MQIM R BOOTCAMP

Introduction

Casey T Li

Monday, Dec 7, 2020

Section 1

Bootcamp Week Schedule

Table of Content

Section 2

Introduction to R

Introduction

- In this class, we will use R as the primary computational environment.
- R is a statistical programming language and computing environment that easily handles statistical analysis, numerical computing as well as mathematical modeling.
- R is based on the S language, which was developed at Bell Labs, and is maintained by the R Project (R Dev Core Team) and the R Foundation.
- R is free and open source, it is extremely popular in the financial industry (as well as among analytics firms and statistical researchers).

Why Use R?

As with any programming language, learning R requires a significant initial time commitment (as well as a commitment to continue using it in the future to maintain skills). An important question is "why bother?"

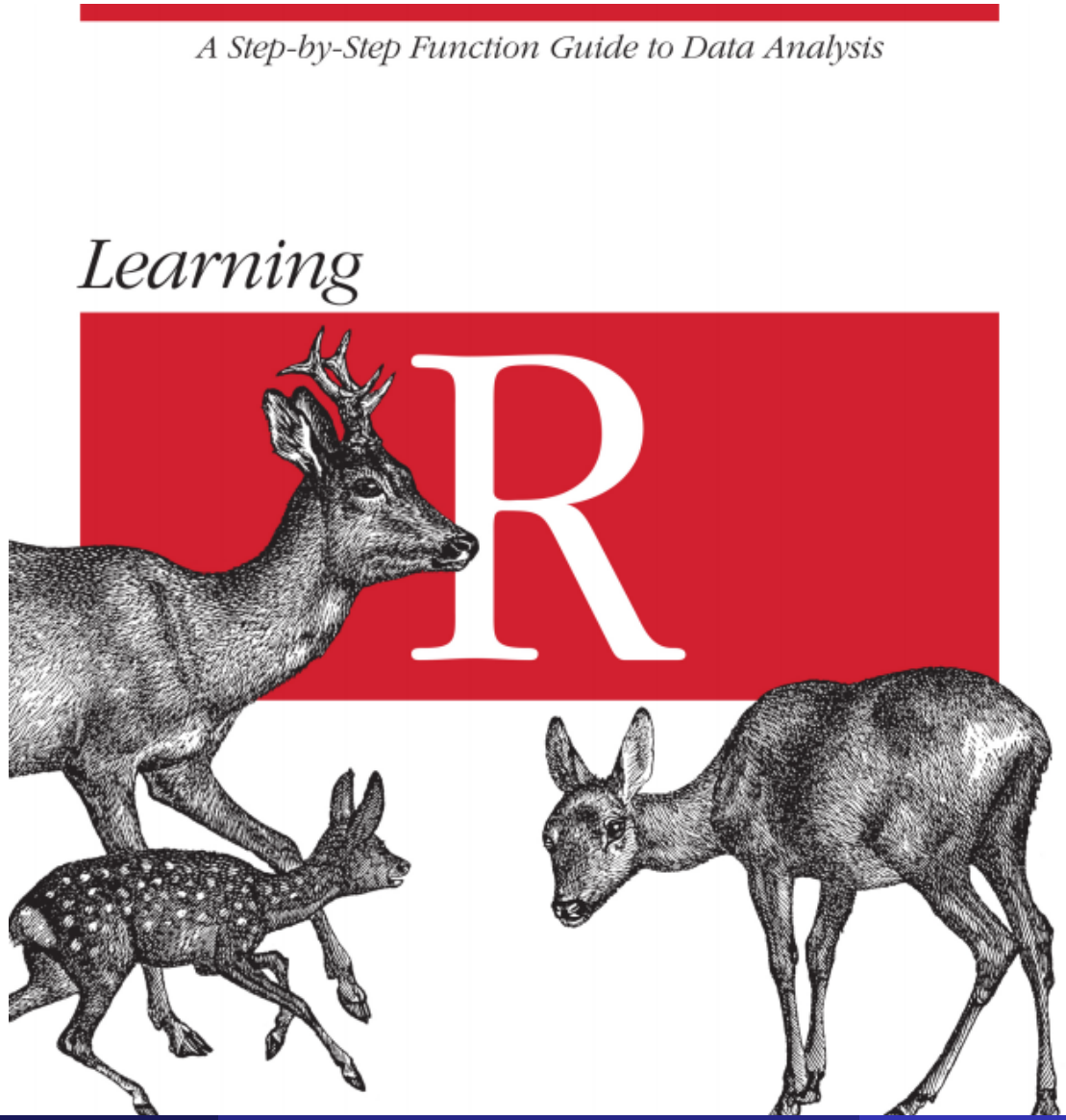
- R is highly flexible, and can be used for statistical analysis, mathematical modeling, and a variety of other tasks (these slides were written in Rstudio using the Rmarkdown format, for example)
- For one off analyses, R may be "overkill", but in the context of institutional investment management, where we might need to simulate millions of random numbers, or run thousands of regressions for example, R is much more robust and flexible than MS Excel
- Historically, the statistical routines and solvers available in commercial spreadsheet products were somewhat less than reliable
- With larger tasks, spreadsheets are prone to error and difficult to audit, while well written software is easy to debug and maintain.
- It's widely used in industry, and is a useful skill to have on the job market

Why Avoid R?

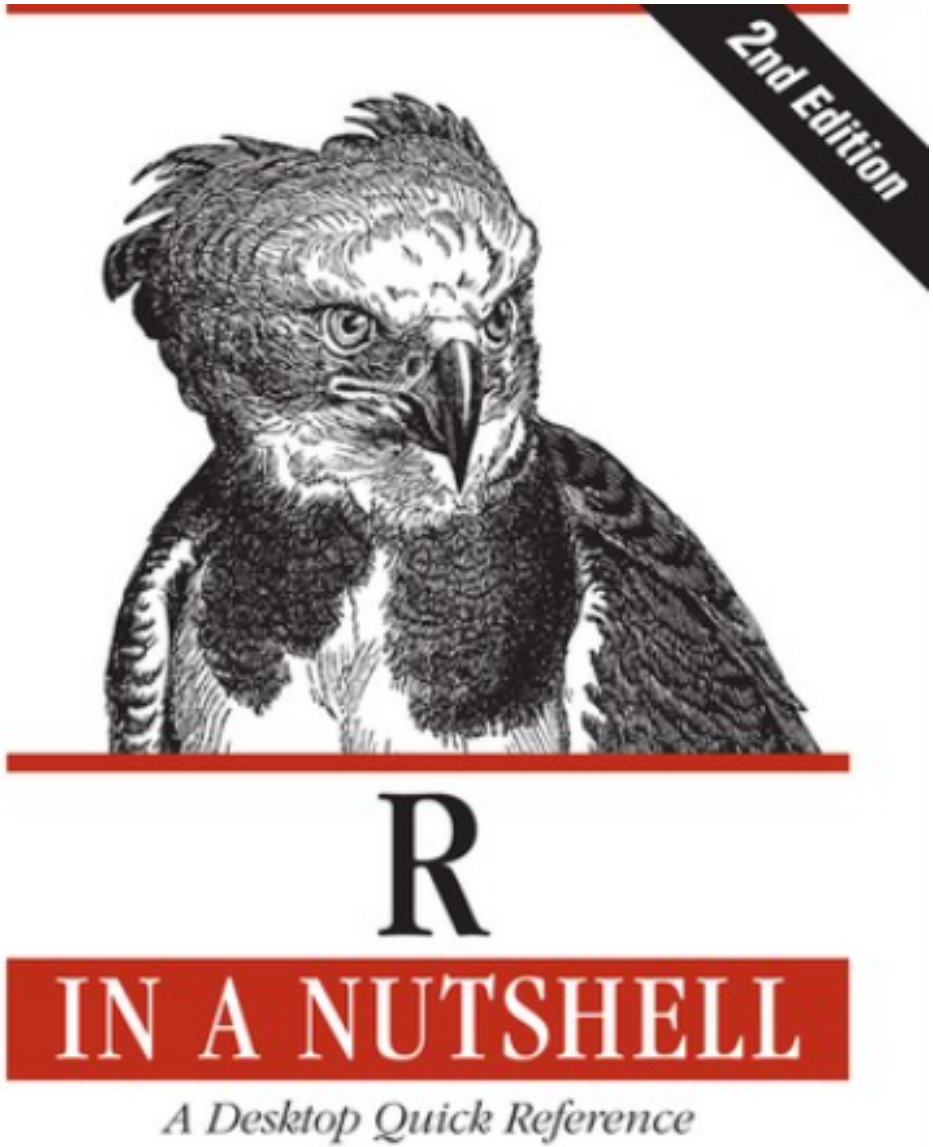
- R is slow. There are ways to write efficient R code, but if extreme low-latency is required, R is probably not the correct tool.
- R (and the packages) is documented unevenly - some functions and packages are extremely well documented, while others are not.
- R is open source and constantly evolving, and so the requirement is on the user to ensure that "everything works together" (note: I suggest that once you install an R version for this course, you stick with that version until the semester is finished)
- Support is generally not available on demand, only via the generosity of other users who often volunteer help on various sites and mailing lists - be polite and follow the guidelines, and you can usually get help from extremely knowledgeable domain experts in your area.

“Learning R” by Cotton

- http://duhi23.github.io/Analisis-de-datos/Cotton.pdf

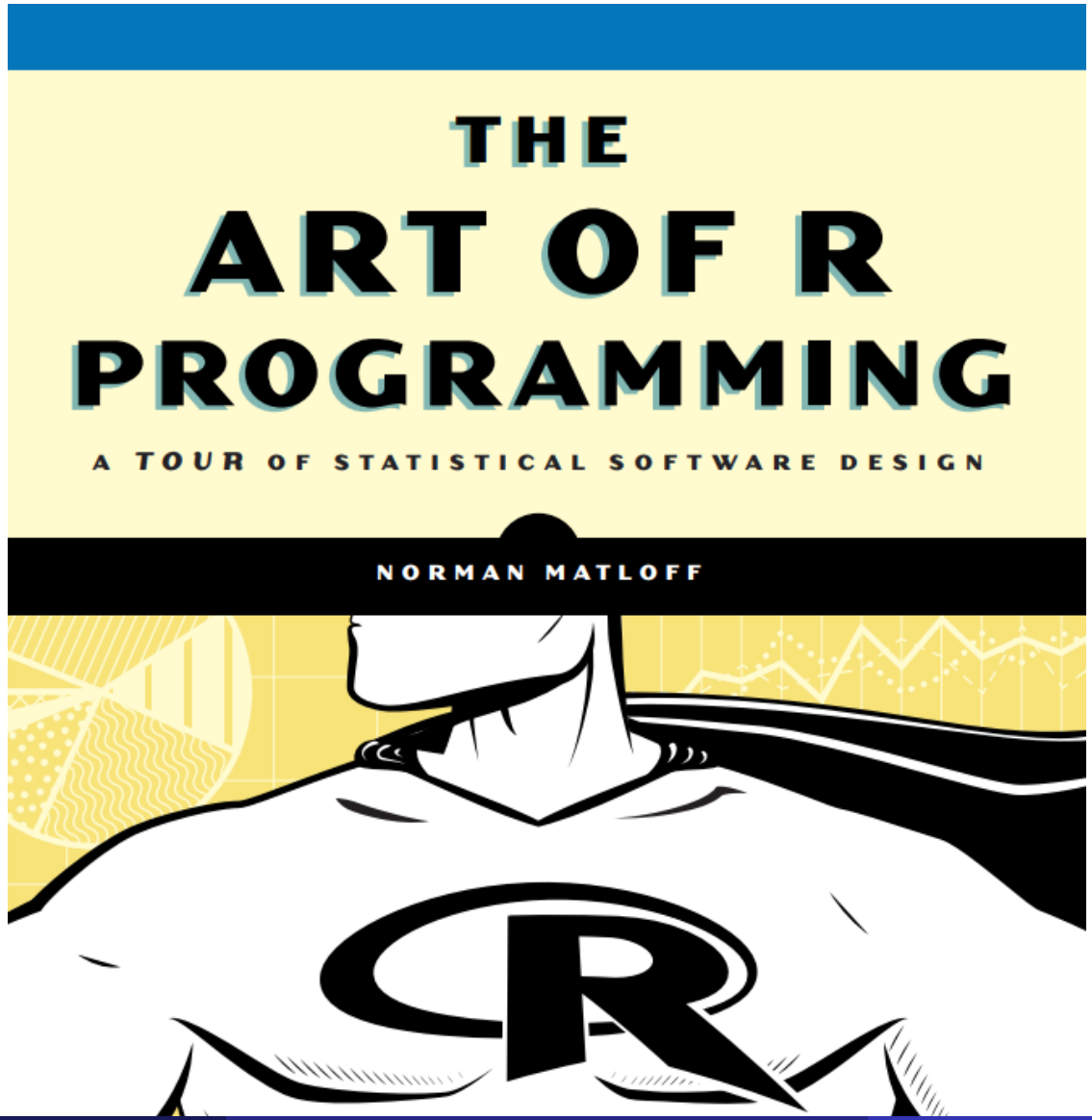


“R in a Nutshell” by Adler



"The Art of R Programming" by Matloff

- http://diytranscriptomics.com/Reading/files/TheArtofRProgramming.pdf



Free Online Resources

- http://www.r-bloggers.com/how-to-learn-r-2/
- Note that is might be worth monitoring the site r-bloggers.com for frequent posts on what others are doing with R in finance and other fields.
- Venables & Smith's "An Introduction to R" is available for free at the r-project website and is a good introduction to the language.
- https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

R for Excel Users

- If you are familiar with Excel but not R, you should immediately check out this tutorial on R for Excel users:
- https://districtdatalabs.silvrback.com/intro-to-r-for-microsoft-excel-users
- This contains some basic information on R that should be helpful in getting started with the course.

R for Matlab Users

- People who are familiar with Matlab should have very little trouble learning R, although the main problem will be becoming familiar with R's quirks.
- This link contains a 50 page (!) document on "R for Matlab Users": https://cran.r-project.org/doc/contrib/Hiebeler-matlabR.pdf
- For people looking for a quicker solution, the following contains a list of Matlab commands and their R analogues: http://mathesaurus.sourceforge.net/octave-r.html

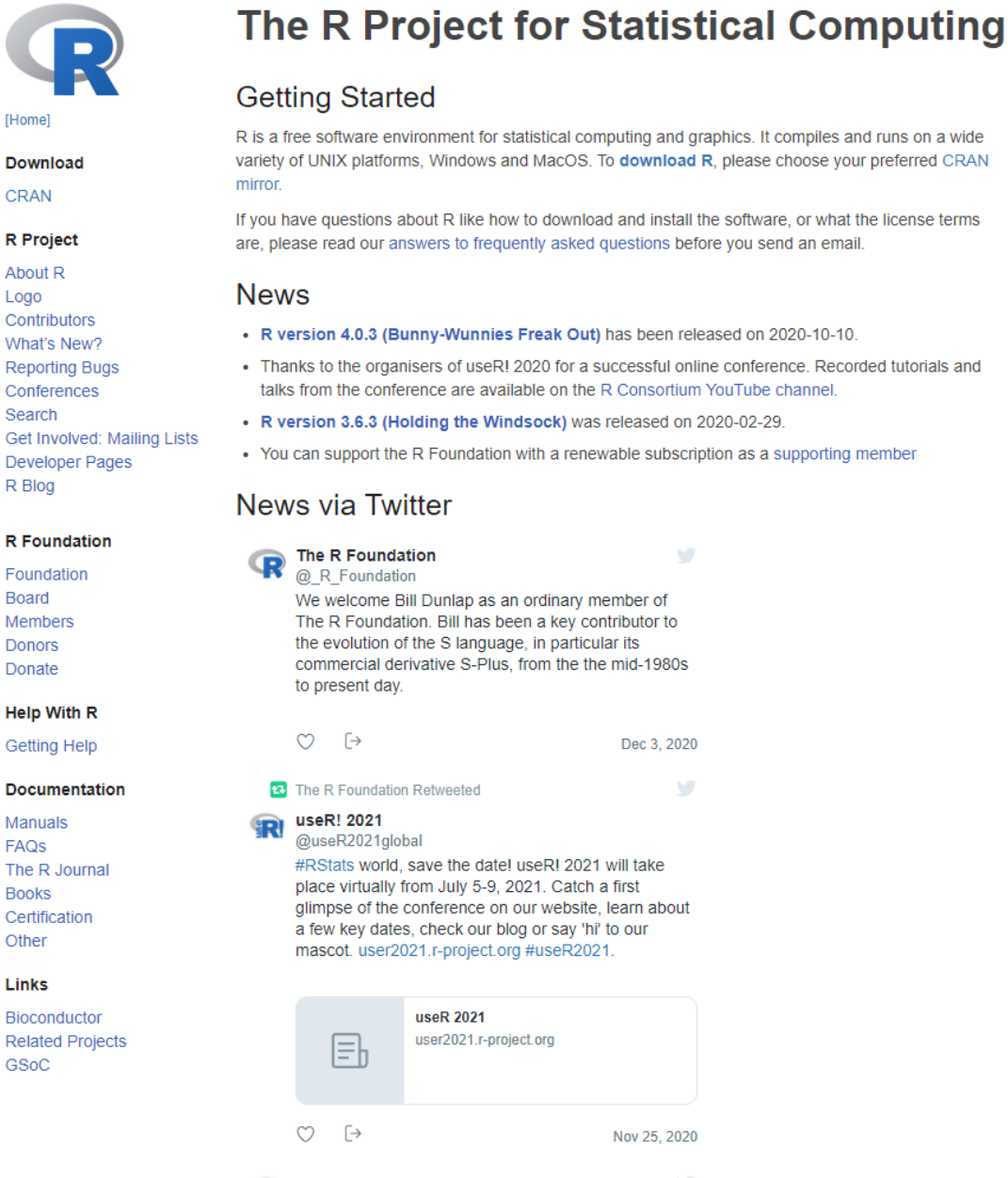
Next Steps

- This slide deck contains a *very brief, very minimal* introduction to R. It contains, at least, examples that will provide enough information/hints to get complete most of Homework #1.
- I recommend that you make an effort to explore R and work through some tutorials/get started working through your R reference book of choice immediately.
- Time spent on learning R and programming concepts in the next 1-2 weeks will make the rest of the course much, much easier.

Getting R

- The core language, plus the R interpreter and a few key add on packages are available by installing what is known as "Base R"
- While R is frequently updated, I recommend that for the duration of the semester you stick with one version (and in general, it might be a good idea to wait a bit after new versions are released before you upgrade to ensure that all your add on packages will work with the new version)
- Base R can be downloaded from https://www.r-project.org/

Base R



[Home]

Download

CRAN

R Project

About R
Logo
Contributors
What's New?
Reporting Bugs
Conferences
Search
Get Involved: Mailing Lists
Developer Pages
R Blog

R Foundation

Foundation
Board
Members
Donors
Donate

Help With R

Getting Help

Documentation

Manuals
FAQs
The R Journal
Books
Certification
Other

Links

Bioconductor
Related Projects
GSoC

The R Project for Statistical Computing

Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News

- **R version 4.0.3 (Bunny-Wunnies Freak Out)** has been released on 2020-10-10.
- Thanks to the organisers of useR! 2020 for a successful online conference. Recorded tutorials and talks from the conference are available on the R Consortium YouTube channel.
- **R version 3.6.3 (Holding the Windsock)** was released on 2020-02-29.
- You can support the R Foundation with a renewable subscription as a supporting member

News via Twitter

The R Foundation
@_R_Foundation
We welcome Bill Dunlap as an ordinary member of The R Foundation. Bill has been a key contributor to the evolution of the S language, in particular its commercial derivative S-Plus, from the the mid-1980s to present day.

Dec 3, 2020

The R Foundation Retweeted

useR! 2021
@useR2021global
#RStats world, save the date! useR! 2021 will take place virtually from July 5-9, 2021. Catch a first glimpse of the conference on our website, learn about a few key dates, check our blog or say 'hi' to our mascot. user2021.r-project.org #useR2021.

useR 2021
user2021.r-project.org

Nov 25, 2020

Rstudio

- While Base R technically does contain everything we need to get started, we'll actually use a 3rd party *Integrated Development Environment* or IDE to interact with R and write our scripts.
- Rstudio works with R and adds features like code completion and other enhancement tools, to allow higher productivity.

Rstudio

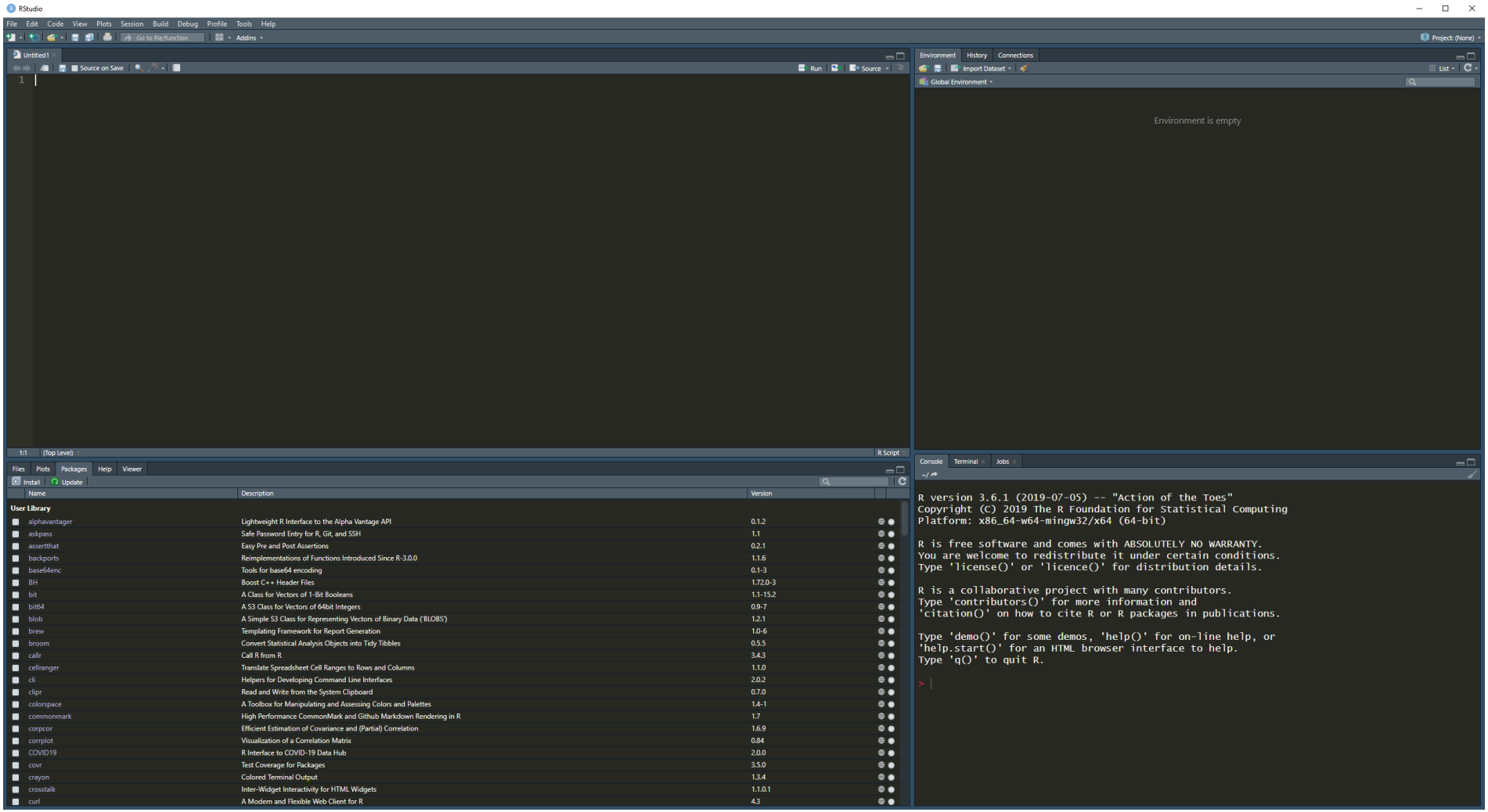


Figure 5: The Rstudio IDE

Wrap Up

You should:

- Install Base R
- Install R Studio

and spend time learning R. If you do this, the rest of the course will be quite manageable.

Note: For Rstudio in particular, a handy tip sheet can be found at http://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf

A Guide to Reading These Slides

These slides were written in Rstudio using the Rmarkdown files format. This is a way to integrate R code and data with the presentation format. I can write R code within the slides themselves, and then when I compile the document, the code executes, and produces the desired output.

```
> rnorm(3)
```

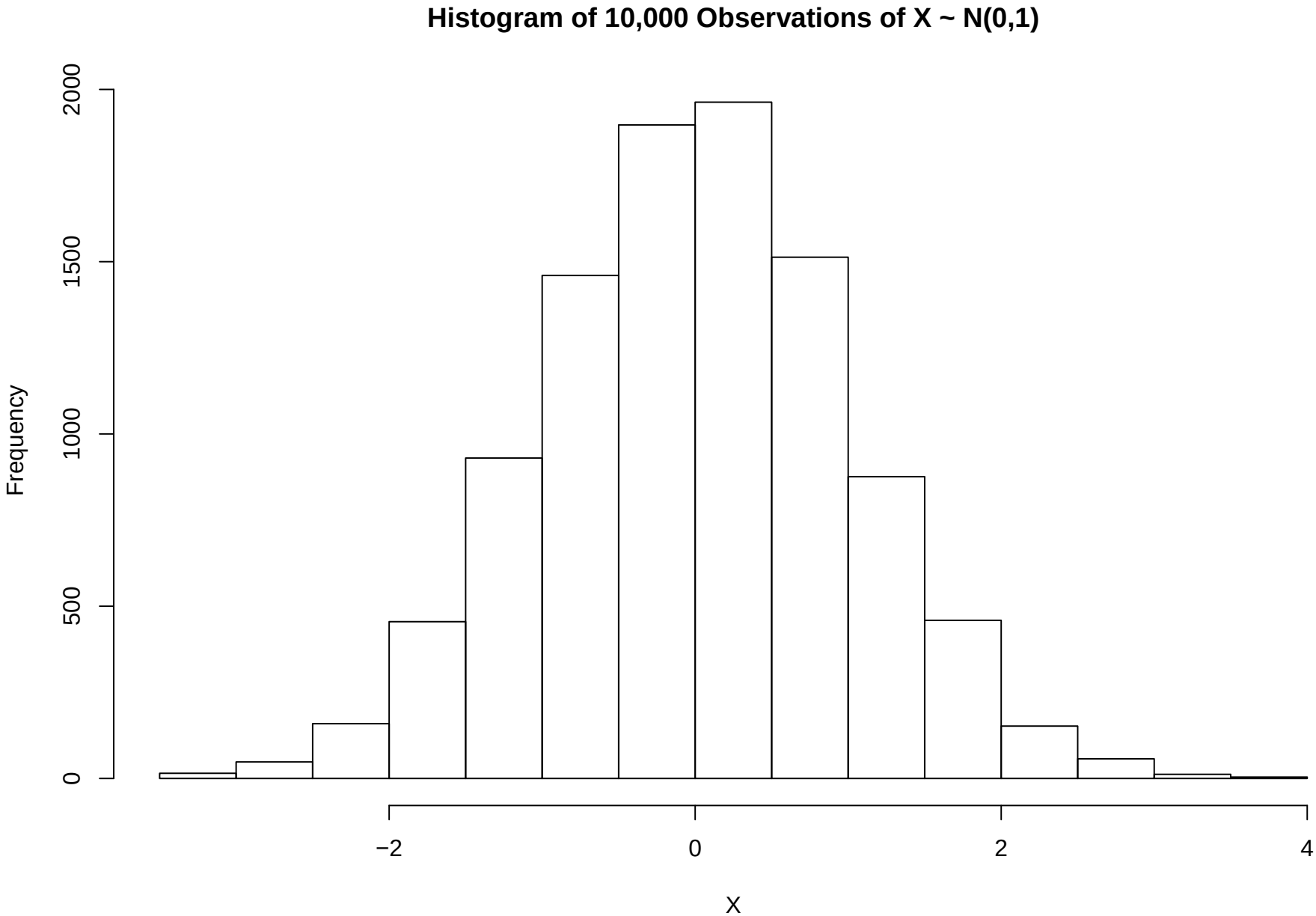
And will produce output along the lines of

```
## [1] -0.3169354  0.2416046 -0.2219875
```

The following code produces a histogram of some normally distributed random numbers:

```
> hist(rnorm(10000))
```

A Guide to Reading These Slides

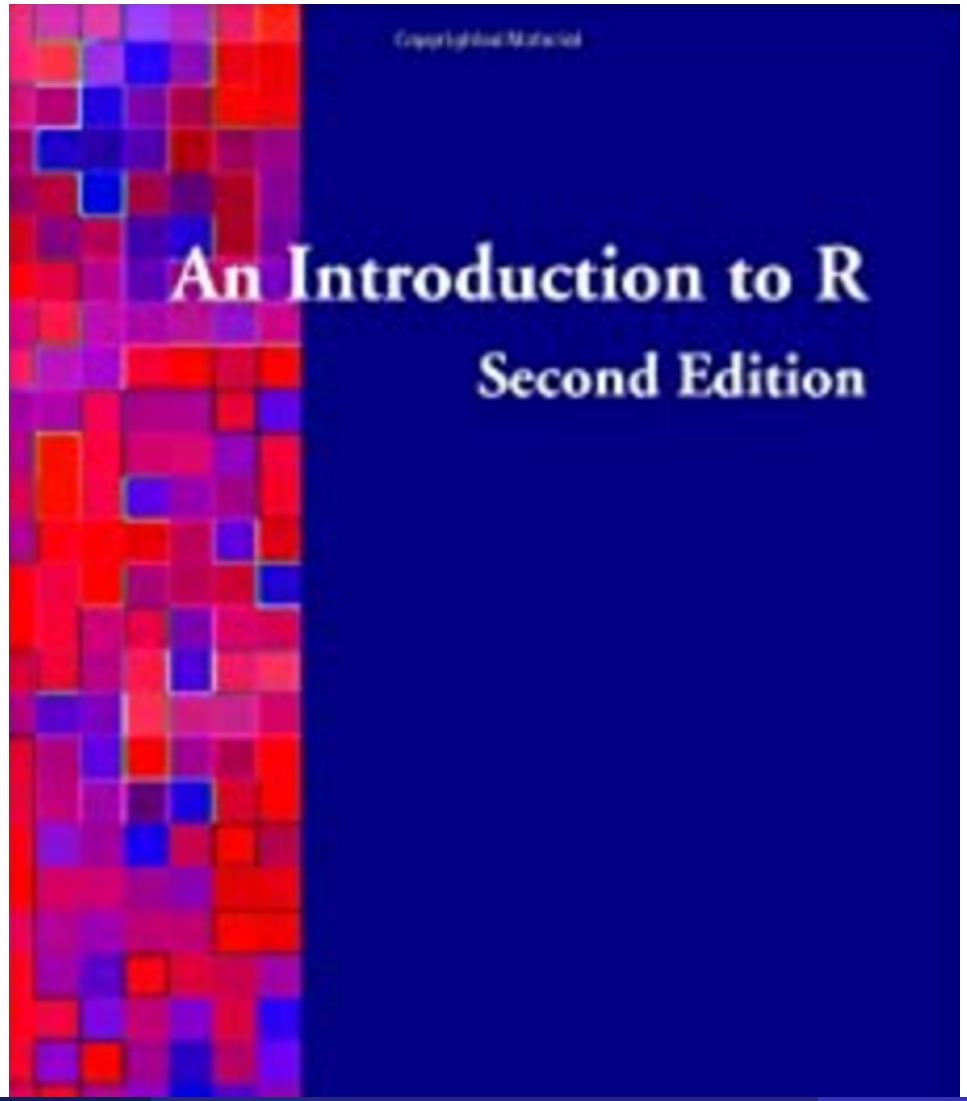


Getting Help

- Pretty much everything you need to do in this class has been done before and has been struggled with before.
- If you run in to trouble, google "r [whatever]" and most likely there is a Stack Exchange or forum post on exactly the trouble you're having.
- However, there are some keyways to get help in learning R.

Read the Manual

- A great basic resource for R is Venables & Smith's "An Introduction to R" - this covers all the basics of the core R language



R's Internal Help

```
> help.start()
```

Figure 7: R Help

help.search()

At the command line, you can use *help.search()* when you don't know what you are looking for exactly (that is, you don't know the name of the function you're looking for):

```
> help.search("random")
```

This will launch a website with the search results.

help() and args()

When you do know the name of the function you want help with (for example, you want to know the details of a function's usage):

```
> help(rnorm)
```

The args() function will tell you the arguments used for a certain function:

```
> args(rnorm)
```

```
## function (n, mean = 0, sd = 1)
## NULL
```

Online Help

- http://rseek.org is likely the best R search engine for R specific help.
- Stack Exchange has become a very solid Q&A site for help on many programming languages and concepts including R.
- Finally, there are the mailing lists available for signup at https://www.r-project.org/mail.html - the R-Sig-Finance list is particularly useful for finance specify questions (please obey the "Rules" of this list, as failure to do so often gets a less than helpful response. . .)

How is R Used?

- You may be familiar with performing statistical routines in software like MS Excel, which is menu driven - you select data, select a routine, and perform an operation.
- R is not natively menu driven - it is command (text) driven.
- We interact with R either by entering commands one at a time at the interpreter prompt, or we write programs or scripts (basically text files saved with a ".R" extension) and run them (by using the *source("filename.R")* command.)

The Rstudio Interface

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa
21	5.4	3.4	1.7	0.2	setosa
22	5.1	3.7	1.5	0.4	setosa
23	4.6	3.6	1.0	0.2	setosa
24	5.1	3.3	1.7	0.5	setosa
25	4.8	3.4	1.9	0.2	setosa
26	5.0	3.0	1.6	0.2	setosa

R as a Big Calculator

We can use R interactively at the command prompt. R recognizes typical arithmetic operators like +, -, *,and /:

```
> 2+2
```

```
## [1] 4
```

```
> 8-5
```

```
## [1] 3
```

```
> 2*5
```

```
## [1] 10
```

```
> 16/4
```

```
## [1] 4
```

```
> 3^3
```

R as a Big Calculator

Note that unlike Matlab, R is not natively a matrix language, and so the standard operators may not produce the expected output if you are working with matrices:

```
> mat1 <- matrix(1:4, nrow=2, byrow=TRUE)
> mat2 <- matrix(5:8, nrow=2, byrow=TRUE)
> mat1*mat2
```

```
##      [,1] [,2]
## [1,]    5   12
## [2,]   21   32
```

Note that this is NOT the result of a matrix multiplication of mat1 and mat2!

Matrix Operators

The standard R multiplication operator * performs "elementwise" multiplication, not a matrix multiplication. For matrix operations, we need to use the special matrix operators:

- Matrix Multiplication: A %*% B
- Matrix Transpose: t(A)
- Extract Diagonal: diag(A)
- Sum of Row Elements: rowSums(A)
- Sum of Column Elements: colSums(A)

Etc...

Using R Interactively

Anything that can be done in R can be done interactively - the command line/prompt is a great way to experiment and learn new functions and explore the help features:

```
> x <- c(1, 3, 5)
> sqrt(x)
```

```
## [1] 1.000000 1.732051 2.236068
```

```
> y <- c(5, 8, 4)
> x%*%y
```

```
##      [,1]
## [1,]   49
```

Section 3

The R Language

R Packages

- Functions in R are stored in *packages*. Base R includes several "core" packages such as "base", "stats", and "graphics".
- Add-on on packages are stored online at the *Comprehensive R Archive Network - CRAN* (or, for more experimental work, at a 3rd party host) and can be installed with the *install.packages()* function.
- Once a package is installed, it can be "loaded" into memory (this needs to be done each time you start a session) with the *library()* functions - once this is done, all the functionality of that package is available to you.
- Currently, there are just over 12,000 add on packages hosted at CRAN. A handy way to view them in a meaningful way is to use "CRAN task views" - these are sites maintained by a subject matter expert who collects a list of all packages useful to users in a field (like "Finance").
- CRAN Task Views: https://cran.r-project.org/web/views/

The R Language

- R is an *object oriented* language that makes extensive use of functions to act on objects.
- For example, using the *combine* function *c()*, I can create a a *vector* object containing the values 1, 5, and 8, and assign that object to a variable named "my.vector":

```
> my.vector <- c(1, 5, 8)
> my.vector
```

```
## [1] 1 5 8
```

Section 4

Data Types

Data Types

- R, like any programming language, contains a number of primitive data types.
- These include *integer, double, complex, character, logical, and raw*.
- The *double* data type is used to represent floating point numbers (basically, numbers with decimals), while *character* data is just text (often called a "string" in other languages).
- The *logical* data type can take only two values - TRUE or FALSE (also called "Boolean" values.)
- Other (compound) data structures area available in R to hold collections of these more primitive data types.

R Data Structures

- In R, we store (temporarily, for future retrieval or analysis) data in various data structures, including *vectors*, *lists*, *matrices*, *arrays*. *factors* and *data frames*
- Chapter 7 of "R in a Nutshell" has a good overview of these, but we'll look at each briefly now.

Vectors

Vectors are a simple structure that contain a single type of data. We generally use the *c()* combine function to create vectors:

```
> my.vector <- c(1, 5, 8)
```

c() will also *coerce* all elements of a vector to be the same *type*:

```
> my.vector2 <- c(1, "horse", 4)
> typeof(my.vector2)
```

```
## [1] "character"
```

Vectors

Once we have created a vector, we can use *indexing* to access subsets of the array. Vectors are indexed by *position* by, starting with 1 as the first position.

```
> my.vector <- c(1, 5, 8)
> my.vector[1]
```

```
## [1] 1
```

```
> my.vector[4]
```

```
## [1] NA
```

Lists

Lists are more complicated than vectors, in that they can contain multiple data types and also can contain *names* for each of the elements of the list:

```
> my.list <- list(firstname="Richard", lastname="Jones",
+                 age=40)
```

Like vectors, we can index lists by position:

```
> my.list[1]
```

```
## $firstname
## [1] "Richard"
```

But we can also index lists by name:

```
> my.list$firstname
```

```
## [1] "Richard"
```

Matrices

A matrix extends the concept of a vector into two dimensions:

```
> my.matrix <- matrix(c(8, 1, 6, 3, 5, 7, 4, 9, 2),nrow=3,
+                     byrow=TRUE)
> my.matrix
```

```
##      [,1] [,2] [,3]
## [1,]    8    1    6
## [2,]    3    5    7
## [3,]    4    9    2
```

As with vectors, matrices can contain only one data type.

Arrays

Arrays extend vectors to multiple dimensions:

```
> my.array <- array(1:8, dim=c(2,2,2))
> my.array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Arrays

Like vectors and matrices, we index arrays by position:

```
> my.array <- array(1:8, dim=c(2,2,2))
> my.array[,,1] # all rows and columns of the "first slice"
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Data Frames

Data frames are the most common data structures in Base R, and are tabular representations of potentially multiple types of data (similar to a spreadsheet):

```
> my.data.frame <- data.frame(cbind(c("Linda","Dave","Karen"),
+                                   c("F","M","F"),
+                                   c(25, 24, 27)))
> colnames(my.data.frame) <- c("Name","Gender","Age")
> my.data.frame
```

```
##    Name Gender Age
## 1 Linda      F  25
## 2  Dave      M  24
## 3 Karen      F  27
```

Other Data Structures

- There are other data structures that we have not discussed here, such as "factors" (another way of representing strings).
- Add on packages for R can implement their own data structures - some common ones include various structures for representing time series data (such as stock prices, for example) as implemented by the *zoo* and *xts* packages.
- In general, most of our time in this class will be working with the main data structures we've viewed so far as well as time series objects.

Section 5

Functions

Functions

- R implements subroutines in functions - base R, and the add on packages, accomplish tasks via the use of functions.
- As example that we have already used is the combine() function, which combines multiple elements into a vector. R functions, in general, can be nested together to combine multiple operations into one line of code - we can, for example, embed the c() function inside a call to the sum() function to get the sum of the elements of a vector:

```
> sum(c(1, 5, 8))
```

```
## [1] 14
```

Functions

Although Base R and the core packages come pre-loaded with extensive functionality, we can also create our own functions in the R language, using the function() command:

```
> my.function <- function(input) {
+   # code
+ }
```

This shows the structure of a call to the function command. As a specific example, we can create a function to calculate the standard deviation of a set of numbers, and then compare our results against R's built in sd() function.

Functions

```
> sample.data <- rnorm(30)*2 # 30 random variates X ~ N(0,2)
> my.sd <- function(x) {
+   sqrt(sum((x-mean(x))^2)/(length(x)-1))
+ }
> my.sd(sample.data)
```

```
## [1] 1.888333
```

```
> sd(sample.data)
```

```
## [1] 1.888333
```

Obviously a trivial example, but we will see through the course that writing functions becomes a significant part of writing R code.

Programming in R

- "Computer programming (often shortened to programming, sometimes called coding) is a process that leads from an original formulation of a computing problem to executable computer programs."
(https://en.wikipedia.org/wiki/Computer_programming)
- In general, we write *functions* that take *inputs*, perform some sort of routine to modify, analyze, or otherwise process those inputs, and produce *outputs*, and we link these functions sequentially, often with commands that conditionally execute certain statements or alternatively repeat the same commands multiple times.
- In this class, and in general in using R, these routines or algorithms tend to be statistical or mathematical in nature, in that they explicitly seek to implement statistical techniques to analyze data, for example.

Booleans/Conditional Statements

In general, we need the ability to perform one set of actions if a certain condition is met, and another set of actions if the condition is not met. In R, the basic way to do that is with the if() statement:

```
> # test whether  a randomly generated number is greater than .5
> (x <- runif(1))
```

```
## [1] 0.6429476
```

```
> {if (x > .5) {
+   print("x is greater than .5")
+ } else {
+   print("x is not greater than .5")
+ }}
```

```
## [1] "x is greater than .5"
```

Conditional Statements

- Other conditional statements include the ifelse() function as well as the switch() function.
- Each of these allow for executing code blocks conditionally on whether a condition is met (that is, whether the result of a test is “TRUE” or “FALSE”)

Apply Functions

```
> help.search("apply",package="base")
```

Apply Functions

While standard loops are available in R, they tend to be fairly slow, and alternatives exist, including a set of *apply()* functions. *apply()* applies a function over one "margin" of an array or matrix - in english, this means that, for example, we can apply a function to the columns of a matrix, or the rows of a matrix, for example:

```
> my.matrix <- matrix(c(8, 1, 6, 3, 5, 7, 4, 9, 2),nrow=3,
+                     byrow=TRUE)
> apply(my.matrix,1,sum) # row sums (margin = 1)
```

```
## [1] 15 15 15
```

```
> apply(my.matrix,2,sum) # column sums (margin = 2)
```

```
## [1] 15 15 15
```

Apply Functions

Other apply functions are available for more complicated data structures. For example, *lapply()* applies a function to each element of a list, and returns a list the same length of your original list:

```
> my.list <- list(element1 = 1:5, element2 = 6:10,
+                 element3 = 11:15)
> lapply(my.list, sum)
```

```
## $element1
## [1] 15
##
## $element2
## [1] 40
##
## $element3
## [1] 65
```

Apply Functions

- Other apply functions can be a bit tricky to implement.
- The site https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/ has examples on all the Base apply functions.

Section 6

Loops

for loop

We often need to execute code repetitively - for example, performing a test for each element of a vector, or some calling a function with multiple inputs. The simplest way to do this is with a *for* loop:

```
> for (i in 1:4) {print(i)}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

for loop

for loop iterators do not necessarily need to be numbers:

```
> my.vec <- c("cat","crocodile","ocelot")
> for (i in my.vec) {print(paste(i, "is an animal",sep=" "))}
```

```
## [1] "cat is an animal"
## [1] "crocodile is an animal"
## [1] "ocelot is an animal"
```

while loop

Another loop construct is the *while* loop - this continues to execute as long as a condition is true:

```
> i <- 1
> while (i <= 4) {
+   print(i)
+   i <- i+1
+ }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Section 7

Import/Export

Reading Data into R

One of the easiest ways to get data into an R environment is to read it from a text file, such as a .csv file. We can do this with the read.table() function:

```
> mydata <- read.table("data/mydata.csv",header=TRUE,
+                      sep=",",as.is=TRUE)
> mydata
```

```
##    Name Gender Age
## 1 Linda      F  25
## 2  Dave      M  24
## 3 Karen      F  27
```

Exporting Data from R

Similarly, once we have data in R that we want to save, a convenient method to export it is to write it back to a .csv file, with the write.table() function:

```
> write.table(mydata,file="mynewdata.csv",sep=",",row.names=FALSE,
+             col.names=TRUE)
```

This will write the mydata data frame out to a .csv file named "mynewdata.csv", using commas as the separater, and preserving the column names from the data frame. The data can then be re-imported later for more analysis, or viewed in other software like MS Excel.

Section 8

Plotting in R

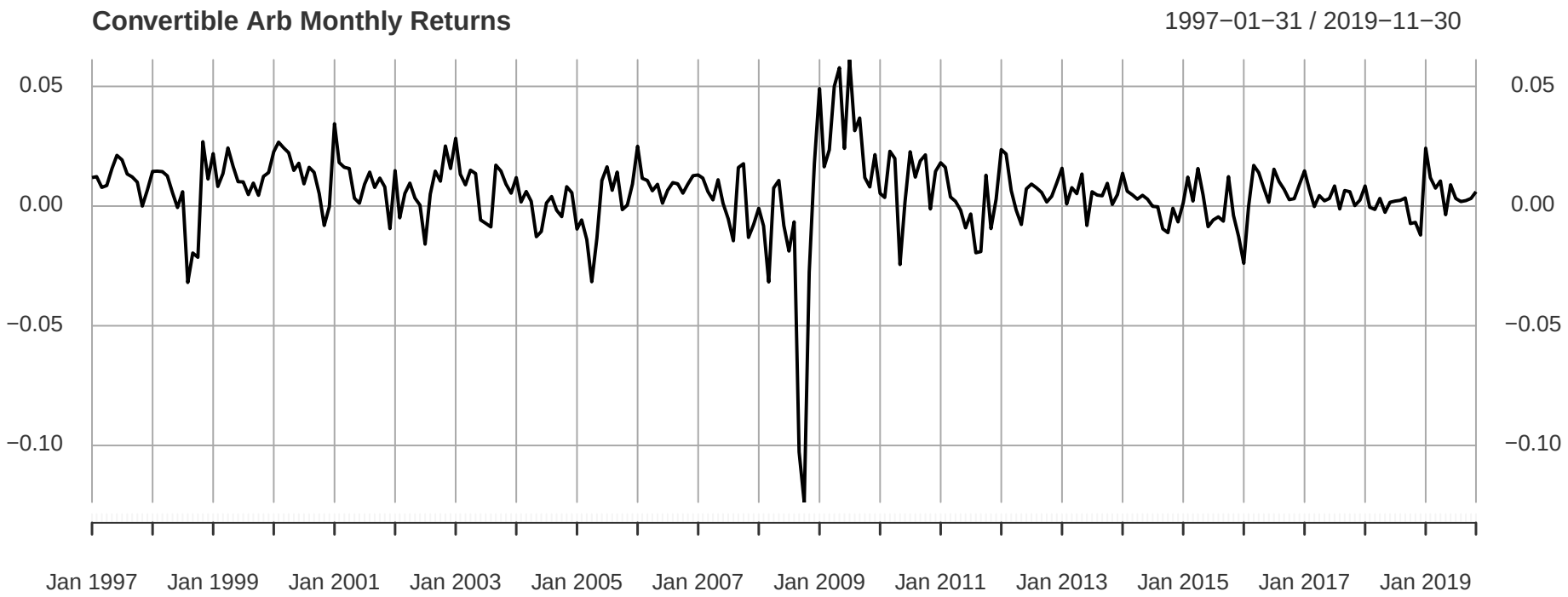
Plotting in R

- In R, we have legacy "Base" plotting capabilities as well as newer *ggplot2* style graphics.
- Base R graphics are powerful, and can produce publication quality plots, but can be a bit tricky to learn, as the controls are a mix of high level easy to read code and more fundamental or abstract elements that can be more difficult to master.
- ggplot2 (Grammar of Graphics) is a package developed by Hadley Wickham that has become extremely popular in recent years for its ability to produce highly readable graphics.
- For now, we won't worry about learning ggplot2 syntax, and will focus on plotting methods from Base R.

R Base - Line Plot

After loading the PerformanceAnalytics package and calling the data(edhec) command, we can produce a simple line plot of the first column of the EDHEC data:

```
> plot(edhec[,1],main="Convertible Arb Monthly Returns",
+      xlab="", type="l")
```



R Base - Line Plot

Note that EDHEC is an xts object:

```
> class(edhec)
```

```
## [1] "xts" "zoo"
```

- This is a good point to come back to R's functions, which are "generic" in nature. Since R is object oriented, functions can take on different features depending on the class of an object.
- In this case, the plot() function works differently depending on the class of the object that you are passing to it as a parameter (in this, case, we got "plot.zoo()")
- We don't need to worry about this too much for now, but you should know that R functions may behave differently depending on the class of the object you pass to it - when you call a generic function that has methods specific to an object type, R automatically attempts to determine what object type you've passed to the function, and uses the most appropriate version of that function.

R Base - Line Plot

Lets plot a non-xts object:

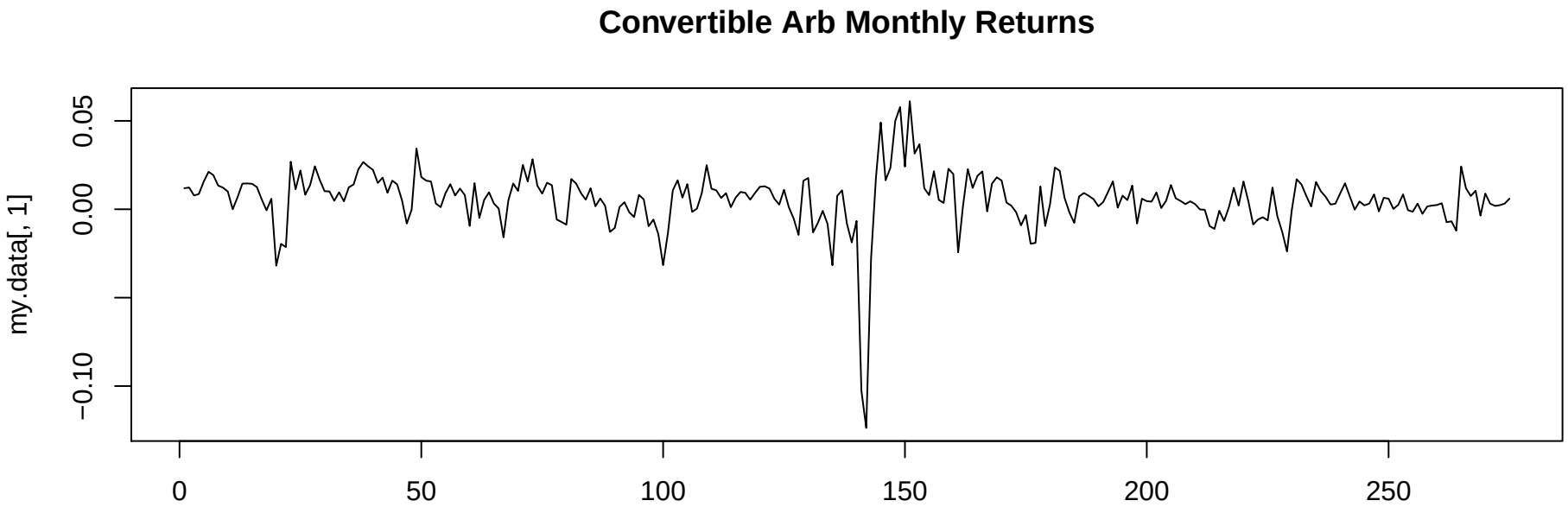
```
> my.data <- coredata(edhec[,1])
> class(my.data)
```

```
## [1] "matrix"
```

We'll observe a few differences.

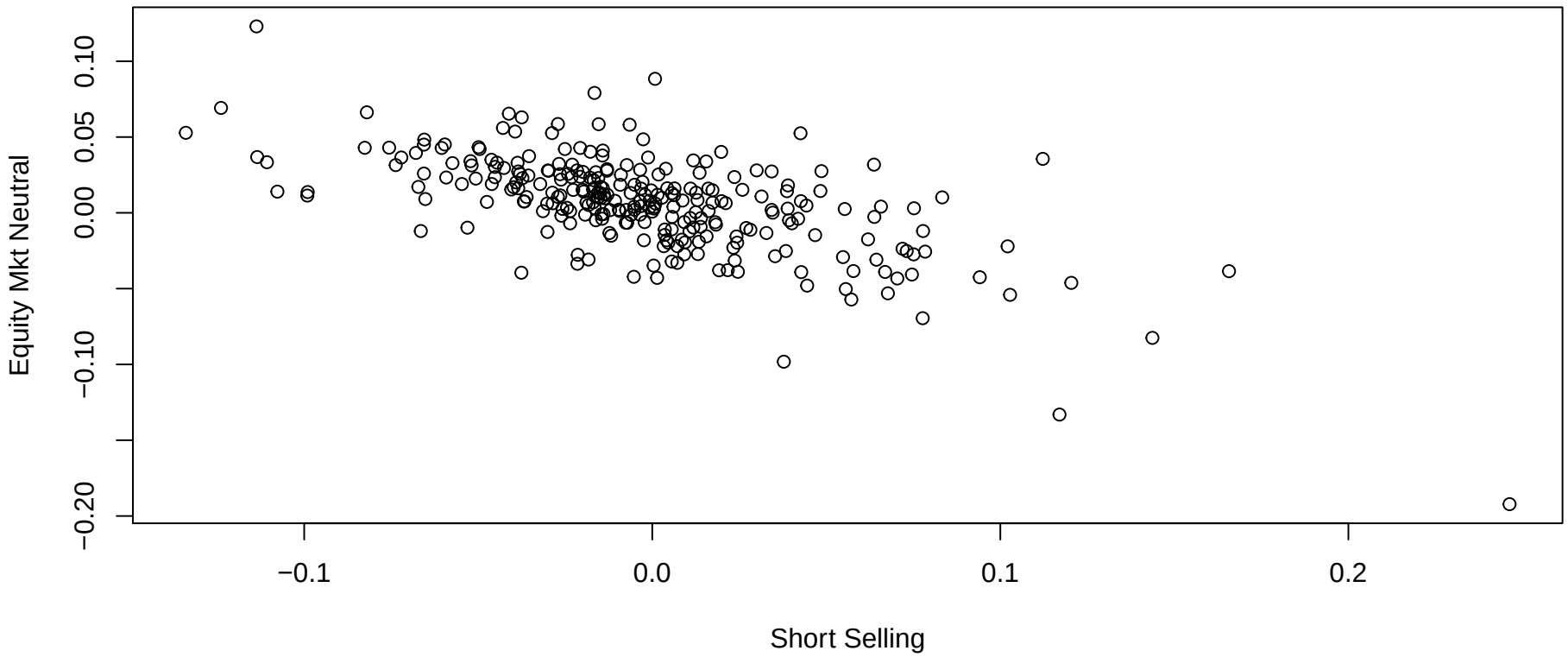
R Base - Line Plot

```
> plot(my.data[,1],main="Convertible Arb Monthly Returns",
+      xlab="", type="l")
```



R Base - Scatter Plots

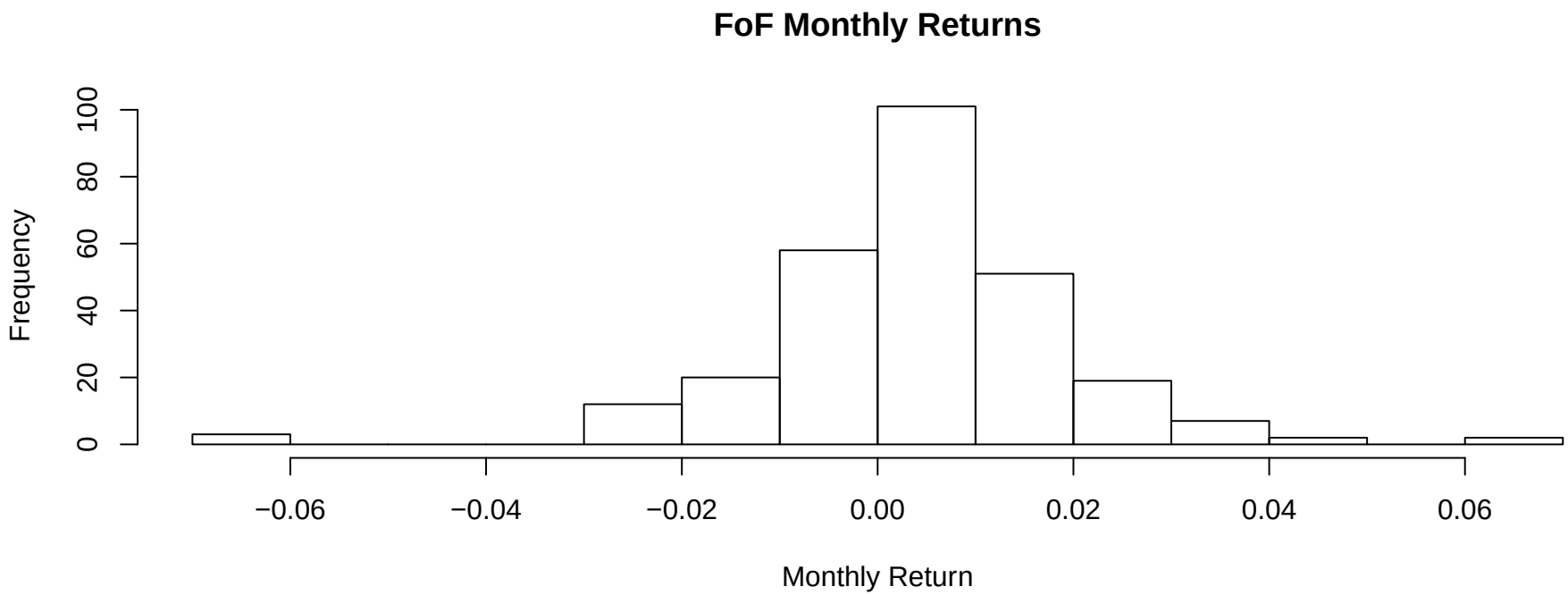
```
> plot(coredata(edhec[,12]),coredata(edhec[,4]),
+      xlab="Short Selling",ylab="Equity Mkt Neutral")
```



R Base - Histograms

Histograms allow us to graphically describe the empirical distribution of a random variable by placing observations into bins:

```
> hist(coredata(edhec[,13]),main="FoF Monthly Returns",
+      breaks=10,xlab="Monthly Return")
```

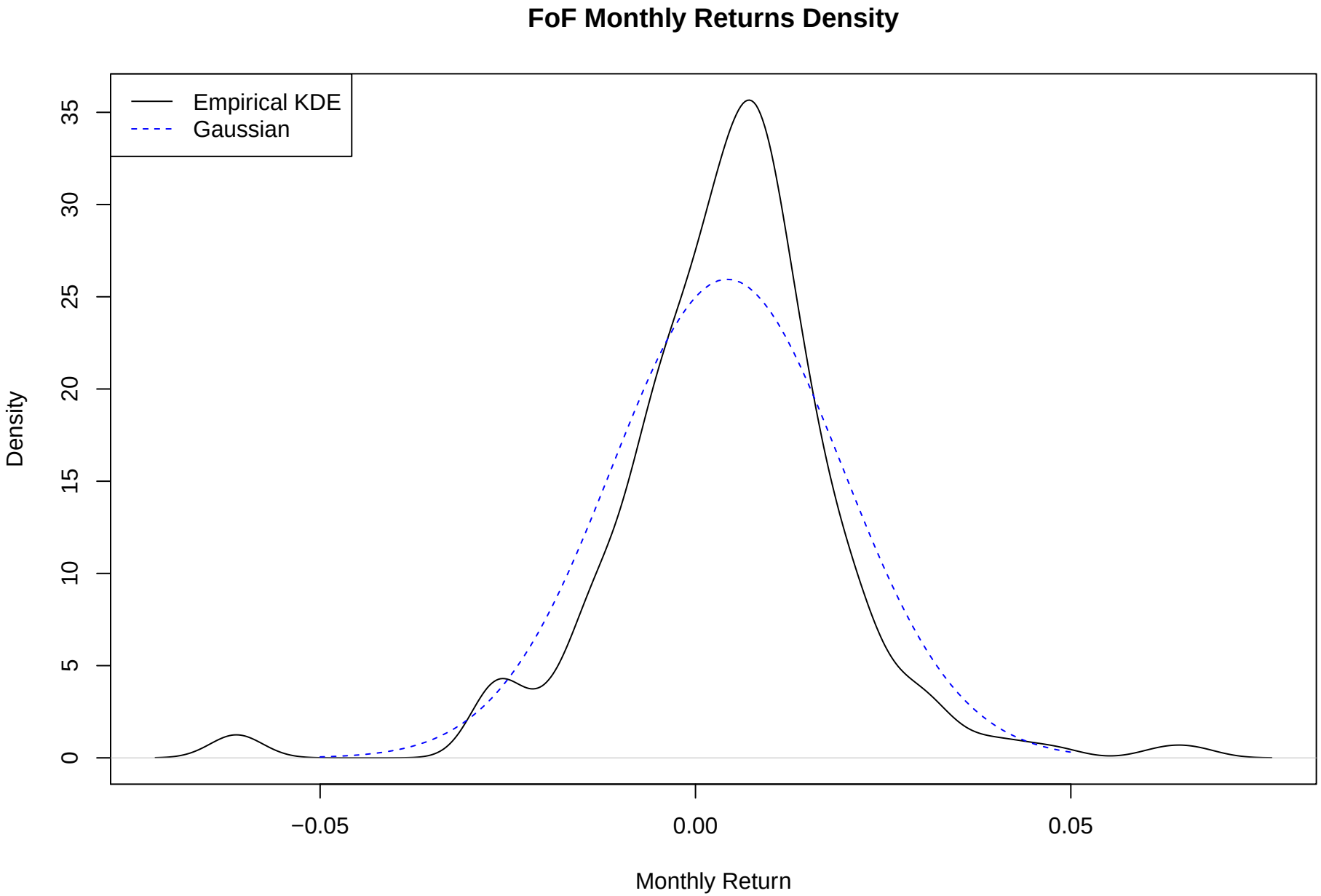


R Base - Kernel Density Estimates

Kernel density estimates are another way of graphically describing univariate distributions in a manner similar to histograms:

```
> plot(density(coredata(edhec[,13])),
+      main="FoF Monthly Returns Density",
+      xlab="Monthly Return")
> lines(seq(-.05, .05, .001),
+       dnorm(seq(-.05, .05, .001),
+             mean(edhec[,13]),
+             sd(edhec[,13])),col="blue",lty=2)
> legend("topleft",
+        legend=c("Empirical KDE","Gaussian"),
+        lty=c(1,2),col=c("black","blue"))
```

R Base - Kernel Density Estimates



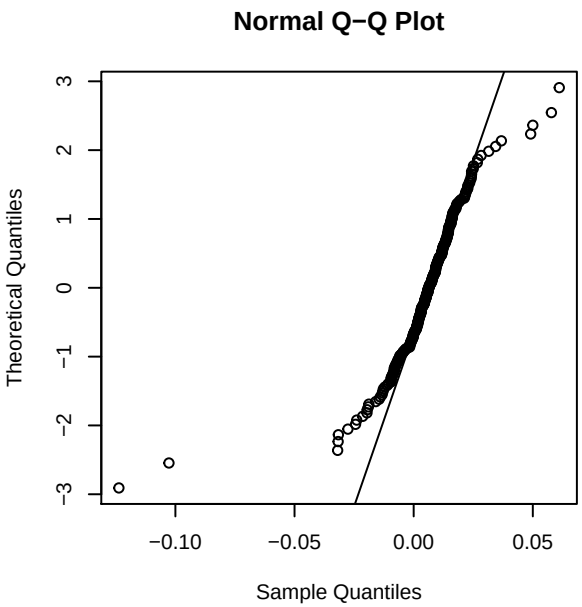
R Base - QQ Plots

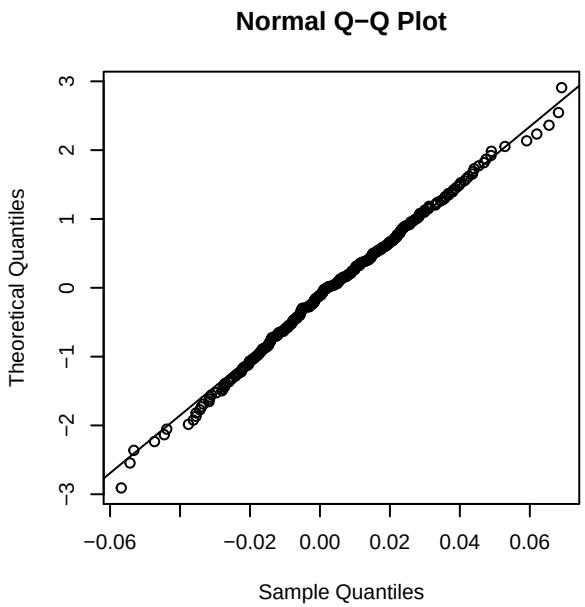
QQ Plots allow us to visually inspect the sample quantiles against a theoretical distribution such as the Gaussian/Normal:

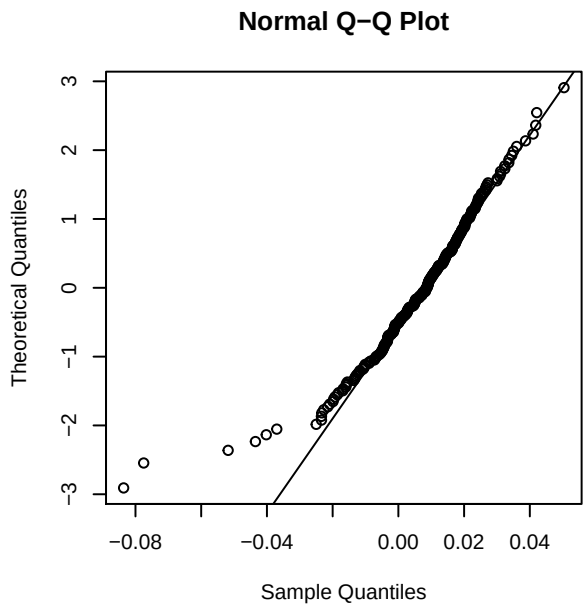
```
> par(mfrow=c(2,3)) # creates a 2x3 grid of plots
> for (i in 1:6) {
+   qqnorm(coredata(edhec[,i]),datax=TRUE)
+   qqline(coredata(edhec[,i]),datax=TRUE)
+ }
```

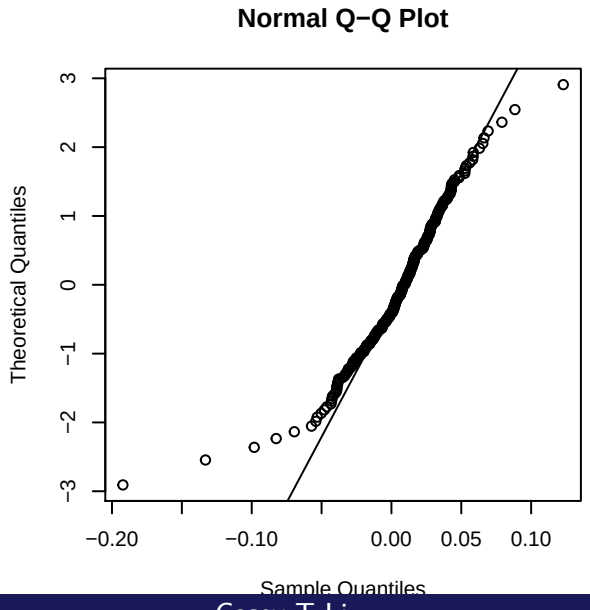
The above code creates QQ normal plots for the first 6 series in the EDHEC dataset.

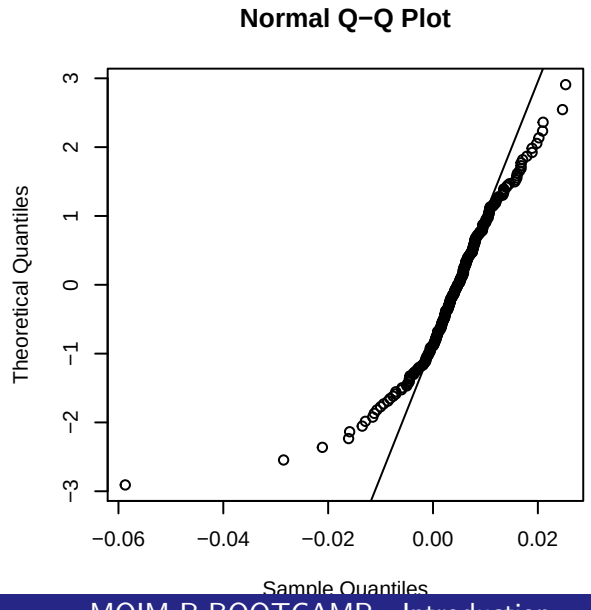
R Base - QQ Plots

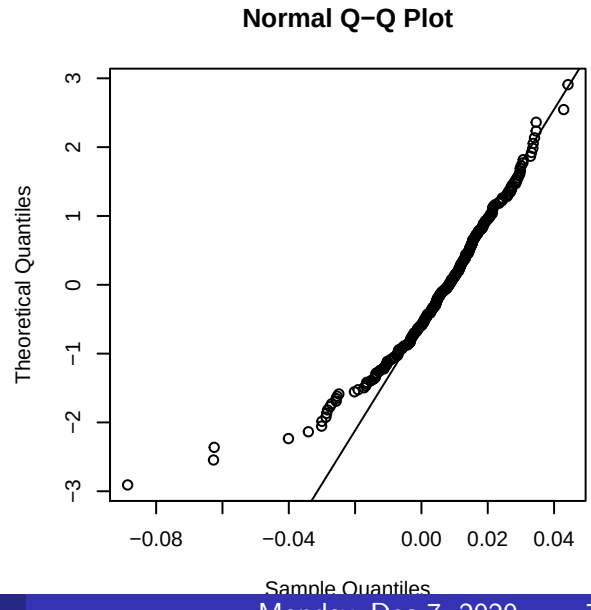








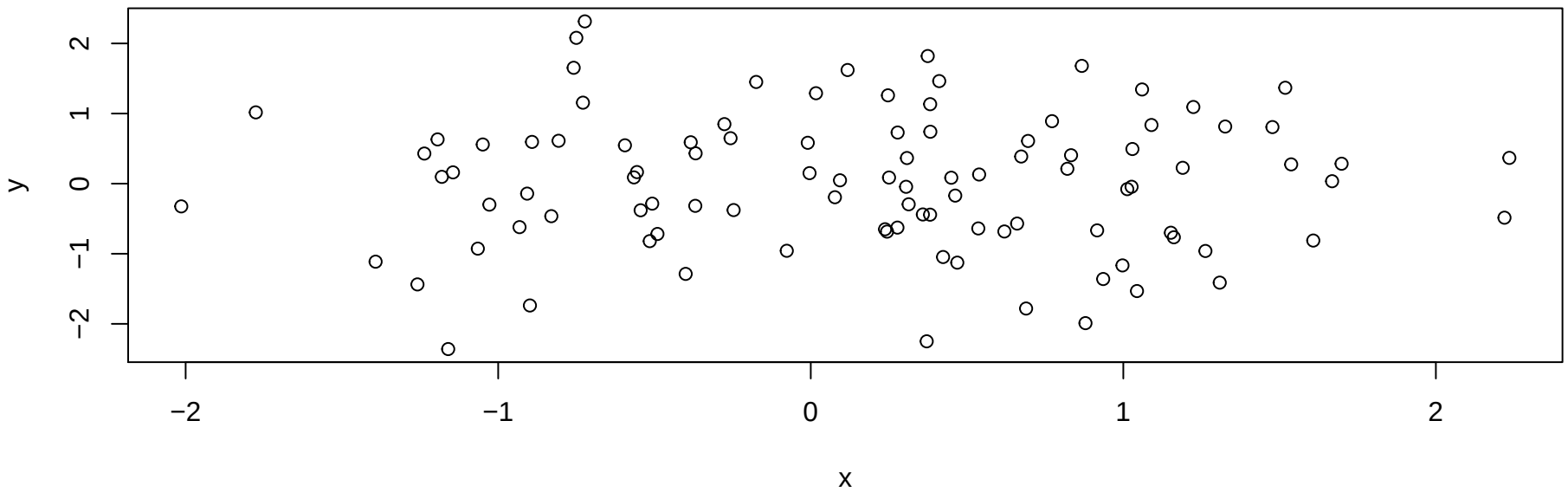




Exporting Graphics from R

It is often handy to be able to export a figure we've created in R to a .png or .pdf file to use in other programs, we can do this easily. Assuming we have a plot to export:

```
> x <- rnorm(100); y = rnorm(100)
> plot(x,y)
```



Exporting Graphics from R

We can export this to a png file using the png "Device":

```
> png(filename="scatterplot.png",width=7,height=5,units="in",
+     res=300)
> plot(x,y)
> dev.off()
```

This will save a 5x7 png file with the name "scatterplot.png" in the working directory for use elsewhere.

- Note that we could also accomplish this vie the GUI of Rstudio (that is, not programmatically).

Section 9

Some Notes

Environment Management

In R, your workspace is known as an "Environment" - all the variables and functions you create are loaded into that environment. Consequently, it is good practice, when you start a new task (or launch a new script) to clear all variables out of the environment with the rm() function:

```
> rm(list=ls())
```

We won't worry too much about managing environments in this class other than to be careful to start new scripts/sessions with an empty environment with *rm(list=ls())*.

rep()

The rep() function allows us to create vectors with (initially) all the same values:

```
> # create a vector of zeros with length 5
> my.empty.vec <- rep(0,5)
> my.empty.vec
```

```
## [1] 0 0 0 0 0
```

cbind() and rbind()

Like c(), the functions cbind() and rbind() combine objects. cbind() does this by columns, while rbind() does this by rows:

```
> my.array <- cbind(c(1, 2, 3),c(4, 5, 6))
> my.array
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
> my.array <- rbind(c(1, 2, 3),c(4, 5, 6))
> my.array
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Testing for Equality

```
> x <- "A"
> y <- "A"
> x == y
```

```
## [1] TRUE
```

```
> x <- "A"
> y <- "a"
> x == y
```

```
## [1] FALSE
```

Testing for Equality

```
> x <- seq(.1,.15,by=.01)
> y <- 10:15/100
> x
```

```
## [1] 0.10 0.11 0.12 0.13 0.14 0.15
```

```
> y
```

```
## [1] 0.10 0.11 0.12 0.13 0.14 0.15
```

```
> x == y
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

Testing for Equality

```
> x <- seq(.1,.15,by=.01)
> y <- 10:15/100
> all.equal(x,y) # tests "near" equality
```

```
## [1] TRUE
```

```
> identical(x,y) # tests "exact" equality
```

```
## [1] FALSE
```

Be careful in how you test for equality.