R for Finance Primer

MAFS6010R- Portfolio Optimization with R MSc in Financial Mathematics Fall 2018-19, HKUST, Hong Kong

Prof. Daniel P. Palomar

Hong Kong University of Science and Technology (HKUST)

- What is R?
- Installation
- Packages
- Variables: vectors, matrices, data frames, and lists
- Package **xts**
- Package quantmod
- Package TTR
- Package PerformanceAnalytics
- Example of a technical trading strategy combining the packages xts, quantmod, TTR, and PerformanceAnalytics
- R Scripts and R Markdown
- To explore further

This R session will introduce the basics on R for finance.

What is R?

- R is a language and environment for statistical computing and graphics.
- R is based on the S language originally developed by John Chambers and colleagues at AT&T Bell Labs in the late 1970s and early 19802.
- R (sometimes called "GNU S") is free open source software licensed under the GNU general public license (GPL 2).
- R development was initiated by Robert Gentleman and Ross Ihaka at the University of Auckland, New Zealand.
- R is formally known as The R Project for Statistical Computing
- Useful R links:
    - R homepage (https://www.r-project.org/)
    - Comprehensive R Archive Network (CRAN) (https://cran.r-project.org/)
    - METACRAN (https://r-pkg.org/)
    - R documentation (https://www.rdocumentation.org/)
    - RStudio (https://www.rstudio.com/)
    - Cookbook R (http://www.cookbook-r.com/)
    - Quick-R (https://www.statmethods.net/)
    - R tutorial (https://www.statmethods.net/r-tutorial/index.html)
    - R seek (http://rseek.org/)
    - R-bloggers (https://www.r-bloggers.com/)

- Stack Overflow (https://stackoverflow.com/)
- Blog from David Smith of Revolution (http://blog.revolutionanalytics.com/)
- Official R Manual (https://cran.r-project.org/doc/manuals/R-lang.html)
- RMarkdown (http://rmarkdown.rstudio.com/index.html)
- Conference R/Finance (http://www.rinfinance.com/)
- Book: Processing and Analyzing Financial Data with R (https://msperlin.github.io/pafdR/)
- DataCamp free course Introduction to R (https://www.datacamp.com/courses/free-introduction-to-r)

Installation

To install, just follow the following simple steps:

1. Install R from CRAN (https://cran.r-project.org/).
2. Install the free code editor RStudio (https://www.rstudio.com/).

Now you are ready to start using R from within RStudio (note that you can also use R directly from the command line without the need for RStudio).

Packages

To see the versions of R and the installed packages just type `sessionInfo()`:

```
sessionInfo()
#> R version 3.5.0 (2018-04-23)
#> Platform: x86_64-apple-darwin15.6.0 (64-bit)
#> Running under: macOS High Sierra 10.13.6
#>
#> Matrix products: default
#> BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
#>
#> locale:
#> [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> loaded via a namespace (and not attached):
#>  [1] compiler_3.5.0  backports_1.1.2 magrittr_1.5    rprojroot_1.3-2
#>  [5] tools_3.5.0     htmltools_0.3.6 yaml_2.1.19     Rcpp_0.12.18
#>  [9] stringi_1.2.2   rmarkdown_1.9   knitr_1.20      stringr_1.3.1
#> [13] digest_0.6.15   evaluate_0.10.1
```

To see the version of a specific package use `packageVersion("package_name")`.

As time progresses, you will have to install different packages from CRAN with the command `install.packages("package_name")` or from GitHub with the command `devtools::install_github("package_name")`. After installing a package, it needs to be loaded before it can be used with the command `library("package_name")` or `library(package_name)`:

```
# let's try to use the function xts() from package xts:
x <- xts()
#> Error in xts() : could not find function "xts"

# let's try to load the package first:
library(xts)
#> Error in library(xts) : there is no package called ‘xts’

# let's first install it:
install.packages("xts")
```

```
# now we can load it and use it:
library(xts)
#> Loading required package: zoo
#>
#> Attaching package: 'zoo'
#> The following objects are masked from 'package:base':
#>
#>     as.Date, as.Date.numeric
x <- xts()
```

Variables: vectors, matrices, data frames, and lists

In R, we can easily assign a value to a variable or object with `<-` (if the variable does not exist it will be created):

```
x <- "Hello"
x
#> [1] "Hello"
```

We can combine several elements with `c()`:

```
y <- c("Hello", "everyone")
y
#> [1] "Hello"    "everyone"
```

We can always see the variables in memory with `ls()`:

```
ls()
#> [1] "x" "y"
```

My favorite command is `str(variable)`. It gives you various information about the variable, i.e., type of variable, dimensions, contents, etc.

```
str(x)
#>  chr "Hello"
str(y)
#>  chr [1:2] "Hello" "everyone"
```

Another useful pair of commands are `head()` and `tail()`. They are specially good for variables of large dimensions showing you the first and last few elements, respectively.

```
x <- c(1:1000)
str(x)
#>  int [1:1000] 1 2 3 4 5 6 7 8 9 10 ...
head(x)
#> [1] 1 2 3 4 5 6
tail(x)
#> [1]  995  996  997  998  999 1000
```

It is important to remark that R is a functional language where almost everything is done through functions of all sorts (such as `str()`, `print()`, `head()`, `ls()`, `tail()`, `max()`, etc.).

There are a variety of functions for getting help:

```
help(matrix)       # help about function matrix()
?matrix            # same thing
example(matrix)    # show an example of function matrix()
apropos("matrix")  # list all functions containing string "matrix"

# get vignettes of installed packages
vignette()       # show available vingettes
vignette("xts")  # show specific vignette
```

Data types

Operators in R: arithmetic operators include `+, -, *, /, ^` and logical operators `>, >=, ==, !=`.

R has a wide variety of data types including scalars, vectors, matrices, data frames, and lists.

Vectors

A vector is just a collection of several variables of the same type (numerical, character, logical, etc.).

```
a <- c(1, 2, 5.3, 6, -2, 4)  # numeric vector
b <- c("one", "two", "three")  # character vector
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE) # logical vector
```

Refer to elements of a vector using subscripts:

```
a[c(2, 4)]  # 2nd and 4th elements of vector
#> [1] 2 6
```

Note that in R vectors are not column vectors or row vectors, they do not have any orientation. If one desires a column vector, then that is actually an $n \times 1$ matrix.

It is also important to differentiate elementwise multiplication `*` from inner product `%*%` and outer product `%o%`:

```
x <- c(1, 2)
y <- c(10, 20)
x * y
#> [1] 10 40
x %*% y
#>      [,1]
#> [1,]   50
x %o% y
#>      [,1] [,2]
#> [1,]   10   20
#> [2,]   20   40
```

One can name the elements of a vector:

```
names(y)
#> NULL
names(y) <- c("convex", "optimization")
y  # same as print(y)
#>       convex optimization
#>           10           20
str(y)
#>  Named num [1:2] 10 20
#>  - attr(*, "names")= chr [1:2] "convex" "optimization"

# we can get the length
length(y)
#> [1] 2
```

Matrices

A matrix is two-dimensional collection of several variables of the same type (numerical, character, logical, etc.).

We can easily create a matrix with `matrix()` :

```r
# generate 5 x 4 numeric matrix
x <- matrix(1:20, nrow = 5, ncol = 4)
x
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    6   11   16
#> [2,]    2    7   12   17
#> [3,]    3    8   13   18
#> [4,]    4    9   14   19
#> [5,]    5   10   15   20

# we can name the columns and the rows
colnames(x) <- c("col1", "col2", "col3", "col4")
rownames(x) <- c("row1", "row2", "row3", "row4", "row5")
x
#>      col1 col2 col3 col4
#> row1    1    6   11   16
#> row2    2    7   12   17
#> row3    3    8   13   18
#> row4    4    9   14   19
#> row5    5   10   15   20
str(x)
#>  int [1:5, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:5] "row1" "row2" "row3" "row4" ...
#>   ..$ : chr [1:4] "col1" "col2" "col3" "col4"

# we can get the dimensions or number of rows/columns
dim(x)
#> [1] 5 4
nrow(x)
#> [1] 5
ncol(x)
#> [1] 4
```

Identify rows, columns or elements using subscripts:

```
x[, 4]  # 4th column of matrix (returned as vector)
#> row1 row2 row3 row4 row5
#>   16   17   18   19   20
str(x[, 4])
#>  Named int [1:5] 16 17 18 19 20
#>  - attr(*, "names")= chr [1:5] "row1" "row2" "row3" "row4" ...
x[, 4, drop = FALSE]  # 4th column of matrix (returned as one-column matrix)
#>      col4
#> row1   16
#> row2   17
#> row3   18
#> row4   19
#> row5   20
str(x[, 4, drop = FALSE])
#>  int [1:5, 1] 16 17 18 19 20
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:5] "row1" "row2" "row3" "row4" ...
#>   ..$ : chr "col4"
x[3, ]  # 3rd row of matrix
#> col1 col2 col3 col4
#>    3    8   13   18
x[2:4, 1:3]  # rows 2,3,4 of columns 1,2,3
#>      col1 col2 col3
#> row2    2    7   12
#> row3    3    8   13
#> row4    4    9   14
str(x[2:4, 1:3])
#>  int [1:3, 1:3] 2 3 4 7 8 9 12 13 14
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:3] "row2" "row3" "row4"
#>   ..$ : chr [1:3] "col1" "col2" "col3"
```

Arrays

Arrays are similar to matrices but can have more than two dimensions.

Data frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

```
d <- c(1, 2, 3, 4)
e <- c("red", "white", "red", NA)
f <- c(TRUE, TRUE, TRUE, FALSE)
myframe <- data.frame(d, e, f)
names(myframe) <- c("ID", "Color", "Passed")  # variable names
myframe
#>   ID Color Passed
#> 1  1   red   TRUE
#> 2  2 white   TRUE
#> 3  3   red   TRUE
#> 4  4  <NA>  FALSE
str(myframe)
#> 'data.frame':    4 obs. of  3 variables:
#>  $ ID    : num  1 2 3 4
#>  $ Color : Factor w/ 2 levels "red","white": 1 2 1 NA
#>  $ Passed: logi  TRUE TRUE TRUE FALSE
```

There are a variety of ways to identify the elements of a data frame:

```
myframe[c(1, 3)]  # columns 1,3 of data frame
#>   ID Passed
#> 1  1   TRUE
#> 2  2   TRUE
#> 3  3   TRUE
#> 4  4  FALSE
myframe[c("ID", "Color")] # columns ID and Color from data frame
#>   ID Color
#> 1  1   red
#> 2  2 white
#> 3  3   red
#> 4  4  <NA>
myframe["ID"]  # select column ID
#>   ID
#> 1  1
#> 2  2
#> 3  3
#> 4  4
myframe$ID  # extract variable ID in the data frame (as a vector), like myframe[["I
D"]]
#> [1] 1 2 3 4
```

Data frames in R are very powerful and versatile. They are commonly used in machine learning where each row is one observation and each column one variable (each variable can be of different types). For financial applications, we mainly deal with multivariate time series, which can be seen as a matrix or data frame but with some particularities: each row is an observation but in a specific order (properly indexed with dates or times) and each column is of the same time (numeric). For multivariate time series we will later explore the class `xts`, which is more appropriate than matrices or data frames.

Lists

A list is an ordered collection of objects (components) of (possibly) different types. A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components: a string, a numeric vector, a matrix, and a sca
lar
w <- list(name = "Fred", mynumbers = c(1:10), mymatrix = matrix(NA, 3, 3), age = 5.3)
w
#> $name
#> [1] "Fred"
#>
#> $mynumbers
#>  [1]  1  2  3  4  5  6  7  8  9 10
#>
#> $mymatrix
#>      [,1] [,2] [,3]
#> [1,]   NA   NA   NA
#> [2,]   NA   NA   NA
#> [3,]   NA   NA   NA
#>
#> $age
#> [1] 5.3
str(w)
#> List of 4
#>  $ name     : chr "Fred"
#>  $ mynumbers: int [1:10] 1 2 3 4 5 6 7 8 9 10
#>  $ mymatrix : logi [1:3, 1:3] NA NA NA NA NA NA ...
#>  $ age      : num 5.3
```

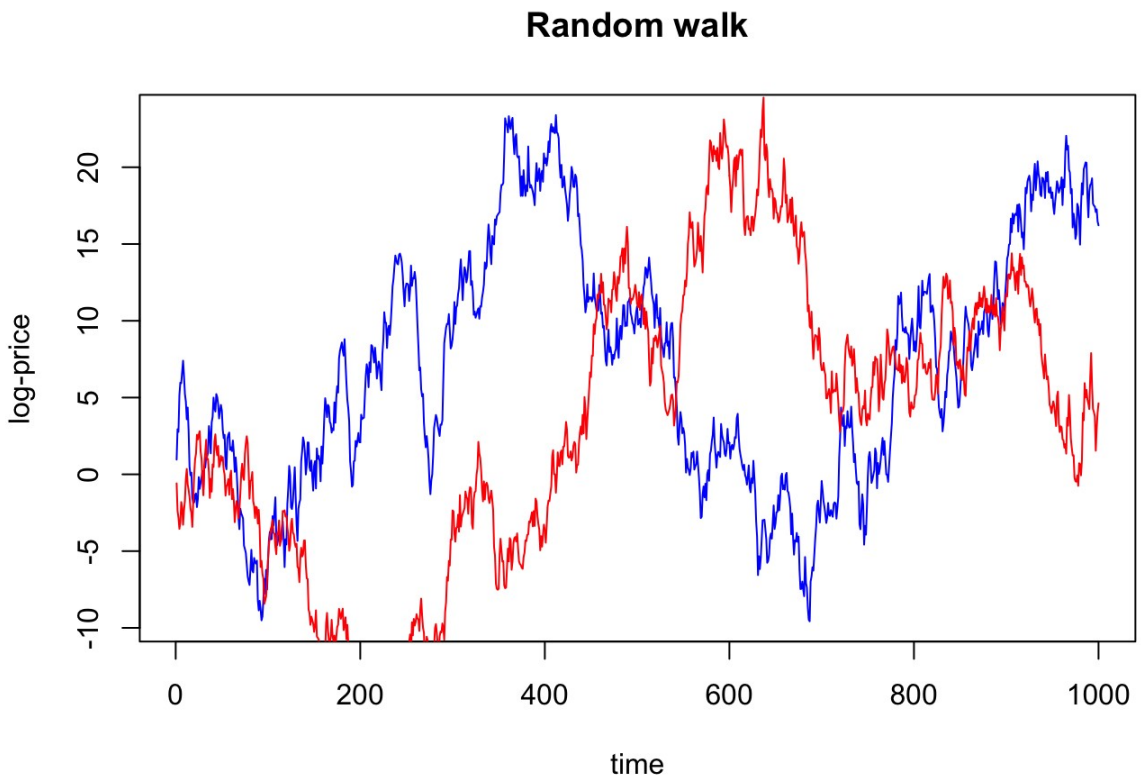
Useful functions

```
length(object)  # number of elements or components of a vector
str(object)     # structure of an object
class(object)   # class or type of an object
names(object)   # names

c(object, object, ...)       # combine objects into a vector
cbind(object, object, ...)   # combine objects as columns
rbind(object, object, ...)   # combine objects as rows

object          # prints the object
print(object)   # prints the object

ls()        # list current objects
rm(object)  # delete an object
```

Plotting

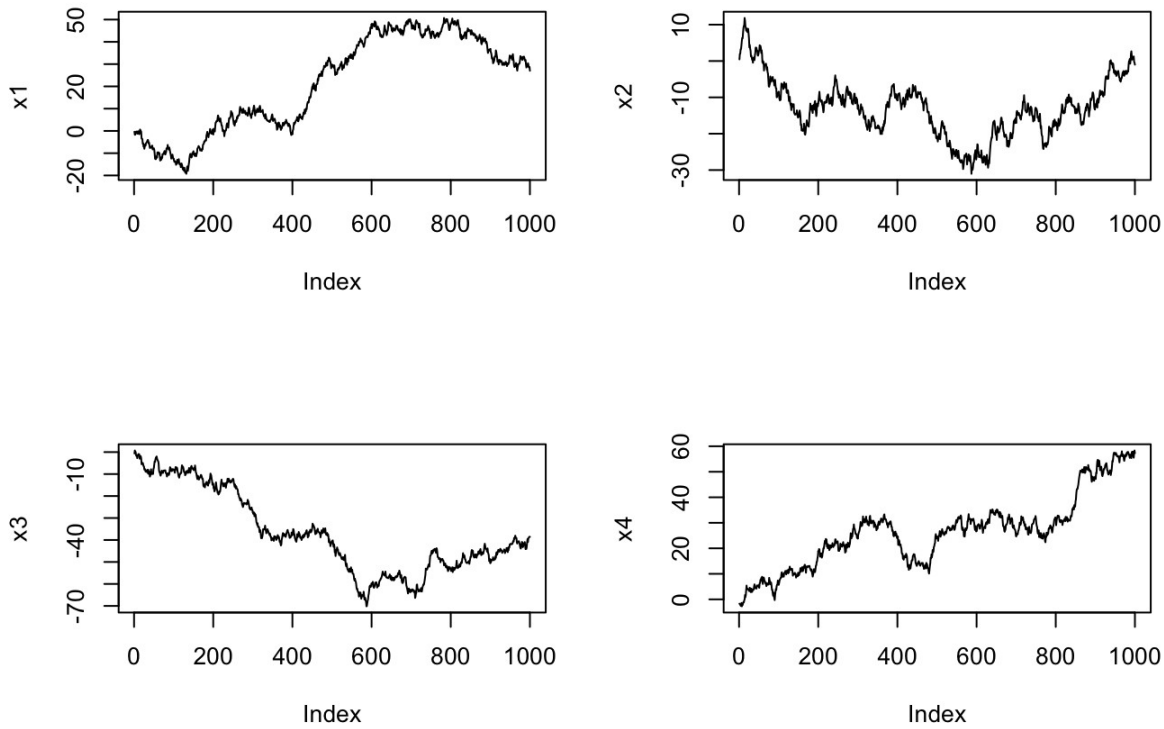
The native plotting functions in R are quite simple and may not look as appealing as desired:

```
x <- rnorm(1000)  # generate normal random numbers
x <- cumsum(x)   # random walk
plot(x, type = "l", col = "blue",
     main = "Random walk", xlab = "time", ylab = "log-price")
lines(cumsum(rnorm(1000)), col = "red")
```



For multiple plots:

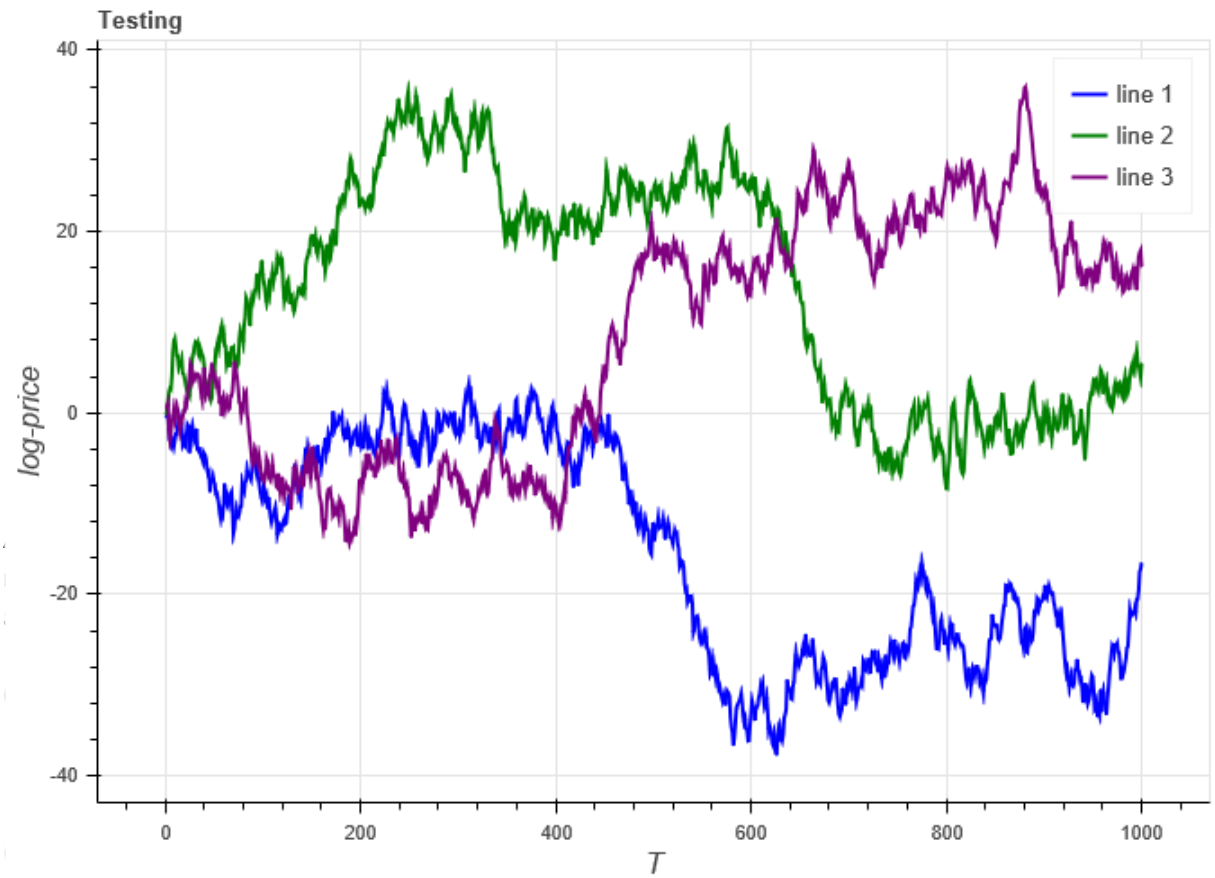
```
par(mfrow = c(2, 2))  # define a 2x2 matrix of plots
plot(cumsum(rnorm(1000)), type = "l", ylab = "x1")
plot(cumsum(rnorm(1000)), type = "l", ylab = "x2")
plot(cumsum(rnorm(1000)), type = "l", ylab = "x3")
plot(cumsum(rnorm(1000)), type = "l", ylab = "x4")
```




```
par(mfrow = c(1, 1))  # set it to default single plot
```

There are two highly recommended packages for plotting: **ggplot2** and **rbokeh**. The package **ggplot2** is extremely popular in the R community; it is particularly versatile and adapted to data frames, and defines a whole grammar for plotting. The package **rbokeh** (http://hafen.github.io/rbokeh/) is adopted from Python and allows for interactive plotting:

```
library(rbokeh)  # install.packages("rbokeh")

figure(width = 700, title = "Testing",
       xlab="T", ylab="log-price", legend_location = "top_right") %>%
  ly_lines(cumsum(rnorm(1000)), color = "blue", width = 2, legend = "line 1") %>%
  ly_lines(cumsum(rnorm(1000)), color = "green", width = 2, legend = "line 2") %>%
  ly_lines(cumsum(rnorm(1000)), color = "purple", width = 2, legend = "line 3")
```



One can easily convert an existing time series data into xts with `as.xts()`:

```
library(xts)

data(sample_matrix)  # load some data from package xts
class(sample_matrix)
#> [1] "matrix"
str(sample_matrix)
#>  num [1:180, 1:4] 50 50.2 50.4 50.4 50.2 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:180] "2007-01-02" "2007-01-03" "2007-01-04" "2007-01-05" ...
#>   ..$ : chr [1:4] "Open" "High" "Low" "Close"

matrix_xts <- as.xts(sample_matrix, dateFormat = "Date")
class(matrix_xts)
#> [1] "xts" "zoo"
str(matrix_xts)
#> An 'xts' object on 2007-01-02/2007-06-30 containing:
#>   Data: num [1:180, 1:4] 50 50.2 50.4 50.4 50.2 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:4] "Open" "High" "Low" "Close"
#>   Indexed by objects of class: [Date] TZ: UTC
#>   xts Attributes:
#>  NULL
```

Alternatively, one can create new data with the xts constructor `xts()`:

```
xts(1:10, as.Date("2000-01-01") + 1:10)
#>            [,1]
#> 2000-01-02    1
#> 2000-01-03    2
#> 2000-01-04    3
#> 2000-01-05    4
#> 2000-01-06    5
#> 2000-01-07    6
#> 2000-01-08    7
#> 2000-01-09    8
#> 2000-01-10    9
#> 2000-01-11   10
```

Subsetting xts

The most noticable difference in the behavior of xts objects will be apparent in the use of the "[" operator. Using special notation, one can use date-like strings to extract data based on the time-index. Using increasing levels of time-detail, it is possible to subset the object by year, week, days - or even seconds.

The i (row) argument to the subset operator "[", in addition to accepting numeric values for indexing, can also be a character string, a time-based object, or a vector of either. The format must left-specified with respect to the standard ISO:8601 time format — "CCYY-MM-DD HH:MM:SS" [5]. This means that for one to extract a particular month, it is necesssary to fully specify the year as well. To identify a particular hour, say all observations in the eighth hour on January 1, 2007, one would likewise need to include the full year, month and day - e.g. "2007-01-01 08".

It is also possible to explicitly request a range of times via this index-based subsetting, using the ISO-recommended "/" as the range separator. The basic form is "from/to", where both from and to are optional. If either side is missing, it is interpretted as a request to retrieve data from the beginning, or through the end of the data object.

Another benefit to this method is that exact starting and ending times need not match the underlying data - the nearest available observation will be re- turned that is within the requested time period.

The following example shows how to extract the entire month of March 2007:

```
matrix_xts["2007-03"]
#>                Open     High      Low    Close
#> 2007-03-01 50.81620 50.81620 50.56451 50.57075
#> 2007-03-02 50.60980 50.72061 50.50808 50.61559
#> 2007-03-03 50.73241 50.73241 50.40929 50.41033
#> 2007-03-04 50.39273 50.40881 50.24922 50.32636
#> 2007-03-05 50.26501 50.34050 50.26501 50.29567
#> ...
```

Now extract all the data from the beginning through January 7, 2007:

```
matrix_xts["/2007-01-07"]
#>                Open     High      Low    Close
#> 2007-01-02 50.03978 50.11778 49.95041 50.11778
#> 2007-01-03 50.23050 50.42188 50.23050 50.39767
#> 2007-01-04 50.42096 50.42096 50.26414 50.33236
#> 2007-01-05 50.37347 50.37347 50.22103 50.33459
#> 2007-01-06 50.24433 50.24433 50.11121 50.18112
#> 2007-01-07 50.13211 50.21561 49.99185 49.99185
```

Additional xts tools providing subsetting are the first and last functions. In the spirit of head and tail from the utils recommended package, they allow for string based subsetting, without forcing the user to conform to the specifics of the time index. Here is the first 1 week of the data:

```
first(matrix_xts,"1 week")
#>                Open     High      Low    Close
#> 2007-01-02 50.03978 50.11778 49.95041 50.11778
#> 2007-01-03 50.23050 50.42188 50.23050 50.39767
#> 2007-01-04 50.42096 50.42096 50.26414 50.33236
#> 2007-01-05 50.37347 50.37347 50.22103 50.33459
#> 2007-01-06 50.24433 50.24433 50.11121 50.18112
#> 2007-01-07 50.13211 50.21561 49.99185 49.99185
```

… and here is the first 3 days of the last week of the data.

```
first(last(matrix_xts,'1 week'),'3 days')
#>                Open     High      Low    Close
#> 2007-06-25 47.20471 47.42772 47.13405 47.42772
#> 2007-06-26 47.44300 47.61611 47.44300 47.61611
#> 2007-06-27 47.62323 47.71673 47.60015 47.62769
```

While the subsetting ability of the above makes exactly which time-based class you choose for your index a bit less relevant, it is none-the-less a factor that is beneficial to have control over.

To that end, xts provides facilities for indexing based on any of the current time-based classes. These include Date, POSIXct, chron, yearmon, yearqtr, and timeDate. The index itself may be accessed via the zoo generics extended to xts — index and the replacement function index<-.

It is also possible to directly query and set the index class of an xts object by using the respective functions indexClass and indexClass<-. Temporary conversion, resulting in a new object with the requested index class, can be accomplished via the convertIndex function.

```
indexClass(matrix_xts)
#> [1] "Date"
indexClass(convertIndex(matrix_xts,'POSIXct'))
#> [1] "POSIXct" "POSIXt"
```

Of course one can also use the traditional indexing for matrices:

```
matrix_xts[1:5]  # same as matrix_xts[1:5, ]
#>                Open     High      Low    Close
#> 2007-01-02 50.03978 50.11778 49.95041 50.11778
#> 2007-01-03 50.23050 50.42188 50.23050 50.39767
#> 2007-01-04 50.42096 50.42096 50.26414 50.33236
#> 2007-01-05 50.37347 50.37347 50.22103 50.33459
#> 2007-01-06 50.24433 50.24433 50.11121 50.18112
matrix_xts[1:5, 4]
#>               Close
#> 2007-01-02 50.11778
#> 2007-01-03 50.39767
#> 2007-01-04 50.33236
#> 2007-01-05 50.33459
#> 2007-01-06 50.18112
matrix_xts[1:5, "Close"]
#>               Close
#> 2007-01-02 50.11778
#> 2007-01-03 50.39767
#> 2007-01-04 50.33236
#> 2007-01-05 50.33459
#> 2007-01-06 50.18112
matrix_xts[1:5]$Close
#>               Close
#> 2007-01-02 50.11778
#> 2007-01-03 50.39767
#> 2007-01-04 50.33236
#> 2007-01-05 50.33459
#> 2007-01-06 50.18112
```

Finally, it is straightforward to combine different xts objects into one with multiple columns and properly aligned by the time index with `merge()` or simply the more standard `cbind()` (which calls `merge()`):

```
open_close <- cbind(matrix_xts$Open, matrix_xts$Close)
str(open_close)
#> An 'xts' object on 2007-01-02/2007-06-30 containing:
#>   Data: num [1:180, 1:2] 50 50.2 50.4 50.4 50.2 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:2] "Open" "Close"
#>   Indexed by objects of class: [Date] TZ: UTC
#>   xts Attributes:
#>  NULL
```

Plotting xts

Another advantage of using the class xts is for plotting. While the basic plot function in R is not very visually appealing, when plotting an xts object with `plot()` it is actually `plot.xts()` that is invoked and it is much prettier:

```
plot(sample_matrix[, 4], type = "l", main = "Stock prices")  # plot for matrices
```


```
plot(matrix_xts$Close, main = "Stock prices")  # plot for xts
```

Additional time-based tools

Calculate periodicity: The periodicity function provides a quick summary as to the underlying periodicity of time series objects:

```
periodicity(matrix_xts)
#> Daily periodicity from 2007-01-02 to 2007-06-30
```

Find endpoints by time: Another common issue with time-series data is identifying the endpoints with respect to time. Often it is necessary to break data into hourly or monthly intervals to calculate some statistic. A simple call to endpoints offers a quick vector of values suitable for subsetting a dataset by. Note that the first element it zero, which is used to delineate the end.

```
endpoints(matrix_xts, on = "months")
#> [1]   0  30  58  89 119 150 180

matrix_xts[endpoints(matrix_xts, on = "months")]
#>                 Open     High      Low    Close
#> 2007-01-31 50.07049 50.22578 50.07049 50.22578
#> 2007-02-28 50.69435 50.77091 50.59881 50.77091
#> 2007-03-31 48.95616 49.09728 48.95616 48.97490
#> 2007-04-30 49.13825 49.33974 49.11500 49.33974
#> 2007-05-31 47.82845 47.84044 47.73780 47.73780
#> 2007-06-30 47.67468 47.94127 47.67468 47.76719

endpoints(matrix_xts, on = "weeks")
#>  [1]   0   6  13  20  27  34  41  48  55  62  69  76  83  90  97 104 111
#> [18] 118 125 132 139 146 153 160 167 174 180

head(matrix_xts[endpoints(matrix_xts, on = "weeks")])
#>                 Open     High      Low    Close
#> 2007-01-07 50.13211 50.21561 49.99185 49.99185
#> 2007-01-14 50.46359 50.62395 50.46359 50.60145
#> 2007-01-21 50.16188 50.42090 50.16044 50.42090
#> 2007-01-28 49.96586 50.00217 49.87468 49.88096
#> 2007-02-04 50.48183 50.55509 50.40203 50.55509
#> 2007-02-11 50.67849 50.91776 50.67849 50.91160
```

Change periodicity: One of the most ubiquitous type of data in finance is OHLC data (Open-High- Low-Close). Often is is necessary to change the periodicity of this data to something coarser, e.g. take daily data and aggregate to weekly or monthly. With `to.period` and related wrapper functions it is a simple proposition.

```
to.period(matrix_xts, "months")
#>            matrix_xts.Open matrix_xts.High matrix_xts.Low matrix_xts.Close
#> 2007-01-31        50.03978        50.77336       49.76308         50.22578
#> 2007-02-28        50.22448        51.32342       50.19101         50.77091
#> 2007-03-31        50.81620        50.81620       48.23648         48.97490
#> 2007-04-30        48.94407        50.33781       48.80962         49.33974
#> 2007-05-31        49.34572        49.69097       47.51796         47.73780
#> 2007-06-30        47.74432        47.94127       47.09144         47.76719

periodicity(to.period(matrix_xts, "months"))
#> Monthly periodicity from 2007-01-31 to 2007-06-30

# changing the index to something more appropriate
to.monthly(matrix_xts)
#>          matrix_xts.Open matrix_xts.High matrix_xts.Low matrix_xts.Close
#> Jan 2007        50.03978        50.77336       49.76308         50.22578
#> Feb 2007        50.22448        51.32342       50.19101         50.77091
#> Mar 2007        50.81620        50.81620       48.23648         48.97490
#> Apr 2007        48.94407        50.33781       48.80962         49.33974
#> May 2007        49.34572        49.69097       47.51796         47.73780
#> Jun 2007        47.74432        47.94127       47.09144         47.76719
```

Periodically apply a function: Often it is desirable to be able to calculate a particular statistic, or evaluate a function, over a set of non-overlapping time periods. With the `period.apply` family of functions it is quite simple. The following examples illustrate a simple application of the max function to our example data:

```
# the general function, internally calls sapply
period.apply(matrix_xts[, "Close"], INDEX = endpoints(matrix_xts), FUN = max)
#>               Close
#> 2007-01-31 50.67835
#> 2007-02-28 51.17899
#> 2007-03-31 50.61559
#> 2007-04-30 50.32556
#> 2007-05-31 49.58677
#> 2007-06-30 47.76719

# same result as above, just a monthly interface
apply.monthly(matrix_xts[, "Close"], FUN = max)
#>               Close
#> 2007-01-31 50.67835
#> 2007-02-28 51.17899
#> 2007-03-31 50.61559
#> 2007-04-30 50.32556
#> 2007-05-31 49.58677
#> 2007-06-30 47.76719

# using one of the optimized functions - about 4x faster
period.max(matrix_xts[,4], endpoints(matrix_xts))
#>                [,1]
#> 2007-01-31 50.67835
#> 2007-02-28 51.17899
#> 2007-03-31 50.61559
#> 2007-04-30 50.32556
#> 2007-05-31 49.58677
#> 2007-06-30 47.76719
```

In addition to `apply.monthly`, there are wrappers to other common time frames including: `apply.daily`, `apply.weekly`, `apply.quarterly`, and `ap- ply.yearly`. Current optimized functions include `period.max`, `period.min`, `period.sum`, and `period.prod`.

Package quantmod

The package quantmod (https://www.rdocumentation.org/packages/quantmod)is designed to assist the quantitative trader in the development, testing, and deployment of statistically based trading models.

Getting data: The most useful function in quantmod is `getSymbol()`, which allows to conveniently load data from several websites like YahooFinance, GoogleFinance, FRED, etc.:

```
library(quantmod)

getSymbols(c("AAPL", "GOOG"), from = "2013-01-01", to = "2015-12-31")
#> [1] "AAPL" "GOOG"

str(AAPL)
#> An 'xts' object on 2013-01-02/2015-12-30 containing:
#>   Data: num [1:755, 1:6] 79.1 78.3 76.7 74.6 75.6 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:6] "AAPL.Open" "AAPL.High" "AAPL.Low" "AAPL.Close" ...
#>   Indexed by objects of class: [Date] TZ: UTC
#>   xts Attributes:
#> List of 2
#>  $ src    : chr "yahoo"
#>  $ updated: POSIXct[1:1], format: "2018-09-05 15:21:33"

head(AAPL)
#>            AAPL.Open AAPL.High AAPL.Low AAPL.Close AAPL.Volume
#> 2013-01-02  79.11714  79.28571 77.37572   78.43285   140129500
#> 2013-01-03  78.26857  78.52428 77.28571   77.44286    88241300
#> 2013-01-04  76.71000  76.94714 75.11857   75.28571   148583400
#> 2013-01-07  74.57143  75.61429 73.60000   74.84286   121039100
#> 2013-01-08  75.60143  75.98428 74.46429   75.04429   114676800
#> 2013-01-09  74.64286  75.00143 73.71286   73.87143   101901100
#>            AAPL.Adjusted
#> 2013-01-02      56.11887
#> 2013-01-03      55.41053
#> 2013-01-04      53.86707
#> 2013-01-07      53.55021
#> 2013-01-08      53.69434
#> 2013-01-09      52.85514
```

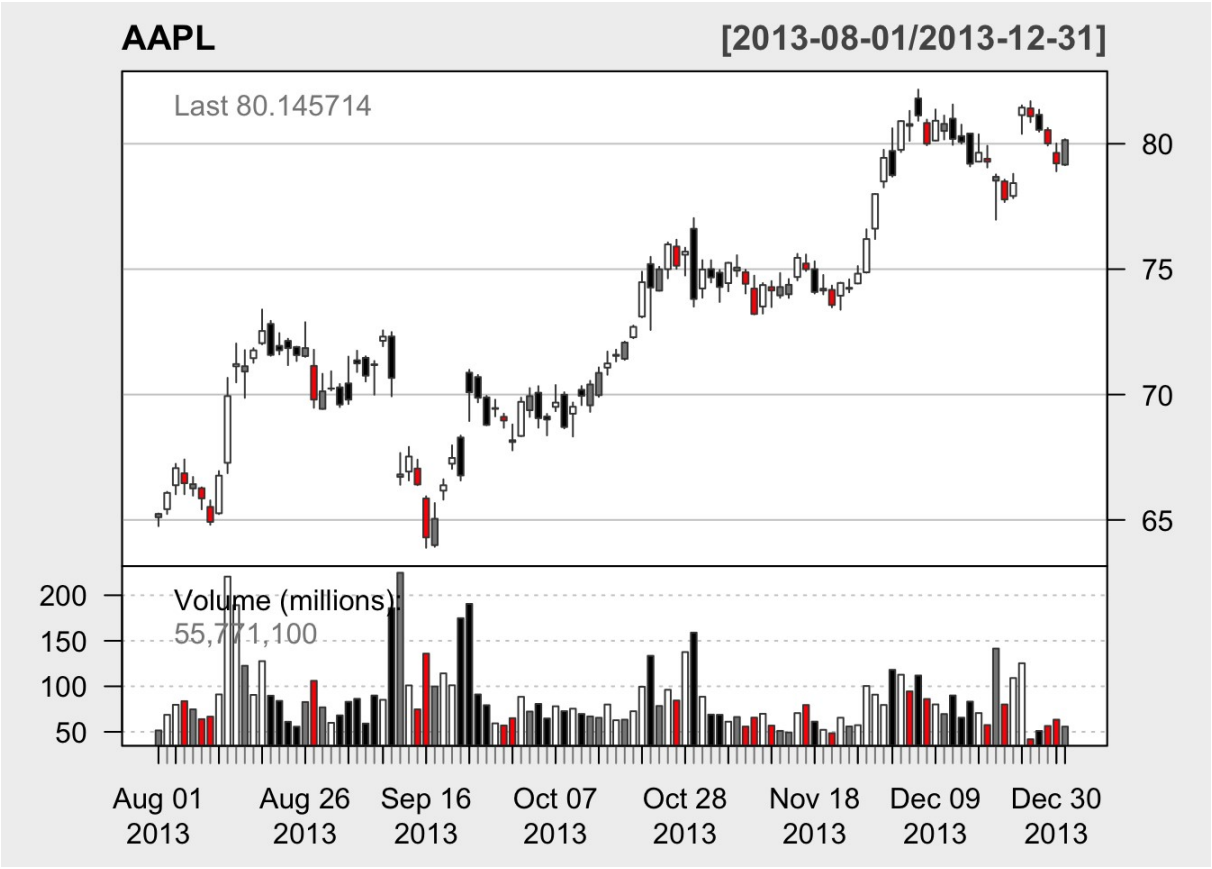
The OHLCV basics: Data commonly has the prices open, high, low, close, adjusted close, as well as volume. There are many handy functions to extract those data, e.g., `Op()`, `Hi()`, `Lo()`, `Cl()`, `Ad()`, `Vo()`, as well as to query a variety of questions such as `is.OHLC()`, `has.Vo()`, etc.

```
getSymbols("GS")  # Goldman OHLC from yahoo
#> [1] "GS"

is.OHLC(GS)  # does the data contain at least OHL and C?
#> [1] TRUE

has.Vo(GS)  # how about volume?
#> [1] TRUE

head(Op(GS))  # just the Open column please.
#>            GS.Open
#> 2007-01-03  200.60
#> 2007-01-04  200.22
#> 2007-01-05  198.43
#> 2007-01-08  199.05
#> 2007-01-09  203.54
#> 2007-01-10  203.40
```

Charting with quantmod: The function `chartSeries()` is a nice tool to visualize financial time series in a way that many practicioners are familiar with–line charts, as well as OHLC bar and candle charts. There are convenience wrappers to these different styles (`lineChart()`, `barChart()`, and `candleChart()`), though `chartSeries()` does quite a bit to automatically handle data in the most appropriate way.

```
chartSeries(AAPL["2013-8/2013-12"], name = "AAPL")
```


```
# Add multi-coloring and change background to white
candleChart(AAPL["2013-8/2013-12"], multi.col = TRUE, theme = "white", name = "AAPL")
```

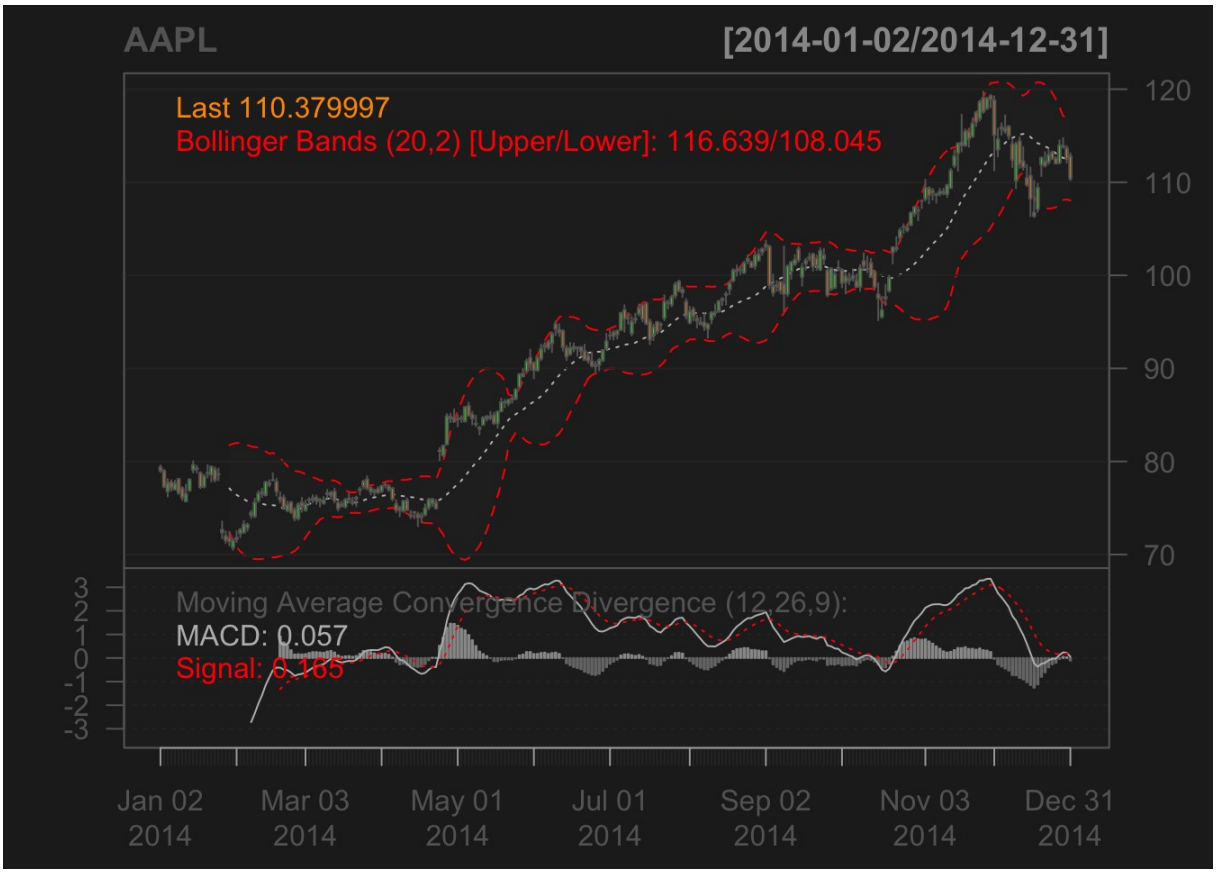



```
# Now weekly with custom color candles using the quantmod function to.weekly
chartSeries(to.weekly(AAPL), up.col = "white", dn.col = "blue", name = "AAPL")
```

Technical analysis charting tools: One can add technical analysis studies from package TTR to the above

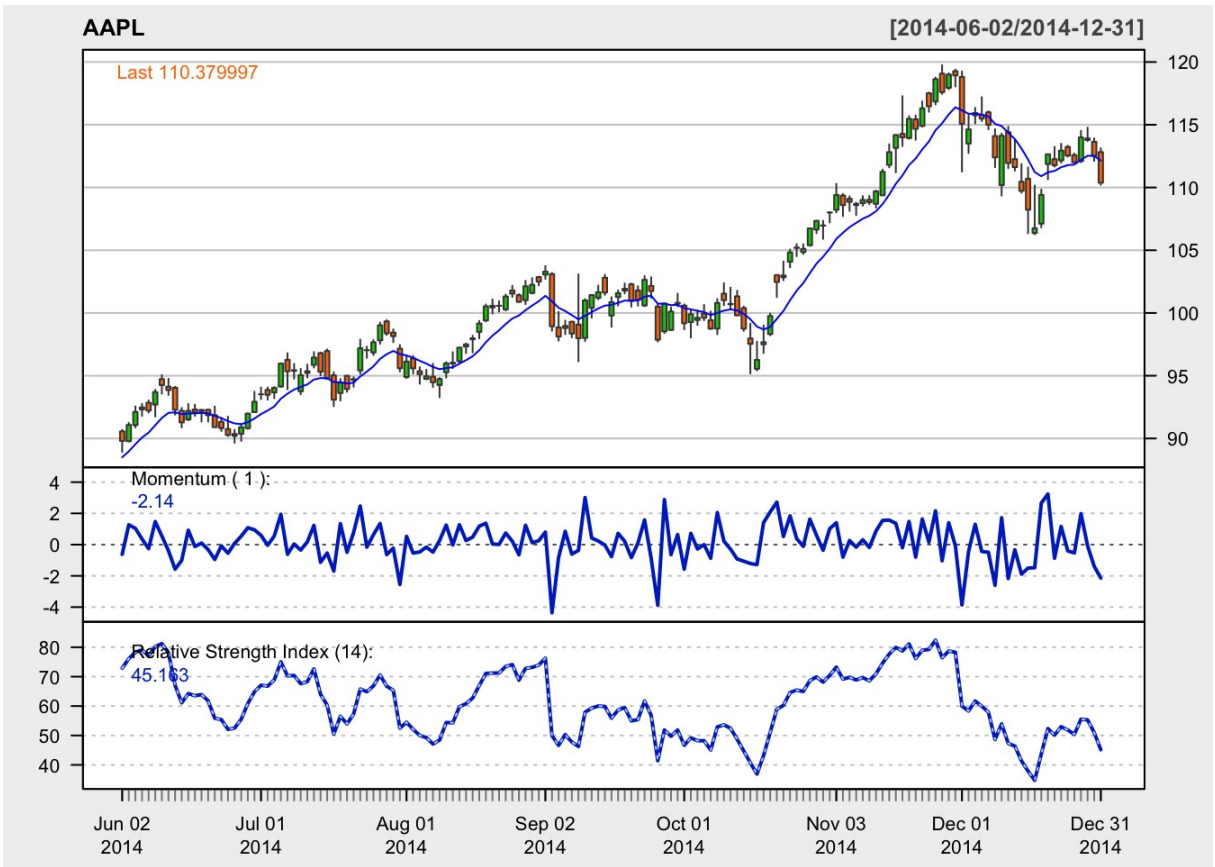
charts:

```
chartSeries(AAPL["2014"], name = "AAPL",
            TA = "addMACD(); addBBands()")
```




```
chartSeries(AAPL["2014"], name = "AAPL",
            TA = "addMomentum(); addEMA(); addRSI()")
```


```
reChart(subset = "2014-6/2014-12", theme = "white", type = "candles")
```



Package TTR

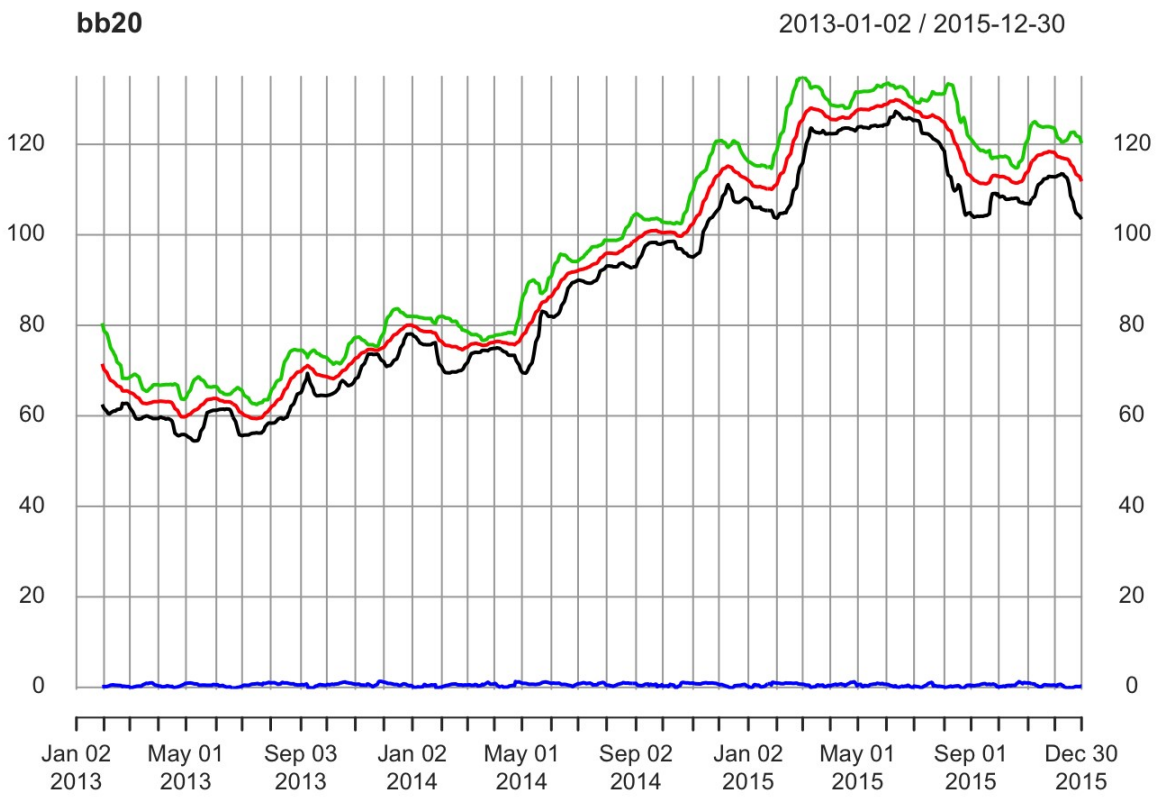
The package TTR (Technical Trading Rules) (https://www.rdocumentation.org/packages/TTR) is designed for traditional technical analysis and charting.

Moving averages: One can easily compute moving averages (https://www.rdocumentation.org/packages/TTR/versions/0.23-2/topics/SMA).

```
library(TTR)

# simple moving average
sma10 <- SMA(Cl(AAPL), n = 10)
head(sma10, 20)
#>                  SMA
#> 2013-01-02       NA
#> 2013-01-03       NA
#> 2013-01-04       NA
#> 2013-01-07       NA
#> 2013-01-08       NA
#> 2013-01-09       NA
#> 2013-01-10       NA
#> 2013-01-11       NA
#> 2013-01-14       NA
#> 2013-01-15 74.51314
#> 2013-01-16 73.89971
#> 2013-01-17 73.33657
#> 2013-01-18 72.95086
#> 2013-01-22 72.67757
#> 2013-01-23 72.51614
#> 2013-01-24 71.56472
#> 2013-01-25 70.37000
#> 2013-01-28 69.36329
#> 2013-01-29 68.74214
#> 2013-01-30 68.32657

# exponentially moving average
ema10 <- EMA(Cl(AAPL), n = 10)
```

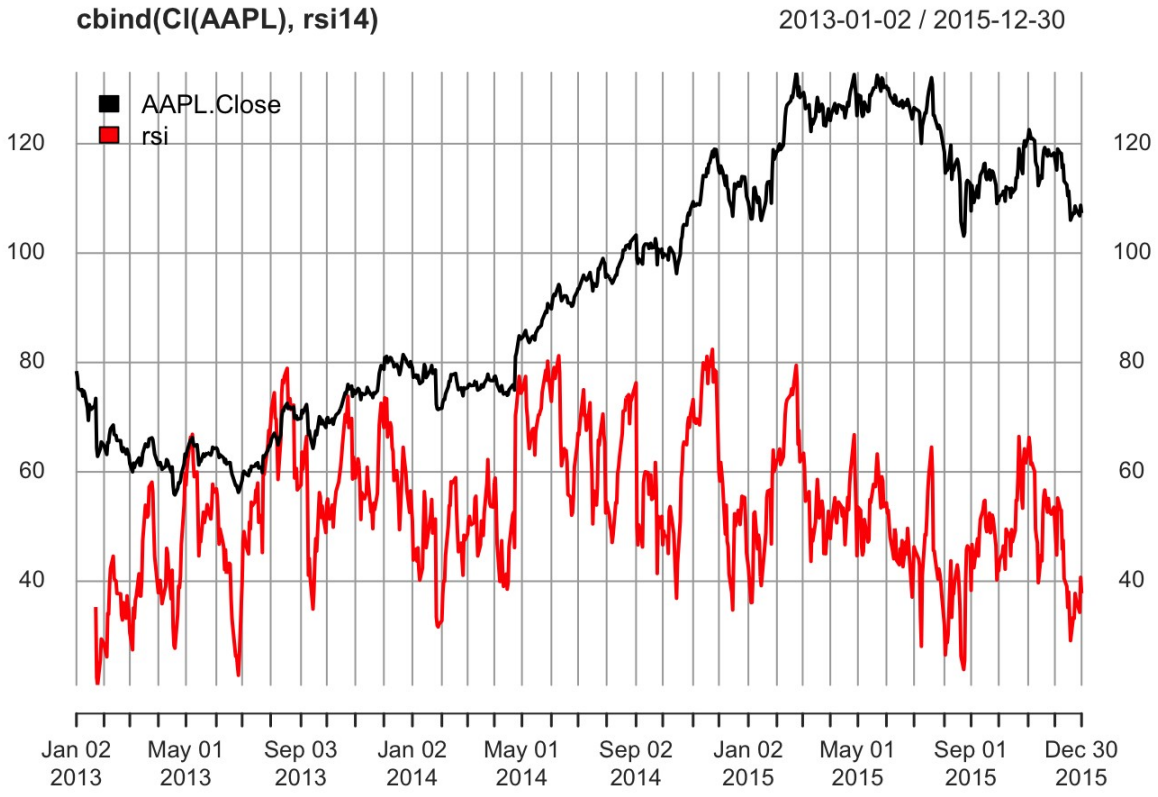
Bollinger Bands:

```
bb20 <- BBands(HLC(AAPL), sd = 2.0)
str(bb20)
#> An 'xts' object on 2013-01-02/2015-12-30 containing:
#>   Data: num [1:755, 1:4] NA NA NA NA NA NA NA NA NA NA ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:4] "dn" "mavg" "up" "pctB"
#>   Indexed by objects of class: [Date] TZ: UTC
#>   xts Attributes:
#> List of 2
#>  $ src    : chr "yahoo"
#>  $ updated: POSIXct[1:1], format: "2018-09-05 15:21:33"
plot(bb20)
```



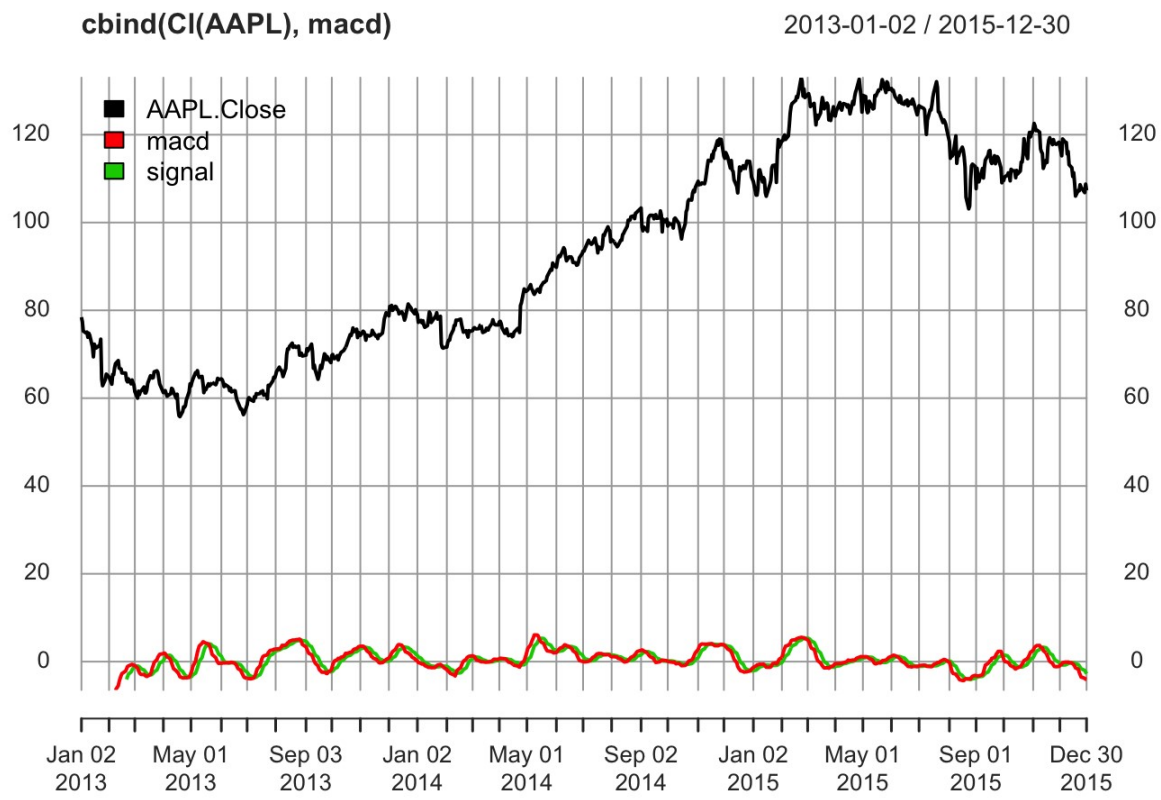
RSI – Relative Strength Indicator:

```
rsi14 <- RSI(Cl(AAPL), n = 14)
plot(cbind(Cl(AAPL), rsi14), legend.loc = "topleft")
```



MACD:

```
macd = MACD(Cl(AAPL), nFast = 12, nSlow = 26, nSig = 9, maType = SMA)
str(macd)
#> An 'xts' object on 2013-01-02/2015-12-30 containing:
#>   Data: num [1:755, 1:2] NA NA NA NA NA NA NA NA NA NA ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:2] "macd" "signal"
#>   Indexed by objects of class: [Date] TZ: UTC
#>   xts Attributes:
#> List of 2
#>  $ src    : chr "yahoo"
#>  $ updated: POSIXct[1:1], format: "2018-09-05 15:21:33"
plot(cbind(Cl(AAPL), macd), legend.loc = "topleft")
```

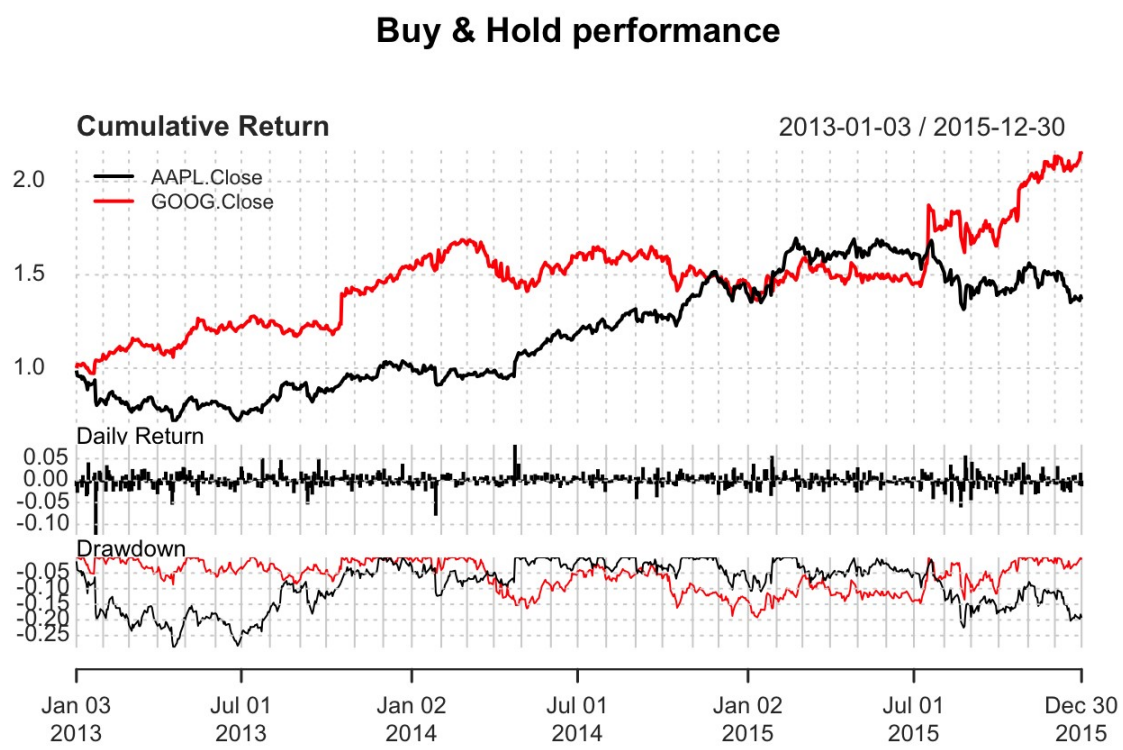


Package PerformanceAnalytics

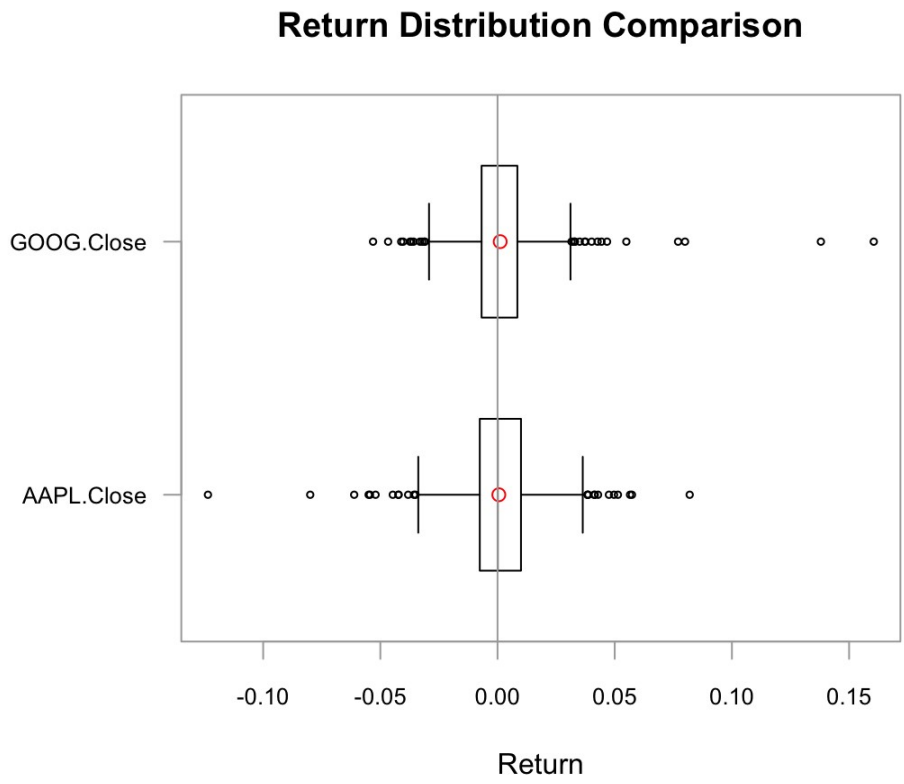
The package PerformanceAnalytics (https://www.rdocumentation.org/packages/PerformanceAnalytics) contains a large list of convenient functions for plotting and evaluation of performance.

```
library(PerformanceAnalytics)
#>
#> Attaching package: 'PerformanceAnalytics'
#> The following object is masked from 'package:graphics':
#>
#>     legend

# compute returns
ret <- CalculateReturns(cbind(Cl(AAPL), Cl(GOOG)))  # same as Cl(AAPL)/lag(Cl(AAPL)) -
 1)
head(ret)
#>              AAPL.Close    GOOG.Close
#> 2013-01-02           NA            NA
#> 2013-01-03 -0.012622236  0.0005807288
#> 2013-01-04 -0.027854642  0.0197603692
#> 2013-01-07 -0.005882338 -0.0043633560
#> 2013-01-08  0.002691399 -0.0019734357
#> 2013-01-09 -0.015628904  0.0065730603

# performance measures
table.AnnualizedReturns(ret)
#>                           AAPL.Close GOOG.Close
#> Annualized Return             0.1105     0.2907
#> Annualized Std Dev            0.2575     0.2449
#> Annualized Sharpe (Rf=0%)     0.4291     1.1872
table.CalendarReturns(ret)
#>       Jan  Feb  Mar  Apr  May Jun  Jul  Aug  Sep  Oct  Nov  Dec AAPL.Close
#> 2013 -0.3 -0.7 -2.1  2.9 -0.4 0.7 -0.2 -0.9 -1.2 -0.4  1.9  1.2        0.3
#> 2014  0.2 -0.3  0.0 -0.4 -0.4 1.0 -2.6  0.2  0.6  1.0 -0.1 -1.9       -2.6
#> 2015 -1.5 -1.5 -1.5 -2.7 -1.1 0.7 -0.9 -0.5  1.1 -0.9  0.4 -1.3       -9.2
#>      GOOG.Close
#> 2013       -0.3
#> 2014        1.5
#> 2015       -2.8
table.Stats(ret)
#>                 AAPL.Close GOOG.Close
#> Observations      754.0000   754.0000
#> NAs                 1.0000     1.0000
#> Minimum            -0.1236    -0.0531
#> Quartile 1         -0.0076    -0.0068
#> Median              0.0001     0.0000
#> Arithmetic Mean     0.0005     0.0011
#> Geometric Mean      0.0004     0.0010
#> Quartile 3          0.0100     0.0084
#> Maximum             0.0820     0.1605
#> SE Mean             0.0006     0.0006
#> LCL Mean (0.95)    -0.0006     0.0000
#> UCL Mean (0.95)     0.0017     0.0022
#> Variance            0.0003     0.0002
#> Stdev               0.0162     0.0154
#> Skewness           -0.5988     2.6873
#> Kurtosis            6.5048    24.2017
table.DownsideRisk(ret)
```

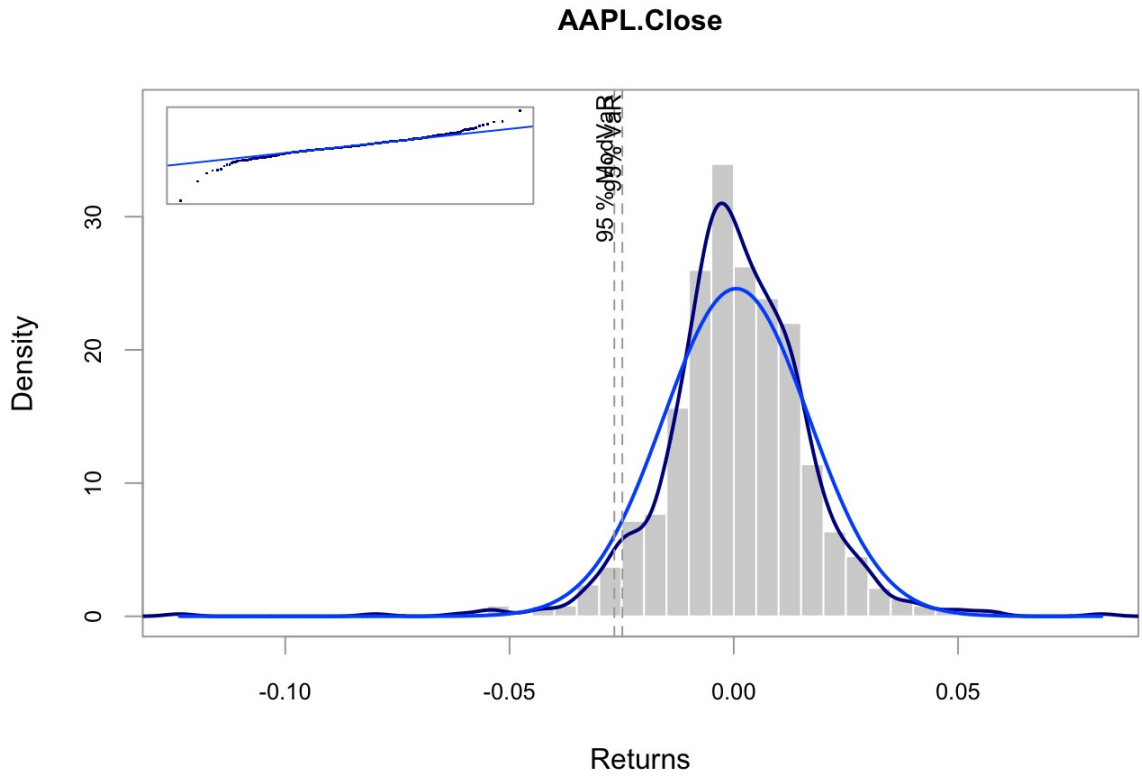
```
#>                                AAPL.Close GOOG.Close
#> Semi Deviation                     0.0118     0.0092
#> Gain Deviation                     0.0105     0.0142
#> Loss Deviation                     0.0120     0.0082
#> Downside Deviation (MAR=210%)      0.0163     0.0138
#> Downside Deviation (Rf=0%)         0.0116     0.0086
#> Downside Deviation (0%)            0.0116     0.0086
#> Maximum Drawdown                   0.2887     0.1918
#> Historical VaR (95%)              -0.0249    -0.0196
#> Historical ES (95%)               -0.0369    -0.0273
#> Modified VaR (95%)                -0.0266    -0.0028
#> Modified ES (95%)                 -0.0538    -0.0241

# plots
charts.PerformanceSummary(ret, wealth.index = TRUE, main = "Buy & Hold performance")
```



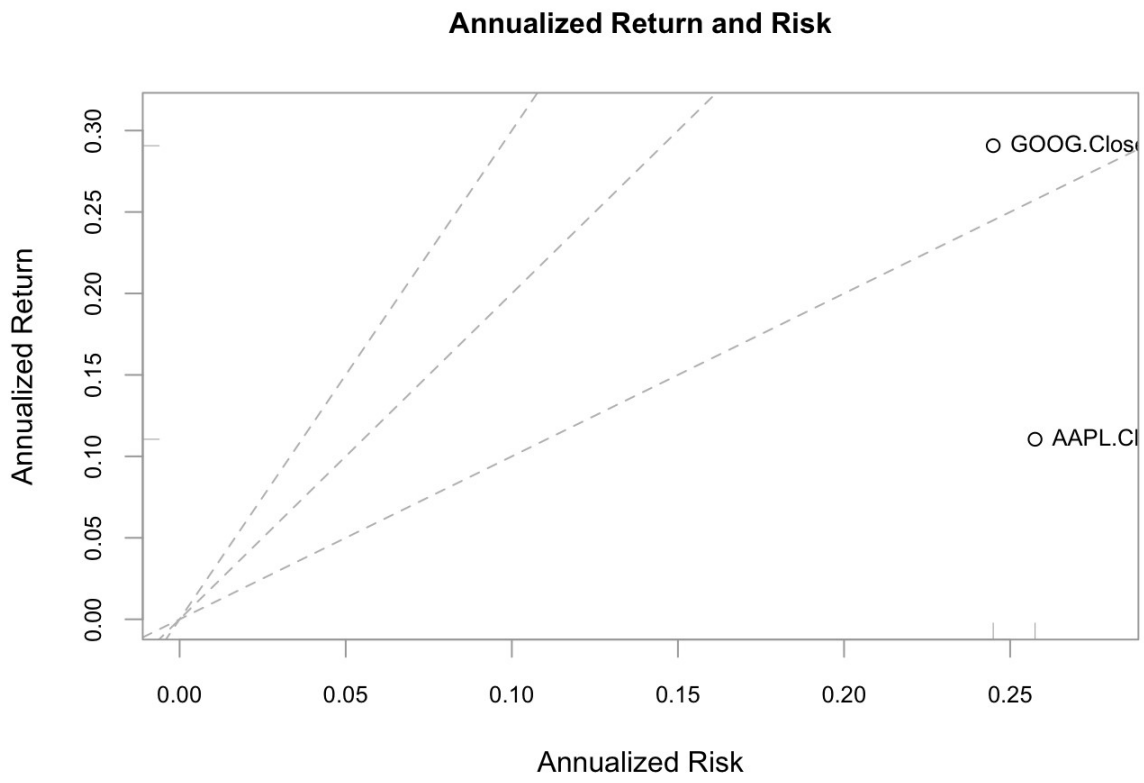

```
chart.Boxplot(ret)
```




```
chart.Histogram(ret, note.cex = 0.5,
                methods = c("add.density", "add.normal", "add.risk", "add.qqplot"))
```



```
chart.RiskReturnScatter(ret)
```



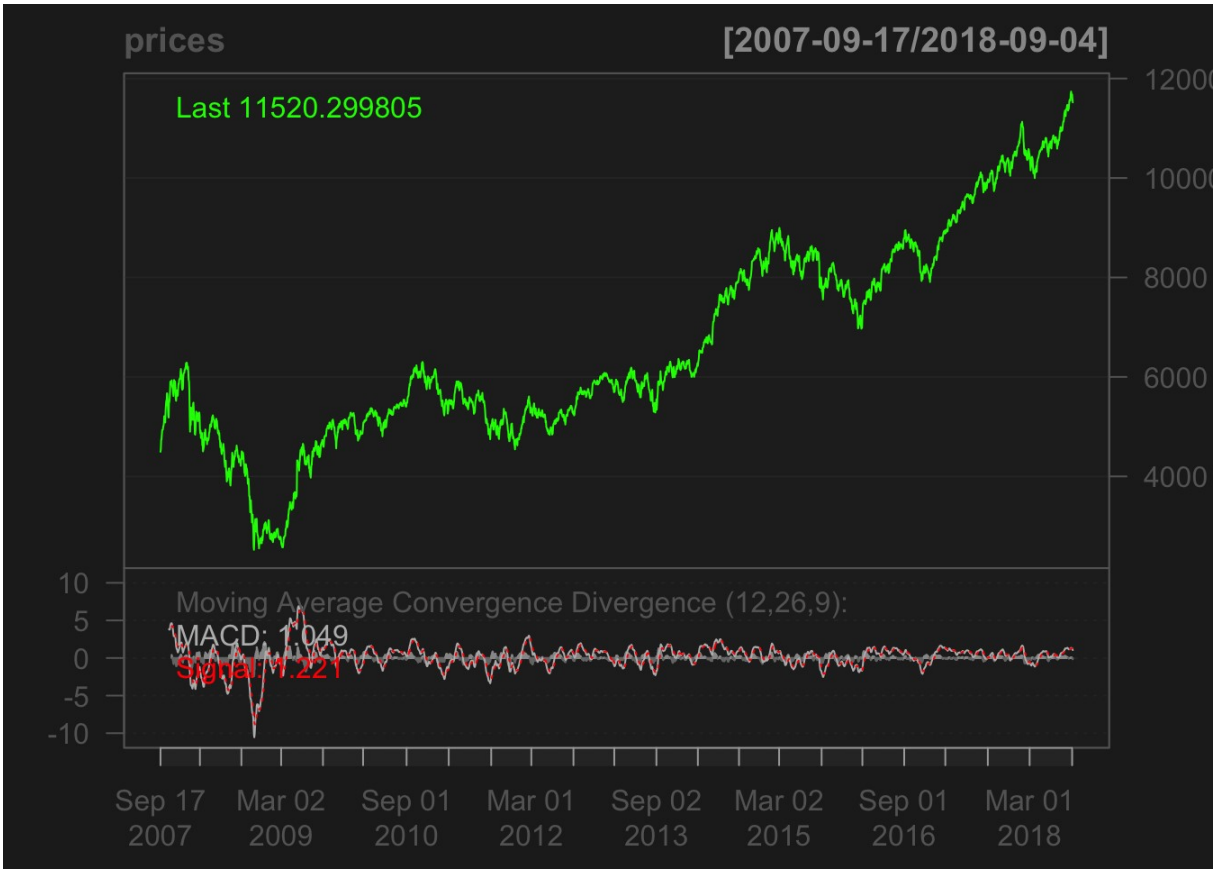
Example of a technical trading strategy combining the packages xts, quantmod, TTR, and PerformanceAnalytics

Let's now put into practice the different packages we have learned with a very simple example of a trading strategy.

```
# download NIFTY 50 (Indian index)
getSymbols("^NSEI")
#> Warning: ^NSEI contains missing values. Some functions will not work if
#> objects contain missing values in the middle of the series. Consider using
#> na.omit(), na.approx(), na.fill(), etc to remove or replace them.
#> [1] "NSEI"
chartSeries(NSEI, TA = NULL)
```

As a trading stragey, we choose MACD (Moving Average Convergence Divergence) for this example. In a moving average crossovers strategy two averages are computed, a slow moving average and a fast moving average. The difference between the fast moving average and slow moving average is called MACD line. A third average called signal line; a 9 day exponential moving average of MACD signal, is also computed. If the MACD line crosses above the signal line then it is a bullish sign and we go long. If the MACD line crosses below the signal line then it is a bearish sign and we go short. We choose closing price of NSE data to calculate the averages.

```
prices <- na.omit(Cl(NSEI))
macd <- MACD(prices, nFast = 12, nSlow = 26, nSig = 9, maType = SMA, percent = FALSE)
chartSeries(prices, TA = "addMACD()")
```



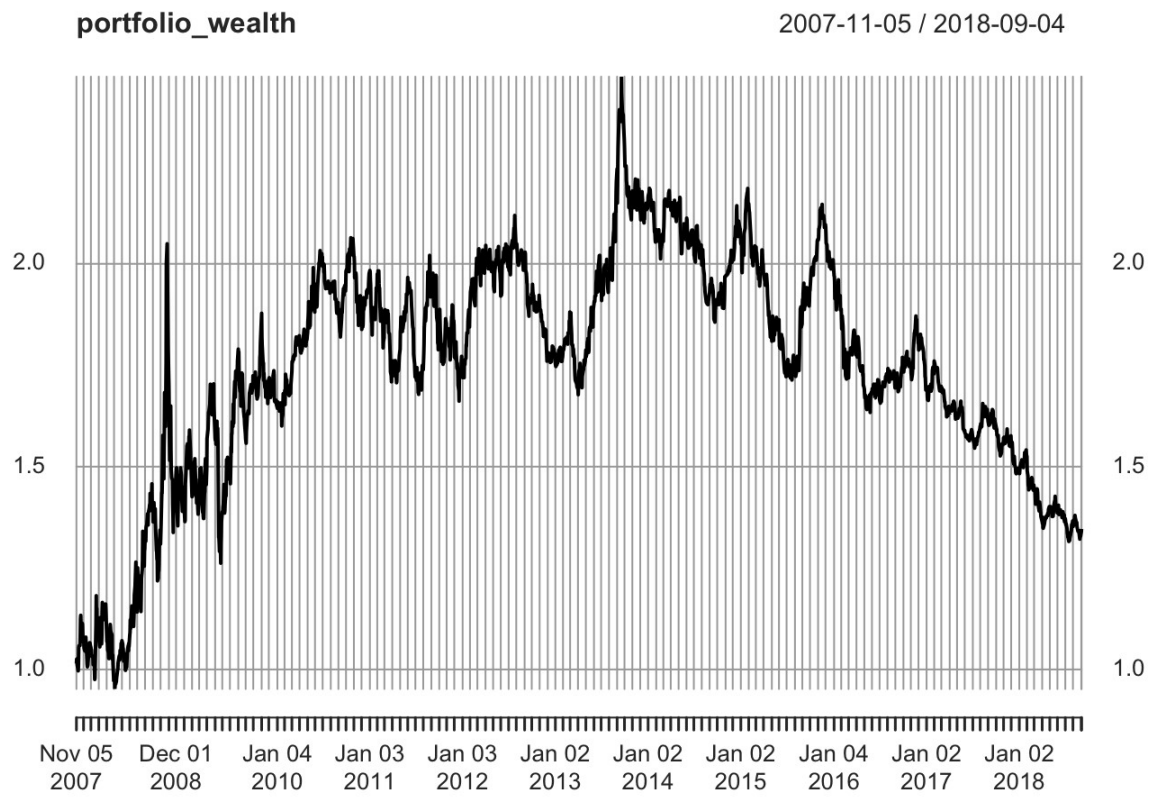
We define our trading signal as follows:

- If the MACD signal crossed above the signal line we go long on NSE
- If the MACD signal crossed below the signal line we go short on NSE

```
signal <- lag(ifelse(macd$macd < macd$signal, -1, 1))
```

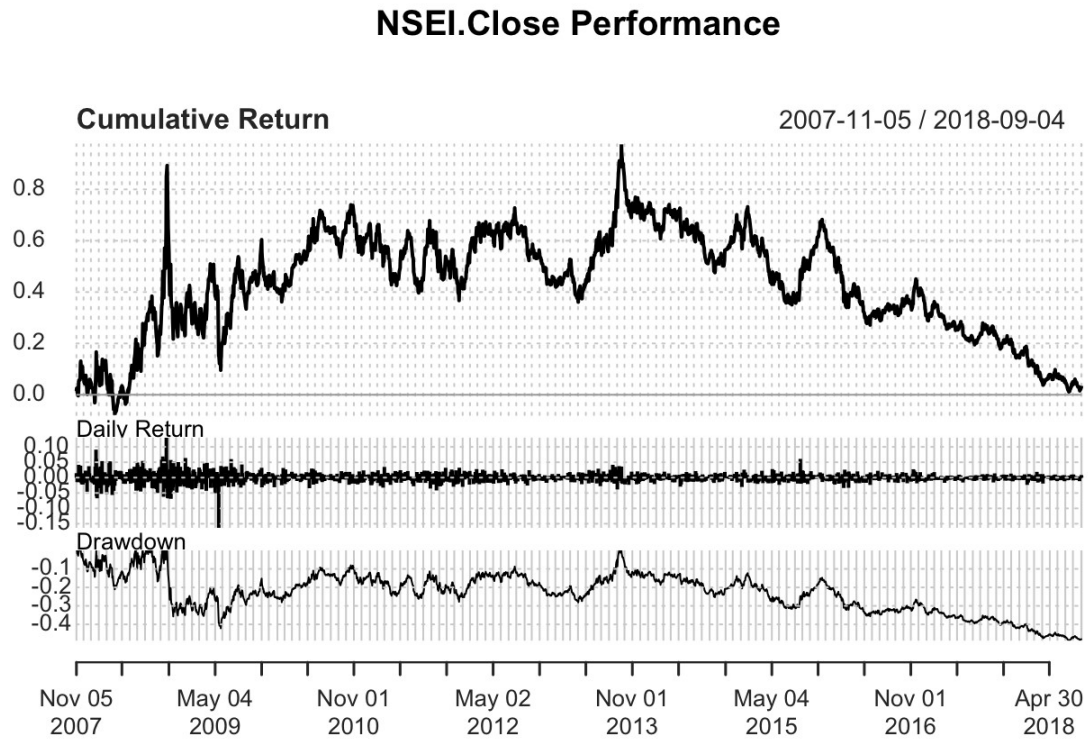
The trading signal is then applied to the closing price to obtain the returns of our strategy:

```
returns <- ROC(prices)*signal
returns <- na.omit(returns)
portfolio_wealth <- exp(cumsum(returns))
plot(portfolio_wealth)
```



Finally, we can evaluate the performance:

```
table.Drawdowns(returns, top=10)
#>          From     Trough         To   Depth Length To Trough Recovery
#> 1  2013-09-20 2018-07-16       <NA> -0.4880   1214      1179       NA
#> 2  2008-10-29 2009-05-26 2013-09-10 -0.4211   1189       133     1056
#> 3  2008-01-23 2008-04-04 2008-06-24 -0.2105    103        50       53
#> 4  2008-08-29 2008-09-19 2008-10-08 -0.1679     27        15       12
#> 5  2007-11-23 2008-01-16 2008-01-22 -0.1431     41        37        4
#> 6  2008-06-27 2008-07-16 2008-07-23 -0.1021     19        14        5
#> 7  2008-10-13 2008-10-14 2008-10-17 -0.0698      5         2        3
#> 8  2008-07-24 2008-07-29 2008-08-05 -0.0651      9         4        5
#> 9  2008-10-20 2008-10-21 2008-10-22 -0.0504      3         2        1
#> 10 2007-11-08 2007-11-12 2007-11-14 -0.0288      4         2        2
table.DownsideRisk(returns)
#>                                NSEI.Close
#> Semi Deviation                     0.0099
#> Gain Deviation                     0.0106
#> Loss Deviation                     0.0104
#> Downside Deviation (MAR=210%)      0.0147
#> Downside Deviation (Rf=0%)         0.0098
#> Downside Deviation (0%)            0.0098
#> Maximum Drawdown                   0.4880
#> Historical VaR (95%)              -0.0202
#> Historical ES (95%)               -0.0321
#> Modified VaR (95%)                -0.0199
#> Modified ES (95%)                 -0.0224
charts.PerformanceSummary(returns)
```



R Scripts and R Markdown

R scripts

One simple way to use R is by typing the commands in the command window one by one. However, this quickly becomes inconvenient and it is necessary to write scripts. In RStudio one can simply create a new R script or open a .R file. Then there the commands are written in the same order as they will be later executed (this point cannot be overemphasized).

With the R script open, one can execute line by line (either clicking a button or with a keyboard shortcut) or source the whole R file (also either clicking a button or with a keyboard shortcut). Alternatively, one can also source the R file from the command line with `source("filename.R")`, but first one has to make sure to be in the correct folder (to see and set the current directory use the commands: `getwd()` and `setwd("folder_name")`). Sourcing using the command `source("filename.R")` is very convenient when one has a library of useful functions or data that is needed prior to the execution of another main R script.

R Markdown

Another important type of scripts is the R Markdown (http://rmarkdown.rstudio.com/index.html) format (with file extension .Rmd). It is an extremely versatile format that allows the combination of formattable text, mathematics based on Latex codes, R code (or any other language), and then automatic inclusion of the results from the execution of the code (plots or just other type of output). This type of format also exists for Python and they are generally referred to as Notebooks and have recently become key in the context of reproducible research (because anybody can execute the source .Rmd file and reproduce all the plots and output). This document that you are now reading is an example of an R Markdown script.

R Markdown files can be directly created or opened from within RStudio. To compile the source .Rmd file, just click the button called Knit and an html will be automatically generated after executing all the chunks of code (other formats can also be generated like pdf).

The following is a simple header/body template that can be used to prepare the projects in this course:

```
---
title: "Hello R Markdown"
author: "Awesome Me"
date: "2018-09-05"
output: html_document
---
```

````
Summary of this document here.

# First header

## First subheader

## Second subheader

# Second header

- bullet list 1
- bullet list 2
    + more 2a
    + more 2b

This is a link: [R Markdown tutorial](http://rmarkdown.rstudio.com)

```{r}
# here some R code
plot(cumsum(rnorm(100)), type = "l")
```
````

For more information on the R Markdown formatting:

- R Markdown tutorial (http://rmarkdown.rstudio.com)
- R Markdown Cheat Sheet (https://www.rstudio.org/links/r_markdown_cheat_sheet)
- R Markdown Reference Guide (https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf)

To explore further

There are several CRAN Task Views relevant to financial applications, each of them encompases many packages:

- Finance (https://cran.r-project.org/web/views/Finance.html)
- Time Series (https://cran.r-project.org/web/views/TimeSeries.html)
- Econometrics (https://cran.r-project.org/web/views/Econometrics.html)
- Optimization (https://cran.r-project.org/web/views/Optimization.html)
- Machine Learning (https://cran.r-project.org/web/views/MachineLearning.html)