MQIM R BOOTCAMP

Introduction

Casey T Li

2023 August

1. Bootcamp Week Schedule
2. Introduction to R
3. The R Language
4. Data Types
5. Functions
6. Loops
7. Import/Export
8. Plotting in R

9. Some Notes

10. Basic Exploratory Data Analysis

11. Basic Statistics

Section 1

Bootcamp Week Schedule

Table of Content

- Introduction to R
- The R Language, Data Types, Functions, Loops, Import/Export, Plot
- Basic Exploratory Data Analysis
- Basic Statistics
- Getting Financial Data in R
- R Class, Object and functions
- Data Preparation, Transformation and Visualization (tidyverse package)
- Model Building
- Advanced topics: Rmarkdown, Shiny, Github
- Basic Machine Learning and Deep Learning using R
- Introduction to Python/Matlab
- Introduction to BQL/BQUANT

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?"

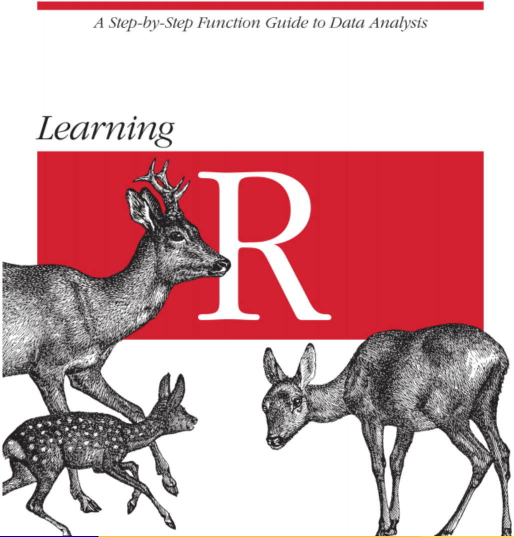
- R is highly flexible, and can be used for statistical analysis, mathematical modeling, and a variety of other tasks (these slides were written in Rstudio using the Rmarkdown format, for example)
- For one off analyses, R may be "overkill", but in the context of institutional investment management, 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?

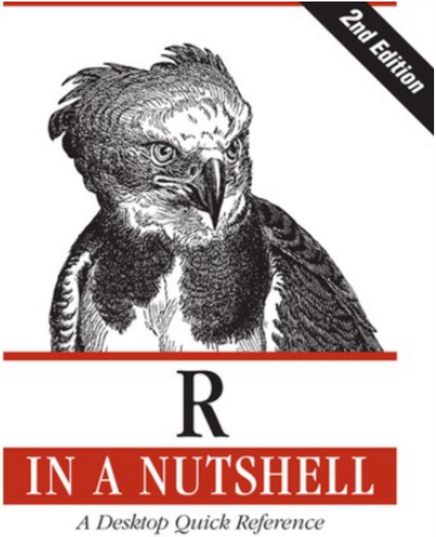
- R is slow. There are ways to write efficient R code, but if extreme low-latency is required, R is probably not the correct tool.
- R (and the packages) is documented unevenly - some functions and packages are extremely well documented, while others are not.
- R is open source and constantly evolving, 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

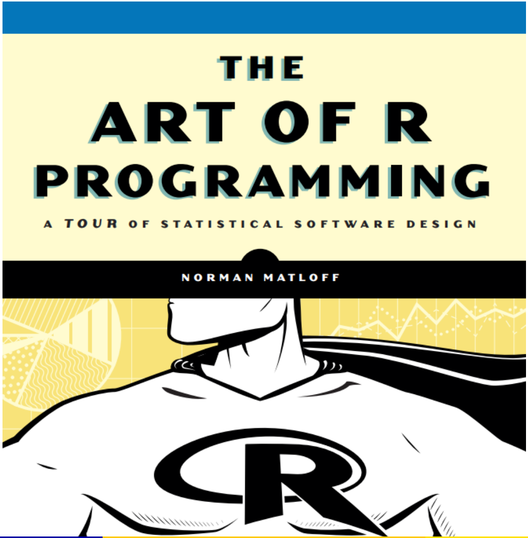


“R in a Nutshell” by Adler



“The Art of R Programming” by Matloff

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



THE ART OF R PROGRAMMING

A TOUR OF STATISTICAL SOFTWARE DESIGN

NORMAN MATLOFF

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

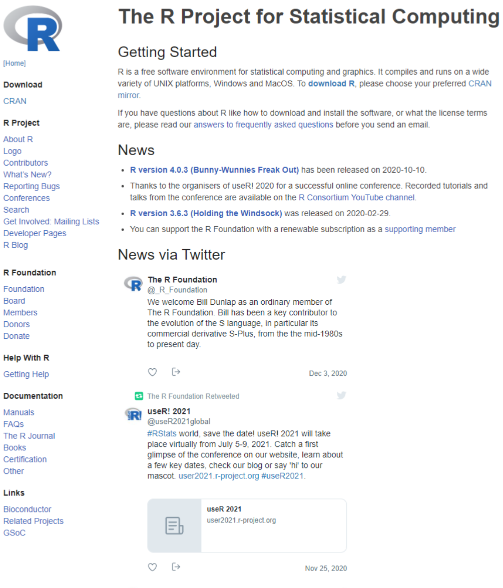
Next Steps

- 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.

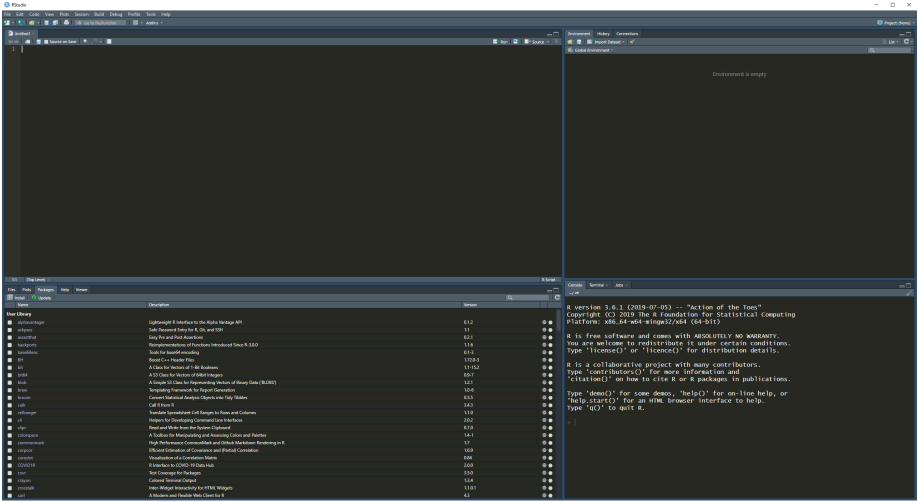
useR 2021
user2021.r-project.org

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



Name	Description	Version
User Library		
alphavantager	Lightweight R Interface to the Alpha Vantage API	0.1.2
askpass	Safe Password Entry for R, Git, and SSH	1.1
assertthat	Easy Pre and Post Assertions	0.2.1
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.6
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.72.0-3
bit	A Class for Vectors of 1-Bit Booleans	1.1-15.2
bit64	A S3 Class for Vectors of 64bit Integers	0.9-7
blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.1
brew	Templating Framework for Report Generation	1.0-6
broom	Convert Statistical Analysis Objects into Tidy Tibbles	0.5.5
callr	Call R from R	3.4.3
cellranger	Translate Spreadsheet Cell Ranges to Rows and Columns	1.1.0
cli	Helpers for Developing Command Line Interfaces	2.0.2
clipr	Read and Write from the System Clipboard	0.7.0
colorspace	A Toolbox for Manipulating and Assessing Colors and Palettes	1.4-1
commonmark	High Performance CommonMark and Github Markdown Rendering in R	1.7
corpcor	Efficient Estimation of Covariance and (Partial) Correlation	1.6.9
corrplot	Visualization of a Correlation Matrix	0.84
COVID19	R Interface to COVID-19 Data Hub	2.0.0
covr	Test Coverage for Packages	3.5.0
crayon	Colored Terminal Output	1.3.4
crosstalk	Inter-Widget Interactivity for HTML Widgets	1.1.0.1
curl	A Modern and Flexible Web Client for R	4.3

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

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.

https://posit.co/download/rstudio-desktop/

Note: For Rstudio in particular, a handy tip sheet can be found at https://rstudio.github.io/cheatsheets/html/rstudio-ide.html

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)
```

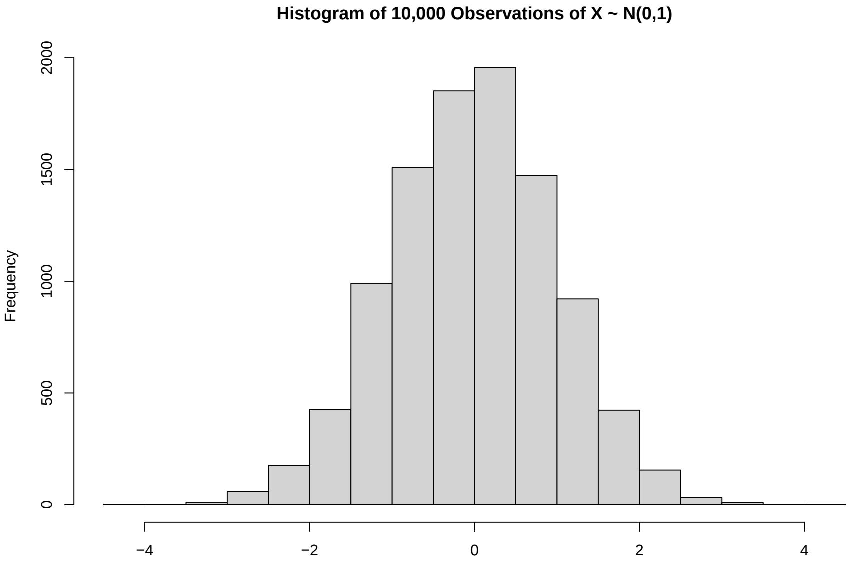
And will produce output along the lines of

```
## [1] 0.1980622 1.6496527 0.7421011
```

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

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

A Guide to Reading These Slides

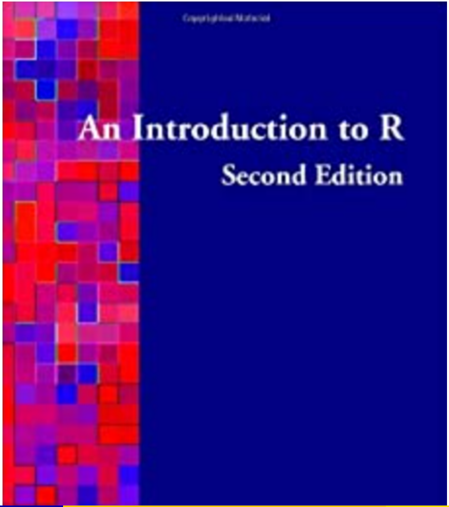


Getting Help

- 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.

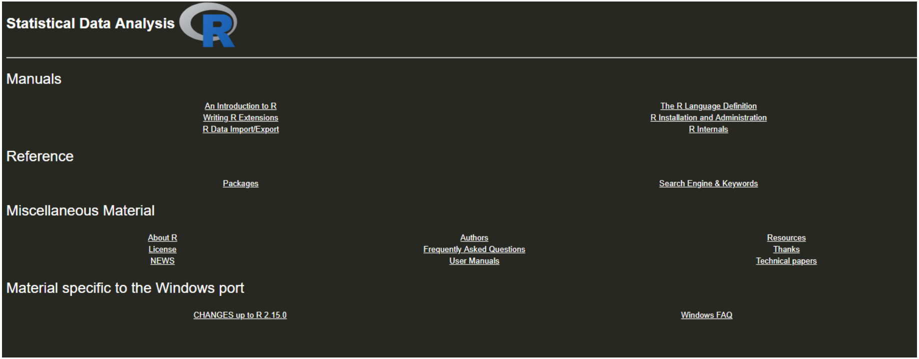
Read the Manual

- 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()
```



Statistical Data Analysis

Manuals

An Introduction to R
Writing R Extensions
R Data Import/Export

The R Language Definition
R Installation and Administration
R Internals

Reference

Packages

Search Engine & Keywords

Miscellaneous Material

About R
License
NEWS

Authors
Frequently Asked Questions
User Manuals

Resources
Thanks
Technical papers

Material specific to the Windows port

CHANGES up to R 2.15.0

Windows FAQ

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
```

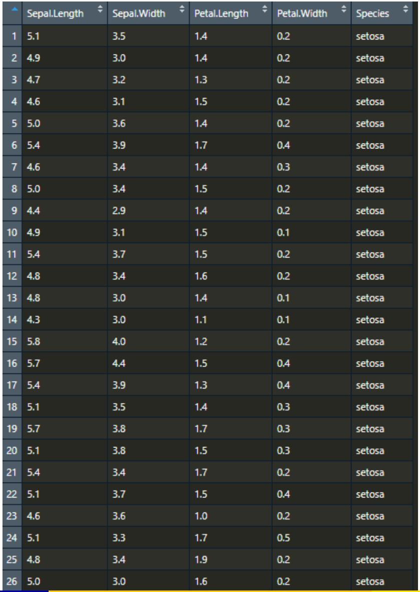
Online Help

- 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] 1.970078
```

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

```
## [1] 1.970078
```

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] 0.2213366
```

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

```
## [1] "x is not 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

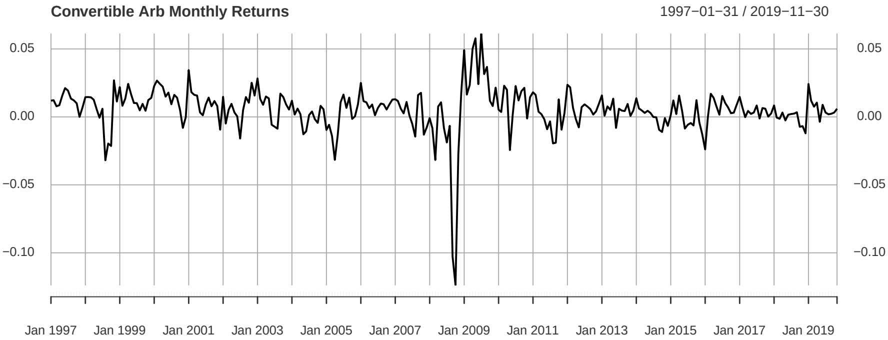
Plotting in R

- 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:

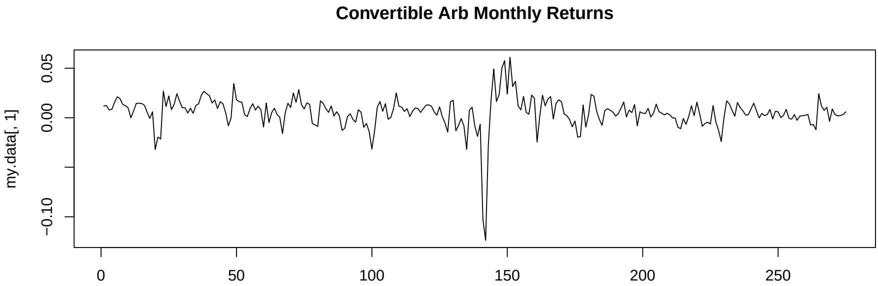
```
> my.data <- coredata(edhec[,1])
> class(my.data)
```

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

We'll observe a few differences.

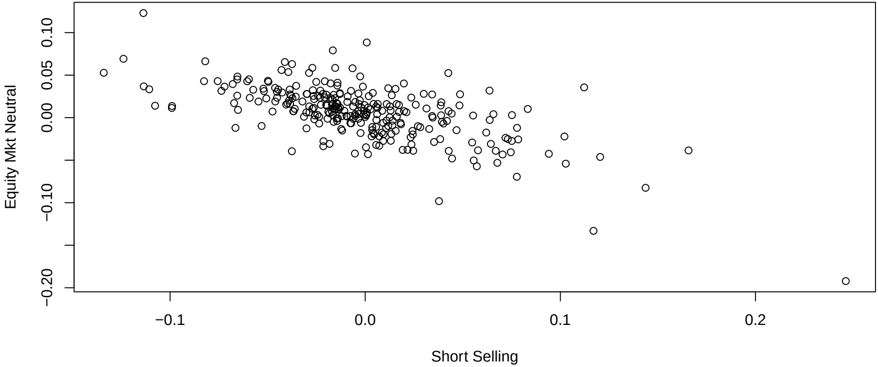
R Base - Line Plot

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



R Base - Scatter Plots

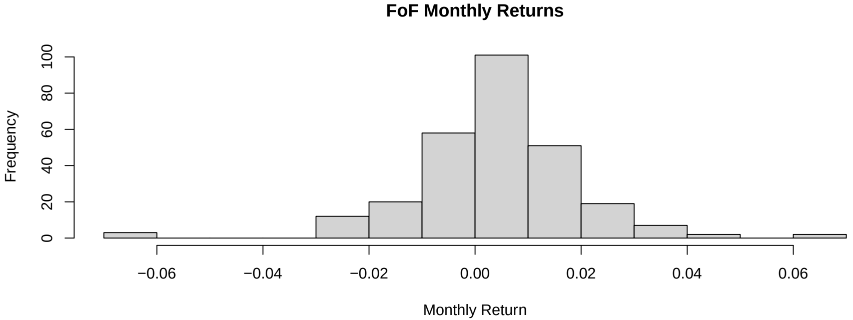
```
> plot(coredata(edhec[,12]),coredata(edhec[,4]),
+       xlab="Short Selling",ylab="Equity Mkt Neutral")
```



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")
```

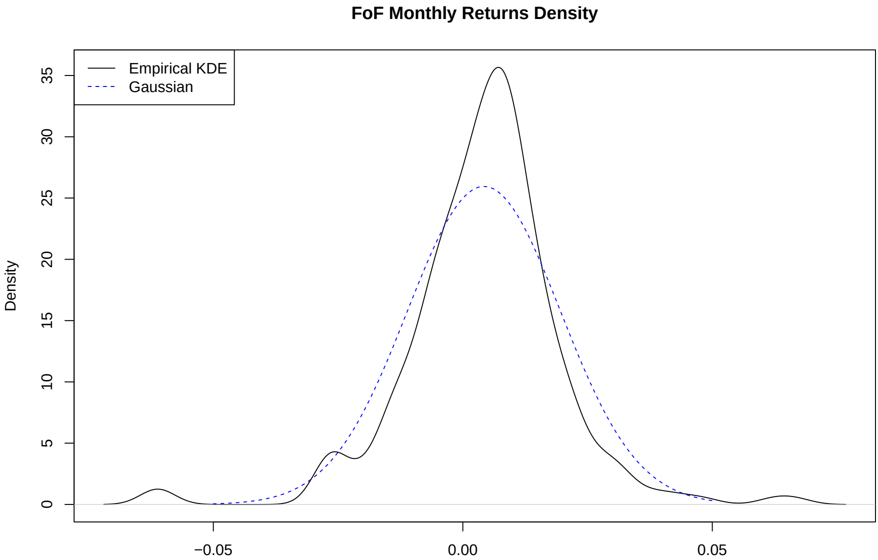


R Base - Kernel Density Estimates

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



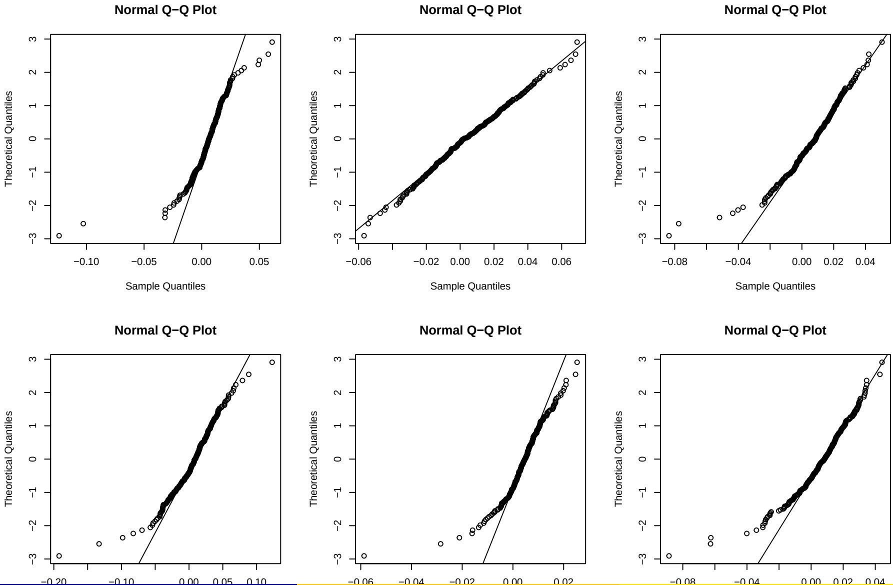
R Base - QQ Plots

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.

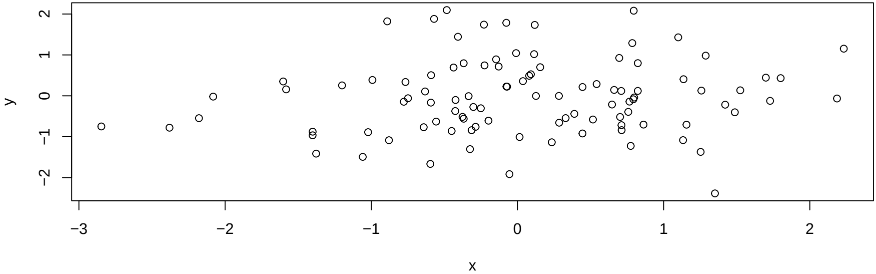
R Base - QQ Plots



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.

Section 10

Basic Exploratory Data Analysis

Time Series Objects

In finance, we usually work with *time series* data - data that has a specific order or time/date component. R has a variety of time series objects available to users:

- ts
- timeseries
- zoo
- xts

These are objects and associated functions that make working with ordered data (such as financial time series) much more convenient. *xts* is one that we will use extensively in this course, and it has many functions for cleaning and manipulating data sets where order is important and dates are used to determine order.

xts Object Example

xts stands for Extensible Time Series - this is an extension of the zoo package.

```
> library(xts)
```

We'll look at an example from the xts document available at
https://cran.r-project.org/web/packages/xts/vignettes/xts.pdf

```
> data(sample_matrix)
```

xts Object Example

```
> head(sample_matrix)
```

```
##                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
```

xts Object Example

```
> 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-
##   ..$ : chr [1:4] "Open" "High" "Low" "Close"
```

str() returns a list showing the internal structure of an R object - in this case, we see that the sample_matrix xts object has a 180x4 matrix of numeric data as well as an dimnames attribute, which is a list of 2 (the row names or dates and the column names)

xts Object Example

We won't worry too much about the power of xts so quickly, but you should be aware that having properly formatted time series data in R can often make your work much easier. Consider the problem of extracting the month-end data points from the daily data of the sample_matrix object:

```
> sample.monthly <- apply.monthly(sample_matrix,tail,1)
> head(sample.monthly)
```

```
##                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
```

xts Object Example

Assuming we have proper time/date stamps, we can coerce other data types into xts objects. Let's say we had a matrix of prices in a csv file called "ts.csv" with the associated dates in the first column:

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

```
## [1] "data.frame"
```

```
> head(ts, 2)
```

```
##         Date  Open  High   Low Close Adj.Close Volume
## 1 2018-07-20 40.49 40.58 40.22 40.38     40.38 146400
## 2 2018-07-23 40.39 40.39 39.82 40.27     40.27 114100
```

xts Object Example

We can now convert this data frame into an xts object using the xts() function:

```
> ts <- read.table("data/ts.csv",header=TRUE,sep=",",as.is=TRUE)
> ts.xts <- xts(ts[,-1],order.by=as.Date(ts[,1],
+                                       format="%Y-%m-%d"))
> class(ts.xts)
```

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

```
> head(ts.xts,2)
```

```
##             Open  High   Low Close Adj.Close Volume
## 2018-07-20 40.49 40.58 40.22 40.38     40.38 146400
## 2018-07-23 40.39 40.39 39.82 40.27     40.27 114100
```

Calculating Returns

Linear Returns:

$$L_t = \frac{P_t}{P_{t-1}} - 1$$

If $\omega_1, ..., \omega_n$ are n portfolio weights of the securities in portfolio P, then

$$L_{t,P} = \omega_1 L_{t,1} + ... + \omega_n L_{t,n}$$

But:

$$\frac{P_{t+1}}{P_{t-1}} - 1 \neq L_t + L_{t+1}$$

Calculating Returns

Compounded Returns:

$$C_t = \ln \frac{P_t}{P_{t-1}}$$

If $\omega_1, ..., \omega_n$ are n portfolio weights of the securities in portfolio P, then

$$C_{t,P} \neq \omega_1 C_{t,1} + ... + \omega_n C_{t,n}$$

But:

$$\ln \frac{P_{t+1}}{P_{t-1}} = C_t + C_{t+1}$$

Calculating Returns in R

Linear Returns:

```
> ts.ret.lin <- sample_matrix[-1,]/sample_matrix[-nrow(sample_matrix
```

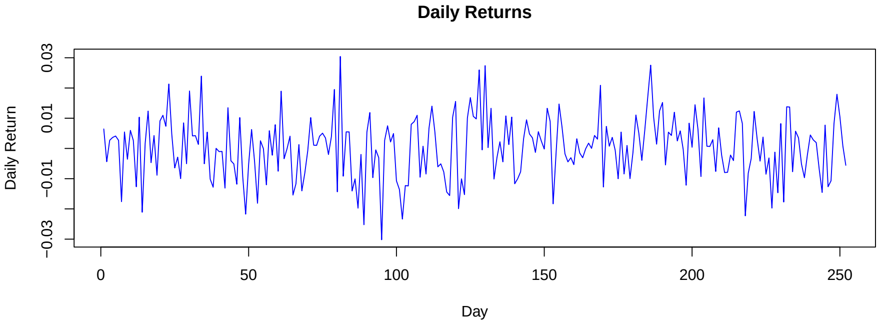
Log Returns:

```
> ts.ret.log <- diff(log(sample_matrix))
> head(ts.ret.log,2)
```

```
##                   Open          High          Low        Close
## 2007-01-03 0.003804009  6.049348e-03 0.0055915300  0.005569091
## 2007-01-04 0.003784530 -1.826194e-05 0.0006694959 -0.001296719
```

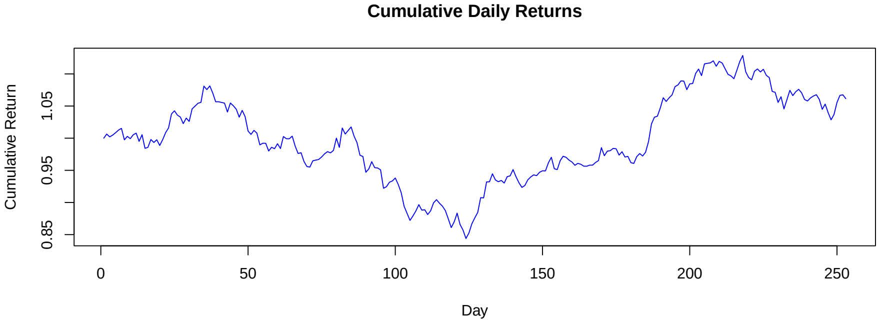
Recovering Prices from Returns

```
> my.returns <- rnorm(252,mean=0.1/252,sd=.16/sqrt(252))
> plot(my.returns,type="l",col="blue",ylab="Daily Return",
+      xlab="Day",main="Daily Returns")
```



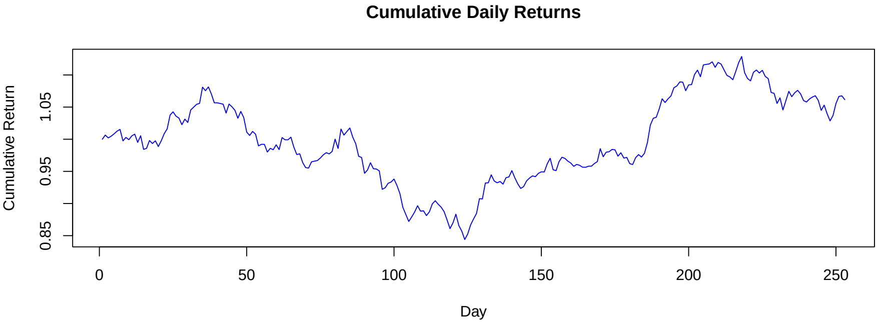
Recovering Prices from Returns

```
> cumulative.returns <- cumprod(1+my.returns)
> plot(c(1,cumulative.returns),type="l",col="blue",xlab="Day",
+       ylab="Cumulative Return",
+       main="Cumulative Daily Returns")
```



Working with Logarithmic Returns

```
> log.returns <- log(1+my.returns)
> cumulative.returns = cumsum(log.returns)
> plot(c(1,exp(cumulative.returns)),type="l",col="blue",xlab="Day",
+       ylab="Cumulative Return",
+       main="Cumulative Daily Returns")
```



Correlations

Let's investigate the correlations of various EDHEC Hedge Fund Indices using data available from the *PerformanceAnalytics* library:

```
> library(PerformanceAnalytics)
> data(edhec)
> names(edhec)
```

```
##  [1] "Convertible Arbitrage"  "CTA Global"             "Distresse
##  [4] "Emerging Markets"       "Equity Market Neutral"  "Event Dri
##  [7] "Fixed Income Arbitrage" "Global Macro"           "Long/Shor
## [10] "Merger Arbitrage"       "Relative Value"         "Short Sel
## [13] "Funds of Funds"
```

Correlations

```
> colnames(edhec) = c("CA","CTA","DS","EM","EMN","ED","FIA",
+                      "GM","LS","MA","RV","SS","FoF")
> class(edhec)
```

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

```
> dim(edhec)
```

```
## [1] 275  13
```

Correlations

Correlation matrix:

```
> cor(edhec)
```

```
##              CA          CTA          DS          EM       EMN
## CA   1.00000000 -0.020397460  0.73056360  0.59581596  0.4945214
## CTA -0.02039746  1.000000000 -0.02066553  0.04076115  0.1979645
## DS   0.73056360 -0.020665526  1.00000000  0.77939305  0.5905784
## EM   0.59581596  0.040761152  0.77939305  1.00000000  0.5148509
## EMN  0.49452139  0.197964540  0.59057839  0.51485086  1.0000000
## ED   0.72719021  0.012591155  0.92272943  0.82251064  0.6233498
## FIA  0.77812105  0.009559439  0.65919584  0.53515775  0.4091773
## GM   0.39842200  0.567719030  0.53364027  0.65577006  0.5602918
## LS   0.60221113  0.104484319  0.77027731  0.80929883  0.6592900
## MA   0.55391291  0.032051520  0.63369670  0.60931248  0.5322269
## RV   0.85007159  0.025297119  0.84793072  0.77635211  0.6360594
## SS  -0.35079353  0.113935995 -0.56542087 -0.65724377 -0.3106873 -
## FoF  0.64603284  0.192717727  0.81442193  0.84885755  0.6886186
##               FIA           GM          LS          MA          RV
```

Correlations

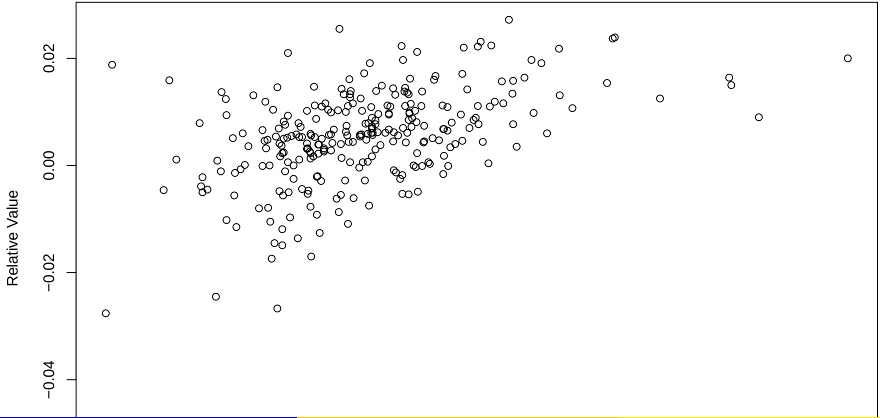
First 4 rows & first 4 columns of the correlation matrix:

```
> cor(edhec)[1:4,1:4]
```

```
##              CA         CTA          DS         EM
## CA   1.00000000 -0.02039746  0.73056360 0.59581596
## CTA -0.02039746  1.00000000 -0.02066553 0.04076115
## DS   0.73056360 -0.02066553  1.00000000 0.77939305
## EM   0.59581596  0.04076115  0.77939305 1.00000000
```

Plot Two Strategies

```
> plot(coredata(edhec[,8]),coredata(edhec[,10]),
+       xlab="Long/Short Equity",ylab="Relative Value")
```



Section 11

Basic Statistics

Summarizing Data

```
> summary(coredata(edhec))
```

```
##        CA                CTA                  DS
##  Min.   :-0.12370   Min.   :-0.056800   Min.   :-0.083600   Min.
##  1st Qu.:-0.00005   1st Qu.:-0.011900   1st Qu.:-0.002150   1st Q
##  Median : 0.00640   Median : 0.002000   Median : 0.008600   Media
##  Mean   : 0.00550   Mean   : 0.004158   Mean   : 0.006622   Mean
##  3rd Qu.: 0.01340   3rd Qu.: 0.020250   3rd Qu.: 0.017500   3rd Q
##  Max.   : 0.06110   Max.   : 0.069100   Max.   : 0.050400   Max.
##       EMN                  ED                 FIA
##  Min.   :-0.058700   Min.   :-0.088600   Min.   :-0.086700
##  1st Qu.: 0.001050   1st Qu.:-0.001450   1st Qu.: 0.001550
##  Median : 0.004700   Median : 0.008300   Median : 0.005400
##  Mean   : 0.004356   Mean   : 0.006216   Mean   : 0.004267
##  3rd Qu.: 0.008100   3rd Qu.: 0.015900   3rd Qu.: 0.009250
##  Max.   : 0.025300   Max.   : 0.044200   Max.   : 0.036500
##        GM                  LS                 MA
##  Min.   :-0.031300   Min.   :-0.06750   Min.   :-0.054400   Min.
```

Summarizing Data

```
> summary(coredata(edhec[,1:3]))
```

```
##          CA                    CTA                    DS
##  Min.   :-0.12370   Min.   :-0.056800   Min.   :-0.083600
##  1st Qu.:-0.00005   1st Qu.:-0.011900   1st Qu.:-0.002150
##  Median : 0.00640   Median : 0.002000   Median : 0.008600
##  Mean   : 0.00550   Mean   : 0.004158   Mean   : 0.006622
##  3rd Qu.: 0.01340   3rd Qu.: 0.020250   3rd Qu.: 0.017500
##  Max.   : 0.06110   Max.   : 0.069100   Max.   : 0.050400
```

Basic Regression

In R, regression is easily done with the lm() function.

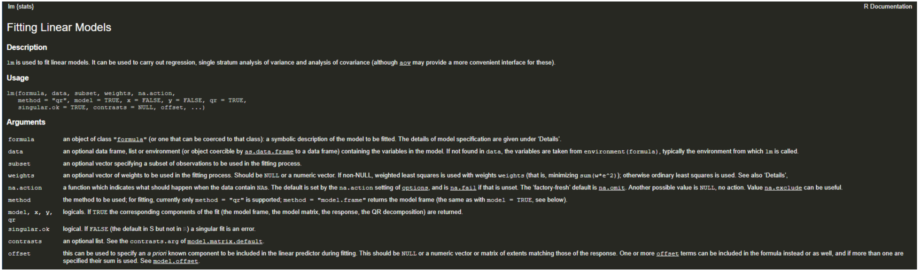
```
> args(lm)
```

function (formula, data, subset, weights, na.action, method = “qr”, model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, . . .) NULL

In its simplest form, this will look like:

my.regression <- lm(dep.var ~ indep.var)

Basic Regression



lm {stats} R Documentation

Fitting Linear Models

Description

`lm` is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although `aov` may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

`formula` an object of class `"formula"` (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under ‘Details’.

`data` an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `lm` is called.

`subset` an optional vector specifying a subset of observations to be used in the fitting process.

`weights` an optional vector of weights to be used in the fitting process. Should be `NULL` or a numeric vector. If non-NULL, weighted least squares is used with weights `weights` (that is, minimizing `sum(w*e^2)`); otherwise ordinary least squares is used. See also ‘Details’,

`na.action` a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The ‘factory-fresh’ default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.

`method` the method to be used; for fitting, currently only `method = "qr"` is supported; `method = "model.frame"` returns the model frame (the same as with `model = TRUE`, see below).

`model, x, y, qr` logicals. If `TRUE` the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.

`singular.ok` logical. If `FALSE` (the default in S but not in R) a singular fit is an error.

`contrasts` an optional list. See the `contrasts.arg` of `model.matrix.default`.

`offset` this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be `NULL` or a numeric vector or matrix of extents matching those of the response. One or more `offset` terms can be included in the formula instead or as well, and if more than one are specified their sum is used. See `model.offset`.

Figure 9: lm()

Basic Regression

Lets estimate the equity market betas of the EDHEC hedge fund indices over the sample period of the dataset. First, download the history of the S&P 500 from Dec 31 1996 to August 31 2008, store as an xts object, and convert to monthly data:

```
> library(quantmod)
> library(xts)
> getSymbols("^GSPC",src="yahoo",
+            from="1996-12-31",to="2009-08-31")
```

```
## [1] "^GSPC"
```

```
> spx.dat = apply.monthly(GSPC[,6],tail,1)
> spx.ret = (exp(diff(log(spx.dat)))-1)[-1,]
> my.df = cbind(coredata(spx.ret), coredata(edhec[1:152,]))
> my.data.xts = xts(my.df,order.by=as.Date(index(spx.ret)))
```

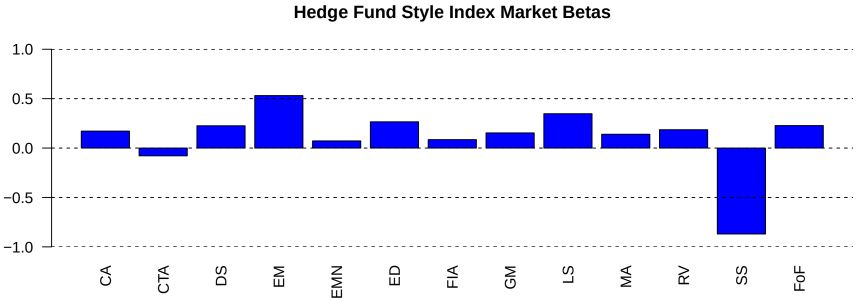
Basic Regression

Regress each EDHEC time series on the S&P 500 returns to estimate the market beta of each hedge fund style over the 1996-2009 period:

```
> betas <- rep(0,ncol(edhec))
> for (i in 1:ncol(edhec)) {
+   betas[i] = lm(my.data.xts[,(i+1)] ~
+                    my.data.xts[,1])$coef[[2]]
+ }
```

Basic Regression

```
> barplot(betas, ylim=c(-1,1),col="blue",
+          names=c("CA","CTA","DS","EM","EMN","ED","FIA",
+                  "GM","LS","MA","RV","SS","FoF"),las=2,
+          main="Hedge Fund Style Index Market Betas")
> abline(h=c(-1,-.5,0,.5,1),lty=2)
```


Hedge Fund Style Index Market Betas
CA
CTA
DS
EM
EMN
ED
FIA
GM
LS
MA
RV
SS
FoF