raw - R Actuarial Workshops

Brian A. Fannin, ACAS

2016-07-30

Contents

Introduction

> "R isn't software. It's a community." — John Chambers

Hello! Very happy to have you here and I hope you find this useful. This book is meant to serve as a companion to any of the R training sessions that I'm involved in. A few quick notes before we proceed.

Why does this book exist?

For over three years now, I've joined other actuaries in teaching R at events sponsored by the Casualty Actuarial Society[1]. I've learned a lot about what questions get asked, where folks get stuck and what content matters most. We've reached the place where, to be honest, attendees can get more out of the live sessions if they come in having done some preliminary work. That should give us more opportunity for hands on instruction. At a minimum, this book should serve as a handy reference before, during and after a live training to reinforce what we're trying to teach.

That understood, this book is hardly the only game in town. There are loads of good books about R and I can easily recommend many of them. [**?**] is a great one. Go check them out. I have.

This is an organic book

You'll not find this in Barnes & Nobel or Amazon and I'll strongly suggest that you resist the temptation to print this. I fully expect that there will corrections, additions and updates as the technology changes. The book will live on the internet as long as I can support it and it's probably best to check it out there. By all means, download the PDF if you'd like a local copy, but do check back for updates.

You don't need to read this from start to finish

Though I've done my best to give this book a clear flow, I've had to make a few sacrifices in order to get in all the material that I needed. This means that there are some, sorry, boring bits like a page about data types or how to write loops and such. Some people will find this stuff fascinating, some people will find this … necessary. Feel free to treat this like a reference text and not like a Michael Chabon novel and you're likely to get more out of it.

So, that's all the preliminaries. Away we go!

[1] The CAS has not sponsored this publication and no one should construe my involvement with the CAS as constituting their endorsement of the material presented here.

Part I

First Steps

Chapter 1

Setup

By the end of this chapter, you will have done the following:

- Install R
- Install RStudio

1.1 Operating systems

Although R was developed primarily on Linux, it is used on many different systems. This book may be read by users of any of the three major operating systems, that is: Windows, Mac OS and Linux. In order to be concise, I will only refer to the CTRL key. Mac OS users will understand that this key is CMD on their keyboards.

R may be used on any of the popular operating systems available today. I've used R on Windows, Linux and a Mac and the experience is pretty much the same everywhere. This is one of the fantastic features of the software. In each case, what you'll do is download a file from the internet and then follow the standard process you go through to install software on whichever system you're using. For the most part, installation is quick and painless, but there may be limitations placed on you by your IT department. I have a few suggestions which I hope can help overcome any difficulties you might experience.

The version of R and system architecture used to write this book are noted below:

```
## R version 3.3.1 (2016-06-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 14.04.4 LTS
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  base
##
## other attached packages:
## [1] pander_0.6.0
##
```

```
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.6     bookdown_0.1    digest_0.6.9    mime_0.5
##  [5] R6_2.1.2        xtable_1.8-2    formatR_1.4     magrittr_1.5
##  [9] evaluate_0.9    stringi_1.1.1   miniUI_0.1.1    rstudioapi_0.6
## [13] rmarkdown_1.0   tools_3.3.1     stringr_1.0.0   shiny_0.13.2
## [17] yaml_2.1.13     httpuv_1.3.3    htmltools_0.3.5 knitr_1.13
## [21] methods_3.3.1
```

1.1.1 Installing R

The first place to look for installation is [cran.r-project.org]. From there, you will see links to downloads for Windows, Mac and Linux. Clicking on the appropriate link will take you to the page that's relevant for your operating system. You may see lots of bizarre, arcane language around binaries, source and tarballs. If those words (in this context) mean nothing to you, don't panic. Some folk like to build their own version of R directly from the source code. If you're reading these instructions, you're probably not one of those people.

I recommend getting familiar with the CRAN website and reading the documentation there. If you get totally lost, try the links below which should take you directly to the download site for Windows and Mac. (If you're running Linux, I can't imagine you need my help.)

- Windows install
- Mac install

It's possible that you'll be asked to identify a "mirror". R is hosted on a number of servers throughout the world. It's all the same R, but distributing it in this way helps to minimized load on servers which host the files.

1.1.2 Installing R Studio

Installing R is most of the battle. Depending on the sort of person you are, it may even be all of the battle (see the following section on environments). R comes with a fairly spartan user interface, which is sufficient to get work done. However, most folk find that they enjoy using an Integrated Development Environment (IDE). This allows one to work on several source files at the same time, read help, observe console output, see what variables are loaded in memory, etc. There are a few options, but I've not yet found anything better than RStudio.

RStudio's main website may be found at www.rstudio.com. At the time of writing, the download page may be found at http://www.rstudio.com/products/rstudio/download/. Here you will find links to specific systems. The browser will even attempt to detect what operating system you're using and suggest a link for you. Cool, huh?

1.1.3 IT

I don't think I've ever met anyone who's made it through a white-collar existence without at least one or two frustrating exchanges with a corporate IT department. If you work for a large- or even small- company, you likely have a staff of folks who keep the network running and handle software requests from every user in the company. To ensure that your company's network is free from malicious attack or well-intentioned, but careless or imperfect users, most computers have sensitive areas restricted. This means that if you want to install software, you need an administrator to do it for you.

What this also means is that your IT department might not be as delighted as you are to install open-source software on the company laptop. This might be a problem that they're not inclined to solve for you and you may find your interaction with IT folks to a bit frustrating and they may seem as though they're not at all helpful.

The first thing to bear in mind is that, despite any appearance to the contrary, your IT staff is there to help you. Moreover, they're people. They have families who love them, possibly small children who think their moms and dads are awesome, pets who miss them and lives outside of work. They have to deal with ridiculous hours to accommodate you and they get far more complaints than they do praise. Be nice to them and you may be surprised how supportive they can be.

With that understood, there are several situations you may find yourself in.:

- You have- or can talk your way into- admin rights to your computer

Lucky you. Also lucky me as this is the happy situation that I enjoy. How do you handle this situation? Don't blow it! Be careful what you download, don't greedily consume bandwidth, server space or any of your company's other scarce resources and be VERY NICE to your IT staff. Acknowledge that you're grateful to have been given such trust and pledge not to do anything to have it removed.

- You don't have admin rights to your computer

What to do? Request to be given admin rights. Explain why, in detail and don't be evasive or vague. Trust and mutual respect help. Talk to other folks in your department and get them on board. Present a strong business case for why use of this software will permit you and your department to work more efficiently. Show them this book and underline the parts where I tell folk to be nice and respectful towards IT staff.

- IT won't give you admin rights to your computer

In this case, you may ask them to install it for you. Pool your resources. Talk to other actuaries and analysts in your company. Talk to your boss.

- IT won't install the software.

Solution? Install the software to a memory stick. Yes, it is often (but not always!) possible to do this. This is is obviously not a preferred option, but it will get you up and running and enable you to attend the workshop.

- Memory sticks are locked down

In this case, your IT department really wants you all to be running terminals. OK. Suggest an install of RStudio Server. This enables R to run on a server with controlled user access. This is quite a lot more work for your IT staff and you'll need to make a strong use case for it. If you're a large organization which has a predictive modelling or analytics area, they'll likely want this software. This won't allow you to use R remotely, so getting the most out of the workshop will be tough. However, there's still one more option.

- The nuclear option

Your IT staff won't run R on a server, won't give you a laptop with R installed. They're really against this software. I'd like to advise you to get another job, but that's defeatist. This is where we reach the nuclear option, which is to use your own computer. This will drive folks at your company nuts. Now you're transferring data from a secure machine to one which you use for personal e-mail, Facebook, sports, personal finance and other activities that we needn't dwell on here. This is an absolute last resort and the overheard of moving stuff from one device to another will obviate most of the efficiency gains that open source software will provide. Here's how to make it work: produce work that is ONLY POSSIBLE using R, or Python or any of the tools which we will discuss. Show a killer visual and then patiently explain to your boss why it can't be done in Excel and why you can't share it with other departments and why it can't be done every quarter. This is a tall order, but it just might get someone's attention.

Chapter 2

Getting started

> "R isn't software. It's a community." — John Chambers

This chapter will give you a short tour through R.

- Enter a few basic commands

2.1 The Operating Environment

Right. So, you've got R installed. Now what? Among the first differences you'll encounter relative to Excel is that you now have several different options when it comes to using R. R is an engine designed to process R commands. Where you store those commands and how you deal with that output is something over which you have a great deal of control. Terrible, frighening control. Here are those options in a nutshell:

- Command-line interface (CLI)
- RGui
- RStudio
- Others

2.1.1 Command-line interface

R, like S before it, presumed that users would interact with the program from the command line. And, if you invoke the R command from a terminal, that's exactly what you'll get. The image below is from my

Throughout this book, I will assume that you're using RStudio. You don't have to, but I will strongly recommend it. Why?

- Things are easier with RStudio

RStudio, keeps track of all the variables in memory

- Everyone else is using it.

OK, not much of an argument. This is the exact opposite of the logic our parents used to try and discourage us from smoking. However, in this case, it makes sense. When you're talking with other people and trying to reproduce your problem or share your awesome code, they're probably using RStudio. Using the same tool reduces the amount of effort needed to communicate.

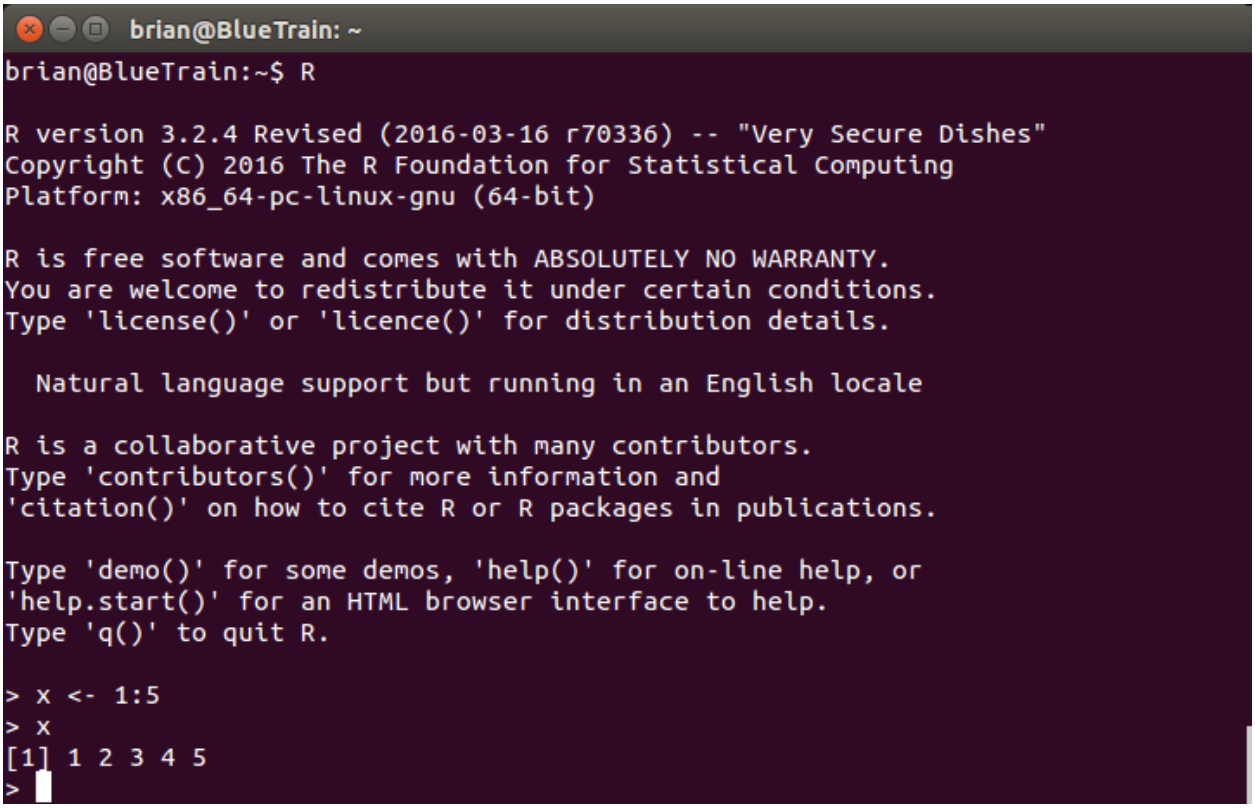


Figure 2.1: R at the command-line

2.2 Entering Commands

Now that you've got an is environment, you're ready to go. That cursor is blinking and waiting for you to tell it what to do! So what's the first thing you'll accomplish?

Well, not much. We'll get into more fun stuff in the next chapter, but for now let's play it safe. You can use R a basic calculator, so take a few minutes to enter some basic mathematical expressions.

```
1 + 1
```

```
## [1] 2
```

```
pi
```

```
## [1] 3.141593
```

```
2*pi*4^2
```

```
## [1] 100.531
```

- I can't find the console

In RStudio, the console may be reached by pressing CTRL-2 (Command-2 on Mac).

2.3 Getting help

```
?plot

??cluster
```

Within RStudio, the TAB key will autocomplete

2.4 The working directory

The source of much frustration when starting out.

Where am I?

```
getwd()
```

```
## [1] "/home/brian/Projects/books/raw"
```

How do I get somewhere else?

```
setwd("~/SomeNewDirectory/SomeSubfolder")
```

Try to stick with relative pathnames. This makes work portable.

2.4.1 Directory paths

R prefers *nix style directories, i.e. "/", NOT "\". Windows prefers "\".

"\" is an "escape" character, used for things like tabs and newline characters. To get a single slash, just type it twice.

More on file operations in the handout.

2.4.2 Source files

Typing, editing and debugging at the command line will get tedious quickly.

A source file (file extension .R) contains a sequence of commands.

Analogous to the formulae entered in a spreadsheet (but so much more powerful!)

2.5 Your first script

```
N <- 100
B0 <- 5
B1 <- 1.5

set.seed(1234)

e <- rnorm(N, mean = 0, sd = 1)
X1 <- rep(seq(1,10),10)

Y <- B0 + B1 * X1 + e

myFit <- lm(Y ~ X1)
```

Save this file.

CTRL-S on Windows/Linux, CMD-S on Mac.

2.5.1 Executing a script

Either:

1. Open the file and execute the lines one at a time, or
2. Use the "source" function.

```
source("SomefileName.R")
```

Within RStudio, you may also click the "Source" button in the upper right hand corner.

Chapter 3

Elements of the Language

There are certain concepts common to virtually all programming languages. Those elements are: variables, functions and operators. This chapter will discuss what those are and how they're implemented in R. By the end of this chapter, you will be able to answer the following:

- What is a variable and how do I create and modify them?
- How do functions work?
-

If you're familiar with other languages like Visual Basic, Python or Java Script, you may be tempted to skip this section. If you do, you'll survive, but I'd suggest giving it a quick read. You may learn something about how R differs from those other languages.

3.1 Variables

Programming languages work by assigning values to space in you computer's memory. Those values are then available for computation. Because the value of what's stored in memory may change, we call these things "variables". Think of a cell in a spreadsheet. Before we put something in it, it's just an empty box. We can fill it with whatever we like, be it a person's name, their birthdate, their age, whatever.

3.1.1 Assignment

Assignment will create a variable which contains a value. This value may be used later.

```
r <- 4

r + 2
```

```
## [1] 6
```

Both "<-" and "=" will work for assignment.

3.1.2 Data types

To a human, the difference between something numeric- like a person's age- and something textual - like their name - isn't a big deal. To a computer, however, this matters a lot. In order to ensure that there is sufficient memory to store the information and to ensure that it may be used in an operation, the computer needs to know what type of data it's working with. In other words: 5 + "Steve" = Huh?

3.2 Operators

3.2.1 Mathematical Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

3.2.2 Logical Operators

Operator	Operation
&	and
|	or
!	not
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
xor()	exclusive or
&&	non-vector and
||	non-vector or

3.3 Functions

Functions in R are very similar to functions in a spreadsheet. The function takes in arguments and returns a result.

```
sqrt(4)
```

```
## [1] 2
```

Functions may be composed by having the output of one serve as the input to another.

$\sqrt{e^{sin\pi}}$

```
sqrt(exp(sin(pi)))
```

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

3.3.1 A few mathematical functions

```
?S3groupGeneric
```

- abs, sign
- floor, ceiling, trunc, round, signif
- sqrt, exp, log

- cos, sin, tan (and many others)
- lgamma, gamma, digamma, trigamma

3.3.2 Comments

R uses the hash/pound character "#" to indicate comments.

SQL or C++ style multiline comments are not supported.

Comment early and often!

Comments should describe "why", not "what".

3.3.2.1 Bad comment

```
# Take the ratio of loss to premium to determine the loss ratio

lossRatio <- Losses / Premium
```

3.3.2.2 Good comment

```
# Because this is a retrospective view of
# profitability, these losses have been
# developed, but not trended to a future
# point in time
lossRatio <- Losses / Premium
```

3.4 Exercises

- What is the area of a cylinder with radius = e and height = pi?
- What arguments are listed for the "plot" function?
- Find the help file for a generalized linear model
- Create a script which calculates the area of a cylinder. From a new script, assign the value 4 to a variable and source the other file. Assign the value 8 to your variable and source again. What happened?

Part II

Data

Chapter 4

Vectors

In this chapter, we're going to learn about vectors, which are the key building blocks of R programming. By the end of this chapter, you will know:

- What is a vector?
- How are vectors created?
- What are data types and how can I tell what sort of data I'm working with?
- What is metadata?
- How can I summarize a vector?

4.1 What is a vector?

Enter a value at the console and hit enter. What do we see?

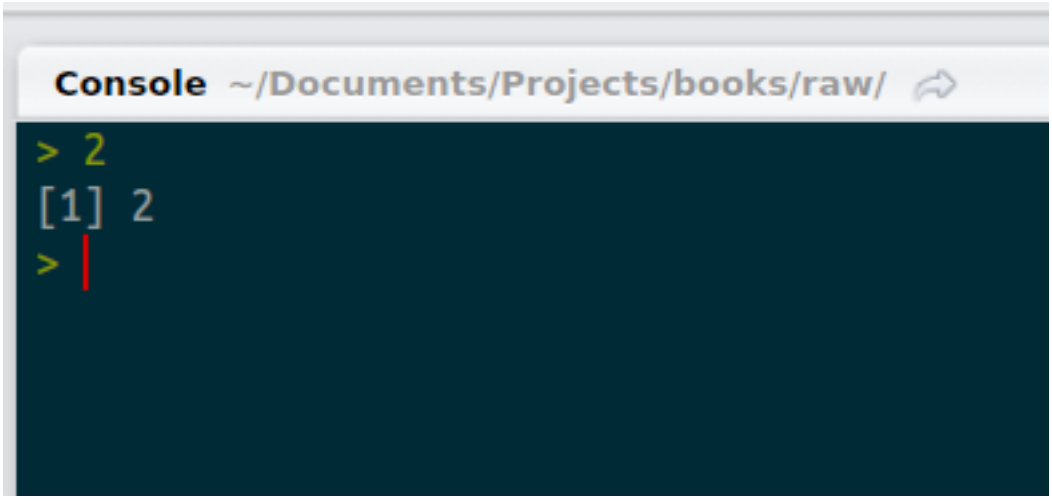


Figure 4.1: Console returning one value

This makes a bit of sense. We entered 2 and we got back 2. But what's that 1 in brackets? Things get even weirder when we ask R to return more than one value. Type "letters" (without the quotes) and have a look.

Now there's not only a 1 in brackets, there's also a 16 on the second line. (Note that your console may appear a bit different than mine.) You're probably clever enough to have figured out that the numbers in brackets have something to do with the number of outputs generated. In the second case, "p" is the 16th

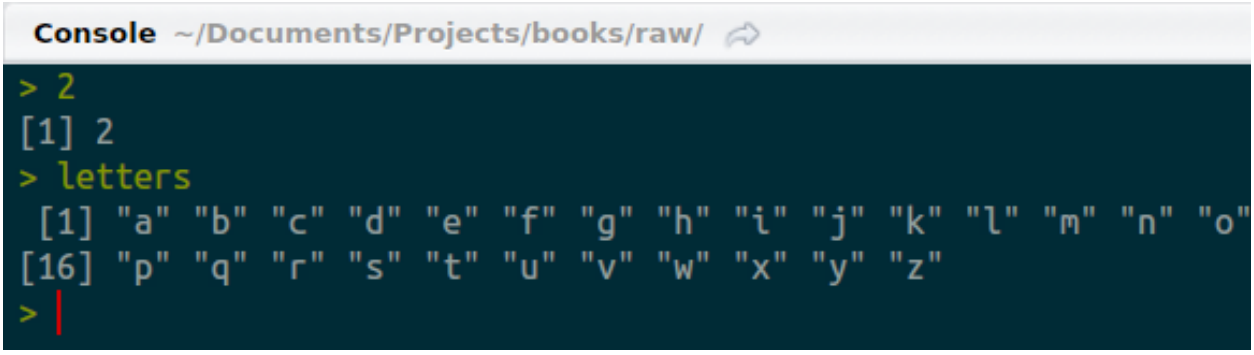


Figure 4.2: Console returning more than one value

letter of the alphabet and the bracketed 16 helps us know where we are in the sequence when it spills onto multiple lines. So, the bracketed figures are there to indicate how many numbers have been returned.

OK, cool. So what?

So everything! In R, every variable is a vector. Think back to When we entered the number 2 at the console, we were creating (briefly) a vector which had a length of 1. "letters" is a special vector with one element for each letter of the English alphabet. Vectors allow us to reason about a lot of data at once. The variable "letters" for instance enables us to store 26 values in one place. Further, it allows us to make changes to all of the elements of the vector at the same time. For example:

```
paste("Letter", letters)
```

```
##  [1] "Letter a" "Letter b" "Letter c" "Letter d" "Letter e" "Letter f"
##  [7] "Letter g" "Letter h" "Letter i" "Letter j" "Letter k" "Letter l"
## [13] "Letter m" "Letter n" "Letter o" "Letter p" "Letter q" "Letter r"
## [19] "Letter s" "Letter t" "Letter u" "Letter v" "Letter w" "Letter x"
## [25] "Letter y" "Letter z"
```

Using the `paste` command, we took each element of "letters" and prefixed it with the text "Letter". This is similar to applying the same function to a set of contiguous cells in a spreadsheet. But in this case, I didn't need to copy and paste something 26 times. I didn't even need to worry about how many times the command needed to be repeated. Vectors can grow and shrink automatically. No need to move cells around on a sheet. No need to copy formulas or change named ranges. R just did it. (Note that by default the `paste` function will automatically add a blank space between elements. The function `paste0` will concatenate elements without a space. Try it.)

4.1.1 Vector properties

All vectors share some basic properties

- Every element in a vector must be the same type.
  - R will change data types if they are not!
  - Different types are possible by using a list or a data frame (later)
- It's possible to add metadata (like names) via attributes
- Vectors have one dimension
- Higher dimensions are possible via matrices and arrays

As we'll see later, the issue of dimension is a bit arbitrary. At this point, the key thing to bear in mind is that *all of the data is of the same type*. Later on, we'll talk about the various data types that R supports. For now, it should be fairly clear from the context.

4.1.2 Vector construction

There's no real trick here. You'll be constructing vectors whether you want to be or not. But let's talk about a few core functions for vector construction and manipulation.

4.1.3 seq

`seq` is used often to generate a sequence of values. The colon operator `:` is a shortcut for a sequence of integers.

```
pies = seq(from = 0, by = pi, length.out = 5)
i <- 1:5
year = 2000:2004
```


4.1.4 rep

The `rep` function will replicate its input

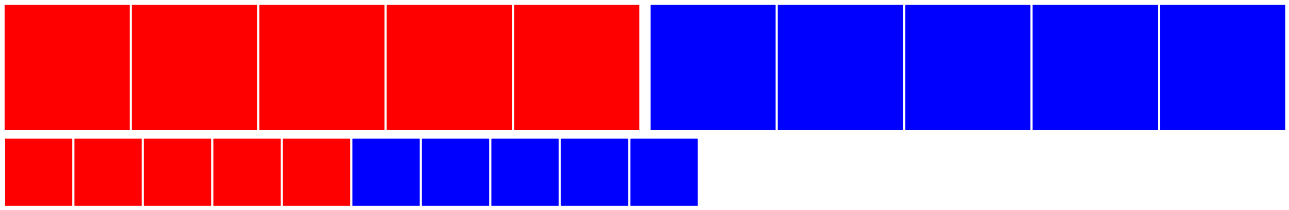
```
i = rep(pi, 100)
head(i)
```

```
## [1] 3.141593 3.141593 3.141593 3.141593 3.141593 3.141593
```

4.1.5 Concatenation

The `c()` function will concatenate values.

```
i <- c(1, 2, 3, 4, 5)
j <- c(6, 7, 8, 9, 10)
k <- c(i, j)
l <- c(1:5, 6:10)
```



4.1.6 Growth by assignment

Assigning a value beyond a vectors limits will automatically grow the vector. Interim values are assigned `NA`.

```
i <- 1:10
i[30] = pi
i
```

```
##  [1]  1.000000  2.000000  3.000000  4.000000  5.000000  6.000000  7.000000
##  [8]  8.000000  9.000000 10.000000        NA        NA        NA        NA
## [15]        NA        NA        NA        NA        NA        NA        NA
## [22]        NA        NA        NA        NA        NA        NA        NA
## [29]        NA  3.141593
```

4.1.7 Vector access - by index

Vectors may be accessed by their numeric indices. Remember, ':' is shorthand to generate a sequence.

```
set.seed(1234)
e <- rnorm(100)
e[1]
```

```
## [1] -1.207066
```

```
e[1:4]
```

```
## [1] -1.2070657  0.2774292  1.0844412 -2.3456977
```

```
e[c(1,3)]
```

```
## [1] -1.207066  1.084441
```

4.1.8 Vector access - logical access

Vectors may be accessed logically. This may be done by passing in a logical vector, or a logical expression.

```
i = 5:9
i[c(TRUE, FALSE, FALSE, FALSE, TRUE)]
```

```
## [1] 5 9
```

```
i[i > 7]
```

```
## [1] 8 9
```

```
b = i > 7
b
```

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

```
i[b]
```

```
## [1] 8 9
```

4.1.9 `which`

The `which` function returns indices that match a logical expression.

```
i <- 11:20
which(i > 12)
```

```
## [1]  3  4  5  6  7  8  9 10
```

```
i[which(i > 12)]
```

```
## [1] 13 14 15 16 17 18 19 20
```

4.1.10 sample

The `sample` function will generate a random sample. Great to use for randomizing a vector.

```
months <- c("January", "February", "March", "April"
            , "May", "June", "July", "August"
            , "September", "October", "November", "December")

set.seed(1234)
mixedMonths <- sample(months)
head(mixedMonths)
```

```
## [1] "February" "July"     "November" "June"     "October"  "May"
```

Get lots of months with the `size` parameter:

```
set.seed(1234)
lotsOfMonths <- sample(months, size = 100, replace = TRUE)
head(lotsOfMonths)
```

```
## [1] "February" "August"   "August"   "August"   "November" "August"
```

4.1.11 sample II

Sample may also be used within the indexing of the vector itself:

```
set.seed(1234)
moreMonths <- months[sample(1:12, replace=TRUE, size=100)]
head(moreMonths)
```

```
## [1] "February" "August"   "August"   "August"   "November" "August"
```

```
# Cleaner with sample.int
set.seed(1234)
evenMoreMonths <- months[sample.int(length(months), size=100, replace=TRUE)]
head(evenMoreMonths)
```

```
## [1] "February" "August"   "August"   "August"   "November" "August"
```

4.1.12 order

The function `order` will return the indices of the vector in order.

```
set.seed(1234)
x <- sample(1:10)
x
```

```
##  [1]  2  6  5  8  9  4  1  7 10  3
```

```
order(x)
```

```
##  [1]  7  1 10  6  3  2  8  4  5  9
```

```
x[order(x)]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

4.1.13 Vector arithmetic

Vectors may be used in arithmetic operations.

```
B0 <- 5
B1 <- 1.5

set.seed(1234)

e <- rnorm(N, mean = 0, sd = 1)
X1 <- rep(seq(1,10),10)

Y <- B0 + B1 * X1 + e
```

Y is now a vector with length equal to the longest vector used in the calculation.

Question: B0 and B1 are vectors of length 1.

X1 and e are vectors of length 100.

How are they combined?

4.1.14 Recycling

R will "recycle" vectors until there are enough elements to perform an operation. Everything gets as "long" as the longest vector in the operation. For scalar operations on a vector this doesn't involve any drama. Try the following code:

```
vector1 = 1:10
vector2 = 1:5
scalar = 3

print(vector1 + scalar)
```

```
##  [1]  4  5  6  7  8  9 10 11 12 13
```

```
print(vector2 + scalar)
```

```
## [1] 4 5 6 7 8
```

```
print(vector1 + vector2)
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15
```

4.1.15 Set theory - Part I

The `%in%` operator will return a logical vector indicating whether or not an element of the first set is contained in the second set.

```
x <- 1:10
y <- 5:15
x %in% y
```

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

4.1.16 Set theory - Part II

- `union`

- intersect
- setdiff
- setequal
- is.element

```
?union
```

```
x <- 1900:1910
y <- 1905:1915
intersect(x, y)
```

```
## [1] 1905 1906 1907 1908 1909 1910
```

```
setdiff(x, y)
```

```
## [1] 1900 1901 1902 1903 1904
```

```
setequal(x, y)
```

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

```
is.element(1941, y)
```

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

4.1.17 Summarization

Loads of functions take vector input and return scalar output. Translation of a large set of numbers into a few, informative values is one of the cornerstones of statistics.

```
x = 1:50
sum(x)
mean(x)
max(x)
length(x)
var(x)
```

4.1.18 Vectors

Vectors are like atoms. If you understand vectors- how to create them, how to manipulate them, how to access the elements, you're well on your way to grasping how to handle other objects in R.

Vectors may combine to form molecules, but fundamentally, *everything* in R is a vector.

4.2 From data

- Data types
- From vectors to matrices and lists

4.2.1 Data types

- logical
- integer
- double
- character

4.2.2 What is it?

```
x <- 6
y <- 6L
z <- TRUE
typeof(x)
```

```
## [1] "double"
```

```
typeof(y)
```

```
## [1] "integer"
```

```
typeof(z)
```

```
## [1] "logical"
```

```
is.logical(x)
```

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

```
is.double(x)
```

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

4.2.3 Data conversion

Most conversion is implicit. For explicit conversion, use the `as.*` functions.

Implicit conversion alters everything to the most complex form of data present as follows:

logical -> integer -> double -> character

Explicit conversion usually implies truncation and loss of information.

```
# Implicit conversion
w <- TRUE
x <- 4L
y <- 5.8
z <- w + x + y
typeof(z)
```

```
## [1] "double"
```

```
# Explicit conversion. Note loss of data.
as.integer(z)
```

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

4.2.4 Class

A class is an extension of the basic data types. We'll see many examples of these. The class of a basic type will be equal to its type apart from 'double', whose class is 'numeric' for reasons I don't pretend to understand.

```
class(TRUE)
```

```
## [1] "logical"
```

```
class(pi)
```

```
## [1] "numeric"
```

```
class(4L)
```

```
## [1] "integer"
```

The type and class of a vector is returned as a scalar. Remember a vector is a set of elements which all have the same type.

```
class(1:4)
```

```
## [1] "integer"
```

4.2.5 Mode

There is also a function called 'mode' which looks tempting. Ignore it.

4.2.6 Dates and times

Dates in R can be tricky. Two basic classes: `Date` and `POSIXt`. The `Date` class does not get more granular than days. The `POSIXt` class can handle seconds, milliseconds, etc.

My recommendation is to stick with the "Date" class. Introducing times means introducing time zones and possibility for confusion or error. Actuaries rarely need to measure things in minutes.

```
x <- as.Date('2010-01-01')
class(x)
```

```
## [1] "Date"
```

```
typeof(x)
```

```
## [1] "double"
```

4.2.7 More on dates

The default behavior for dates is that they don't follow US conventions.

Don't do this:

```
x <- as.Date('06-30-2010')
```

```
## Error in charToDate(x): character string is not in a standard unambiguous format
```

But this is just fine:

```
x <- as.Date('30-06-2010')
```

If you want to preserve your sanity, stick with year, month, day.

```
x <- as.Date('2010-06-30')
```

4.2.8 What day is it?

To get the date and time of the computer, use the either `Sys.Date()` or `Sys.time()`. Note that `Sys.time()` will return both the day AND the time as a POSIXct object.

```
x <- Sys.Date()
y <- Sys.time()
```

4.2.9 More reading on dates

Worth reading the documentation about dates. Measuring time periods is a common task for actuaries. It's easy to make huge mistakes by getting dates wrong.

The `lubridate` package has some nice convenience functions for setting month and day and reasoning about time periods. It also enables you to deal with time zones, leap days and leap seconds. Probably more than you need.

`mondate` was written by an actuary and supports (among other things) handling time periods in terms of months.

- Date class
- lubridate
- Ripley and Hornik
- mondate

4.2.10 Factors

Another gotcha. Factors were necessary many years ago when data collection and storage were expensive. A factor is a mapping of a character string to an integer. Particularly when importing data, R often wants to convert character values into a factor. You will often want to convert a factor into a string.

```
myColors <- c("Red", "Blue", "Green", "Red", "Blue", "Red")
myFactor <- factor(myColors)
typeof(myFactor)
```

```
## [1] "integer"
```

```
class(myFactor)
```

```
## [1] "factor"
```

```
is.character(myFactor)
```

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

```
is.character(myColors)
```

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

4.2.11 Altering factors

```
# This probably won't give you what you expect
myOtherFactor <- c(myFactor, "Orange")
myOtherFactor
```

```
## [1] "3"      "1"      "2"      "3"      "1"      "3"      "Orange"
```

```
# And this will give you an error
myFactor[length(myFactor)+1] <- "Orange"
```

```
## Warning in `[<-.factor`(`*tmp*`, length(myFactor) + 1, value = "Orange"):
## invalid factor level, NA generated
```

```
# Must do things in two steps
myOtherFactor <- factor(c(levels(myFactor), "Orange"))
myOtherFactor[length(myOtherFactor)+1] <- "Orange"
```

4.2.12 Avoid factors

Now that you know what they are, you can spend the next few months avoiding factors. When R was created, there were compelling reasons to include factors and they still have some utility. More often than not, though, they're a confusing hindrance.

If characters aren't behaving the way you expect them to, check the variables with `is.factor`. Convert them with `as.character` and you'll be back on the road to happiness.

4.3 Exercises

- Create a logical, integer, double and character variable.
- Can you create a vector with both logical and character values?
- What happens when you try to add a logical to an integer? An integer to a double?

4.3.1 Answers

```
myLogical <- TRUE
myInteger <- 1:4
myDouble <- 3.14
myCharacter <- "Hello!"

y <- myLogical + myInteger
typeof(y)
```

```
## [1] "integer"
```

```
y <- myInteger + myDouble
typeof(y)
```

```
## [1] "double"
```

4.3.2 From vectors to matrices and lists

A matrix is a vector with higher dimensions.

A list has both higher dimensions, but also different data types.

4.3.3 A matrix

Two ways to construct:

1. Use the `matrix` function.
2. Change the dimensions of a `vector`.

```
myVector <- 1:100
myMatrix <- matrix(myVector, nrow=10, ncol=10)

myOtherMatrix <- myVector
dim(myOtherMatrix) <- c(10,10)

identical(myMatrix, myOtherMatrix)
```

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

4.3.4

```
myMatrix <- matrix(nrow=10, ncol=10)
```


4.3.5

```
dim(myMatrix) <- c(25, 4)
```

4.3.6 Matrix metadata

Possible to add metadata. This is typically a name for the columns or rows.

```
myMatrix <- matrix(nrow=10, ncol=10, data = sample(1:100))
colnames(myMatrix) <- letters[1:10]
head(myMatrix, 3)
```

```
##       a  b  c  d  e  f  g  h  i  j
## [1,] 70 96 37 39  5 51 60 55 43 13
## [2,] 54 94 21 84 73  3 35 34 92 26
## [3,] 28 14 24 91 67 62 61 36 65 17
```

```
rownames(myMatrix) <- tail(letters, 10)
head(myMatrix, 3)
```

```
##    a  b  c  d  e  f  g  h  i  j
## q 70 96 37 39  5 51 60 55 43 13
## r 54 94 21 84 73  3 35 34 92 26
## s 28 14 24 91 67 62 61 36 65 17
```

4.3.7 Data access for a matrix

Matrix access is similar to vector, but with additional dimensions. For two-dimensional matrices, the order is row first, then column.

```
myMatrix[2, ]
```

```
##  a  b  c  d  e  f  g  h  i  j
## 54 94 21 84 73  3 35 34 92 26
```

```
myMatrix[, 2]
```

```
##  q  r  s  t  u  v  w  x  y  z
## 96 94 14  4 19 69 45 76 85 87
```

4.3.8 Data access continued

Single index will return values by indexing along only one dimension.

```
myMatrix[2]
```

```
## [1] 54
```

```
myMatrix[22]
```

```
## [1] 21
```

4.3.9 Matrix summary

```
sum(myMatrix)
```

```
## [1] 5050
```

```
colSums(myMatrix)
```

```
##   a   b   c   d   e   f   g   h   i   j
## 443 589 498 495 589 436 617 418 525 440
```

```
rowSums(myMatrix)
```

```
##   q   r   s   t   u   v   w   x   y   z
## 469 516 465 534 354 551 540 490 445 686
```

```
colMeans(myMatrix)
```

```
##    a    b    c    d    e    f    g    h    i    j
## 44.3 58.9 49.8 49.5 58.9 43.6 61.7 41.8 52.5 44.0
```

4.3.10 More than two dimensions

Like more than two dimensions? Shine on you crazy diamond.

4.4 Exercises

Create a vector of length 10, with years starting from 1980.

Create a vector with values from 1972 to 2012 in increments of four (1972, 1976, 1980, etc.)

Construct the following vectors (feel free to use the `VectorQuestion.R` script):

```
FirstName <- c("Richard", "James", "Ronald", "Ronald"
               , "George", "William", "William", "George"
               , "George", "Barack", "Barack")
LastName <- c("Nixon", "Carter", "Reagan", "Reagan"
              , "Bush", "Clinton", "Clinton", "Bush"
```

```
              , "Bush", "Obama", "Obama")
ElectionYear <- seq(1972, 2012, 4)
```

- List the last names in alphabetical order
- List the years in order by first name.
- Create a vector of years when someone named "George" was elected.
- How many Georges were elected before 1996?
- Generate a random sample of 100 presidents.

4.5 Answers

```
LastName[order(LastName)]
```

```
##  [1] "Bush"    "Bush"    "Bush"    "Carter"  "Clinton" "Clinton" "Nixon"
##  [8] "Obama"   "Obama"   "Reagan"  "Reagan"
```

```
ElectionYear[order(FirstName)]
```

```
##  [1] 2008 2012 1988 2000 2004 1976 1972 1980 1984 1992 1996
```

```
ElectionYear[FirstName == 'George']
```

```
## [1] 1988 2000 2004
```

```
myLogical <- (FirstName == 'George') & (ElectionYear < 1996)
length(which(myLogical))
```

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

```
sum(myLogical)
```

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

```
sample(LastName, 100, replace = TRUE)
```

```
##   [1] "Bush"    "Carter"  "Obama"   "Carter"  "Reagan"  "Obama"   "Obama"
##   [8] "Reagan"  "Carter"  "Bush"    "Bush"    "Obama"   "Obama"   "Obama"
##  [15] "Clinton" "Reagan"  "Reagan"  "Clinton" "Clinton" "Reagan"  "Obama"
##  [22] "Clinton" "Carter"  "Bush"    "Obama"   "Clinton" "Obama"   "Clinton"
##  [29] "Clinton" "Obama"   "Clinton" "Obama"   "Reagan"  "Clinton" "Reagan"
##  [36] "Clinton" "Bush"    "Clinton" "Obama"   "Clinton" "Bush"    "Reagan"
##  [43] "Nixon"   "Obama"   "Reagan"  "Obama"   "Clinton" "Obama"   "Bush"
##  [50] "Clinton" "Bush"    "Clinton" "Bush"    "Reagan"  "Nixon"   "Bush"
##  [57] "Bush"    "Nixon"   "Bush"    "Reagan"  "Bush"    "Clinton" "Bush"
##  [64] "Bush"    "Reagan"  "Bush"    "Bush"    "Clinton" "Carter"  "Reagan"
##  [71] "Clinton" "Reagan"  "Clinton" "Nixon"   "Obama"   "Nixon"   "Obama"
##  [78] "Clinton" "Reagan"  "Bush"    "Bush"    "Obama"   "Carter"  "Obama"
##  [85] "Bush"    "Obama"   "Obama"   "Bush"    "Obama"   "Bush"    "Bush"
##  [92] "Clinton" "Reagan"  "Bush"    "Clinton" "Bush"    "Reagan"  "Bush"
##  [99] "Reagan"  "Obama"
```

4.5.1 Next

Having sorted out vectors, we're next going to turn out attention to lists. Lists are similar to vectors in that they enable us to bundle large amounts of data in a single construct. However, they're far more flexible and require a bit more thought.

Chapter 5

Lists

In this chapter, we're going to learn about lists. Lists can be a bit confusing the first time you begin to use them. Heaven knows it took me ages to get comfortable with them. However, they're a very powerful way to structure data and, once mastered, will give you all kinds of control over pretty much anything the world can throw at you. If vectors are R's atom, lists are molecules.

By the end of this chapter, you will know:

- What is a list and how are they created?
- What is the difference between a list and vector?
- When and how do I use `lapply`?

Lists have data of arbitrary complexity. Any type, any length. Note the new `[[ ]]` double bracket operator.

```
x <- list()
typeof(x)
```

```
## [1] "list"
```

```
x[[1]] <- c("Hello", "there", "this", "is", "a", "list")
x[[2]] <- c(pi, exp(1))
summary(x)
```

```
##      Length Class  Mode
## [1,] 6      -none- character
## [2,] 2      -none- numeric
```

```
str(x)
```

```
## List of 2
##  $ : chr [1:6] "Hello" "there" "this" "is" ...
##  $ : num [1:2] 3.14 2.72
```

5.1 Lists Overview

5.1.1 [vs. [[

`[` is (almost always) used to set and return an element of the same type as the *containing* object.

`[[` is used to set and return an element of the same type as the *contained* object.

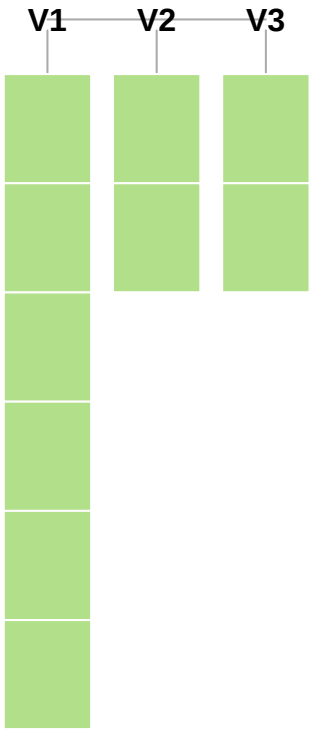
This is why we use `[[` to set an item in a list.

Don't worry if this doesn't make sense yet. It's difficult for most R programmers.

5.1.2 Recursive storage

Lists can contain other lists as elements.

```
y <- list()
y[[1]] <- "Lou Reed"
y[[2]] <- 45

x[[3]] <- y
```



5.1.3 List metadata

Again, typically names. However, these become very important for lists. Names are handled with the special $ operator. $ permits access to a single element. (A single element of a list can be a vector!)

```
y[[1]] <- c("Lou Reed", "Patti Smith")
y[[2]] <- c(45, 63)

names(y) <- c("Artist", "Age")

y$Artist
```

```
## [1] "Lou Reed"    "Patti Smith"
```

```
y$Age
```

```
## [1] 45 63
```

5.1.4 lapply

lapply is one of many functions which may be applied to lists. Can be difficult at first, but very powerful. Applies the same function to each element of a list.

```
myList <- list(firstVector = c(1:10)
               , secondVector = c(89, 56, 84, 298, 56)
               , thirdVector = c(7,3,5,6,2,4,2))
lapply(myList, mean)
```

```
## $firstVector
## [1] 5.5
##
```

```
## $secondVector
## [1] 116.6
##
## $thirdVector
## [1] 4.142857
```

```
lapply(myList, median)
```

```
## $firstVector
## [1] 5.5
##
## $secondVector
## [1] 84
##
## $thirdVector
## [1] 4
```

```
lapply(myList, sum)
```

```
## $firstVector
## [1] 55
##
## $secondVector
## [1] 583
##
## $thirdVector
## [1] 29
```

5.1.5 Why `lapply`?

Two reasons:

1. It's expressive. A loop is a lot of code which does little to clarify intent. `lapply` indicates that we want to apply the same function to each element of a list. Think of a formula that exists as a column in a spreadsheet.
2. It's easier to type at an interactive console. In its very early days, `S` was fully interactive. Typing a `for` loop at the console is a tedius and unnecessary task.

5.1.6 Summary functions

Because lists are arbitrary, we can't expect functions like `sum` or `mean` to work. Use `lapply` to summarize particular list elements.

5.2 Exercises

- Create a list with two elements. Have the first element be a vector with 100 numbers. Have the second element be a vector with 100 dates. Give your list the names: "Claim" and "AccidentDate".
- What is the average value of a claim?

5.3 Answers

```
myList <- list()
myList$Claims <- rlnorm(100, log(10000))
myList$AccidentDate <- sample(seq.Date(as.Date('2000-01-01'), as.Date('2009-12-31'), length.out = 1000)
mean(myList$Claims)
```

```
## [1] 18339.42
```

Chapter 6

Data Frames

Finally! This

By the end of this chapter you will know:

- What's a data frame and how do I create one?
- How do I read and write external data?

6.1 What's a data frame?

All of that about vectors and lists was prologue to this. The data frame is a seminal concept in R. Most statistical operations expect one and they are the most common way to pass data in and out of R.

Although critical to understand, this is very, very easy to get. What's a data frame? It's a table. That's it. No, really, that's it.

A data frame is a `list` of `vectors`. Each `vector` may have a different data type, but all must be the same length.

6.1.1 Creating a data frame

```
set.seed(1234)
State = rep(c("TX", "NY", "CA"), 10)
EarnedPremium = rlnorm(length(State), meanlog = log(50000), sdlog=1)
EarnedPremium = round(EarnedPremium, -3)
Losses = EarnedPremium * runif(length(EarnedPremium), min=0.4, max = 0.9)

df = data.frame(State, EarnedPremium, Losses, stringsAsFactors=FALSE)
```

6.1.2 Basic properties of a data frame

```
summary(df)
```

```
##     State           EarnedPremium        Losses
##  Length:30          Min.   :  5000   Min.   :  2034
##  Class :character   1st Qu.: 21250   1st Qu.: 12461
##  Mode  :character   Median : 30500   Median : 17363
```

```
##                     Mean   : 61500   Mean   : 39006
##                     3rd Qu.: 63750   3rd Qu.: 41465
##                     Max.   :560000   Max.   :411179
```

```
str(df)
```

```
## 'data.frame':    30 obs. of  3 variables:
##  $ State        : chr  "TX" "NY" "CA" "TX" ...
##  $ EarnedPremium: num  15000 66000 148000 5000 77000 83000 28000 29000 28000 21000 ...
##  $ Losses       : num  12486 27781 82671 2034 40002 ...
```

```
names(df)
```

```
## [1] "State"         "EarnedPremium" "Losses"
```

```
colnames(df)
```

```
## [1] "State"         "EarnedPremium" "Losses"
```

```
length(df)
```

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

```
dim(df)
```

```
## [1] 30  3
```

```
nrow(df)
```

```
## [1] 30
```

```
ncol(df)
```

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

```
head(df)
```

```
##   State EarnedPremium    Losses
## 1    TX         15000 12486.254
## 2    NY         66000 27781.290
## 3    CA        148000 82671.479
## 4    TX          5000  2034.375
## 5    NY         77000 40002.490
## 6    CA         83000 62519.527
```

```
head(df, 2)
```

```
##   State EarnedPremium   Losses
## 1    TX         15000 12486.25
## 2    NY         66000 27781.29
```

```
tail(df)
```

```
##    State EarnedPremium    Losses
## 25    TX         25000 12452.183
## 26    NY         12000 10191.483
## 27    CA         89000 52932.740
## 28    TX         18000  9997.837
## 29    NY         49000 23520.702
## 30    CA         20000 16961.858
```

6.1.3 Referencing

Very similar to referencing a 2D matrix.

```
df[2,3]
df[2]
df[2,]
df[2, -1]
```

Note the `$` operator to access named columns. A data frame uses the 'name' metadata in the same way as a list.

```
df$EarnedPremium
# Columns of a data frame may be treated as vectors
df$EarnedPremium[3]
df[2:4, 1:2]
df[, "EarnedPremium"]
df[, c("EarnedPremium", "State")]
```

6.1.4 Ordering

```
order(df$EarnedPremium)
```

```
##  [1]  4 26  1 12 28 18 30 10 19 13 25  7  9  8 17 11 22 23 16 29 14 21  2
## [24]  5 24  6 27 15  3 20
```

```
df = df[order(df$EarnedPremium), ]
```

6.1.5 Altering and adding columns

```
df$LossRatio = df$EarnedPremium / df$Losses
df$LossRatio = 1 / df$LossRatio
```

6.1.6 Eliminating columns

```
df$LossRatio = NULL
df = df[, 1:2]
```

6.1.7 rbind, cbind

`rbind` will append rows to the data frame. New rows must have the same number of columns and data types. `cbind` must have the same number of rows as the data frame.

```
dfA = df[1:10,]
dfB = df[11:20, ]
rbind(dfA, dfB)
dfC = dfA[, 1:2]
cbind(dfA, dfC)
```

6.1.8 Merging

```
dfRateChange = data.frame(State =c("TX", "CA", "NY"), RateChange = c(.05, -.1, .2))
df = merge(df, dfRateChange)
```

Merging is VLOOKUP on steroids. Basically equivalent to a JOIN in SQL.

6.1.9 Altering column names

```
df$LossRation = with(df, Losses / EarnedPremium)
names(df)
```

```
## [1] "State"         "EarnedPremium" "RateChange"    "LossRation"
```

```
colnames(df)[4] = "Loss Ratio"
colnames(df)
```

```
## [1] "State"         "EarnedPremium" "RateChange"    "Loss Ratio"
```

6.1.10 Subsetting - The easy way

```
dfTX = subset(df, State == "TX")
dfBigPolicies = subset(df, EarnedPremium >= 50000)
```

6.1.11 Subsetting - The hard(ish) way

```
dfTX = df[df$State == "TX", ]
dfBigPolicies = df[df$EarnedPremium >= 50000, ]
```

6.1.12 Subsetting - Yet another way

```
whichState = df$State == "TX"
dfTX = df[whichState, ]

whichEP = df$EarnedPremium >= 50000
dfBigPolicies = df[whichEP, ]
```

I use each of these three methods routinely. They're all good.

6.2 Summarizing

```
sum(df$EarnedPremium)
```

```
## [1] 1845000
```

```
sum(df$EarnedPremium[df$State == "TX"])
```

```
## [1] 233000
```

```
aggregate(df[,-1], list(df$State), sum)
```

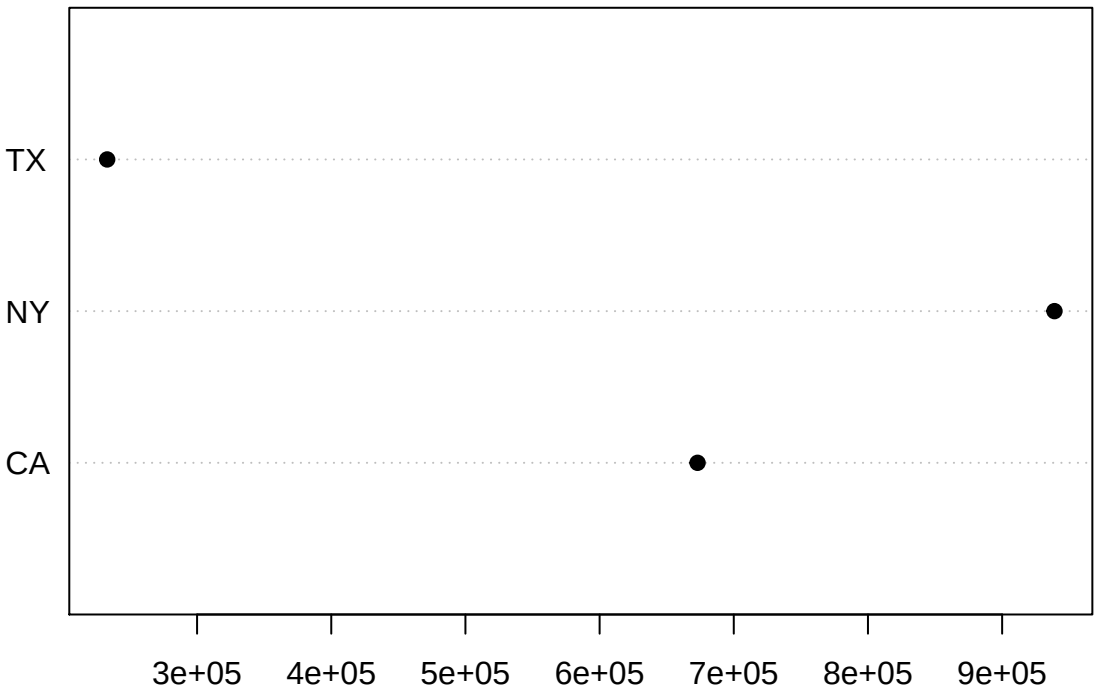
```
##   Group.1 EarnedPremium RateChange Loss Ratio
## 1      CA        673000       -1.0   8.257125
## 2      NY        939000        2.0  11.755127
## 3      TX        233000        0.5  18.745791
```

6.2.1 Summarizing visually - 1

```
dfByState = aggregate(df$EarnedPremium, list(df$State), sum)
colnames(dfByState) = c("State", "EarnedPremium")
barplot(dfByState$EarnedPremium, names.arg=dfByState$State, col="blue")
```

6.2.2 Summarizing visually - 2

```
dotchart(dfByState$EarnedPremium, dfByState$State, pch=19)
```



6.2.3 Advanced data frame tools

- dplyr
- tidyr
- reshape2
- data.table

Roughly 90% of your work in R will involve manipulation of data frames. There are truckloads of packages designed to make manipulation of data frames easier. Take your time getting to learn these tools. They're all powerful, but they're all a little different. I'd suggest learning the functions in `base` R first, then moving on to tools like `dplyr` and `data.table`. There's a lot to be gained from understanding the problems those packages were created to solve.

6.2.4 Reading data

```
myData = read.csv("SomeFile.csv")
```

6.2.5 Reading from Excel

Actually there are several ways: * XLConnect * xlsx * Excelsi-r

```
library(XLConnect)
wbk = loadWorkbook("myWorkbook.xlsx")
df = readWorksheet(wbk, someSheet)
```

6.2.6 Reading from the web - 1

```
URL = "http://www.casact.org/research/reserve_data/ppauto_pos.csv"
df = read.csv(URL, stringsAsFactors = FALSE)
```

6.2.7 Reading from the web - 2

```
library(XML)
URL = "http://www.pro-football-reference.com/teams/nyj/2012_games.htm"
games = readHTMLTable(URL, stringsAsFactors = FALSE)
```

6.2.8 Reading from a database

```
library(RODBC)
myChannel = odbcConnect(dsn = "MyDSN_Name")
df = sqlQuery(myChannel, "SELECT stuff FROM myTable")
```

6.2.9 Read some data

```
df = read.csv("../data-raw/StateData.csv")
```

```
View(df)
```

6.2.10 Exercises

- Load the data from "StateData.csv" into a data frame.
- Which state has the most premium?

6.2.11 Answer

Chapter 7

Basic Visualization

In this chapter, we're going to talk about data visualization. By the end of this chapter, you will be able to:

- Create a scatter plot
- Display categorical information in a bar plot
- Visualize univariate sample data with histograms and density plots
- Emphasize outliers
- Alter visual characteristics based on your dataf

7.1 Overview

It's impossible to overstate the importance of visualization in data analysis. Rendering quantitative information visually is a critical aid in understanding our data, helping to model it and communicating our results. For a non-technical audience, it's

- Helps us explore data
- Suggest a model
- Assess the validity of a model and its parameters
- Vital for a non-technical audience

Although R has very powerful capabilities, its basic visualization is, well, basic. However, R's flexibility has allowed users to develop additional plotting engines that can produce some dazzling displays.

4 plotting engines (at least)

- base plotting system
- lattice
- ggplot2
- rCharts

7.1.1 Common geometric objects

- scatter
- line
- hist
- density
- boxplot
- barplot
- dotplot

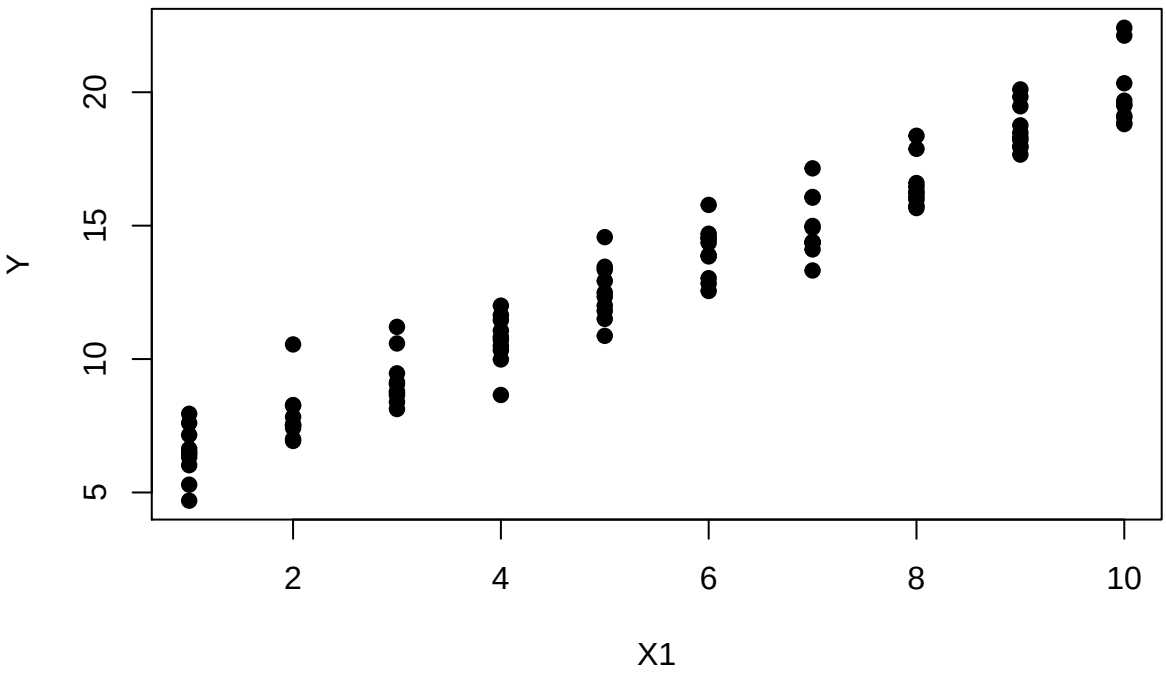
7.2 The `plot` function

plot is the most basic graphics command. There are several dozen options that you can set. Spend a lot of time reading the documentation and experimenting.

Open your first script.

7.2.1 A basic scatter plot

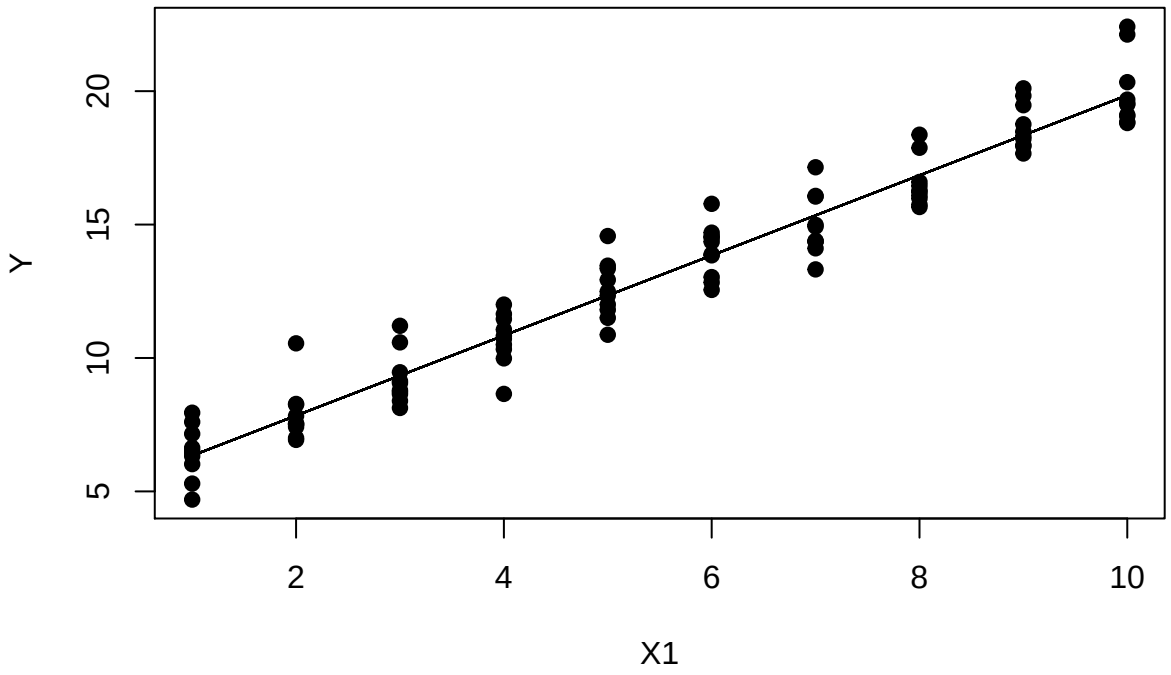
```
source("./scripts/BasicScript.R")
plot(X1, Y, pch=19)
```



7.2.2 Add lines

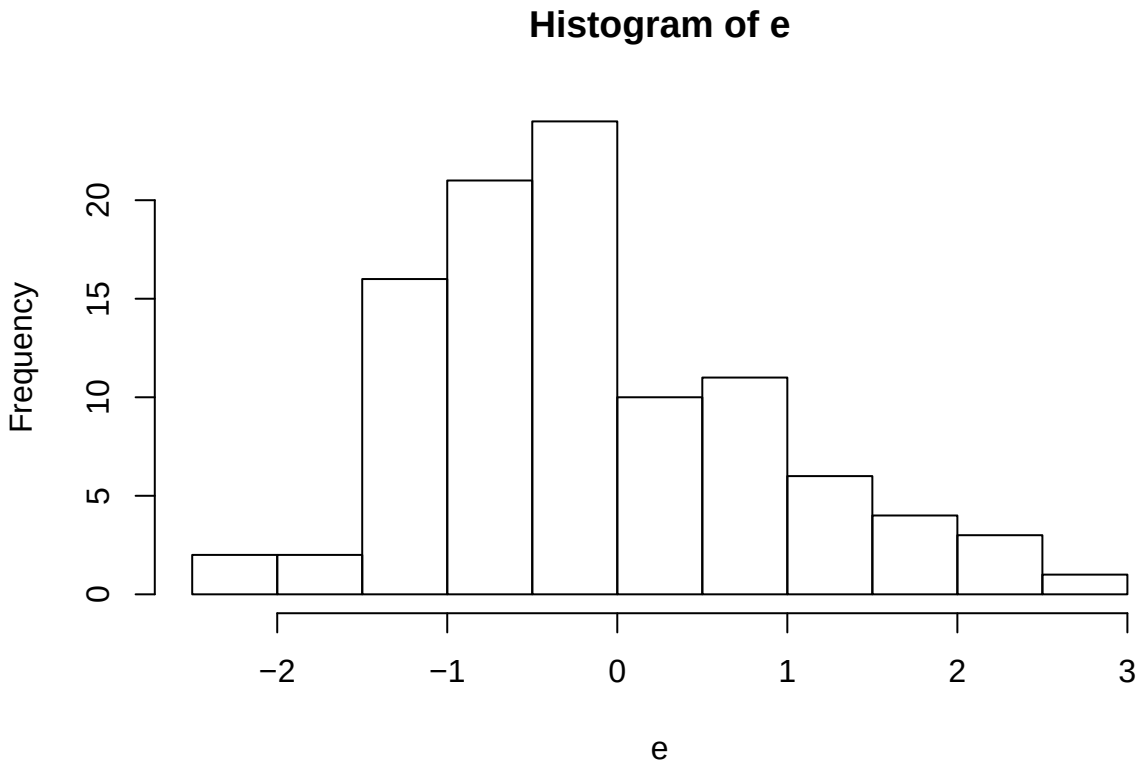
The functions 'lines' and 'points' will add (wait for it) lines and points to a pre-existing plot.

```
plot(X1, Y, pch=19)
lines(X1, yHat)
```



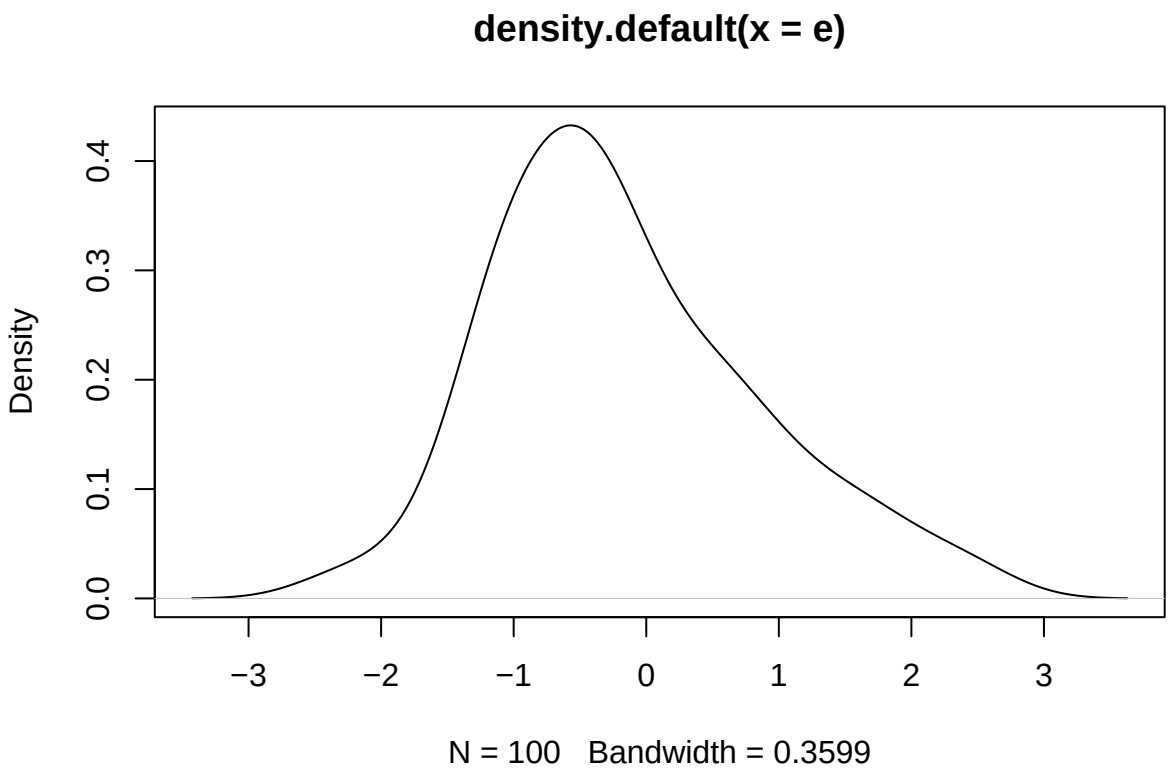
7.2.3 Histogram

```
hist(e)
```



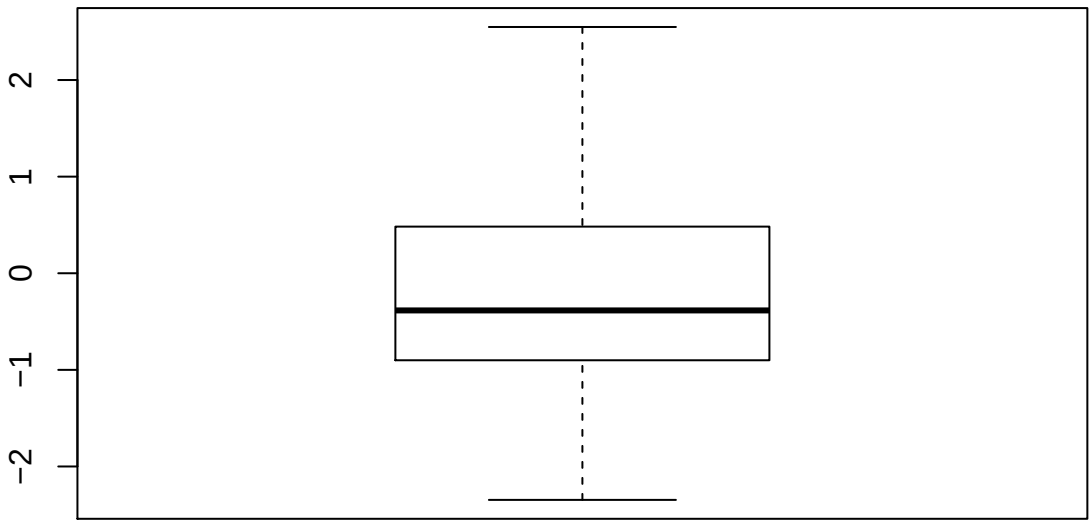
7.2.4 Density plot

```
plot(density(e))
```



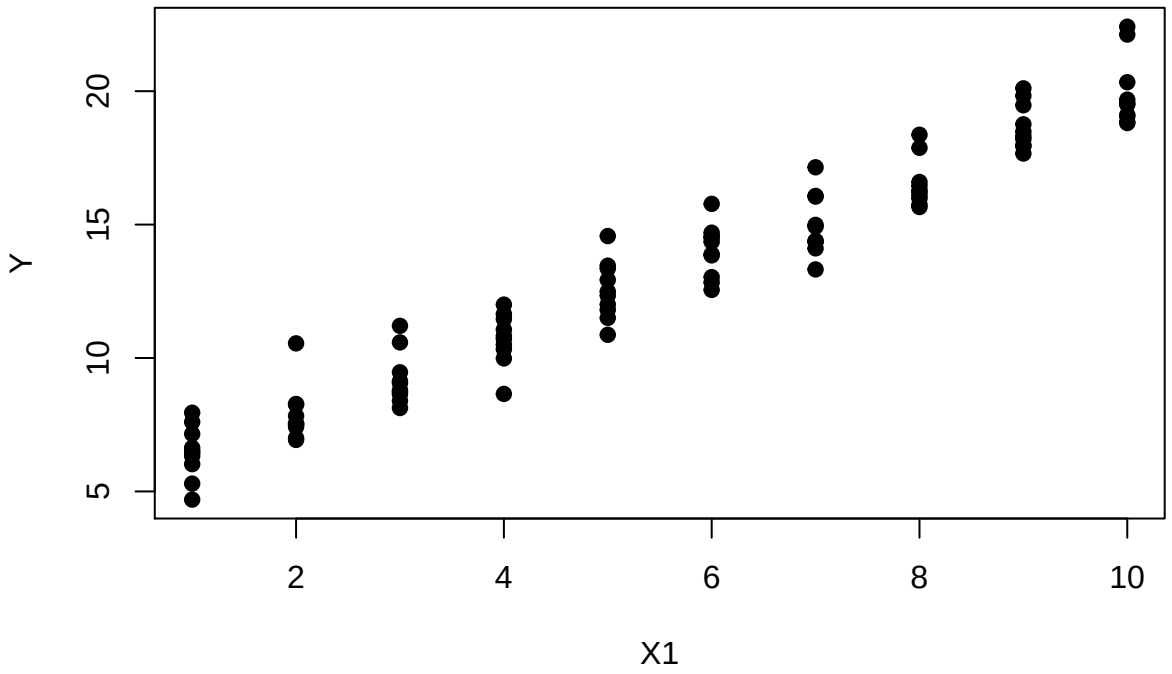
7.2.5 Boxplot

```
boxplot(e, pch=19)
```



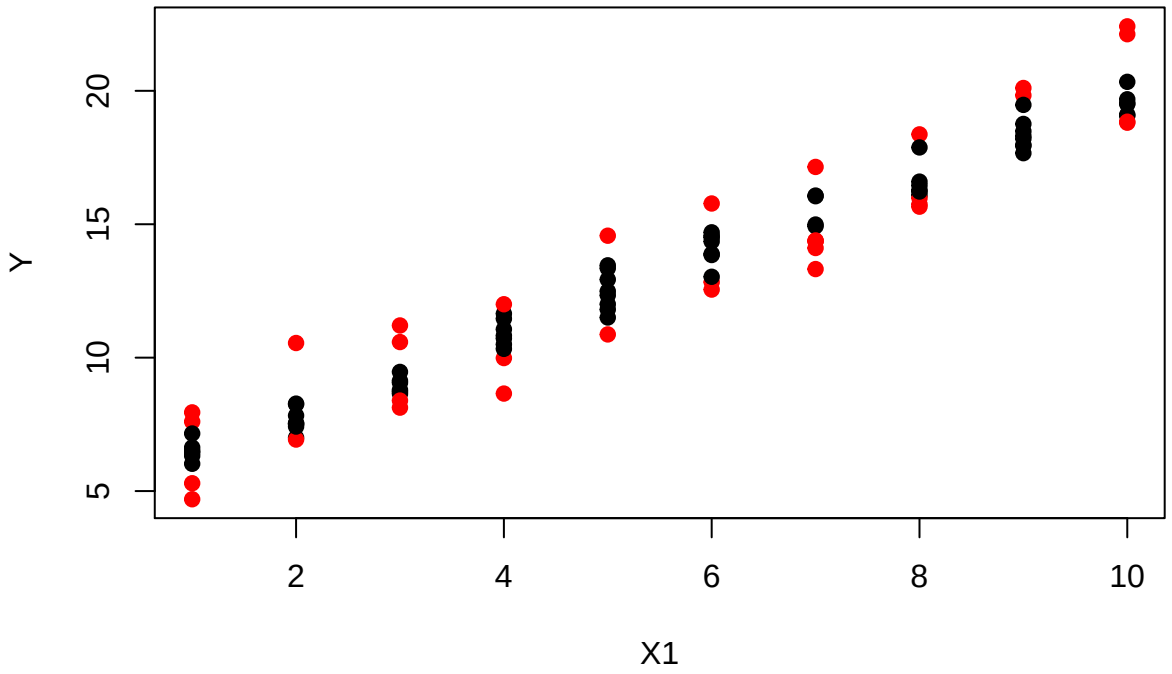
7.2.6 Plotting a formula

```
plot(Y ~ X1, pch=19)
```



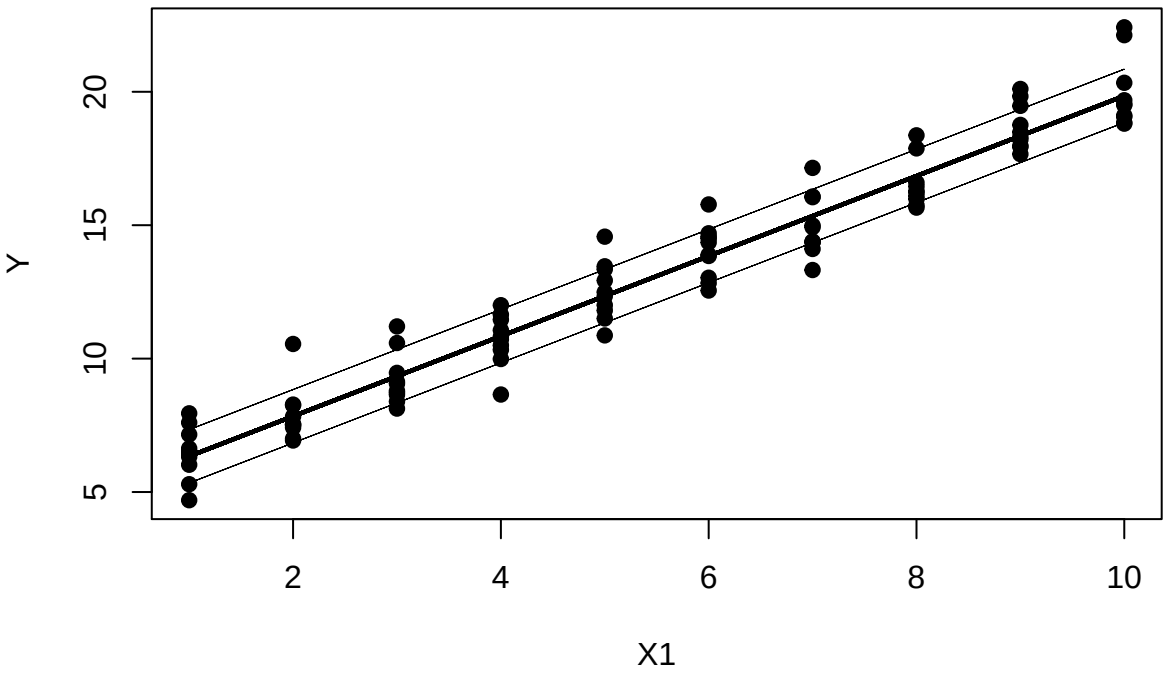
7.2.7 Emphasizing outliers

```
colors = ifelse(abs(e) > 1.0, "red", "black")
plot(Y ~ X1, pch=19, col=colors)
```



7.2.8 Other ways to emphasize outliers

```
plot(Y ~ X1, pch=19)
lines(X1, yHat, lwd=2)
lines(X1, yHat+1, lty="dotted", lwd=0.5)
lines(X1, yHat-1, lty="dotted", lwd=0.5)
```

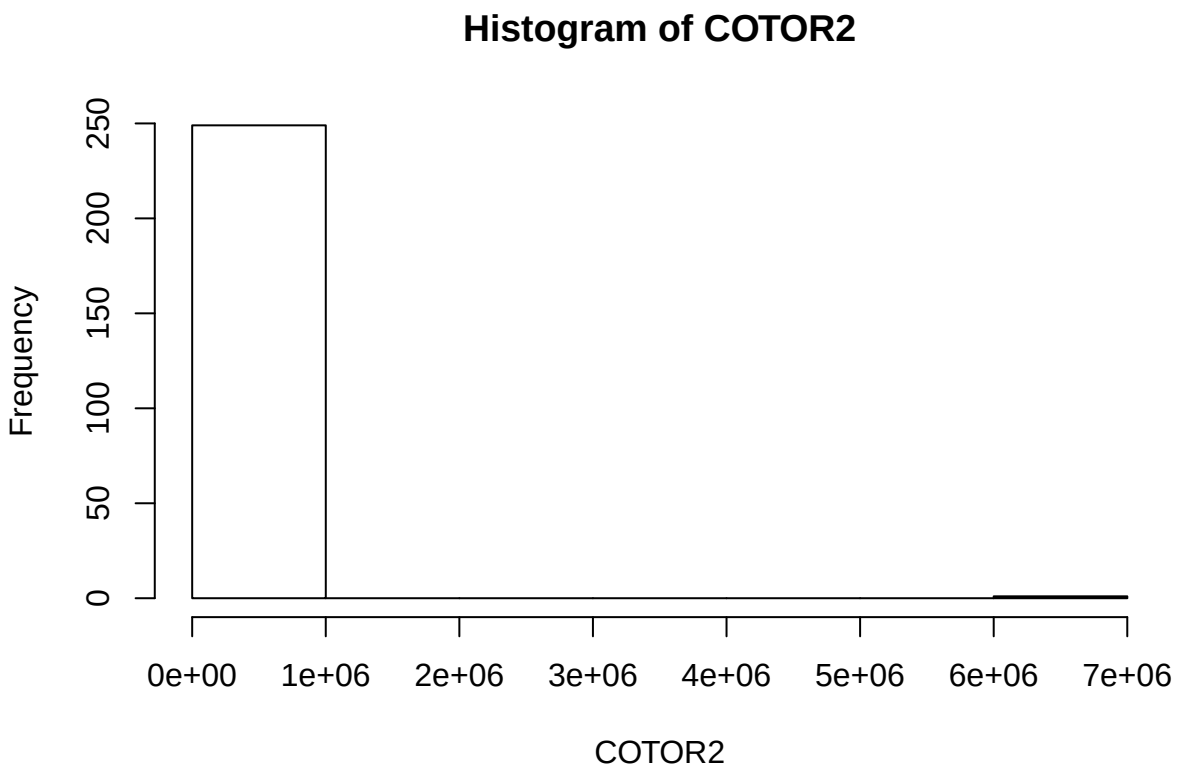


7.3 Exercise

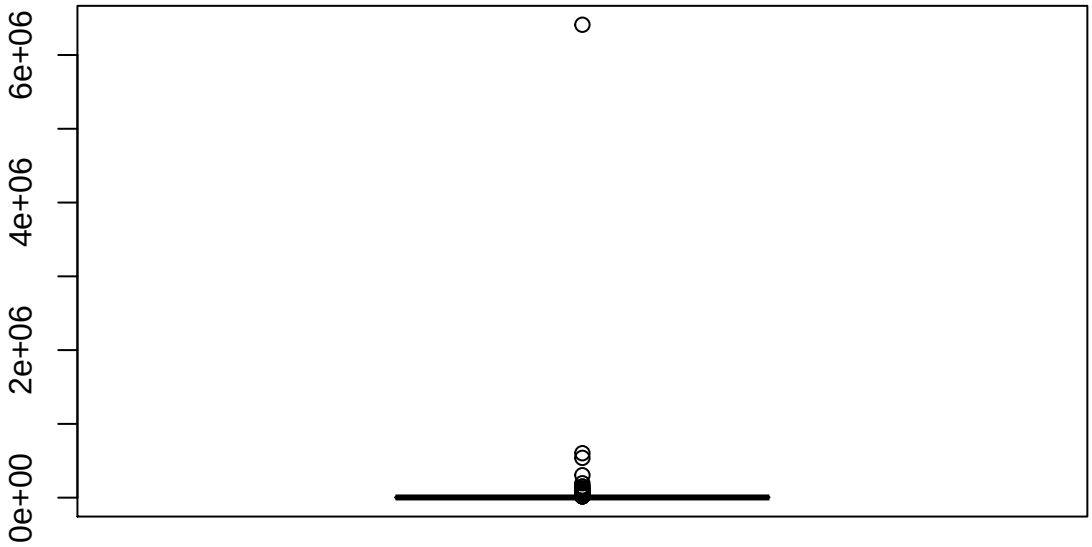
- Load the COTOR2 data from the raw package.
- Create a histogram, kernel density plot and box plot for the claims data

7.4 Answer

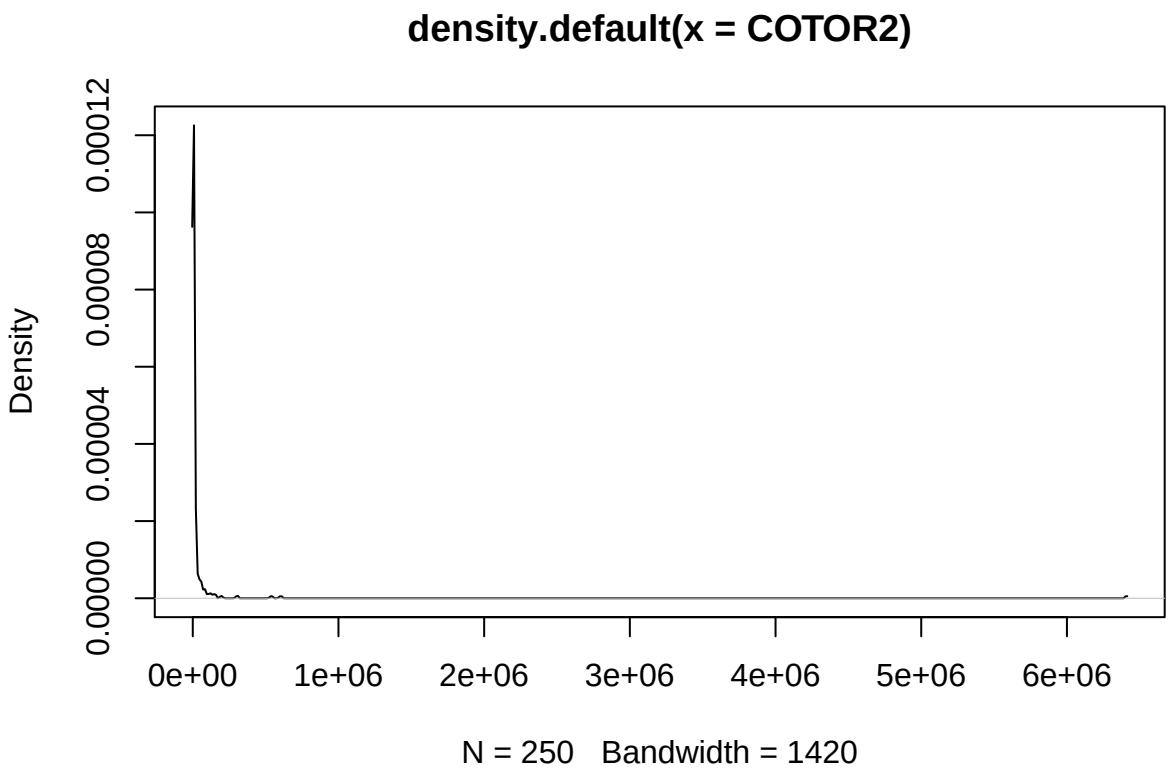
```
library(raw)
data(COTOR2)
hist(COTOR2)
```



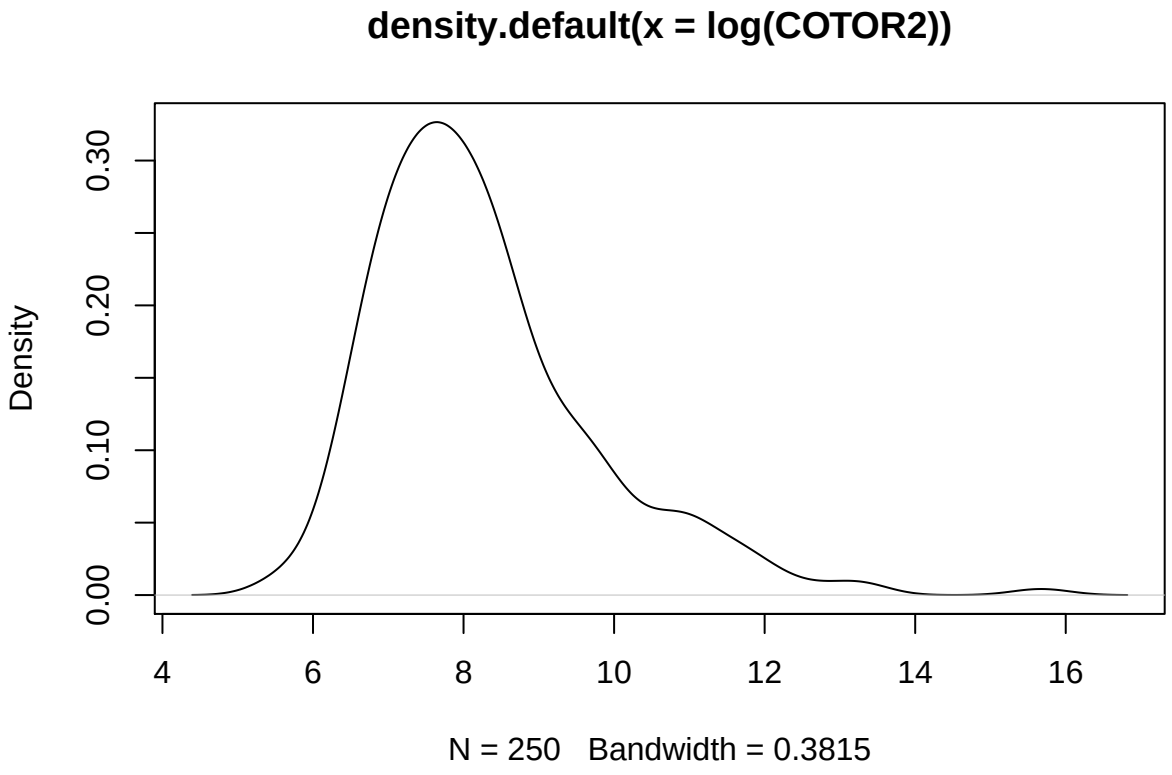

```
boxplot(COTOR2)
```



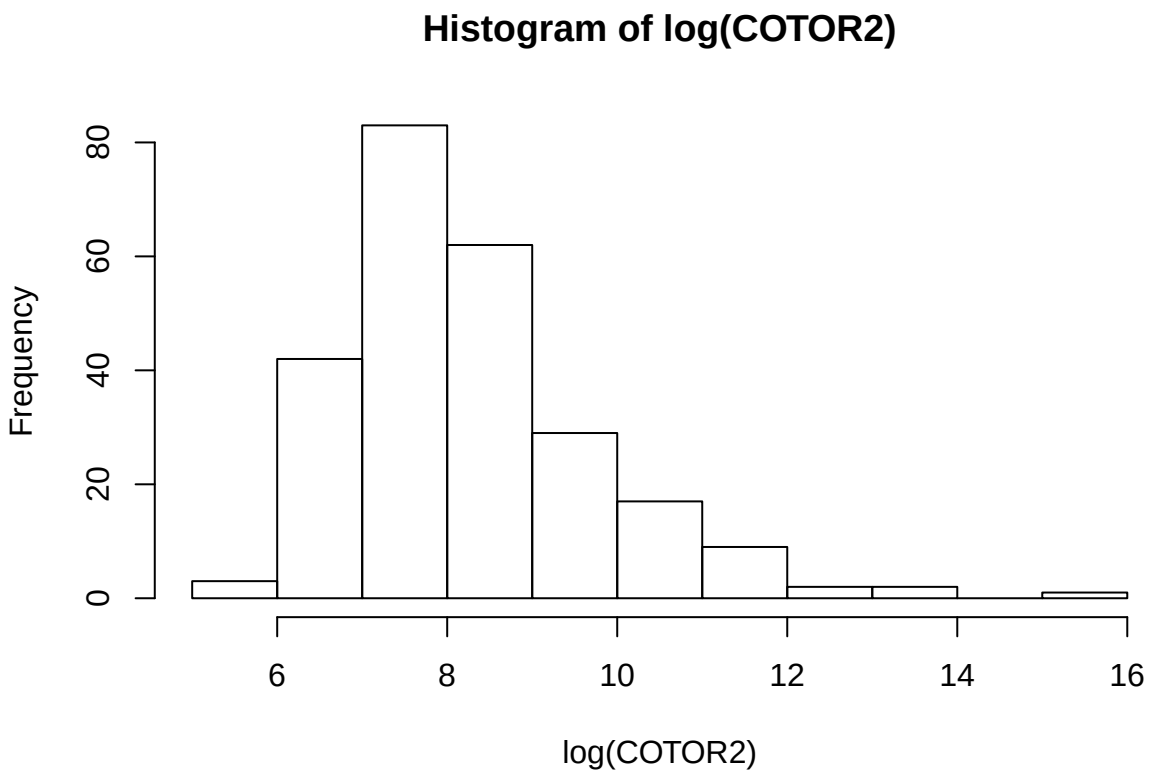

```
plot(density(COTOR2))
```



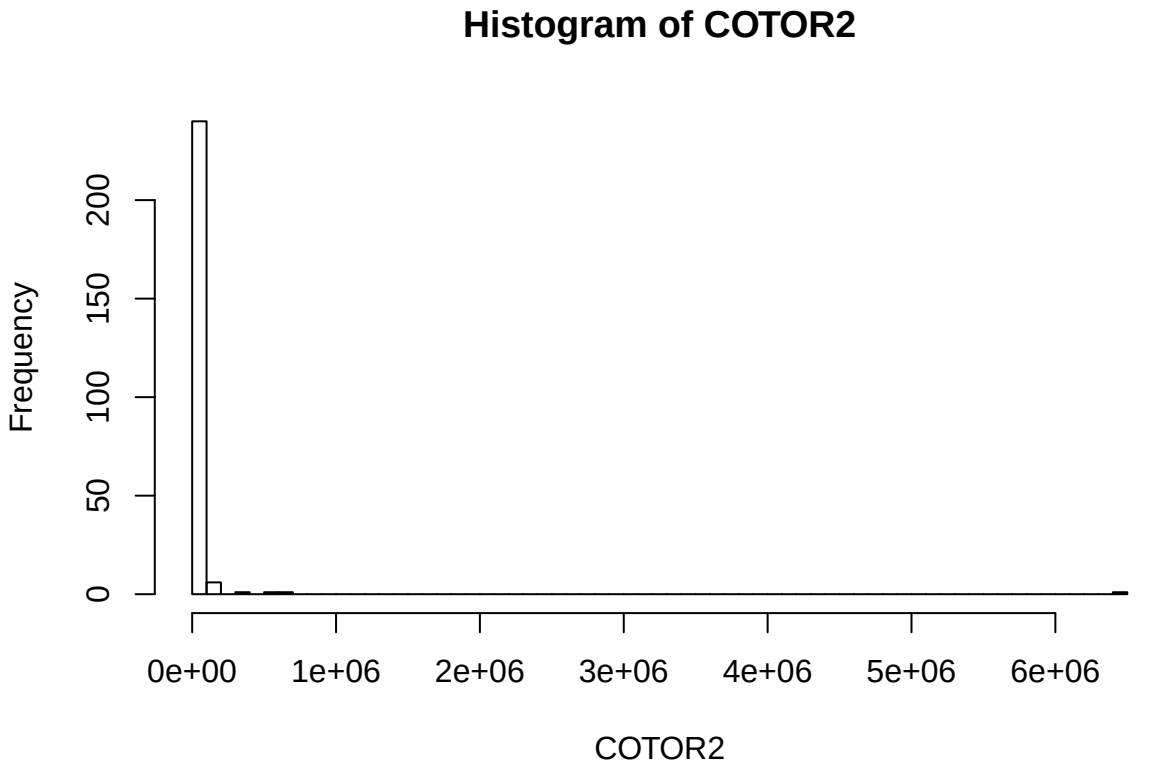

```
plot(density(log(COTOR2)))
```




```
hist(log(COTOR2))
```




```
hist(COTOR2, breaks=80)
```



7.5 Resources

- Nathan Yau - FlowingData.com
- Stephen Few - PerceptualEdge.com
- Edward Tufte - edwardtufte.com
- junkcharts.typepad.com

Chapter 8

Loss Distributions

By the end of this chapter, you will know the following:

- Simulation with `base` functions
- How to perform basic visualization of loss data
- How to fit a loss distributions
- Goodness of fit

8.1 Packages we'll use

- `MASS` (MASS = Modern Applied Statistics in S)
  - `fitdistr` will fit a distribution to a loss distribution function
- `actuar`
  - `emm` calculates empirical moments
  - `lev` limited expected value
  - `coverage` modifies a loss distribution for coverage elements
  - Contains many more distributions than are found in `base` R such as Burr, Pareto, etc. Basically, anything in "Loss Models" is likely to be found here.
  - Contains the dental claims data from "Loss Models"
- Direct optimization
  - `optim` function

8.1.1 Statistical distributions in R

Function names are one of 'd', 'p', 'q', 'r' + function name

- d - probability density
- p - cumulative distribution function
- q - quantiles
- r - random number generator

8.1.2 Examples

```
mu <- 10000
CV <- 0.30
sd <- mu * CV
x <- seq(mu - sd*3, mu + sd * 3, length.out = 20)
```

```
p <- seq(.05, .95, by = .05)

dnorm(x, mu, sd)
```

```
##  [1] 1.477283e-06 3.624482e-06 8.048577e-06 1.617645e-05 2.942646e-05
##  [6] 4.844888e-05 7.219719e-05 9.737506e-05 1.188683e-04 1.313334e-04
## [11] 1.313334e-04 1.188683e-04 9.737506e-05 7.219719e-05 4.844888e-05
## [16] 2.942646e-05 1.617645e-05 8.048577e-06 3.624482e-06 1.477283e-06
```

```
pnorm(x, mu, sd)
```

```
##  [1] 0.001349898 0.003635066 0.008932096 0.020054161 0.041207522
##  [6] 0.077650730 0.134522788 0.214917602 0.317862557 0.437269873
## [11] 0.562730127 0.682137443 0.785082398 0.865477212 0.922349270
## [16] 0.958792478 0.979945839 0.991067904 0.996364934 0.998650102
```

```
qnorm(p, mu, sd)
```

```
##  [1]  5065.439  6155.345  6890.700  7475.136  7976.531  8426.798  8844.039
##  [8]  9239.959  9623.016 10000.000 10376.984 10760.041 11155.961 11573.202
## [15] 12023.469 12524.864 13109.300 13844.655 14934.561
```

```
rnorm(10, mu, sd)
```

```
##  [1] 11243.571  8575.845 10197.980  8492.567  7522.004 10500.968  7311.206
##  [8] 10504.556 11064.905  9843.685
```

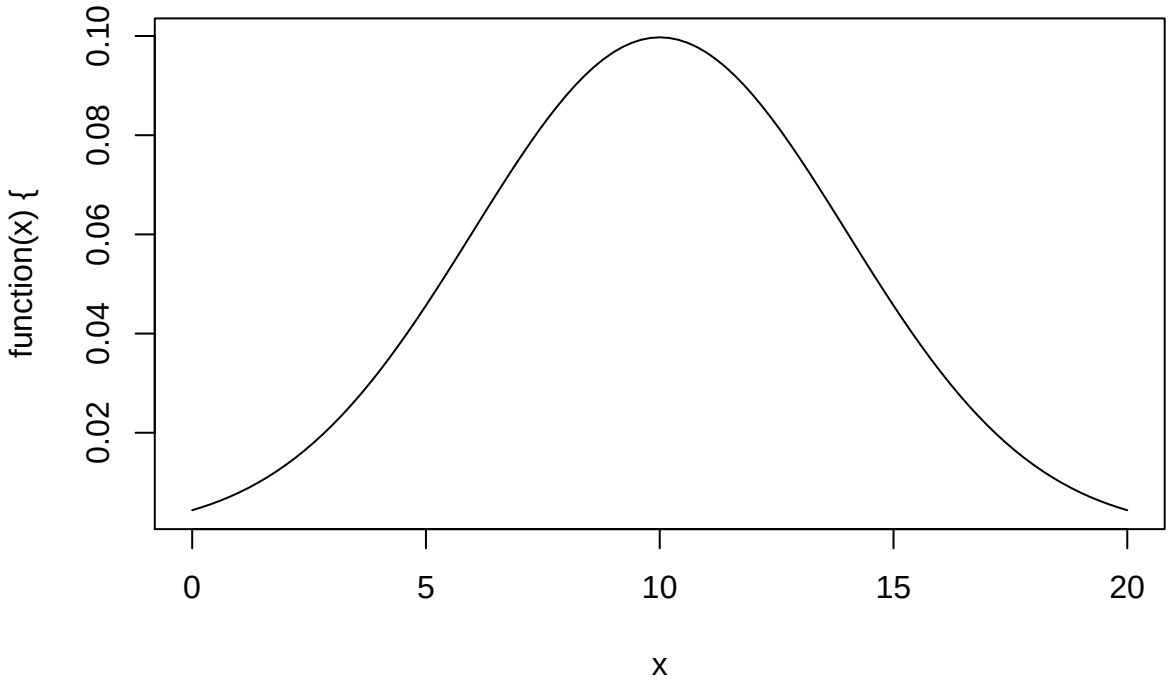
```
dlnorm(x, log(mu), log(sd))
```

```
##  [1] 4.780950e-05 2.505870e-05 1.700824e-05 1.287673e-05 1.035979e-05
##  [6] 8.664742e-06 7.445171e-06 6.525483e-06 5.807122e-06 5.230481e-06
## [11] 4.757380e-06 4.362230e-06 4.027237e-06 3.739640e-06 3.490050e-06
## [16] 3.271406e-06 3.078292e-06 2.906489e-06 2.752656e-06 2.614119e-06
```

```
plnorm(x, log(mu), log(sd))
```

```
##  [1] 0.3868287 0.4190398 0.4384745 0.4524493 0.4633699 0.4723339 0.4799359
##  [8] 0.4865345 0.4923631 0.4975820 0.5023061 0.5066205 0.5105903 0.5142660
## [15] 0.5176879 0.5208885 0.5238944 0.5267278 0.5294071 0.5319482
```

```
plot(function(x) {dnorm(x, 10, 4)}, 0, 20)
```



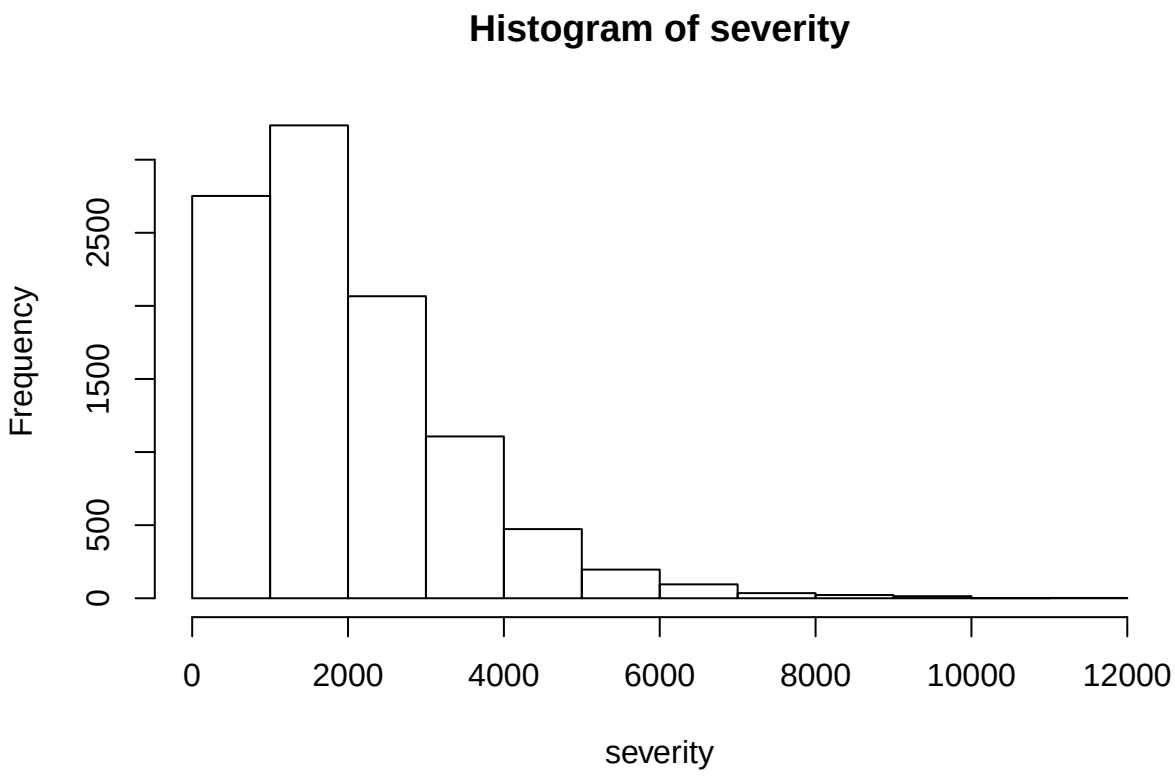
8.1.3 Generate some loss data

```
set.seed(8910)
years <- 2001:2010
frequency <- 1000

N <- rpois(length(years), frequency)

sevShape <- 2
sevScale <- 1000
severity <- rgamma(sum(N), sevShape, scale = sevScale)

summary(severity)
```

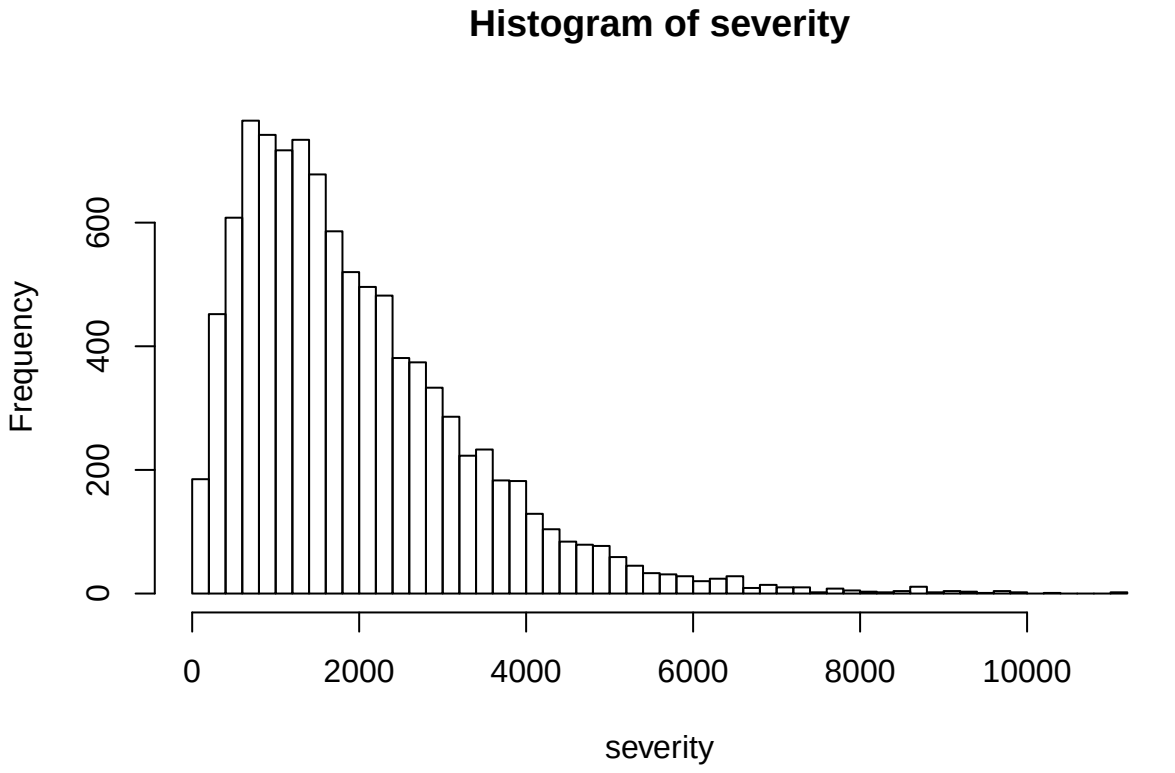
```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    21.93   928.90  1640.00  1970.00  2676.00 11130.00
```

8.1.4 Histograms

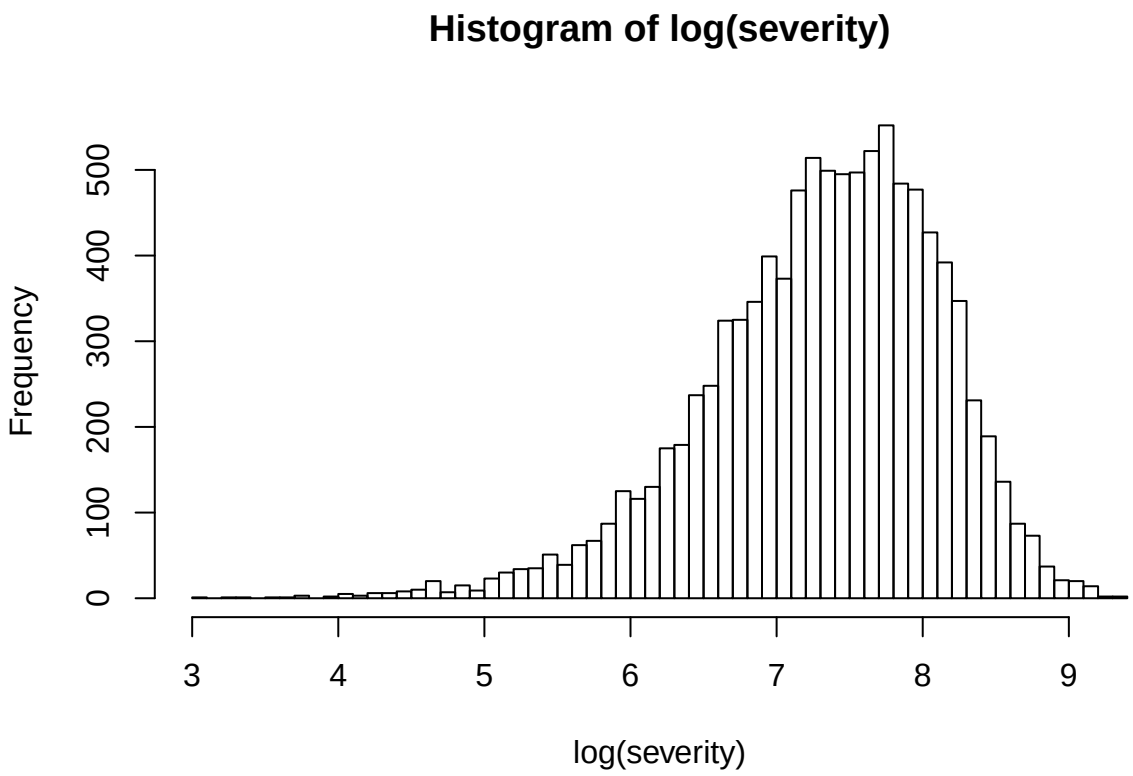
```
hist(severity)
```




```
hist(severity, breaks = 50)
```



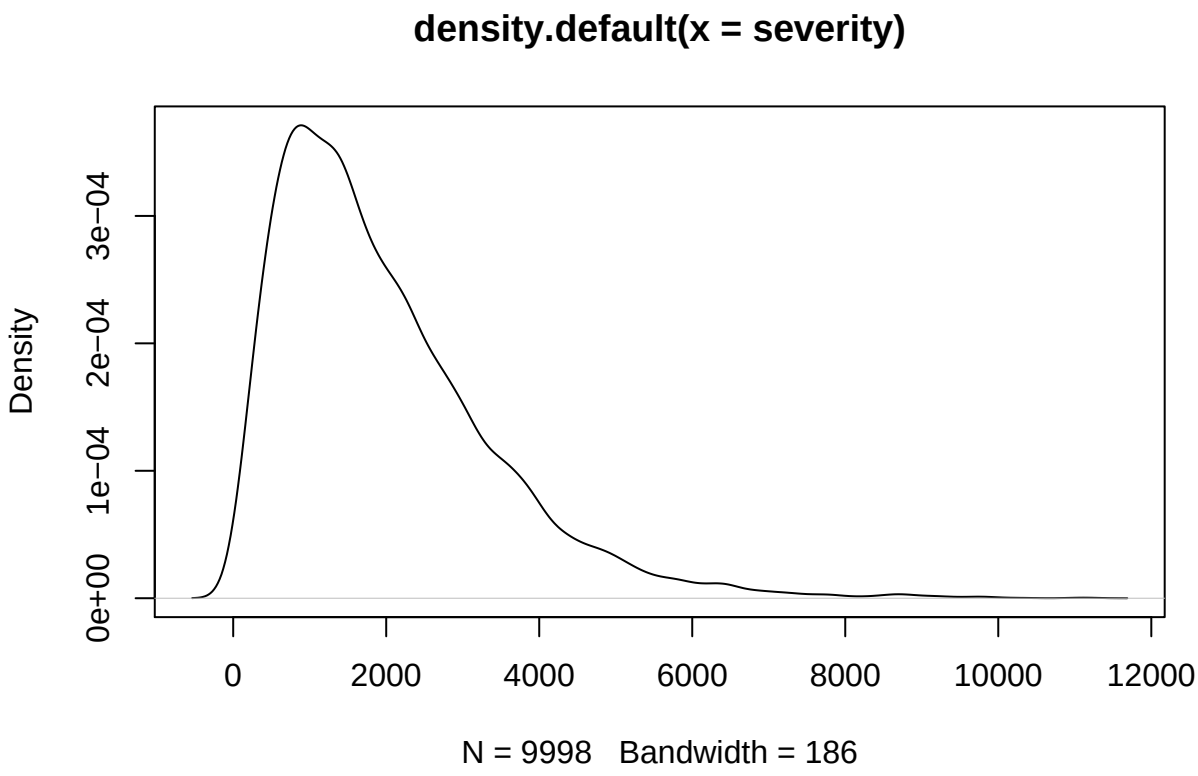

```
hist(log(severity), breaks = 50)
```



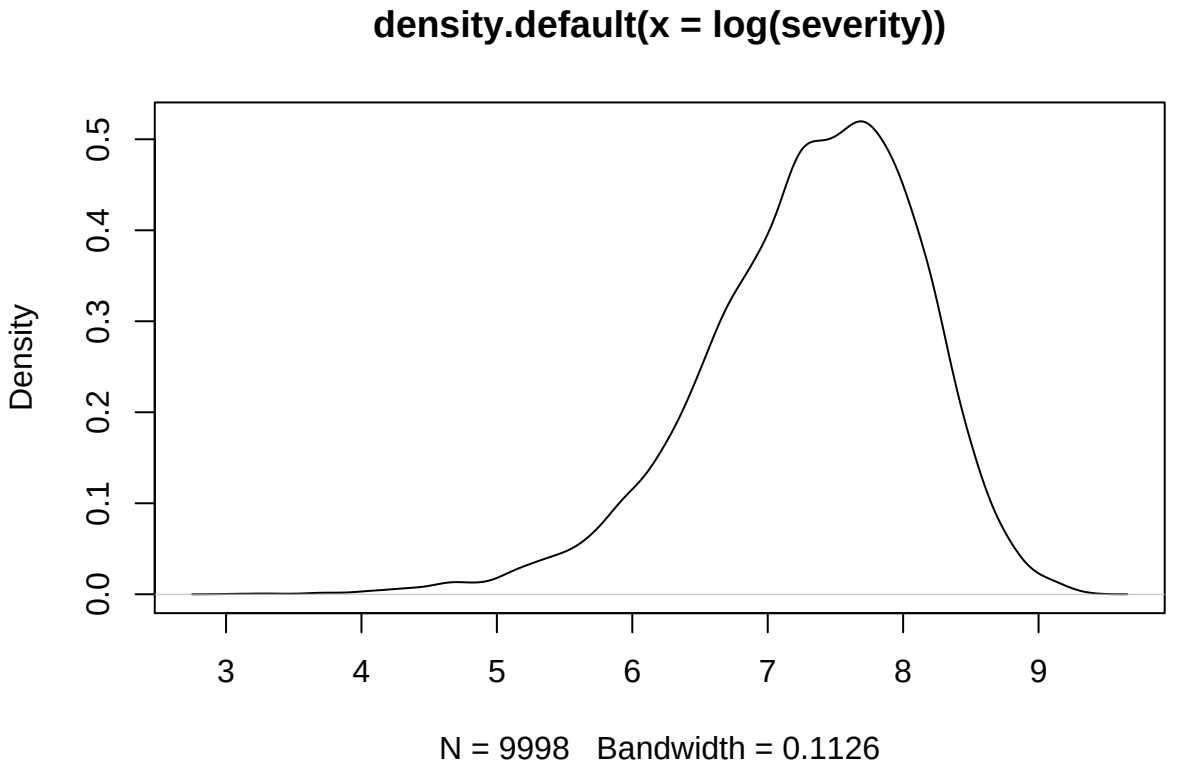
8.1.5 Density

The kernel density is effectively a smoothed histogram.

```
plot(density(severity))
```




```
plot(density(log(severity)))
```



8.1.6 fitdistr

```
library(MASS)

fitGamma <- fitdistr(severity, "gamma")
fitLognormal <- fitdistr(severity, "lognormal")
fitWeibull <- fitdistr(severity, "Weibull")
```

```
## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced

## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced

## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced
```

```
fitGamma
```

```
##       shape           rate
##   1.981942e+00   1.006292e-03
##  (1.261418e-02) (8.096328e-07)
```

```
fitLognormal
```

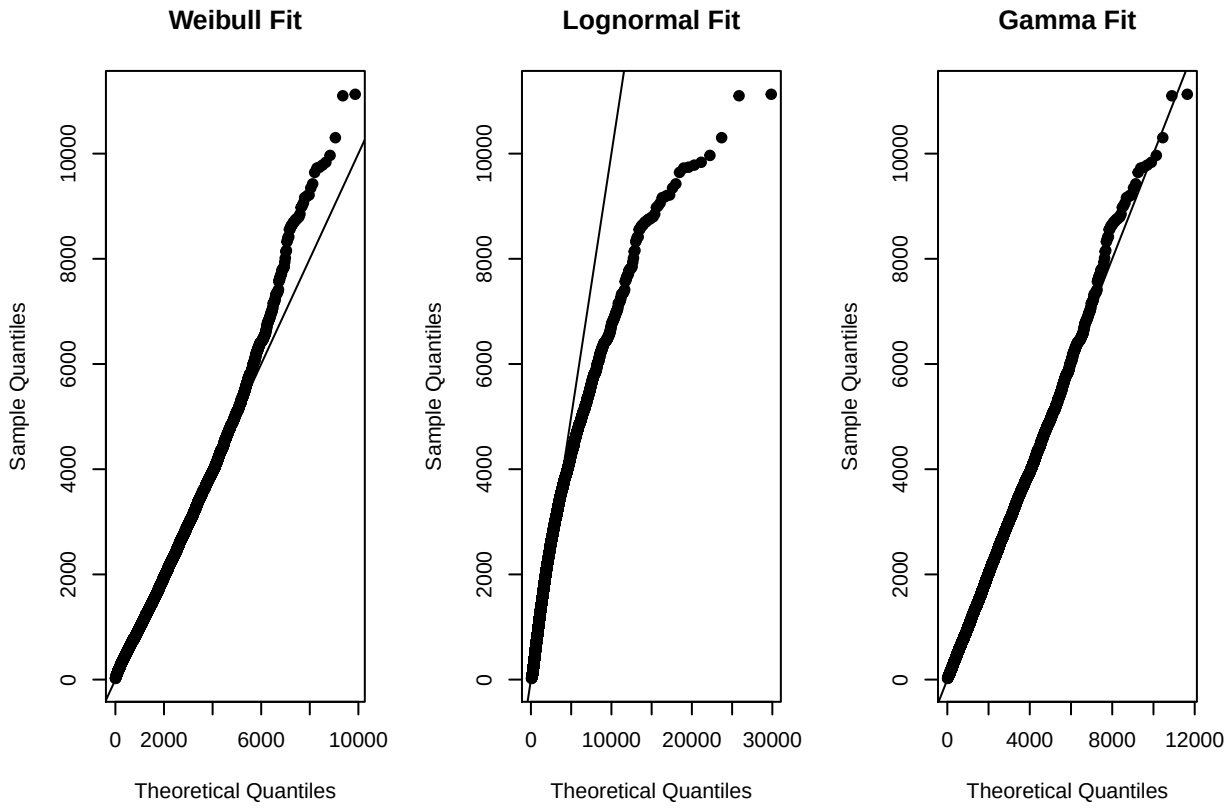
```
##      meanlog        sdlog
##   7.312554108   0.804493446
##  (0.008045739) (0.005689197)
```

```
fitWeibull
```

```
##       shape           scale
##   1.472049e+00   2.184117e+03
##  (1.117737e-02) (1.566082e+01)
```

8.1.7 q-q plot

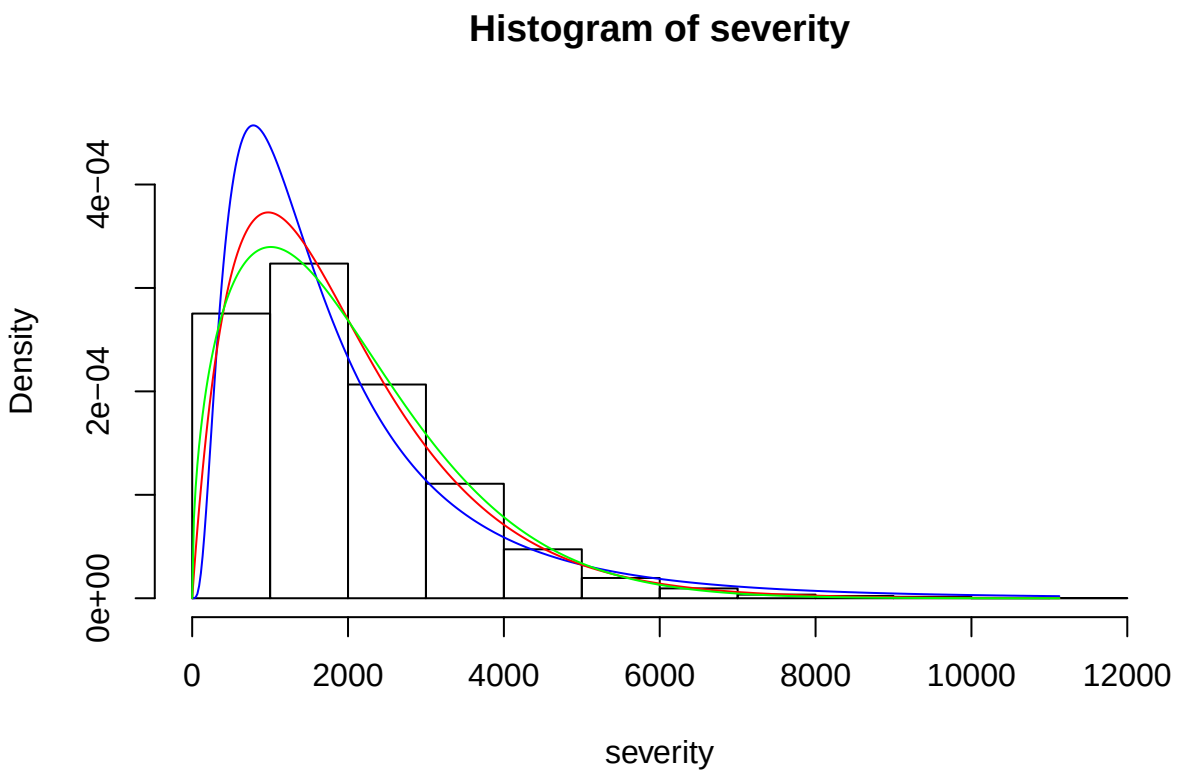
```
probabilities = (1:(sum(N)))/(sum(N)+1)

weibullQ <- qweibull(probabilities, coef(fitWeibull)[1], coef(fitWeibull)[2])
lnQ <- qlnorm(probabilities, coef(fitLognormal)[1], coef(fitLognormal)[2])
gammaQ <- qgamma(probabilities, coef(fitGamma)[1], coef(fitGamma)[2])

sortedSeverity <- sort(severity)
oldPar <- par(mfrow = c(1,3))
plot(sort(weibullQ), sortedSeverity, xlab = 'Theoretical Quantiles', ylab = 'Sample Quantiles', pch=19,
abline(0,1)

plot(sort(lnQ), sortedSeverity, xlab = 'Theoretical Quantiles', ylab = 'Sample Quantiles', pch=19, main
abline(0,1)

plot(sort(gammaQ), sortedSeverity, xlab = 'Theoretical Quantiles', ylab = 'Sample Quantiles', pch=19, ma
abline(0,1)
```




```
par(oldPar)
```

8.1.8 Compare fit to histogram

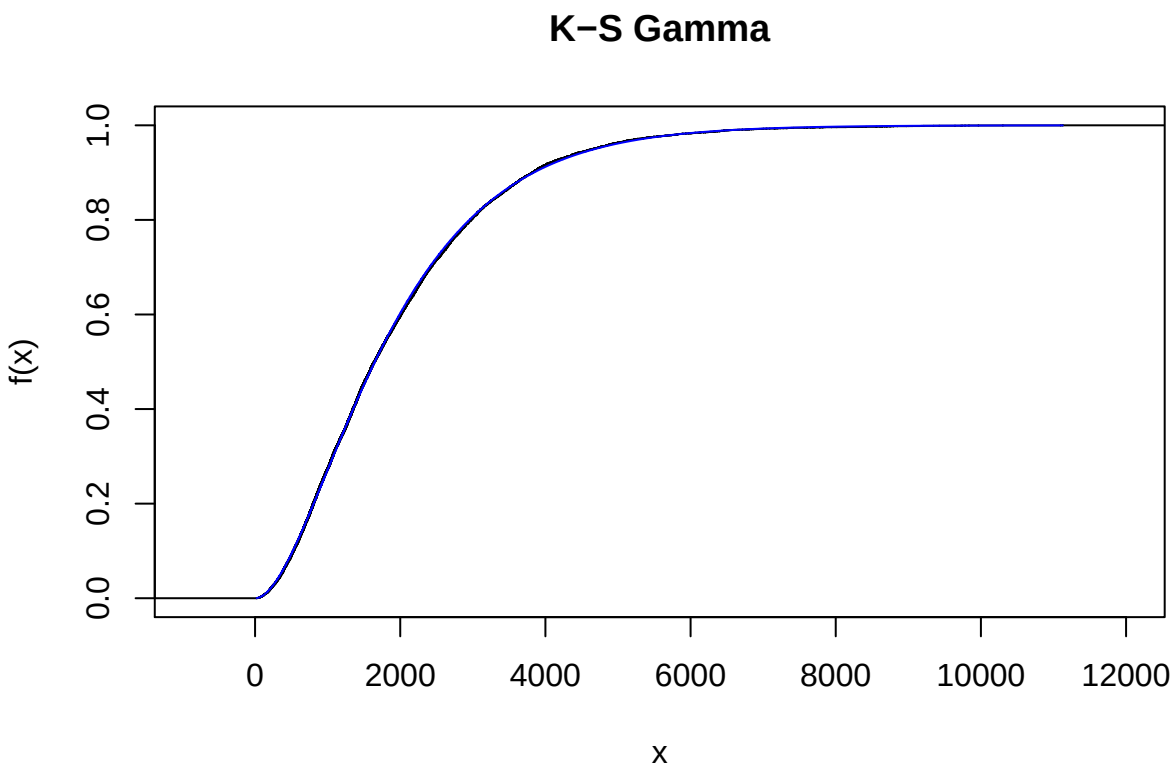
```
sampleLogMean <- fitLognormal$estimate[1]
sampleLogSd <- fitLognormal$estimate[2]

sampleShape <- fitGamma$estimate[1]
sampleRate <- fitGamma$estimate[2]

sampleShapeW <- fitWeibull$estimate[1]
sampleScaleW <- fitWeibull$estimate[2]

x <- seq(0, max(severity), length.out=500)
yLN <- dlnorm(x, sampleLogMean, sampleLogSd)
yGamma <- dgamma(x, sampleShape, sampleRate)
yWeibull <- dweibull(x, sampleShapeW, sampleScaleW)

hist(severity, freq=FALSE, ylim=range(yLN, yGamma))

lines(x, yLN, col="blue")
lines(x, yGamma, col="red")
lines(x, yWeibull, col="green")
```



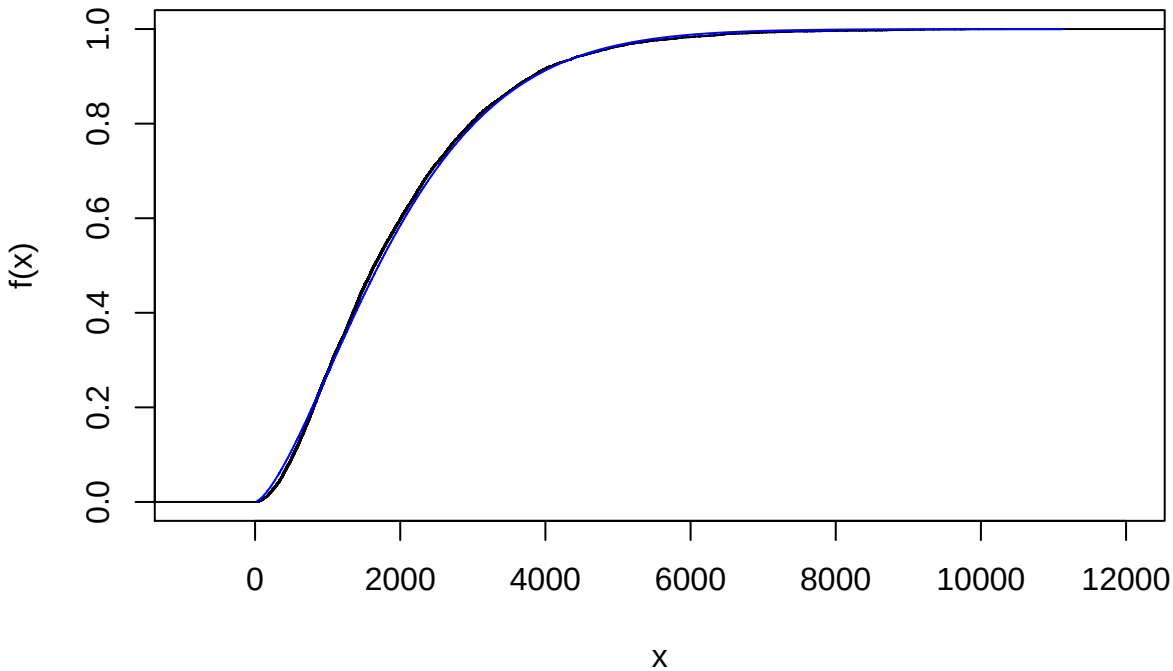
8.1.9 Kolmogorov-Smirnov

The Kolmogorov-Smirnov test measures the distance between an sample distribution and a candidate loss distribution. More formal than q-q plots.

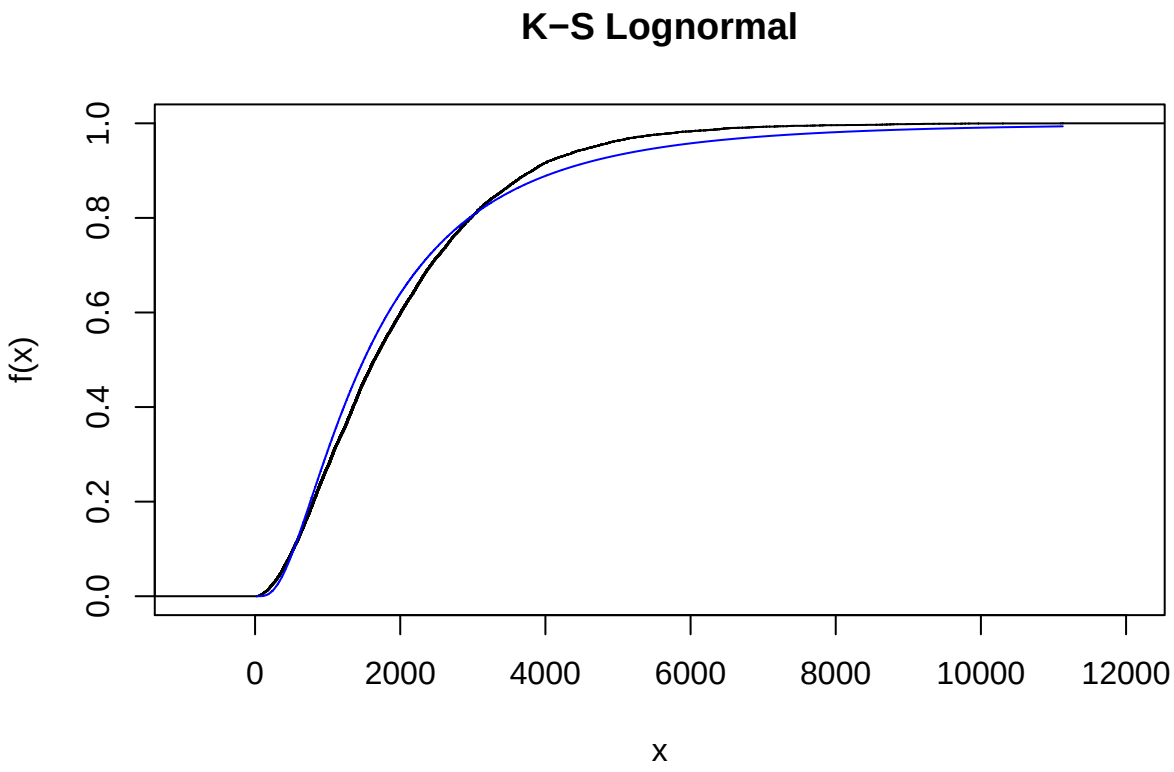
```
sampleCumul <- seq(1, length(severity)) / length(severity)
stepSample  <- stepfun(sortedSeverity, c(0, sampleCumul), f = 0)
yGamma <- pgamma(sortedSeverity, sampleShape, sampleRate)
yWeibull <- pweibull(sortedSeverity, sampleShapeW, sampleScaleW)
yLN <- plnorm(sortedSeverity, sampleLogMean, sampleLogSd)

plot(stepSample, col="black", main = "K-S Gamma")
lines(sortedSeverity, yGamma, col = "blue")
```




```
plot(stepSample, col="black", main = "K-S Weibull")
lines(sortedSeverity, yWeibull, col = "blue")
```




```
plot(stepSample, col="black", main = "K-S Lognormal")
lines(sortedSeverity, yLN, col = "blue")
```



8.1.10 More K-S

A low value for D indicates that the selected curve is fairly close to our data. The p-value indicates the chance that D was produced by the null hypothesis.

```
testGamma <- ks.test(severity, "pgamma", sampleShape, sampleRate)
testLN <- ks.test(severity, "plnorm", sampleLogMean, sampleLogSd)
testWeibull <- ks.test(severity, "pweibull", sampleShapeW, sampleScaleW)

testGamma
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  severity
## D = 0.0066186, p-value = 0.7735
## alternative hypothesis: two-sided
```

```
testLN
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  severity
## D = 0.047763, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

```
testWeibull
```

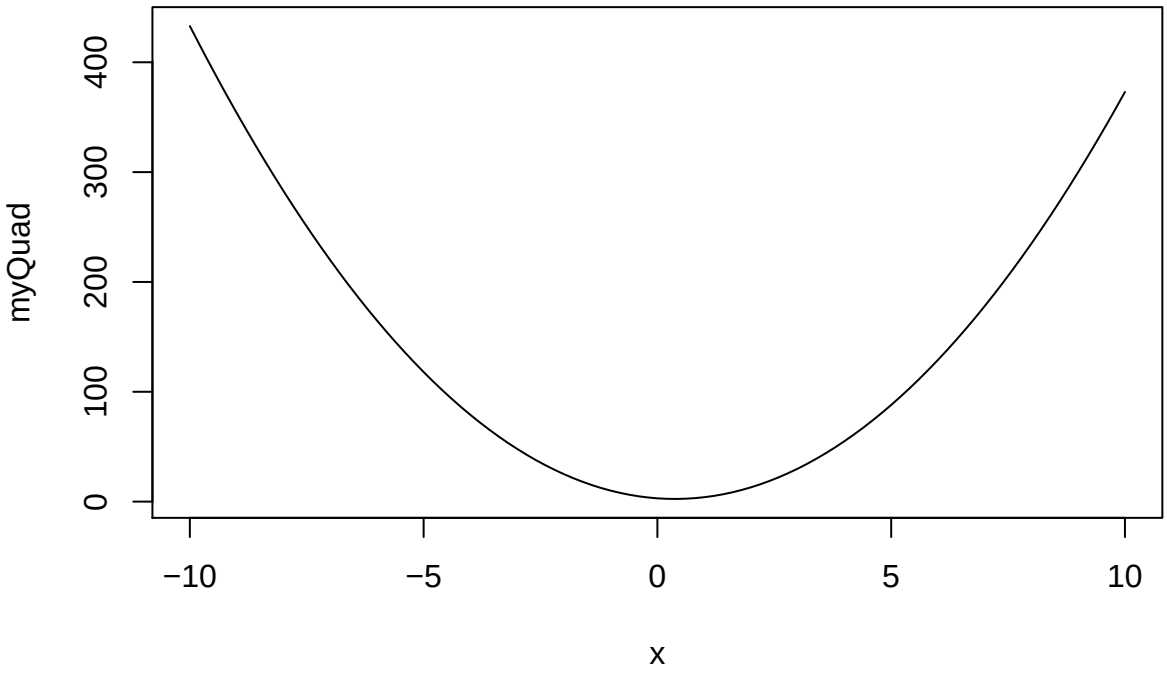
```
##
##  One-sample Kolmogorov-Smirnov test
##
```

```
## data:  severity
## D = 0.02053, p-value = 0.0004373
## alternative hypothesis: two-sided
```

8.1.11 Direct optimization

The `optim` function will optimize a function. Works very similar to the Solver algorithm in Excel. `optim` takes a function as an argument, so let's create a function.

```
quadraticFun <- function(a, b, c){
  function(x) a*x^2 + b*x + c
}

myQuad <- quadraticFun(a=4, b=-3, c=3)
plot(myQuad, -10, 10)
```



8.1.12 Direct optimization

8 is our initial guess. A good initial guess will speed up conversion.

```
myResult <- optim(8, myQuad)
```

```
## Warning in optim(8, myQuad): one-dimensional optimization by Nelder-Mead is unreliable:
## use "Brent" or optimize() directly
```

```
myResult
```

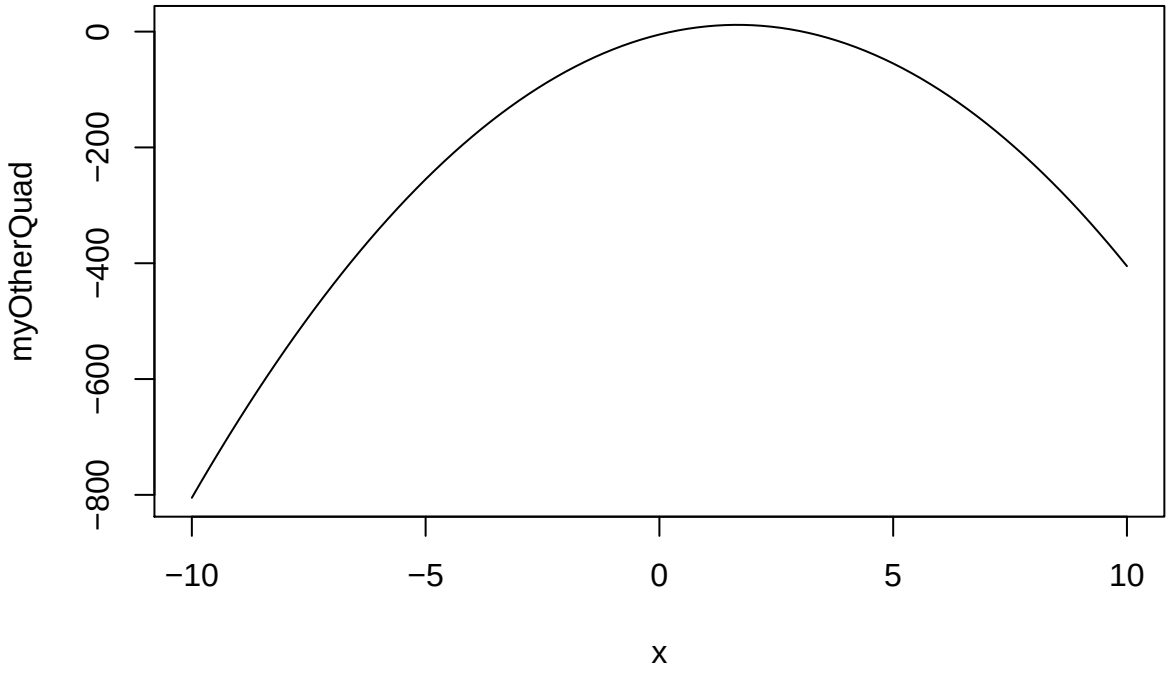
```
## $par
## [1] 0.4
##
## $value
## [1] 2.44
##
```

```
## $counts
## function gradient
##       20       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

8.2 Direct optimization

Default is to minimize. Set the parameter `fnscale` to something negative to convert to a maximization problem.

```
myOtherQuad <- quadraticFun(-6, 20, -5)
plot(myOtherQuad, -10, 10)
```




```
myResult <- optim(8, myOtherQuad)
```

```
## Warning in optim(8, myOtherQuad): one-dimensional optimization by Nelder-Mead is unreliable:
## use "Brent" or optimize() directly
```

```
myResult <- optim(8, myOtherQuad, control = list(fnscale=-1))
```

```
## Warning in optim(8, myOtherQuad, control = list(fnscale = -1)): one-dimensional optimization by Nelder-Me
## use "Brent" or optimize() directly
```

8.2.1 Direct optimization

Direct optimization allows us to create another objective function to maximize, or work with loss distributions for which there isn't yet support in a package like `actuar`. May be used for general purpose optimization problems, e.g. maximize rate of return for various capital allocation methods.

Note that optimization is a general, solved problem. Things like the simplex method already have package solutions in R. You don't need to reinvent the wheel!

8.3 Exercises

- Plot a lognormal distribution with a mean of $10,000 and a CV of 30%.
- For that distribution, what is the probability of seeing a claim greater than $100,000?
- Generate 100 and 1,000 observations from that distribution.
- Draw a histogram for each sample.
- What are the mean, standard deviation and CV of each sample?
- Convince yourself that the sample data were not produced by a Weibull distribution.
- Assuming that losses are Poisson distributed, with expected value of 200, estimate the aggregate loss distribution.
- What is the cost of a $50,000 xs $50,000 layer of reinsurance?

8.3.1 Answers

```
severity <- 10000
CV <- .3
sigma <- sqrt(log(1 + CV^2))
mu <- log(severity) - sigma^2/2
plot(function(x) dlnorm(x), mu, sigma, ylab="LN f(x)")
```