, origin])
> # Select column of dtable and return a dtable, not vector
> head(dtable[, list(origin)])
> head(dtable[, .(origin)])
> # Select two columns of dtable
> dtable[, list(origin, month)]
> dtable[, .(origin, month)]
> columnv <- c("origin", "month")
> dtable[, ..columnv]
> dtable[, month, origin]
> # Select two columns and rename them
> dtable[, .(orig=origin, mon=month)]
> # Select all columns except origin
> head(dtable[, !"origin"])
> head(dtable[, -"origin"])
```

Performing Computations on *Data Table* Columns

If the second argument in the brackets "[]" operator is a function of the columns, then the brackets return the result of the function's computations on those columns.

The second argument in the brackets "[]" can also be a list of functions, in which case the brackets return a vector of computations.

The brackets "[]" can evaluate most standard R functions, but they are executed using *data.table* code, which is usually much faster than the equivalent R functions.

The operator .N returns the number of observations (rows) in the *data table*.

```
> # Select flights with positive carrier_delay
> dtable[carrier_delay > 0]
> # Number of flights with carrier_delay
> dtable[, sum(carrier_delay > 0)]
> # Or standard R commands
> sum(dtable[, carrier_delay > 0])
> # microbenchmark speed of data.table syntax
> summary(microbenchmark(
+   dt=dtable[, sum(carrier_delay > 0)],
+   rcode=sum(dtable[, carrier_delay > 0]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average carrier_delay
> dtable[, mean(carrier_delay)]
> # Average carrier_delay and aircraft_delay
> dtable[, .(carrier=mean(carrier_delay),
+      aircraft=mean(aircraft_delay))]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Number of flights from JFK
> dtable[origin=="JFK", NROW(aircraft_delay)]
> # Or
> dtable[origin=="JFK", .N]
> # In R
> sum(dtable[, origin]=="JFK")
```

Grouping *Data Table* Computations by Factor Columns

The *data table* brackets `"[]"` operator can accept three arguments: `[i, j, by]`

- `i`: the row index to select,
- `j`: a list of columns or functions on columns,
- `by`: the columns of factors to aggregate over.

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

The `"keyby"` argument is similar to `"by"`, but it sorts the output according to the categories used to group by.

Multiple *data table* columns can be referenced by passing a list of names.

The dot `.()` operator is equivalent to the list function `list()`.

```
> # Number of flights from each airport
> dtable[, .N, by=origin]
> # Same, but add names to output
> dtable[, .(flights=.N), by=.(airport=origin)]
> # Number of AA flights from each airport
> dtable[carrier=="AA", .(flights=.N), by=.(airport=origin)]
> # Number of flights from each airport and airline
> dtable[, .(flights=.N), by=.(airport=origin, airline=carrier)]
> # Average aircraft_delay
> dtable[, mean(aircraft_delay)]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Average aircraft_delay from each airport
> dtable[, .(delay=mean(aircraft_delay)), by=.(airport=origin)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      by=.(airport=origin, month=month)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      keyby=.(airport=origin, month=month)]
```

Sorting *Data Table* Rows by Columns

Standard `R` functions can be used inside the brackets `"[]"` operator.

The function `order()` calculates the permutation index, to sort a given vector into ascending order.

The function `setorder()` sorts the rows of a *data table* by reference (in place), without copying data in memory.

`setorder()` is over 10 times faster than `order()`, because it doesn't copy data in memory.

Several brackets `"[]"` operators can be chained together to perform several consecutive computations.

```
> # Sort ascending by origin, then descending by dest
> dtables <- dtable[order(origin, -dest)]
> dtables
> # Doesn't work outside dtable
> order(origin, -dest)
> # Sort dtable by reference
> setorder(dtable, origin, -dest)
> all.equal(dtable, dtables)
> # setorder() is much faster than order()
> summary(microbenchmark(
+   order=dtable[order(origin, -dest)],
+   setorder=setorder(dtable, origin, -dest),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average aircraft_delay by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+       by=.(month=month)]
> # Chained brackets to sort output by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+   by=.(month=month)][order(month)]
```

Subsetting, Computing, and Grouping *Data Table* Objects

The special symbol `.SD` selects a subset of a *data table*.

The symbol `.SDcols` specifies the columns to select by the symbol `.SD`.

Inside the brackets `"[]"` operator, the `.SD` symbol can be treated as a virtual *data table*, and standard `R` functions can be applied to it.

The `"by"` argument can be used to group the outputs produced by the functions applied to the `.SD` symbol.

If the symbol `.SDcols` is not defined, then the symbol `.SD` returns the remaining columns not passed to the `"by"` operator.

```
> # Select weather_delay and aircraft_delay in two different ways
> dtable[1:7, .SD,
+       .SDcols=c("weather_delay", "aircraft_delay")]
> dtable[1:7, .(weather_delay, aircraft_delay)]
> # Calculate mean of weather_delay and aircraft_delay
> dtable[, sapply(.SD, mean),
+       .SDcols=c("weather_delay", "aircraft_delay")]
> sapply(dtable[, .SD,
+       .SDcols=c("weather_delay", "aircraft_delay")], mean)
> # Return origin and dest, then all other columns
> dtable[1:7, .SD, by=.(origin, dest)]
> # Return origin and dest, then weather_delay and aircraft_delay co
> dtable[1:7, .SD, by=.(origin, dest),
+       .SDcols=c("weather_delay", "aircraft_delay")]
> # Return first two rows from each month
> dtable[, head(.SD, 2), by=.(month)]
> dtable[, head(.SD, 2), by=.(month),
+       .SDcols=c("weather_delay", "aircraft_delay")]
> # Calculate mean of weather_delay and aircraft_delay, grouped by o
> dtable[, lapply(.SD, mean),
+       by=.(origin),
+       .SDcols=c("weather_delay", "aircraft_delay")]
> # Or simply
> dtable[, .(weather_delay=mean(weather_delay),
+           aircraft_delay=mean(aircraft_delay)),
+       by=.(origin)]
```

Modifying *Data Table* Objects by Reference

The special assignment operator ":=" allows modifying *data table* columns by reference (in place), without copying data in memory.

The computations on columns by reference can be *grouped* over categories defined by one or more columns passed to the "by" argument.

The computations are recycled to fit the size of each group.

The selected parts of columns can also be modified by reference, by combining the `i` and `j` arguments.

The special symbols `.SD` and `.SDcols` can be used to perform computations on several columns.

Modifying by reference is several times faster than standard `R` assignment.

```
> # Add tot_delay column
> dtable[, tot_delay := (carrier_delay + aircraft_delay)]
> head(dtable, 4)
> # Delete tot_delay column
> dtable[, tot_delay := NULL]
> # Add max_delay column grouped by origin and dest
> dtable[, max_delay := max(aircraft_delay), by=.(origin, dest)]
> dtable[, max_delay := NULL]
> # Add date and tot_delay columns
> dtable[, c("date", "tot_delay") :=
+        list(paste(month, day, year, sep="/"),
+             (carrier_delay + aircraft_delay))]
> # Modify select rows of tot_delay column
> dtable[month == 12, tot_delay := carrier_delay]
> dtable[, c("date", "tot_delay") := NULL]
> # Add several columns
> dtable[, c("max_carrier", "max_aircraft") := lapply(.SD, max),
+  by=.(origin, dest),
+  .SDcols=c("carrier_delay", "aircraft_delay")]
> # Remove columns
> dtable[, c("max_carrier", "max_aircraft") := NULL]
> # Modifying by reference is much faster than standard R
> summary(microbenchmark(
+   dt=dtable[, tot_delay := (carrier_delay + aircraft_delay)],
+   rcode=(dtable[, "tot_delay"] <- dtable[, "carrier_delay"] + dtab
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Adding *keys* to *Data Tables* for Fast Binary Search

The *key* of a *data table* is analogous to the row indices of a *data frame*, and it determines the ordering of its rows.

The function `data.table::setkey()` adds a *key* to a *data table*, and sorts the *data table* rows by reference according to the key.

`setkey()` creates the *key* from one or more columns of the *data frame*.

Subsetting rows using a *key* can be several times faster than standard `R`.

```
> # Add a key based on the "origin" column
> setkey(dtable, origin)
> haskey(dtable)
> key(dtable)
> # Select rows with LGA using the key
> dtable["LGA"]
> all.equal(dtable["LGA"], dtable[origin == "LGA"])
> # Select rows with LGA and JFK using the key
> dtable[c("LGA", "JFK")]
> # Add a key based on the "origin" and "dest" columns
> setkey(dtable, origin, dest)
> key(dtable)
> # Select rows with origin from JFK and MIA
> dtable[c("JFK", "MIA")]
> # Select rows with origin from JFK and dest to MIA
> dtable[.("JFK", "MIA")]
> all.equal(dtable[.("JFK", "MIA")],
+     dtable[origin == "JFK" & dest == "MIA"])
> # Selecting rows using a key is much faster than standard R
> summary(microbenchmark(
+   with_key=dtable[.("JFK", "MIA")],
+   standard_r=dtable[origin == "JFK" & dest == "MIA"],
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing *Data Table* Objects Into *Data Frames*

The functions `data.table::setDT()` and `data.table::setDF()` coerce *data frames* to *data tables*, and vice versa.

The *set* functions `data.table::set*()` perform their operations by reference (in place), without returning any values or copying data to a new memory location, which makes them very fast.

Data table objects can also be coerced into *data frames* using the function `as.data.frame()`, but it's much slower because it makes copies of data.

```
> # Create data frame and coerce it to data table
> dtable <- data.frame(col1=sample(7), col2=sample(7), col3=sample(7
> class(dtable); dtable
> data.table::setDT(dtable)
> class(dtable); dtable
> # Coerce dtable into data frame
> data.table::setDF(dtable)
> class(dtable); dtable
> # Or
> dtable <- data.table:::as.data.frame.data.table(dtable)
> # SetDF() is much faster than as.data.frame()
> summary(microbenchmark(
+   asdataframe=data.table:::as.data.frame.data.table(dtable),
+   setDF=data.table::setDF(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing xts Time Series Into *Data Tables*

An xts time series can be coerced into a *data table* by first coercing it into a *data frame* and then into a *data table* using the function data.table::setDT().

But then the time index of the xts series is coerced into strings, not dates.

An xts time series can also be coerced directly into a *data table* using the function data.table::as.data.table().

```
> # Coerce xts to a data frame
> pricev <- rutils::etfenv$VTI
> class(pricev); head(pricev)
> pricev <- as.data.frame(pricev)
> class(pricev); head(pricev)
> # Coerce data frame to a data table
> data.table::setDT(pricev, keep.rownames=TRUE)
> class(pricev); head(pricev)
> # Dates are coerced to strings
> sapply(pricev, class)
> # Coerce xts directly to a data table
> dtable <- as.data.table(rutils::etfenv$VTI,
+   keep.rownames=TRUE)
> class(dtable); head(dtable)
> # Dates are not coerced to strings
> sapply(dtable, class)
> all.equal(pricev, dtable, check.attributes=FALSE)
```

Package *fst* for High Performance Data Management

The package *fst* provides functions for very fast writing and reading of *data frames* from *compressed binary files*.

The package *fst* writes to *compressed binary files* in the `fst` fast-storage format.

The package *fst* uses the `LZ4` and `ZSTD` compression algorithms, and utilizes multithreaded (parallel) processing on multiple CPU cores.

The package *fst* has extensive documentation:
http://www.fstpackage.org/

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

Writing and Reading Data Using Package *fst*

The package *fst* allows very fast writing and reading of *data frames* from *compressed binary files* in the `fst` fast-storage format.

The function `fst::write_fst()` writes to `.fst` files over 10 times faster than the function `write.csv()`, and 300 times faster than function `cat()` write to `.csv` files!

The function `fst::fread()` reads from `.fst` files over 10 times faster than the function `read.csv()` from `.csv` files!

```
> # Read a data frame from CSV file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> filen <- file.path(dirn, "weather_delays14.csv")
> data.table::setDF(dframe)
> class(dframe); dim(dframe)
> # Write data frame to .fst file in different ways
> fst::write_fst(dframe, path="dframe.fst")
> write.csv(dframe, file="dframe2.csv")
> # microbenchmark speed of fst::write_fst()
> library(microbenchmark)
> summary(microbenchmark(
+   fst=fst::write_fst(dframe, path="dframe.csv"),
+   write_csv=write.csv(dframe, file="dframe2.csv"),
+   cat=cat(unlist(dframe), file="dframe3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # fst::read_fst() reads the same data as read.csv()
> all.equal(read.csv(filen),
+     fst::read_fst("dframe.fst"))
> # fst::read_fst() is 10 times faster than read.csv()
> summary(microbenchmark(
+   fst=fst::read_fst("dframe.fst"),
+   read_csv=read.csv(filen),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Random Access to Large Data Files

The package *fst* allows *random access* to very large *data frames* stored in compressed data files in the `.fst` format.

Data frames can be accessed *randomly* by loading only the selected rows and columns into memory, without fully loading the whole data frame.

function `fst::fst()` reads an `.fst` file and returns an *fst_table* reference object (pointer) to the data, without loading the whole data into memory.

The *fst_table* reference provides access to the data similar to a regular *data frame*, but it requires only a small amount of memory because the data isn't loaded into memory.

```
> # Coerce TAQ xts to a data frame
> library(HighFreq)
> taq <- HighFreq::SPY_TAQ
> taq <- as.data.frame(taq)
> class(taq)
> # Coerce data frame to a data table
> data.table::setDT(taq, keep.rownames=TRUE)
> class(taq); head(taq)
> # Get memory size of data table
> format(object.size(taq), units="MB")
> # Save data table to .fst file
> fst::write_fst(taq, path="/Users/jerzy/Develop/data/taq.fst")
> # Create reference to .fst file similar to a data frame
> refst <- fst::fst("/Users/jerzy/Develop/data/taq.fst")
> class(refst)
> # Memory size of reference to .fst is very small
> format(object.size(refst), units="MB")
> # Get sizes of all objects in workspace
> sort(sapply(mget(ls()), object.size))
> # Reference to .fst can be treated similar to a data table
> dim(taq); dim(refst)
> fst:::print.fst_table(refst)
> # Subset reference to .fst just like a data table
> refst[1e4:(1e4+5), ]
```

Reading Data From Excel Files

The package *readxl* reads data from Excel spreadsheet files into R.

The function `read_excel()` reads a single sheet (tab) from an Excel file.

The function `read_xlsx()` reads a single sheet (tab) from an Excel file in `.xlsx` format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes `tbl` and `tbl_df` are derived from the *data frame* class `data.frame`.

tibble objects are also used by the package *dplyr*.

DataCamp offers a Tutorial on Importing Excel Files into R.

```
> # Install and load package readxl
> install.packages("readxl")
> library(readxl)
> dirn <- "/Users/jerzy/Develop/lecture_slides/data"
> filev <- file.path(dirn, "multi_tabs.xlsx")
> # Read a time series from first sheet of xlsx file
> tibblev <- readxl::read_xlsx(filev)
> class(tibblev)
> # Coerce POSIXct dates into Date class
> class(tibblev$Dates)
> tibblev$Dates <- as.Date(tibblev$Dates)
> # Some columns are character strings
> sapply(tibblev, class)
> sapply(tibblev, is.character)
> # Coerce columns with strings to numeric
> listv <- lapply(tibblev, function(x) {
+   if (is.character(x))
+     as.numeric(x)
+   else
+     x
+ })  # end lapply
> # Coerce list into xts time series
> xtsv <- xts::xts(do.call(cbind, listv)[, -1], listv[[1]])
> class(xtsv); dim(xtsv)
> # Replace NA values with the most recent non-NA values
> sum(is.na(xtsv))
> xtsv <- zoo::na.locf(xtsv, na.rm=FALSE)
> xtsv <- zoo::na.locf(xtsv, fromLast=TRUE)
```

Reading Multiple Sheets From Excel Files

The function `readxl::excel_sheets()` returns a vector of character strings with the names of all the sheets in an Excel spreadsheet.

The package *readxl* reads data from Excel spreadsheet files into R.

The function `read_excel()` reads a single sheet (tab) from an Excel file.

The function `read_xlsx()` reads a single sheet (tab) from an Excel file in `.xlsx` format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes `tbl` and `tbl_df` are derived from the *data frame* class `data.frame`.

tibble objects are also used by the package *dplyr*.

```
> # Read names of all the sheets in an Excel spreadsheet
> namev <- readxl::excel_sheets(filev)
> # Read all the sheets from an Excel spreadsheet
> sheets <- lapply(namev, read_xlsx, path=filev)
> names(sheets) <- namev
> # sheets is a list of tibbles
> sapply(sheets, class)
> # Create function to coerce tibble to xts
> to_xts <- function(tibblev) {
+   tibblev$Dates <- as.Date(tibblev$Dates)
+   # Coerce columns with strings to numeric
+   listv <- lapply(tibblev, function(x) {
+     if (is.character(x))
+       as.numeric(x)
+     else
+       x
+   })  # end lapply
+   # Coerce list into xts series
+   xts::xts(do.call(cbind, listv)[, -1], listv$Dates)
+ }  # end to_xts
> # Coerce list of tibbles to list of xts
> class(sheets)
> sheets <- lapply(sheets, to_xts)
> sapply(sheets, class)
> # Replace NA values with the most recent non-NA values
> sapply(sheets, function(xtsv) sum(is.na(xtsv)))
> sheets <- lapply(sheets, zoo::na.locf, na.rm=FALSE)
> sheets <- lapply(sheets, zoo::na.locf, fromLast=TRUE)
```

Performing Calculations in Excel Using R

Excel can run R using either VBA scripts, or through a *COM* interface (available on *Windows* only).

R can perform calculations and export its output to Excel files, or it can modify Excel files (requires packages using Java or Perl code).

Calculations in R and Excel can be combined in several different ways:

- Data from Excel can be exchanged with R via .csv files (simplest and best method),
- Excel can execute R commands using VBA scripts, and then import the R output from .csv files,
- An Excel add-in can execute R commands as Excel functions (relies on *COM* protocol, so works only for *Windows*): add-ins *BERT*, *RExcel*,
- R can modify Excel files and run Excel functions (requires packages using Java or Perl code): packages *xlsx*, *XLConnect*, *excel.link*,

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> all.equal(readf, dframe)
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code from Excel

There are several ways of performing calculations in R and exporting the outputs to Excel:

- Export data from Excel via .csv files to R, perform the calculations in R, and import the outputs back to Excel via .csv files (simplest and best method),
- Run R from Excel using VBA scripts, and exchange data via .csv files,
- Run R from Excel using an Excel add-in, and execute R commands as Excel functions (relies on the *COM* protocol, so works only for *Windows*),

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> all.equal(readf, dframe)
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code Using VBA Scripts

An R session can be launched from Excel using a VBA script (macro).

The VBA function shell() executes a program by running an executable *exe* file (with extension *exe*).

A VBA script can also run an R *batch* process.

The R *batch* process can write to .csv files, which can then be imported into Excel.

```
' VBA macro to run R process
Sub run_r()
 Call shell("R", vbNormalFocus)
End Sub
```

```
' VBA macro to run interactive R process
Sub run_rinteractive()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_interactive.
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

```
' VBA macro to run batch R process
Sub run_rbatch()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_to_file.R"
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

BERT Excel Add-in for Running R Code

BERT is an Excel add-in which allows executing R commands as Excel functions:

- http://bert-toolkit.com/
- http://bert-toolkit.com/bert-quick-start
- https://github.com/sdllc/Basic-Excel-R-Toolkit/wiki
- https://github.com/sdllc/Basic-Excel-R-Toolkit

BERT launches its own R process from Excel.

BERT can create its own menu in the Excel add-ins tab:
After installing *BERT*, click on upper-left *Office Button*, click Excel options, on the bottom of the window choose (Manage: *COM* Add-ins) Go, add the *COM* add-in BERTRibbon2x86.dll.

BERT relies on the *COM* protocol, so it works only for *Windows*.

```
' calculate sum of Excel cells using R
R.Add(B1:D1)

' remove NAs over Excel cell range using R function
R.na_omit(F2:H4)

' calculate eigenValues of Excel matrix using R function
R.EigenValues(A1:H8)
```

Package *googlesheets* for Interacting with *Google Sheets*

The package *googlesheets* allows interacting with *Google Sheets* using R commands.

If you already have a *Google* account, then your personal *Google Sheets* can be found at:
https://docs.google.com/spreadsheets/

The function `gs_ls()` listv the files in *Google Sheets*.

The function `gs_title()` registers a *Google* sheet, and returns a `googlesheet` object.

A `googlesheet` object contains information (metadata) about a *Google* sheet, such as its name and key, but not the sheet data itself.

The function `gs_browse()` opens a *Google* sheet in an internet browser.

You can find online a document about using *googlesheets*.

You can find online a document about managing authentication tokens.

```
> # Install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets")
> # Load package googlesheets
> library(googlesheets)
> library(dplyr)
> # Authenticate authorize R to view and manage your files
> gs_auth(new_user=TRUE)
> # List the files in Google Sheets
> googlesheets::gs_ls()
> # Register a sheet
> googsheet <- gs_title("my_data")
> # view sheet summary
> googsheet
> # List tab names in sheet
> tabv <- gs_ws_ls(googsheet)
> # Set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # Read data from sheet
> gs_read(googsheet)
> # Read data from single tab of sheet
> gs_read(googsheet, ws=tabv[1])
> gs_read_csv(googsheet, ws=tabv[1])
> # Or using dplyr pipes
> googsheet %>% gs_read(ws=tabv[1])
> # Download data from sheet into file
> gs_download(googsheet, ws=tabv[1],
+       to="/Users/jerzy/Develop/lecture_slides/data/googsheet.csv")
> # Open sheet in internet browser
> gs_browse(googsheet)
```

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

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

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

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

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

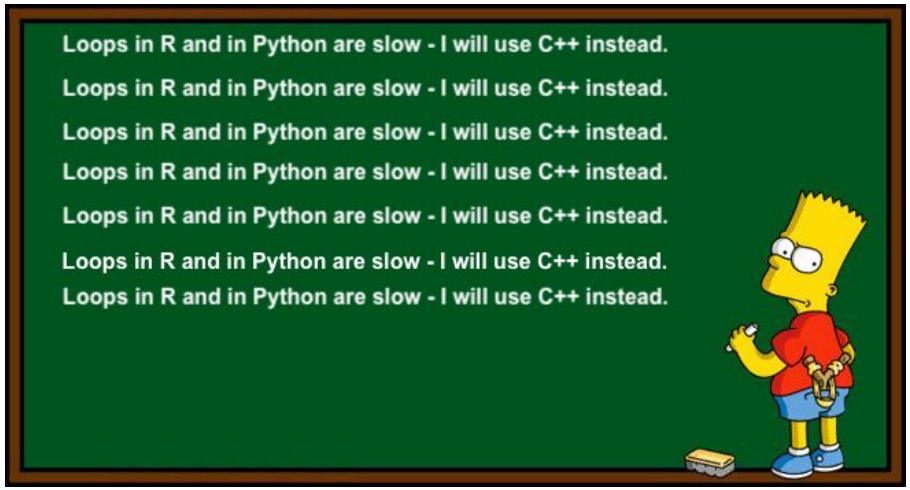
https://cran.r-project.org/bin/windows/Rtools/

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

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

http://www.rcpp.org/

http://gallery.rcpp.org/




```
> # Verify that Rtools or XCode are working properly:
> devtools::find_rtools()  # Under Windows
> devtools::has_devel()
> # Install the packages Rcpp and RcppArmadillo
> install.packages(c("Rcpp", "RcppArmadillo"))
> # Load package Rcpp
> library(Rcpp)
> # Get documentation for package Rcpp
> # Get short description
> packageDescription("Rcpp")
> # Load help page
> help(package="Rcpp")
> # List all datasets in "Rcpp"
> data(package="Rcpp")
> # List all objects in "Rcpp"
> ls("package:Rcpp")
> # Remove Rcpp from search path
> detach("package:Rcpp")
```

Function cppFunction() for Compiling C++ code

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

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

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

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

Performing Loops in *Rcpp Sugar*

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

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

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

```
> # Define R function with loop
> inner_multr <- function(x, y) {
+     sumv <- 0
+     for(i in 1:NROW(x)) {
+ sumv <- sumv + x[i] * y[i]
+     }
+     sumv
+ }  # end inner_multr
> # Run R function
> inner_multr(1:3, 6:4)
> inner_multr(1:3, 6:3)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=inner_multr(1:10000, 1:10000),
+   innerp=1:10000 %*% 1:10000,
+   Rcpp=inner_mult(1:10000, 1:10000),
+   sugar=inner_sugar(1:10000, 1:10000),
+   times=10))[, c(1, 4, 5)]
```

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

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

```
> # Define Ornstein-Uhlenbeck function in R
> sim_our <- function(nrows=1000, eq_price=5.0,
+               volat=0.01, theta=0.01) {
+   retp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- eq_price
+   for (i in 2:nrows) {
+     retp[i] <- theta*(eq_price - pricev[i-1]) + volat*rnorm(1)
+     pricev[i] <- pricev[i-1] + retp[i]
+   }  # end for
+   pricev
+ }  # end sim_our
> # Simulate Ornstein-Uhlenbeck process in R
> eq_price <- 5.0; sigmav <- 0.01
> thetav <- 0.01; nrows <- 1000
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # R
> ousim <- sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=t
```

```
> # Define Ornstein-Uhlenbeck function in Rcpp
> Rcpp::cppFunction("
+ NumericVector sim_oucpp(double eq_price,
+                   double volat,
+                   double thetav,
+                   NumericVector innov) {
+   int nrows = innov.size();
+   NumericVector pricev(nrows);
+   NumericVector retv(nrows);
+   pricev[0] = eq_price;
+   for (int it = 1; it < nrows; it++) {
+     retv[it] = thetav*(eq_price - pricev[it-1]) + volat*innov[it-1
+     pricev[it] = pricev[it-1] + retv[it];
+   }  // end for
+   return pricev;
+ }")  # end cppFunction
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Res
> oucpp <- sim_oucpp(eq_price=eq_price,
+   volat=sigmav, theta=thetav, innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=thet
+   Rcpp=sim_oucpp(eq_price=eq_price, volat=sigmav, theta=thetav, in
+   times=10))[, c(1, 4, 5)]
```

Rcpp Attributes

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

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

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

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

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

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

// The function sim_oucpp() simulates an Ornstein-Uhlenb
// export the function roll_maxmin() to R
// [[Rcpp::export]]
NumericVector sim_oucpp(double eq_price,
                        double volat,
                        double thetav,
                        NumericVector innov) {
  int(nrows = innov.size();
  NumericVector pricev*nrows);
  NumericVector retp*nrows);
  pricev[0] = eq_price;
  for (int it = 1; it < nrows; it++) {
    retp[it] = thetav*(eq_price - pricev[it-1]) + volat*
    pricev[it] = pricev[it-1] + retp[it];
  }  // end for
  return pricev;
}  // end sim_oucpp
```

```
> # Source Rcpp function for Ornstein-Uhlenbeck process from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Reset random numbers
> oucpp <- sim_oucpp(eq_price=eq_price,
+   volat=sigmav,
+   theta=thetav,
+   innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=thetav),
+   Rcpp=sim_oucpp(eq_price=eq_price, volat=sigmav, theta=thetav, innov=rnorm(nrows)),
+   times=10))[, c(1, 4, 5)]
```

Generating Random Numbers Using Logistic Map in *Rcpp*

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

```
> # Calculate uniformly distributed pseudo-random sequence
> unifun <- function(seedv, nrows=10) {
+   datav <- numeric(nrows)
+   datav[1] <- seedv
+   for (i in 2:nrows) {
+     datav[i] <- 4*datav[i-1]*(1-datav[i-1])
+   }  # end for
+   acos(1-2*datav)/pi
+ }  # end unifun
>
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts,
> # Microbenchmark Rcpp code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=runif(1e5),
+   rloop=unifun(0.3, 1e5),
+   Rcpp=unifuncpp(0.3, 1e5),
+   times=10))[, c(1, 4, 5)]
```

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

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

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

Package *RcppArmadillo* for Fast Linear Algebra

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

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

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

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

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

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

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

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vecv2, const arma::mat
  return arma::as_scalar(trans(vecv2) * (matv * vecv1));
}  // end inner_mat
```

```
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = inner_vec(vec1, vec2),
+   rcode = (vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Microbenchmark shows:
> # inner_vec() is several times faster than %*%, especially for lo
> #     expr     mean   median
> # 1 inner_vec 110.7067 110.4530
> # 2 rcode 585.5127 591.3575
```

Simulating *ARIMA* Processes Using *RcppArmadillo*

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

```
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Define AR(2) coefficients
> coeff <- c(0.9, 0.09)
> nrows <- 1e4
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> innov <- rnorm(nrows)
> # Simulate ARIMA using filter()
> arimar <- filter(x=innov, filter=coeff, method="recursive")
> # Simulate ARIMA using sim_ar()
> innov <- matrix(innov)
> coeff <- matrix(coeff)
> arimav <- sim_ar(coeff, innov)
> all.equal(drop(arimav), as.numeric(arimar))
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = sim_ar(coeff, innov),
+   filter = filter(x=innov, filter=coeff, method="recursive"),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

//' @export
// [[Rcpp::export]]
arma::vec sim_ar(const arma::vec& innov, const arma::vec
  uword nrows = innov.n_elem;
  uword lookb = coeff.n_elem;
  arma::vec arimav[nrows);

  // startup period
  arimav(0) = innov(0);
  arimav(1) = innov(1) + coeff(lookb-1) * arimav(0);
  for (uword it = 2; it < lookb-1; it++) {
    arimav(it) = innov(it) + arma::dot(coeff.subvec(lookb
  }  // end for

  // remaining periods
  for (uword it = lookb; it < nrows; it++) {
    arimav(it) = innov(it) + arma::dot(coeff, arimav.sub
  }  // end for

  return arimav;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

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

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

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> matv <- matrix(runif(1e5), nc=1e3)
> # Center matrix columns using apply()
> matd <- apply(matv, 2, function(x) (x-mean(x)))
> # Center matrix columns in place using Rcpp demeanr()
> demeanr(matv)
> all.equal(matd, matv)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = (apply(matv, 2, mean)),
+   rcpp = demeanr(matv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Perform matrix inversion
> # Create random positive semi-definite matrix
> matv <- matrix(runif(25), nc=5)
> matv <- t(matv) %*% matv
> # Invert the matrix
> matrixinv <- solve(matv)
> inv_mat(matv)
> all.equal(matrixinv, matv)
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcode = solve(matv),
+   rcpp = inv_mat(matv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

// Examples of \emph{RcppArmadillo} functions below

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

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

Fast Correlation Matrix Inverse Using *RcppArmadillo*

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

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("/Users/jerzy/Develop/lecture_slides/scripts/High
> # Calculate matrix of random returns
> matv <- matrix(rnorm(300), nc=5)
> # Regularized inverse of correlation matrix
> dimax <- 4
> cormat <- cor(matv)
> eigend <- eigen(cormat)
> invmat <- eigend$vectors[, 1:dimax] %*%
+   (t(eigend$vectors[, 1:dimax]) / eigend$values[1:dimax])
> # Regularized inverse using \emph{RcppArmadillo}
> invarma <- calc_inv(cormat, dimax=dimax)
> all.equal(invmat, invarma)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = {eigend <- eigen(cormat)
+ eigend$vectors[, 1:dimax] %*% (t(eigend$vectors[, 1:dimax]) / eige
+   rcpp = calc_inv(cormat, dimax=dimax),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

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

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& matv,
                   arma::uword dimax = 0, // Max number
                   double eigen_thresh = 0.01) { // Thre

  // Allocate SVD variables
  arma::vec svdval;  // Singular values
  arma::mat svdu, svdv;  // Singular matrices
  // Calculate the SVD
  arma::svd(svdu, svdval, svdv, tseries);
  // Calculate the number of non-small singular values
  arma::uword svdnum = arma::sum(svdval > eigen_thresh*a

  // If no regularization then set dimax to (svdnum - 1)
  if (dimax == 0) {
    // Set dimax
    dimax = svdnum - 1;
  } else {
    // Adjust dimax
    dimax = stdev::min(dimax - 1, svdnum - 1);
  }  // end if

  // Remove all small singular values
  svdval = svdval.subvec(0, dimax);
  svdu = svdu.cols(0, dimax);
  svdv = svdv.cols(0, dimax);

  // Calculate the regularized inverse from the SVD deco
  return svdv*arma::diagmat(1/svdval)*svdu.t();
```

Portfolio Optimization Using *RcppArmadillo*

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

```
// Fast portfolio optimization using matrix algebra and \emph{RcppArmadillo}
arma::vec calc_weights(const arma::mat& returns, // Asset returns
                       Rcpp::List controlv) { // List of portfolio optimization parameters

  // Apply different calculation methods for weights
  switch(calc_method(method)) {
  case methodenum::maxsharpe: {
    // Mean returns of columns
    arma::vec colmeans = arma::trans(arma::mean(returns, 0));
    // Shrink colmeans to the mean of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::mean(colmeans));
    // Calculate weights using regularized inverse
    weights = calc_inv(covmat, dimax, eigen_thresh)*colmeans;
    break;
  }  // end maxsharpe
  case methodenum::maxsharpemed: {
    // Median returns of columns
    arma::vec colmeans = arma::trans(arma::median(returns, 0));
    // Shrink colmeans to the median of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::median(colmeans));
    // Calculate weights using regularized inverse
    weights = calc_inv(covmat, dimax, eigen_thresh)*colmeans;
    break;
  }  // end maxsharpemed
  case methodenum::minvarlin: {
    // Minimum variance weights under linear constraint
    // Multiply regularized inverse times unit vector
    weights = calc_inv(covmat, dimax, eigen_thresh)*arma::ones(ncols);
    break;
  }  // end minvarlin
  case methodenum::minvarquad: {
    // Minimum variance weights under quadratic constraint
    // Calculate highest order principal component
    arma::vec eigenval;
    arma::mat eigenvec;
```

Strategy Backtesting Using *RcppArmadillo*

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

```
arma::mat back_test(const arma::mat& excess, // Asset excess returns
                    const arma::mat& returns, // Asset returns
                    Rcpp::List controlv, // List of portfolio optimization model parameters
                    arma::uvec startp, // Start points
                    arma::uvec endd, // End points
                    double lambdaf = 0.0, // Decay factor for averaging the portfolio weights
                    double coeff = 1.0, // Multiplier of strategy returns
                    double bidask = 0.0) { // The bid-ask spread

  double lambda1 = 1-lambdaf;
  arma::uword nweights = returns.n_cols;
  arma::vec weights(nweights, fill::zeros);
  arma::vec weights_past = ones(nweights)/stdev::sqrt(nweights);
  arma::mat pnls = zeros(returns.n_rows, 1);

  // Perform loop over the end points
  for (arma::uword it = 1; it < endd.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate the portfolio weights
    weights = coeff*calc_weights(excess.rows(startp(it-1), endd(it-1)), controlv);
    // Calculate the weights as the weighted sum with past weights
    weights = lambda1*weights + lambdaf*weights_past;
    // Calculate out-of-sample returns
    pnls.rows(endd(it-1)+1, endd(it)) = returns.rows(endd(it-1)+1, endd(it))*weights;
    // Add transaction costs
    pnls.row(endd(it-1)+1) -= bidask*sum(abs(weightv - weights_past))/2;
    // Copy the weights
    weights_past = weights;
  }  // end for

  // Return the strategy pnls
  return pnls;

}  // end back_test
```

Package *reticulate* for Running Python from RStudio

The package *reticulate* allows running Python functions and scripts from RStudio.

The package *reticulate* relies on Python for interpreting the Python code.

```
> # Install package reticulate
> install.packages("reticulate")
> # Start Python session
> reticulate::repl_python()
> # Exit Python session
> exit
```

You must set your Global Options in RStudio to your Python executable, for example:
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3.10

You can learn more about the package *reticulate* here:
https://rstudio.github.io/reticulate/

Running Python Under *reticulate*

```
"""
Script for loading OHLC data from a CSV file and plotting a candlestick plot.
"""
# Import packages
import pandas as pd
import numpy as np
import plotly.graph_objects as go
# Load OHLC data from csv file - the time index is formatted inside read_csv()
symbol = "SPY"
range = "day"
filename = "/Users/jerzy/Develop/data/" + symbol + "_" + range + ".csv"
ohlc = pd.read_csv(filename)
datev = ohlc.Date
# Calculate log stock prices
ohlc[["Open", "High", "Low", "Close"]] = np.log(ohlc[["Open", "High", "Low", "Close"]])
# Calculate moving average
lookback = 55
closep = ohlc.Close
pricema = closep.ewm(span=lookback, adjust=False).mean()
# Plotly simple candlestick with moving average
# Create empty graph object
plotfig = go.Figure()
# Add trace for candlesticks
plotfig = plotfig.add_trace(go.Candlestick(x=datev,
  open=ohlc.Open, high=ohlc.High, low=ohlc.Low, close=ohlc.Close,
  name=symbol+" Log OHLC Prices", showlegend=False))
# Add trace for moving average
plotfig = plotfig.add_trace(go.Scatter(x=datev, y=pricema,
  name="Moving Average", line=dict(color="blue")))
# Customize plot
plotfig = plotfig.update_layout(title=symbol + " Log OHLC Prices",
  title_font_size=24, title_font_color="blue", yaxis_title="Price",
  font_color="black", font_size=18, xaxis_rangeslider_visible=False)
# Customize legend
plotfig = plotfig.update_layout(legend=dict(x=0.2, y=0.9, traceorder="normal",
  itemsizing="constant", font=dict(family="sans-serif", size=18, color="blue")))
# Render the plot
plotfig.show()
```

Homework Assignment

No homework!