

ESMA 6835: Computing with R

Dr. Wolfgang Rolke

August 12, 2018

Contents

0.1 Syllabus	4
1 Installation and Updating	5
2 R Markdown	6
2.1 snippets	11
3 History	11
3.1 S	11
3.2 R	12
3.3 RStudio	12
3.4 Getting Started	12
3.5 Saving your Stuff	13
3.6 Assignments	13
3.7 Help!	14
3.8 ls, rm	14
3.9 Naming Conventions	15
4 Basic Data Types	17
4.1 Type Conversion	19
4.2 Dates:	21
4.3 Factor	21
5 Data Formats: Vectors, Matrices, Arrays, Dataframes and Lists	22
5.1 Vectors	22
5.2 Matrix	23
5.3 Arrays	26
5.4 Dataframe	27
5.5 Lists	28
6 Generating Vectors	29
6.1 Expressions	32
7 Subsetting / Data Wrangling	34
7.1 Vectors	34
7.2 Useful logic commands:	38
7.3 <i>subset</i>	40
8 Generate Random Variates	40
8.1 Random Numbers	40
8.2 Pseudo-Random Numbers	41
8.3 Standard Probability Distributions	42
8.4 Other Variates	44
9 Writing your own functions	47
9.1 Basic programming structures in R	48

9.2	Some general advice on computer programming	50
9.3	Finding Errors, Debugging	51
9.4	Functional Programming	51
9.5	Infix functions	57
10	Object-Oriented Programming	62
11	Vector Arithmetic	68
11.1	Matrix Algebra	69
11.2	Cycling	71
11.3	Vectorize	72
11.4	apply family of functions	73
12	Character Manipulation	75
12.1	Regular Expressions	79
12.2	POSIX	81
12.3	Some Examples	81
13	Data Input/Output, Transferring R Objects	91
13.1	Printing Info to Screen	91
13.2	Reading a Vector	92
13.3	Data Frames	94
13.4	Transferring Objects from one R to another	94
13.5	Special File Formats	95
13.6	Packages	95
13.7	Working on Files	95
14	Graphs	96
14.1	Barcharts	96
14.2	Histogram	98
14.3	Empirical Distribution Function	99
14.4	Boxplot	100
14.5	Normal Probability Plot	101
14.6	Scatterplot	102
14.7	Multiple Graphs	105
14.8	Functions that create graphs	107
14.9	Graphs that don't look like Graphs	107
14.10	Printing Graphs	109
15	Model Notation	110
16	Numerical Methods	115
16.1	Integration	115
16.2	Differentiation	118
16.3	Root Finding	123
17	Environments, Libraries	126
17.1	Environments	126
17.2	Packages	140
17.3	Creating your own library	141
18	Costumizing R: .First, .Rprofile, Dropbox	144
18.1	Dropbox	145
19	Graphics with ggplot2	146
19.1	Why ggplot2?	146

19.2 Grammar of Graphics	146
19.3 Some standard graphs	156
19.4 Axis Ticks and Legend Keys	163
19.5 Saving the graph	165
20 Rcpp	166
20.1 Debugging	168
20.2 Sugar	168
20.3 STD (standard template library)	169
20.4 Using C++ routines.	171
21 Parallel and GPU Computing	172
21.1 GPU Programming	177
22 Input/Output Part 2	178
22.1 rio	178
22.2 readr	179
22.3 data.table	179
22.4 pdftools	180
23 The pipe, dplyr, tibbles, tidyverse	185
23.1 tibbles	187
23.2 dplyr	188
23.3 The tidyverse	192
24 Character Manipulation with stringr	194
24.1 Dracula by Bram Stoker	198
25 Dates with lubridate	203
25.1 Time Spans	206
26 Factors with forcats	206
27 Iteration with purrr	213
28 Interactive Web Applications with shiny	222
28.1 Create/Run/Distribute Shiny Apps	228
29 Version Control and Collaboration, Github	229
29.1 Setting up a new repo	230
30 Estimation	230
30.1 Packages	244
31 The Bootstrap	244
31.1 Confidence Intervals	249
32 Curve Fitting	252
32.1 Parametric Models	252
32.2 Nonparametric Regression	260
33 Bayesian Analysis	265
33.1 Prior and Posterior Distribution	265
33.2 Inference	269
34 Classification	279

34.1 Example: Fisher's Iris data set	279
34.2 Misclassification Rate	299
34.3 Fisher's iris data	301

0.1 Syllabus

Professor: Dr. Wolfgang Rolke

Time and Place:

Tuesday, Thursday 2:00 - 3:15pm SH005

0.1.1 Reading List:

There are literally 100s of books about R, with new ones coming out almost every week. Here are some that I found useful:

Learning R

Introduction to Data Science with R

R Cookbook

Advanced R

0.1.2 Office hours

Tuesday, Thursday 12:00-12:30pm OF407

Tuesday, Thursday 3:15-5:15pm OF407 (by appointment)

Wednesday 1:30-3:00pm via email

email: wolfgang[dot]rolke[at]upr[dot]edu

when you send me an email **ALWAYS** start the subject line with ESMA6835

0.1.3 Grading:

1. Homework: 50%
2. Midterm 25%
3. Final 25%

0.1.4 Submit your work

all homeworks and exams have to be done using RMarkdown and have to be submitted via Dropbox. You will be sent an invitation to join Dropbox soon (if you are not already a user). A little while later you will receive an email with an invitation to share a Dropbox folder with me.

From then on anytime you put something in this folder I will immediately have it in mine as well. So when you are ready to submit your homework simply drop the file into the folder.

1 Installation and Updating

You can get a free version of R for your computer from a number of sources. The download is about 70MB and setup is fully automatic. Versions for several operating systems can be found on the R web site

<https://cran.r-project.org>

Note

- the one item you should change from the defaults is to install R into a folder under the root, aka C:\R
- You might be asked at several times whether you want to do something (allow access, run a program, save a library, . . .), always just say yes!
- You will need to connect to a reasonably fast internet for these steps.
- This will take a few minutes, just wait until the > sign appears.

FOR MAC OS USERS ONLY

There are a few things that are different from MacOS and Windows. Here is one thing you should do:

Download XQuartz - XQuartz-2.7.11.dmg

Open XQuartz

Type the letter R (to make XQuartz run R)

Hit enter Open R Run the command .First()

Then, every command should work correctly.

1.0.1 RStudio

We will run R using an interface called **RStudio**. You can download it at RStudio.

1.0.2 Updating

R releases new versions about every three months or so. In general it is not necessary to get the latest version every time. Every now and then a package won't run under the old version, and then it is time to do so. In essence this just means to install the latest version of R from CRAN. More important is to now also update ALL your packages to the latest versions. This is done simply by running

```
update.packages(ask=FALSE, dependencies=TRUE)
```

2 R Markdown

R Markdown is a program for making dynamic documents with R. An R Markdown document is written in *markdown*, an easy-to-write plain text format with the file extension .Rmd. It can contain chunks of embedded R code. It has a number of great features:

- easy syntax for a number of basic objects
- code and output are in the same place and so are always synced
- several output formats (html, latex, word)

In recent years I (along with many others) who work a lot with R have made Rmarkdown the basic way to work with R. So when I work on a new project I immediately start a corresponding R markdown document.

2.0.1 Get Started

to start writing an R Markdown document open RStudio, File > New File > R Markdown. You can type in the title and some other things.

The default document starts like this:

```
---
```

title: "My first R Markdown Document"
author: "Dr. Wolfgang Rolke"
date: "April 1, 2018"
output: html_document

```
---
```

This follows a syntax called YAML (also used by other programs). There are other things that can be put here as well, or you can erase all of it.

YAML stands for Yet Another Markup Language. It has become a standard for many computer languages to describe different configurations. For details go to yaml.org

Then there is other stuff you should erase. Next File > Save. Give the document a name with the extension .Rmd

I have a number of things that I need in (almost) all of my Rmd files, and I am to lazy to erase the stuff that the default starting document comes with. So I found the file that RStudio uses at this point, it is called r_markdown_v2.Rmd and sits in the folder/RStudio/resources/templates (on my Win 10 machine, anyway). I renamed it r_markdown_v2OLD.Rmd (just so I can get it back if something goes wrong) and put what I want in it. Now when I click on File > New File > R Markdown, I get a file that is ready to go!

2.0.2 Basic R Markdown Syntax

for a list of the basic syntax go to https://rmarkdown.rstudio.com/articles_intro.html

2.0.3 Embedded Code

There are two ways to include code chunks (yes, that's what they are called!) into an R Markdown document:

- a. stand alone code

simultaneously enter CTRL-ALT-i and you will see this:

```
```{r}
```

```
``
```

you can now enter any R code you like:

```
```{r}
x <- rnorm(10)
mean(x)
``
```

which will appear in the final document as

```
x <- rnorm(10)
mean(x)
```

Actually, it will be like this:

```
x<-rnorm(10)
mean(x)
```

```
## [1] -0.1762089
```

so we can see the result of the R calculation as well. The reason it didn't appear like this before was that I added the argument eval=FALSE:

```
```{r eval=FALSE}
```

which keeps the code chunk from actually executing (aka *evaluating*). This is useful if the code takes along time to run, or if you want to show code that is actually faulty, or ...

there are several useful arguments:

- eval=FALSE (shows but doesn't run the code)
- eval=2:5 (shows all the code but only runs lines 2 to 5)
- echo=FALSE (the code chunk is run but does not appear in the document)
- echo=2:5 (shows only code on lines 2 to 5)
- warning=FALSE (warnings are not shown)
- message=FALSE (messages are not shown)
- cache=TRUE (code is run only if there has been a change, useful for lengthy calculations)

- error=TRUE (if there are errors in the code R normally terminates the parsing (executing) of the markdown document. With this argument it will ignore the error, which helps with debugging)
- engine='Rcpp' (to include C++ code)
  - inline code.

here is a bit of text:

and the mean was -0.1762089.

Now I didn't type in the number, it was done with the chunk

```
`r mean(x)`
```

Many of these options can be set globally, so they are active for the whole document. This is useful so you don't have to type them in every time. I have the following code chunk at the beginning of all my Rmd:

```
library(knitr)
opts_chunk$set(fig.width=6, fig.align = "center",
 out.width = "70%", warning=FALSE, message=FALSE)
```

We have already seen the message and warning options. The other one puts any figure in the middle of the page and sizes it nicely.

If you have to override these defaults just include that in the specific chunk.

## 2.0.4 Creating Output

To create the output you have to “knit” the document. This is done by clicking on the *knit* button above. If you click on the arrow you can change the output format.

### 2.0.4.1 HTML vs Latex(Pdf)

In order to knit to pdf you have to install a latex interpreter. My suggestion is to use Miktex, but if you already have one installed it might work as well.

There are several advantages / disadvantages to each output format:

- HTML is much faster
- HTML looks good on a webpage, pdf looks good on paper
- HTML needs an internet connection to display math, pdf does not
- HTML can use both html and latex syntax, pdf works only with latex (and a little bit of html)

I generally use HTML when writing a document, and use pdf only when everything else is done. There is one problem with this, namely that a document might well knit ok to HTML

but give an error message when knitting to pdf. Moreover, those error messages are weird! Not even the line numbers are anywhere near right. So it's not a bad idea to also knit to pdf every now and then.

## 2.0.5 Tables

One of the more complicated things to do in R Markdown is tables. For a nice illustration look at

<https://stackoverflow.com/questions/19997242/simple-manual-rmarkdown-tables-that-look-good-in-html-pdf>

My preference is to generate a data frame and then use the *kable* function:

```
Gender <- c("Male", "Male", "Female")
Age <- c(20, 21, 19)
knitr::kable(data.frame(Gender, Age))
```

Gender	Age
Male	20
Male	21
Female	19

probably with the argument echo=FALSE so only the table is visible.

It is also possible to use HTML code to make a table:

```
<table border="1">
<tr><th>Gender</th><th>Age</th></tr>
<tr><td>Male</td><td>20</td></tr>
<tr><td>Male</td><td>21</td></tr>
<tr><td>Female</td><td>19</td></tr>
</table>
```

will look like this in HTML:

Gender

Age

Male

20

Male

21

Female

19

but won't look like anything in pdf.

The corresponding latex table will look good in pdf but not in HTML!

There is a solution, however: the document can check what the output format is at run time, and then insert the corresponding code. This works as follows. Say we want to include some code to print a piece of text in red, say for highlighting it. Now in html we would need the code <font color="red">, then the text and finally </font> to get back to black. In latex however we need \textcolor{red}{our text}. Here is a little routine that will do it:

```
fontcolor <- function (txt)
{
 library(knitr)
 output.format = opts_knit$get("rmarkdown.pandoc.to")
 if(output.format == "latex")
 out <- paste0("\\textcolor{red}{", txt, "}")
 else out <- paste0(, txt, "")
 out
}
```

and now if we have

```
`r fontcolor("this is in red")`
```

it will appear as **this is in red** in either html or latex.

## 2.0.6 LATEX

You have not worked with latex (read: latek) before? Here is your chance to learn. It is well worthwhile, latex is the standard document wordprocessor for science. And once you get used to it is WAY better and easier than (say) Word.

Because latex code will generally display correctly in an html document but html will not in a latex document I suggest to stick as much as possible with latex.

A nice list of common symbols is found on <https://artofproblemsolving.com/wiki/index.php/LaTeX:Symbols>.

### 2.0.6.1 Multiline math

say you want the following in your document:

$$\begin{aligned} E[X] &= \int_{-\infty}^{\infty} xf(x)dx = \\ &\int_0^1 xdx = \frac{1}{2}x^2|_0^1 = \frac{1}{2} \end{aligned}$$

for this to display correctly in HTML and PDF you need to use the format

```
@@
\begin{aligned}
&E[X] = \int_{-\infty}^{\infty} xf(x) dx = \\
&\int_0^1 xdx = \frac{1}{2}x^2|_0^1 = \frac{1}{2}
\end{aligned}
```

```
$$
```

By default when you knit to pdf the intermediate latex file is deleted. If you want to keep it, maybe so you can change it in a latex editor, use the following in the YAML header:

```
output:
```

```
 pdf_document:
```

```
 keep_tex: true
```

notice the spaces before the text, they are needed!

## 2.1 snippets

A *snippet* is a short piece of code that one uses quite often, and so it would be nice not to have to type it in every time. RStudio has a number of them pre-defined. Go to Tools > Global Options > Code > Edit Snippets.

There are snippets for various languages, including R Markdown. To use a snippet, simply type the code and then Shift+Tab.

You can even write your own! For example, I have one called *mta* that has all the basics to start a multi-line latex math expression.

## 3 History

### 3.1 S

S is a statistical programming language developed primarily by John Chambers and (in earlier versions) Rick Becker and Allan Wilks of Bell Laboratories. The aim of the language, as expressed by John Chambers, is “to turn ideas into software, quickly and faithfully”.

The first working version of S was built in 1976, and operated on the GCOS operating system. At this time, S was unnamed, and suggestions included Interactive SCS (ISCS), Statistical Computing System, and Statistical Analysis System (which was already taken: see SAS System). The name ‘S’ (used with single quotation marks, until 1979) was chosen, as it has the common letter used in statistical computing, and is consistent with other programming languages designed from the same institution at the time (namely the C programming language).

By 1988, many changes were made to S and the syntax of the language. The New S Language (1988 Blue Book) was published to introduce the new features, such as the transition from macros to functions and how functions can be passed to other functions (such as apply).

In the late 1990s AT&T sold the S license to a private company that started to charge a lot of money for it. This led to

## 3.2 R

R is an implementation of the S programming language combined with lexical scoping semantics inspired by Scheme. There are some important differences between S and R, but much of the code written for S runs unaltered.

R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team, of which Chambers is a member. R is named partly after the first names of the first two R authors and partly as a play on the name of S. The project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.

## 3.3 RStudio

RStudio is a free and open-source integrated development environment (IDE) for R, a programming language for statistical computing and graphics. RStudio was founded by JJ Allaire, creator of the programming language ColdFusion. Hadley Wickham is the Chief Scientist at RStudio.

Work on RStudio started at around December 2010, and the first public beta version (v0.92) was officially announced in February 2011. Version 1.0 was released on 1 November 2016. Version 1.1 was released on 9 October 2017.

R can be run by itself using the command line interface, but it is usually much nicer to use RStudio as a front end.

## 3.4 Getting Started

Once R (or Rstudio) is opened you will see the >. This is the *command prompt*. It means R is waiting for you to do something!

Don't like >? No sweat, you can change it (although I don't recommend changing the prompt). Here is how:

```
options(prompt="@")
```

In fact, you can change almost anything with the options command.

```
length(options())
```

```
[1] 81
```

shows there are 81 of them. Two you might want to remember are

```
options("digits")
```

```
$digits
[1] 7
```

```
pi
```

```
[1] 3.141593
options(digits=5)
pi
```

```
[1] 3.1416
options(digits=15)
```

and

```
options()$warn
```

```
[1] 0
```

this tells R to warn about certain things. Those are not errors, just warnings, and usually they are ok. Sometimes though they are not needed and can be a nuisance, and then you can turn them off with

```
options(warn=-1)
```

### 3.5 Saving your Stuff

The standard setup is to have separate folders for each project. When starting a new one RStudio creates the folder and within the files

- foldername.Rproj is the RStudio project file
  - .RData is the R worksheet

In the past the .RData file was the “heart” of R, and typically there was one for each project, with different names. RStudio wants you to essentially forget about it, there is actually an option in Tools > Global Options to just ignore any RData file. Personally (maybe because I am so used to it) I prefer to also have a distinct name for the Rdata worksheet. You can do that by running

```
save.image("c:/folder/filename.RData")
```

### 3.6 Assignments

The basic assignment character in R is <-

```
x <- 1
x
```

```
[1] 1
```

*Note* = also works:

```
x = 2
x
```

```
[1] 2
```

but for some good reasons is not the best way to go.

RStudio has a lot of useful shortcuts, one of them is ALT - , which enters <- and even the spaces around it!

Check out the list of keyboard shortcuts at Tools > Keyboards shortcuts

### 3.7 Help!

R has a help command. Let's say we want to find out about the *mean* command:

```
?mean
```

Another useful function is *arg*, which lists the arguments of a routine:

```
args(splot)
```

```
function (y, x, z, w, use.facets = FALSE, add.line = 0, plot.points = TRUE,
jitter = FALSE, errorbars = FALSE, label_x, label_y, label_z,
main_title, add.text, add.text_x, add.text_y, plotting.symbols = NA,
plotting.size = 1, plotting.colors = NA, ref_x, ref_y, log_x = FALSE,
log_y = FALSE, no.legend = FALSE, return.graph = FALSE)
NULL
```

*splot* is not a command that is part of R but one I wrote myself to make nice looking scatterplots. Unfortunately *args* often doesn't work well for basic R commands:

```
args(mean)
```

```
function (x, ...)
NULL
```

### 3.8 ls, rm

use the *ls* command to get a listing of the current objects in the worksheet:

```
ls() [1:5]
```

```
[1] "acorn" "Age" "agesex" "agesexUS" "aids"
```

If there are many objects (the worksheet I have open right now has over 250!) you can specify part of the name:

```
ls(pattern = "sa")
```

```
[1] "salaries"
```

the ^ in front means that *ls* will list all object whose name starts with sa. This is what is called a *regular expression*. These are general (non R) rules for specifying patterns. They are a science all their own, and we will need to have a look at them at some point.

If you need some more details on the objects you can use

```
ls.str(pattern = "sa")

salaries : 'data.frame': 25 obs. of 3 variables:
$ Salary: int 21700 24000 23800 26000 27200 25100 26700 27600 28300 29100 ...
$ Years : int 3 9 10 11 12 4 6 7 9 11 ...
$ Level : chr "Low" "Low" "Low" "Low" ...
```

Often you create an object that is only needed for a short time. If you want to get rid of it use `rm` (remove)

```
x <- 1:10
sum(x^3)

[1] 3025

rm(x)
```

you can also remove many objects in one step. Let's say we have worked on a problem for a while and we created a number of objects. All of them start with "my". Now we can do this

```
a <- ls(pattern = "my")
rm(list=a)
```

I have a routine called `cleanup`:

```
cleanup <- function ()
{
 options(warn=-1)
 oldstuff <- c("f", "f1", "F", "n", "m", "a", "A", "b", "B",
 "i", "j", "plt", "plt1", "plt2", "x", "xx", "y",
 "z", "u", "v", "xy", "out", "mu")
 rm(list=oldstuff, envir = .GlobalEnv)
 options(warn = 0)

}
```

- it has the `options(warn=-1)` command because many of these names will not be present when I run the `cleanup` command, and each of those would result in a warning. I already know this, so I don't need the warning. At the end I reset to the default with `options(warn = 0)`
- notice the empty line before the end of the function. This means there will be no return result. Without it the routine would return `NULL`.
- notice the argument `envir`. We will talk about what that means soon.

### 3.9 Naming Conventions

It is good practice to have a consistent style when choosing names for objects, functions etc. There is a detailed description at <https://google.github.io/styleguide/Rguide.xml>. Here are

the most important ones:

- File Names

File names should end in .R and, of course, be meaningful.

GOOD: predict\_ad\_revenue.R

BAD: foo.R

- Identifiers

Don't use underscores ( \_ ) or hyphens ( - ) in identifiers. Identifiers should be named according to the following conventions.

The preferred form for variable names is all lower case letters and words separated with dots.

GOOD: avg.clicks

BAD: avg\_Clicks

Make function names verbs.

GOOD: calculate.avg.clicks

BAD: clicks

constants are named like functions but with an initial k.

```
kbins <- 20
```

```
krun <- 1000
```

- Spacing

Place spaces around all binary operators (=, +, -, <-, etc.). Exception: Spaces around ='s are optional when passing parameters in a function call.

GOOD: x <- 1

BAD: x<-1

- Do not place a space before a comma, but always place one after a comma.

GOOD: x <- c(1, 2, 3)

BAD: x <- c(1,2,3)

These conventions always have a bit of personal style, which one might like or not. Here is another style rule:

Place a space before left parenthesis, except in a function call.

GOOD: if (debug)

BAD: if(debug)

Now personally I really don't like this one and I don't use it!

Instead of the . between words many people prefer to use camel back:

calculateMeans

That is just fine, I would recommend however you stick with one or the other.

## 4 Basic Data Types

Everything in R is an object. All objects have two intrinsic attributes: mode and length. The mode is the basic type of the elements of the object; there are four main modes:

- numeric
- character
- complex
- logical (FALSE or TRUE)

Other modes exist but they do not represent data, for instance function or expression. The length is the number of elements of the object. To display the mode and the length of an object, one can use the functions *mode* and *length*, respectively:

```
x <- 1; mode(x)

[1] "numeric"
y <- "A"; mode(y)

[1] "character"
z <- TRUE; mode(z)

[1] "logical"
```

R is quite different from most computer languages in that it often tries to figure out what you might want to do, even if it is not obvious. For example R can handle some strange calculations:

```
1/0

[1] Inf
0/0

[1] NaN
```

Here *Inf* is of course infinite, and *NaN* stands for *not a number*. These can be used in calculations:

```
exp(-Inf)
```

```
[1] 0
```

Numeric comes in two forms, integer and double. If you want to make sure an object is an integer use

```
n <- 2L
is.integer(n)

[1] TRUE
```

Years ago this was very useful because integers require much less storage space. These days with gigabyte sized memory it is rarely needed.

R can also handle complex numbers:

```
z <- 1i
u <- 1+1i
v <- 1-1i
z^2

[1] -1+0i
u+v

[1] 2+0i
u*v

[1] 2+0i
```

Objects of type character are identified with quotes:

```
y <- "A"
```

sometimes you want the " to be treated as a character. This can be done with the *escape character* \:

```
"color=\"red\""
[1] "color=\"red\""
```

#### 4.0.1 Vectors

the basic data unit of R is a vector. One can create a vector with the *combine* command:

```
x <- c(3, 5, 6, 3, 4, 5)
x
```

```
[1] 3 5 6 3 4 5
```

If you want a vector of characters again use quotes:

```
x <- c("A", "A", "B", "C")
x
```

```
[1] "A" "A" "B" "C"
```

for logical:

```
x <- c(FALSE, FALSE, TRUE)
x
```

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

note that there are no quotes. “FALSE” would be the word *FALSE*, not the logical value.

Note: this also works:

```
x <- c(F, F, T)
x
[1] FALSE FALSE TRUE
```

but I recommend writing FALSE and TRUE because sometimes F and T are used for other things (F=Female)

the symbol R uses for missing values is NA (not available). Again, no quotes:

```
x <- c(3, 5, NA, 3, 4, 5)
x
```

```
[1] 3 5 NA 3 4 5
```

Sometimes you want to create an object without any value:

```
x <- NULL
x
```

```
NULL
```

```
c(x, 1)
```

```
[1] 1
```

and note, the NULL is gone! This is useful when we are building up a vector (maybe inside a function), but we don't know ahead of time how large it will be.

Finally, dates and times are always tricky:

```
Sys.time()
```

```
[1] "2018-08-12 10:58:16 -04"
```

```
Sys.Date()
```

```
[1] "2018-08-12"
```

## 4.1 Type Conversion

Consider the following:

```
x <- c(3, 5, 6, 3, "A", 5)
x
[1] "3" "5" "6" "3" "A" "5"
```

in this case the vector is a mixture of numeric and character. But R vectors can never be such a mixture, so R (by itself!) decides to make it a character vector. This is called *type conversion*, and R does a lot of this, usually in a good way.

There are a number of routines that

- a. test for a data type

b. convert to a data type

they either start with *is.* or with *as.:*

```
x <- c(3, 5, NA, 3, 4, 5)
is.numeric(x)

[1] TRUE

y <- c(2, 1, 5, 2)
y

[1] 2 1 5 2

x <- as.character(y)
x

[1] "2" "1" "5" "2"

as.numeric(x)

[1] 2 1 5 2

x <- c("2", "1", "#", "2", "A")
as.numeric(x)

[1] 2 1 NA 2 NA
```

Consider this:

```
x <- c(1, 2, 5, FALSE, 4, TRUE)
x

[1] 1 2 5 0 4 1

as.character(x)

[1] "1" "2" "5" "0" "4" "1"
```

so FALSE gets turned into 0, TRUE into 1.

But also

```
x <- c("1", "2", "5", FALSE, "4", TRUE)
x

[1] "1" "2" "5" "FALSE" "4" "TRUE"

as.numeric(x)

[1] 1 2 5 NA 4 NA
```

Here FALSE gets first turned into a character, and then stays as such.

R has almost 100 is. and as. functions built in!

## 4.2 Dates:

The default format is yyyy-mm-dd:

```
mydates <- as.Date(c("2018-01-01", "2018-06-13"))
mydates
[1] "2018-01-01" "2018-06-13"
mydates[2]-mydates[1]
Time difference of 163 days
```

## 4.3 Factor

a very common data type in Statistics is a factor. These are vectors with a fixed number of different values (called levels) and possibly an ordering.

Here is an example of their usage. Say we have a list of students, identified by their year:

```
students
[1] "Junior" "Sophomore" "Sophomore" "Senior" "Junior"
[6] "Senior" "Senior" "Sophomore" "Sophomore" "Junior"
```

Let's count how many of each we have:

```
table(students)
students
Junior Senior Sophomore
3 3 4
```

there are two problems with this table:

- a. the ordering is wrong
- b. the Freshman class is missing.

We can fix both of these by turning the vector into a factor:

```
students.fac <- factor(students,
 levels=c("Freshman", "Junior", "Sophomore", "Senior"),
 ordered=TRUE)
table(students.fac)

students.fac
Freshman Junior Sophomore Senior
0 3 4 3
```

Here is another difference:

```
c(students, "Senior")
```

```

[1] "Junior" "Sophomore" "Sophomore" "Senior" "Junior"
[6] "Senior" "Senior" "Sophomore" "Sophomore" "Junior"
[11] "Senior"

c(students, "Graduate")

[1] "Junior" "Sophomore" "Sophomore" "Senior" "Junior"
[6] "Senior" "Senior" "Sophomore" "Sophomore" "Junior"
[11] "Graduate"

c(students.fac, "Senior")

[1] "2" "3" "3" "4" "2" "4" "4"
[8] "3" "3" "2" "Senior"

c(students.fac, "Graduate")

[1] "2" "3" "3" "4" "2" "4" "4"
[7] "4" "3" "3" "2" "Graduate"

```

so we can easily add an element to students, but if we do the same with student.fac it gets very confused! Strangely enough, it doesn't even work when the added item is in the list of levels!

Notice also that with c(students, "Senior") we don't get the lsit of students but a list of numbers (as characters) plus "Senior". This is because internally R stores factors as integers, but when we add "Senior" these get converted to character.

Here is what you can do:

```

lvls <- levels(students.fac)
x <- factor(c(as.character(students.fac), "Senior", "Graduate"),
 levels=c(lvls, "Graduate"), ordered=TRUE)
x

[1] Junior Sophomore Sophomore Senior Junior Senior Senior
[8] Sophomore Sophomore Junior Senior Graduate
Levels: Freshman < Junior < Sophomore < Senior < Graduate

table(x)

x
Freshman Junior Sophomore Senior Graduate
0 3 4 4 1

```

## 5 Data Formats: Vectors, Matrices, Arrays, Dataframes and Lists

### 5.1 Vectors

here are some useful commands for vectors:

```
x <- c(1, 2, 3, 4, 5, 6)
x
```

```
[1] 1 2 3 4 5 6
length(x)

[1] 6
names(x) <- LETTERS[1:6]
x
```

```
A B C D E F
1 2 3 4 5 6
```

If we have several vectors of the same type and length which belong together we can put them in a

## 5.2 Matrix

```
x <- c(1, 2, 3, 4, 5, 6)
y <- c(4, 2, 3, 5, 3, 4)
z <- c(3, 4, 2, 3, 4, 2)
cbind(x, y, z)
```

```
x y z
[1,] 1 4 1
[2,] 2 2 2
[3,] 3 3 3
[4,] 4 5 4
[5,] 5 3 5
[6,] 6 4 6
rbind(x, y, z)
```

```
[,1] [,2] [,3] [,4] [,5] [,6]
x 1 2 3 4 5 6
y 4 2 3 5 3 4
z 3 4 2 3 4 2
```

here are several other ways to make a matrix:

```
matrix(x, 2, 3)
```

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

```
matrix(x, ncol=2)
```

```
[,1] [,2]
```

```

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

matrix(0, 3, 3)

[,1] [,2] [,3]
[1,] 0 0 0
[2,] 0 0 0
[3,] 0 0 0

diag(3)

```

```

[,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1

```

we can control how the matrix is filled in:

```

matrix(1:8, nrow=2)

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

matrix(1:8, nrow=2, byrow = TRUE)

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

```

just like vectors a matrix is all of the same type, with R doing type conversion if necessary:

```

matrix(c(1, 2, "A", 3), 2, 2)

[,1] [,2]
[1,] "1" "A"
[2,] "2" "3"

```

useful commands for R matrices are:

```

x <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)

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

dim(x)

[1] 2 3

```

```

nrow(x)

[1] 2

ncol(x)

[1] 3

dimnames(x) <- list(c("A", "B"), c("a", "b", "c"))

x

a b c
A 1 3 5
B 2 4 6

colnames(x)

[1] "a" "b" "c"

rownames(x)

[1] "A" "B"

colnames(x) <- c("Height", "Age 1", "Age 2")

x

Height Age 1 Age 2
A 1 3 5
B 2 4 6

```

Actually I would recommend

```

colnames(x) <- c("Height", "Age.1", "Age.2")

x

Height Age.1 Age.2
A 1 3 5
B 2 4 6

```

empty spaces in names are allowed but can on occasion lead to problems, so avoiding them is a good idea. Of course they are not nice as labels in graphs or as titles in tables, but it is usually easy to change those.

Here is a strange way to make a matrix:

```

x <- 1:6

x

[1] 1 2 3 4 5 6

dim(x) <- c(2, 3)

x

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

```

What happens if we try to make a matrix out of a vector that doesn't have the right number of entries?

```
matrix(1:5, 2, 2)

Warning in matrix(1:5, 2, 2): data length [5] is not a sub-multiple or
multiple of the number of rows [2]

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

not surprisingly it just uses the elements needed.

```
matrix(1:3, 2, 2)

Warning in matrix(1:3, 2, 2): data length [3] is not a sub-multiple or
multiple of the number of rows [2]

[,1] [,2]
[1,] 1 3
[2,] 2 1
```

In this case R just starts all over again. This is called *recycling*.

In either case R gives a warning.

### 5.3 Arrays

an array is a k-dimensional matrix. For example

```
array(1:8, dim=c(2, 2, 2))

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

Often these come about as a 3-way table:

```
x <- sample(1:3, size=20, replace=T)
y <- sample(c("F", "M"), size=20, replace = T)
z <- sample(c("a", "b", "c"), size=20, replace = T)
table(x,y,z)
```

```

, , z = a
##
y
x F M
1 1 1
2 3 0
3 1 1
##
, , z = b
##
y
x F M
1 1 0
2 2 1
3 1 1
##
, , z = c
##
y
x F M
1 2 1
2 3 0
3 0 1

```

## 5.4 Dataframe

sometimes we have several vectors of the same length but of different types, then we can put them together as a data frame:

```

x <- c(1, 2, 3, 4, 5, 6)
y <- c("a", "a", "b", "c", "a", "c")
z <- c(T, T, F, T, F, T)
xyz <- data.frame(x, y, z)
xyz

```

```

x y z
1 1 a TRUE
2 2 a TRUE
3 3 b FALSE
4 4 c TRUE
5 5 a FALSE
6 6 c TRUE

```

the same commands as for matrices work for data frames as well:

```
dim(xyz)
```

```
[1] 6 3
```

```

nrow(xyz)

[1] 6

ncol(xyz)

[1] 3

dimnames(xyz) <- list(letters[1:6], c("a", "b", "c"))

xyz

a b c
a 1 a TRUE
b 2 a TRUE
c 3 b FALSE
d 4 c TRUE
e 5 a FALSE
f 6 c TRUE

colnames(xyz)

[1] "a" "b" "c"

rownames(xyz)

[1] "a" "b" "c" "d" "e" "f"

```

Finally, if the vectors aren't even of the same lengths we have

## 5.5 Lists

```

x <- c(1, 2, 3, 4, 5, 6)
y <- c("a", "a", "b")
z <- c(T, T)
xyz <- list(x, y, z)
xyz

[[1]]
[1] 1 2 3 4 5 6
##
[[2]]
[1] "a" "a" "b"
##
[[3]]
[1] TRUE TRUE

```

lists are displayed quite differently from the other formats. Here are a number of commands:

```
length(xyz)
```

```

[1] 3
names(xyz) <- c("Count", "Letter", "Married?")
xyz

$Count
[1] 1 2 3 4 5 6
##
$Letter
[1] "a" "a" "b"
##
$`Married?`
[1] TRUE TRUE

```

## 6 Generating Vectors

There are numerous ways to generate vectors which have some structure. Here are some useful commands:

The easiest way to do this is to use the *c* (concatenate) command:

```
x <- c(0, 2, 3, 1, 5)
x
```

```
[1] 0 2 3 1 5
```

to make sequences use :

```
1:5
[1] 1 2 3 4 5
-2:2
[1] -2 -1 0 1 2
```

and then we can combine these:

```
c(1:5, 11:15)
[1] 1 2 3 4 5 11 12 13 14 15
```

there are also a number of commands for this purpose:

- seq

```
seq(1, 10, 1)
[1] 1 2 3 4 5 6 7 8 9 10
seq(1, 10, 1/2)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5
```

```

[15] 8.0 8.5 9.0 9.5 10.0
seq(0, 10, length=20)

[1] 0.000000000000000 0.526315789473684 1.052631578947368
[4] 1.578947368421053 2.105263157894737 2.631578947368421
[7] 3.157894736842105 3.684210526315789 4.210526315789473
[10] 4.736842105263158 5.263157894736842 5.789473684210526
[13] 6.315789473684211 6.842105263157895 7.368421052631579
[16] 7.894736842105263 8.421052631578947 8.947368421052632
[19] 9.473684210526315 10.000000000000000

```

- sequence

this creates a series of sequences of integers each ending by the numbers given as arguments:

```
sequence(10)
```

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

```
sequence(c(2, 5, 3))
```

```
[1] 1 2 1 2 3 4 5 1 2 3
```

- rep

```
rep(1, 10)
```

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

```
rep(1:3, 10)
```

```
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(1:3, each=3)
```

```
[1] 1 1 1 2 2 2 3 3 3
```

```
rep(c("A", "B", "C"), c(4, 7, 3))
```

```
[1] "A" "A" "A" "A" "B" "B" "B" "B" "B" "B" "B" "B" "C" "C" "C"
```

what does this do?

```
rep(1:10, 1:10)
```

- gl

The function *gl* (generate levels) is very useful because it generates regular series of factors. The usage of this function is *gl(k, n)* where k is the number of levels (or classes), and n is the number of replications in each level.

Two options may be used: length to specify the number of data produced, and labels to specify the names of the levels of the factor.

```

gl(3, 5)

[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3

gl(3, 5, length=30)

[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3

gl(2, 6, label=c("Male", "Female"))

[1] Male Male Male Male Male Male Female Female Female Female
[11] Female Female
Levels: Male Female

gl(2, 10)

[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels: 1 2

gl(2, 1, length=20)

[1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels: 1 2

```

as we saw earlier, factors are represented by integers.

- expand.grid

this takes a couple of vectors and writes them as a matrix with each combination.

```
expand.grid(1:2, 1:3)
```

```

Var1 Var2
1 1 1
2 2 1
3 1 2
4 2 2
5 1 3
6 2 3

expand.grid(First=1:2, Second=1:3, Third=c("A", "B"))

```

```

First Second Third
1 1 1 A
2 2 1 A
3 1 2 A
4 2 2 A
5 1 3 A
6 2 3 A
7 1 1 B
8 2 1 B

```

```

9 1 2 B
10 2 2 B
11 1 3 B
12 2 3 B

```

there are a number of R routines who need the data in this format as arguments, so this is an easy way to convert them.

## 6.1 Expressions

up to now we discussed how to generate data objects. Soon we will be talking about how to write your own functions. There is however also a type of object somewhat in between, so called expressions.

An expression is a series of characters which make sense for R. All valid commands are expressions. When a command is typed directly on the keyboard, it is then evaluated by R and executed if it is valid. In many circumstances, it is useful to construct an expression without evaluating it: this is what the function expression is made for. It is, of course, possible to evaluate the expression subsequently with eval().

```

x <- 3; y <- 2.5; z <- 1
exp1 <- expression(x / (y + exp(z)))
exp1

```

```

expression(x/(y + exp(z)))
eval(exp1)

```

```

[1] 0.5749018735705

```

Expressions can be used for many things. Here are two examples:

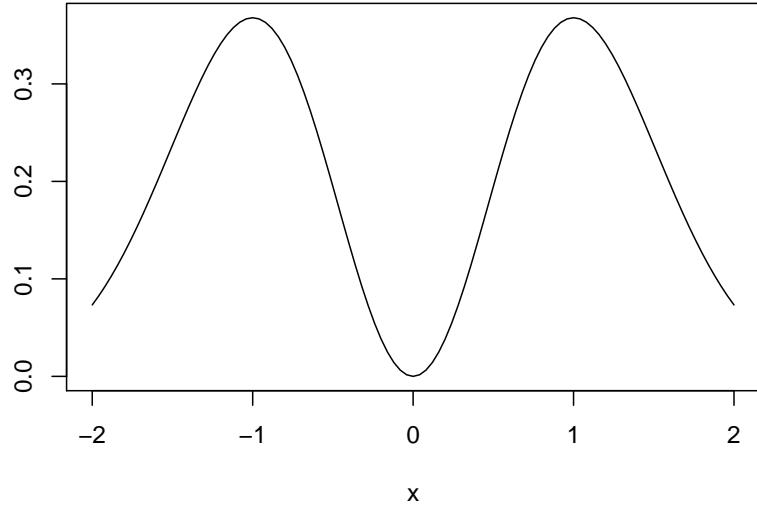
1. I want to draw the graph of a function, including the equation:

```

curve(x^2*exp(-x^2), -2, 2, main=expression(x^2*exp(-x^2)), ylab="")

```

$$x^2 \exp(-x^2)$$



2. Do symbolic math. For derivatives we have the function  $D$ , which returns partial derivatives:

```
D(exp1, "x")
1/(y + exp(z))
eval(D(exp1, "x"))

[1] 0.191633957856833
D(exp1, "y")
-(x/(y + exp(z))^2)
eval(D(exp1, "y"))

[1] -0.110170721411624
D(exp1, "z")
-(x * exp(z)/(y + exp(z))^2)
eval(D(exp1, "z"))

[1] -0.299475070041441
```

In general R's use as a symbolic language is very limited. There are of course many purpose built languages for this, such as Maple or Mathematica.

## 7 Subsetting / Data Wrangling

### 7.1 Vectors

Consider the following vector:

```
x
A B C D E F G H I J
5.7 0.5 1.5 4.7 6.4 9.2 3.5 4.8 1.9 1.2
```

The elements of a vector are accessed with the bracket [ ] notation:

```
x[3]
C
1.5
x[1:3]
A B C
5.7 0.5 1.5
x[c(1, 3, 8)]
A C H
5.7 1.5 4.8
x[-3]
A B D E F G H I J
5.7 0.5 4.7 6.4 9.2 3.5 4.8 1.9 1.2
x[-c(1, 2, 5)]
C D F G H I J
1.5 4.7 9.2 3.5 4.8 1.9 1.2
x["C"]
C
1.5
x[c("A", "D")]
A D
5.7 4.7
```

Another way to subset a vector is with logical conditions:

```
x[x > 4]
A D E F H
5.7 4.7 6.4 9.2 4.8
```

```
x[x>4 & x<7]
```

```
A D E H
5.7 4.7 6.4 4.8
```

It is also possible to replace values in a vector this way:

```
x[x<2] <- 0
x
```

```
A B C D E F G H I J
5.7 0.0 0.0 4.7 6.4 9.2 3.5 4.8 0.0 0.0
```

This can be useful, for example to code a variable:

Gender

```
[1] "Male" "Female" "Female" "Female" "Male" "Female" "Female"
[8] "Female" "Male" "Female"
GenderCode <- rep(0, length(Gender))
GenderCode[Gender=="Male"] <- 1
GenderCode

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

### 7.1.1 Matrices and Data Frames

Consider the following data frame:

students

```
Age GPA Gender
1 22 3.1 Male
2 23 3.2 Male
3 20 2.1 Male
4 22 2.1 Male
5 21 2.3 Female
6 21 2.9 Male
7 18 2.3 Female
8 22 3.9 Male
9 21 2.6 Female
10 18 3.2 Female
```

Because a vector has rows and columns we now need to specify both:

```
students[2, 3]
```

```
[1] "Male"
```

There are a variety of ways to do subsetting:

```
students[, 1]
[1] 22 23 20 22 21 21 18 22 21 18
students[[1]]
[1] 22 23 20 22 21 21 18 22 21 18
students$Age
[1] 22 23 20 22 21 21 18 22 21 18
```

And yet another way to do this:

```
attach(students)
```

```
Age
```

```
[1] 22 23 20 22 21 21 18 22 21 18
```

Although these seem to do the same there actually subtle differences. Consider this:

```
students[, 1]
```

```
[1] 22 23 20 22 21 21 18 22 21 18
```

```
students[1]
```

```
Age
1 22
2 23
3 20
4 22
5 21
6 21
7 18
8 22
9 21
10 18
```

```
students[[1]]
```

```
[1] 22 23 20 22 21 21 18 22 21 18
```

In the first and last case R returns a vector, in the second case a data frame with one column.

It is possible to tell R not to do this type conversion in the first case

```
students[, 1, drop=FALSE]
```

```
Age
1 22
2 23
3 20
4 22
```

```
5 21
6 21
7 18
8 22
9 21
10 18
```

but this does not work for the [[1]] or \\$Age versions.

```
students[1:3, 1]
```

```
[1] 22 23 20
```

```
students[-2,]
```

```
Age GPA Gender
1 22 3.1 Male
3 20 2.1 Male
4 22 2.1 Male
5 21 2.3 Female
6 21 2.9 Male
7 18 2.3 Female
8 22 3.9 Male
9 21 2.6 Female
10 18 3.2 Female
```

```
students[1:4, -1]
```

```
GPA Gender
1 3.1 Male
2 3.2 Male
3 2.1 Male
4 2.1 Male
```

```
students[Age>20,]
```

```
Age GPA Gender
1 22 3.1 Male
2 23 3.2 Male
4 22 2.1 Male
5 21 2.3 Female
6 21 2.9 Male
8 22 3.9 Male
9 21 2.6 Female
```

You can have several conditions, put together with & (AND), | (OR) and ! (NOT), but some care is needed:

```
students[Age>=20 & Age<=22, 1]
```

```
[1] 22 20 22 21 21 22 21
```

is fine but

```
students[20 <= Age <= 22, 1]
```

does not work.

### 7.1.2 Lists

Subsetting of lists is very similar to dataframes:

```
mylist <- list(First=1:5, Second=LETTERS[1:8], Third=20:22)
mylist[1]
```

```
$First
[1] 1 2 3 4 5
mylist[[1]]
```

```
[1] 1 2 3 4 5
mylist$Second
[1] "A" "B" "C" "D" "E" "F" "G" "H"
mylist[1:2]
```

```
$First
[1] 1 2 3 4 5

$Second
[1] "A" "B" "C" "D" "E" "F" "G" "H"
mylist[[1:2]]
```

```
[1] 2
```

so [1] returns a list with just one element whereas [[1]] and \$ do type conversion to a vector if possible. [1:2] yields the first two elements of the list.

The last one is strange, why is the result 2? Actually, it does this:

```
mylist[[1]][2]
```

```
[1] 2
```

## 7.2 Useful logic commands:

```
x<-1; y<-2; z<-3
any(c(x, y, z)>2.5)
[1] TRUE
```

```

all(c(x, y, z)>2.5)

[1] FALSE
x <- 1:3; y <- 1:3
x == y

[1] TRUE TRUE TRUE
identical(x, y)

[1] TRUE
all.equal(x, y)

```

## [1] TRUE

*identical* compares the internal representation of the data and returns TRUE if the objects are strictly identical, and FALSE otherwise.

*all.equal* compares the “near equality” of two objects, and returns TRUE or displays a summary of the differences. The latter function takes the approximation of the computing process into account when comparing numeric values. The comparison of numeric values on a computer is sometimes surprising!

```
0.9 == (1 - 0.1)
```

## [1] TRUE

```
identical(0.9, 1 - 0.1)
```

## [1] TRUE

```
all.equal(0.9, 1 - 0.1)
```

## [1] TRUE

but

```
0.9 == (1.1 - 0.2)
```

## [1] FALSE

```
identical(0.9, 1.1 - 0.2)
```

## [1] FALSE

```
all.equal(0.9, 1.1 - 0.2)
```

## [1] TRUE

How come  $1.1 - 0.2 \neq 0.9$ ? This is because of machine precision issues:

```
all.equal(0.9, 1.1 - 0.2, tolerance = 1e-16)
```

```
[1] "Mean relative difference: 1.2335811384724e-16"
```

### 7.3 *subset*

Finally there is a command that was written for subsetting:

```
subset(students, Age>20)
```

```
Age GPA Gender
1 22 3.1 Male
2 23 3.2 Male
4 22 2.1 Male
5 21 2.3 Female
6 21 2.9 Male
8 22 3.9 Male
9 21 2.6 Female
```

```
subset(students, Age>20 & Gender=="Male")
```

```
Age GPA Gender
1 22 3.1 Male
2 23 3.2 Male
4 22 2.1 Male
6 21 2.9 Male
8 22 3.9 Male
```

```
subset(students, Age>20, select = Gender)
```

```
Gender
1 Male
2 Male
4 Male
5 Female
6 Male
8 Male
9 Female
```

```
subset(students, Age>20, select = Gender, drop=TRUE)
```

```
[1] "Male" "Male" "Male" "Female" "Male" "Male" "Female"
```

Notice that this last one results in a vector.

## 8 Generate Random Variates

### 8.1 Random Numbers

Everything starts with generating  $X_1, X_2, \dots$  iid  $U[0,1]$ . These are simply called random numbers. There are some ways to get these:

- random number tables
- numbers taken from things like the exact (computer) time
- quantum random number generators
- ...

The R package *random* has the routine *randomNumbers* which gets random numbers from a web site which generates them based on (truly random) atmospheric phenomena.

```
require(random)
randomNumbers(20, 0, 100)

V1 V2 V3 V4 V5
[1,] 11 43 73 31 80
[2,] 49 93 84 90 43
[3,] 51 43 14 71 53
[4,] 81 26 21 92 71
```

## 8.2 Pseudo-Random Numbers

These are numbers that look random, smell random ...

Of course a computer can not do anything truly random, so all we can do is generate  $X_1, X_2, \dots$  that **appear** to be iid  $U[0,1]$ , so-called *pseudo-random numbers*. In R we have the function *runif*:

```
runif(5)

[1] 0.5742350621148944 0.0542048825882375 0.1531889955513179
[4] 0.4655727914068848 0.6441916297189891
```

or

```
round(runif(5, min=0, max=100), 1)
```

```
[1] 91.6 34.9 48.0 19.2 11.7
```

If we want to choose from a finite set we have

```
sample(letters, 5)
```

```
[1] "f" "p" "t" "m" "h"
```

if no number is given it yields a random permutation:

```
sample(1:10)
```

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

if we want to allow repetitions we can do this as well. Also, we can give the (relative) probabilities:

```
table(sample(1:5, size=1000, replace=TRUE,
 prob=c(1, 2, 3, 2, 1)))
```

```

1 2 3 4 5
111 224 334 225 106
```

Notice that the probabilities need not be normalized (aka add up to 1)

### 8.3 Standard Probability Distributions

many standard distributions are part of base R. For each the format is

- dname = density
- pname = cumulative distribution function
- rname = random generation
- qname = quantile function

#### 8.3.0.1 Example: Poisson distribution

```
dpois(c(0, 8, 12, 20), lambda=10)
```

```
[1] 4.53999297624849e-05 1.12599032149020e-01 9.47803300917677e-02
[4] 1.86608131399876e-03
```

```
ppois(c(0, 8, 12, 20), 10)
```

```
[1] 4.53999297624849e-05 3.32819678750719e-01 7.91556476394874e-01
[4] 9.98411739338142e-01
```

```
rpois(5, 10)
```

```
[1] 6 8 5 5 17
```

```
qpois(1:4/5, 10)
```

```
[1] 7 9 11 13
```

Here is a list of the distributions included with base R:

- beta distribution: dbeta.
- binomial (including Bernoulli) distribution: dbinom.
- Cauchy distribution: dcauchy.
- chi-squared distribution: dchisq.
- exponential distribution: dexp.

- F distribution: df.
- gamma distribution: dgamma.
- geometric distribution: dgeom.
- hypergeometric distribution: dhyper.
- log-normal distribution: dlnorm.
- multinomial distribution: dmultinom.
- negative binomial distribution: dnbinom.
- normal distribution: dnorm.
- Poisson distribution: dpois.
- Student's t distribution: dt.
- uniform distribution: dunif.
- Weibull distribution: dweibull.

With some of these a bit of caution is needed. For example, the usual textbook definition of the geometric random variable is as the number of tries in a sequence of independent Bernoulli trials until a success. This means that the density is defined as

$$P(X = x) = p(1 - p)^{x-1}, x = 1, 2, \dots$$

R however defines it as the number of failures until the first success, and so it uses

$$P(X^* = x) = \text{dgeom}(x, p) = p(1 - p)^x, x = 0, 1, 2, \dots$$

Of course this is easy to fix. If you want to generate the “usual” geometric do

```
x <- rgeom(10, 0.4) + 1
x
```

```
[1] 2 2 8 1 5 1 3 1 2 9
```

if you want to find the probabilities or cdf:

```
round(dgeom(x-1, 0.4), 4)
```

```
[1] 0.2400 0.2400 0.0112 0.4000 0.0518 0.4000 0.1440 0.4000 0.2400 0.0067
round(0.4*(1-0.4)^(x-1), 4)
```

```
[1] 0.2400 0.2400 0.0112 0.4000 0.0518 0.4000 0.1440 0.4000 0.2400 0.0067
```

Another example is the Gamma random variable. Here most textbooks use the definition

$$f(x; \alpha, \beta) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta}$$

but R uses

$$f^*(x; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

```
dgamma(1.2, 0.5, 2)

[1] 0.0660758383285821

2^0.5/gamma(0.5)*1.2^(0.5-1)*exp(-2*1.2)
```

```
[1] 0.0660758383285821
```

Again, it is easy to *re-parametrize*:

```
dgamma(1.2, 0.5, 1/(1/2))
```

```
[1] 0.0660758383285821
```

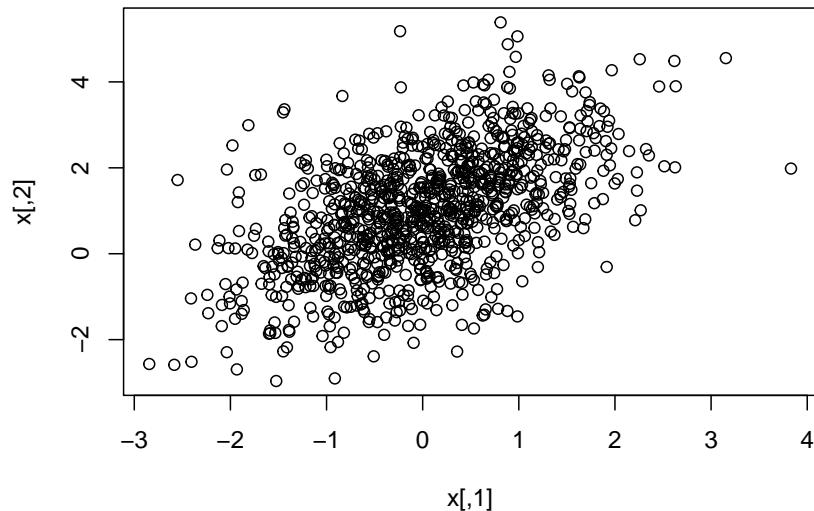
## 8.4 Other Variates

if you need to generate random variates from a distribution that is not part of base R you should first try to find a package that includes it.

**Example** multivariate normal

there are actually several packages, the most commonly used one is *mvtnorm*

```
library(mvtnorm)
x <- rmvnorm(1000, mean=c(0, 1), sigma=matrix(c(1, 0.8, 0.8, 2), 2, 2))
plot(x)
```



```
cor(x)[1, 2]
[1] 0.526416238173195
```

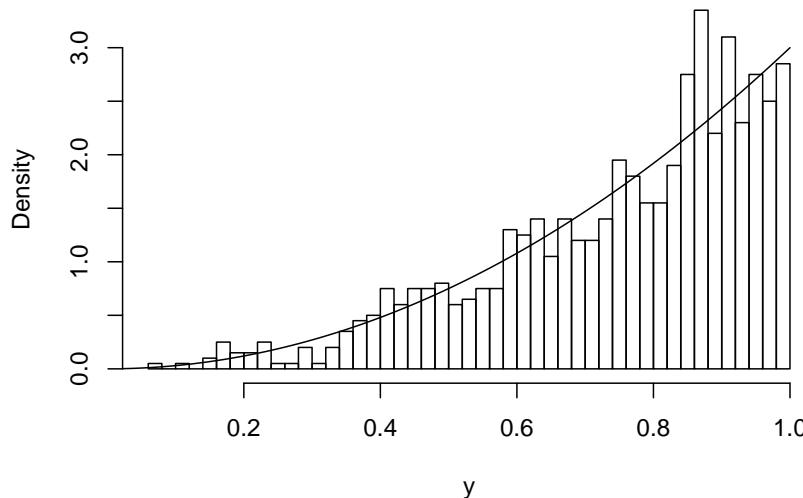
If you can't find a package you have to write your own! Here is a simple routine that will generate random variates from any function *fun* (given as a character vector) in one dimension on a finite interval  $[A, B]$ :

```
r.pit <- function (n, fun, A, B)
{
 f <- function(x) eval(parse(text=fun))
 m <- min(2 * n, 1000)
 x <- seq(A, B, length = m)
 y <- f(x)
 z <- (x[2] - x[1]) / 6 * cumsum((y[-1] + 4 * y[-2] + y[-3]))
 z <- z / max(z)
 y <- c(0, z)
 xyTmp <- cbind(x, y)
 approx(xyTmp[, 2], xyTmp[, 1], runif(n))$y
}
```

*pit* stands for *probability integral transform*, which is a theorem in probability theory that explains why this works.

Let's try it out:

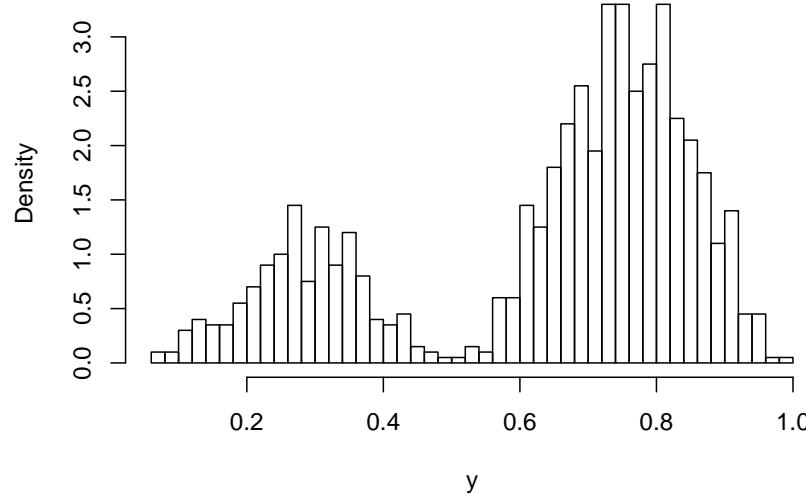
```
y <- r.pit(1000, "x^2", 0, 1)
hist(y, 50, freq=FALSE, main="")
curve(3*x^2, 0, 1, add=TRUE)
```



notice that the function doesn't even have to be normalized!

or a bit more complicated:

```
y <- r.pit(1000, "x*sin(2*pi*x)^2", 0, 1)
hist(y, 50, freq=FALSE, main="")
```

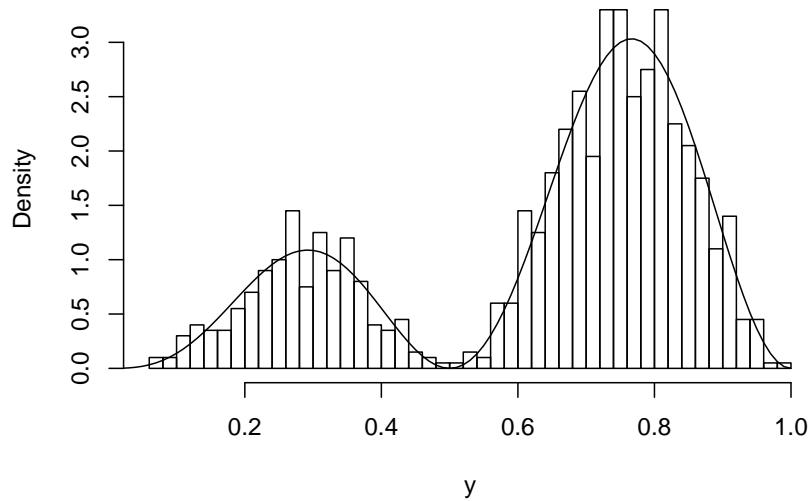


How about adding the density curve? For that we do need to normalize the function, that is we need to make sure that

$$\int_0^1 x \sin(2\pi x)^2 dx = 1$$

but this is not a trivial integral, so we need to use a numerical method:

```
x <- seq(0, 1, length=250)
f <- x*sin(2*pi*x)^2
I <- sum((f[-1]+f[-250])/2 * (x[-1]-x[-250])) #Riemann sum
hist(y, 50, freq=FALSE, main="")
curve(x*sin(2*pi*x)^2/I, 0, 1, add=TRUE)
```



Want to learn how to generate data from any random vector? Come to my course ESMA5015 Simulation!

## 9 Writing your own functions

R has several ways to write your own function:

- RStudio: click on File > New File > R Script. A new empty window pops up. Type fun, hit enter, and the following text appears:

```
name <- function(variables) {
}
```

change the name, save the file as myfun.R with File > Save. Now type in the code. When done click the Source button.

- fix: In the R console run

```
fix(myfun)
```

now a window with an editor pops up and you can type in the code. When you are done click on Save. If there is some syntax error DON'T run fix again, instead run

```
myfun <- edit()
```

- Open any code editor outside of RStudio, type in the code, save it as myfun.R, go to the console and run

```
source('R/myfun.R')
```

Which of these is best? In large part that depends on your preferences. In my case, if I expect to need that function just for a bit I use the fix option. If I expect to need that function again later I start with the first method, but likely soon open the .R file outside RStudio because I can move that window around onto a second monitor.

There are some useful keyboard shortcuts here. If the cursor is on some line in the RStudio editor you can hit

- CTRL-Enter run current line or section
- CTRL-ALT-B run from beginning to line
- CTRL-Shift-Enter run complete chunk
- CTRL-Shift-P rerun previous

## 9.1 Basic programming structures in R

R has all the standard programming features:

### 9.1.1 Conditionals if-else

```
f <- function(x) {
 if(x>0) y <- log(x)
 else y <- NA
 y
}
f(2)

[1] 0.693147180559945
f(-2)

[1] NA
```

A useful variation on the if statement is the *switch* statement:

```
centre <- function(x, type) {
 switch(type,
 mean = mean(x),
 median = median(x),
 trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
centre(x, "mean")

[1] -7.0911552223987
centre(x, "median")

[1] -0.523917057536003
```

```

centre(x, "trimmed")

[1] -0.193166033617042
special R construct: ifelse
x <- sample(1:10, size=7, replace = TRUE)
x

[1] 8 1 4 5 7 8 9
ifelse(x<5, "Yes", "No")

[1] "No" "Yes" "Yes" "No" "No" "No" "No"

```

### 9.1.2 Loops

there are three standard loops in R:

#### 9.1.2.1 for loop

```

y <- vector("integer", 10)
for(i in 1:10) y[i] <- i*(i+1)/2
y

[1] 1 3 6 10 15 21 28 36 45 55

```

sometimes we don't know the length of y ahead of time, then we can use

```

for(i in seq_along(y)) y[i] <- i*(i-1)^2
y

[1] 0 2 12 36 80 150 252 392 576 810

```

If there is more than one statement inside a loop use curly braces:

```

for(i in seq_along(y)) {
 y[i] <- i*(i-1)^2
 if(y[i]>100) y[i] <- 100
}
y

[1] 0 2 12 36 80 100 100 100 100 100

```

#### 9.1.2.2 while loop

this is useful if we don't know how often the loop needs to run.

Let's say we want to do a simulation of rolling three dice and we want to generate the event “number of repetitions needed until a triple” (triple = all three dice equal). If so x has the equal entries, so table(x) has just one:

```

k <- 1
x <- sample(1:6, size=3, replace=TRUE)
while (length(table(x))!=1) {
 k <- k+1
 x <- sample(1:6, size=3, replace=TRUE)
}
k

[1] 4

```

### 9.1.2.3 repeat loop

similar to while loop, except that check is done at the end

```

k <- 0
repeat {
 k <- k+1
 x <- sample(1:6, size=3, replace=TRUE)
 if(length(table(x))==1) break
}
k

[1] 35

```

Notice that a while and repeat loop could in principle run forever. I usually include a counter that ensures the loop will eventually stop:

```

k <- 0
counter <- 0
repeat {
 k <- k+1
 counter <- counter+1
 x <- sample(1:6, size=3, replace=TRUE)
 if(length(table(x))==1 | counter>1000) break
}
k

[1] 53

```

## 9.2 Some general advice on computer programming

- Start small

Even if ultimately you want to write a “big” program, first write a small one, a special case.

- then add to it, piece by piece
- test often

and don’t add more until you have a working program

- break it up in pieces (modularity)

If your program gets too big, break it up into several smaller self-contained programs

- Comment, Comment, Comment

write lots of comments. This is not only so that someone else can understand your program, but so that your future-self can as well!

### 9.3 Finding Errors, Debugging

There are essentially two type of errors:

- syntax
- run-time

A syntax error is a bad piece of code, such as missing braces etc. You will get an error message from R when you try to read such code into R, for example with source. In general these are fairly easy to find, R will tell you the line where (more or less) the error is.

- run-time errors occur when at some point in the execution an illegal command is to be executed. If that happens R generally stops the program and returns an error.

Finding a run-time error can be very time consuming. Here are some simple things to do:

- add print statements

In order to find out where in the execution a problem occurs you might add a print statement in a few places. Sometime that should print out some values of interest, some times it might just be print("A") print("B") etc.

- when an error occurs depending on a random number, use set.seed so it will always occur at the same spot:

```
set.seed(1111)
```

- R has some routines to help with debugging, such traceback, debug and browser. For some explanations of these read <https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio>

### 9.4 Functional Programming

In R everything is an object, and objects can be passed to and returned from functions. This includes functions!

Consider the example from above:

```
centre <- function(x, type) {
 switch(type,
 mean = mean(x),
 median = median(x),
 trimmed = mean(x, trim = .1))
```

```

}

x <- rcauchy(10)
c(centre(x, "mean"), centre(x, "median"), centre(x, "trimmed"))

```

```
[1] 0.539739076299199 0.154889841858633 0.379316727207262
```

We could just as easily have written it this way:

```

centre.alt1 <- function(x, fun) {
 fun(x)
}
c(centre.alt1(x, mean), centre.alt1(x, median))

```

```
[1] 0.539739076299199 0.154889841858633
```

How about the last one, the trimmed mean? Here we also need to pass an argument (trim=0.1) on to function inside centre.alt. One way would be to define the function explicitly:

```
centre.alt1(x, function(x) {mean(x, trim = 0.1)})
```

```
[1] 0.379316727207262
```

Notice that here the function we created doesn't have a name. It is called an *anonymous* function.

There is another way to pass an argument on to a function, the ... syntax:

```

centre.alt2 <- function(x, fun, ...) {
 fun(x, ...)
}
centre.alt2(x, mean, trim=0.1)

```

```
[1] 0.379316727207262
```

Finally, a function can also be the output of a function:

```

centre.alt3 <- function(type) {
 switch(type,
 mean = mean,
 median = median,
 trimmed = function(x) {mean(x, trim = .1)})
}
c(centre.alt3("mean")(x), centre.alt3("median")(x),
 centre.alt3("trimmed")(x))

```

```
[1] 0.539739076299199 0.154889841858633 0.379316727207262
```

Sometimes it is better to return a list of functions:

```

centre.alt4 <- function(type) {
 list(Mean = mean,
 Median = median,
 Trimmed = function(x) {mean(x, trim = .1)})
}

```

```

}

c(centre.alt4()$Mean(x), centre.alt4()$Median(x),
 centre.alt4()$Trimmed(x))

[1] 0.539739076299199 0.154889841858633 0.379316727207262

```

#### 9.4.1 Turning a string into a function.

Let's say we want to write a function that reads some text from a file. The text is actually the name of an R function, and we want to execute that function. We can use the following (very famous or infamous!) construct:

```

txt <- "exp(-x^2)"
fun <- function(x) eval(parse(text=txt))
x <- 0.5
c(exp(-x^2), fun(x))

[1] 0.778800783071405 0.778800783071405

```

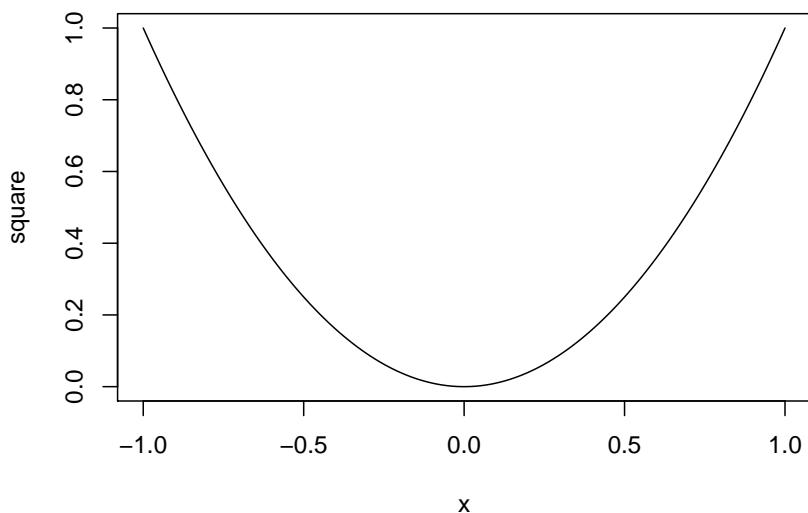
#### 9.4.2 Computing on the Language

Consider this example:

```

x <- seq(-1, 1, length=250)
square <- x^2
plot(x, square, type = "l")

```

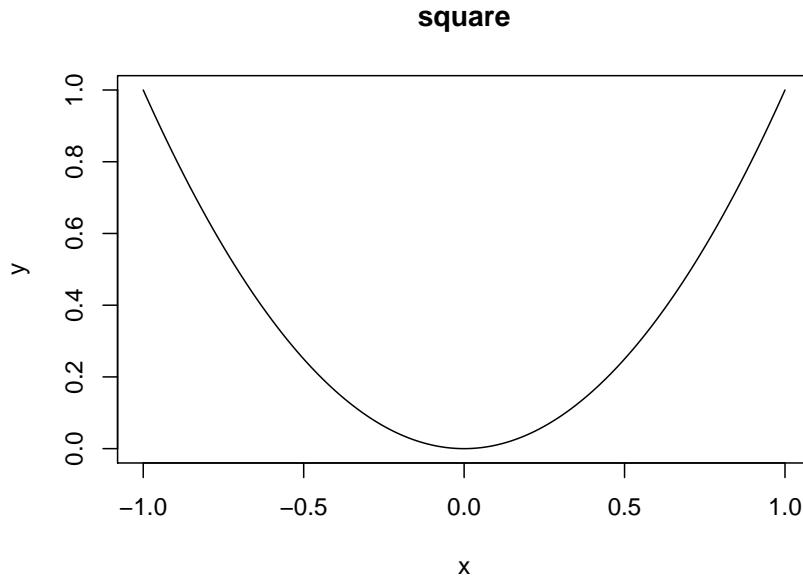


Notice something strange? In the graph we have not only the values of  $x$  and  $\text{square}$ , also the names “ $x$ ” and “ $\text{square}$ ”, used as labels. How is this done? Could we use the name of the

variables also (for example) in the title of the graph?

The R functions we need for this are *deparse* and *substitute*:

```
myplot <- function(x, y) {
 yname <- deparse(substitute(y))
 plot(x, y, type="l", main=yname)
}
myplot(x, square)
```



Notice the call to the function *rm*:

```
ls()

character(0)
x <- 1
ls()

[1] "x"

rm(x)
ls()

character(0)
x <- 1
rm("x")
ls()

character(0)
```

Apparently it does not matter whether we call *rm* with or without quotes, it works either way. There are a number such routines, for example *library*.

How does this work?

Let's write one:

```
f <- function(expr) {
 sexpr <- substitute(expr)
 if(!is.character(sexpr))
 sexpr <- deparse(sexpr)
 sexpr
}
f(z)
```

```
[1] "z"
```

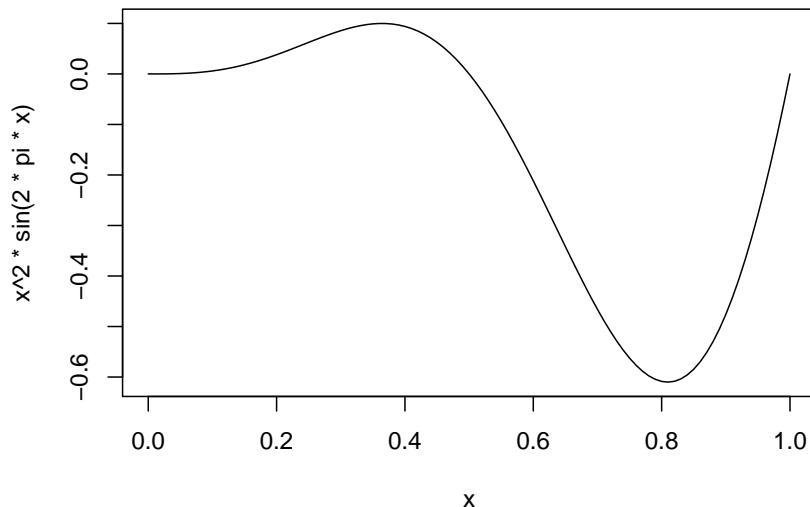
```
f("z")
```

```
[1] "z"
```

so either way now we have a character string.

We have previously used a nice function in R to draw graphs of functions called *curve*:

```
curve(x^2*sin(2*pi*x), 0, 1)
```



notice how the function is entered here, without quotes. So it is not a character string.

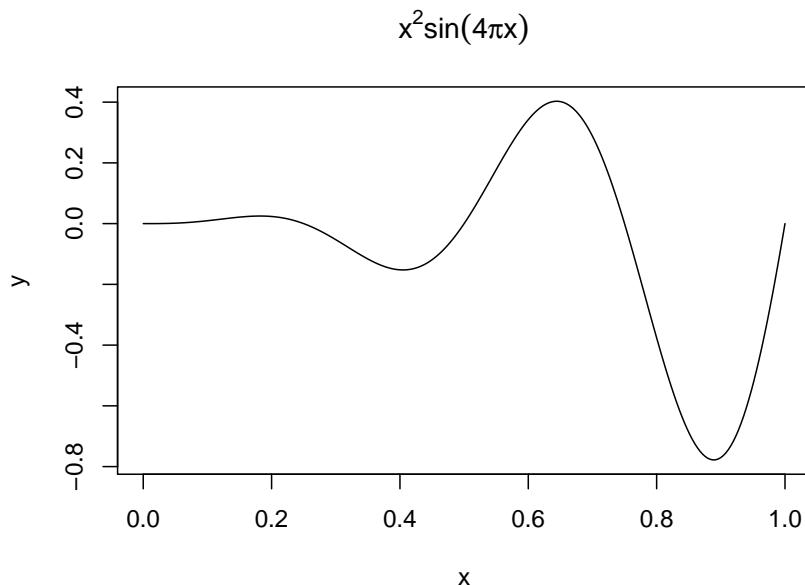
Let's write our own curve routine. In addition we are going to show the function as the title! Finally, our routine will also allow us to pass two parameters to the function.

```
mycurve <- function(fn, A=0, B=1, np=250, par1=0, par2=0) {
 sexpr <- deparse(substitute(fn)) #get function as character string
 x <- seq(A, B, length=np) #equal-spaced grid
 f <- function(x) eval(parse(text=sexpr),
```

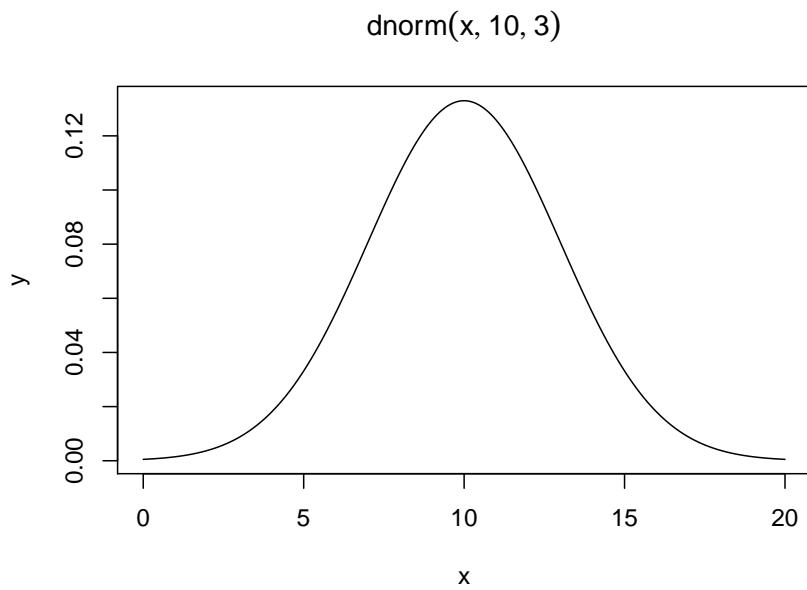
```

 list(x=x, par1=par1, par2=par2))
now as function, with x and parameters passed to it
y <- f(x)
sexpr <- gsub("par1", par1, sexpr) #replace text with value
sexpr <- gsub("par2", par2, sexpr) #but as text!
plot(x, y, type="l", main=parse(text=sexpr))
}
mycurve(x^par1*sin(par2*pi*x), par1=2, par2=4)

```



```
mycurve(dnorm(x, par1, par2), A=0, B=20, par1=10, par2=3)
```



## 9.5 Infix functions

In R every object is either a data set or a function. As a general principle, if it does something it is a function. This is true even for things that don't look that way, for example the addition operation `+`:

```
1 + 2
```

```
[1] 3
```

In fact we can write this explicitly as a function:

```
'+'(1, 2)
```

```
[1] 3
```

Because these types of functions are sitting between their arguments they are called *infix* operators, in contrast to *prefix* operators that have the name of the function first.

Because they are functions we can write our own. User-written prefix functions always have to start and end with %.

### 9.5.1 Case Study: Binary Arithmetic

In our everyday world we use 10 digits to express numbers, a system called decimal. There is however nothing special about 10 (except that we have 10 fingers and toes!) In fact, computers only use 2 digits, 0 and 1 (or electricity is on and off), a system called binary. Every decimal number has an equivalent binary number, and one can do arithmetic in either system.

Note that we will restrict ourselves to integers only, but in general this works for any real or even complex number.

Here are the first numbers:

$$\begin{aligned}0 &= 0 \\1 &= 1 \\2 &= 10 \\3 &= 11 \\4 &= 100 \\5 &= 101 \\6 &= 110 \\7 &= 111 \\8 &= 1000\end{aligned}$$

We want to implement a system that allows us to work with binary numbers in R.

First we need to figure out how to represent these numbers in R. The easiest way is as a sequence of 0's and 1's, like (1, 0, 1) (equivalent to 5).

We begin by writing some routines that convert a number in the decimal system to binary and vice versa:

How can we turn a decimal into a binary? Notice the following

$$\begin{aligned}0 &= 0 = 0 \times 2^0 \\1 &= 1 = 1 \times 2^0 \\2 &= 10 = 1 \times 2^1 + 0 \times 2^0 \\3 &= 11 = 1 \times 2^1 + 1 \times 2^0 \\4 &= 100 = 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\5 &= 101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\6 &= 110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\7 &= 111 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\8 &= 1000 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\end{aligned}$$

so the idea is to write x in the form  $\sum i_k 2^k$  where  $i_k \in \{0, 1\}$

Let's try an example:

$$26 = 2^4 + 2^3 + 2^1 = (1, 1, 0, 1, 0)$$

why does this start with  $2^4$ ? Because  $2^4$  is the largest power of 2 still less or equal to 26, or  $4 = \max \{i \in N : 2^i < 26\}$ .

For a general number m, how can we find out what this largest i is? We have

$$\begin{aligned}2^i &\leq m \\i \log(2) &\leq \log(m) \\i &\leq \log(m)/\log(2)\end{aligned}$$

and in fact if we use log base 2 we have

$$i = \text{floor}(\log(m, \text{base} = 2))$$

Once we have the first such  $i$ , we can simply subtract  $2^i$  from  $m$  and start all over again, until there is nothing left:

```
decimal.2.binary <- function(x) {
 if(x==0) return(0) #simple cases
 if(x==1) return(1)
 i <- floor(log(x, base=2)) #largest power of 2 less than x
 bin.x <- rep(1, i+1) #we will need i+1 0's and 1's,
 #first is always a 1
 x <- x-2^i #done with largest 2^i
 for(j in (i-1):0) {
 if(2^j>x)
 bin.x[j+1] <- 0
 else {
 bin.x[j+1] <- 1
 x <- x-2^j
 }
 }
 bin.x[length(bin.x):1]
}
decimal.2.binary(7)
```

```
[1] 1 1 1
decimal.2.binary(8)
```

```
[1] 1 0 0 0
decimal.2.binary(26)
```

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

Of course, the other way around is much simpler:

```
binary.2.decimal <- function(x) sum(x*2^(length(x):1-1))
binary.2.decimal(c(1, 1, 1))

[1] 7
binary.2.decimal(c(1, 0, 0, 0))

[1] 8
```

---

**Algorithm 3.1** Binary addition

---

**input:**  $N$ -bit augend  $a$  and addend  $b$ , and a 1-bit carry in  $c$

**output:**  $N$ -bit sum  $s$ , and a 1-bit carry out  $c$

```
function ADD(a, b, c)
 for $i \leftarrow 0$ to $N - 1$ do
 $n \leftarrow a_i + b_i + c$ ▷ two-bit sum
 $s_i \leftarrow n_0$
 $c \leftarrow n_1$
 end for
 return c, s ▷ carry out and sum
end function
```

---

Figure 1:

```
binary.2.decimal(c(1, 1, 0, 1, 0))
[1] 26
binary.2.decimal(decimal.2.binary(126))
[1] 126
```

How does addition work? essentially we add the numbers piece by piece from right to left, with a 1 carried over whenever there are two 1's. For example

$$\begin{array}{r} 0010 \quad (2) \\ + 0110 \quad (6) \\ = 1000 \quad (8) \end{array}$$

A general algorithm for addition is given here:

It uses some strange words, but those are not important to us. Here is the corresponding R function:

```
binary_addition <- function(x, y) {
#First make x and y of equal length and with one extra
#slot in case it's needed for carry over
#Fill x and y with 0's as needed.
 n <- length(x)
 m <- length(y)
 N <- max(n, m)+1
 x <- c(rep(0, N-n), x)
 y <- c(rep(0, N-m), y)
 s <- rep(0, N) # for result
 ca <- 0 #for carry over term
 for(i in N:1) {
 n <- x[i]+y[i]+ca
 if(n<=1) {#no carry over
```

```

 s[i] <- n
 ca <- 0
}
else {#with carry over
 s[i] <- 0
 ca <- 1
}
}
if(s[1]==0) s <- s[-1] #leading 0 removed if necessary
s
}
binary_addition(c(1, 0), c(1, 1, 0))

```

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

Let's turn this into an infix addition operator. we could just call it `%+%` but instead I will call it `%+b%` so the chances of this name already being used by R somewhere else are small.

```
'%+b%' <- function(x, y) binary_addition(x, y)
x <- c(1, 0) # 2
y <- c(1, 1, 0) # 6
x %+b% y #2+6=8
```

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

Notice a big problem with how we have defined binary numbers: how are we going to define a vector of them, as a vector of vectors? Lists might work, but would be a bit ugly. A better idea might be as a character string like "0101". But before we can handle that we will need to learn how to work with strings.

Let's write two more functions:

- 1) `is.binary` should check whether a vector is (can be) a binary number. For this it has to consist entirely of 0's and 1's:

```
is.binary <- function(x) {
 if(all(x==0)) return(TRUE)
 x <- x[x!=0]
 x <- x[x!=1]
 if(length(x)==0) return(TRUE)
 return(FALSE)
}
is.binary(c(1, 0, 1, 1))
```

```
[1] TRUE
```

```
is.binary(1)
```

```
[1] TRUE
```

```

is.binary(0)

[1] TRUE

is.binary(c(1, 2, 1, 1))

```

```

[1] FALSE

```

2) *as.binary* should turn vectors into a binary number. For this we will use the following rules:

- $0 \rightarrow 0$ ,  $x \neq 0 \rightarrow 1$
- FALSE  $\rightarrow 0$ , TRUE  $\rightarrow 1$
- Anything else NA

```

as.binary <- function(x) {
 if(is.logical(x)) return(as.numeric(x))
 if(is.numeric(x)) return(ifelse(x==0, 0, 1))
 NA
}

```

```

as.binary(c(0, 1, 2, 1, 2))

```

```

[1] 0 1 1 1 1

```

```

as.binary(1:4 > 2)

```

```

[1] 0 0 1 1

```

```

as.binary(0)

```

```

[1] 0

```

```

as.binary(c(1, 2, 1, "a"))

```

```

[1] NA

```

## 10 Object-Oriented Programming

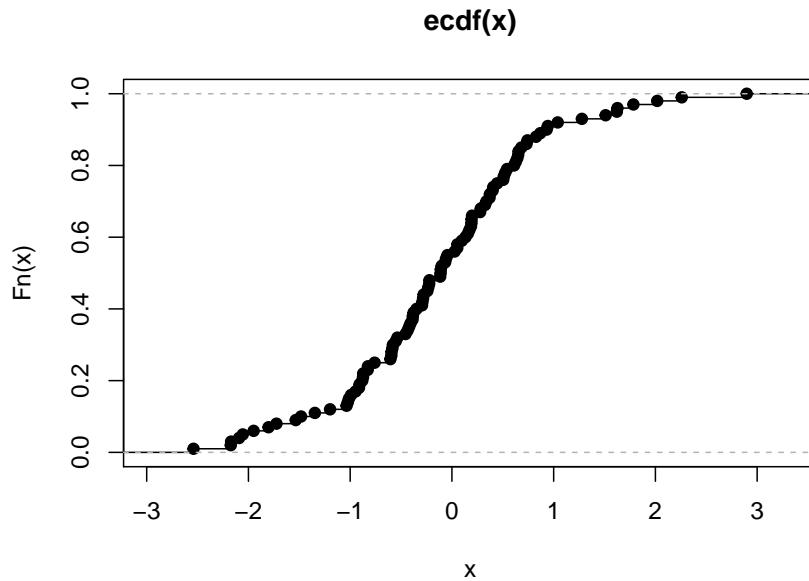
Like C++ and just about every other modern programming language R is *object oriented*. This is huge topic and we will only discuss the basic ideas. It is also only worth while (and in fact absolutely necessary) when writing large programs, at least several 100 lines.

We start with the following:

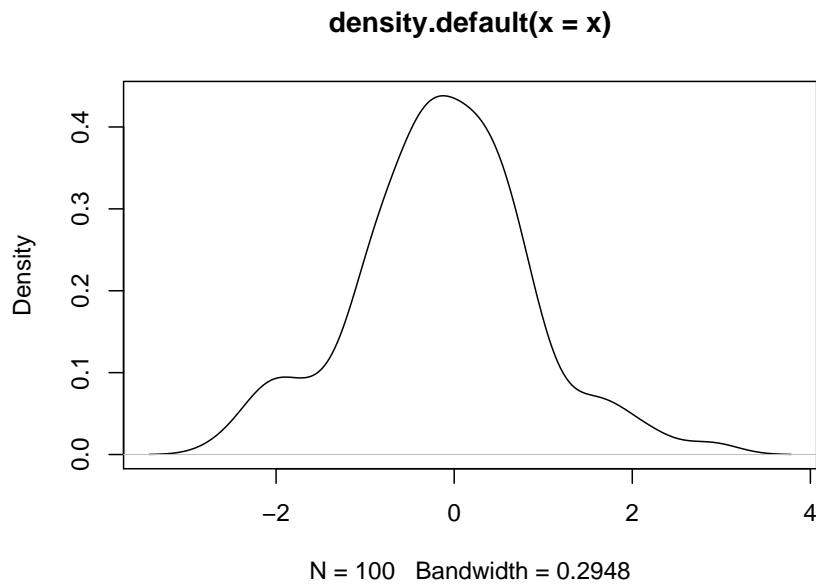
```

x <- rnorm(100)
a <- ecdf(x)
plot(a)

```



```
a <- density(x)
plot(a)
```



so also we call the same command (plot) we get different graphs, one of the empirical distribution function and the other a non-parametric density estimate.

But how does R know what to do in either case? The reason is that each object has a class property:

```
a <- ecdf(x)
class(a)
```

```

[1] "ecdf" "stepfun" "function"
a <- density(x)
class(a)

```

```
[1] "density"
```

There are also many different plot functions (or methods)

```
methods(plot)
```

```

[1] plot,ANY-method plot,color-method plot.acf*
[4] plot.data.frame* plot.decomposed.ts* plot.default
[7] plot.dendrogram* plot.density* plot.ecdf
[10] plot.factor* plot.formula* plot.function
[13] plot.ggplot* plot.gtable* plot.hclust*
[16] plot.histogram* plot.HoltWinters* plot.isoreg*
[19] plot.lm* plot.medpolish* plot.mlm*
[22] plot.ppr* plot.prcomp* plot.princomp*
[25] plot.profile.nls* plot.R6* plot.raster*
[28] plot.spec* plot.stepfun plot.stl*
[31] plot.table* plot.ts plot.tskernel*
[34] plot.TukeyHSD*
see '?methods' for accessing help and source code

```

so what happens is that when we call `plot` with some object R examines its `class` property, and the picks the specific `plot` method that corresponds to it.

R actually has three different ways to use object-oriented programming, called S3, S4 and RC. We won't go into the details and which of them is more useful under what circumstances. In the following examples we use S3, which is the easiest to use but also usually sufficient.

Say we work for a store. At the end of each day we want to create a short report that

- gives the number of sales, their mean and standard deviation for each salesperson.
- does a boxplot of the sales, grouped by salesperson

Say the data is in a data frame where the first column is the amount of a sale and the second column identifies the salesperson. So it might look like this:

Sales	Salesperson
50.85	Mary
37.61	Ann
14.64	Mary
22.03	Jim
55.73	Jim

Here is the non-object oriented solution:

```

report <- function(dta) {
 salespersons <- unique(dta$Salesperson)
 tbl <- matrix(0, length(salespersons), 3)

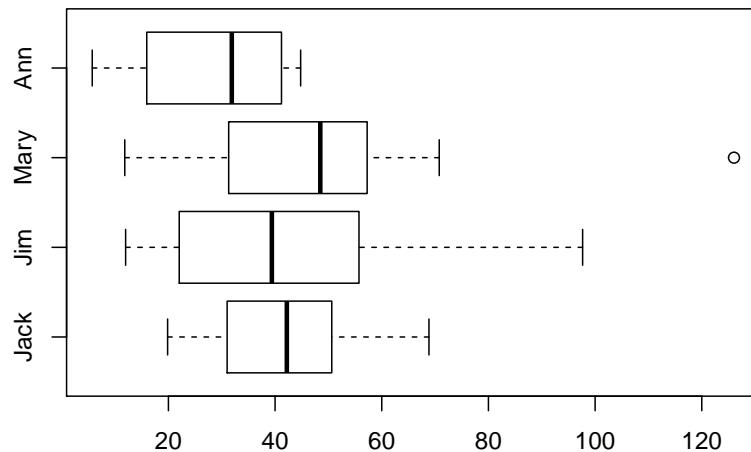
```

```

rownames(tbl) <- salespersons
colnames(tbl) <- c("Sales", "Mean", "SD")
tbl[, 1] <- tapply(dta$Sales, dta$Salesperson, length)
tbl[, 2] <- round(tapply(dta$Sales, dta$Salesperson, mean), 2)
tbl[, 3] <- round(tapply(dta$Sales, dta$Salesperson, sd), 2)
print(tbl)
boxplot(dta$Sales~dta$Salesperson, horizontal=TRUE)
}
report(sales.data)

Sales Mean SD
Mary 8 42.05 15.52
Ann 14 43.60 27.35
Jim 14 48.77 28.27
Jack 4 28.57 17.06

```



Here is the object oriented one. First we have to define a new class:

```

as.sales <- function(x) {
 class(x) <- "sales"
 return(x)
}

```

Next we have to define the methods:

```

stats <- function(x) UseMethod("stats")
stats.sales <- function(dta) {
 salespersons <- unique(dta$Salesperson)
 tbl <- matrix(0, length(salespersons), 3)
 rownames(tbl) <- salespersons
}

```

```

colnames(tbl) <- c("Sales", "Mean", "SD")
tbl[, 1] <- tapply(dta$Sales, dta$Salesperson, length)
tbl[, 2] <- round(tapply(dta$Sales, dta$Salesperson, mean), 2)
tbl[, 3] <- round(tapply(dta$Sales, dta$Salesperson, sd), 2)
tbl
}

plot <- function(x) UseMethod("plot")
plot.sales <- function(dta)
 boxplot(dta$Sales~dta$Salesperson, horizontal=TRUE)

```

and now we can run

```

sales.data <- as.sales(sales.data)
stats(sales.data)

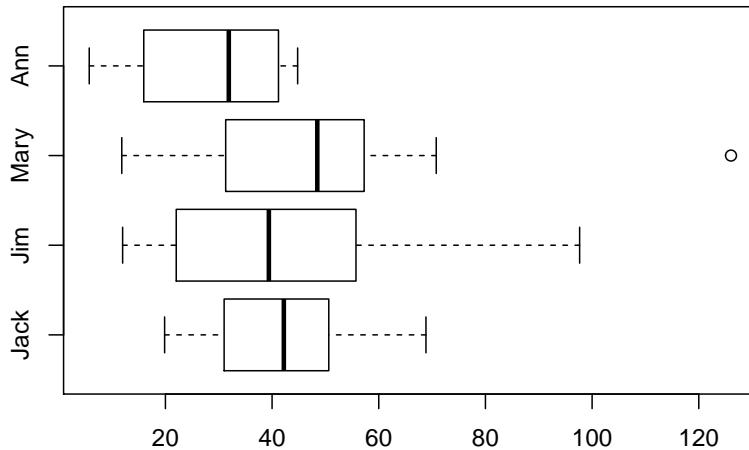
```

```

Sales Mean SD
Mary 8 42.05 15.52
Ann 14 43.60 27.35
Jim 14 48.77 28.27
Jack 4 28.57 17.06

plot(sales.data)

```

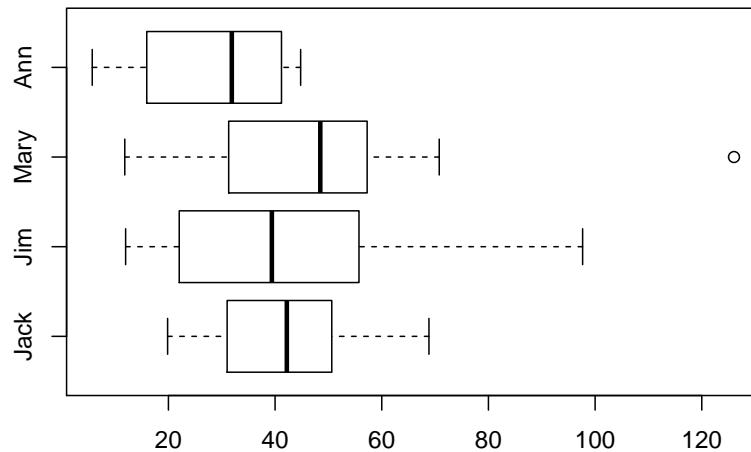


So far not much has been gained. But let's say that sometimes we also have information on the whether the sales person was on the morning or on the afternoon shift, and we want to include this in our report. One great feature of object-oriented programming is *inheritance*, that is we can define a new class that already has all the features of the old one, plus whatever new one we want.

so say now the data is

Sales	Salesperson	Time
50.85	Mary	Afternoon
37.61	Ann	Morning
14.64	Mary	Morning
22.03	Jim	Morning
55.73	Jim	Afternoon

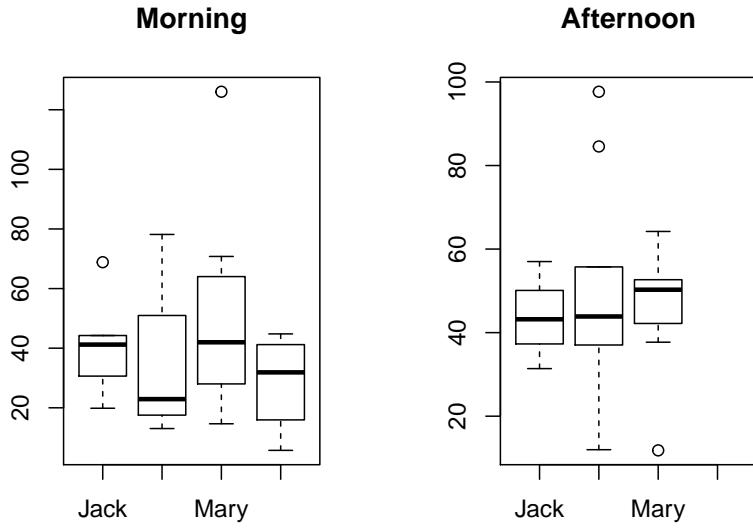
```
class(sales.time.data) <- c("salestime", "sales")
plot(sales.time.data)
```



and so we see that because sales.data is also of class sales plot still works. But we can also define its own plot method:

```
plot <- function(x) UseMethod("plot")
plot.salestime <- function(dta) {
 par(mfrow=c(1,2))
 Sales <- dta$Sales[dta$Time=="Morning"]
 Salesperson <- dta$Salesperson[dta$Time=="Morning"]
 boxplot(Sales~Salesperson, main="Morning")
 Sales <- dta$Sales[dta$Time=="Afternoon"]
 Salesperson <- dta$Salesperson[dta$Time=="Afternoon"]
 boxplot(Sales~Salesperson, main="Afternoon")

}
plot(sales.time.data)
```



generally every class has at least three methods:

- print
- summary (stats)
- plot

## 11 Vector Arithmetic

One of the most useful features of R is its ability to do math on vectors. In fact we have already used this feature many times, but now we will study it explicitly.

```
x <- 1:10
x

[1] 1 2 3 4 5 6 7 8 9 10
x^2
[1] 1 4 9 16 25 36 49 64 81 100
sqrt(x)

[1] 1.00000000000000 1.41421356237310 1.73205080756888 2.00000000000000
[5] 2.23606797749979 2.44948974278318 2.64575131106459 2.82842712474619
[9] 3.00000000000000 3.16227766016838
x^2*exp(-(x/10)^2)

[1] 0.990049833749168 3.843157756609293 8.225380667441053
```

```

[4] 13.634300623459382 19.470019576785123 25.116347738557117
[7] 30.018693315036391 33.746715138755107 36.033503364058227
[10] 36.787944117144235

y <- rep(1:2, 5)
y

[1] 1 2 1 2 1 2 1 2 1 2

x+y

[1] 2 4 4 6 6 8 8 10 10 12

x^2+y^2

[1] 2 8 10 20 26 40 50 68 82 104

```

To some degree that also works for matrices:

```

x <- matrix(1:10, nrow=2)
x

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

x^2

[,1] [,2] [,3] [,4] [,5]
[1,] 1 9 25 49 81
[2,] 4 16 36 64 100

```

## 11.1 Matrix Algebra

R can also handle basic matrix calculations:

```

x

[,1] [,2]
[1,] 1 2
[2,] 3 4

y

[,1] [,2]
[1,] 2 2
[2,] 2 2

x*y

[,1] [,2]
[1,] 2 4
[2,] 6 8

```

so this does elementwise multiplication. If we want to do actual matrix multiplication we have

```
x %*% y

[,1] [,2]
[1,] 6 6
[2,] 14 14

rbind(1:3) %*% cbind(1:3)

[,1]
[1,] 14
```

we can use the *solve* command to solve a system of linear equations. Say we have the system

$$\begin{aligned} 2x + 3y - z &= 1 \\ x - y &= 0 \\ y + 3z &= 2 \end{aligned}$$

```
A <- rbind(c(2, 3, -1), c(1, -1, 0), c(0, 1, 3))
A

[,1] [,2] [,3]
[1,] 2 3 -1
[2,] 1 -1 0
[3,] 0 1 3

solve(A, c(1, 0, 2))
```

```
[1] 0.3125 0.3125 0.5625
```

or to find the inverse matrix:

```
solve(A)

[,1] [,2] [,3]
[1,] 0.1875 0.625 0.0625
[2,] 0.1875 -0.375 0.0625
[3,] -0.0625 0.125 0.3125
```

transposition is done with

```
t(A)

[,1] [,2] [,3]
[1,] 2 1 0
[2,] 3 -1 1
[3,] -1 0 3
```

Other functions for matrices are *qr* for decomposition, *eigen* for computing eigenvalues and eigenvectors, and *svd* for singular value decomposition. There are also packages for dealing with things like sparse matrices etc.

## 11.2 Cycling

consider the following:

```
x <- 1:3
y <- 1:4
x+y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
[1] 2 4 6 5
```

so although we are adding a vector of length 3 to a vector of length 4 R still does it. It simply takes the shorter vector and starts it from the first element. It does however give a warning.

```
x <- 1:3
y <- 1:6
x+y
```

```
[1] 2 4 6 5 7 9
```

here the length of the longer obejct is a multiple of the shorter one, so the shorter one “fits” into the longer one. Now R does not print a warning.

In general you should try to avoid this sort of thing, it usually stems from an error in your program! The one exception is if one of the vectors is of length one:

```
cbind(1:5, "A")
```

```
[,1] [,2]
[1,] "1" "A"
[2,] "2" "A"
[3,] "3" "A"
[4,] "4" "A"
[5,] "5" "A"
```

even then I recommend to write this as

```
cbind(1:5, rep("A", 5))
```

```
[,1] [,2]
[1,] "1" "A"
[2,] "2" "A"
[3,] "3" "A"
[4,] "4" "A"
[5,] "5" "A"
```

just for clarities sake.

### 11.3 Vectorize

When you write your own functions you should write them in such a way that they in turn are vectorized, that is can be applied to vectors. Here is one way to do this. Consider the function *integrate*, which does numerical integration:

```
f <- function(x) {x^2}
I <- integrate(f, 0, 1)
is.list(I)

[1] TRUE

names(I)

[1] "value" "abs.error" "subdivisions" "message"
[5] "call"
```

as we see the result of a call to the *integrate* function is a list. The important part is the value, so we can write

```
integrate(f, 0, 1)$value

[1] 0.333333333333333
```

but let's say we want to calculate this integral for an interval of the form  $[0, A]$ , not just  $[0, 1]$ . Here  $A$  might be many possible values. We can do this:

```
fA <- function (A) integrate(f, 0, A)$value
fA(1)
```

```
[1] 0.333333333333333
fA(2)
```

```
[1] 2.666666666666667
```

but not

```
fA(1:2)
```

```
Error in integrate(f, 0, A): 'upper' must be of length one
```

so we need to “vectorize”:

```
fAvec <- Vectorize(fA)
fAvec(c(1, 2))
```

```
[1] 0.333333333333333 2.666666666666667
```

This works fine, but does have some drawbacks. General functions like *Vectorize* have to work in a great many different cases, so they need to do a lot of checking, which takes time to do. In practice it is often better to vectorize your routine yourself:

```
fA.vec <- function (A) {
 y <- 0*A
 for(i in seq_along(A))
```

```

 y[i] <- integrate(f, 0, A[i])$value
 }
fA.vec(c(1, 2))

[1] 0.333333333333333 2.666666666666667

```

## 11.4 apply family of functions

There is a set of routines that can be used to vectorize. Say we want to do a simulation to study the variance of the mean and the median in a sample from the normal distribution.

```

sim1 <- function(n, B=1e4) {
 y <- matrix(0, B, 2)
 for(i in 1:B) {
 x <- rnorm(n)
 y[i, 1] <- mean(x)
 y[i, 2] <- median(x)
 }
 c(sd(y[, 1]), sd(y[, 2]))
}
sim1(50)

[1] 0.142262254363946 0.175534581887725

```

Here is an alternative:

```

sim2 <- function(n, B=1e4) {
 x <- matrix(rnorm(n*B), B, 50)
 c(sd(apply(x, 1, mean)), sd(apply(x, 1, median)))
}
sim2(50)

```

```
[1] 0.140288656373574 0.173575145638526
```

Now this obviously has the advantage of being shorter.

If you read books on R written more than a few year ago you find many comments warning against the use of loops. They used to be very slow, much slower than using apply. Let's check the speed of the calculation with the *microbenchmark* package:

```

library(microbenchmark)
microbenchmark(sim1(50), times = 10)

Unit: milliseconds
expr min lq mean median uq max
sim1(50) 369.174234 374.67713 402.2375165 378.018616 386.477539 616.89057
neval
10

```

```

microbenchmark(sim2(50), times = 10)

Unit: milliseconds
expr min lq mean median uq
sim2(50) 380.344856 382.791074 402.2721239 394.621933 403.798347
max neval
490.901991 10

```

so the loop is actually faster! A few versions ago the whole implementation of loops in R was rewritten, and these days they are actually quite fast!

That still leaves the advantage of short code. There are variants of apply for other data structures:

- *lapply* for lists:

```

x <- list(A=rnorm(10), B=rnorm(20), C=rnorm(30))
lapply(x, mean)

```

```

$A
[1] -0.443335461253751
##
$B
[1] -0.113300259712128
##
$C
[1] -0.0342860249402195

```

as we see the result is again a list. Often we want it to be a vector. We could use *unlist*, or *sapply*

```

A B C
-0.4433354612537506 -0.1133002597121275 -0.0342860249402195

```

- *tapply*

apply a function to the numbers in one vector, grouped by the values in another (categorical) vector:

```

GPA <- round(runif(100, 2, 4), 1)
Gender <- sample(c("Male", "Female"),
 size=100, replace=TRUE)
tapply(GPA, Gender, mean)

```

```

Female Male
3.00930232558140 3.04736842105263

```

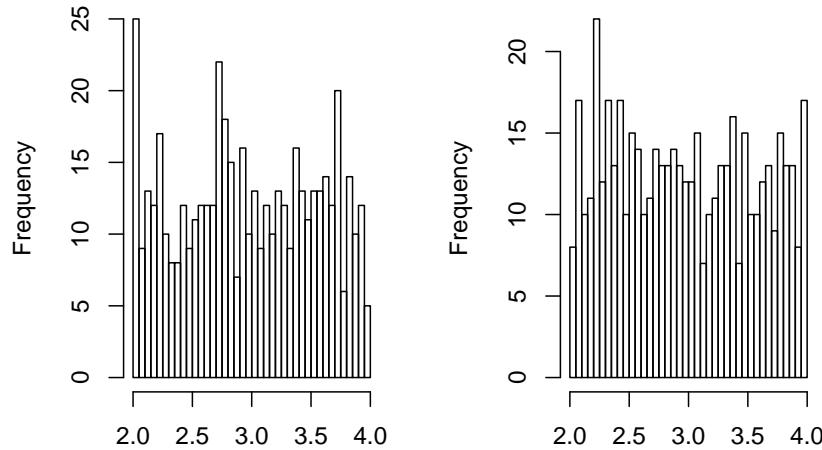
Here is another less obvious example:

```

GPA <- round(runif(1000, 2, 4), 2)
Gender <- sample(c("Male", "Female"),
 size=1000, replace=TRUE)

```

```
par(mfrow=c(1,2))
tapply(GPA, Gender, hist, breaks=50, main="", xlab="")
```



## 12 Character Manipulation

working with character strings is one of the most common tasks in R. In this section we discuss some of the routines we have for that.

Character strings can use single or double quotes:

```
'this is a string'
```

```
[1] "this is a string"
```

```
"this is a string"
```

```
[1] "this is a string"
```

Say you want to type in a vector of names. Having to type all those quotes is hazzle, but there is a nice routine in the *Hmisc* package that helps:

```
library(Hmisc)
Cs(Joe, Jack, Ann, Laura)
```

```
[1] "Joe" "Jack" "Ann" "Laura"
```

Let's say we have a vector with the names of the states in the USA (plus DC and PR)

```
head(states)
```

```
[1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"
```

```
[6] "Colorado"
```

to find out how long a string is use

```
nchar(states)[1:4]
```

```
[1] 7 6 7 8
```

What state has the longest name?

```
states[which(nchar(states) == max(nchar(states)))]
```

```
[1] "District of Columbia"
```

Say we want to shorten the strings to just the first three letters:

```
substring(states, first=1, last=3)[1:4]
```

```
[1] "Ala" "Ala" "Ari" "Ark"
```

Now, though, several of the strings are the same ("Ala"). If that is a problem use

```
abbreviate(states)[1:4]
```

```
Alabama Alaska Arizona Arkansas
"Albm" "Alsk" "Arzn" "Arkn"
```

this routine figures out what the length of the shortest string is that makes all of them unique (here 4). We can make it a little longer if we want:

```
abbreviate(states, minlength = 6)[1:4]
```

```
Alabama Alaska Arizona Arkansas
"Alabam" "Alaska" "Arizon" "Arknss"
```

Say we want the last 3 letters of the states names:

```
substring(states, first=nchar(states)-2, last=nchar(states))[1:4]
```

```
[1] "ama" "ska" "ona" "sas"
```

Let's say we want all the states whose name starts with P:

```
grep("P", states)
```

```
[1] 39 52
```

tells us those are the states at position 39 and 52, so now

```
states[grep("P", states)]
```

```
[1] "Pennsylvania" "Puerto Rico"
```

or

```
grep("P", states, value = TRUE)
```

```
[1] "Pennsylvania" "Puerto Rico"
```

Here is another way to do this:

```
states[startsWith(states, "P")]
[1] "Pennsylvania" "Puerto Rico"
```

Very useful is its partner:

```
states[endsWith(states, "o")]
[1] "Colorado" "Idaho" "New Mexico" "Ohio" "Puerto Rico"
```

This can be used for example to find the files of a certain type in a folder:

```
dir()[endsWith(dir(), ".Rmd")][1:5]
[1] "_main.Rmd" "assign.Rmd" "bayes.Rmd" "blank.Rmd"
[5] "bootstrap.Rmd"
```

Notice that above we only got the states whose names have a capital P. What if we want all states with either p or P?

```
grep(pattern = "[pP]", x = states, value = TRUE)
[1] "Mississippi" "New Hampshire" "Pennsylvania" "Puerto Rico"
```

or we can use the function *tolower*, which turns all the letters into lower case:

```
grep("p", tolower(states), value = TRUE)
[1] "mississippi" "new hampshire" "pennsylvania" "puerto rico"
```

Of course there is also a *toupper* function.

We have the *grepl* function, which returns TRUE if the string contains the pattern:

```
grepl(pattern = "A", states)[1:6]
[1] TRUE TRUE TRUE TRUE FALSE FALSE
```

Suppose we want to replace all the A's with \*'s:

```
gsub("A", "*", states)[1:6]
[1] "*labama" "*laska" "*rizona" "*rkansas" "California"
[6] "Colorado"
```

There is also the *sub* function, which does the same as gsub but only to the first occurrence:

```
sub("a", "A", c("abba"))
[1] "Abba"
gsub("a", "A", c("abba"))
[1] "AbbA"
```

Let's ask the following question: what is the distribution of the vowels in the names of the states? For instance, let's start with the number of a's in each name. There's a very useful function for this purpose: *gregexpr*.

We can use it to get the number of times that a searched pattern is found in a character vector. When there is no match, we get a value -1.

```
positions_a <- gregexpr(pattern = "a", text = states,
 ignore.case = TRUE)
positions_a[[1]]

[1] 1 3 5 7
attr(,"match.length")
[1] 1 1 1 1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

tells us that in “Alabama” there are a's in positions 1, 3, 5 and 7.

Now we need to go through all the states names and find out how many a's there are in each. Here is a fast way to do this, using one of the *apply* functions:

```
num_a <- sapply(positions_a, function(x) ifelse(x[1] > 0, length(x), 0))

[1] 4 3 2 3 2 1 0 2 1 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2
[36] 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0 0
```

Now let's do this for all the vowels:

```
vowels <- c("a", "e", "i", "o", "u")
num_vowels <- rep(0, 5)
names(num_vowels) <- vowels
for(i in seq_along(vowels)) {
 positions <- gregexpr(pattern = vowels[i],
 text = states, ignore.case = TRUE)
 num_vowels[i] <- sum(sapply(positions,
 function(x) ifelse(x[1] > 0, length(x), 0)))
}
num_vowels

a e i o u
62 29 48 40 10
```

### 12.0.1 paste

One of the most useful commands in R is *paste*. It let's us put together various parts as a string:

```
paste(1:3)

[1] "1" "2" "3"

paste("a", 1:3)

[1] "a 1" "a 2" "a 3"

paste("a", 1:3, sep=":")

[1] "a:1" "a:2" "a:3"

paste("a", 1:3, sep="")

[1] "a1" "a2" "a3"

paste0("a", 1:3)

[1] "a1" "a2" "a3"
```

Notice that the last two are the same.

If we want to make a single string use

```
paste0("a", 1:3, collapse="")

[1] "a1a2a3"

paste0("a", 1:3, collapse="-")

[1] "a1-a2-a3"
```

paste “combines” stuff into a string. Sometimes we want to do the opposite:

```
txt <- "This is a short sentence"
strsplit(txt, " ")

[[1]]
[1] "This" "is" "a" "short" "sentence"
```

notice that the result is a list, so often we use

```
unlist(strsplit(txt, " "))

[1] "This" "is" "a" "short" "sentence"
```

## 12.1 Regular Expressions

A *regular expression* (a.k.a. regex) is a special text string for describing a certain amount of text. This “certain amount of text” receives the formal name of pattern. Hence we say that

a regular expression is a pattern that describes a set of strings.

Tools for working with regular expressions can be found in virtually all scripting languages (e.g. Perl, Python, Java, Ruby, etc). R has some functions for working with regular expressions although it does not provide the wide range of capabilities that other scripting languages do. Nevertheless, they can take us very far with some workarounds (and a bit of patience).

To know more about regular expressions in general, you can find some useful information in the following resources:

- Regex wikipedia [http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)
- Regular-Expressions.info website (by Jan Goyvaerts) <http://www.regular-expressions.info>

The main purpose of working with regular expressions is to describe patterns that are used to match against text strings. Simply put, working with regular expressions is nothing more than pattern matching.

The result of a match is either successful or not. The simplest version of pattern matching is to search for one occurrence (or all occurrences) of some specific characters in a string. For example, we might want to search for the word “programming” in a large text document, or we might want to search for all occurrences of the string “apply” in a series of files containing R scripts.

The most important use of regular expressions is in the replacement of a pattern, say using gsub. Regular expressions allow us to not just use specific characters as patterns but much more general things:

Let's take the vector

```
txt
[1] "In" "2017" "there" "where" "17"
[6] "hurricanes"
```

Let's say I want to pick out those elements of the vector that are (or at least could be) numeric. Here is one way to do it:

```
as.numeric(txt)

[1] NA 2017 NA NA 17 NA
txt[!is.na(as.numeric(txt))]

[1] "2017" "17"
```

or we can use regexp:

```
txt[grep1("\\d", txt)]

[1] "2017" "17"
```

say we want to replace the spaces in a sentence with the underscore:

```
gsub("\\s", "_", "Not a very interesting sentence")
```

```
[1] "Not_a_very_interesting_sentence"
```

We already used [pP] before to match both small and large cap p's. This is in fact a regular expression:

```
gsub("[0-9]", "%", txt)
```

```
[1] "In" "%%%%" "there" "where" "%%"
[6] "hurricanes"
```

A ^ in front is NOT:

```
gsub("[^0-9]", "%", txt)
```

```
[1] "%%" "2017" "%%%%%%" "%%%%%%" "17"
[6] "%%%%%%%%%%"
```

## 12.2 POSIX

Closely related to the regex character classes we have what is known as POSIX character classes. In R, POSIX character classes are represented with expressions inside double brackets [[ ]].

```
[:lower:] Lower-case letters
[:upper:] Upper-case letters [:alpha:] Alphabetic characters ([[:lower:]] and [:upper:])
[:digit:] Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[:alnum:] Alphanumeric characters ([[:alpha:]] and [:digit:])
[:blank:] Blank characters: space and tab [:cntrl:] Control characters
[:punct:] Punctuation characters: ! " # % & ' () * + , - . / : ;
[:space:] Space characters: tab, newline, vertical tab, form feed, carriage return, and space
[:xdigit:] Hexadecimal digits: 0-9 A B C D E F a b c d e f
[:print:] Printable characters ([[:alpha:]], [:punct:] and space) [:graph:] Graphical characters ([[:alpha:]] and [:punct:]))
```

so we could also do this

```
as.numeric(txt[grep("[[:digit:]]", txt)])
```

```
[1] 2017 17
```

## 12.3 Some Examples

### 12.3.1 Palindrome

a *palindrome* is a word that is the same when read forwards or backwards. Some examples are noon, civic, radar, level, rotor, kayak, reviver, racecar, redder, madam, and refer. Let's write a sequence of commands that take a sentence and return any palindromes. As an example, consider

```
txt <- "At Noon the Meteorologist is checking the Radar"
```

which should result in the vector ("noon", "radar").

First we need to split the sentence into words:

```
wrds <- unlist(strsplit(txt, " "))

[1] "At" "Noon" "the" "Meteorologist"
[5] "is" "checking" "the" "Radar"
```

Next we need to turn each word around:

```
n <- length(wrds)
rev.wrds <- rep("", n)
for(i in 1:n)
 rev.wrds[i] <- paste(unlist(strsplit(wrds[i], ""))[nchar(wrds[i]):1], collapse = "")
rev.wrds

[1] "tA" "nooN" "eht" "tsigoloroeteM"
[5] "si" "gnikcehc" "eht" "radaR"
```

Finally, let's check whether the words are the same, but taking into account that Noon is still a palindrome!

```
wrds[tolower(wrds) == tolower(rev.wrds)]

[1] "Noon" "Radar"
```

### 12.3.2 HTML Table

Let's write a routine that takes a data frame and creates a vector of characters which correspond to an html table.

The general format of such a table is this

```
<table border="1">
<tr><th>First</th><th>Second</th></tr>
<tr><td>1.0</td><td>2.5</td></tr>
<tr><td>1.5</td><td>1.5</td></tr>
</table>
```

which results in this table:

First

Second

1.0

2.5

1.5

## 1.5

As an example consider this table for the data set *brainsize*:

Animal	body.wt.kg	brain.wt.g
African elephant	6654.000	5712.0
African giant pouched rat	1.000	6.6
Arctic Fox	3.385	44.5
Arctic ground squirrel	0.920	5.7
Asian elephant	2547.000	4603.0

First we write a routine that adds the tags to a vector:

```
add.tags <- function(x, which="header") {
 if(which=="header")
 out <- paste0("<th>", x, "</th>", collapse="")
 else
 out <- paste0("<td>", x, "</td>", collapse="")
 out
}
```

and now we can write the routine:

```
make.html.table <- function(x) {
 txt <- c(add.tags(colnames(x), "header"),
 apply(x, 1, add.tags, which="body"))
 html.tbl <- c("<table border=\"1\">",
 paste0("<tr>", txt, "</tr>"),
 "</table>")
 write(html.tbl, "clipboard", ncol=1)
}
```

Notice the routine writes the table to the clipboard. You can now run the routine, go to any html editor and simply paste the text in there.

### 12.3.3 Email Addresses

Consider the web site of the Math department at <http://math.uprm.edu/academic/people.php>

Let's say we want to write a routine that picks out all the email addresses.

First we need to download the web site. This can be done with the scan command because as is explained in the help file the argument can be a *connection*, which includes URL's

```
txt <- scan("http://math.uprm.edu/academic/people.php",
 what="char", sep="\n")
```

Next we need to figure out what defines an email address. Obviously it needs to have the @ symbol, so let's go through the text and pick out those lines that have the @ symbol:

```
sum(str_count(txt, "@"))
```

```
[1] 236
txt <- grep("@", txt, value = TRUE)
length(txt)
```

```
[1] 5
```

So the @ symbol appears 236 time. but strangely there are only 5 lines with @ symbols! That is because

```
substring(txt[1], 1, 500)
```

```
[1] "<table id=\"people\" cellspacing='0' border='0' cellpadding='2px'><tr>
```

so on the website the addresses are in a table, which we read in as a single string. We can see that immediately after each email address is the text `</a>`, which is the html tag to end a link. Let's split up the text according to the `</a>` tag:

```
txt <- unlist(str_split(paste(txt, collapse=""), ""))
txt[1:2]
```

```
[1] "<table id=\"people\" cellspacing='0' border='0' cellpadding='2px'><tr>
[2] "</td><td> robert.acar@upr.edu"
```

Notice that here I use the routine `str_split` from the package `stringr`, because the basic R routine `strsplit` does not recognize regular expressions. Don't worry about this now, we will discuss the package `stringr` soon.

Ok, next we have to eliminate all the lines that don't have a @ in it:

```
txt <- grep("@", txt, value = TRUE)
length(txt)
```

```
[1] 118
```

which is good because  $2 \times 118 = 236$ , and each address appeared twice in the table. So we got all of them.

Finally we need to extract the email adress from each line. Checking them we see that just before each address is an empty space, so maybe this will work:

```
txt <- unlist(str_split(txt, " "))
txt <- grep("@", txt, value = TRUE)
txt[1:4]

[1] "href=\"mailto:robert.acar@upr.edu\""
[2] "robert.acar@upr.edu"
[3] "href=\"mailto:edgar.acuna@upr.edu\""
[4] "edgar.acuna@upr.edu"
```

almost, we just need to get rid of those lines starting with href:

```
txt <- txt[!grepl("href", txt)]
txt
```

```

[1] "robert.acar@upr.edu" "edgar.acuna@upr.edu"
[3] "dorothy.bollman@gmail.com" "luis.caceres1@upr.edu"
[5] "gabriele.castellini@upr.edu" "paul.castillo@upr.edu"
[7] "omar.colon4@upr.edu" "silvestre.colon@upr.edu"
[9] "angel.cruz14@upr.edu" "stan.dziobiak@upr.edu"
[11] "wieslaw.dziobiak@upr.edu" "anacarmen.gonzalez@upr.edu"
[13] "marggie.gonzalez@upr.edu" "darrell.hajek@upr.edu"
[15] "edgardo.lorenzo1@upr.edu" "flor.narciso@upr.edu"
[17] "victor.ocasio1@upr.edu" "juan.ortiz35@upr.edu"
[19] "reyes.ortiz@upr.edu" "arturo.portnoy1@upr.edu"
[21] "wilfredo.quinones2@upr.edu" "karen.rios3@upr.edu"
[23] "olgamar.y.rivera@upr.edu" "yuri.rojas@upr.edu"
[25] "wolfgang.rolke@upr.edu" "juan.romero4@upr.edu"
[27] "samuel.rosario1@upr.edu" "krzysztof.rozga@upr.edu"
[29] "hector.salas@upr.edu" "damaris.santana2@upr.edu"
[31] "freddie.santiago1@upr.edu" "marko.schutz@upr.edu"
[33] "alexander.shramchenko@upr.edu" "lev.steinberg@upr.edu"
[35] "nilsa.toro@upr.edu" "pedro.torres14@upr.edu"
[37] "pedro.vasquez@upr.edu" "alejandro.velez2@upr.edu"
[39] "julio.vidaurreza@upr.edu" "uroyoan.walker@upr.edu"
[41] "keith.wayland@upr.edu" "xuerong.yong@upr.edu"
[43] "zoraida.arroyo@upr.edu" "carmen.gonzalez23@upr.edu"
[45] "tania.lopez@upr.edu" "javier.mercado3@upr.edu"
[47] "madeline.ramos3@upr.edu" "robert.trabal@upr.edu"
[49] "luisa.andino@upr.edu" "julio.barety@upr.edu"
[51] "eliseo.cruz1@upr.edu" "gladys.dicristina@upr.edu"
[53] "enriquejose.gallo@upr.edu" "cesar.herrera@upr.edu"
[55] "rafael.martinez13@upr.edu" "julioc.quintana@upr.edu"
[57] "tokuji.saito@upr.edu" "arlin.alvarado@upr.edu"
[59] "alcibiades.bustillo@upr.edu" "andres.chamorro@upr.edu"
[61] "edwin.florez@upr.edu" "einstein.morales@upr.edu"
[63] "velcy.palomino@upr.edu" "walter.quispe@upr.edu"
[65] "carlos.theran@upr.edu" "roberto.trespalacio@upr.edu"
[67] "andrea.angarita@upr.edu" "hillary.bermudez@upr.edu"
[69] "sergioi.betancourt@upr.edu" "cesar.bolanos@upr.edu"
[71] "hilda.calderon@upr.edu" "joseemilio.calderon@upr.edu"
[73] "victor.cardenas@upr.edu" "alexis.carrillo@upr.edu"
[75] "carlos.carvajal@upr.edu" "jose.cordoba@upr.edu"
[77] "henrry.cortez@upr.edu" "saed.cruz@upr.edu"
[79] "victor.diaz16@upr.edu" "francisco.dejesus3@upr.edu"
[81] "alix.enriquez@upr.edu" "angel.figueroa1@upr.edu"
[83] "jean.galan@upr.edu" "cristian.gomez1@upr.edu"
[85] "sergio.gomez@upr.edu" "cristian.gutierrez1@upr.edu"
[87] "hassam.hayek@upr.edu" "javier.henriquez@upr.edu"
[89] "sahily.hilerio@upr.edu" "ricardo.lopez9@upr.edu"
[91] "ruth.lopez5@upr.edu" "lesbia.lopez@upr.edu"

```

```

[93] "christian.lopez29@upr.edu"
[95] "alibeth.luna@upr.edu"
[97] "robert.medina@upr.edu"
[99] "kevin.molina1@upr.edu"
[101] "ana.moreno@upr.edu"
[103] "bernnie.murillo@upr.edu"
[105] "felix.pabon@upr.edu"
[107] "jessenia.quintero@upr.edu"
[109] "daniel.rocha@upr.edu"
[111] "jose.santos7@upr.edu"
[113] "maria.torres65@upr.edu"
[115] "juan.valera@upr.edu"
[117] "diana.vargas1@upr.edu" "rodrigo.leon@upr.edu"
 "eddie.mendez@upr.edu"
 "luis.mestre2@upr.edu"
 "bayron.morales@upr.edu"
 "didier.murillo@upr.edu"
 "daniel.ovalle@upr.edu"
 "cristian.perdomo@upr.edu"
 "eric.rivera8@upr.edu"
 "william.rueda@upr.edu"
 "deiver.suarez@upr.edu"
 "ana.trujillo2@upr.edu"
 "raul.valerio@upr.edu"
 "cesaraugusto.vega@upr.edu"

```

If we wanted to send an email to all the people in the department we could now use the write command to copy them to the clipboard, switch to a mail program and copy them into the address box.

Programs like these are routinely used to go through millions of websites and search for email addresses, which are then sent spam emails. This is why I only write mine like this:

wolfgang[dot]rolke[at]upr[dot]edu

#### 12.3.4 Example: Binary Arithmetic

We have previously discussed binary arithmetic. There we used a simple vector of 0's and 1's. The main problem with that is that it is hard to vectorize the routines. Instead we will now use character sequences like "1001".

We have previously written several functions for this. We will want to reuse them but also adapt them to this new format. To do so we need to turn a character string into a vector of numbers and vice versa. Also we want our routines to be vectorized:

- Decimal to Binary

```

decimal.2.binary <- function(x) {
 n <- length(x)
 y <- rep("0", n)
 for(k in 1:n) {
 if(x[k]==0 | x[k]==1) { #simple cases
 y[k] <- x[k]
 next
 }
 i <- floor(log(x[k], base=2)) #largest power of 2 less than x
 bin.x <- rep(1, i+1) #we will need i+1 0's and 1's, first is 1
 x[k] <- x[k]-2^i
 for(j in (i-1):0) {
 if(2^j>x[k])
 bin.x[j+1] <- 0
 }
 }
}

```

```

 else {
 bin.x[j+1] <- 1
 x[k] <- x[k]-2^j
 }
 }
 y[k] <- paste(bin.x[length(bin.x):1], collapse="")
}
y
}
decimal.2.binary(c(7, 8, 26))

```

```
[1] "111" "1000" "11010"
```

- Binary to Decimal

```

binary.2.decimal <- function(x){
 n <- length(x)
 y <- rep(0, n)
 for(i in 1:n) {
 tmp <- as.numeric(strsplit(x[i], "")[[1]])
 y[i] <- sum(tmp*2^(length(tmp)-1:1))
 }
 y
}
binary.2.decimal(c("111", "1000", "11010"))

```

```
[1] 7 8 26
```

```
binary.2.decimal(decimal.2.binary(126))
```

```
[1] 126
```

```
decimal.2.binary(binary.2.decimal(c("100101")))
```

```
[1] "100101"
```

- *is. binary*:

```

is.binary <- function(x) {
 n <- length(x)
 y <- rep(TRUE, n)
 for(i in 1:n) {
 x.vec <- as.numeric(strsplit(x[i], "")[[1]])
 if(all(x.vec==0)) {
 y[i] <- TRUE
 next
 }
 x.vec <- x.vec[x.vec!=0]
 x.vec <- x.vec[x.vec!=1]
 if(length(x.vec)==0) y[i] <- TRUE
 }
}

```

```

 else y[i] <- FALSE
 }
 y
}
is.binary(c("1001", "0", "11a1"))

[1] TRUE TRUE FALSE

• addition

Here I am going to reuse the routine we had already written:

binary_addition <- function(x, y) {
#First make x and y of equal length and with one extra
#slot in case it's needed for carry over
#Fill x and y with 0's as needed.
 n <- length(x)
 m <- length(y)
 N <- max(n, m)+1
 x <- c(rep(0, N-n), x)
 y <- c(rep(0, N-m), y)
 s <- rep(0, N) # for result
 ca <- 0 #for carry over term
 for(i in N:1) {
 n <- x[i]+y[i]+ca
 if(n<=1) {#no carry over
 s[i] <- n
 ca <- 0
 }
 else {#with carry over
 s[i] <- 0
 ca <- 1
 }
 }
 if(s[1]==0) s <- s[-1] #leading 0 removed if necessary
 s
}
binary_addition(c(1, 0), c(1, 1, 0))

[1] 1 0 0 0

binary.addition<- function(x, y) {
 n <- length(x)
 m <- length(y)
 if(m!=n) cat("Vectors have to have the same length!\n")
 s <- rep("0", n)
 for(i in 1:n) {
 x.vec <- as.numeric(strsplit(x[i], "")[[1]])

```

```

y.vec <- as.numeric(strsplit(y[i], "")[[1]])
tmp <- binary_addition(x.vec, y.vec)
s[i] <- paste(tmp, collapse="")
}
s
}
binary.addition(c("0", "10", "1001"), c("0", "110", "1101"))

[1] "0" "1000" "10110"

```

Let's turn this into an infix addition operator:

```

%+b% <- function(x, y) binary.addition(x, y)
x <- c("10", "1001", "100101", "11101001")
y <- c("101", "1001", "10101", "1001100")
binary.2.decimal(x)

[1] 2 9 37 233
binary.2.decimal(y)

[1] 5 9 21 76
z <- x %+b% y
x

[1] "10" "1001" "100101" "11101001"
binary.2.decimal(z)

[1] 7 18 58 309

```

Let's define a new class of objects “binary numbers”:

```

as.binary <- function(x) {
 class(x) <- "binary"
 return(x)
}

```

what methods might be useful here? Let's write two:

- print

this is how our number will appear when we use print(x):

```

print <- function(x) UseMethod("print")
print.binary <- function(x) {
 n <- length(x)
 for(i in 1:length(x)) {
 y <- as.numeric(strsplit(x[i], "")[[1]])
 y <- paste(y, collapse = ".")
 cat(y, "\n")
 }
}

```

```

}
x <- as.binary(c("10", "1001", "100101"))
print(x)

```

```

1.0
1.0.0.1
1.0.0.1.0.1

```

- summary

what should we calculate as summary statistics? Let's do three:

- how many
- most frequent (mode, NA if all only once)
- percentage of 0's

```

summary <- function(x) UseMethod("summary")
summary.binary <- function(x) {
 n <- length(x)
 if(length(unique(x))==length(x)) mode <- NA
 else {
 z <- table(x)
 z <- z[z==max(z)]
 mode <- names(z)
 }
 y <- paste(x, collapse = "") #one long string
 y <- as.numeric(strsplit(y, "")[[1]]) #vector of 0's and 1's
 y <- round(sum(y==0)/length(y)*100, 1)
 cat("N =", n, "\n")
 cat("Mode =", mode, "\n")
 cat("% 0's =", y, "\n")
}
x <- sample(1:100, size=1000, replace=TRUE)
x <- as.binary(decimal.2.binary(x))
print(x[1:5])

```

```

[1] "111010" "110" "10000" "101111" "1000001"
summary(x)

N = 1000
Mode = 1010
% 0's = 45.3

```

# 13 Data Input/Output, Transferring R Objects

## 13.1 Printing Info to Screen

The basic functions to display information on the screen are

```
x <- rnorm(5, 100, 30)
print(x)

[1] 105.6150030061991 69.3144499392920 111.0905688663961 88.3848902129131
[5] 73.8488145787354

cat(x)

105.615003006199 69.314449939292 111.090568866396 88.3848902129131 73.8488145787354
```

Both of these have certain advantages:

- with print you can easily control the number of digits:

```
print(x, digits=4)

[1] 105.62 69.31 111.09 88.38 73.85
```

- with cat you can easily mix text and numeric:

```
cat("The mean is ", round(mean(x), 1), "\n")

The mean is 89.7
```

The “\n” (newline) is needed so that the cursor moves to the next line.

Some times you need a high level of control over the output, for example when writing data to a file that then will be read by a computer program that wants things just so. For this you can use the *sprintf* command.

```
sprintf("%f", pi)
```

```
[1] "3.141593"
```

Here the f stands for floating point, the most common type. Also note that the result of a call to sprintf is a character vector.

Here are some variations:

```
sprintf("%.3f", pi) #everything before the ., 3 digits after
```

```
[1] "3.142"
```

```
sprintf("%1.0f", pi) #1 space, 0 after
```

```
[1] "3"
```

```
sprintf("%5.1f", pi) #5 spaces total, 1 after
```

```
[1] " 3.1"
```

```

sprintf("%05.1f", pi) #same but fill with 0

[1] "003.1"

sprintf("%+f", pi) #all with + in front

[1] "+3.141593"

sprintf("% f", pi) #space in front

[1] " 3.141593"

sprintf("%e", pi) #in scientific notation, small e

[1] "3.141593e+00"

sprintf("%E", pi) #or large E

[1] "3.141593E+00"

sprintf("%g", 1e6*pi)

[1] "3.14159e+06"

```

Here is another example. In Statistics we often find a p value. These should generally be quoted to three digits. But when the p value is less than  $10^{-3}$  R uses scientific notation. If you want to avoid that do this

```

p <- c(0, 0.213, 0.0001)

[1] 0.0000 0.2130 0.0001

sprintf("%.3f", p)

[1] "0.000" "0.213" "0.000"

```

## 13.2 Reading a Vector

Often the easiest thing to do is to use copy-paste on a vector and then simply *scan* it into R:

```
x <- scan("clipboard")
```

- use the argument `sep=";"` to change the symbol that is being used as a separator. The default is empty space, common cases include comma, semi-colon, and newline (`\n`)
- `scan` assumes that the data is numeric, if not use the argument `what="character"`.

I need to do this so often I wrote a little routine for it:

```

getx <- function(sep="") {
 options(warn=-1) #It might give a warning, I don't care
 x <- scan("clipboard", what="character", sep=sep)
}

```

```

#always read as character
if(all(!is.na(as.numeric(x)))) #are all elements numeric?
 x <- as.numeric(x) #then make it numeric
options(warn=0) #reset warning
x
}

```

Notice some features:

- the routine always reads the data as character vector, whether it is character or numeric.
- it then tries to turn it into numeric. If that works, fine, otherwise it stays character. This is done with `as.numeric(x)`, which returns NA if it can't turn an entry into numeric, so `is.na(as.numeric(x))` returns TRUE if x can't be made numeric.
- when trying to turn a character into a number R prints a warning. This is good in general to warn you that you are doing something strange. Here, though, it is expected behaviour and we don't need the warning. The routine suppresses them by setting `options(warn=-1)`, and setting it back to the default afterwards.

If the data is in a stand-alone file saved on your hard drive you can also read it from there:

```
x <- scan("c:/folder/file.R")
```

Notice the use of / in writing folders. \ does not work on Windows because it is already used for other things, \\ would work but is more work to type!

`scan` has a lot of arguments:

```
args(scan)
```

```

function (file = "", what = double(), nmax = -1L, n = -1L, sep = "",
quote = if (identical(sep, "\n")) "" else "'\"'", dec = ".",
skip = 0L, nlines = 0L, na.strings = "NA", flush = FALSE,
fill = FALSE, strip.white = FALSE, quiet = FALSE, blank.lines.skip = TRUE,
multi.line = TRUE, comment.char = "", allowEscapes = FALSE,
fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
NULL

```

the most useful are

- `what`
- `sep`
- `nmax`: maximum number of lines to read, useful if you don't know just how large the file is and want to read just some of it to check it out.
- `skip`: number of lines at beginning to skip, for example if there is some header.
- `quiet=FALSE`: by default R will say how many lines have been read, this can be a nuisance if you have a routine that reads in many files.

- blank.lines.skip=TRUE: does not read in empty lines. This is a problem if you want to write the file out again later and want it to look as much as possible like the original.

### 13.3 Data Frames

the standard command to read data from a table into a data frame is *read.table*.

```
x <- read.table("c:/folder/file.R")
```

it has many of the same arguments as scan (for example sep). It also has the argument header=FALSE. If your table has column names use header=TRUE. The same for row names.

Example: say the following data is saved in a file named *student.data.R*:

```
students <- data.frame(ID, Age, GPA, Gender)
```

ID	Age	GPA	Gender
63368	22	2.9	Male
75382	22	2.6	Female
43337	18	2.7	Male
56341	18	2.8	Male
43988	19	3.9	Male
47648	21	2.6	Female
10959	19	3.3	Male
57902	25	2.6	Female
48890	20	3.6	Female
18430	22	3.2	Female

Now we can use

```
read.table("c:/folder/student.data.R",
 header=TRUE, row.names = 1)
```

the row.names=1 tells R to use the first column as row names.

### 13.4 Transferring Objects from one R to another

Say you have a few data sets and routines you want to send to someone else. The easiest thing to do is use *dump* and *source*.

```
dump(c("data1", " data2", "fun1"), "c:/folder/mystuff.R")
```

Now to read in the stuff simply use

```
source("c:/folder/mystuff.R")
```

I often have to transfer stuff from one R project to another, so I wrote myself these two routines:

```
dp <- function (x) dump(x, "clipboard")
sc <- function () source("clipboard")
```

## 13.5 Special File Formats

There are routines to read all sorts of file formats. The most important one is likely *read.csv*, which can read Excel files saved in the comma delimited format.

## 13.6 Packages

there are a number of packages written to help with data I/O. We will discuss some of them later.

## 13.7 Working on Files

R can also be used to create, copy, move and delete files and folders on your hard drive. The routines are

```
dir.create(...)
dir.exists(...)
file.create(...)
file.exists(...)
file.remove(...)
file.rename(from, to)
file.append(file1, file2)
file.copy(from, to)
```

You can also get a listing of the files in a folder:

```
head(dir("c:/R"))

[1] "bin" "CHANGES" "COPYING" "doc" "etc" "include"
```

for the folder from which R started use

```
head(dir(getwd()))

[1] " (Asus's conflicted copy 2018-05-31).Rhistory"
[2] "_book"
[3] "_bookdown.yml"
[4] "_bookdown_files"
[5] "_main.out"
[6] "_main.Rmd"
```

## 14 Graphs

In this section we will discuss some graphs that are part of base R. Later we will study more advanced graphics, but it is still a good idea to know how to draw a base graph.

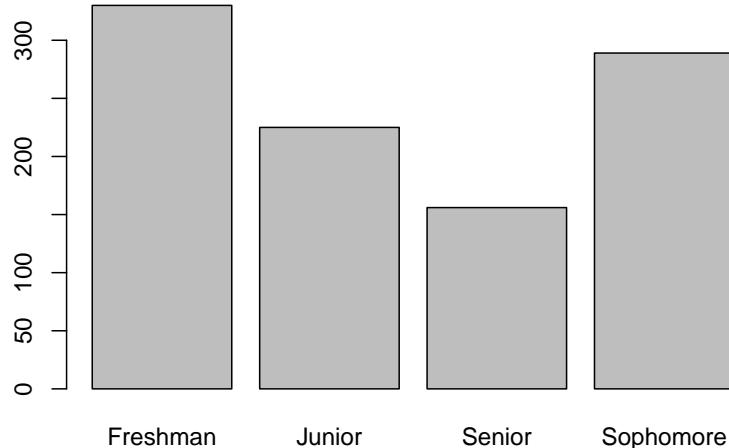
### 14.1 Barcharts

Say we have the data on the number of freshman, sophomores etc at some college:

Class	Counts
Freshman	330
Sophomore	289
Junior	225
Senior	156

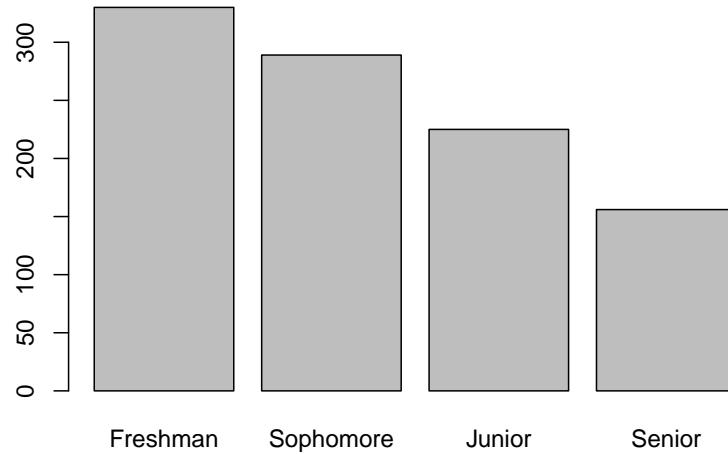
```
head(students)
```

```
[1] "Sophomore" "Freshman" "Freshman" "Sophomore" "Junior" "Senior"
barplot(table(students))
```



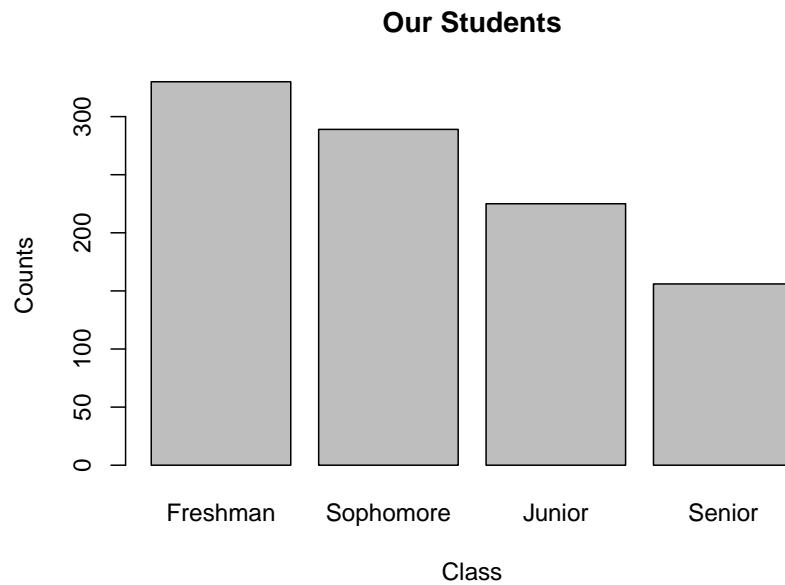
But this is not quite right, Sophomores should come second. In general, when we have a categorical variable which has an ordering we should turn it into a factor:

```
students.fac <- factor(students,
 levels = c("Freshman", "Sophomore", "Junior", "Senior"),
 ordered = TRUE)
tbl.students.fac <- table(students.fac)
barplot(tbl.students.fac)
```



There are a number of arguments we find in most graphics routines:

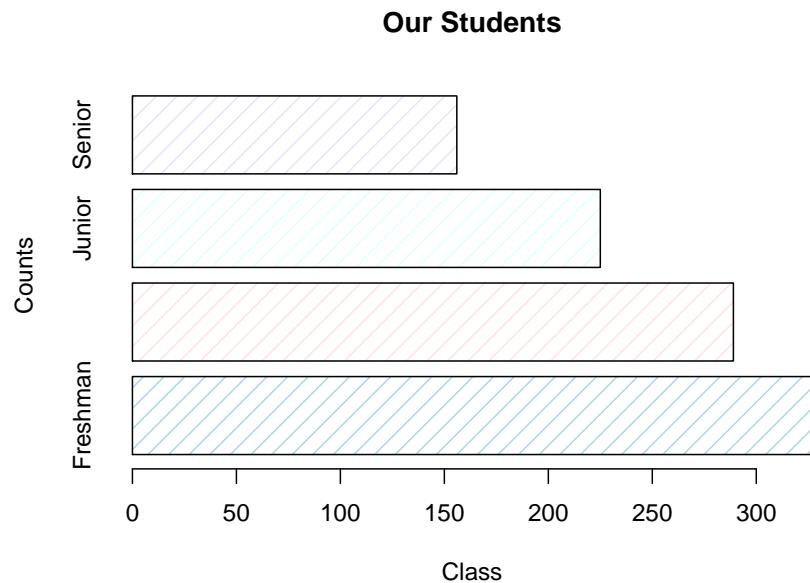
```
barplot(tbl.students.fac, main = "Our Students",
 xlab = "Class", ylab = "Counts")
```



and then there are arguments specific to the graph:

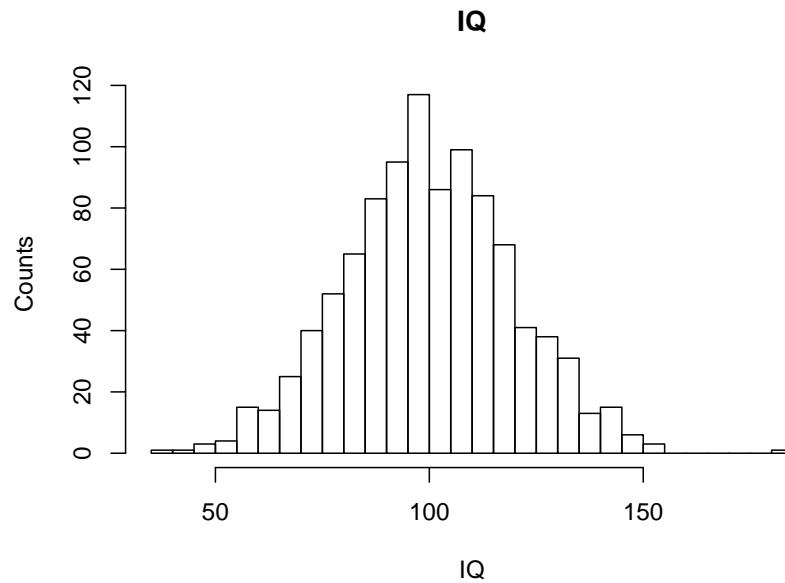
```
barplot(tbl.students.fac, main = "Our Students",
 xlab = "Class", ylab = "Counts",
 horiz = TRUE, density = 10,
 col = c("lightblue", "mistyrose",
```

```
"lightcyan", "lavender"),)
```



## 14.2 Histogram

```
x <- rnorm(1000, 100, 20)
hist(x, breaks=50, main = "IQ",
 xlab = "IQ", ylab = "Counts")
```



Often the hist command is useful for its side effect of counting:

```
range(x)

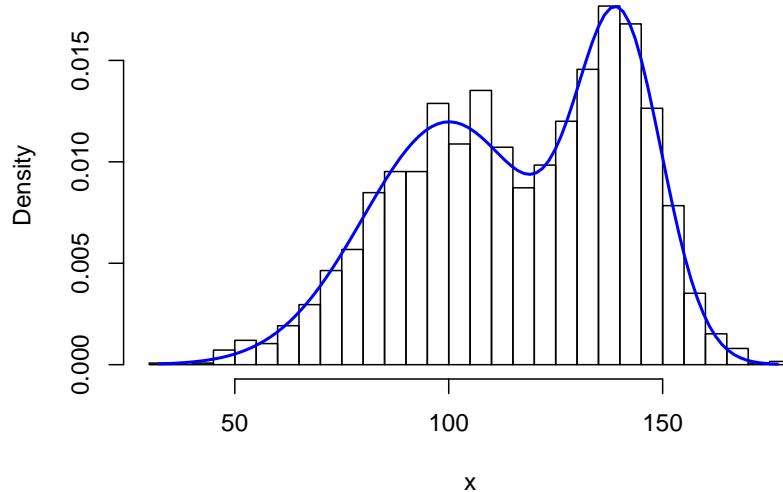
[1] 35.8045686949389 184.2477873369957

hist(x, breaks=c(0, 50, 80, 100, 120, 150, 250), plot = FALSE)$counts

[1] 5 150 360 337 144 4
```

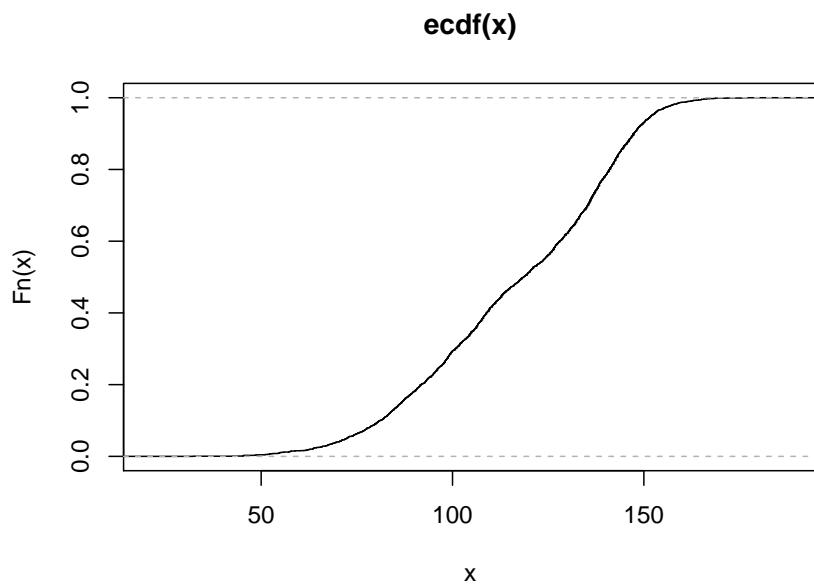
Often we want to compare a histogram with a theoretical curve, say a probability density. Then we can use the curve command with the argument add=TRUE:

```
x <- c(rnorm(1500, 100, 20), rnorm(1000, 140, 10))
hist(x, breaks=50, main = "", freq=FALSE)
f <- function(x) 1.5*dnorm(x, 100, 20) + dnorm(x, 140, 10)
I <- integrate(f, 0, 200)$value
curve(f(x)/I, from=min(x), to=max(x), add=TRUE, col="blue", lwd=2)
```



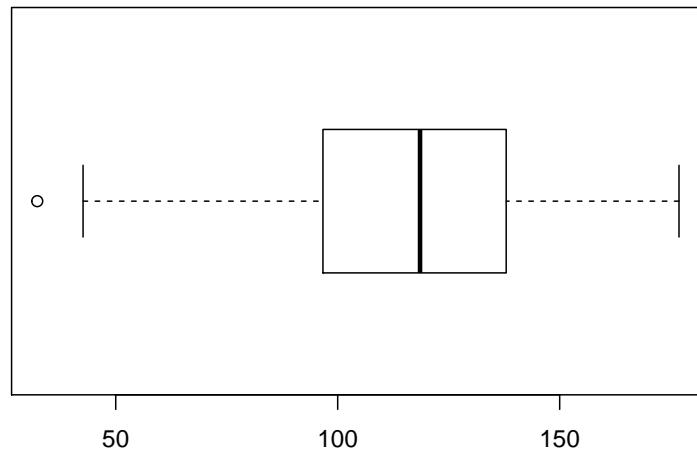
### 14.3 Empirical Distribution Function

```
plot(ecdf(x))
```

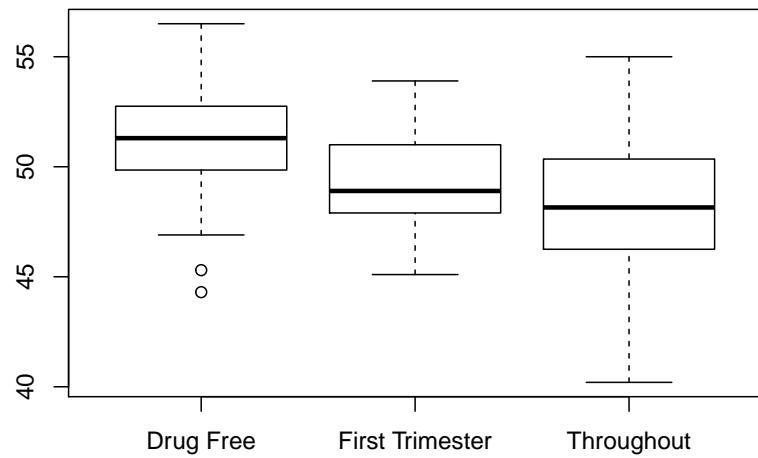


## 14.4 Boxplot

```
boxplot(x, horizontal = TRUE)
```

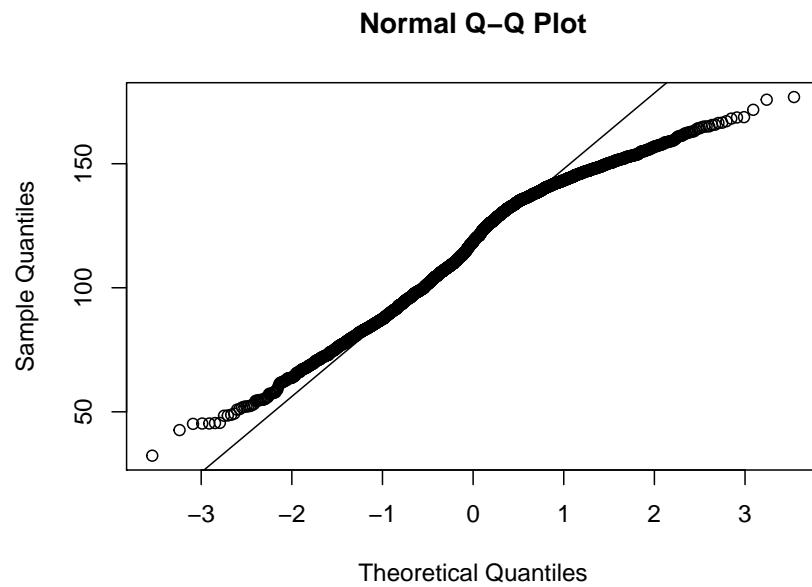


```
attach(mothers)
boxplot(Length~Status)
```



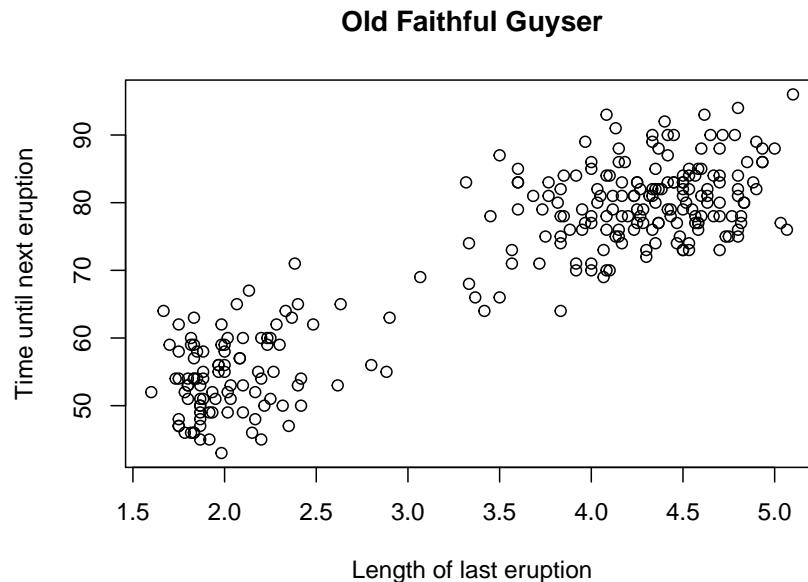
## 14.5 Normal Probability Plot

```
qqnorm(x)
qqline(x)
```



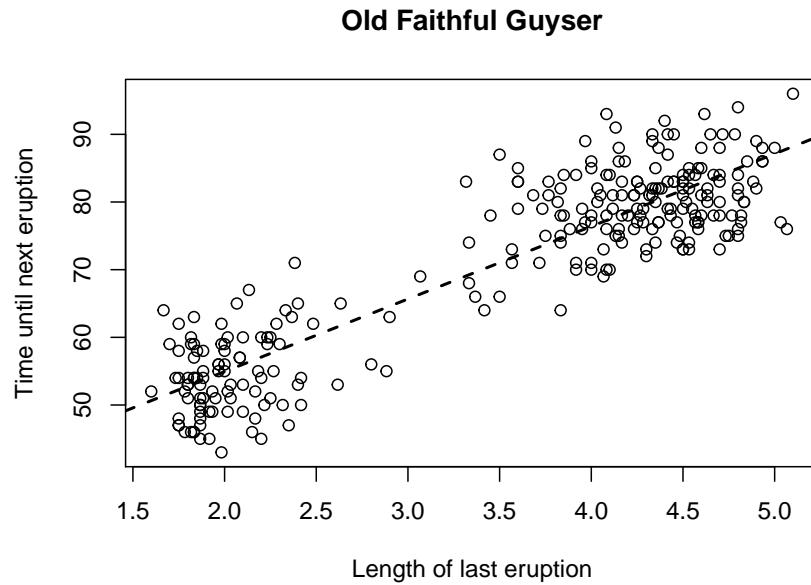
## 14.6 Scatterplot

```
attach(faithful)
plot(Eruptions, Waiting.Time,
 xlab = "Length of last eruption",
 ylab = "Time until next eruption",
 main = "Old Faithful Guyser")
```



Let's add the least squares regression fit to the graph:

```
plot(Eruptions, Waiting.Time,
 xlab = "Length of last eruption",
 ylab = "Time until next eruption",
 main = "Old Faithful Guyser")
abline(lm(Waiting.Time~Eruptions), lwd=2, lty=2)
```



There are a lot of different plotting symbols:

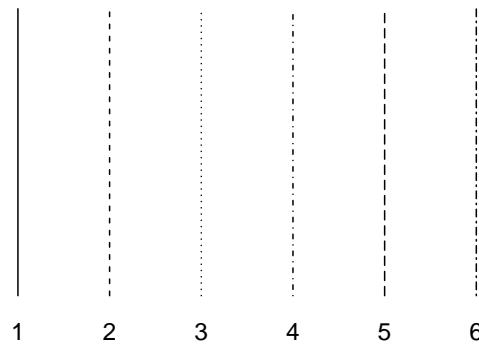
```
plot(0:16, rep(0, 17), ylim=c(0,1),
 type="n", xlab="", ylab="", axes=F)
for(i in 1:15) {
 for(k in 1:3) {
 points(i, c(0.8, 0.5, 0.2)[k], pch=i+(k-1)*15)
 text(i, c(0.9, 0.6, 0.3)[k], label=i+(k-1)*15, adj = 0)
 }
}
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
○	△	+	×	◊	▽	◻	*	◊	⊕	⊗	田	⊗	□	■
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
●	▲	◆	●	●	○	□	◊	△	▽					
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
!	"	#	\$	%	&	'	( )	*	+	,	-			

Notice the setup of an empty graph!

Also, different line types:

```
plot(0:7, rep(0, 8), ylim=c(0,1),
 type="n", xlab="", ylab="", axes=F)
for(i in 1:6) {
 segments(i, 0.2, i, 1, lty=i)
 text(i, 0.1, i)
}
```



Consider the data set mtcars, which is part of base R. It has data on 32 different cars:

```
head(mtcars)
```

```
mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4
Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4
Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1
Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1
Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2
Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1
```

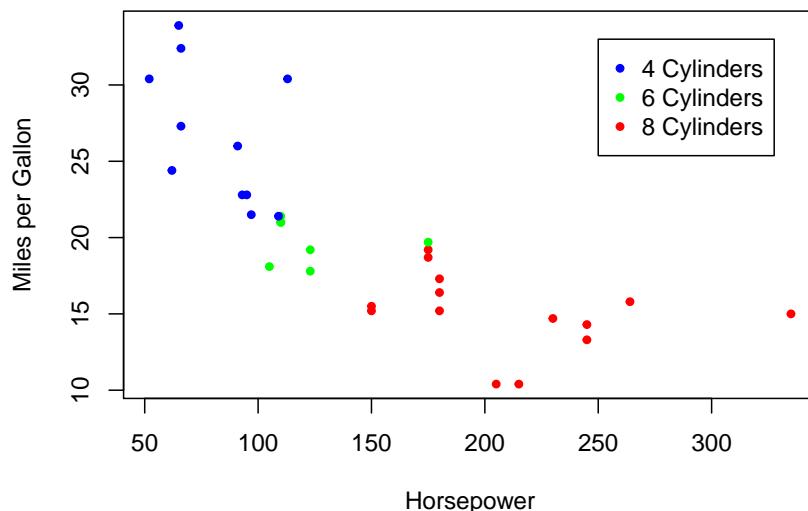
say we wish to study the relationship of mpg (Miles per Gallon) and hp (Horsepower) but also include information on the number of cylinders (either 4, 6 or 8):

```
attach(mtcars)
cols <- rep("blue", 32)
cols[cyl==6] <- "green"
cols[cyl==8] <- "red"
plot(hp, mpg,
 xlab = "Horsepower",
```

```

ylab = "Miles per Gallon",
col = cols, pch = 20)
legend(250, 33, paste(2*2:4, "Cylinders"),
 pch = rep(20, 3),
 col = c("blue", "green", "red"))

```



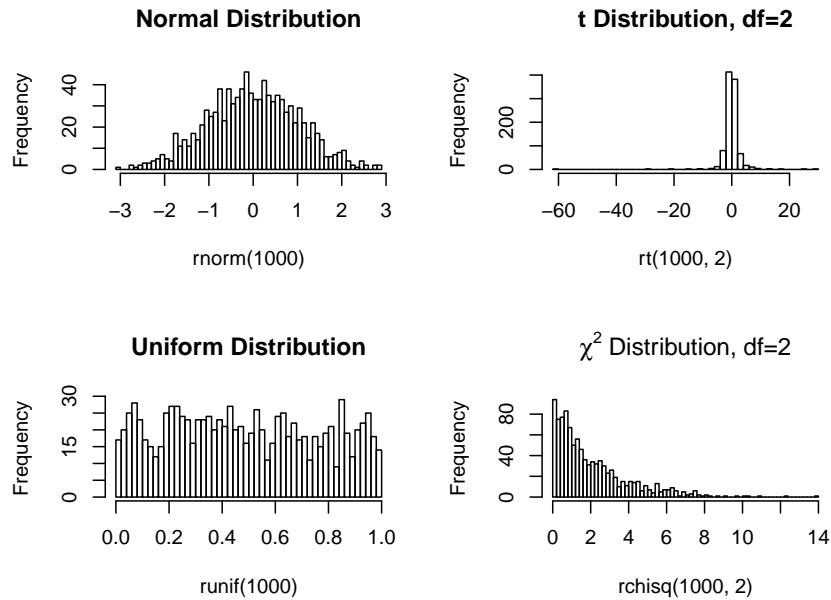
## 14.7 Multiple Graphs

sometimes we want to combine several graphs into one

```

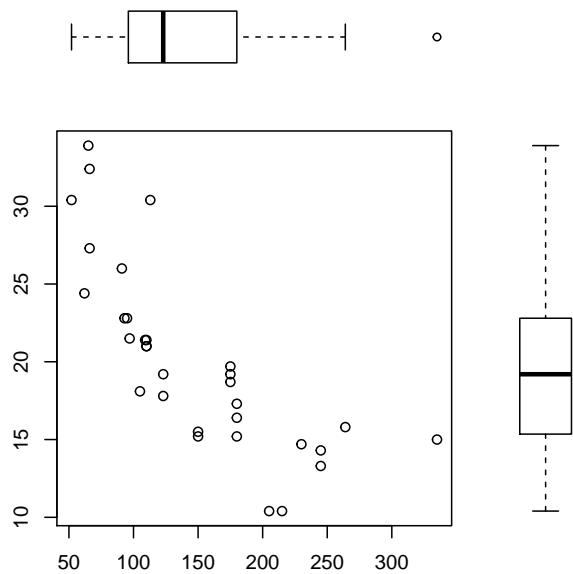
par(mfrow=c(2, 2))
hist(rnorm(1000), 50, main="Normal Distribution")
hist(rt(1000, 2), 50, main="t Distribution, df=2")
hist(runif(1000), 50, main="Uniform Distribution")
hist(rchisq(1000, 2), 50,
 main=expression(paste(chi^2, " Distribution, df=2")))

```



This works fine for rectangular arrays. For more complicated graphs we have the *layout* command. Let's create a scatterplot of mpg by hp with marginal boxplots:

```
layout(matrix(c(2, 0, 1, 3), 2, 2, byrow = TRUE),
 c(3, 1), c(1, 3), TRUE)
par(mar = c(3, 3, 1, 1))
plot(hp, mpg)
par(mar = c(0, 3, 1, 1))
boxplot(hp, axes = FALSE, horizontal = TRUE)
par(mar = c(3, 0, 1, 1))
boxplot(mpg, axes = FALSE)
```



## 14.8 Functions that create graphs

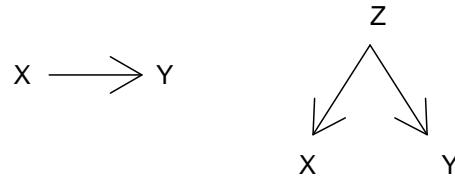
When I write a paper or a talk or anything else that requires some graphs I always write a routine *plot.mytalk*, which has everything to recreate every graph. This is a great way to not only make changes if necessary but also to remind myself what I did back years ago when I wrote it.

Caveat: these days R Markdown can be used for the same purpose.

## 14.9 Graphs that don't look like Graphs

we can use R to make other types of pictures. Consider this diagram, which I use to illustrate the topic of Cause vs Effect:

Cause–Effect                      Confounding Variable



it is actually done with these commands:

```

plot(c(10, 100), c(40, 100), axes=F, xlab="", ylab="", type="n")
text(15, 90, "Cause–Effect", cex=1.2, adj = 0)
text(c(20, 40), c(65, 65), c("X", "Y"), cex=1.1, adj=0)
arrows(25, 65, 38, 65)
text(56, 90, "Confounding Variable", cex=1.2, adj=0)
text(c(60, 70, 80), c(50, 75, 50), c("X", "Z", "Y"), cex=1.1, adj=0)
arrows(70, 70, 62, 55)
arrows(70, 70, 78, 55)
}

```

An important example of this are maps:

```

library(maps)
map("usa")
text(-87.623177, 41.881832, "Chicago", cex=1.3)

```



## 14.10 Printing Graphs

Often we need to save a graph as a png or a postscript file, so we can include it in a webpage or a latex document. I have to do this often enough I wrote a routine for it:

```
graph.out <-
function (f, foldername, graphname, format = "png")
{
 file <- paste0(foldername, graphname, ".", format)
 cat(file)
 if (format == "png")
 png(file)
 if (format == "pdf")
 pdf(file)
 if (format == "ps")
 postscript(file, horizontal = F, pointsize = 17)
 if (format == "eps") {
 setEPS()
 postscript(file, horizontal = F, pointsize = 17)
 }
 f()
 dev.off()

}
```

here  $f$  is a function that produces a graph, so it might be called like this:

```
f <- function(x) hist(rnorm(1000), 50)
graph.out(f, "C:/mygraphs", "myhist")
```

and now there should be a file myhist.png in the folder C:/mygraphs.

## 15 Model Notation

A number of R routines (for example boxplot and lm) use the *model* notation  $y \sim x$ , which you should read as  $y$  modeled as a function of  $x$ . So for example if we want to find the least squares regression model of  $y$  on  $x$  we use

```
lm(y ~ x)
```

In standard math notation that means fitting an equation of the form

$$Y = \beta_0 + \beta_1 x + \epsilon$$

Some times one wants to fit a no-intercept model:

$$Y = \beta_1 x + \epsilon$$

and this is done with

```
lm(y ~ x - 1)
```

If there are several predictors you can

- fit an additive model with

```
lm(y ~ x + z)
```

- fit the interaction term with

```
lm(y ~ x * z)
```

In the case of three (or more predictors) there are all sorts of possibilities:

- model without interactions

$$Y_i = \beta_0 + \sum_{i=1}^n \beta_i x_i + \epsilon$$

```
lm(y ~ x1 + x2 + x3)
```

- model with all interactions

$$Y_i = \beta_0 + \sum_{i=1}^n \beta_i x_i + \sum_{i,j=1}^n \beta_{ij} x_i x_j + \beta_{123} x_1 x_2 x_3 + \epsilon$$

```
lm(y ~ (x1 + x2 + x3)^3)
```

- model with all pairwise interactions

$$Y_i = \beta_0 + \sum_{i=1}^n \beta_i x_i + \sum_{i,j=1}^n \beta_{ij} x_i x_j + \epsilon$$

```
lm(y ~ (x1 + x2 + x3)^2)
```

these model descriptions are not unique, for example the last one is equivalent to

```
lm(y ~ x1 * x2 * x3 - x1:x2:x3)
```

Sometime we want \* to indicate actual multiplication and not interaction. This can be done with

```
lm(y ~ x1 + x2 + I(x1*x2))
```

Another useful one is ., which stands for *all +’s*, so say (y, x1, x2, x3) are the columns of a dataframe df, then

```
lm(y ~ x1 + x2 + x3, data=df)
```

is the same as

```
lm(y ~ ., data=df)
```

and

```
lm(y ~ .*x3, data=df)
```

is the same as

```
lm(y ~ x1 + x2 + x3 + x1*x3 +x2*x3)
```

### 15.0.1 Case Study

we have a list of prices and other information on houses in Albuquerque, New Mexico:

```
head(albuquerquehouseprice)
```

```
Price Sqfeet Feature Corner Tax
1 2050 2650 7 0 1639
2 2080 2600 4 0 1088
3 2150 2664 5 0 1193
4 2150 2921 6 0 1635
5 1999 2580 4 0 1732
6 1900 2580 4 0 1534
```

- additive model, all four predictors:

```

attach(albuquerquehouseprice)
summary(lm(Price ~ Sqfeet + Feature + Corner + Tax))

##
Call:
lm(formula = Price ~ Sqfeet + Feature + Corner + Tax)
##
Residuals:
Min 1Q Median 3Q
-541.9151883840 -73.6694900066 -12.8667004518 66.7386866962
Max
617.6778086492
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 76.6016037998690 60.7064418596984 1.26184 0.209885
Sqfeet 0.2667999639776 0.0616725731875 4.32607 3.5463e-05
Feature 13.6326113580407 13.2901318030308 1.02577 0.307427
Corner -89.0333805533419 42.2067494695365 -2.10946 0.037353
Tax 0.6619305456578 0.1088240774497 6.08257 2.0814e-08
##
Residual standard error: 171.047648266 on 102 degrees of freedom
(10 observations deleted due to missingness)
Multiple R-squared: 0.809064303949, Adjusted R-squared: 0.801576629594
F-statistic: 108.052816615 on 4 and 102 DF, p-value: < 2.220446049e-16
• additive model, Sqfeet and Features

summary(lm(Price ~ Sqfeet + Feature))

##
Call:
lm(formula = Price ~ Sqfeet + Feature)
##
Residuals:
Min 1Q Median 3Q
-1005.4032118154 -99.1425730556 -3.1552297695 75.9254360074
Max
781.9953419223
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.5588177016308 67.2958303004005 -0.02316 0.981560
Sqfeet 0.5842183005865 0.0390052774245 14.97793 < 2e-16
Feature 27.7858505794393 14.5344423142302 1.91172 0.058422
##
Residual standard error: 202.131447928 on 114 degrees of freedom

```

```

Multiple R-squared: 0.722572714369, Adjusted R-squared: 0.717705569008
F-statistic: 148.459242664 on 2 and 114 DF, p-value: < 2.220446049e-16
• model with interaction, Sqfeet and Features

summary(lm(Price ~ Sqfeet * Feature))

##
Call:
lm(formula = Price ~ Sqfeet * Feature)
##
Residuals:
Min 1Q Median 3Q
-987.9174314531 -98.8386424416 -0.5012577672 84.9693625323
Max
834.5872605298
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 396.3758257696671 172.4254022919212 2.29883 0.0233533
Sqfeet 0.3203005179411 0.1123713156903 2.85038 0.0051917
Feature -75.5710433105479 43.7668405798820 -1.72667 0.0869591
Sqfeet:Feature 0.0658465891098 0.0263720246208 2.49683 0.0139725
##
Residual standard error: 197.645040274 on 113 degrees of freedom
Multiple R-squared: 0.737078057607, Adjusted R-squared: 0.730097829048
F-statistic: 105.595117891 on 3 and 113 DF, p-value: < 2.220446049e-16

```

- model with pairwise interactions:

```

summary(lm(Price ~ (Sqfeet + Feature + Corner + Tax)^2))

##
Call:
lm(formula = Price ~ (Sqfeet + Feature + Corner + Tax)^2)
##
Residuals:
Min 1Q Median 3Q
-383.8692542868 -82.0322252256 -2.7033486544 56.0559234291
Max
644.2507021699
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.64208616519e+02 1.88358561215e+02 1.40269 0.1639351
Sqfeet 2.20423209861e-01 2.01160179692e-01 1.09576 0.2759253
Feature -3.65041896850e+01 4.60340291755e+01 -0.79298 0.4297434
Corner 4.36683964252e+02 1.55862761422e+02 2.80172 0.0061485
Tax 3.18047965786e-01 3.20493293867e-01 0.99237 0.3235117

```

```

Sqfeet:Feature 4.61790695050e-02 4.44127929699e-02 1.03977 0.3010587
Sqfeet:Corner -3.96333165362e-01 1.34686439656e-01 -2.94264 0.0040801
Sqfeet:Tax 1.10869019935e-04 1.06326136852e-04 1.04273 0.2996940
Feature:Corner -1.87535312229e+01 3.62567630738e+01 -0.51724 0.6061768
Feature:Tax -3.49205929379e-02 6.52763869864e-02 -0.53497 0.5939109
Corner:Tax 2.51096171352e-01 3.33181121888e-01 0.75363 0.4529143
##
Residual standard error: 156.80277528 on 96 degrees of freedom
(10 observations deleted due to missingness)
Multiple R-squared: 0.848981043431, Adjusted R-squared: 0.833249902122
F-statistic: 53.9681785791 on 10 and 96 DF, p-value: < 2.220446049e-16

```

- model with all possible terms:

```
summary(lm(Price ~ (Sqfeet + Feature + Corner + Tax)^4))
```

```

##
Call:
lm(formula = Price ~ (Sqfeet + Feature + Corner + Tax)^4)
##
Residuals:
Min 1Q Median 3Q
-379.5882862880 -80.0527937107 4.7408811793 55.1871201545
Max
650.8444026848
##
Coefficients:
Estimate Std. Error t value
(Intercept) 4.81103279366e+02 3.68356786083e+02 1.30608
Sqfeet 4.32121245230e-02 2.88604336160e-01 0.14973
Feature -8.40764259892e+01 9.75554953268e+01 -0.86183
Corner 2.49498924681e+03 2.26258986843e+03 1.10271
Tax 2.60830379790e-01 5.24694669197e-01 0.49711
Sqfeet:Feature 8.53642227168e-02 6.72565724382e-02 1.26923
Sqfeet:Corner -1.94127939214e+00 1.58956404358e+00 -1.22127
Sqfeet:Tax 1.93133568534e-04 2.55230235497e-04 0.75670
Feature:Corner -7.82061826535e+02 6.41862700790e+02 -1.21843
Feature:Tax -3.01996989181e-02 1.34921362745e-01 -0.22383
Corner:Tax -2.41006092234e+00 2.48411446014e+00 -0.97019
Sqfeet:Feature:Corner 5.43226683062e-01 4.32556093909e-01 1.25585
Sqfeet:Feature:Tax -1.47157919615e-05 5.93051852916e-05 -0.24814
Sqfeet:Corner:Tax 1.93309510062e-03 1.45983263033e-03 1.32419
Feature:Corner:Tax 9.27529470935e-01 7.04233090763e-01 1.31708
Sqfeet:Feature:Corner:Tax -6.34044336167e-04 3.95506738461e-04 -1.60312
Pr(>|t|)
(Intercept) 0.19482
Sqfeet 0.88131
Feature 0.39105

```

```

Corner 0.27306
Tax 0.62031
Sqfeet:Feature 0.20759
Sqfeet:Corner 0.22514
Sqfeet:Tax 0.45118
Feature:Corner 0.22621
Feature:Tax 0.82339
Corner:Tax 0.33452
Sqfeet:Feature:Corner 0.21238
Sqfeet:Feature:Tax 0.80459
Sqfeet:Corner:Tax 0.18876
Feature:Corner:Tax 0.19112
Sqfeet:Feature:Corner:Tax 0.11237
##
Residual standard error: 153.293168194 on 91 degrees of freedom
(10 observations deleted due to missingness)
Multiple R-squared: 0.86318310356, Adjusted R-squared: 0.840630867883
F-statistic: 38.2748351838 on 15 and 91 DF, p-value: < 2.220446049e-16

```

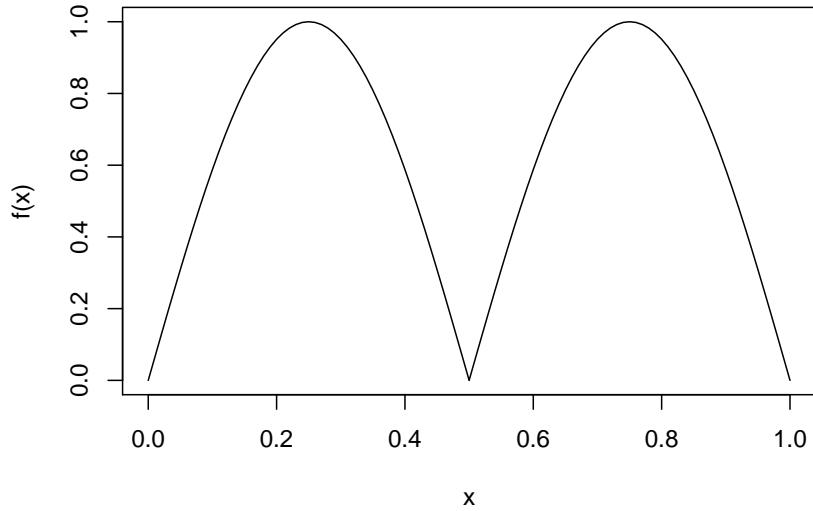
## 16 Numerical Methods

R has a number of routines useful for numerical analysis

### 16.1 Integration

the basic function for numerical integration is *integrate*

```
f <- function(x, a=2) abs(sin(a*pi*x))
curve(f, 0, 1)
```



```
integrate(f, 0, 1)
```

```
0.636619772367581 with absolute error < 7e-15
```

the routine returns a list, usually we only want the value of the integral, so run

```
integrate(f, 0, 1)$value
```

```
[1] 0.636619772367581
```

integrate allows us to pass additional arguments to f with the ... convention:

```
integrate(f, 0, 1, a=1.4)$value
```

```
[1] 0.611832671644403
```

Internally integrate subdivides the interval into 100 sub-intervals of equal length. Usually this is enough, but if the function has some sharp peaks this does not work very well. One solution is to increase the number of intervals:

```
integrate(f, 0, 1, subdivisions=1e4, a=1.4)$value
```

```
[1] 0.611832671644403
```

This again can be trouble if the evaluation of the function takes time. In that case you might want to write your own numerical integration function:

- simple Riemann sum

$$\int_a^b f(x)dx \approx \sum_{i=1}^n f(x_i^*)(x_i - x_{i-1})$$

where  $x_{i-1} \leq x_i^* \leq x_i$ .

```

x <- seq(0, 1, length=500)
y <- f(x)
mid <- (y[-1]+y[-500])/2
sum(mid)*(x[2]-x[1])

```

## [1] 0.63661766956831

- Simpson's Rule

$$\int_a^b f(x)dx \approx \frac{x_2 - x_1}{6} \sum_{i=1}^{n-1} f(x_{i-1} + 4f(x_i) + f(x_{i+1}))$$

```

sum(y[-1]+4*y[-2]+y[-3])*(x[2]-x[1])/6

```

## [1] 0.636592437327256

or any other standard numerical integration formula, see for example Numerical Integration Formulas

### 16.1.1 Double Integrals

R doesn't have a dedicated routine for double integrals but it is easy to use the integrate function for this as well using the fact that

$$\int \int f(x, y)d(x, y) = \int \left\{ \int f(x, y)dx \right\} dy$$

```

double.integral <-
function (f, low = c(0, 0), high = c(Inf, Inf))
{
 integrate(function(y) {
 sapply(y, function(y) {
 integrate(function(x) f(x, y), low[1], high[1])$value
 })
 }, low[2], high[2])$value
}

f <- function(x, y) exp(2*(-x^2+x*y-y^2)/3)
double.integral(f, low=c(-Inf, -Inf))

```

## [1] 5.44139794413109

Is this right? Let's check:

$$\exp(2(-x^2+xy-y^2)/3) = 2\pi\sqrt{1-(1/2)^2} \left[ \frac{1}{2\pi\sqrt{1-(1/2)^2}} \exp \left\{ -\frac{1}{2(1-(1/2)^2)} (x^2 - 2xy(1/2) + y^2) \right\} \right]$$

but the function inside the brackets is the density of a bivariate normal random variable with means  $(0, 0)$ , standard deviations  $(1, 1)$  and correlation coefficient  $1/2$ , so this integrates out to 1. Therefore the integral is

```
2*pi*sqrt(1-(1/2)^2)
[1] 5.44139809270265
```

## 16.2 Differentiation

We saw before the D function for finding derivatives:

```
f.prime <- D(expression(x^3), "x")
f.prime
3 * x^2
x <- 3.4; eval(f.prime)
[1] 34.68
f.prime <- D(expression(x^2*sin(2*pi*x)), "x")
f.prime
2 * x * sin(2 * pi * x) + x^2 * (cos(2 * pi * x) * (2 * pi))
x <- 0.4; eval(f.prime)
[1] -0.343084388936463
```

This of course works for higher order derivatives as well:

```
f.double.prime <- D(D(expression(x^3), "x"), "x")
f.double.prime
3 * (2 * x)
x <- 3.4; eval(f.double.prime)
[1] 20.4
f.double.prime <- D(D(expression(x^2*sin(2*pi*x)), "x"), "x")
f.double.prime
2 * sin(2 * pi * x) + 2 * x * (cos(2 * pi * x) * (2 * pi)) +
(2 * x * (cos(2 * pi * x) * (2 * pi)) - x^2 * (sin(2 * pi *
x) * (2 * pi) * (2 * pi)))
x <- 0.4; eval(f.double.prime)
[1] -10.670328467389
```

*Example* The Taylor polynomial of order k of a function f at a point  $x=a$  is defined by

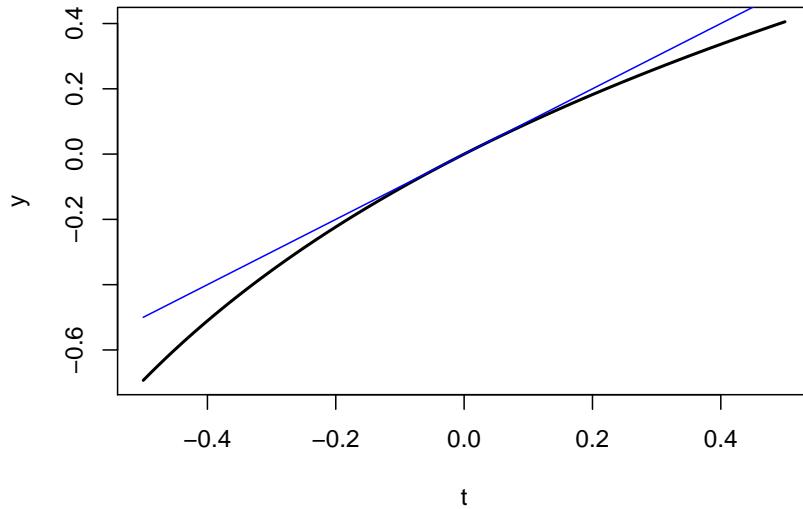
$$T(x; a, k) = \sum_{i=0}^k \frac{d^i f}{dx^i}(a)(x - a)^i$$

Let's write a function that draws the function and it's Taylor polynomial of order k:

```
taylor <- function(f, a, k=1, from, to) {
 expr <- parse(text=f)
 x <- seq(from, to, length=250)
 y <- eval(expr)
 t <- x
 x <- a
 z <- eval(expr)
 taylor.coefficients <- rep(z, k+1)
 x.text <- ifelse(a==0, "x", paste0("(x-", a, ")"))
 ttl <- paste0(f, " ~ ")
 if(z==0) nosgn <- TRUE
 else {
 nosgn <- FALSE
 ttl <- paste0(ttl, round(z, 2))
 }
 for(i in 1:k) {
 expr <- D(expr, "x")
 tmp <- eval(expr)
 z <- z + eval(expr)*(t-a)^i
 tmp <- round(tmp, 2)
 if(tmp!=0) {
 ttl <- paste0(ttl, " ",
 ifelse(nosgn, "", ifelse(tmp>0, "+", "-")),
 " ", ifelse(abs(tmp)==1, "", abs(tmp)),
 x.text,
 ifelse(i==1, "", paste0("^", i)))
 }
 nosgn <- FALSE
 }
 plot(t, y, type="l", lwd=2, main=ttl)
 lines(t, z, col="blue")
}

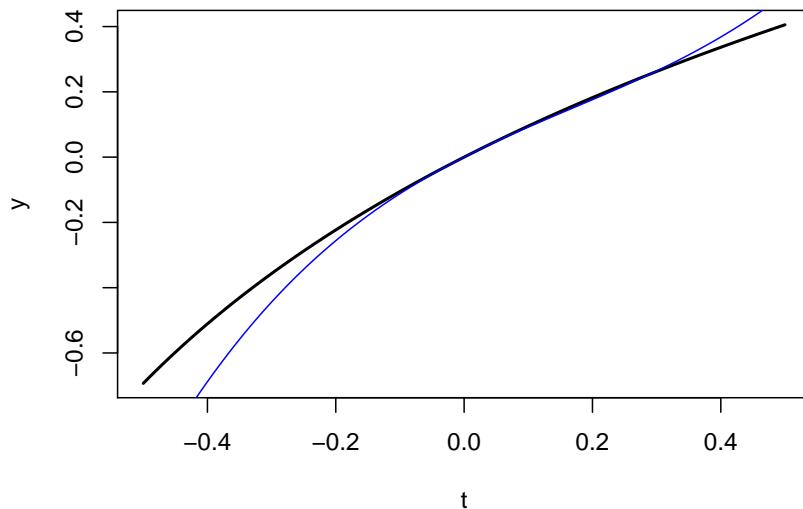
taylor("log(x+1)", a=0, k=1, -0.5, 0.5)
```

$$\log(x+1) \sim x$$



```
taylor("log(x+1)", a=0, k=3, -0.5, 0.5)
```

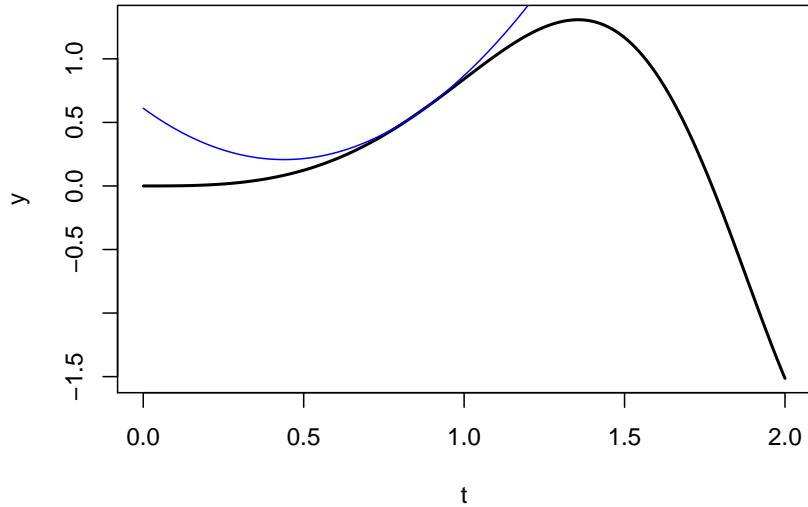
$$\log(x+1) \sim x - x^2 + 2x^3$$



Here is another example

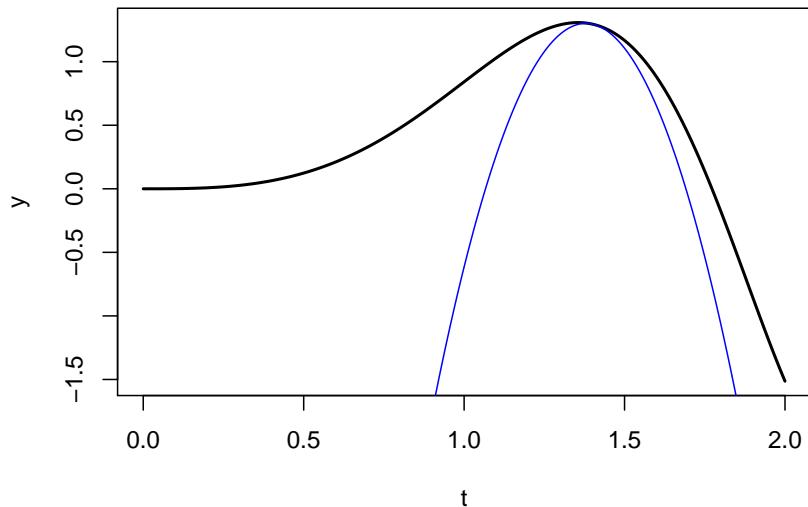
```
taylor("x*sin(x^2)", a=0.86, k=2, from=0, to=2)
```

$$x \cdot \sin(x^2) \sim 0.58 + 1.77(x - 0.86) + 2.1(x - 0.86)^2$$



```
taylor("x*sin(x^2)", a=1.4, k=2, from=0, to=2)
```

$$x \cdot \sin(x^2) \sim 1.3 - 0.56(x - 1.4) - 13.34(x - 1.4)^2$$



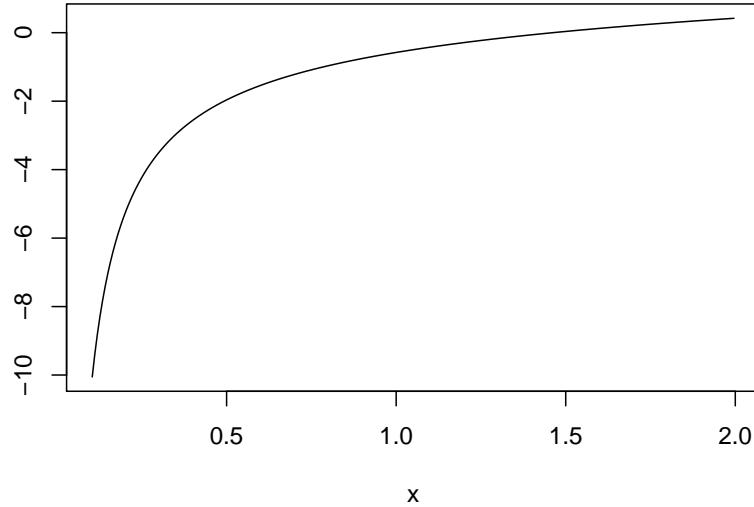
The D function only works on functions that R recognizes. For other cases you might have to write your own:

```
f <- function(x) log(gamma(x))
x <- seq(0.1, 2, length=250)
h <- (x[2]-x[1])
y <- f(x)
y.prime <- (y[-1]-y[-250])/h
```

```

mid <- (x[-1]+x[-250])/2
plot(mid, y.prime, type="l",
 xlab="x", ylab="")

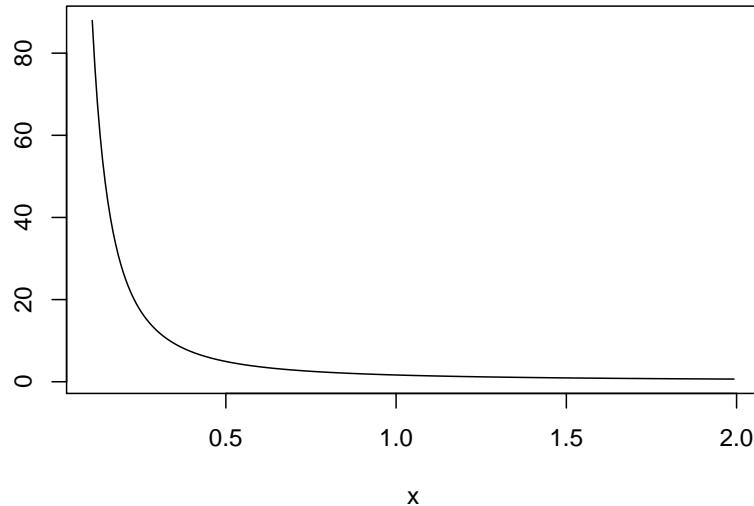
```



```

#Second derivative
y.2.prime <- rep(0, 248)
for(i in 2:249)
 y.2.prime[i-1] <- (y[i-1]-2*y[i]+y[i+1])/h^2
plot(x[-c(1, 250)], y.2.prime, type="l",
 xlab="x", ylab="")

```

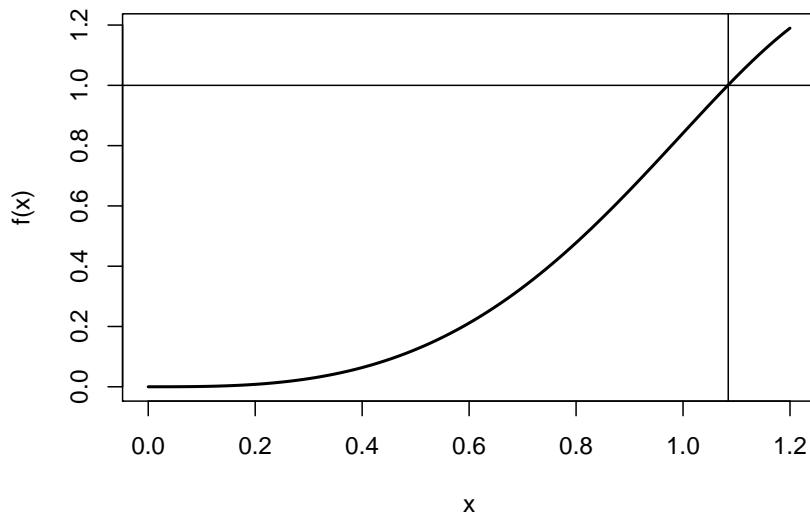


## 16.3 Root Finding

A very common problem is to have to solve an equation of the form  $f(x) = a$ . As a specific example we will consider  $x \sin(x^2) = 1$ . Here are some ideas:

### 16.3.1 Direct Method (Grid Search)

```
f <- function(x) x*sin(x^2)
x <- seq(0, 1.2, length=500)
y <- f(x)
curve(f, 0, 1.2, lwd=2)
abline(h=1)
y0 <- x[abs(y-1)==min(abs(y-1))]
abline(v=y0)
```

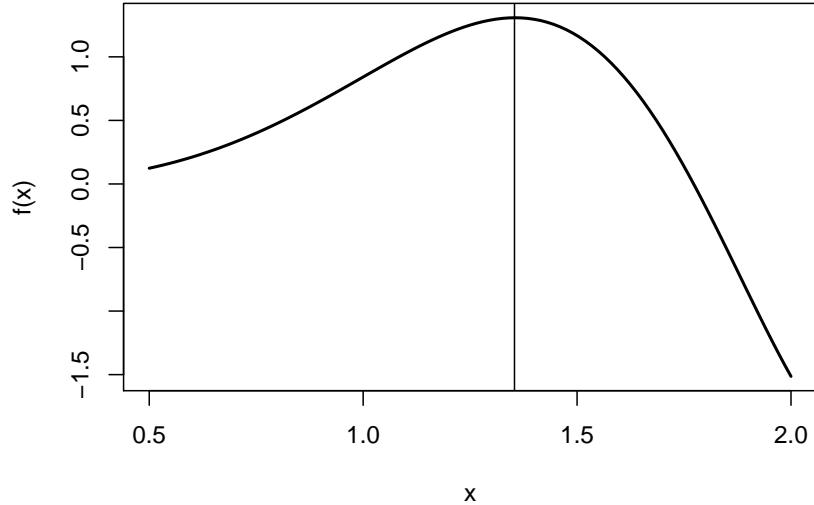


```
y0
```

```
[1] 1.08456913827655
```

Of course we can combine that with D to find extrema of a function:

```
f <- function(x) x*sin(x^2)
x <- seq(0.5, 2, length=500)
y <- f(x)
curve(f, 0.5, 2, lwd=2)
y.prime <- eval(D(expression(x*sin(x^2)), "x"))
x0 <- x[abs(y.prime)==min(abs(y.prime))]
abline(v=x0)
```



```
c(x0, f(x0))
```

```
[1] 1.35370741482966 1.30760615848904
```

An alternative is to use *optimize*. By default it finds minima, so

```
optimize(f, c(0.5, 2), maximum = TRUE)
```

```
$maximum
[1] 1.35521700211705
##
$objective
[1] 1.3076194127889
```

### 16.3.2 uniroot

R has the function *uniroot* to find the roots of a univariate function:

```
f <- function(x) x*sin(x^2)-1
uniroot(f, c(0, 1.2))$root
[1] 1.08387985197034
```

### 16.3.3 polyroot

In the case of a polynomial one can also use *polyroot*. Say we want to find the roots of

$$p(x) = 1 + x - x^2 + x^4$$

```
polyroot(c(1, 1, -1, 0, 1))
```

```
[1] 0.877438833123346+0.744861766619744i
[2] -0.754877666246693-0.0000000000000000i
[3] -1.000000000000000+0.0000000000000000i
[4] 0.877438833123346-0.744861766619744i
```

as we can see that gieves also the complex solutions. Often we only want the real ones:

```
z <- polyroot(c(1, 1, -1, 0, 1))
z <- Re(z[round(Im(z), 10)==0])
z

[1] -0.754877666246693 -1.0000000000000000
```

#### 16.3.4 Higher Dimensional Optimization

If we have a function of more than one variable we can use *nlm*. Again it finds minima, but there is no argument maximum=TRUE, so if we want a maximum we have to use -f.

Say we want to maximize

$$f(x, y) = e^{-(x-1)^2-(y-2)^2}$$

(of course it is obvious the answer is 1, 2). Now

```
f <- function(x) -exp(-(x[1]-1)^2-(x[2]-2)^2)
nlm(f, c(0, 0))
```

```
$minimum
[1] -0.99999999998743
##
$estimate
[1] 0.999999576040224 1.999998961967737
##
$gradient
[1] 1.52100554373646e-07 -7.60503166581643e-08
##
$code
[1] 1
##
$iterations
[1] 16
```

Quite useful is the ability to calculate the Hessian matrix, that is  $H = (h_{ij})$  and

$$h_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

```

nlm(f, c(0, 0), hessian = TRUE)

$minimum
[1] -0.999999999998743
##
$estimate
[1] 0.999999576040224 1.999998961967737
##
$gradient
[1] 1.52100554373646e-07 -7.60503166581643e-08
##
$hessian
[,1] [,2]
[1,] 1.9999993233435e+00 -1.66533540127557e-08
[2,] -1.66533540127557e-08 1.99999972134229e+00
##
$code
[1] 1
##
$iterations
[1] 16

```

This works well as long as we can play a bit with the starting point but can be quite hard to do in a simulation.

An alternative is *optim*, which let's us define boundaries within which the optimum is to be found. It also has a choice of methods, and sometimes one will work where the others do not.

## 17 Environments, Libraries

### 17.1 Environments

WARNING: In what follows I will only discuss a FEW of the issues involved with environments, and I will simplify them greatly. For a much more detailed discussion see <http://adv-r.had.co.nz/Environments.html>.

Let's start with this:

```

search()

[1] ".GlobalEnv" "albuquerquehouseprice"
[3] "package:maps" "mtcars"
[5] "faithful" "mothers"
[7] "package:Hmisc" "package:Formula"
[9] "package:survival" "package:lattice"
[11] "package:mvtnorm" "package:random"
[13] "package:stringr" "package:knitr"
[15] "package:bookdown" "tools:rstudio"

```

```

[17] "package:stats" "package:graphics"
[19] "package:grDevices" "package:utils"
[21] "package:datasets" ".MyEnv"
[23] "package:moodler" "package:wolfr"
[25] "package:shiny" "package:Rcpp"
[27] "package:grid" "package:ggplot2"
[29] "package:methods" "Autoloads"
[31] "package:base"

```

These are the environments currently loaded into my R. In some ways you can think of this as a folder tree, like:

```
paste0(search(), "/", collapse="")
```

```
[1] ".GlobalEnv/albuquerquehouseprice/package:maps/mtcars/faithful/mothers/package:Hm
```

This has the following effect. Say you type

```
x <- runif(10)
mean(x)
```

```
[1] 0.394534953567199
```

What has R just done? First it created an object called “x”, and stored it in the folder “.GlobalEnv”. We can check:

```
ls()
```

```

[1] "%+b%" "a"
[3] "A" "acorn"
[5] "add.tags" "Age"
[7] "agesex" "agesexUS"
[9] "aids" "airfilters"
[11] "airpollution" "albuquerquehouseprice"
[13] "alcohol" "arrange.arguments"
[15] "as.binary" "as.sales"
[17] "babe" "balance"
[19] "barchart" "barley"
[21] "bayes.prop" "berkeleyadmissions"
[23] "binary.2.decimal" "binary.addition"
[25] "binary_addition" "binner"
[27] "birthrate" "birthweight"
[29] "bloodpressure" "bplot"
[31] "brainandiq" "brainsize"
[33] "butterflies" "calcium"
[35] "cancersurvival" "cardeaths"
[37] "cateyes" "cats"
[39] "causeofdeath" "change.order"
[41] "check.packages" "cheese"
[43] "chi.gof.test" "chi.ind.test"
[45] "chromatography" "church"

```

```

[47] "ci.mean.sim" "classsurvey"
[49] "clouds" "clt.illustration"
[51] "cols" "company"
[53] "cont.table" "countries"
[55] "crop" "cuckoo"
[57] "culture" "decimal.2.binary"
[59] "df" "df1"
[61] "diamond" "dlr"
[63] "dlr.predict" "double.integral"
[65] "dp" "draft"
[67] "draw.cause.effect" "draw.hist"
[69] "dropcourse" "drownings"
[71] "drugaddiction" "elusage"
[73] "esh" "euros"
[75] "examscores" "f"
[77] "f.double.prime" "f.prime"
[79] "fA" "fA.vec"
[81] "fabricwear" "faithful"
[83] "fAvec" "fermentation"
[85] "fiber" "filmcoatings"
[87] "find.data.set" "fish"
[89] "fivenumber" "flammability"
[91] "flplot" "foods"
[93] "forbes" "friday13"
[95] "galton" "gasoline"
[97] "gen.cont.table.data" "Gender"
[99] "gestation" "get.moodle.data"
[101] "getfun" "getx"
[103] "ggMarginal" "ggQQ"
[105] "golfscores" "GPA"
[107] "graph.out" "gunsandmurder"
[109] "h" "headache"
[111] "hearingaid" "highways"
[113] "homework" "horsekicks"
[115] "hostility" "hotdogs"
[117] "houseprice" "hpd.beta"
[119] "hplot" "hubble"
[121] "hyptest.helper" "i"
[123] "I" "ibayesprop"
[125] "ID" "idataio"
[127] "igetdata" "ihist"
[129] "ihplot" "info"
[131] "inormal" "instructor"
[133] "intplot" "iplot"
[135] "iplotone" "iq"
[137] "iqandsize" "is.binary"

```

```

[139] "isCat" "isplot"
[141] "issubset" "issummary"
[143] "itemplate" "k"
[145] "kruskalwallis" "larvea"
[147] "lobster" "longjump"
[149] "lunatics" "make.html.table"
[151] "mallows" "mannwhitney"
[153] "mcat" "mendel"
[155] "mid" "mines"
[157] "mlr" "mlr.predict"
[159] "mothers" "moths"
[161] "mplot" "multiple.graphs"
[163] "mycurve" "n"
[165] "nested.models.test" "newcomb"
[167] "normal.check" "normal.ex"
[169] "nplot" "num_a"
[171] "num_vowels" "olympics"
[173] "one.sample.prop" "one.sample.t"
[175] "one.sample.wilcoxon" "one.time.setup"
[177] "oneway" "p"
[179] "pcbtrout" "pearson.cor"
[181] "plot.sales" "plot.salestime"
[183] "pop1950x2010" "popular"
[185] "popularvote" "population"
[187] "positions" "positions_a"
[189] "print.binary" "prop.ps"
[191] "race.table" "ratings"
[193] "report" "rev.wrds"
[195] "rice" "rocks"
[197] "rogaine" "run.app"
[199] "salaries" "sales.data"
[201] "sales.time.data" "salesperson"
[203] "sc" "seatbelt"
[205] "sexratio" "shoesales"
[207] "sim1" "sim2"
[209] "singer" "skulls"
[211] "sleep" "slr"
[213] "slr.predict" "smoking"
[215] "smokingandjob" "splot"
[217] "st.y" "stat.table"
[219] "states" "stats"
[221] "stats.sales" "student.levels"
[223] "students" "students.fac"
[225] "students.ord" "studentsurvey"
[227] "studyhabits" "sugar"
[229] "summary" "summary.binary"

```

```

[231] "sv" "t.ps"
[233] "taylor" "tbl.students.fac"
[235] "test.mean.sim" "test_sim"
[237] "titanic" "tms"
[239] "trout" "tukey"
[241] "tv" "twoway"
[243] "txt" "u"
[245] "upr" "us.population.2010"
[247] "usmap" "ustemperature"
[249] "vowels" "whichcomp"
[251] "wilcoxon" "wine"
[253] "world.mortality.2017" "worldpopulation"
[255] "worldseries" "wrds"
[257] "wrinccensus" "wrinccsample"
[259] "x" "x0"
[261] "y" "y.2.prime"
[263] "y.prime" "y0"
[265] "z"

```

Next R starts looking for an object called *mean*. To do that it again first looks into “`.GlobalEnv`”, but we already know it is not there.

Next R looks into “`package:knitr`”, which we can do with

```
ls(2, pattern="mean")
```

```
character(0)
```

and again no luck. This continues until we get to “`package:base`”:

```
ls(17, pattern="mean")
```

```
[1] "kmeans" "weighted.mean"
```

and there it is!

If an object is not in any of these environments it will give an error:

```
ddgdg
```

```
Error in eval(expr, envir, enclos): object 'ddgdg' not found
```

This makes it clear that a routine that is part of a library can only be found if that library is loaded.

One difference between a folder tree and this is that R starts looking at the top (in `.GobalEnv`) and then works its way down.

There is an easy way to find out in which environment an object is located, with the routine *where* in the library *pryr*:

```
library(pryr)
where("x")
```

```

<environment: R_GlobalEnv>
where("mean")

<environment: base>

Another important consequence of this is that R stops when it finds an object, even if the one you want is in a later environment. Here is an example:

my.data <- data.frame(x=1:10)
attach(my.data)

The following object is masked _by_ .GlobalEnv:
##
x
mean(x)

[1] 0.394534953567199
rm(x)
mean(x)

[1] 5.5
search()[1:3]

[1] ".GlobalEnv" "my.data" "package:pryr"

```

So here is what happens:

- the first time we call mean(x) R finds an x (the original one) in .GlobalEnv, and so calculates its mean.
- after removing this x, the next time we call mean(x) it looks into the data frame my.data, finds a variable called x, and now calculates its mean.

Notice that R gives a warning when we attach the data frame, telling us that there are now two x's.

The rules that R uses to find things are called *scoping rules*.

Let's clean up before we continue:

```
detach(2)
```

### 17.1.1 runtime environments

How does this work when we run a function? To find out we can write a little function:

```

show.env <- function(){
 x <- 1
 print(list(ran.in=environment(),
 parent=parent.env(environment())),

```

```

 objects=ls.str(environment())))
}

show.env()

$ran.in
<environment: 0x0000000016d22558>
##
$parent
<environment: R_GlobalEnv>
##
$objects
x : num 1

```

this tells us that R ran the function in an environment with a very strange name, which usually means it was created randomly. We can also see that its parent environment was .GlobalEnv and that x is an object in it.

This means that any object created inside a function is only known there, it does not overwrite any objects outside the function. One consequence is that if we need to create some temporary objects we can use simple names like x or i, even if these already exist outside of the function.

Now where does *show.env* live?

```
environment(show.env)
```

```
<environment: R_GlobalEnv>
```

Obvious, because that is where we created it!

How about a function inside a function?

```

show.env <- function() {
 f <- function(){
 print(list(ran.in=environment(),
 parent=parent.env(environment()),
 objects=ls.str(environment())))
 }
 f()
 x <- 1
 print(list(ran.in=environment(),
 parent=parent.env(environment()),
 objects=ls.str(environment())))
}

show.env()

$ran.in
<environment: 0x00000000eb739c8>
##
```

```

$parent
<environment: 0x00000000eb73878>
##
$objects
##
$ran.in
<environment: 0x00000000eb73878>
##
$parent
<environment: R_GlobalEnv>
##
$objects
f : function ()
x : num 1

```

As we expect, the parent environment of f is the runtime environment of show.env.

Sometimes we want to save an object created inside a function to the global environment:

```

f <- function() {
 a<-1
 assign("a", a, envir=.GlobalEnv)

}

ls()

[1] "%+b%" "a"
[3] "A" "acorn"
[5] "add.tags" "Age"
[7] "agesex" "agesexUS"
[9] "aids" "airfilters"
[11] "airpollution" "albuquerquehouseprice"
[13] "alcohol" "arrange.arguments"
[15] "as.binary" "as.sales"
[17] "babe" "balance"
[19] "barchart" "barley"
[21] "bayes.prop" "berkeleyadmissions"
[23] "binary.2.decimal" "binary.addition"
[25] "binary_addition" "binner"
[27] "birthrate" "birthweight"
[29] "bloodpressure" "bplot"
[31] "brainandiq" "brainsize"
[33] "butterflies" "calcium"
[35] "cancersurvival" "cardeaths"
[37] "cateyes" "cats"
[39] "causeofdeath" "change.order"
[41] "check.packages" "cheese"
[43] "chi.gof.test" "chi.ind.test"

```

```

[45] "chromatography" "church"
[47] "ci.mean.sim" "classsurvey"
[49] "clouds" "clt.illustration"
[51] "cols" "company"
[53] "cont.table" "countries"
[55] "crop" "cuckoo"
[57] "culture" "decimal.2.binary"
[59] "df" "df1"
[61] "diamond" "dlr"
[63] "dlr.predict" "double.integral"
[65] "dp" "draft"
[67] "draw.cause.effect" "draw.hist"
[69] "dropcourse" "drownings"
[71] "drugaddiction" "elusage"
[73] "esh" "euros"
[75] "examscores" "f"
[77] "f.double.prime" "f.prime"
[79] "fA" "fA.vec"
[81] "fabricwear" "faithful"
[83] "fAvec" "fermentation"
[85] "fiber" "filmcoatings"
[87] "find.data.set" "fish"
[89] "fivenumber" "flammability"
[91] "flplot" "foods"
[93] "forbes" "friday13"
[95] "galton" "gasoline"
[97] "gen.cont.table.data" "Gender"
[99] "gestation" "get.moodle.data"
[101] "getfun" "getx"
[103] "ggMarginal" "ggQQ"
[105] "golfscores" "GPA"
[107] "graph.out" "gunsandmurder"
[109] "h" "headache"
[111] "hearingaid" "highways"
[113] "homework" "horsekicks"
[115] "hostility" "hotdogs"
[117] "houseprice" "hpd.beta"
[119] "hplot" "hubble"
[121] "hyptest.helper" "i"
[123] "I" "ibayesprop"
[125] "ID" "idataio"
[127] "igetdata" "ihist"
[129] "ihplot" "info"
[131] "inormal" "instructor"
[133] "intplot" "iplot"
[135] "iplotone" "iq"

```

```

[137] "iqandsize" "is.binary"
[139] "isCat" "isplot"
[141] "issubset" "issummary"
[143] "itemplate" "k"
[145] "kruskalwallis" "larvea"
[147] "lobster" "longjump"
[149] "lunatics" "make.html.table"
[151] "mallows" "mannwhitney"
[153] "mcat" "mendel"
[155] "mid" "mines"
[157] "mlr" "mlr.predict"
[159] "mothers" "moths"
[161] "mplot" "multiple.graphs"
[163] "my.data" "mycurve"
[165] "n" "nested.models.test"
[167] "newcomb" "normal.check"
[169] "normal.ex" "nplot"
[171] "num_a" "num_vowels"
[173] "olympics" "one.sample.prop"
[175] "one.sample.t" "one.sample.wilcoxon"
[177] "one.time.setup" "oneway"
[179] "p" "pcbtrout"
[181] "pearson.cor" "plot.sales"
[183] "plot.salestime" "pop1950x2010"
[185] "popular" "popularvote"
[187] "population" "positions"
[189] "positions_a" "print.binary"
[191] "prop.ps" "race.table"
[193] "ratings" "report"
[195] "rev.wrds" "rice"
[197] "rocks" "rogaine"
[199] "run.app" "salaries"
[201] "sales.data" "sales.time.data"
[203] "salesperson" "sc"
[205] "seatbelt" "sexratio"
[207] "shoessales" "show.env"
[209] "sim1" "sim2"
[211] "singer" "skulls"
[213] "sleep" "slr"
[215] "slr.predict" "smoking"
[217] "smokingandjob" "splot"
[219] "st.y" "stat.table"
[221] "states" "stats"
[223] "stats.sales" "student.levels"
[225] "students" "students.fac"
[227] "students.ord" "studentsurvey"

```

```

[229] "studyhabits" "sugar"
[231] "summary" "summary.binary"
[233] "sv" "t.ps"
[235] "taylor" "tbl.students.fac"
[237] "test.mean.sim" "test_sim"
[239] "titanic" "tms"
[241] "trout" "tukey"
[243] "tv" "twoway"
[245] "txt" "u"
[247] "upr" "us.population.2010"
[249] "usmap" "ustemperature"
[251] "vowels" "whichcomp"
[253] "wilcoxon" "wine"
[255] "world.mortality.2017" "worldpopulation"
[257] "worldseries" "wrds"
[259] "wrinccensus" "wrincsample"
[261] "x0" "y"
[263] "y.2.prime" "y.prime"
[265] "y0" "z"

```

```

f()
ls()

```

```

[1] "%+b%" "a"
[3] "A" "acorn"
[5] "add.tags" "Age"
[7] "agesex" "agesexUS"
[9] "aids" "airfilters"
[11] "airpollution" "albuquerquehouseprice"
[13] "alcohol" "arrange.arguments"
[15] "as.binary" "as.sales"
[17] "babe" "balance"
[19] "barchart" "barley"
[21] "bayes.prop" "berkeleyadmissions"
[23] "binary.2.decimal" "binary.addition"
[25] "binary_addition" "binner"
[27] "birthrate" "birthweight"
[29] "bloodpressure" "bplot"
[31] "brainandiq" "brainsize"
[33] "butterflies" "calcium"
[35] "cancersurvival" "cardeaths"
[37] "cateyes" "cats"
[39] "causeofdeath" "change.order"
[41] "check.packages" "cheese"
[43] "chi.gof.test" "chi.ind.test"
[45] "chromatography" "church"
[47] "ci.mean.sim" "classsurvey"

```

```

[49] "clouds" "clt.illustration"
[51] "cols" "company"
[53] "cont.table" "countries"
[55] "crop" "cuckoo"
[57] "culture" "decimal.2.binary"
[59] "df" "df1"
[61] "diamond" "dlr"
[63] "dlr.predict" "double.integral"
[65] "dp" "draft"
[67] "draw.cause.effect" "draw.hist"
[69] "dropcourse" "drownings"
[71] "drugaddiction" "elusage"
[73] "esh" "euros"
[75] "examscores" "f"
[77] "f.double.prime" "f.prime"
[79] "fA" "fA.vec"
[81] "fabricwear" "faithful"
[83] "fAvec" "fermentation"
[85] "fiber" "filmcoatings"
[87] "find.data.set" "fish"
[89] "fivenumber" "flammability"
[91] "flplot" "foods"
[93] "forbes" "friday13"
[95] "galton" "gasoline"
[97] "gen.cont.table.data" "Gender"
[99] "gestation" "get.moodle.data"
[101] "getfun" "getx"
[103] "ggMarginal" "ggQQ"
[105] "golfscores" "GPA"
[107] "graph.out" "gunsandmurder"
[109] "h" "headache"
[111] "hearingaid" "highways"
[113] "homework" "horsekicks"
[115] "hostility" "hotdogs"
[117] "houseprice" "hpdbeta"
[119] "hplot" "hubble"
[121] "hyptest.helper" "i"
[123] "I" "ibayesprop"
[125] "ID" "idataio"
[127] "igetdata" "ihist"
[129] "ihplot" "info"
[131] "inormal" "instructor"
[133] "intplot" "iplot"
[135] "iplotone" "iq"
[137] "iqandsize" "is.binary"
[139] "isCat" "isplot"

```

```

[141] "isubset" "isummary"
[143] "itemplate" "k"
[145] "kruskalwallis" "larvea"
[147] "lobster" "longjump"
[149] "lunatics" "make.html.table"
[151] "mallows" "mannwhitney"
[153] "mcat" "mendel"
[155] "mid" "mines"
[157] "mlr" "mlr.predict"
[159] "mothers" "moths"
[161] "mplot" "multiple.graphs"
[163] "my.data" "mycurve"
[165] "n" "nested.models.test"
[167] "newcomb" "normal.check"
[169] "normal.ex" "nplot"
[171] "num_a" "num_vowels"
[173] "olympics" "one.sample.prop"
[175] "one.sample.t" "one.sample.wilcoxon"
[177] "one.time.setup" "oneway"
[179] "p" "pcbtrout"
[181] "pearson.cor" "plot.sales"
[183] "plot.salestime" "pop1950x2010"
[185] "popular" "popularvote"
[187] "population" "positions"
[189] "positions_a" "print.binary"
[191] "prop.ps" "race.table"
[193] "ratings" "report"
[195] "rev.wrds" "rice"
[197] "rocks" "rogaine"
[199] "run.app" "salaries"
[201] "sales.data" "sales.time.data"
[203] "salesperson" "sc"
[205] "seatbelt" "sexratio"
[207] "shoessales" "show.env"
[209] "sim1" "sim2"
[211] "singer" "skulls"
[213] "sleep" "slr"
[215] "slr.predict" "smoking"
[217] "smokingandjob" "splot"
[219] "st.y" "stat.table"
[221] "states" "stats"
[223] "stats.sales" "student.levels"
[225] "students" "students.fac"
[227] "students.ord" "studentsurvey"
[229] "studyhabits" "sugar"
[231] "summary" "summary.binary"

```

```

[233] "sv" "t.ps"
[235] "taylor" "tbl.students.fac"
[237] "test.mean.sim" "test_sim"
[239] "titanic" "tms"
[241] "trout" "tukey"
[243] "tv" "twoway"
[245] "txt" "u"
[247] "upr" "us.population.2010"
[249] "usmap" "ustemperature"
[251] "vowels" "whichcomp"
[253] "wilcoxon" "wine"
[255] "world.mortality.2017" "worldpopulation"
[257] "worldseries" "wrds"
[259] "wrinccensus" "wrincsample"
[261] "x0" "y"
[263] "y.2.prime" "y.prime"
[265] "y0" "z"

```

One place where this is useful is if you have a routine like a simulation that runs for a long time and you want to save intermediate results.

As we just saw, environments can come about by loading libraries, by attaching data frames (also lists) and (at least for a short while) by running a function. In fact we can also make our own:

```

test_env <- new.env()
attach(test_env)
search()[1:3]

[1] ".GlobalEnv" "test_env" "package:pryr"

```

Now we can add stuff to our environment using the list notation:

```

test_env$a <- 1
test_env$fun <- function(x) x^2
ls(2)

```

```
character(0)
```

Where are a and fun? Ops, we forgot to attach test\_env:

```

attach(test_env)
ls(2)

```

```

[1] "a" "fun"

search()[1:3]

```

```
[1] ".GlobalEnv" "test_env" "test_env"
```

note that we had to attach the environment again for the two new objects to be useful, but now we have two of them. It would be better if we detached it first.

Actually, let's detach it completely

```
detach(2)
search()

[1] ".GlobalEnv" "package:pryr"
[3] "albuquerquehouseprice" "package:maps"
[5] "mtcars" "faithful"
[7] "mothers" "package:Hmisc"
[9] "package:Formula" "package:survival"
[11] "package:lattice" "package:mvtnorm"
[13] "package:random" "package:stringr"
[15] "package:knitr" "package:bookdown"
[17] "tools:rstudio" "package:stats"
[19] "package:graphics" "package:grDevices"
[21] "package:utils" "package:datasets"
[23] ".MyEnv" "package:moodler"
[25] "package:wolfr" "package:shiny"
[27] "package:Rcpp" "package:grid"
[29] "package:ggplot2" "package:methods"
[31] "Autoloads" "package:base"
```

Why would you want to make a new environment? I have one called `.MyEnv` that is created at startup. It has a set of small functions that I like to have available at all times but I don't want to "see" them when I run `ls()`.

```
ls(".MyEnv")

[1] "dp" "h" "hh" "ht" "ip" "mcat" "s" "sc" "sr" "trw"
```

If an object is part of a package that is installed on your computer you can also use it without loading the package with the `::` operator. As an example consider the package `mailR`, which has the function `send.mail` to send emails from within R:

```
args(mailR::send.mail)

function (from, to, subject = "", body = "", encoding = "iso-8859-1",
html = FALSE, inline = FALSE, smtp = list(), authenticate = FALSE,
send = TRUE, attach.files = NULL, debug = FALSE, ...)
NULL
```

Some R texts suggest to avoid using `attach` at all, and to always use `::`. The reason is that what works on your computer with its specific setup may not work on someone else's. My preference is to use `::` if I use a function in this package just once but to attach the package if I use the function several times.

## 17.2 Packages

As we have already seen, packages/libraries are at the heart of R. Mostly it is where we can find routines already written for various tasks. The main repository is at [https:](https://)

//cran.r-project.org/web/packages/. Currently there are over 12000!

In fact, that is one problem: for any one task there are likely a dozen packages that would work. Finding the one that works for you is not easy!

Once you decide which one you want you can download it easily by clicking on the Packages tab in RStudio, select Install and typing the name. Occasionally RStudio won't find it, then you can do it manually:

```
install.packages("pckname")
```

Useful arguments are

- lib: the folder on your hard drive where you want to store the package (usually c:/R/lib).
- repos: the place on the internet where the package is located (if not it pops up a list to choose from).
- dependencies=TRUE will also download any additional packages required.

Notice that this only downloads the package, you still have to load it into R:

```
library(mypcks)
```

If you install a new version of R you want to update all the packages:

```
update.packages(ask=FALSE)
```

### 17.3 Creating your own library

It has been said that *as soon as your project has two functions, make a library*. While that might be a bit extreme, putting a collection of routines and data sets into a common library certainly is worthwhile. Here are the main steps to do so:

First we need a couple of libraries. If you are using RStudio (and you really should when creating a library), you likely have them already. If not get them as usual:

```
install.packages("devtools")
library(devtools)
devtools::install_github("klutometis/roxygen")
library(roxygen2)
```

First let's make a new folder for our project and a folder called R inside of it:

```
create("../testlib")
```

Open an explorer window and go to the folder testlib

Open the file DESCRIPTION. It looks like this:

Package: testlib

Title: What the Package Does (one line, title case)

```
Version: 0.0.0.9000
Authors@R: person("First", "Last", email = "first.last@example.com", role = c("aut", "cre"))
Description: What the package does (one paragraph).
Depends: R (>= 3.4.3)
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

and so we can change it to

```
Package: testlib
Title: Test Library Version: 0.0.0.9000
Authors@R: person("W", "R", email = "w.r@gmail.com", role = c("aut", "cre"))
Description: Let's us learn how to make our own libraries
Depends: R (>= 3.4.3)
License: Free
Encoding: UTF-8
LazyData: true
```

Next we have to put the functions we want to have in our library into the R folder:

```
f1 <- function(x) x^2
f2 <- function(x) sqrt(x)
dump("f1", "../testlib/R/f1.R")
dump("f2", "../testlib/R/f2.R")
```

Let's change the working directory to testlib and check what we have in there:

```
setwd("../testlib")
dir()

[1] "Data" "DESCRIPTION" "NAMESPACE" "R"
[5] "testlib.Rproj"

dir("R")

[1] "f1.R" "f2.R"
```

Often we also want some data sets as part of the library:

```
test.x <- 1:10
test.y <- c(2, 3, 7)
use_data(test.x, test.y)
dir("Data")

[1] "test.x.rda" "test.y.rda"
```

Notice that this saves the data in the .rda format, which is good because this format can be read by R very fast.

In the next step we need to add comments to the functions.

Eventually these are the things will appear in the help files. They are

```
#'f1 Function
#'
#'This function finds the square.
#' @param x.
#' @keywords square
#' @export
#' @examples
#' f1(2)
```

and the corresponding one for f2.

Now we need to process the documentation:

```
document()
```

One step left. You need to do this one from the parent working directory that contains the testlib folder.

```
setwd("..")
install("testlib")
```

Let's check:

```
library(testlib)
search()[1:4]

[1] ".GlobalEnv" "package:testlib" "package:roxygen2"
[4] "package:devtools"

ls(2)

[1] "f1" "f2" "test.x" "test.y"

f1(2)

[1] 4

f2(2)

[1] 1.4142135623731
```

And that's it!

I have several libraries that I often change, so I wrote a small routine to make it easy:

```
' make.library
'
#' This function creates a library called name in folder
#' @param name name of library
```

```

#' @param folder with R routines
#' @export
#' @examples
#' make.library("moodler")

make.library <- function (name, folder)
{
 whichcomp <- strsplit(getwd(), "/")[[1]][3]
 if(name=="moodler")
 folder <- paste0("C:/users/", whichcomp, "/Dropbox/teaching/moodle/moodler")
 if(name=="wolfr")
 folder <- paste0("C:/users/", whichcomp, "/Dropbox/R/wolfr")
 if(folder=="Resma3")
 folder <- paste0("C:/users/", whichcomp, "/Dropbox/teaching/Resma3/Resma3")
 library(devtools) #make sure packages are loaded
 library(roxygen2)
 oladdir <- getwd() #remember where you are
 setwd(folder) #go where you need to be
 document() #make lib
 setwd("../")
 install(name)
 setwd(oladdir) #go back
}

}

```

so when I make a change to one of the routines in (say) wolfr all I need to do is run

```
make.library("wolfr")
```

## 18 Costumizing R: .First, .Rprofile, Dropbox

There are quite a few things that one might want to change from the defaults of R. For example, I prefer a certain editor when writing a function, and there are a number of libraries that I will need sooner or later, so I would like to have them loaded. Also, over the years I have written a number of small routines that I use for various tasks, and I want them available at any time. The two routines to set up things the way I want are

- .First, to set up stuff specific to the project I am working on.
- .Rprofile, to set up stuff I need regardless of the current project.

Note that the . in front means that you can't see this object when you run `ls()`.

The .First is part of the .RData file whereas the .Rprofile is separate.

When R starts it looks for a file called .Rprofile and executes any commands therein. Then it runs the routine .First.

Well, at least that is how it used to be when we were using the command console. For reasons never explained (and much complained about by the users) RStudio ignores the .First file at startup, so we will need to have a work around.

Let's start with a simple. Rprofile. This is a stand-alone file usually located in your default working directory. You can find out where it is by running

```
dir(~)
```

So here is what (part of) mine looks like:

```
options(help_type="html") #ignored in RStudio options(show.signif.stars=FALSE) #for p-values options(stringsAsFactors=FALSE) #a classic source of errors library(ggplot2)
library(wolfr) library(moodler)
.MyEnv <- new.env()
.MyEnv$sc <- function() source("clipboard")
.MyEnv$dp <- function(x) { dump(x,"clipboard") }
.MyEnv$ip <- function(x) { #Install and immediately load a package install.packages(x, lib = "C:/R/library")
library(x, character.only = TRUE)
}
attach(.MyEnv)
if("First.R" %in% dir()) {#check whether there is a .First
source("First.R")
.First()
}
cat("\nSuccessfully loaded .Rprofile at", date(), "\n")
```

---

So it changes a few options, loads some libraries, makes a new environment and defines a few functions. Finally it checks the directory from where the Rproject was started to see whether it contains a file First.R. If so it loads and executes it as well.

Say this project is about writing a shiny app, then the .First.R would have the line

```
library(shiny)
```

Notice because a single .Rprofile sits in the default working directory the same one gets executed everytime I start RStudio, but different working directories might have different First.R's (or none)

## 18.1 Dropbox

Disclaimer: I discuss here Dropbox, but there are other companies that offer the same service and this is not meant as an endorsement of Dropbox (although I do like it myself!).

Dropbox is a cloud based storage site. For me it helps to solve a number of issues:

- backup: done automatically

- version control: Dropbox keeps all old versions, so if (when!!!) you save a file and then see that this was a mistake you can always go back and find a previous (good) version.
- keep things consistent: I have a number of computers, I don't want to have to work on keeping them synchronized. When I change a file in a Drpbox folder on one computer and then go to another, as soon as the new file is downloaded I got it there as well.
- availability: I can get to my files anyplace, anytime.

so I have a folder named R in the Dropbox folder. Inside that I have many folders, one for each project.

There is a bit of an issue with this: every minute or so Dropbox wants to synchronize, but RStudio “sees” this and shows an annoying pop-up. To avoid that, open Dropbox by right-clicking the icon on the task bar, select the settings icon, choose preferences, click on the Sync button, selective sync, and uncheck the box of the R folder.

Now of course Dropbox won't save anything automatically, so don't forget to do it regularly yourself!

## 19 Graphics with ggplot2

A large part of this chapter is taken from various works of Hadley Wickham. Among others The layered grammar of graphics and R for Data Science.

### 19.1 Why ggplot2?

Advantages of ggplot2

- consistent underlying grammar of graphics (Wilkinson, 2005)
- plot specification at a high level of abstraction
- very flexible
- theme system for polishing plot appearance
- mature and complete graphics system
- many users, active mailing list

### 19.2 Grammar of Graphics

In 2005 Wilkinson, Anand, and Grossman published the book “The Grammar of Graphics”. In it they layed out a systematic way to describe any graph in terms of basic building blocks. ggplot2 is an implementation of their ideas.

The use of the word *grammar* seems a bit strange here. The general dictionary meaning of the word grammar is:

*the fundamental principles or rules of an art or science*

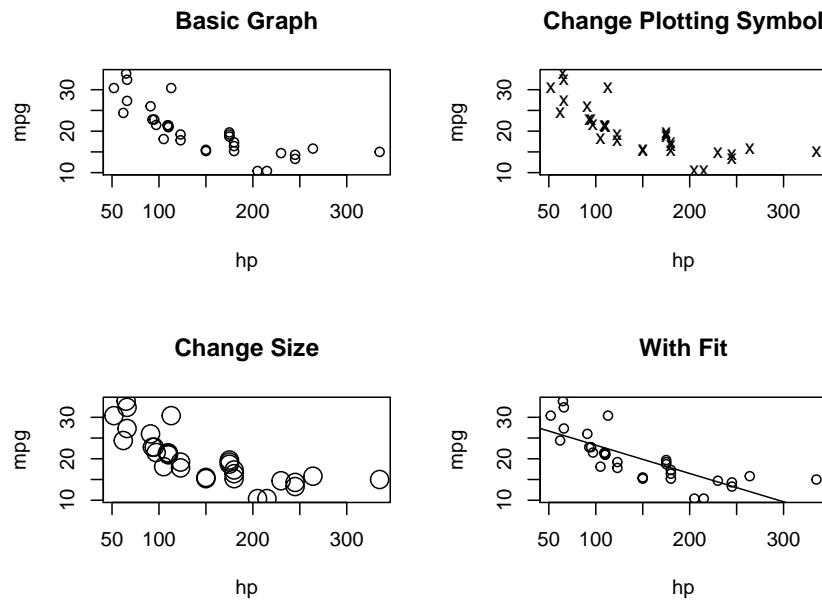
so it is not only about language.

As our running example we will use the *mtcars* data set. It is part of base R and has information on 32 cars:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Say we want to study the relationship of hp and mpg. So we have two quantitative variables, and therefore the obvious thing to do is a scatterplot. But there are a number of different ways we can this:

```
attach(mtcars)
par(mfrow=c(2, 2))
plot(hp, mpg, main="Basic Graph")
plot(hp, mpg, pch="x", main="Change Plotting Symbol")
plot(hp, mpg, cex=2, main="Change Size")
plot(hp, mpg, main="With Fit");abline(lm(mpg~hp))
```



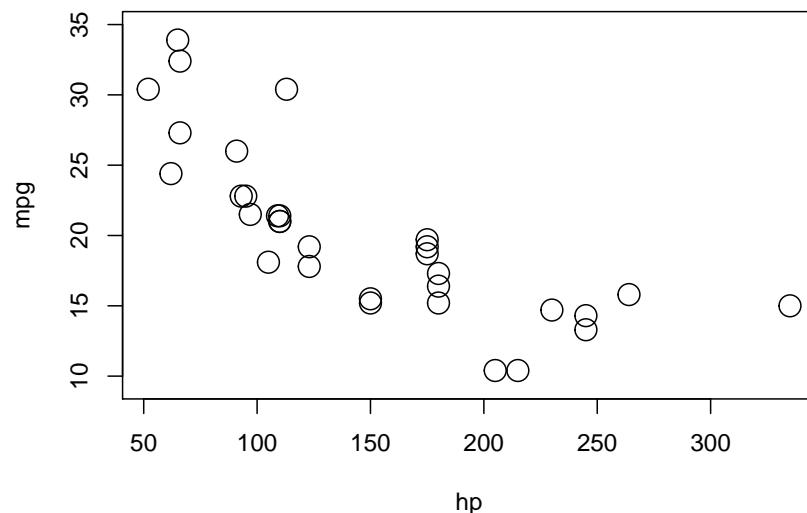
The basic idea of the grammar of graphs is to separate out the parts of the graphs: there is the basic layout, there is the data that goes into it, there is the way in which the data is displayed. Finally there are annotations, here the titles, and possibly more things added,

such as a fitted line. In ggplot2 you can always change one of these without worrying how that change effects any of the others.

Another way of looking at it this: in ggplot2 you build a graph like a lasagne: layer by layer.

Take the graph on the lower left. Here I made the plotting symbol bigger (with cex=2). But now the graph doesn't look nice any more, the first and the last circle don't fit into the graph. The only way to fix this is to start all over again, by making the margins bigger:

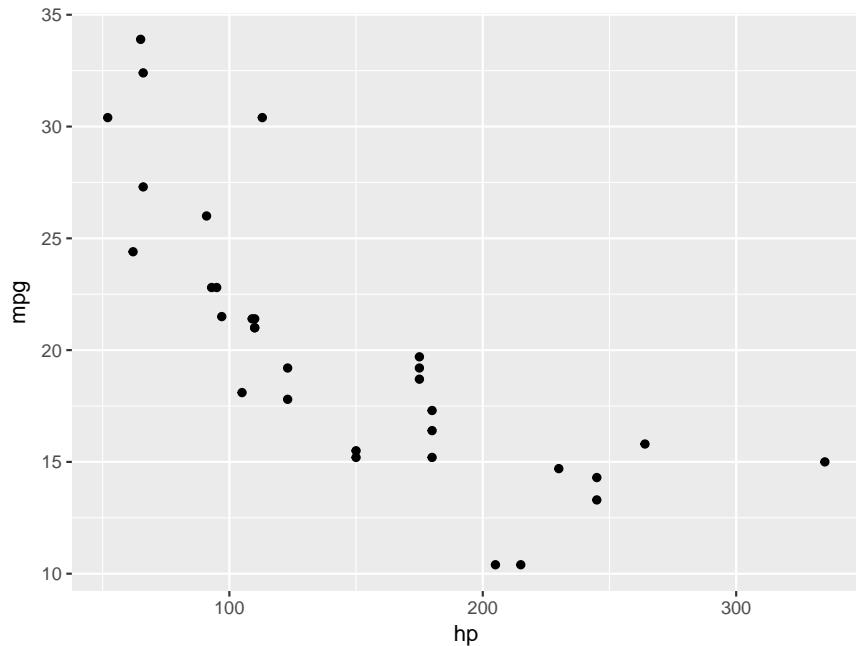
```
plot(hp, mpg, cex=2, ylim=range(mpg)+c(-1, 1))
```



and that is a bit of work because I have to figure out how to change the margins. In ggplot2 that sort of thing is taken care of automatically!

Let's start by recreating the first graph above.

```
ggplot(mtcars, aes(hp, mpg)) +
 geom_point()
```

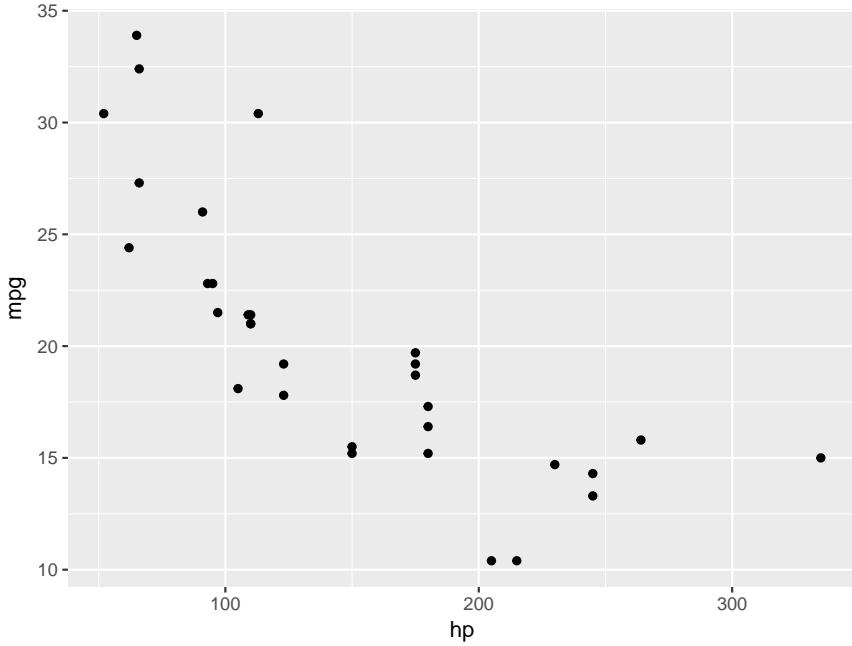


this has the following logic:

- *ggplot* sets up the graph
- it's first argument is the data set (which has to be a dataframe)
- *aes* is the *aesthetic mapping*. It connects the data to the graph
- *geom* is the geometric object (circle, square, line) to be used in the graph. Here it is points.

Note *ggplot2* also has the *qplot* command. This stands for *quick plot*

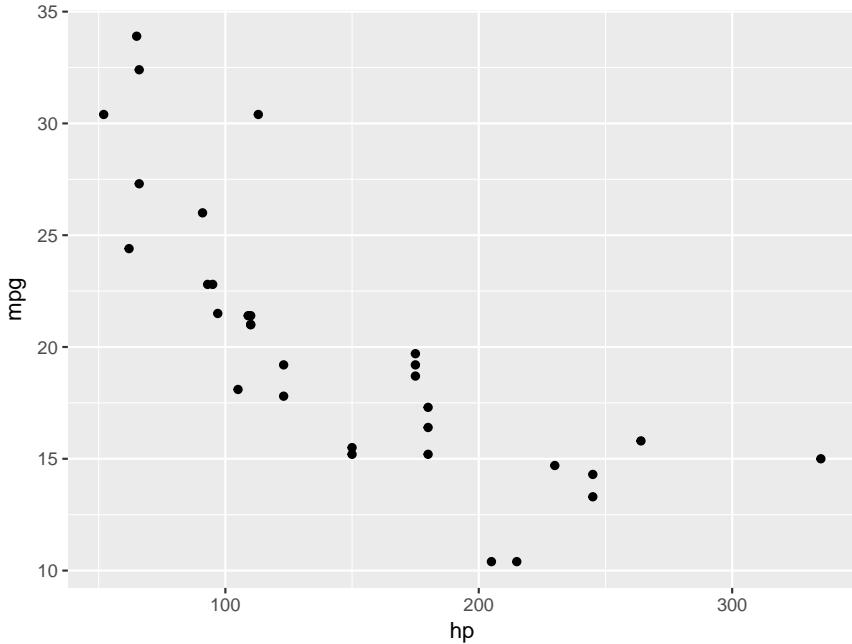
```
qplot(hp, mpg, mtcars)
```



This seems much easier at first (and it is) but the qplot command is also very limited. Very quickly you want to do things that aren't possible with qplot, and so I won't discuss it further here.

**Note** consider the following variation:

```
ggplot(mtcars) +
 geom_point(aes(hp, mpg))
```



again it seems to do the same thing, but there is a big difference:

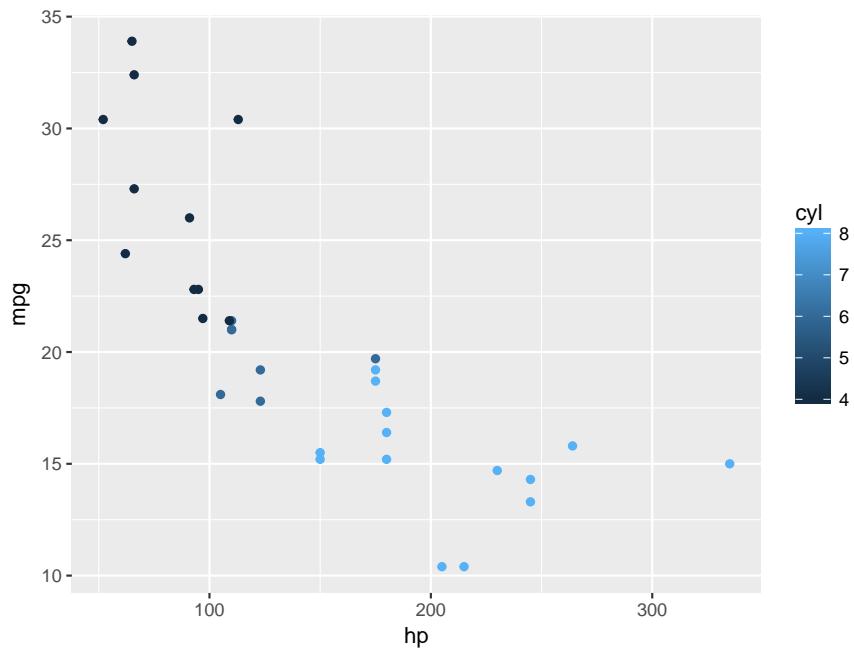
- if `aes(x, y)` is part of `ggplot`, it applies to all the geom's that come later (unless a

different one is specified)

- an aes(x, y) as part of a geom applies only to it.

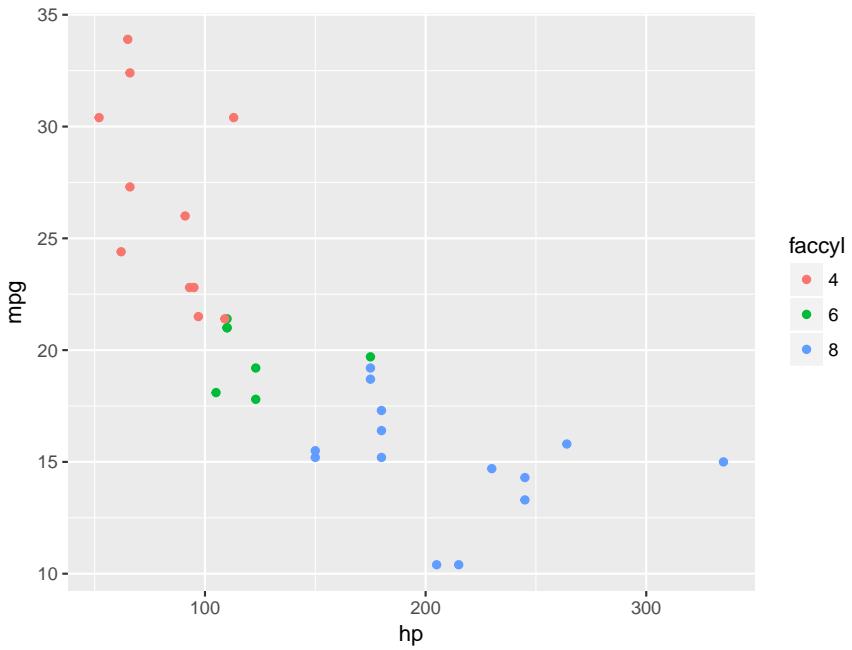
Let's say we want to identify the cars by the cylinders:

```
ggplot(mtcars, aes(hp, mpg, color=cyl)) +
 geom_point()
```



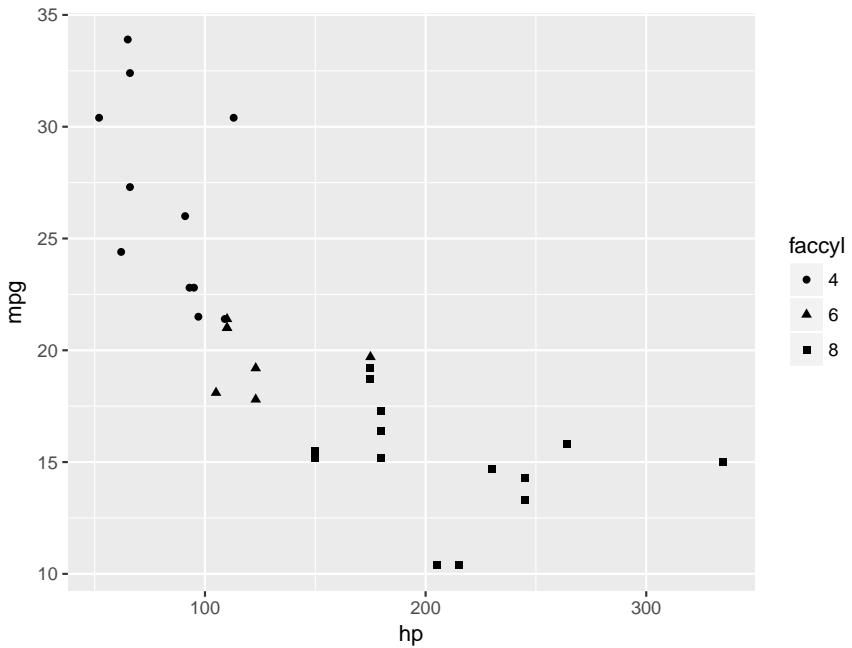
Notice that the legend is a continuous color scale. This is because the variable cyl has values 4, 6, and 8, and so is identified by R as a numeric variable. In reality it is categorical (ever seen a car with 1.7 cylinders?), and so we should change that:

```
mtcars$faccyl <- factor(cyl,
 levels = c(4, 6, 8), ordered = TRUE)
ggplot(mtcars, aes(hp, mpg, color=faccyl)) +
 geom_point()
```



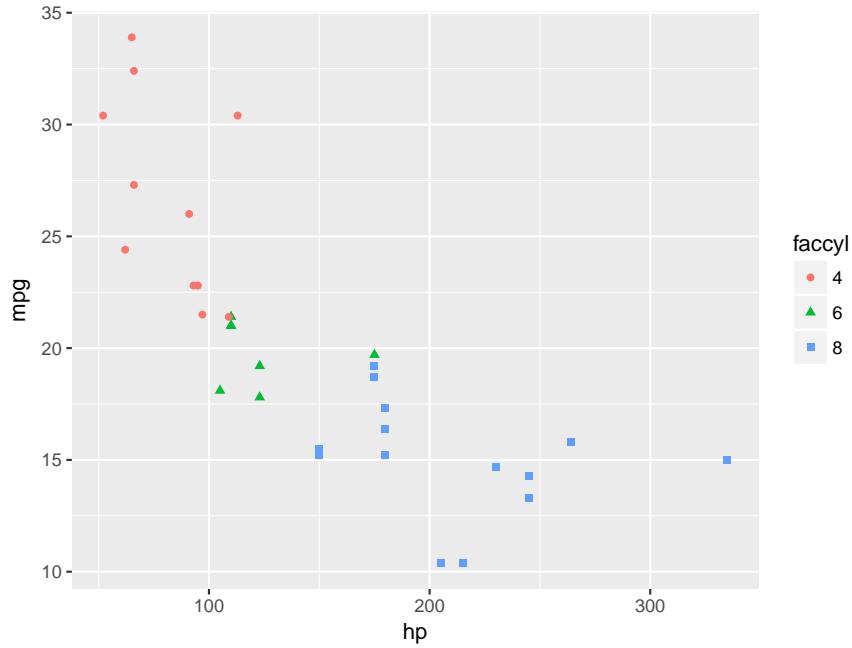
we can also change the shape of the plotting symbols:

```
ggplot(mtcars, aes(hp, mpg, shape=faccyl)) +
 geom_point()
```



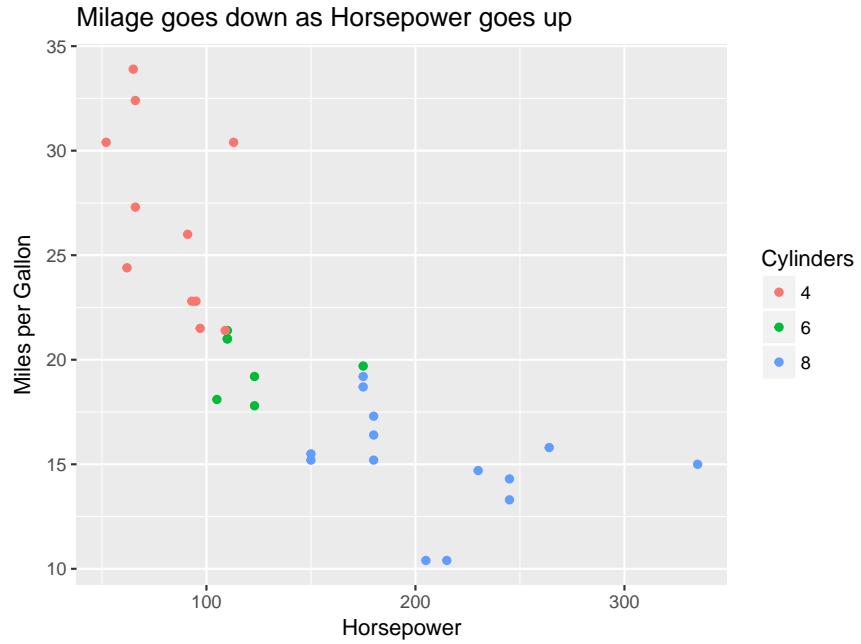
or both:

```
ggplot(mtcars, aes(hp, mpg, shape=faccyl, color=faccyl)) +
 geom_point()
```



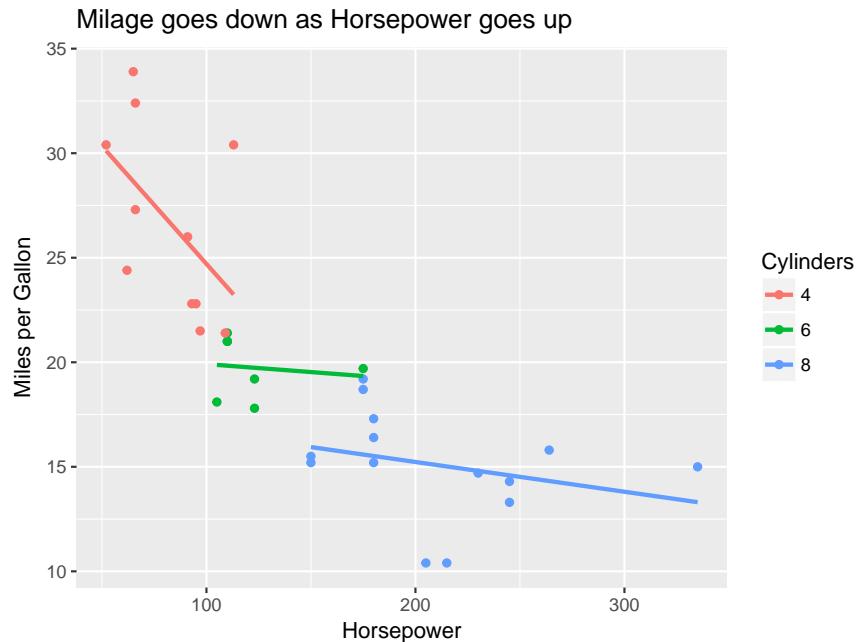
let's pretty up the graph a bit with some labels and a title. We will be playing around with this graph for a while, so I will save some intermediate versions:

```
plt1 <- ggplot(mtcars, aes(hp, mpg, color=faccyl)) +
 geom_point()
plt2 <- plt1 +
 labs(x = "Horsepower",
 y = "Miles per Gallon",
 color = "Cylinders") +
 labs(title = "Milage goes down as Horsepower goes up")
plt2
```



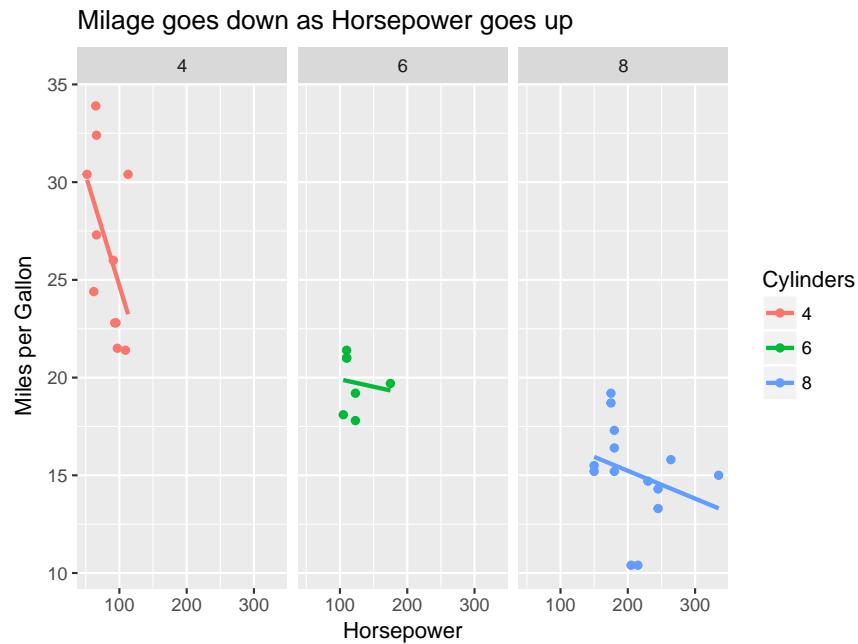
Say we want to add the least squares regression lines for cars with the same number of cylinders:

```
plt3 <- plt2 +
 geom_smooth(method = "lm", se = FALSE)
plt3
```



There is another way to include a categorical variable in a scatterplot. The idea is to do several graphs, one for each value of the categorical variable. These are called *facets*:

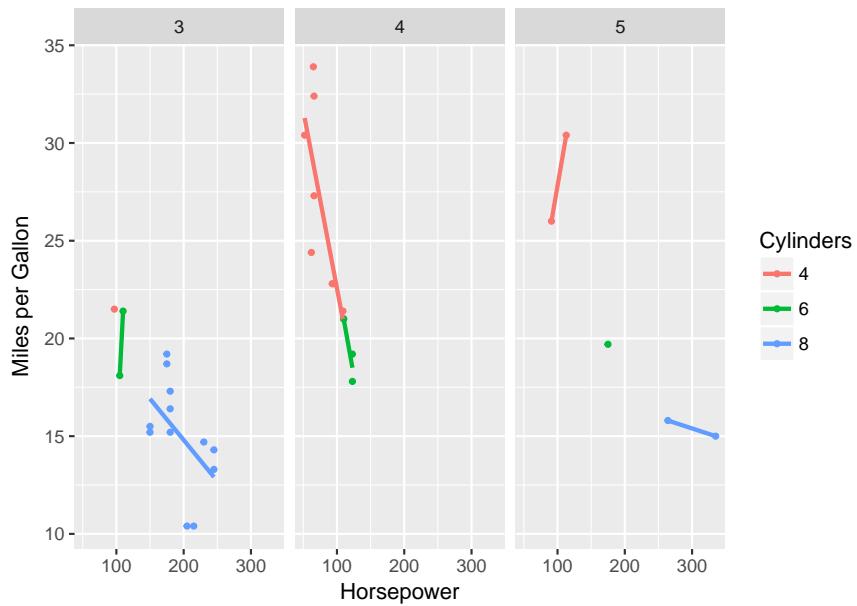
```
plt3 +
 facet_wrap(~cyl)
```



The use of facets also allows us to include two categorical variables:

```
mtcars$facgear <-
 factor(gear, levels = 3:5, ordered = TRUE)
plt4 <- ggplot(aes(hp, mpg, color=faccyl),
 data = mtcars) +
 geom_point(size = 1)
plt4 <- plt4 +
 facet_wrap(~facgear)
plt4 <- plt4 +
 labs(x = "Horsepower",
 y = "Miles per Gallon",
 color = "Cylinders") +
 labs(title = "Milage goes down as Horsepower goes up")
plt4 <- plt4 +
 geom_smooth(method = "lm", se = FALSE)
plt4
```

Milage goes down as Horsepower goes up

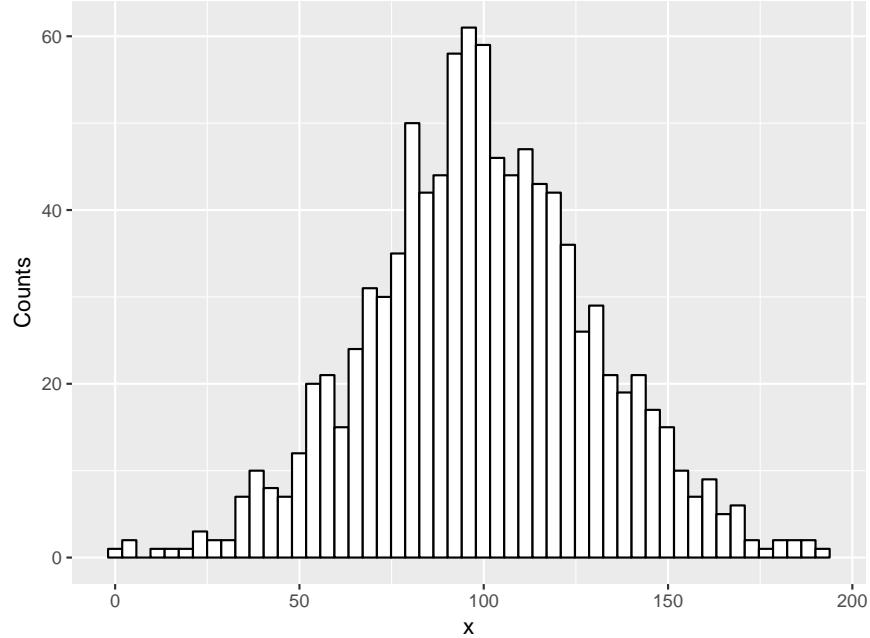


This is almost a bit to much, with just 32 data points ther is not really enough for such a split.

## 19.3 Some standard graphs

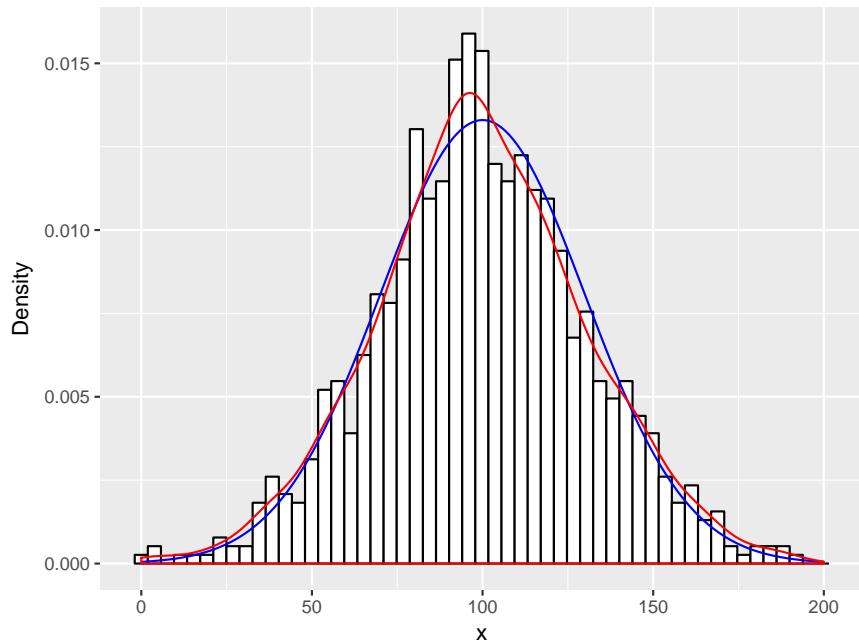
### 19.3.1 Histograms

```
x <- rnorm(1000, 100, 30)
df3 <- data.frame(x = x)
bw <- diff(range(x))/50
ggplot(df3, aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x = "x", y = "Counts")
```



Often we do histograms scaled to integrate to one. Then we can add the theoretical density and/or a nonparametric density estimate:

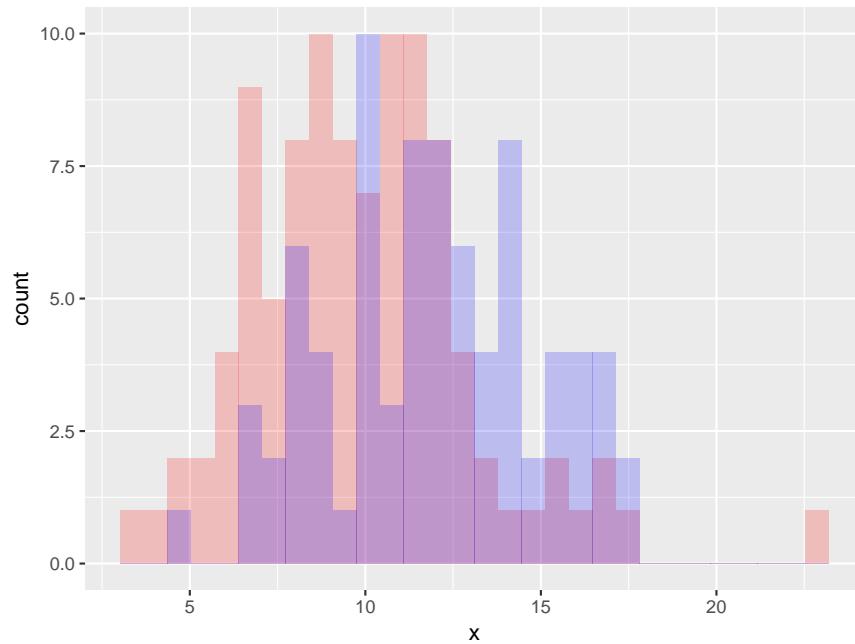
```
x <- seq(0, 200, length=250)
df4 <- data.frame(x=x, y=dnorm(x, 100, 30))
ggplot(df3, aes(x)) +
 geom_histogram(aes(y = ..density..),
 color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x = "x", y = "Density") +
 geom_line(data = df4, aes(x, y), colour = "blue") +
 geom_density(color = "red")
```



**Notice** the red line on the bottom. This should not be there but seems almost impossible to get rid of!

Here is another interesting case: say we have two data sets and we wish to draw the two histograms, one overlayed on the other:

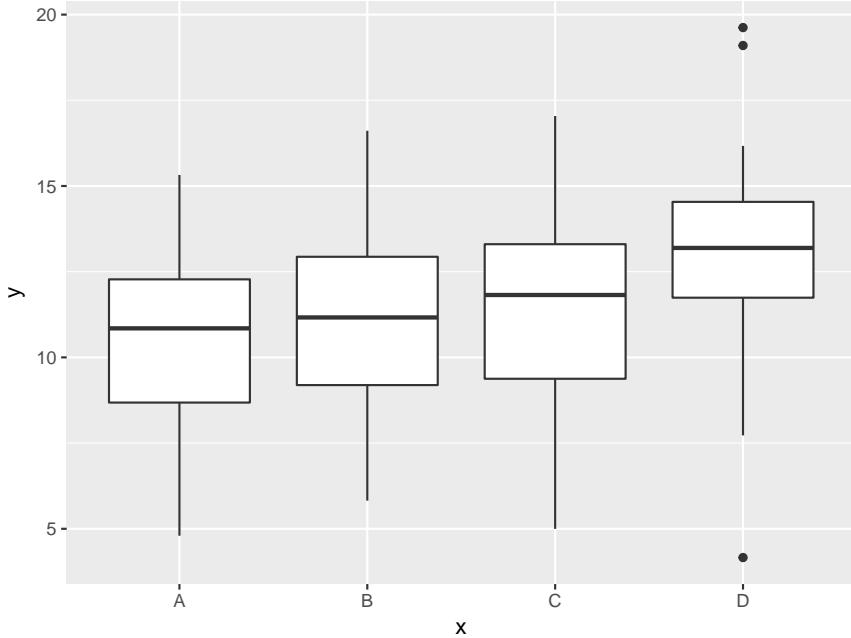
```
df5 <- data.frame(
 x = c(rnorm(100, 10, 3), rnorm(80, 12, 3)),
 y = c(rep(1, 100), rep(2, 80)))
ggplot(df5, aes(x=x)) +
 geom_histogram(data = subset(df5, y == 1),
 fill = "red", alpha = 0.2) +
 geom_histogram(data = subset(df5, y == 2),
 fill = "blue", alpha = 0.2)
```



Notice the use of alpha. In general this “lightens” the color so we can see “behind”.

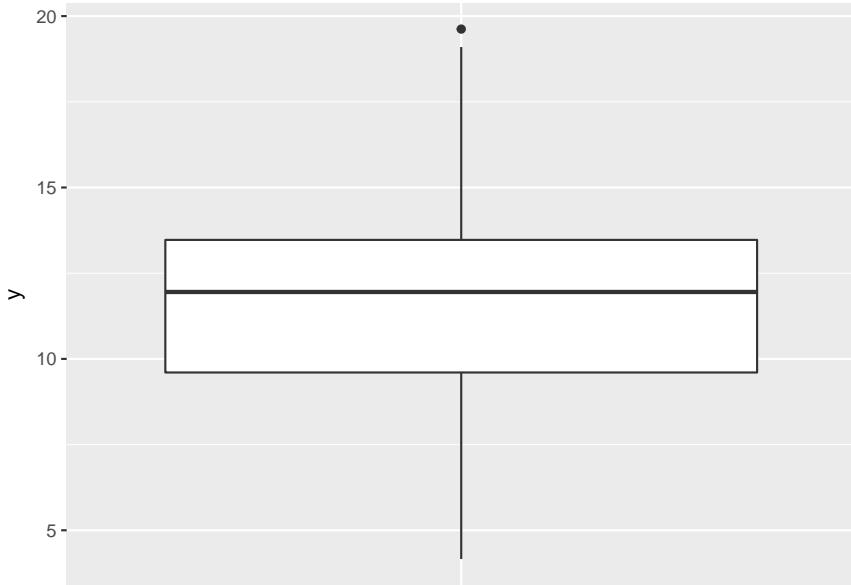
### 19.3.2 Boxplots

```
y <- rnorm(120, 10, 3)
x <- rep(LETTERS[1:4], each=30)
y[x=="B"] <- y[x=="B"] + rnorm(30, 1)
y[x=="C"] <- y[x=="C"] + rnorm(30, 2)
y[x=="D"] <- y[x=="D"] + rnorm(30, 3)
df6 <- data.frame(x=x, y=y)
ggplot(df6, aes(x, y)) +
 geom_boxplot()
```



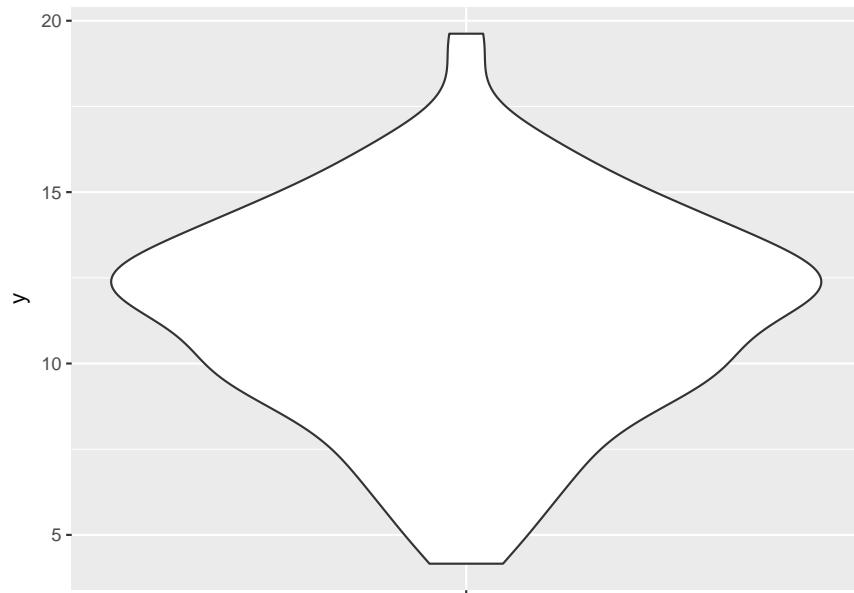
strangely enough doing a boxplot without groups takes a little work. We have to “invent” a categorical variable:

```
df6$z <- rep(" ", length(y))
ggplot(df6, aes(z, y)) +
 geom_boxplot() +
 xlab("")
```



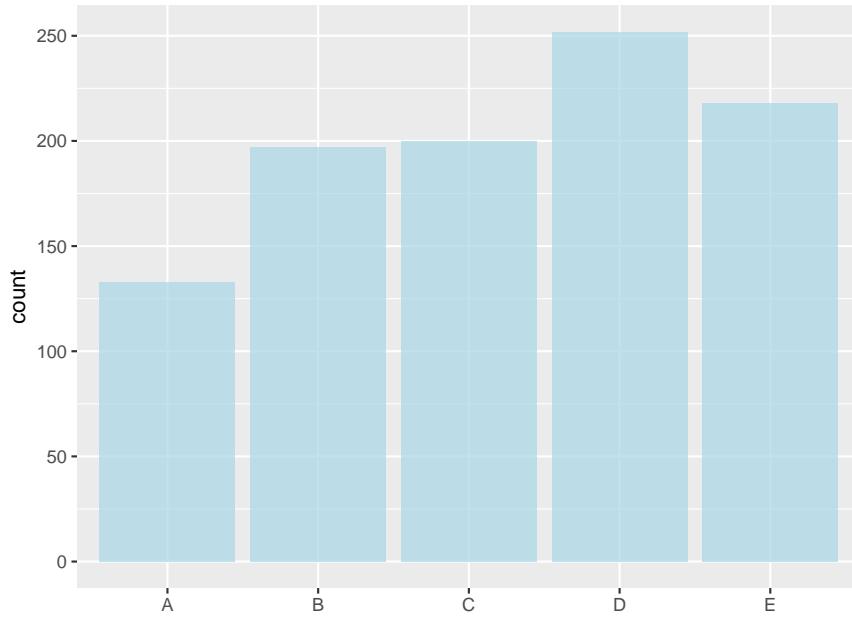
There is a modern version of this graph called a violin plot:

```
ggplot(df6, aes(z, y)) +
 geom_violin() +
 xlab("")
```



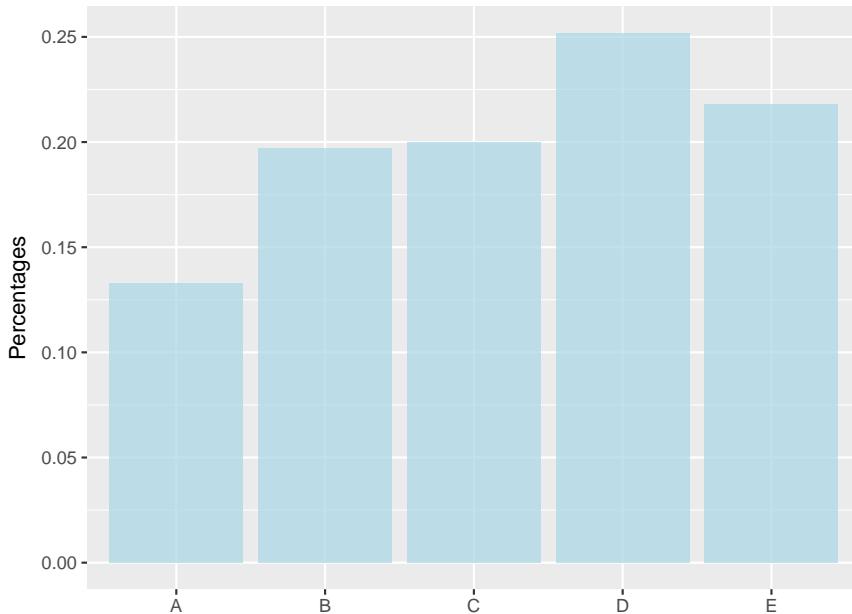
### 19.3.3 Barcharts

```
x <- sample(LETTERS[1:5],
 size = 1000,
 replace = TRUE,
 prob = 6:10)
df7 <- data.frame(x=x)
ggplot(df7, aes(x)) +
 geom_bar(alpha=0.75, fill="lightblue") +
 xlab("")
```



Say we want to draw the graph based on percentages. Of course we could just calculate them and then do the graph. Here is another way:

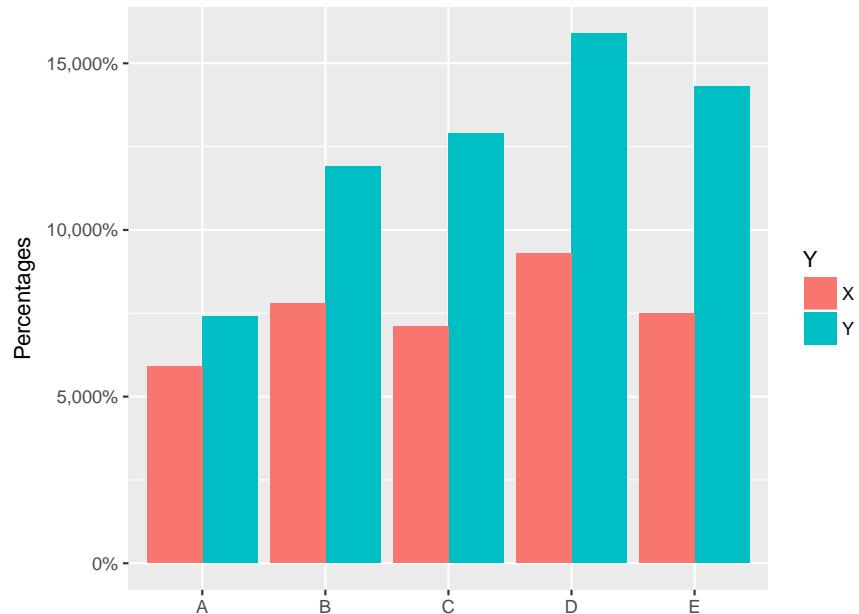
```
ggplot(df7, aes(x=x)) +
 geom_bar(aes(y=..count../sum(..count..)),
 alpha = 0.75,
 fill = "lightblue") +
 labs(x="", y="Percentages")
```



Notice how this works: in `geom_bar` we use a new `aes`, but the values in it are calculated from the old data frame.

Finally an example of a contingency table:

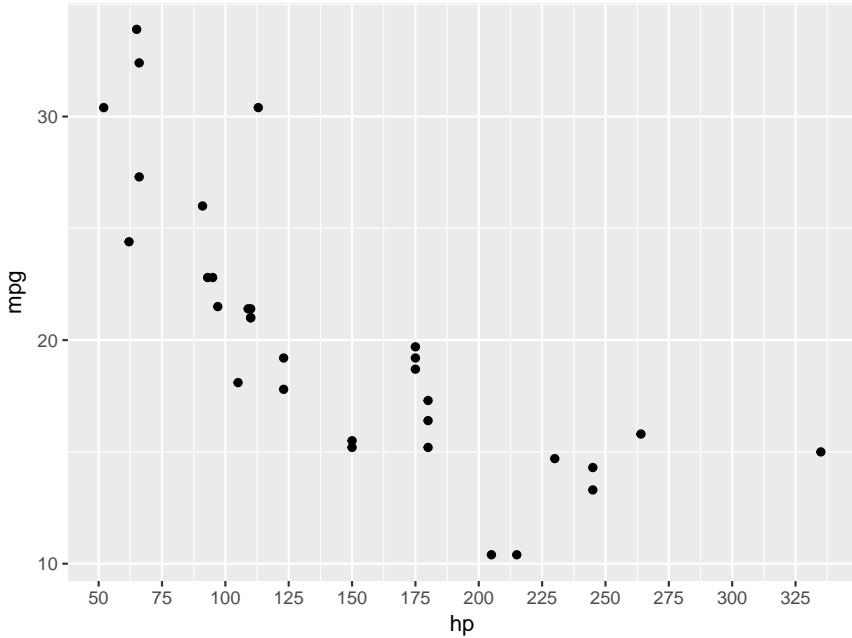
```
df7$y <- sample(c("X", "Y"),
 size = 1000,
 replace = TRUE,
 prob = 2:3)
ggplot(df7, aes(x=x, fill = y)) +
 geom_bar(position = "dodge") +
 scale_y_continuous(labels=scales::percent) +
 labs(x="", y="Percentages", fill="Y")
```



## 19.4 Axis Ticks and Legend Keys

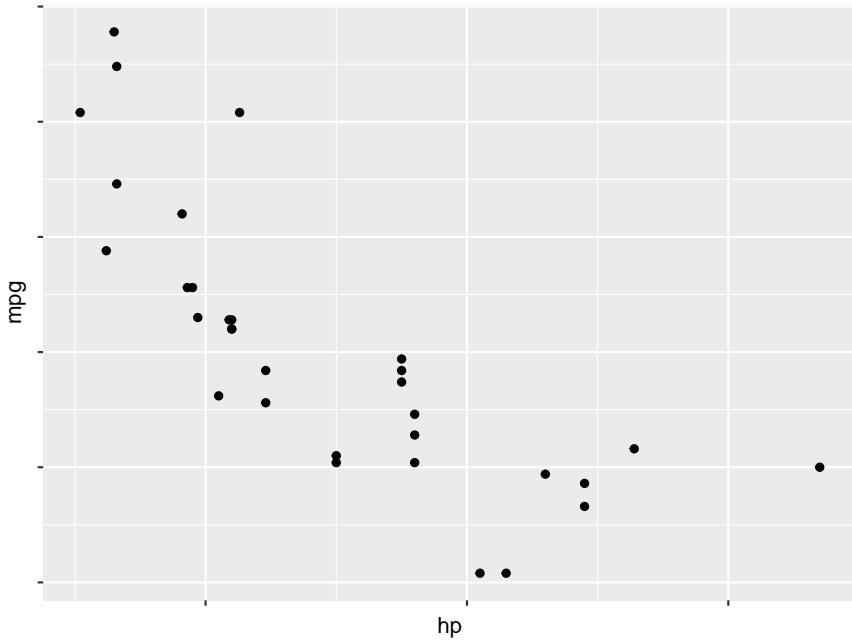
Let's return to the basic plot of mpg by hp. Let's say we want to change the axis tick marks:

```
ggplot(mtcars, aes(hp, mpg)) +
 geom_point() +
 scale_x_continuous(breaks = seq(50, 350, by=25)) +
 scale_y_continuous(breaks = seq(0, 50, by=10))
```



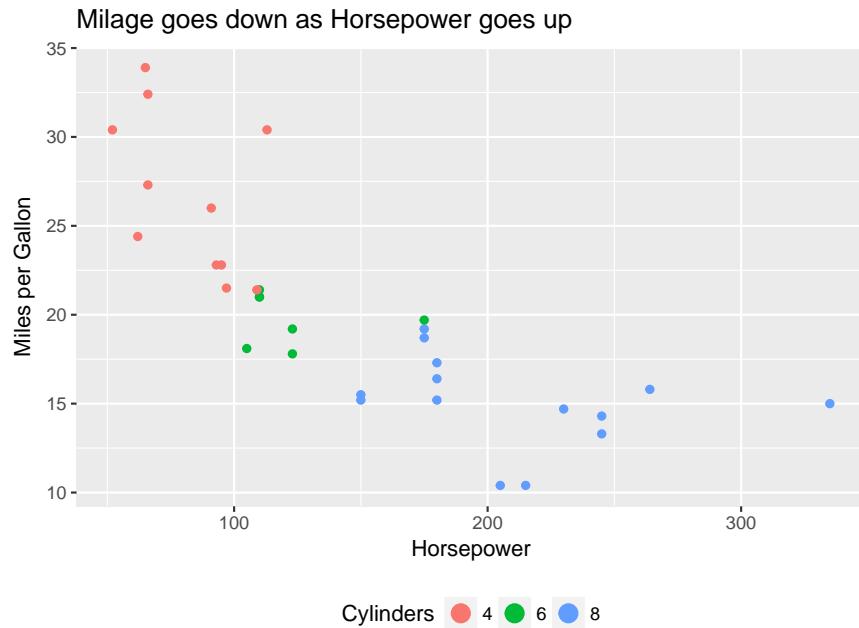
sometimes we want to do graphs without any tick labels. This is useful for example for maps and also for confidential data, so the viewer sees the relationship but can't tell the sizes:

```
ggplot(mtcars, aes(hp, mpg)) +
 geom_point() +
 scale_x_continuous(labels = NULL) +
 scale_y_continuous(labels = NULL)
```



By default ggplot2 draws the legends on the right. We can however change that. We can also change appearance of the legend. Recall that the basic graph is in *plt2*. Then

```
plt2 +
 theme(legend.position = "bottom") +
 guides(color=guide_legend(nrow = 1,
 override.aes = list(size=4)))
```



## 19.5 Saving the graph

It is very easy to save a ggplot2 graph. Simply run

```
ggsave("myplot.pdf")
```

it will save the last graph to disc.

One issue is figure sizing. You need to do this so that a graph looks “good”. Unfortunately this depends on where it ends up. A graph that looks good on a webpage might look ugly in a pdf. So it is hard to give any general guidelines.

If you use R markdown, a good place to start is with the chunk arguments `fig.width=6` and `out.width="70%"`. In fact on top of every R markdown file I have a chunk with

```
library(knitr)
opts_chunk$set(fig.width=6,
 fig.align = "center",
 out.width = "70%",
 warning=FALSE,
 message=FALSE)
```

so that automatically every graph is sized that way. I also change the default behavior of the chunks to something I like better!

## 20 Rcpp

For this section you will need the Rcpp and the microbenchmark packages.

```
library(Rcpp)
library(microbenchmark)
```

Sometimes when you have some code that takes a while to run it is worthwhile to spend some time speeding it up. One way to do this is to rewrite part of the code in C++.

Say we have the following problem: we have a data set with points  $(x, y)$  and for each point we want to find the Euclidean distance to the origin  $d = \sqrt{x^2 + y^2}$ . Here is a simple routine to do this:

```
dist1 <- function(x, y) {
 n <- length(x)
 d <- rep(0, n)
 for(i in 1:n) d[i] <- sqrt(x[i]^2+y[i]^2)
 d
}
```

Let's see how long this takes:

```
x <- rnorm(1e6)
y <- rnorm(1e6)
summary(microbenchmark(dist1(x, y)))["mean"]

[1] NA
```

Now of course we can immediately speed things up by vectorizing the routine:

```
dist2 <- function(x, y) {
 sqrt(x^2+y^2)
}
summary(microbenchmark(dist2(x, y)))["mean"]

[1] NA
```

Can we do even better?

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector dist3(NumericVector x, NumericVector y) {
 int n=x.length();
 NumericVector dist(n);
 for (int i=0; i<n; ++i) dist[i]=sqrt(x[i]*x[i]+y[i]*y[i]);
 return dist;
}

summary(microbenchmark(dist3(x, y)))["mean"]
```

```
[1] NA
```

Notice that the above chunk starts like this:

```
“{r engine='Rcpp'}
```

Actually, we can even do better than that:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector dist4(NumericVector x, NumericVector y) {
 NumericVector dist;
 dist=sqrt(x*x+y*y);
 return dist;
}

summary(microbenchmark(dist4(x, y)))["mean"]
```

```
[1] NA
```

Those of you who already know a bit of C++ are going to be quite amazed, because this is not even C++, it is sort of “vectorized” C++!

---

Now the difference between 3.6 and 12.1 milliseconds (the fastest non-C++ time) might not seem like much, but imagine if we had to run this routine one million times, then the R routine would take  $12 * 10^{-3} \times 10^6 / 3600 = 3.3$  hours, but the Cpp routine takes only 1.5 hours. Writing it takes literally 1 minute! (assuming you know how)

To start let's discuss a few differences between R and C++ syntax:

- in C++ every variable has to be explicitly defined.
- in C++ (almost) every line ends in ;
- vectors start with index 0, not 1
- the for loop is for(int i=0;i<n;++i)

Moreover, to be linked to R the C++ routine has to start with

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
```

The easiest way to get started is to just use RStudio - File - New File - C++ File.

There are some special variable types you need to use when moving data from R to C++ and back. The most important is the one used in our programs above: NumericVector. It is just what it says it is.

## 20.1 Debugging

Generally you should only turn fairly short code into C++, so debugging is not too big a problem. However, on occasion you might want to add a print statement to your code, so you can find out where it fails. here is how:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector dist5(NumericVector x, NumericVector y) {
 int n=x.length();
 NumericVector dist(n);
 for(int i=0; i<n; ++i) {
 dist[i]=sqrt(x[i]*x[i]+y[i]*y[i]);
 Rcout<<i<<" "<<dist(i)<<"\n";
 }
 return dist;
}

dist5(x[1:5], y[1:5])

0 0.207442
1 1.98019
2 1.52449
3 0.902759
4 0.99759

[1] 0.207441906847783 1.980187474733545 1.524488303323749 0.902758522189152
[5] 0.997589533522500
```

## 20.2 Sugar

Not only is Rcpp vectorized, many standard R functions have been ported to Rcpp. For example, say we want to write a routine that simulates Brownian motion in  $R^2$ . That is, a stochastic process that moves as follows: if at time  $t$  it is at  $(x_0, y_0)$ , then at time  $t + \delta$  it is at  $(x_0 + \delta X, y_0 + \delta Y)$  where  $X, Y \sim N(0, 1)$ .

Note: generating stochastic process can be quite slow in R because they are difficult to vectorize, with one step of a loop depending on the previous one.

The output of our function is going to be a  $nx2$  matrix, so we will use the data type *NumericMatrix*.

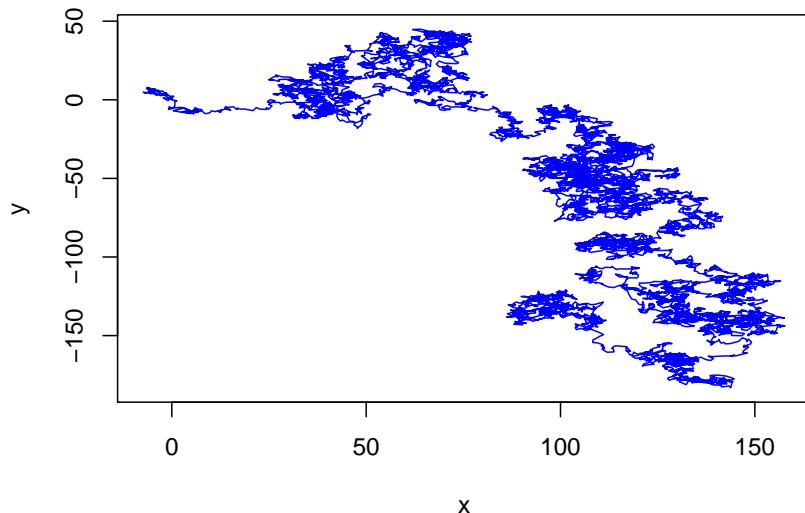
```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericMatrix bw(int n, double delta) {
 NumericMatrix xy(n, 2);
 xy(0, 0) = 0;
```

```

xy(0, 1) = 0;
for(int i=2; i<n; ++i) {
 xy(i, 0) = xy(i-1, 0) + delta*rnorm(1)[0];
 xy(i, 1) = xy(i-1, 1) + delta*rnorm(1)[0];
}
return xy;
}

plot(bw(10000, 1),
 type = "l",
 xlab = "x",
 ylab = "y",
 col = "blue")

```



Notice the term  $rnorm(1)[0]$ , a little different from the standard R usage.

Anyone (like me!) who ever had to write a routine in C++ and needed a simple routine like `rnorm` which does not exist in C++ will find this very sweet! And that is why it is called sugar!

### 20.3 STD (standard template library)

The standard template library (STL) provides a set of extremely useful data structures and algorithms.

Let's start with a simple C++ implementation of the R `sum` function:

```

#include <Rcpp.h>
using namespace Rcpp;

```

```

// [[Rcpp::export]]
double sum_cpp(NumericVector x) {
 double total = 0.0;
 for(int i=0; i<x.length(); ++i) total +=x(i);
 return total;
}

sum(x)

[1] 496.704636477974

```

```
sum_cpp(x)
```

```
[1] 496.704636478
```

Notice the term  $total+=x(i)$ ; which is standard C++ shorthand for  $total = total + x(i)$ ;

Now the same can be written as:

```

#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double sum_cpp1(NumericVector x) {
 double total=0.0;
 NumericVector::iterator it;
 for(it=x.begin(); it!=x.end(); ++it) total += *it;
 return total;
}

sum_cpp1(x)

[1] 496.704636478

```

## [1] 496.704636478

*Example* let's write a routine that calculates the golden ratio via the Fibonacci numbers. First, these are defined by

$$\begin{aligned} n_0 &= 1 \\ n_1 &= 1 \\ n_k &= n_{k-1} + n_{k-2} \end{aligned}$$

and the golden ratio is the limit

$$\lim_{k \rightarrow \infty} \frac{n_k}{n_{k-1}}$$

because of its definition the Fibonacci numbers are most easily calculated using recursion:

```

fibR <- function(n) {
 if(n==0) return(0)
 if(n==1) return(1)
 return (fibR(n-1)+fibR(n-2))
}

```

```
}
```

```
fibR(10)
```

```
[1] 55
```

Let's write this with Rcpp:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int fib_cpp(const int n) {
 if(n==0) return(0);
 if(n==1) return(1);
 return fib_cpp(n-1)+fib_cpp(n-2);
}
```

```
fib_cpp(10)
```

```
[1] 55
```

and now for the golden ratio:

```
golden_ratio <- function(n, fun) {
 fun(n)/fun(n-1)
}
summary(microbenchmark(golden_ratio(10,
 fun=fibR)))["mean"]
```

```
[1] NA
```

```
summary(microbenchmark(golden_ratio(10,
 fun=fib_cpp)))["mean"]
```

```
[1] NA
```

not only is the cpp version much faster, in this example R is actually quite useless: while it does recursion, doing it to often quickly becomes a problem (because of memory issues).

Now

```
golden_ratio(30, fun=fib_cpp)
```

```
[1] 1.6180339887482
```

The actual value of the golden ratio is of course  $\frac{1+\sqrt{5}}{2} = 1.618\dots$

## 20.4 Using C++ routines.

So far we used C++ to speed up calculations, and we could use Sugar to call standard R functions in the C++ routine. There is another use, though. C++ has been the main computing language in many fields for a long time, and so there exist a large number of

excellent routines already written. Say you found one of these and want to use it in your R program. Here is how:

What we need to do is to write a Rcpp wrapper routine, that eventually calls the C++ subroutine. Here is an example:

```
#include <Rcpp.h>
using namespace Rcpp;

//function declaration
double sum_of_squares(double x[], int n);

// [[Rcpp::export]]
double sub_routine(NumericVector x) {
 int n=x.length();
 double y[n];
 double ssq;
 for(int i=0;i<n;++i) y[i]=x[i];
 ssq=sum_of_squares(y, n);
 return ssq;
}

double sum_of_squares (double x[], int n)
{
 double r;
 for(int i=0;i<n;++i) r+=x[i]*x[i];
 return r;
}

sub_routine(1:10)
```

```
[1] 385
```

The *sum\_of\_squares* routine is pure C++, like any you might find on the web!

## 21 Parallel and GPU Computing

```
library(microbenchmark)
```

Many modern computers have several processor cores. This makes it easy to do parallel computing with R. Also, many simulation problems are *embarrassingly parallel*, that is they can be run in parallel very easily.

There are a number of packages that help here. I will discuss

```
library(parallel)
```

If you don't know how many processors (called cores) your computer has you can check:

```
detectCores()
```

```
[1] 6
```

It is usually a good idea to leave one core for other tasks, so let's use

```
num_cores <- detectCores() - 1
```

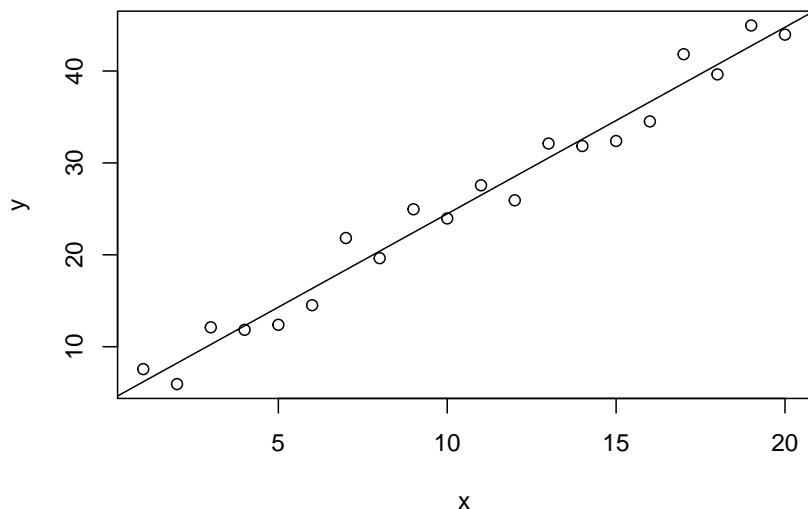
of them.

Let's consider a simple simulation problem. We wish to study the standard estimators of the least squares regression line. That is we have a data set of the form  $(x, y)$  and we want to fit an equation of the form  $y = \beta_0 + \beta_1 x$ . The parameters  $\beta_0$  and  $\beta_1$  are estimated by minimizing the least squares criterion

$$L(a, b) = \sum_{i=1}^n (y_i - a - bx_i)^2$$

We can use the R function `lm` to this:

```
x <- 1:20
y <- 5 + 2*x + rnorm(10, 0, 3)
fit <- lm(y~x)
plot(x, y)
abline(fit)
```



```
coef(fit)
```

```
(Intercept) x
4.14863977212587 2.03073653768910
```

Now a simulation will fix some numbers  $n$ ,  $\beta_0$ ,  $\beta_1$  and  $\sigma$ , generate  $B$  data sets and find the coefficients. Finally it will study the estimates.

```
sim_lm <- function(param) {
 beta0 <- param[1]
 beta1 <- param[2]
 sigma <- param[3]
 n <- param[4]
 B <- param[5]
 coefs <- matrix(0, B, 2)
 x <- 1:n
 for(i in 1:B) {
 y <- beta0 + beta1*x + rnorm(n, 0, sigma)
 coefs[i,] <- coef(lm(y~x))
 }
 coefs
}
tm <- proc.time()
z1 <- sim_lm(c(5, 2, 3, 20, 50000))
tm <- round(proc.time()-tm)[3]
tm

elapsed
28
```

so this takes almost 28 seconds. In real life we would repeat this now for different values of the parameters, so you see this can take quite some time. Instead let's parallelize the task:

```
cl <- makeCluster(num_cores)
params <- c(5, 2, 3, 20, 10000)
tm <- proc.time()
z2<-clusterCall(cl, sim_lm, params)
tm <- round(proc.time()-tm)[3]
tm

elapsed
6
```

and so this took only about 6 seconds!

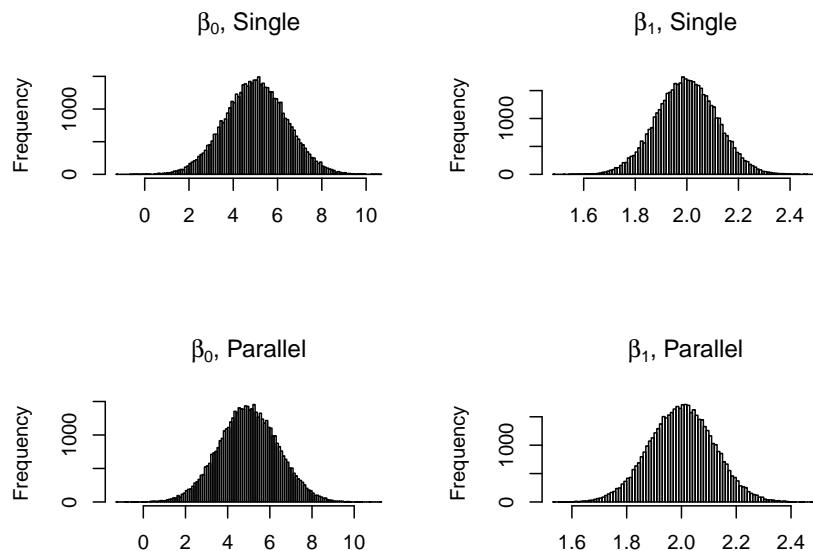
Did it really calculate the same thing? Let's see. Please note that parallel returns a list, one for each cluster:

```
par(mfrow=c(2, 2))
hist(z1[, 1], 100,
 main = expression(paste(beta[0], ", Single")),
 xlab = ""))
hist(z1[, 2], 100,
 main=expression(paste(beta[1], ", Single")),
 xlab = "")
```

```

a <- rbind(z2[[1]], z2[[2]], z2[[3]], z2[[4]], z2[[5]])
hist(a[, 1], 100,
 main = expression(paste(beta[0], ", Parallel")),
 xlab = "")
hist(a[, 2], 100,
 main = expression(paste(beta[1], ", Parallel")),
 xlab = "")

```



Certainly looks like it!

We previously discussed the `apply` family of functions. If one of these is what you want to use, they have equivalents in *parallel*.

So say we have a large matrix and want to find the maximum in each row:

```

B <- 1e5
A <- matrix(runif(10*B), B, 10)
tm <- proc.time()
a <- apply(A, 1, max)
proc.time() - tm

user system elapsed
0.0900000000000034 0.0000000000000000 0.0900000000000034

tm <- proc.time()
a <- parRapply(cl, A, max)
proc.time() - tm

user system elapsed
0.0000000000000000 0.0100000000000002 0.0399999999999920

```

In general the easiest case of parallelizing a calculation is if you have already used the *lapply* command: Let's say we have the scores of students in a number of exams. Because each exam had a different number of students we have the data organized as a list:

```
Exam 1: 72 65 71 70 70 69 75 63 63 68
cat("Exam 2: ", grades$Exam_2, "\n")

Exam 2: 77 76 67 76 68 67 64 70 72 71

length(grades)

[1] 50
```

Now we want to find the minimum , mean, standard deviation and maximum of each exam. We can do that with

```
z <- lapply(grades,
 function(x) {round(c(min(x),
 mean(x),
 sd(x),
 max(x)), 1)}
)
z[1:2]

$Exam_1
[1] 63.0 68.6 3.9 75.0
##
$Exam_2
[1] 64.0 70.8 4.4 77.0
```

and to run this in parallel:

```
z <- parLapply(cl, grades, function(x) {c(min(x),
 mean(x),
 sd(x),
 max(x))})
z[1:2]

$Exam_1
[1] 63.0000000000000 68.6000000000000 3.9214509786274 75.0000000000000
##
$Exam_2
[1] 64.0000000000000 70.8000000000000 4.44222166638871 77.0000000000000
```

When you are done with the parallel calculations

```
stopCluster(cl)
```

## 21.1 GPU Programming

20 years ago or so there was a lot of talk about *massively parallel* computing. This was the idea of using 100s or 1000s of cpu's (*computer processing units*). It never went very far because such computers were way to expensive.

However, there was one area where such chips were in fact developed, namely for graphics cards. The difference is that these cpus are extremely simple, they don't have to do much, just determine the colors of a few pixels. Eventually it occurred to people that as long as the computations were simple as well, such gpu's (*graphics processing units*), could also be used for other purposes. So if your computer has a dedicated graphics card you can do this.

Not all cards, however, will work. The most widely available ones are NVIDIA, but some others work as well.

To do gpu programming you need to get the *gpuR* library. Unlike most libraries this one is distributed as a *source*, so before you can use it it needs to be compiled. This will happen automatically but does take a bit of time.

To make sure you have all that is needed install the package and then run

```
library(gpuR)
detectGPUs()
```

```
[1] 1
```

The *gpuR* package has mostly routines for matrix algebra. So let's say we have a large matrix which we want to invert.

```
A <- matrix(rnorm(100), 10, 10)
summary(microbenchmark(solve(A)))["mean"]
```

```
[1] NA
```

To use *gpuR* we have to turn the matrix into a *gpuR* object, and then we can run *solve* again:

```
A_gpuR <- vclMatrix(A, type="float")
summary(microbenchmark(solve(A_gpuR)))["mean"]
```

```
[1] NA
```

Most linear algebra methods have been created to be executed for the *gpuMatrix* and *gpuVector* objects. These methods include basic arithmetic functions `%*%`, `+`, `-`, `*`, `/`, `t`, `,`, `crossprod`, `tcrossprod`, `colMeans`, `colSums`, `rowMean`, and `rowSums`.

Math functions include `sin`, `asin`, `sinh`, `cos`, `acos`, `cosh`, `tan`, `atan`, `tanh`, `log`, `log10`, `exp`, `abs`, `max`, `andmin`. Additional operations include some linear algebra routines such as `cov`(Pearson Covariance) and `eigen`.

A few 'distance' routines have also been added with the `dist` and `distance` (for pairwise) functions. These currently include 'Euclidean' and 'SqEuclidean' methods.

## 22 Input/Output Part 2

There are a number of packages that help with data I/O. Some are specialized for certain data types, others are more general

### 22.1 rio

One of my favorites is *rio*. An introduction can be found at <https://cran.r-project.org/web/packages/rio/vignettes/rio.html>. The list of supported file formats is quite impressive! You use the import function to import data and the export function to export. The routine uses the extension to figure out the file format. So say you have a file called mytestdata.csv in a folder called c:/tmpdata, just run

```
B <- 2*1e6
x <- round(rnorm(B), 3)
y <- round(rnorm(B), 3)
z <- sample(letters[1:5], size=B, replace=TRUE)
xyz <- data.frame(x, y, z)
head(xyz, 3)
```

```
x y z
1 0.187 -0.301 e
2 -1.023 2.037 a
3 0.370 -1.882 c
```

```
dir.create("c:/tmpdata")
library(rio)
export(xyz, "c:/tmpdata/mytestdata.csv")
detach(2)
rm(xyz)
```

```
library(rio)
head(import("c:/tmpdata/mytestdata.csv"), 3)
```

```
x y z
1 0.187 -0.301 e
2 -1.023 2.037 a
3 0.370 -1.882 c
```

#### 22.1.1 Minitab to R

rio has the ability to read Minitab files. Unfortunately they have to be in the portable format, and Minitab stopped using that some versions ago. So the easiest thing to do is save files in Minitab as .csv.

## 22.2 readr

This package is specific to rectangular data, such as data from an Excel spreadsheet. Its main advantage is its speed when compared to the corresponding base R routine read.csv:

```
tm <- proc.time()
head(read.csv("c:/tmpdata/mytestdata.csv"), 2)

x y z
1 0.187 -0.301 e
2 -1.023 2.037 a
(proc.time()-tm)[3]

elapsed
1.5999999999999999

library(readr)
tm <- proc.time()
head(import("c:/tmpdata/mytestdata.csv"), 2)

x y z
1 0.187 -0.301 e
2 -1.023 2.037 a
(proc.time()-tm)[3]

elapsed
0.0900000000000034
```

Note that the data is in the form of a *tibble*. This is a special kind of data frame which we will talk about later.

## 22.3 data.table

Similar to read.table but faster and more convenient. The command is called *fread*:

```
library(data.table)
tm <- proc.time()
head(fread("c:/tmpdata/mytestdata.csv"), 2)

x y z
1: 0.187 -0.301 e
2: -1.023 2.037 a
(proc.time()-tm)[3]

elapsed
0.0600000000000023
```

This command is what I would recommend if you deal with Big Data. These days we classify data as follows:

- big data (a few hundred thousand rows, about 20 MB)
- Big Data (5 million rows, about 1GB)
- Bigger Data (over 100 million rows, over 10GB)

In the case of Bigger Data you can no longer have all of it in memory, but it becomes necessary to use a hard drive as memory. A useful package for that is *bigmemory*.

## 22.4 pdftools

The pdf format is the most common document format on the internet, so quite often we are faced with the following problem: we want to extract some data from a pdf. Here we can use

```
library(pdftools)
```

As an example consider the report on the World Mortality Rate 2017. It has a large table with mortality information. We begin by downloading it:

```
download.file("http://academic.uprm.edu/wrolke/esma6835/World-Mortality-2017-Data-Booklet.pdf")
```

and then turn it into a text file:

```
txt <- pdf_text("world-mortality.pdf")
nchar(txt)

[1] 73 1582 5395 2295 2332 3494 5456 4959 4766 3546 11 5683 5588 6350
[15] 6156 5955 5928 6258 6232 2625 5764 3288 0 0
```

Notice that the document has 24 pages, and each is read in as one character string.

The data table starts on page 10, which is txt[12], and ends on page 19, which is txt[20]. Let's begin by making a long character string with each piece of text/numbers separate. We want to split the string at white spaces, which we can do with the reg expression

```
strsplit(txt, "\\s+")[1]
```

However, notice that some countries have a superscript (which is also separated from the name by a white space) and that the large numbers have one white space in between also. So what we need to do is split if there are two or more white spaces:

```
txt <- paste(txt[12:20], collapse = " ")
txt <- strsplit(txt, "\\s{2,}")[1]
```

There is a problem, though: some large numbers are written with a single space between (for example Africa 10 596), so now there is a character vector “10 596” which we want to turn into a number:

```
as.numeric("10 596")
```

```
[1] NA
```

so we need to remove those white spaces. It would be easy to remove all of them, but then we would turn “Puerto Rico” into “PuertoRico”, and we don’t want that!

While we are at it, let’s also remove the superscripts on some of the country names.

```
for(i in seq_along(txt)) {
 tmp <- strsplit(txt[i], "\\s+")[[1]]
 if(length(tmp)==1) next #single item, nothing to do
 if(any(is.na(as.numeric(tmp)))) { #some parts are character
 if(!all(is.na(as.numeric(tmp)))) #not all parts are character
 tmp <- tmp[is.na(as.numeric(tmp))]
 #drop numbers (superscripts)
 txt[i] <- paste(tmp, collapse = " ")
 #all text, leave space between
 }
 else
 txt[i] <- paste(tmp, collapse = "")
 #all numbers, no spaces
}
```

For some reasons some are not working (maybe there was more than one white space?), so we fix them directly:

```
k <- seq_along(txt)[txt=="Western Africa"]
txt[k+c(0,1)]

[1] "Western Africa" "3"

txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="China"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="South-Central Asia"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Malaysia"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Azerbaijan"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Cyprus"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Republic of Moldova"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Northern Europe"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Norway"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Southern Europe"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Serbia"]
txt <- txt[-(k+1)]
```

```

k <- seq_along(txt)[txt=="Spain"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="TFYR Macedonia"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Caribbean"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Guadeloupe"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="NORTHERN AMERICA"]
txt <- txt[-(k+1)]
k <- seq_along(txt)[txt=="Micronesia"]
txt <- txt[-(k+1)]

```

How can we pick out the parts of text that we want?

Notice that the information starts with Burundi, so we remove everything before that:

```

k <- seq_along(txt)[txt=="Burundi"]
txt <- txt[k:length(txt)]

```

The last country is Tonga, so we remove everything after it's row:

```

k <- seq_along(txt)[txt=="Tonga"]
txt <- txt[1:(k+13)]

```

Unfortunately the table is not contiguous, eventually there is a page break and then the title text repeats. However, this part always starts with the words “World Mortality” and ends with “(13)”, so it is easy to remove all of them:

```

k <- seq_along(txt)[txt=="World Mortality"]
j <- seq_along(txt)[txt=="(13)"]
k

[1] 155 569 1049 1517 1983 2451 2917 3385
j

```

```

[1] 203 614 1095 1562 2029 2496 2963 3430

```

so we see that the first “World Mortality” is at position 155, and the first “(13)” is at 203, so we can remove everything in between.

But wait: the top of page 13 reads “World Mortality” on the left and “13” on the right, but on page 14 it is “14” on the left and “World Mortality” on the right! These alternate, so we need

```

k[c(2, 4, 6, 8)] <- k[c(2, 4, 6, 8)]-1

```

Let's get rid of all of this:

```

for(i in 1:8) txt[k[i]:j[i]] <- NA
txt <- txt[!is.na(txt)]

```

Now we can turn this into a data frame:

```
data.tbl <- as.data.frame(matrix(txt, byrow = TRUE, ncol=14))
for(i in 2:14)
 data.tbl[, i] <- as.numeric(data.tbl[, i])
colnames(data.tbl) <- c("Country",
 "Deaths", "Rate", "LifeExpectancy.Both",
 "LifeExpectancy.Males", "LifeExpectancy.Females",
 "InfantMortality", "UnderFive", "Prob15.60", "Prob0.70",
 "PercUnder5", "PercUnder5.25", "Perc25.65", "PercOver65")
row.names(data.tbl) <- NULL
```

Let's check a few to make sure we got it right:

```
k <- seq_along(txt)[txt=="Germany"]
txt[k:(k+13)]
[1] "Germany" "910" "11.1" "80.8" "78.4" "83.2" "3"
[8] "4" "71" "170" "0" "0" "14" "85"

k <- seq_along(txt)[txt=="Puerto Rico"]
txt[k:(k+13)]
[1] "Puerto Rico" "29" "7.9" "79.7" "75.8"
[6] "83.6" "6" "7" "96" "199"
[11] "1" "1" "21" "77"
```

One final problem: every now and then the table has the means for various regions (Middle Africa, etc). There is no other way but to get rid of them one by one. Just going through the list I can find their row numbers: ‘

```
not.country <- c(21, 31, 39, 45, 62, 63, 72, 73, 79, 89,
 101, 120, 121, 132, 144, 157, 165, 166,
 184, 193, 207, 210)
data.tbl <- data.tbl[-not.country,]

dump("data.tbl", "world.mortality.2017.R")
```

Let's look at the life expectancy, sorted from highest to lowest:

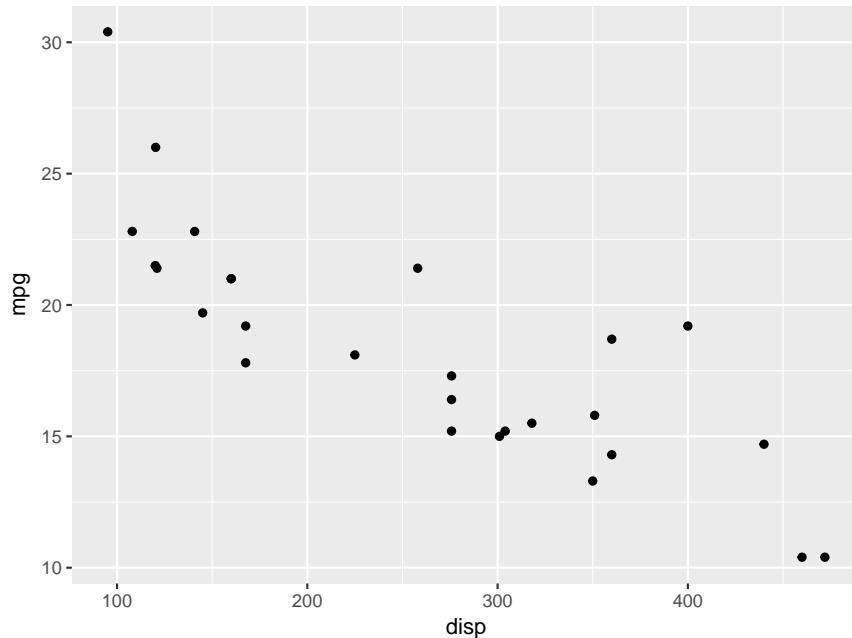
```
dta <- data.tbl[order(data.tbl[, "LifeExpectancy.Both"],
 decreasing = TRUE), c(1, 4)]
colnames(dta)[2] <- "Life Expectancy"
row.names(dta) <- NULL
kable(dta)
```

Country	Life Expectancy
China, Hong Kong SAR	83.8
China, Macao SAR	83.7
Japan	83.6
Switzerland	83.1
Spain	83.0
Singapore	82.8
Italy	82.8
Australia	82.7
Iceland	82.6
Australia/New Zealand	82.6
France	82.4
Israel	82.3
Sweden	82.3
Canada	82.2
Norway	82.0
Republic of Korea	81.9
Martinique	81.8
Netherlands	81.7
New Zealand	81.7
Luxembourg	81.6
United Kingdom	81.4
Austria	81.4
Ireland	81.3
Finland	81.1
Guadeloupe	81.1
Channel Islands	81.0
Greece	81.0
Portugal	81.0
Belgium	81.0
Slovenia	80.8
Germany	80.8
Malta	80.7
Denmark	80.6
Cyprus	80.3
Réunion	80.1
Mayotte	79.9
China, Taiwan Province of China	79.7
Puerto Rico	79.7
French Guiana	79.7
Cuba	79.6
United States Virgin Islands	79.6
Costa Rica	79.6
Lebanon	79.4
Guam	79.4
Chile	79.3
United States of America	79.2
Czechia	184 78.6
Curaçao	78.3
Albania	78.2

## 23 The pipe, dplyr, tibbles, tidyverse

The traditional workflow of R comes in large part from other computer languages. So a typical sequence would be like this:

```
df <- mtcars
df1 <- subset(df, hp>70)
ggplot(data=df1, aes(disp, mpg)) +
 geom_point()
```



This is not how we think, though. That would go something like this:

- take the mtcars data set
- then pick only cars with hp over 70
- then do the scatterplot of mpg vs. disp

In addition there is also the issue that we had to create an intermediate data set (df1).

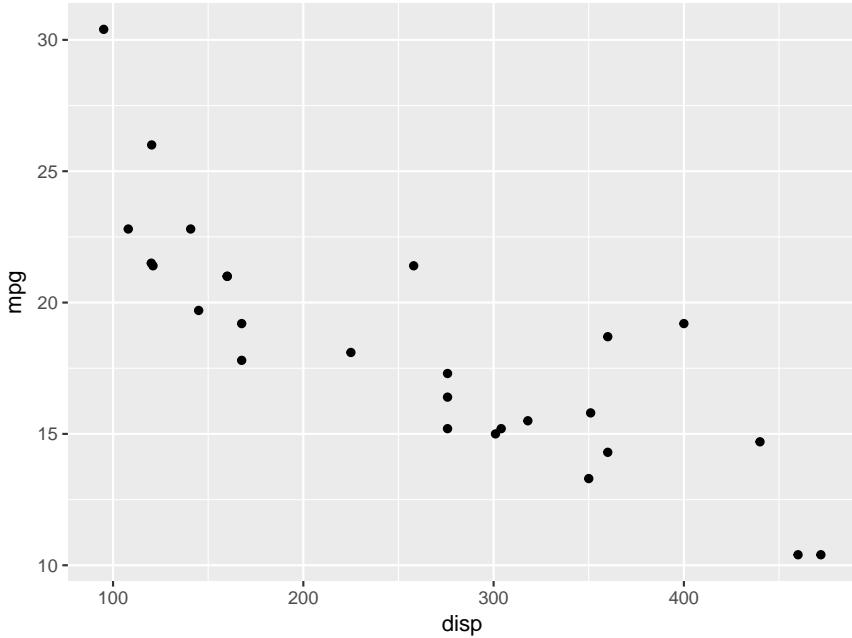
The *pipe* was invented to fix all of these problems.

The basic package to use piping is

```
library(magrittr)
```

invented by Stefan Milton. The basic operator is `%>%`. The same as above can be done with

```
mtcars %>%
 subset(x=., hp>70) %>%
 ggplot(data=., aes(disp, mpg)) +
 geom_point()
```



In principle the pipe can always be used in this way:

```
x <- rnorm(10)
round(mean(x), 3)

[1] -0.176

x %>%
 mean() %>%
 round(digits = 3)

[1] -0.176
```

Notice that here we called both mean and round without a needed argument. In principle the pipe will always use the data on the left of `%>%` as the first argument of the command on the right.

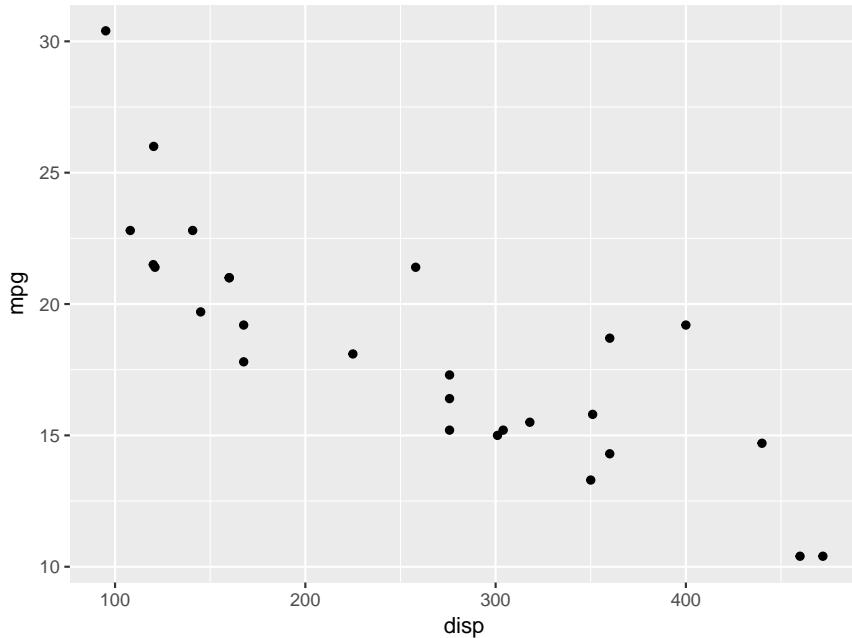
At first it may not seem like writing `x %>% f()` is any easier than writing `f(x)`, but this style of coding becomes very useful when applying multiple functions; in this case piping will allow one to think from left to right in the logical order of functions rather than from inside to outside in an ugly large nested statement of functions.

The pipe is only a few years old, but there are already many packages that take advantage of it. The most important one, and a package useful in and of itself, is

```
library(dplyr)
```

written by Hadley Wickham. In essence it is a replacement for the *apply* family of R routines. We can also write the above with

```
mtcars %>%
 filter(hp>70) %>%
 ggplot(aes(disp, mpg)) +
 geom_point()
```



Notice how *filter* is aware of the pipe, it doesn't need to be told that it is supposed to work with mtcars. So far, *ggplot* is not fully pipe aware (otherwise we could have written `%>% geom_point()`), but this will change in the near future.

### 23.1 tibbles

Dataframes have been the main data format of R since its beginnings, and are likely to stay that way for a long time. They do, however have some shortcomings. Among other things, when you type the name of a dataframe and hit enter, all of it is shown, even if the data set is huge. On the other hand, interesting information such as the data types of the columns is not shown. To eliviate these (and some other) issues the data format *tibble* was invented. We can turn a dataframe into a tibble with

```
tmtcars <- as.tbl(mtcars)
tmtcars
```

```
A tibble: 32 x 13
mpg cyl disp hp drat wt qsec vs am gear carb
* <dbl> <dbl>
1 21 6 160 110 3.9 2.62 16.5 0 1 4 4
2 21 6 160 110 3.9 2.88 17.0 0 1 4 4
3 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1
4 21.4 6 258 110 3.08 3.22 19.4 1 0 3 1
```

```

5 18.7 8 360 175 3.15 3.44 17.0 0 0 3 2
6 18.1 6 225 105 2.76 3.46 20.2 1 0 3 1
7 14.3 8 360 245 3.21 3.57 15.8 0 0 3 4
8 24.4 4 147. 62 3.69 3.19 20 1 0 4 2
9 22.8 4 141. 95 3.92 3.15 22.9 1 0 4 2
10 19.2 6 168. 123 3.92 3.44 18.3 1 0 4 4
... with 22 more rows, and 2 more variables: faccyl <ord>, facgear <ord>

```

so we have all relevant information about the data set: its size (32x11), the variables and their formats, and the begining of the data set.

*tibbles* are also designed to work well with piping and with the package *dplyr*.

If you want to create a tibble from scratch use:

```
tibble(x=1:5, y=x^2)
```

```

A tibble: 5 x 2
x y
<int> <dbl>
1 1 1
2 2 4
3 3 9
4 4 16
5 5 25

```

Also, tibbles never use row.names, and it only recycles vectors of length 1. This is because recycling vectors of greater lengths is a frequent source of bugs.

## 23.2 dplyr

We have already seen the filter command, the *dplyr* version of subset. Here are the most important *dplyr* commands:

- filter selects part of a data set by conditions (base R command: subset)
- select selects columns (base R command: [ ])
- arrange re-orders or arranges rows (base R commands: sort, order)
- mutate creates new columns (base R commands: any math function)
- summarise summarises values (base R commands: mean, median etc)
- group\_by allows for group operations in the “split-apply-combine” concept (base R command: none)

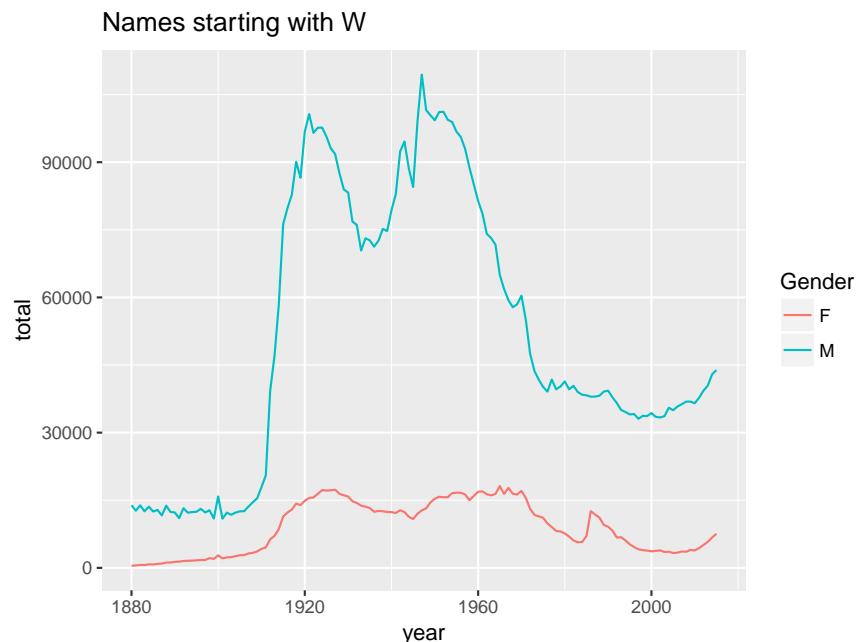
### 23.2.1 Example: babynames

The library *babynames* (also by Hadley Wickham) has the number of children of each sex given each name for each year from 1880 to 2015 according to the US census. All names with more than 5 uses are given.

We want to do the following:

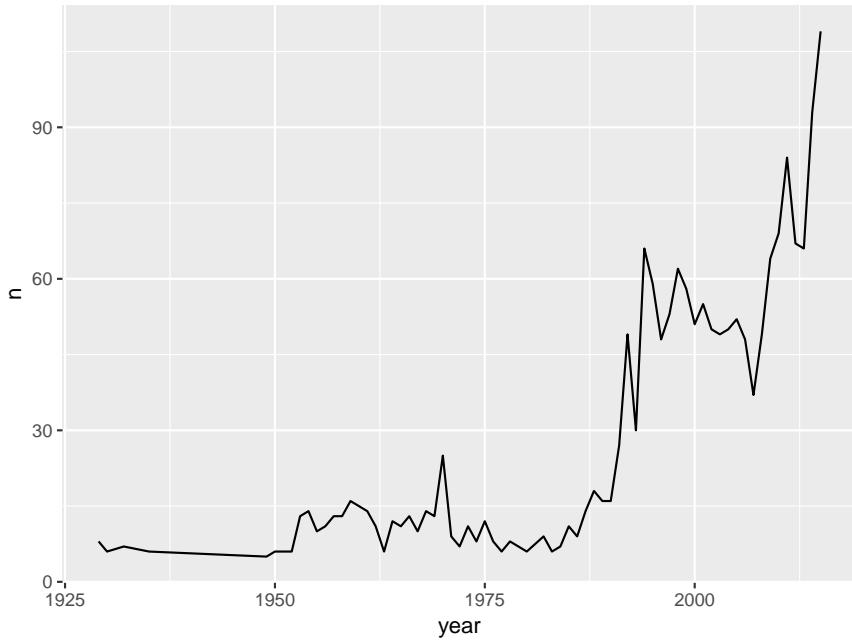
- take the names
- pick out all of those that start with “W”
- separate the genders
- find the total for each year
- do the line graph

```
library(babynames)
babynames %>%
 filter(name %>% substr(1, 1) %>% equals("W")) %>%
 group_by(year, sex) %>%
 summarize(total = sum(n)) %>%
 ggplot(data = ., aes(year, total, color = sex)) +
 geom_line() +
 labs(color="Gender") +
 ggtitle('Names starting with W')
```



How often is my name used for a baby in the US?

```
babynames %>%
 filter(name == "Wolfgang") %>%
 ggplot(data = ., aes(year, n)) +
 geom_line()
```



Looks like my name is getting more popular (even if it is still rare)!

What were the most popular girls names each year?

```

babynames %>%
 filter(sex=="F") %>%
 group_by(year) %>%
 mutate(M=max(n)) %>%
 filter(n==M) %>%
 ungroup() %>%
 select(name) %>%
 table() %>%
 sort(decreasing = TRUE) %>% #then organize data
 cbind() # then turn data around for easier reading

```

```

.
Mary 76
Jennifer 15
Emily 12
Jessica 9
Lisa 8
Linda 6
Emma 3
Sophia 3
Ashley 2
Isabella 2

```

Let's say we want to save a data set made with the pipe. Logically we should be able to do this

```

babynames %>% #take babynames
 filter(name=="Wolfgang") %>% #then pick me
 wolfgangs #then give new data set a name

```

```
Error in wolfgangs(.): could not find function "wolfgangs"
```

but that results in an error, only functions can be used in a pipe. So it is done like this:

```

wolfgangs <- babynames %>% #take babynames
 filter(name=="Wolfgang") #then pick me
 print(wolfgangs, n=3)

```

```

A tibble: 69 x 5
year sex name n prop
<dbl> <chr> <chr> <int> <dbl>
1 1929 M Wolfgang 8 0.00000722
2 1930 M Wolfgang 6 0.00000531
3 1932 M Wolfgang 7 0.00000652
... with 66 more rows

```

This unfortunately breaks the logic of piping. There is a better way, though. Just remember the logic of the assignment character <-, it's an arrow!

```

babynames %>% #take babynames
 filter(name=="Wolfgang") -> #then pick me
 wolfgangs #then assign it a name
 print(wolfgangs, n=3)

```

```

A tibble: 69 x 5
year sex name n prop
<dbl> <chr> <chr> <int> <dbl>
1 1929 M Wolfgang 8 0.00000722
2 1930 M Wolfgang 6 0.00000531
3 1932 M Wolfgang 7 0.00000652
... with 66 more rows

```

---

Here is a common problem: say you have these two data sets:

```
students1
```

```

A tibble: 3 x 2
name exam1
<chr> <dbl>
1 Alex 78
2 Ann 85
3 Marie 93

```

```
students2
```

```
A tibble: 3 x 2
```

```

name exam2
<chr> <dbl>
1 Alex 75
2 Ann 89
3 Marie 97

```

and we want to join them into one data set:

```

students1 %>%
 left_join(students2)

A tibble: 3 x 3
name exam1 exam2
<chr> <dbl> <dbl>
1 Alex 78 75
2 Ann 85 89
3 Marie 93 97

```

Let's say we want to find the times out of 100000 that the most popular names occurred in the 2015:

```

babynames %>%
 filter(year==2015) %>%
 mutate(freq=round(n/sum(n)*100000)) %>%
 select(name, freq) %>%
 arrange(desc(freq)) %>%
 print(n=5)

```

```

A tibble: 32,952 x 2
name freq
<chr> <dbl>
1 Emma 555
2 Olivia 533
3 Noah 532
4 Liam 498
5 Sophia 472
... with 3.295e+04 more rows

```

so the *mutate* command let's us calculate new variables and the *arrange* command let's us change the order of the rows.

### 23.3 The tidyverse

*ggplot2* and *dplyr* are two of a number of packages that together form the *tidyverse*. They are centered around what is called *tidy data*. For a detailed discussion go to <https://www.tidyverse.org/>.

The core packages are

- ggplot2
- dplyr
- tidyr
- readr
- purrr
- tibble
- stringr
- forcats

but you can get all of them in one step with

```
install.packages("tidyverse")
```

*tidy* data is defined as data were

1. Each variable you measure should be in one column.
2. Each different observation of that variable should be in a different row.
3. There should be one table for each “kind” of variable.
4. If you have multiple tables, they should include a column in the table that allows them to be linked.

This is essentially the definition of a data frame, but it is enforced even more so by the *tibbles* format. The theory behind tidy data was described by Hadley Wickham in the article Tidy Data, Journal of Statistical Software. The packages in the tidyverse are all written to have a consistent look and feel and work naturally with tidy data.

One big difference between dataframes and tibbles is that tibbles automatically ignore row names:

```
head(mtcars, 3)
```

```
mpg cyl disp hp drat wt qsec vs am gear carb faccyl
Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4 6
Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4 6
Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1 4
facgear
Mazda RX4 4
Mazda RX4 Wag 4
Datsun 710 4
```

```

tbl.mtcars <- as.tbl(mtcars)
print(tbl.mtcars, n=3)

A tibble: 32 x 13
mpg cyl disp hp drat wt qsec vs am gear carb faccyl
* <dbl> <ord>
1 21 6 160 110 3.9 2.62 16.5 0 1 4 4 6
2 21 6 160 110 3.9 2.88 17.0 0 1 4 4 6
3 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1 4
... with 29 more rows, and 1 more variable: facgear <ord>

```

This of course is no good here, the names of the cars are important. One way to fix this is to use the *rownames\_to\_column* routine in the *tibbles* package:

```

library("tibble")
mtcars %>%
 as.tbl() %>%
 rownames_to_column() %>%
 print(n=3)

A tibble: 32 x 14
rowname mpg cyl disp hp drat wt qsec vs am gear
<chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4 21 6 160 110 3.9 2.62 16.5 0 1 4
2 Mazda RX4 W~ 21 6 160 110 3.9 2.88 17.0 0 1 4
3 Datsun 710 22.8 4 108 93 3.85 2.32 18.6 1 1 4
... with 29 more rows, and 3 more variables: carb <dbl>, faccyl <ord>,
facgear <ord>

```

One difficulty is to remember which routine is in what package. The best way is to simply load them all with

```
library(tidyverse)
```

## 24 Character Manipulation with stringr

```

library(magrittr)
library(dplyr)

```

Thanks to Hadley Wickham, we have the package *stringr* that adds more functionality to the base functions for handling strings in R. According to the description of the package at <http://cran.r-project.org/web/packages/stringr/index.html> *stringr* is a set of simple wrappers that make R's string functions more consistent, simpler and easier to use. It does this by ensuring that: function and argument names (and positions) are consistent, all functions deal with NA's and zero length character appropriately, and the output data structures from each function matches the input data structures of other functions.

We previously looked at the states in the US:

```
head(states)

[1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"
[6] "Colorado"
```

We can find out how many letters each state has with

```
library(stringr)
states %>%
 str_length()

[1] 7 6 7 8 10 8 11 8 20 7 7 6 5 8 7 4 6 8 9 5 8 13 8
[24] 9 11 8 7 8 6 13 10 10 8 14 12 4 8 6 12 12 14 12 9 5 4 7
[47] 8 10 13 9 7 11
```

Also, we found out how many vowels the names of each state had. Here is how we can do that with *stringr*:

```
states %>%
 str_count("a")

[1] 3 2 1 2 2 1 0 2 1 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2
[36] 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0 0
```

Notice that we are only getting the number of a's in lower case. Since `str_count()` does not contain the argument `ignore.case`, we need to transform all letters to lower case, and then count the number of a's like this:

```
states %>%
 tolower() %>%
 str_count("a")

[1] 4 3 2 3 2 1 0 2 1 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2
[36] 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0 0
```

Now let's do this for all the vowels:

```
vowels <- c("a", "e", "i", "o", "u")
states %>%
 tolower() %>%
 str_split("") %>%
 unlist() %>%
 table() ->
 x
x[vowels]

.
a e i o u
62 29 48 40 10
```

*stringr* provides functions for both

1) basic manipulations

2) regular expression operations

The following table contains the *stringr* functions for basic string operations:

Function	Description	Similar to
str_c()	string concatenation	paste()
str_length()	number of characters	nchar()
str_sub()	extracts substrings	substring()
str_dup()	duplicates characters	none
str_trim()	removes leading and trailing whitespace	none
str_pad()	pads a string	none
str_wrap()	wraps a string paragraph	strwrap()
str_trim()	trims a string	none

Here are some examples:

```
paste("It", "is", "a", "nice", "day", "today")
```

```
[1] "It is a nice day today"
```

```
str_c("It", "is", "a", "nice", "day", "today")
```

```
[1] "Itisanicedaytoday"
```

```
str_c("It", "is", "a", "nice", "day", "today",
 sep=" ")
```

```
[1] "It is a nice day today"
```

```
str_c("It", "is", "a", "nice", "day", "today",
 sep="-")
```

```
[1] "It-is-a-nice-day-today"
```

next str\_length. Compared to nchar() it can handle more data types, for example factors:

```
some_factor <- factor(c(1, 1, 1, 2, 2, 2),
 labels = c("good", "bad"))
some_factor
```

```
[1] good good good bad bad bad
Levels: good bad
```

```
str_length(some_factor)
```

```
[1] 4 4 4 3 3 3
```

whereas nchar(some\_factor) results in an error.

A routine that has no direct equivalent in basic R is str\_dup. It is sort of a rep for strings:

```
str_dup("ab", 2)

[1] "abab"

str_dup("ab", 1:3)

[1] "ab" "abab" "ababab"
```

Another handy function that we can find in stringr is str\_pad() for padding a string. This is useful if we want to have a nice alignment when printing some text.

Its default usage has the following form:

```
str_pad(string, width, side = "left", pad = "")
```

The idea of str\_pad() is to take a string and pad it with leading or trailing characters to a specified total width. The default padding character is a space (pad = ""), and consequently the returned string will appear to be either left-aligned (side = "left"), right-aligned (side = "right"), or both (side = "both").

Let's see some examples:

```
str_pad("Great!", width = 7)

[1] " Great!"

str_pad("Great", width = 8, side = "both")

[1] " Great "

str_pad("Great!", width = 7, pad="#")

[1] "#Great!"
```

Often when dealing with character vectors we end up with white spaces. These are easily taken care of with str\_trim:

```
txt <- c("some", " text", "with ", " white ", "space")
str_trim(txt)
```

```
[1] "some" "text" "with" "white" "space"
```

An operation that one needs to do quite often is to extract the last (few) letters from words. Using substring is tricky if the words don't have the same lengths:

```
substring(txt, nchar(txt)-1, nchar(txt))
```

```
[1] "me" "xt" "h" "e" "ce"
```

Much easier with

```
str_sub(txt, -2, -1)
```

```
[1] "me" "xt" "h" "e" "ce"
```

You can use str\_wrap() to modify existing whitespace in order to wrap a paragraph of text, such that the length of each line is as similar as possible.

```
declaration.of.independence <- "We hold these truths to be self-evident, that all men are
cat(str_wrap(declaration.of.independence, width=40))
```

```
We hold these truths to be self-evident,
that all men are created equal, that
they are endowed by their Creator with
certain unalienable Rights, that among
these are Life, Liberty and the pursuit
of Happiness.
```

## 24.1 Dracula by Bram Stoker

Let's do a textual analysis of Bram Stoker's Dracula. We can get an electronic copy of the book from the Project Gutenberg <http://www.gutenberg.org/>. To get a book into R is very easy, there is a package:

```
library(gutenbergr)
dracula <- gutenberg_download(345)
dracula

A tibble: 15,568 x 2
gutenberg_id text
<int> <chr>
1 345 "DRACULA"
2 345 ""
3 345 ""
4 345 ""
5 345 ""
6 345 ""
7 345 ""DRACULA"
8 345 ""
9 345 "_by_"
10 345 ""
... with 15,558 more rows
```

Why 345? This is the id number used by the Gutenberg web site to identify this book. Go to their website and check out what other books they have (there are over 57000 as of 2018).

The first column is the gutenberg\_id, so we can get rid of that

```
dracula <- dracula[, 2]
```

Let's see the beginning of the book:

```
dracula[1:100,] %>%
 str_wrap(width=40) %>%
 cat()
```

```

c(" DRACULA", "", "", "", "", "", "",
" DRACULA", "", " _by_", "", "
Bram Stoker", "", " [Illustration:
colophon]", "", " NEW YORK", "", "
GROSSET & DUNLAP", "", " _Publishers_",
"", " Copyright, 1897, in the United
States of America, according", " to
Act of Congress, by Bram Stoker", "", "
[_All rights reserved._]", "", "
PRINTED IN THE UNITED STATES", " AT",
" THE COUNTRY LIFE PRESS, GARDEN CITY,
N.Y.", "", "", "", "", " TO", "", " MY
DEAR FRIEND", "", " HOMMY-BEG", "", "", "
"", "", "CONTENTS", "", "", "CHAPTER I",
" Page", "", "Jonathan Harker's Journal
1", "", "CHAPTER II", "", "Jonathan
Harker's Journal 14", "", "CHAPTER
III", "", "Jonathan Harker's Journal
26", "", "CHAPTER IV", "", "Jonathan
Harker's Journal 38", "", "CHAPTER V",
"", "Letters--Lucy and Mina 51", "", "
CHAPTER VI", "", "Mina Murray's Journal
59", "", "CHAPTER VII", "", "Cutting
from \"The Dailygraph,\\" 8 August 71",
"", "CHAPTER VIII", "", "Mina Murray's
Journal 84", "", "CHAPTER IX", "", "
Mina Murray's Journal 98", "", "CHAPTER
X", "", "Mina Murray's Journal 111",
"", "CHAPTER XI", "", "Lucy Westenra's
Diary 124", "", "CHAPTER XII", "", "
Dr. Seward's Diary 136", "", "CHAPTER
XIII", "", "Dr. Seward's Diary 152",
"", "CHAPTER XIV", "", "Mina Harker's
Journal 167")

```

What are the most commonly used words in the book? Well, it will be something like “a”, “and” etc. Those kinds of words are called *stop\_words*, and they are not very interesting, and it might be better to just take them out. There are lists of such words. One of them is in the library *tidytext*:

```

library(tidytext)
stop_words

A tibble: 1,149 x 2
word lexicon
<chr> <chr>
1 a SMART
2 a's SMART

```

```

3 able SMART
4 about SMART
5 above SMART
6 according SMART
7 accordingly SMART
8 across SMART
9 actually SMART
10 after SMART
... with 1,139 more rows

```

So we now want to go through Dracula and remove all the appearances of any of the words in stop\_words. This can be done with the *dplyr* command *anti\_join*. However the two lists need to have the same column names, and in Dracula it is *text* whereas in stop\_words it is *word*. Again, the library *tidytext* has a command:

```

dracula %>%
 unnest_tokens(word, text) %>%
 anti_join(stop_words) ->
 dracula
dracula

```

```

A tibble: 48,552 x 1
word
<chr>
1 dracula
2 dracula
3 _by_
4 bram
5 stoker
6 illustration
7 colophon
8 york
9 grosset
10 dunlap
... with 48,542 more rows

```

So now for the most common words:

```

dracula %>%
 count(word, sort=TRUE)

A tibble: 9,072 x 2
word n
<chr> <int>
1 time 390
2 van 323
3 night 310
4 helsing 301
5 dear 224

```

```

6 lucy 223
7 day 220
8 hand 210
9 mina 210
10 door 200
... with 9,062 more rows

```

so *time* is the most common word, it appears 390 times in the book.

How do the words in Dracula compare to another famous fiction book of the era, The Time Machine, by H. G. Wells? This is book #35 in the Gutenberg catalogue:

```

time.machine <- gutenberg_download(35)[, 2]
time.machine %>%
 unnest_tokens(word, text) %>%
 anti_join(stop_words) ->
 time.machine

time.machine %>%
 count(word, sort=T)

A tibble: 4,135 x 2
word n
<chr> <int>
1 time 200
2 machine 85
3 white 59
4 traveller 55
5 world 52
6 hand 49
7 morlocks 46
8 people 46
9 weena 46
10 found 44
... with 4,125 more rows

```

Actually, *time* is the most common word in both books (not a surprise in a book called The Time Machine!)

Can we do a graphical display of the word frequencies? We will need some routines from yet another package called *tidyverse*.

We begin by joining the two books together, with a new column identifying the book:

```

library(tidyverse)
freqs <- bind_rows(mutate(dracula, book="Dracula"),
 mutate(time.machine, book="Time Machine"))
freqs

A tibble: 59,663 x 2
word book

```

```

<chr> <chr>
1 dracula Dracula
2 dracula Dracula
3 _by_ Dracula
4 bram Dracula
5 stoker Dracula
6 illustration Dracula
7 colophon Dracula
8 york Dracula
9 grosset Dracula
10 dunlap Dracula
... with 59,653 more rows

```

Next we add some useful columns:

```

freqs %>%
 mutate(word=str_extract(word, "[a-z']+")) %>%
 #take out things like , etc
 count(book, word) %>%
 group_by(book) %>%
 mutate(prop=n/sum(n)) %>% #take into account
 #different lengths of the books
 ungroup() %>%
 filter(n>10) %>% #consider only words used frequently
 select(-n) %>% #not needed anymore
 arrange(desc(prop)) ->
 freqs

```

Next we find all the words that appear in both books, and look at their relative proportions:

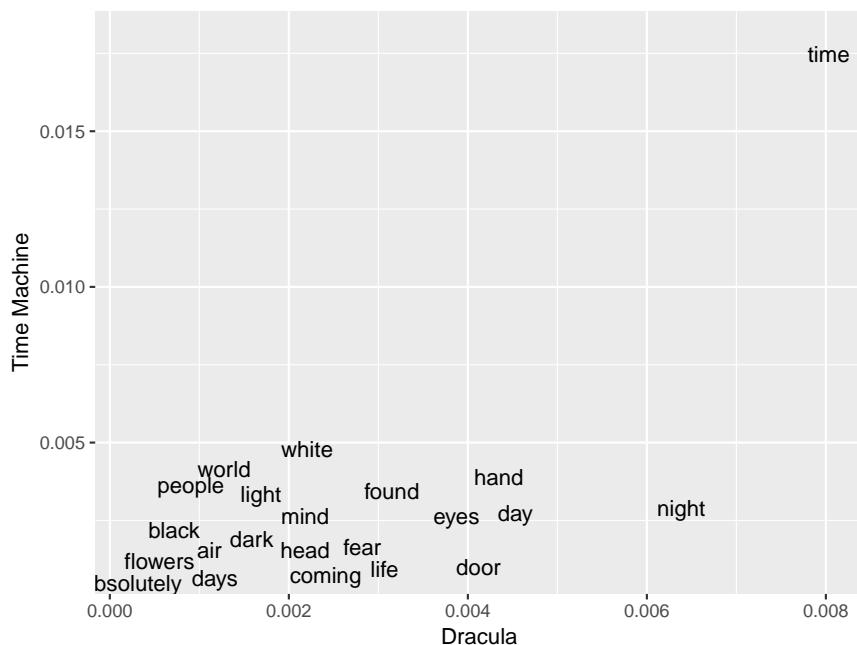
```

freqs %>%
 spread(key=book, value = prop) %>%
 #use one column for Dracula's
 #proportions and another for Time Machine
 na.omit() -> #words that appear in only one book are NA,
 #eliminate them
 common.words
print(common.words, n=4)

A tibble: 103 x 3
word Dracula `Time Machine`
<chr> <dbl> <dbl>
1 absolutely 0.000247 0.000990
2 age 0.000288 0.00126
3 air 0.00111 0.00207
4 altogether 0.000330 0.000990
... with 99 more rows

```

```
common.words %>%
 ggplot(aes(x=Dracula, y= common.words[, "Time Machine"])) +
 labs(x = "Dracula",
 y = "Time Machine") +
 geom_text(aes(label = word),
 check_overlap = TRUE,
 vjust = 1.5)
```



## 25 Dates with lubridate

```
library(tidyverse)
library(lubridate)
```

For a more detailed discussion see Dates and Times Made Easy with lubridate.

At first glance working with dates and times doesn't seem so complicated, but consider the following questions:

- Where all of these years leap years: 1800, 1900, 2000?
- Does every day have 24 hours?
- Does every minute have 60 seconds?

The answer to all these questions is: NO

- in principle every year divisible by 4 is a leap year, except if it is also divisible by 100, but not if also divisible by 400! So 1800 and 1900 were not leap years. 2000 was.

- In countries that have Summer Time there are two days with 23 and 25 hours, respectively.
- Even the above is not enough to bring the time it takes the earth to orbit the sun in perfect alignment with the calendar year, so every now and then there is a minute that has 61 seconds, called a leap second. Since this system of correction was implemented in 1972, 27 leap seconds have been inserted, the most recent on December 31, 2016 at 23:59:60.

There are also many regional differences in how date and time are written:

- USA: 4/29/2018 2.30pm
- Germany: 29/4/2018 14.30

Imagine you need to analyse some stock market data, starting from 1980 to today and in second intervals. You would need to include all of these details!

### 25.0.1 Create a date object

to get todays time and date:

```
today()
```

```
[1] "2018-08-12"
now()
[1] "2018-08-12 11:00:12 -04"
```

there are a number of ways to create a specific date object from a string:

```
ymd("2018-04-29")
[1] "2018-04-29"
mdy("April 29th, 2018")
[1] "2018-04-29"
dmy("29-April-2018")
[1] "2018-04-29"
```

this also works:

```
ymd(20180429)
[1] "2018-04-29"
to add time info use an underscore and the format:
ymd_hm("2018-04-29 2:30 PM")
[1] "2018-04-29 14:30:00 UTC"
```

```
dmy_hms("29-April-2018 2:30:45 PM")
```

```
[1] "2018-04-29 14:30:45 UTC"
```

As an example we will use the data set *flights* in the package *nycflights13*. It has airline on-time data for all flights departing NYC in 2013.

```
library(nycflights13)
```

```
flights %>%
```

```
print(n=4)
```

```
A tibble: 336,776 x 19
year month day dep_time sched_dep_time dep_delay arr_time
<int> <int> <int> <int> <int> <dbl> <int>
1 2013 1 1 517 515 2 830
2 2013 1 1 533 529 4 850
3 2013 1 1 542 540 2 923
4 2013 1 1 544 545 -1 1004
... with 3.368e+05 more rows, and 12 more variables:
sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Let's start by calculating the hour and minutes of the departure from the dep-time. For this we can use `%/%` for integer devision and `%%` for modulo:

```
flights %>%
```

```
 mutate(hour=dep_time %/% 100,
```

```
 minute=dep_time %% 100) ->
```

```
flights
```

In this tibble the parts of the time and date info are in several columns. Let's start by putting them together:

```
flights %>%
```

```
 select(year, month, day, hour, minute) %>%
```

```
print(n=4)
```

```
A tibble: 336,776 x 5
year month day hour minute
<int> <int> <int> <dbl> <dbl>
1 2013 1 1 5 17
2 2013 1 1 5 33
3 2013 1 1 5 42
4 2013 1 1 5 44
... with 3.368e+05 more rows
```

To combine the different columns into one date/time object we can use the command *make\_datetime*:

```

flights %>%
 select(year, month, day, hour, minute) %>%
 mutate(departure =
 make_datetime(year, month, day, hour, minute)) ->
 flights
flights %>%
 select(departure) %>%
 print(n=4)

A tibble: 336,776 x 1
departure
<dttm>
1 2013-01-01 05:17:00
2 2013-01-01 05:33:00
3 2013-01-01 05:42:00
4 2013-01-01 05:44:00
... with 3.368e+05 more rows

```

## 25.1 Time Spans

lubridate has a number of functions to do arithmetic with dates. For example, my age is `today()`

```

[1] "2018-08-12"
my.age <- today() - ymd(19610602)
as.duration(my.age)

[1] "1804896000s (~57.19 years)"

```

## 26 Factors with `forcats`

We have previously discussed factors, that is categorical data with fixed values and ordering. Now we will discuss the package *forcats*, which has a number of useful functions when working with factors.

```

library(tidyverse)
library(forcats)

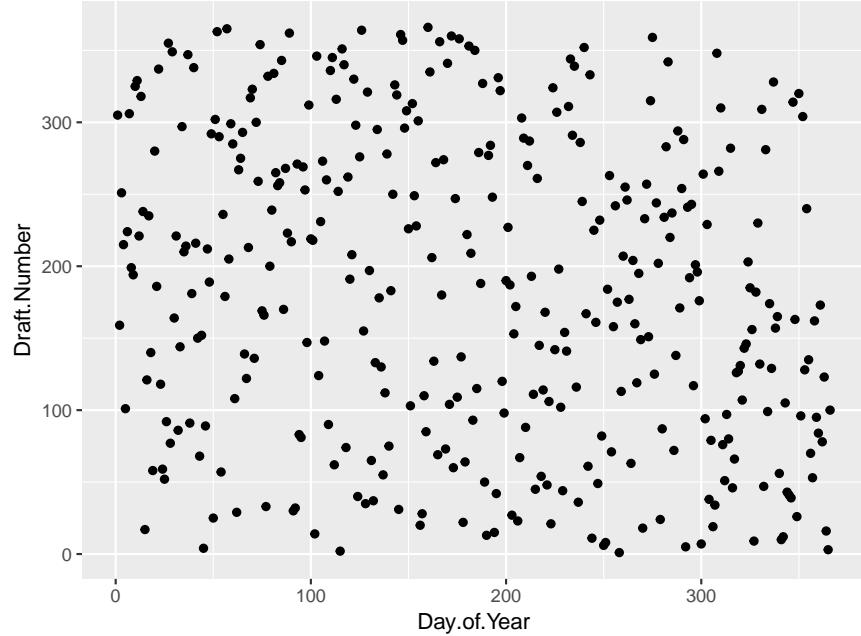
```

Let's remind ourselves of the base R commands first. Consider the data set *draft*, with the results of the 1970s military draft:

```

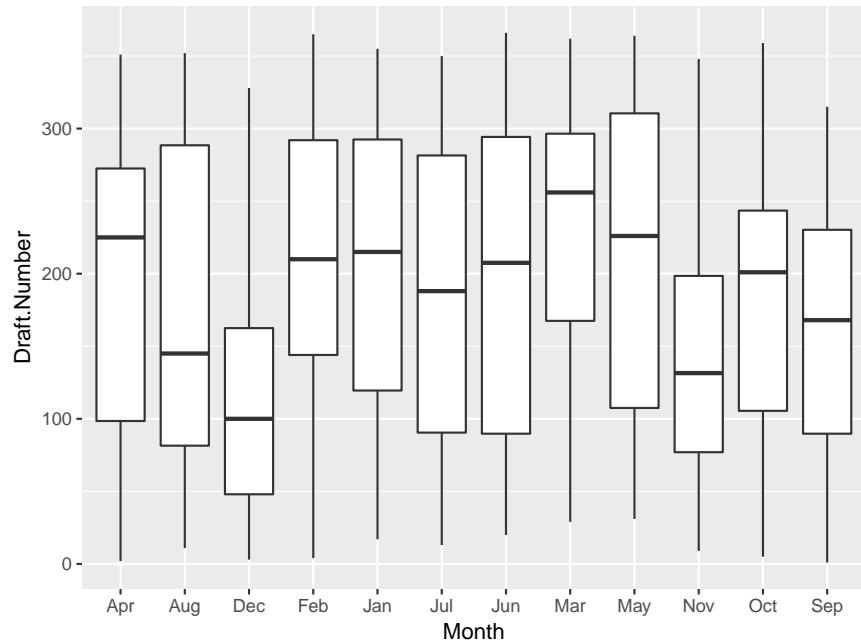
draft %>%
 ggplot(aes(Day.of.Year, Draft.Number)) +
 geom_point()

```



Let's say instead we want to do a box plot of the draft numbers by month:

```
draft %>%
 ggplot(aes(Month, Draft.Number)) +
 geom_boxplot()
```



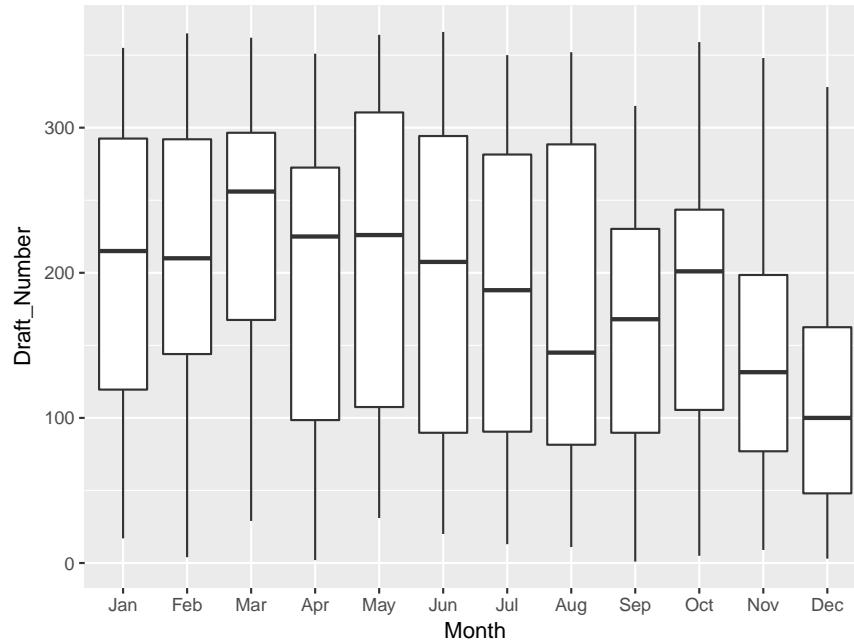
Now this is no good, the ordering of the boxes is alphabetic. So we need to change the variable Month to a factor:

```
lvls <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
```

```

Month.fac <- factor(draft$Month,
 levels = lvl,
 ordered = TRUE)
df <- data.frame(Month=Month.fac,
 Draft_Number=draft$Draft.Number)
df %>%
 ggplot(aes(Month, Draft_Number)) +
 geom_boxplot()

```



Quite often the order we want is the order in which the values appear in the data set, then we can use

```
lvl <- unique(draft$Month)
```

Theforcats package includes a data set called gss\_cat:

```
gss_cat
```

```

A tibble: 21,483 x 9
year marital age race rincome partyid relig denom tvhours
<int> <fct> <int> <fct> <fct> <fct> <fct> <fct> <int>
1 2000 Never married 26 White $8000 t~ Ind,nea~ Prote~ South~ 12
2 2000 Divorced 48 White $8000 t~ Not str~ Prote~ Bapti~ NA
3 2000 Widowed 67 White Not app~ Indepen~ Prote~ No de~ 2
4 2000 Never married 39 White Not app~ Ind,nea~ Ortho~ Not a~ 4
5 2000 Divorced 25 White Not app~ Not str~ None Not a~ 1
6 2000 Married 25 White $20000 ~ Strong ~ Prote~ South~ NA
7 2000 Never married 36 White $25000 ~ Not str~ Chris~ Not a~ 3
8 2000 Divorced 44 White $7000 t~ Ind,nea~ Prote~ Luthe~ NA
9 2000 Married 44 White $25000 ~ Not str~ Prote~ Other 0

```

```
10 2000 Married 47 White $25000 ~ Strong ~ Prote~ South~ 3
... with 21,473 more rows
```

which has the results of the General Social Survey (<http://gss.norc.org>), which a survey in the US done by the University of Chicago. We will use it to illustrateforcats.

Let's begin by considering the variable race:

```
gss_cat$race %>%
 table()

.
Other Black White Not applicable
1959 3129 16395 0
```

We can do the same thing with tidyverse routines:

```
gss_cat %>%
 count(race)

A tibble: 3 x 2
race n
<fct> <int>
1 Other 1959
2 Black 3129
3 White 16395
```

Notice a bit of a difference: In the first case there is the Not applicable group but not in the second. This is because "race" is a factor and this is among its levels. The table command includes all levels, even if the count is 0, whereas count does not. This is likely what we want most times, but not all the times.

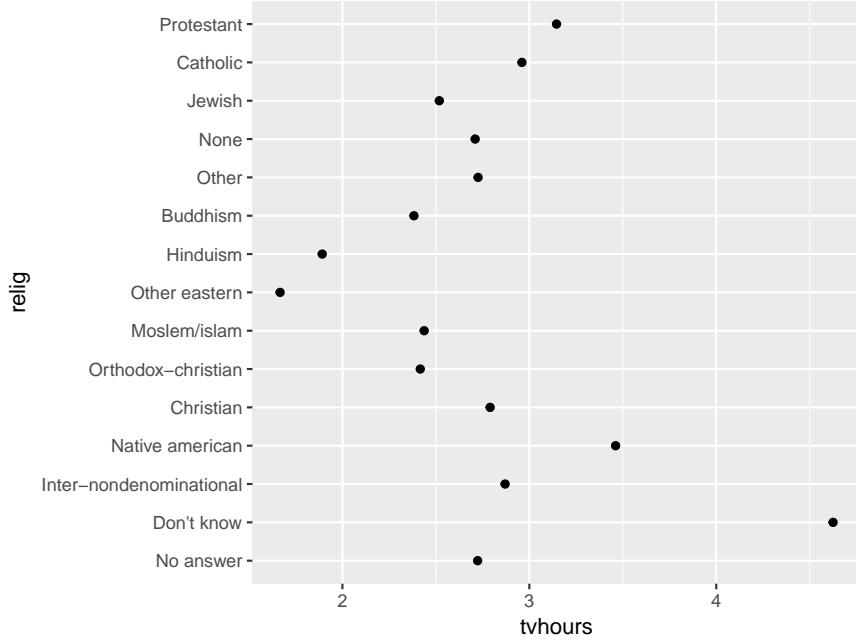
By the way, we can always find out what the levels are:

```
levels(gss_cat$race)

[1] "Other" "Black" "White" "Not applicable"
```

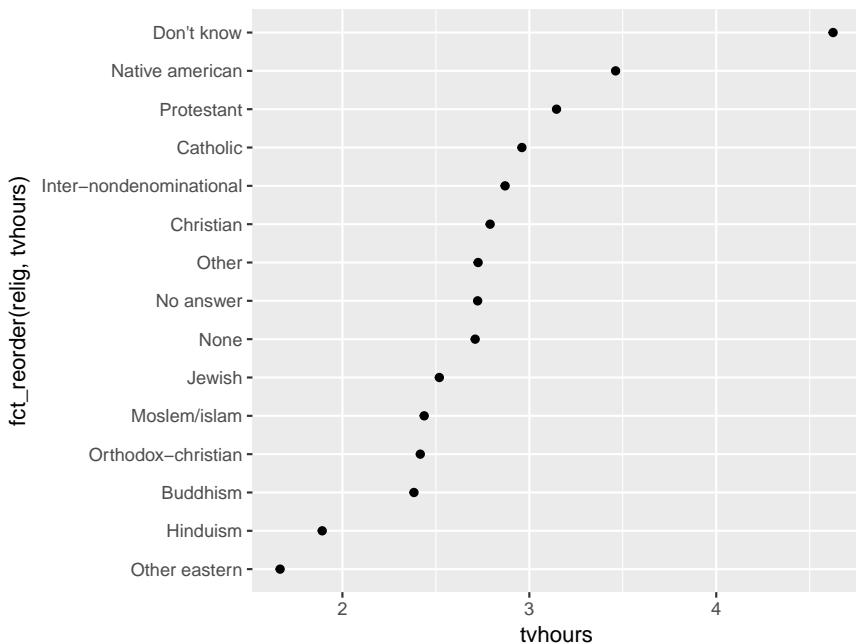
Let's consider the average number of hours that a person spends watching TV per day, depending on their religion:

```
gss_cat %>%
 group_by(relig) %>%
 summarize(
 age = mean(age, na.rm = TRUE),
 tvhours = mean(tvhours, na.rm = TRUE),
 n = n()
) ->
 tv.relig
tv.relig %>%
 ggplot(aes(tvhours, relig)) +
 geom_point()
```



This graph is hard to read, mainly because there is no ordering. But unlike Month the variable itself doesn't have any either. So maybe we should order by size:

```
tv.relig %>%
 ggplot(aes(tvhours, fct_reorder(relig, tvhours))) +
 geom_point()
```



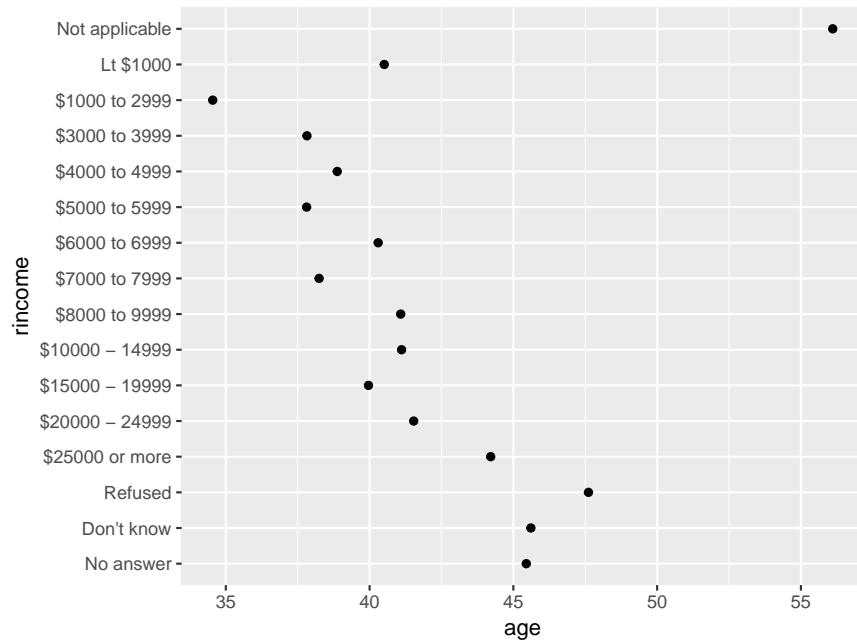
Let's see how income varies with age:

```
gss_cat %>%
 group_by(rincome) %>%
```

```

summarize(
 age = mean(age, na.rm = TRUE),
 n = n()
) ->
 rincome
rincome %>%
 ggplot(aes(age, rincome)) +
 geom_point()

```

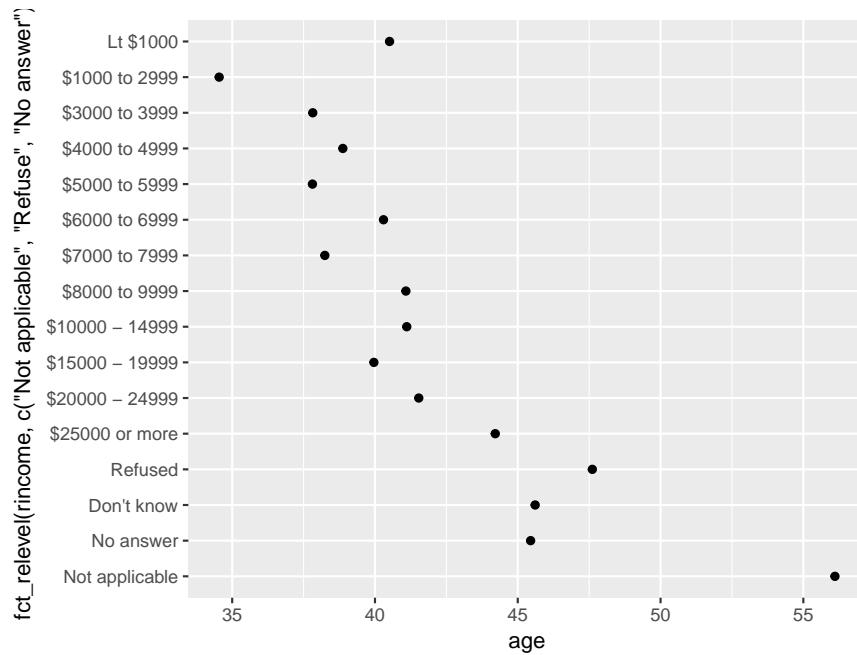


What ordering makes the most sense here? There are two types of levels: those with actual numbers, and those like “Not applicable”. We should probably separate them.

```

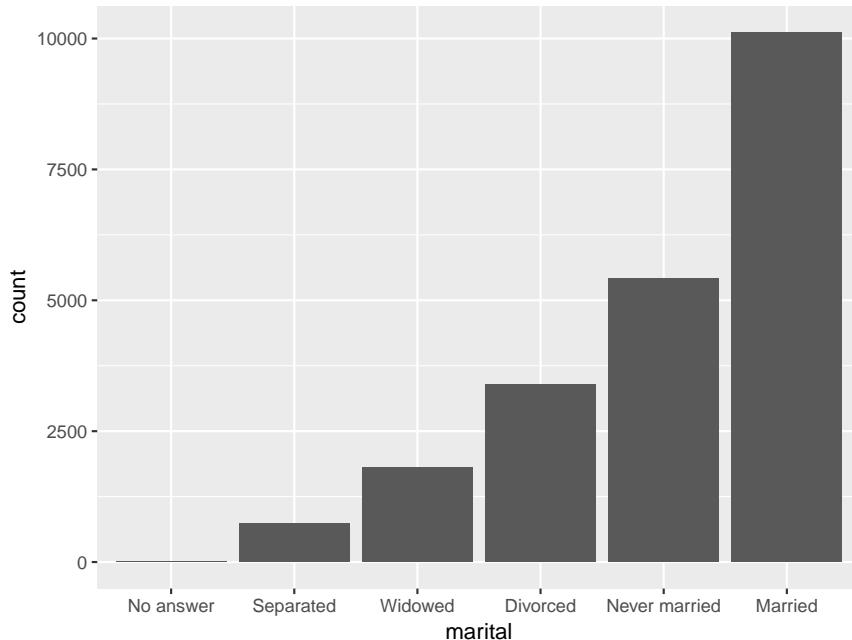
rincome %>%
 ggplot(aes(age, fct_relevel(rincome,
 c("Not applicable", "Refuse", "No answer")))) +
 geom_point()

```



In a bar graph the most common ordering is by size:

```
gss_cat %>%
 mutate(marital = marital %>%
 fct_infreq() %>% fct_rev()
) %>%
 ggplot(aes(marital)) +
 geom_bar()
```



## 27 Iteration with purrr

We have previously learned how to write loops and how to use apply. In this section we will learn about another variation on that theme.

We will need

```
library(tidyverse)
```

Say we have the following problem. We have a data frame

```
df <- tibble(
 a1 = round(rnorm(100), 1)
)
for(i in 2:50) {
 df[[i]] <- round(rnorm(100, i), 1)
 colnames(df)[i] <- paste0("a", i)
}
head(df, 2)

A tibble: 2 x 50
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12
<dbl> <dbl>
1 0.2 2.7 2.1 3.7 5 5 5.8 8.1 9.2 9.1 9.6 13.3
2 -1 2.3 1.9 5.4 7 6.7 7.2 9 8.9 11.1 10.5 12.9
... with 38 more variables: a13 <dbl>, a14 <dbl>, a15 <dbl>, a16 <dbl>,
a17 <dbl>, a18 <dbl>, a19 <dbl>, a20 <dbl>, a21 <dbl>, a22 <dbl>,
a23 <dbl>, a24 <dbl>, a25 <dbl>, a26 <dbl>, a27 <dbl>, a28 <dbl>,
a29 <dbl>, a30 <dbl>, a31 <dbl>, a32 <dbl>, a33 <dbl>, a34 <dbl>,
a35 <dbl>, a36 <dbl>, a37 <dbl>, a38 <dbl>, a39 <dbl>, a40 <dbl>,
a41 <dbl>, a42 <dbl>, a43 <dbl>, a44 <dbl>, a45 <dbl>, a46 <dbl>,
a47 <dbl>, a48 <dbl>, a49 <dbl>, a50 <dbl>
```

and we want to find the mean of the 50 columns.

We can of course use a loop:

```
df.mean <- rep(50, 0)
for(i in seq_along(df))
 df.mean[i] <- mean(df[[i]])
df.mean[1:5]

[1] -0.121 1.956 2.921 4.069 5.046
```

or we can use apply:

```
apply(df, 2, mean)[1:5]

a1 a2 a3 a4 a5
-0.121 1.956 2.921 4.069 5.046
```

and then there is the *purrr* routine

```
map_dbl(df, mean)[1:5]
```

```
a1 a2 a3 a4 a5
-0.121 1.956 2.921 4.069 5.046
```

What is the advantage of map over apply or even just the loop? One is that map is designed to work with the pipe:

```
df %>%
 map_dbl(mean) ->
 out
out[1:5]
```

```
a1 a2 a3 a4 a5
-0.121 1.956 2.921 4.069 5.046
```

Another is that *map* has a number of nice shortcuts. Say we want to fit a linear model to the mtcars data set but a different model for each cylinder:

```
mtcars %>%
 split(.cyl) %>%
 map(~lm(mpg~wt, data=.)) ->
 mtcars.models
summary(mtcars.models[[1]])
```

```

Call:
lm(formula = mpg ~ wt, data = .)

Residuals:
Min 1Q Median 3Q
-4.151278744075 -1.979513870382 -0.627246792778 1.929874065100
Max
5.252259561695

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 39.57119601304 4.34658202893 9.10398 7.7715e-06
wt -5.64702526124 1.85011852932 -3.05225 0.013743

Residual standard error: 3.33228288024 on 9 degrees of freedom
Multiple R-squared: 0.508632596323, Adjusted R-squared: 0.454036218137
F-statistic: 9.31623329642 on 1 and 9 DF, p-value: 0.0137427819867
```

Notice the shorthand *~lm(mpg~wt)* instead of the long form *function(df) lm(mpg~wt, data=df)*

Now say we want to see the R^2's:

```
mtcars.models %>%
 map(summary) %>%
```

```

map_dbl("r.squared")

4 6 8
0.508632596323140 0.464510150550548 0.422965536496111

```

Example: The data in the *gapminder* package has information on most of the countries in the world, including Puerto Rico. For a detailed explanation see its homepage at <https://www.gapminder.org/>.

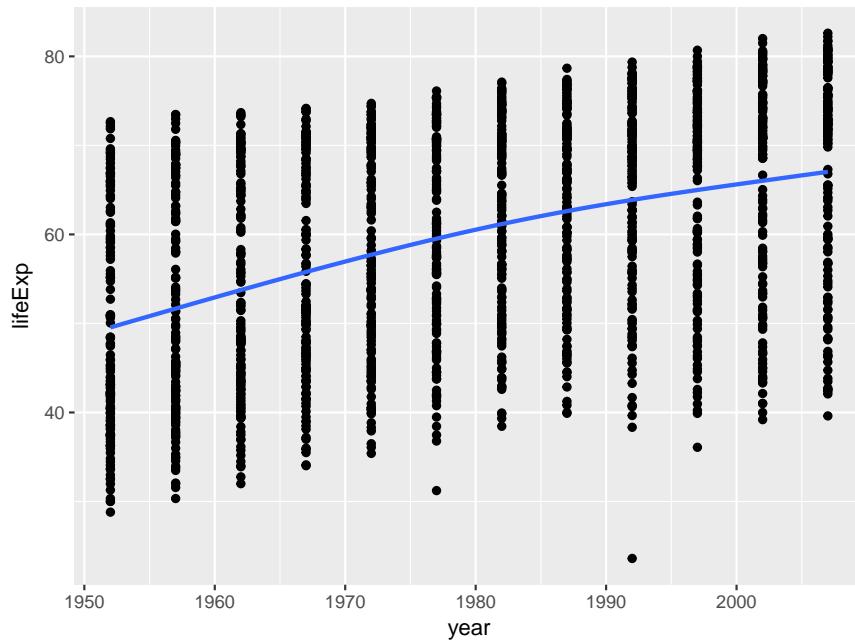
```
library(gapminder)
```

Let's consider for a moment the life expectancy per country over time:

```

gapminder %>%
 ggplot(aes(year, lifeExp)) +
 geom_point() +
 geom_smooth(se=FALSE)

```



Not surprisingly, the life expectancy has increased over the last 70 years or so. Overall we have

```

fit <- lm(gapminder$lifeExp ~ gapminder$year)
summary(fit)

```

```

##
Call:
lm(formula = gapminder$lifeExp ~ gapminder$year)
##
Residuals:
Min 1Q Median 3Q
-39.94923721166 -9.65088195164 1.69683934108 10.33476278834
Max

```

```

22.15791589382
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) -585.6521874415359 32.3139645168428 -18.12381 < 2.22e-16
gapminder$year 0.3259038276371 0.0163236858676 19.96509 < 2.22e-16

Residual standard error: 11.6305545853 on 1702 degrees of freedom
Multiple R-squared: 0.189757138522, Adjusted R-squared: 0.189281085137
F-statistic: 398.604745712 on 1 and 1702 DF, p-value: < 2.220446049e-16

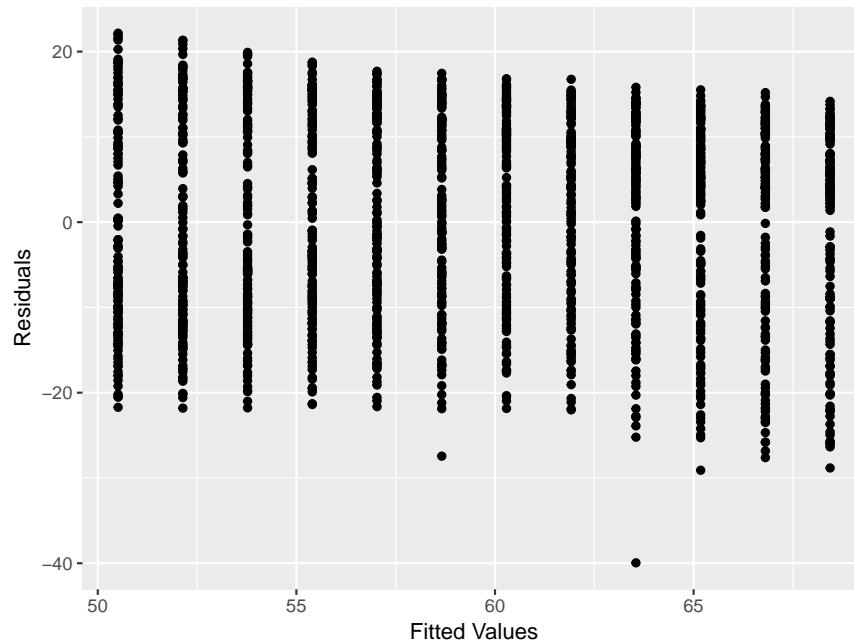
```

It does seem that the linear model is not quite right, and in fact

```

ggplot(data.frame(x=fitted.values(fit), y=residuals(fit)), aes(x,y)) +
 geom_point() +
 labs(x="Fitted Values", y="Residuals")

```



but instead of pursuing this issue let's find a linear model for each country:

```

gapminder %>%
 group_by(country) %>%
 nest() ->
 gap.country
gap.country

```

```

A tibble: 142 x 2
country data
<fct> <list>
1 Afghanistan <tibble [12 x 5]>
2 Albania <tibble [12 x 5]>

```

```

3 Algeria <tibble [12 x 5]>
4 Angola <tibble [12 x 5]>
5 Argentina <tibble [12 x 5]>
6 Australia <tibble [12 x 5]>
7 Austria <tibble [12 x 5]>
8 Bahrain <tibble [12 x 5]>
9 Bangladesh <tibble [12 x 5]>
10 Belgium <tibble [12 x 5]>
... with 132 more rows

```

What is this column “data”? As it says, it is a tibble again. So we have a tibble with a column, each entry of which is again a tibble!

Now

```

I.PR <- c(1:dim(gap.country)[1])[gap.country$country=="Puerto Rico"]
gap.country$data[[I.PR]]

A tibble: 12 x 5
continent year lifeExp pop gdpPercap
<fct> <int> <dbl> <int> <dbl>
1 Americas 1952 64.3 2227000 3082.
2 Americas 1957 68.5 2260000 3907.
3 Americas 1962 69.6 2448046 5108.
4 Americas 1967 71.1 2648961 6929.
5 Americas 1972 72.2 2847132 9123.
6 Americas 1977 73.4 3080828 9771.
7 Americas 1982 73.8 3279001 10331.
8 Americas 1987 74.6 3444468 12281.
9 Americas 1992 73.9 3585176 14642.
10 Americas 1997 74.9 3759430 16999.
11 Americas 2002 77.8 3859606 18856.
12 Americas 2007 78.7 3942491 19329.

gap.country %>%
 mutate(model = map(data, function(df) lm(lifeExp ~ year, data=df))) ->
 gap.country
gap.country

A tibble: 142 x 3
country data model
<fct> <list> <list>
1 Afghanistan <tibble [12 x 5]> <S3: lm>
2 Albania <tibble [12 x 5]> <S3: lm>
3 Algeria <tibble [12 x 5]> <S3: lm>
4 Angola <tibble [12 x 5]> <S3: lm>
5 Argentina <tibble [12 x 5]> <S3: lm>
6 Australia <tibble [12 x 5]> <S3: lm>
7 Austria <tibble [12 x 5]> <S3: lm>

```

```

8 Bahrain <tibble [12 x 5]> <S3: lm>
9 Bangladesh <tibble [12 x 5]> <S3: lm>
10 Belgium <tibble [12 x 5]> <S3: lm>
... with 132 more rows
summary(gap.country$model[[I.PR]])
```

```

##
Call:
lm(formula = lifeExp ~ year, data = df)
##
Residuals:
Min 1Q Median 3Q
-2.668525641026 -0.203235431235 0.397792540792 0.672507575758
Max
1.227103729604
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) -344.0935331002328 42.0048294087635 -8.19176 9.5630e-06
year 0.2105748251748 0.0212191122458 9.92383 1.7049e-06
##
Residual standard error: 1.26871838486 on 10 degrees of freedom
Multiple R-squared: 0.907819120324, Adjusted R-squared: 0.898601032356
F-statistic: 98.4823667891 on 1 and 10 DF, p-value: 1.70487604487e-06
```

Now that we have the linear models we can use them to do the graph. The best way to do this is to undo the `nest()` command and get back to a standard dataframe:

First we add the fitted values and the residuals to the tibble. For this we need a new package called `modelr`

```

library(modelr)
gap.country %>%
 mutate(fits = map2(data, model, add_predictions)) %>%
 mutate(resids = map2(data, model, add_residuals)) ->
 gap.country
gap.country
```

```

A tibble: 142 x 5
country data model fits resids
<fct> <list> <list> <list> <list>
1 Afghanistan <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
2 Albania <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
3 Algeria <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
4 Angola <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
5 Argentina <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
6 Australia <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
7 Austria <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
```

```

8 Bahrain <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
9 Bangladesh <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
10 Belgium <tibble [12 x 5]> <S3: lm> <tibble [12 x 6]> <tibble [12 x~
... with 132 more rows

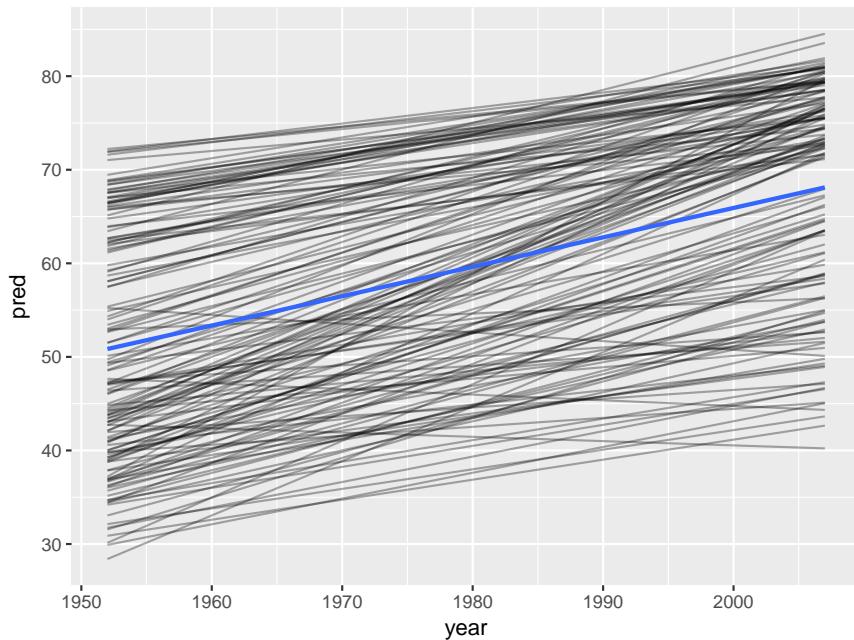
fits <- unnest(gap.country, fits)
fits

A tibble: 1,704 x 7
country continent year lifeExp pop gdpPercap pred
<fct> <fct> <int> <dbl> <int> <dbl> <dbl>
1 Afghanistan Asia 1952 28.8 8425333 779. 29.9
2 Afghanistan Asia 1957 30.3 9240934 821. 31.3
3 Afghanistan Asia 1962 32.0 10267083 853. 32.7
4 Afghanistan Asia 1967 34.0 11537966 836. 34.0
5 Afghanistan Asia 1972 36.1 13079460 740. 35.4
6 Afghanistan Asia 1977 38.4 14880372 786. 36.8
7 Afghanistan Asia 1982 39.9 12881816 978. 38.2
8 Afghanistan Asia 1987 40.8 13867957 852. 39.5
9 Afghanistan Asia 1992 41.7 16317921 649. 40.9
10 Afghanistan Asia 1997 41.8 22227415 635. 42.3
... with 1,694 more rows

resids <- unnest(gap.country, resids)

fits %>%
 ggplot(aes(year, pred)) +
 geom_line(aes(group=country), alpha=1/3) +
 geom_smooth(se=FALSE)

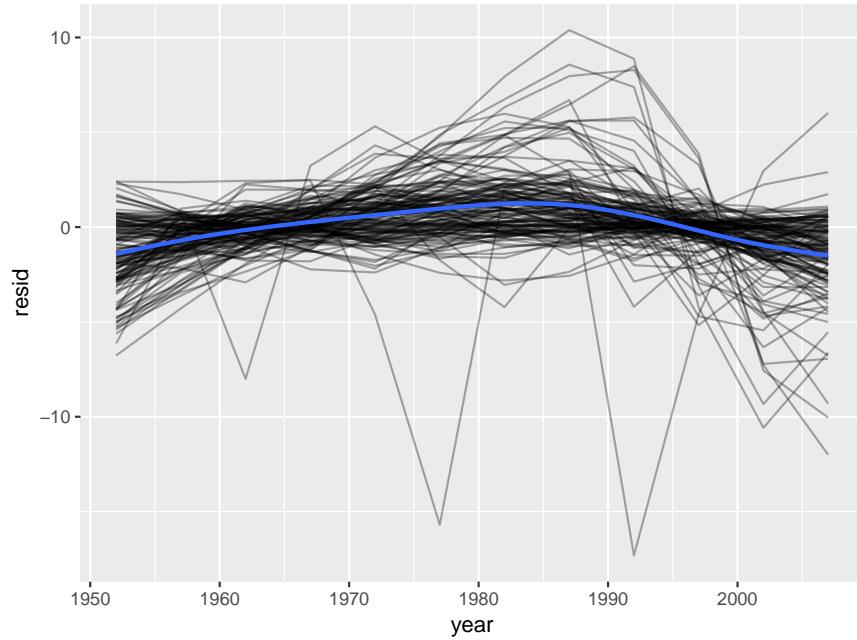
```



and we see that while the general trend is up there are actually exceptions!

Here are the residuals:

```
resids %>%
 ggplot(aes(year, resid)) +
 geom_line(aes(group=country), alpha=1/3) +
 geom_smooth(se=FALSE)
```

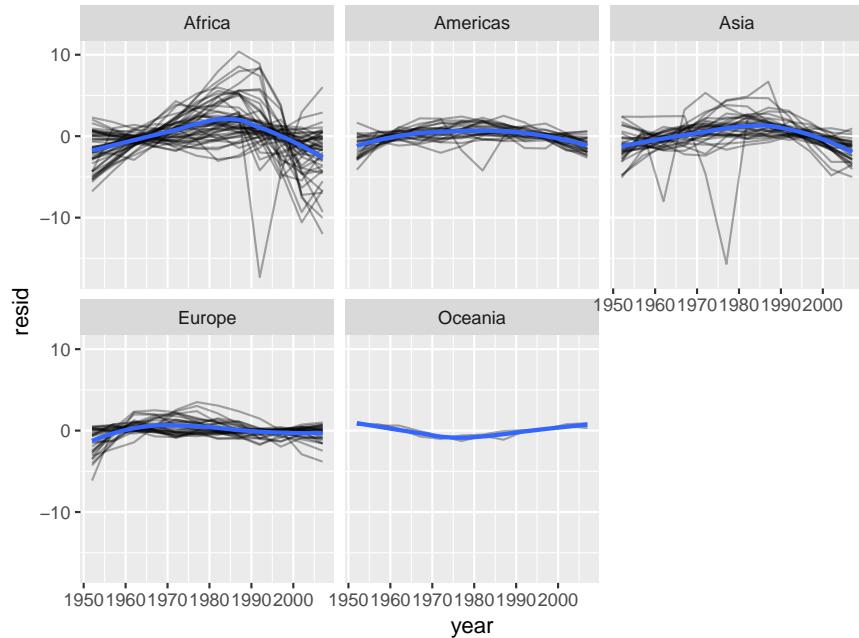


and again we see that for some country the linear model fails badly.

(Notice the use of group instead of color. Any idea why?)

Here is an interesting view:

```
resids %>%
 ggplot(aes(year, resid)) +
 geom_line(aes(group=country), alpha=1/3) +
 geom_smooth(se=FALSE) +
 facet_wrap(~continent)
```

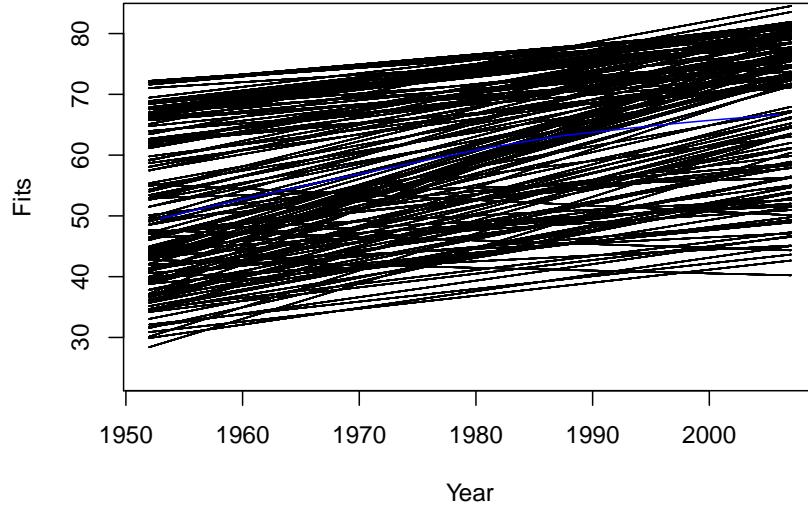


For comparison let's redo this analysis, this time using base R only.

```

df <- gapminder
n.countries <- dim(df)[1]
df$fitted <- rep(0, n.countries)
df$residuals <- rep(0, n.countries)
for(i in 1:n.countries) {
 z <- df[df$country==df$country[i],]
 fit <- lm(z$lifeExp~z$year)
 df[df$country==df$country[i], "fitted"] <- fitted.values(fit)
 df[df$country==df$country[i], "residuals"] <- residuals(fit)
}
plot(df$year, df$lifeExp, type="n", xlab="Year", ylab="Fits")
for(i in 1:n.countries) {
 z <- df[df$country==df$country[i],]
 lines(z$year, z$fitted)
}
loess.fit <- loess(df$lifeExp ~ df$year)
new.x <- seq(1953, 2006, length=100)
new.y <- predict(loess.fit, newdata=new.x)
lines(new.x, new.y, col="blue", size=5)

```



## 28 Interactive Web Applications with shiny

```
library(shiny)
```

shiny is a package that has routines to create interactive web applications that run in a browser.

As an example consider the app Taylor Polynomials.

Every app has two files:

- *ui* contains information about the layout of the page (**user interface**).
- *server* has the R routines that create the content of the page.

Let's begin by creating a page that does the following: it generates a data set with n observations from a standard normal distribution and then draws the histogram. The user can choose n:

**example1**

```
ui <- fluidPage(
 numericInput(inputId = "n", #name of variable
 label = "Sample Size", #text on page
 value = 1000), #starting value
 plotOutput("plot1")
)
server <- function(input, output) {
 output$plot1 <- renderPlot({
```

```

x <- rnorm(input$n)
bw <- diff(range(x))/50
ggplot(data.frame(x=x), aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x = "x",
 y = "Counts")
})
}
shinyApp(ui, server)

```

If you are on a computer that has R running with shiny installed you can run this and all the apps here with

```

runGitHub(repo = "Computing-with-R",
 username = "WolfgangRolle",
 subdir = "shiny/example1")

```

you can also do this: File > New File > Shiny Web App, copy-paste commands into file, click Run App

Next we will improve the app in three ways:

- nicer layout
- add the possibility to change the number of bins
- add a title

```

ui <- fluidPage(
 titlePanel("Histogram App"),
 sidebarLayout(
 sidebarPanel(#Input widgets on left side of page
 numericInput(inputId = "n",
 label = "Sample Size",
 value = 1000,
 width = "40%"), #size of box
 textInput(inputId = "nbins",
 label = "Number of bins",
 value = "50",
 width = "20%") #size of box
),
 mainPanel(
 plotOutput("plot1")
)
))
server <- function(input, output) {
 output$plot1 <- renderPlot({
 x <- rnorm(input$n)
 })
}

```

```

bw <- diff(range(x))/as.numeric(input$nbins)
ggplot(data.frame(x=x), aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x = "x",
 y = "Counts")
})
}
shinyApp(ui, server)

```

notice also that instead of a numericInput I used a textField, although nbins is meant to be a number. I often do this because I find the appearance of text inputs more pleasing. I can of course always turn it into a number with *as.numeric*.

Next we will use one of the most useful commands in shiny: *conditionalPanel*. With this we can control what happens in the app depending on the users choices.

Let's add the possibility to show a boxplot instead of the histogram. Notice that there are no bins in a boxplot, so we want to show the nbins input only when a histogram is done:

```

ui <- fluidPage(
 titlePanel("Histogram or Boxplot App"),
 sidebarLayout(
 sidebarPanel(
 numericInput("n", "Sample Size", 1000),
 selectInput("whichgraph", "What Graph?",
 choices = c("Histogram", "Boxplot"),
 selected = "Boxplot"),
 conditionalPanel(
 condition = "input.whichgraph=='Histogram'",
 textField("nbins",
 "Number of bins",
 "50",
 width = "20%")
)
),
 mainPanel(
 conditionalPanel(
 condition="input.whichgraph=='Histogram'",
 plotOutput("plot1")
),
 conditionalPanel(
 condition="input.whichgraph=='Boxplot'",
 plotOutput("plot2")
)
)
))

```

```

server <- function(input, output) {
 output$plot1 <- renderPlot({
 x <- rnorm(input$n)
 bw <- diff(range(x))/as.numeric(input$nbins)
 ggplot(data.frame(x=x), aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x = "x",
 y = "Counts")
 })
 output$plot2 <- renderPlot({
 x <- rnorm(input$n)
 df <- data.frame(x=x, z=rep(" ", length(x)))
 ggplot(df, aes(z, x)) +
 geom_boxplot() +
 xlab("")
 })
}
shinyApp(ui, server)

```

Another useful feature is the ability to create tabs. Let's say we want to show the histogram on one tab and the boxplot on another:

```

ui <- fluidPage(
 titlePanel("Histogram or Boxplot App"),
 sidebarLayout(
 sidebarPanel(
 numericInput("n", "Sample Size", 1000),
 conditionalPanel(condition="input.mytabs=='Histogram'",
 textInput("nbins", "Number of bins", "50", width = "20%"))
),
 mainPanel(
 tabsetPanel(
 tabPanel("Histogram", plotOutput("plot1")),
 tabPanel("Boxpot", plotOutput("plot2")),
 id="mytabs"
)
)
)

```

Notice again the use of conditionalPanel: now the nbins box appears when the Histogram tab is selected.

Did you notice a slight flaw in the program? When we switch from one graph to the other the data is generated a new, so the graphs are not for the same data set. Let's fix this.

Also let's say we want on another tab to show some summary statistics:

```
ui <- fluidPage(
 titlePanel("Histogram or Boxplot App"),
 sidebarLayout(
 sidebarPanel(
 numericInput("n", "Sample Size", 1000),
 conditionalPanel(condition="input.mytabs=='Histogram'",
 textInput("nbins", "Number of bins", "50", width = "20%")
)
),
 mainPanel(
 tabsetPanel(
 tabPanel("Histogram", plotOutput("plot1")),
 tabPanel("Boxpot", plotOutput("plot2")),
 tabPanel("Summary Statistics", uiOutput("text1")),
 id="mytabs"
)
)
)))
server <- function(input, output) {

 data <- reactive({
 rnorm(input$n)
 })

 summaries <- reactive({
 x <- data()
 list(n = length(x),
 xbar = round(mean(x), 3),
 shat = round(sd(x), 3))
 })

 output$plot1 <- renderPlot({
 bw <- diff(range(data()))/as.numeric(input$nbins)
 ggplot(data.frame(x=data()), aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x="x", y="Counts")
 })

 output$plot2 <- renderPlot({
 x <- data()
 df <- data.frame(x=x, z=rep(" ", length(x)))
 ggplot(df, aes(z, x)) +
 geom_boxplot() +

```

```

 xlab(""))
})

output$text1 <- renderText({
 lns <- "<table>"
 lns[2] <- paste("<tr><th>Sample Size </th><td>", summaries()[1], "</td></tr>")
 lns[3] <- paste("<tr><th>Mean </th><td>", summaries()[2], "</td></tr>")
 lns[4] <- paste("<tr><th>Standard Deviation </th><td>", summaries()
 lns[5] <- "</table>"
 lns
})
}

shinyApp(ui, server)

```

This also shows that we can use html syntax in shiny. This is not a surprise because in the end it is a web page!

shiny apps are incredible versatile, you can write apps with a lot of stuff in it. You can even make little movies:

```

ui <- fluidPage(
 titlePanel("2-D Random Walk"),
 sliderInput("step", "Step",
 min=1, max=100, value=1, step=1,
 animate=animationOptions(interval = 500, loop = FALSE)),
 plotOutput("plot", width="500", height="500")
)

server <- function(input, output) {
 data <- reactive({
 x <- cumsum(c(rep(0, 10), rnorm(1000)))
 y <- cumsum(c(rep(0, 10), rnorm(1000)))
 data.frame(x=x, y=y)
 })

 make.plot <- reactive({
 xymax <- abs(max(data()))
 ggplot(data(), aes(x, y)) +
 lims(x=1.2*c(-xymax, xymax), y=1.2*c(-xymax, xymax)) +
 labs(x="x", y="y")
 })

 output$plot <- renderPlot({
 for(i in 1:input$step) {
 plt <- make.plot() +
 geom_point(data=data()[10*i,],

```

```

 aes(x,y), size=2, color="red") +
 geom_line(data=data()[1:(10*i),], aes(x,y),
 size=0.25, color="blue", alpha=0.5)
 }
 plt
 })
}
shinyApp(ui, server)

```

## 28.1 Create/Run/Distribute Shiny Apps

There are a number of ways to create, run and distribute shiny apps:

1. Create an app:

- small apps are easiest done within RStudio, as in our example1
- for larger apps you should have two separate files called ui.R and server.R in some folder, say myapp1. For example 1 they would look like this:

### ui.R

```

shinyUI(fluidPage(
 numericInput("n", "Sample Size", 1000),
 plotOutput("plot1")
))

```

### server.R

```

shinyServer(function(input, output) {
 output$plot1 <- renderPlot({
 x <- rnorm(input$n)
 bw <- diff(range(x))/50
 ggplot(data.frame(x=x), aes(x)) +
 geom_histogram(color = "black", fill = "white", binwidth = bw) +
 labs(x="x", y="Counts")
 })
})

```

then in the console run

```
runApp("PATH/myapp1")
```

where PATH is the folder path to myapp1.

2. Distribute an app

- you can upload the folder myapp1 to github and then use

```

runGitHub("reponame", "username",
 subdir = "shiny/myapp1")

```

here I assume you have myapp1 in a folder called shiny.

We will talk about Github a lot more soon.

- you can make a zip file out of the folder myapp1, upload it to a web site and then run

```
runUrl("http://websiteurl/shiny/myapp1.zip")
```

- go to <https://www.shinyapps.io>, set up a (free) account. Then you can upload a few apps. The big advantage of this is that even people who don't have R can now run your app. There are restrictions, though, on a free account, for example no more than 10 people can run an app at the same time.

## 29 Version Control and Collaboration, Github

Notice that the title doesn't read *with Github*. This is because github is not an R package. It is much more general.

Once you start working on larger projects (like a thesis?) you quickly run into the following problem: you consider a change to the existing document but you are not sure yet. So you make a copy. Then the same thing happens again, and again ... Eventually you have 10 copies with strange names and no idea what is what. Version control is a general principle to keep track of all these changes.

It gets even more important when you begin to collaborate with others, and everyone makes changes to the same document, possibly at the same time.

There are many version control systems available. In fact Dropbox has a very rudimentary one, it keeps older versions of a file so you can restore it when needed. But one of the very best is Github, located at <https://github.com/>.

For a detailed introduction to git, github and how they work with RStudio see Happy git with R.

Unfortunately getting going with github is not a simple process. You need to install several programs. My advice is to follow the instructions on *Happy git with R precisely*.

The main idea behind github is to *branch a repo* (repository). Essentially that makes a complete copy. You can then make any changes, but without changing the *master* copy. Once you are certain your changes will stay you *commit* them back to the master. You can make such a branch of any repo that was declared public, which is most of them, even if it is not yours. Then if they make changes to the files, all you need to do is *pull* the repo and you also have the latest version!

This course (the Rmds) is available on github at <https://github.com/WolfgangRolle/Computing-with-R>.

## 29.1 Setting up a new repo

github is independent of R and/or RStudio. You can use it for any purpose, even storing your poetry. RStudio however was designed to work closely with github, and I will discuss how to use gitub in this way.

Start by going to <https://github.com> and make sure you are logged in. Of course on your first visit you have to create an account.

Click green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

Repository name: myrepo

Public

YES Initialize this repository with a README

Click big green button “Create repository.”

Copy the HTTPS clone URL to the clipboard

Now go to RStudio

start with File > New Project > Version Control and choose git.

paste the URL into the box and choose an appropriate folder.

Open a file explorer window and go to that folder. You will now find a file README.md as well as an R project file in there.

Copy any files you wish to be part of the repo into this folder.

Go to RStudio and click on Git in the upper right corner, next to Environment etc. You see all these files, check the boxes next to them under Staged.

Click on Commit, type a meassage and click Commit. A new window will pop up, when it is done click Close.

Finally click Push. RStudio will now send those files to github. When it is done click on close  
Switch to your browser, refresh, and you should see those files.

Click on an Rmd and you will see something interesting: these are quite readable. In essence you don't need to knit to html or pdf, if you use github the Rmd itself becomes a webpage.

Now whenever you make a substancial change to one of these files, repeat the Commit-Push steps to upload the file to github.

## 30 Estimation

In this chapter we will study the problem of parameter estimation. In its most general form this is as follows: we have a sample  $X_1, \dots, X_n$  from some probability density  $f(x; \theta)$ . Here

both  $x$  and  $\theta$  might be vectors. Also we will use the term density for both the discrete and the continuous case.

The problem is to find an estimate of  $\theta$  based on the data  $X_1, \dots, X_n$ , that is a function (called a *statistic*)  $T(X_1, \dots, X_n)$  such that in some sense  $T(X_1, \dots, X_n) \approx \theta$ .

Generally one also wants to have some idea of the accuracy of this estimate, that is one wants to calculate the standard error. Most commonly this is done by finding a *confidence interval*.

There are many ways to approach this problem, we will here only discuss the method of maximum likelihood. This works as follows. If the sample is independent the joint density is given by

$$f(x_1, \dots, x_n; \theta) = \prod_{i=1}^n f(x_i, \theta)$$

and the log-likelihood function is defined by

$$l(\theta) = \sum_{i=1}^n \log f(x_i, \theta)$$

the estimate of  $\theta$  is then found by maximizing the function  $l$ . Let's call this  $\hat{\theta}$ .

One major reason for the popularity of this method is the following celebrated theorem, due to Sir R.A. Fisher: under some regularity conditions

$$\sqrt{n}(\hat{\theta} - \theta) \sim N(0, \sqrt{I^{-1}})$$

where  $N(\mu, \sigma)$  is the normal distribution and  $I$  is the *Fisher Information*, given by

$$I(\theta)_{ij} = -E \left[ \frac{\partial^i \partial^j}{\partial \theta^i \partial \theta^j} \log f(x; \theta) \right]$$

and so it is very easy to find a  $(1 - \alpha)100\%$  confidence interval for (say)  $\theta_i$  as

$$\hat{\theta} \pm z_{\alpha/2} \sqrt{I_{ii}^{-1}}$$

where  $z_\alpha$  is the  $(1 - \alpha)100\%$  quantile of the standard normal distribution. In R this is found with

```
qnorm(1-0.05/2)
```

```
[1] 1.95996398454005
```

if  $\alpha = 0.05$ .

### 30.0.1 Example: Inference for mean of normal distribution

$X_1, \dots, X_n \sim N(\mu, \sigma)$ ,  $\sigma$  known.

$$\begin{aligned}
f(x; \mu) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x - \mu)^2\right\} \\
l(\mu) &= \sum \log f(x_i, \mu) = \\
&n \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2} \sum (x_i - \mu)^2 \\
\frac{d}{d\mu} l(\mu) &= \frac{1}{\sigma^2} \sum (x_i - \mu) = 0 \\
\hat{\mu} &= \frac{1}{n} \sum x_i
\end{aligned}$$

Here we have only one parameter ( $\mu$ ), so the Fisher Information is given by

$$I(\mu) = -E\left[\frac{d^2 f(x; \mu)}{d\mu^2}\right]$$

and so we find

$$\begin{aligned}
\frac{d}{d\mu} \log f(x; \mu) &= \frac{1}{\sigma^2}(x - \mu) \\
\frac{d^2}{d\mu^2} \log f(x; \mu) &= -\frac{1}{\sigma^2} \\
-E\left[\frac{d^2 f(x; \mu)}{d\mu^2}\right] &= -E\left[-\frac{1}{\sigma^2}\right] = \frac{1}{\sigma^2} \\
\sqrt{I(\mu)^{-1}} &= \sqrt{\frac{1}{1/\sigma^2}} = \sigma \\
\sqrt{n}(\hat{\mu} - \mu) &\sim N(0, \sigma) \\
\hat{\mu} &\sim N(\mu, \sigma/\sqrt{n})
\end{aligned}$$

and we find the  $(1 - \alpha)100\%$  confidence interval to be

$$\hat{\mu} \pm z_{\alpha/2}\sigma/\sqrt{n}$$

this is of course the standard answer (for known  $\sigma$ ).

### 30.0.2 Example: Beta distribution

Say we have  $X_1, \dots, X_n \sim B(\alpha, \alpha)$ .

Now

$$f(x; \alpha) = \frac{\Gamma(2\alpha)}{\Gamma(\alpha)^2} [x(1-x)]^{\alpha-1}$$

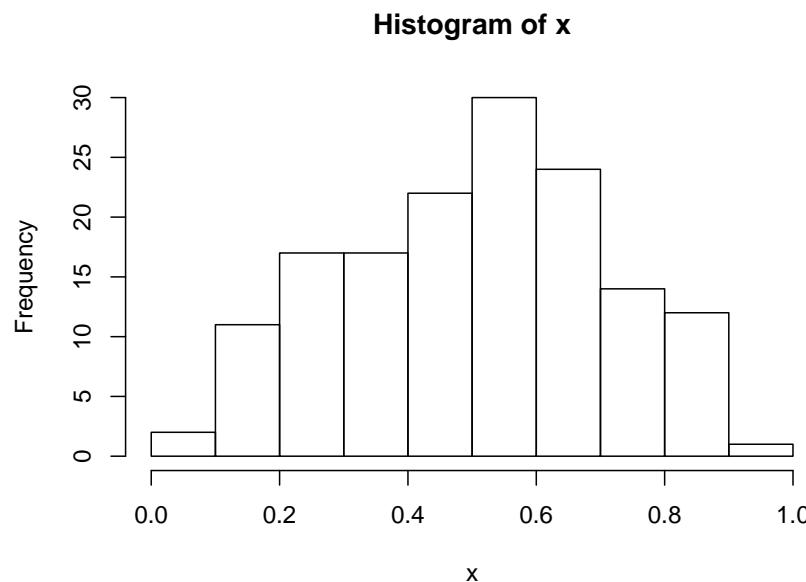
where  $\Gamma(x)$  is the gamma function, defined by

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$$

now finding the mle analytically would require us to find the derivative of  $\log \Gamma(\alpha)$ , which is impossible. We will have to do this numerically.

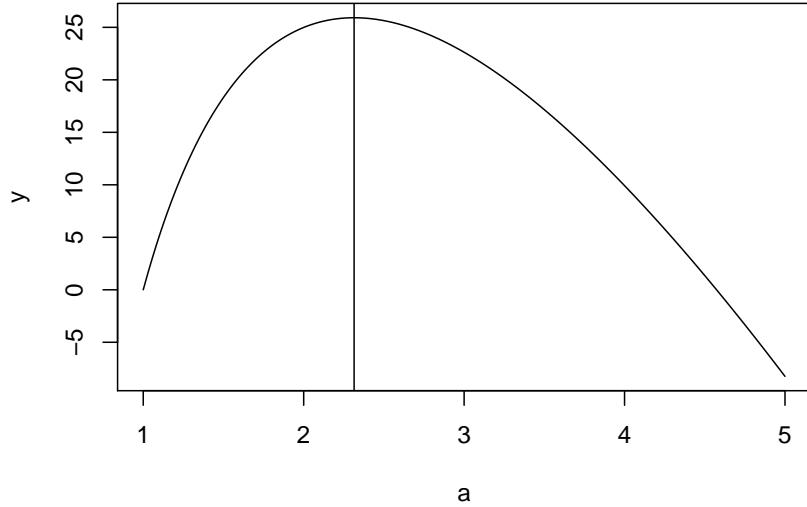
Let's start by creating an example data set:

```
set.seed(111)
n <- 150
x <- rbeta(n, 2.5, 2.5)
hist(x, 10)
```



Now

```
ll <- function(a) {
 -sum(log(dbeta(x, a, a)))
}
tmp <- nlm(ll, 2.5)
mle <- tmp$estimate
a <- seq(1, 5, length=250)
y <- rep(0, 250)
for(i in seq_along(a))
 y[i] <- sum(log(dbeta(x, a[i], a[i])))
plot(a, y, type="l")
abline(v=mle)
```



```
mle
[1] 2.31434157493634
```

How about the Fisher information? Now, we can't even find the first derivative, let alone the second one. We can however estimate it! In fact, we already have all we need.

Notice that the Fisher Information is the (negative of the) expected value of the Hessian matrix, and by the theorem of large numbers  $\frac{1}{n} \sum H \rightarrow I$ . Now if we just replace  $I$  with the *observed* information we get:

a 95% confidence interval is given by

```
hessian <- nlm(ll, 2.5, hessian = TRUE)$hessian
mle + c(-1, 1)*qnorm(1-0.05/2)/sqrt(hessian)
```

```
[1] 1.83755949579425 2.79112365407842
```

Let's put all of this together and write a "find a confidence interval" routine:

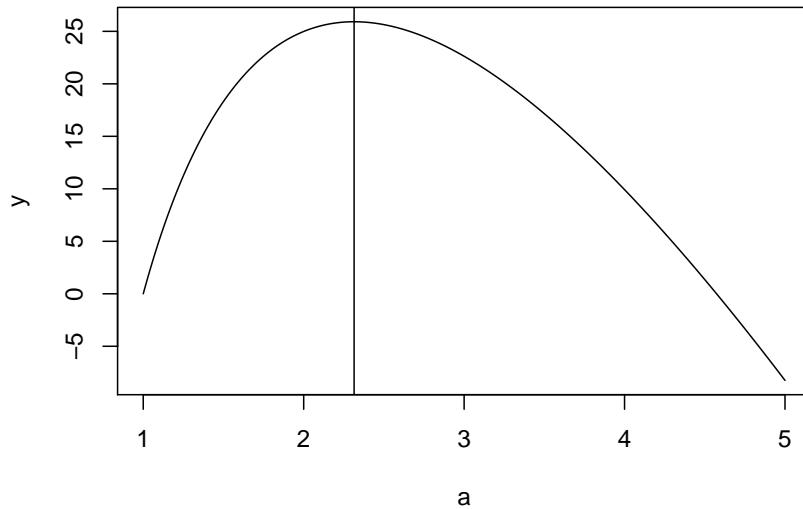
```
mle.est <-
 function(f, #density
 param, #starting value for nlm
 alpha=0.05, #desired confidence level
 rg, #range for plotting log-likelihood function
 do.graph=FALSE #TRUE if we want to look at the
 #log-likelihood function
)
{
 ll <- function(a) { #log-likelihood function
 -sum(log(f(a)))
 }
}
```

```

tmp <- nlm(ll, param, hessian = TRUE)
if(do.graph) { #if you want to see the loglikelihood curve
 a <- seq(rg[1], rg[2], length=250)
 y <- rep(0, 250)
 for(i in seq_along(a))
 y[i] <- sum(log(f(a[i])))
 plot(a, y, type="l")
 abline(v=tmp$estimate)
}
if(length(param)==1) {
 ci <- tmp$estimate + c(-1, 1) *
 qnorm(1-alpha/2)/sqrt(tmp$hessian)
 names(ci) <- c("Lower", "Upper")
}
else {
 I.inv <- solve(tmp$hessian) #find matrix inverse
 ci <- matrix(0, length(param), 2)
 colnames(ci) <- c("Lower", "Upper")
 if(!is.null(names(param)))
 rownames(ci) <- names(param)
 for(i in seq_along(param))
 ci[i,] <- tmp$estimate[i] +
 c(-1, 1)*qnorm(1-alpha/2)*sqrt(I.inv[i, i])
}
list(mle=tmp$estimate, ci=ci)
}

mle.est(f = function(a) {dbeta(x, a, a)},
 param = 2.5,
 rg = c(1, 5),
 do.graph = TRUE)

```



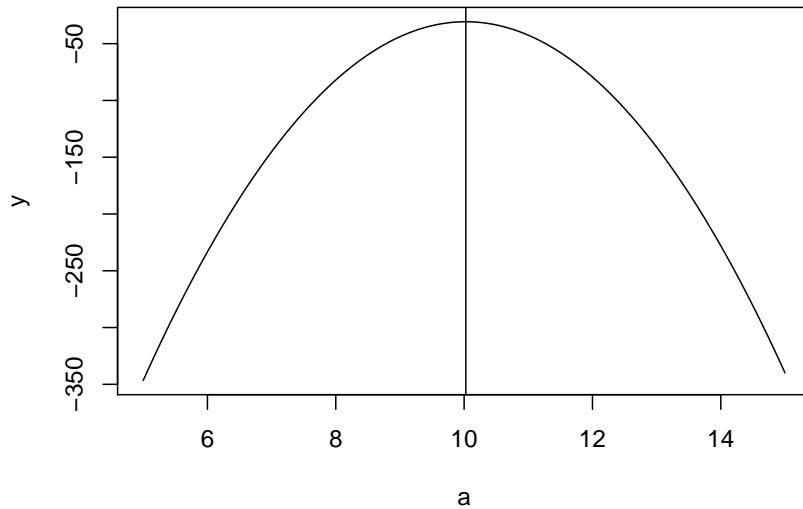
```
$mle
[1] 2.31434157493634
##
$ci
Lower Upper
1.83755949579425 2.79112365407842
```

How about the normal case, where we know the correct answer? Let's compare them:

```
x <- rnorm(25, 10, 1)
c(mean(x), mean(x) + c(-1, 1)*qnorm(1-0.05/2)/sqrt(25))

[1] 10.02657867208128 9.63458587517327 10.41857146898929

mle.est(f = function(a) {dnorm(x, a)},
 param = 10,
 rg = c(5, 15),
 do.graph = TRUE)
```



```
$mle
[1] 10.0265786720809
##
$ci
Lower Upper
9.63458587515676 10.41857146900496
```

And how about the multi dimensional parameter case? First again the normal check:

```
x <- rnorm(200, 5.5, 1.8)
param <- c(5.5, 1.8)
names(param) <- c("mu", "sigma")
mle.est(function(a) {dnorm(x, a[1], a[2])}, param=param)
```

```
$mle
[1] 5.58720720823563 1.67594313683523
##
$ci
Lower Upper
mu 5.35493760180754 5.81947681466372
sigma 1.51166266534230 1.84022360832815
```

and now for the Beta:

```
x <- rbeta(200, 2.5, 3.8)
param <- c(2.5, 3.8)
names(param) <- c("alpha", "beta")
mle.est(function(a) {dbeta(x, a[1], a[2])}, param=param)
```

```
$mle
[1] 2.34925550969544 3.73118756469436
```

```


$ci

Lower Upper

alpha 1.91332505040166 2.78518596898923

beta 3.01138553664854 4.45098959274018

```

### 30.0.3 Example: Old Faithful geyser

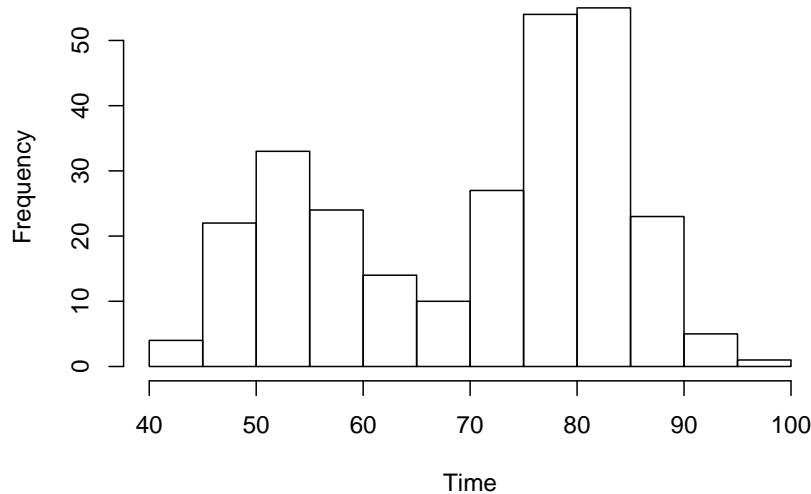
The lengths of an eruption and the waiting time until the next eruption of the Old Faithful geyser in Yellowstone National Park have been studied many times. Let's focus on the waiting times:

```

Time <- faithful$Waiting.Time

hist(Time, main="")

```



How can we model this data? It seems we might have a mixture of two normal distributions. Notice

```

c(mean(Time[Time<65]), mean(Time[Time>65]))

[1] 54.0531914893617 80.0457142857143

c(sd(Time[Time<65]), sd(Time[Time>65]))

[1] 5.36489389261955 5.86712753824920

sum(Time<65)/length(Time)

[1] 0.345588235294118

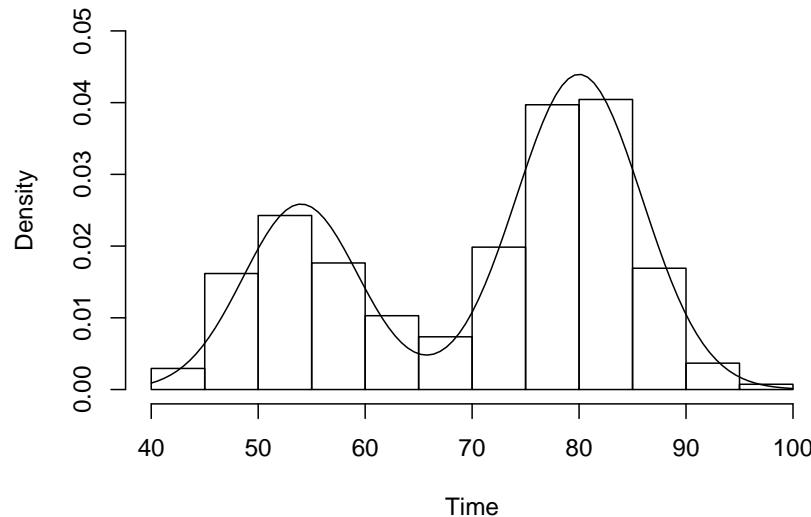
```

so maybe a model like this would work:

$$0.35N(54, 5.4) + 0.65N(80, 5.9)$$

Let's see:

```
hist(Time, main="", freq=FALSE, ylim=c(0, 0.05))
curve(0.35*dnorm(x, 54, 5.4) + 0.65*dnorm(x, 80, 5.9), 40, 100, add=TRUE)
```



Not too bad!

Can we do better? How about fitting for the parameters?

```
x <- Time
f <- function(a)
 a[1]*dnorm(x, a[2], a[3]) + (1-a[1])*dnorm(x, a[4], a[5])
res <- mle.est(f, param=c(0.35, 54, 5.4, 80, 5.9))
res

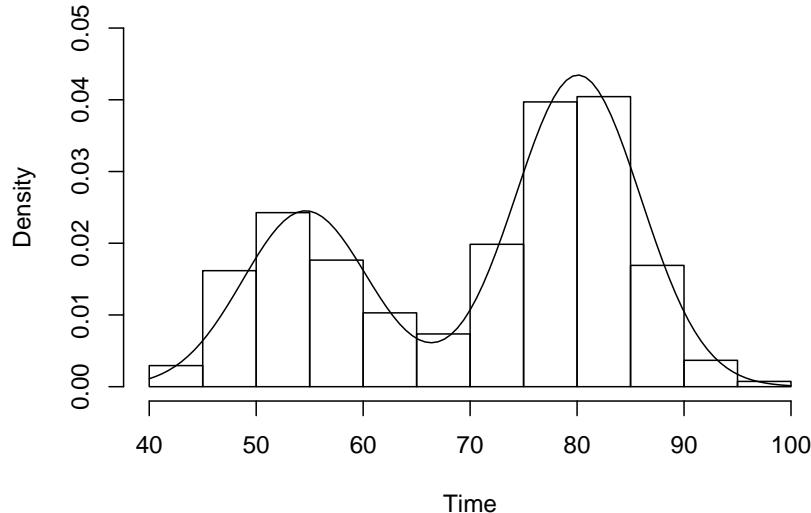
$mle
[1] 0.360886018932726 54.614853664490980 5.871217613510367
[4] 80.091066914803463 5.867735284844155
##
$ci
Lower Upper
[1,] 0.299796805814694 0.421975232050759
[2,] 53.243311294770869 55.986396034211090
[3,] 4.817584313993859 6.924850913026875
[4,] 79.102358089213851 81.079775740393075
[5,] 5.081884029154290 6.653586540534020
```

and this looks like

```

hist(Time, main="", freq=FALSE, ylim=c(0, 0.05))
curve(res$mle[1] * dnorm(x, res$mle[2], res$mle[3]) +
 (1-res$mle[1])*dnorm(x, res$mle[4], res$mle[5]), 40, 100,
 add=TRUE)

```



Now this sounds good, and it is, however this is based on having a *large enough* sample. In order to be sure ours is large enough one usually has to do some kind of coverage study.

#### 30.0.4 Hurricane Maria

How many people died due to Hurricane Maria when it struck Puerto Rico on September 20, 2017? Dr Roberto Rivera and I tried to answer this question. We got the following information from the Department of Health: during the time period September 1st to September 19 there were 1582 deaths. During the period September 20 to October 31 there were 4319.

Now this means that during the time before the hurricane roughly  $1587/19 = 83.5$  people died per day whereas in the 42 days after the storm it was  $4319/42 = 102.8$ , or  $102.8 - 83.5 = 19.3$  more per day. This would mean a total of  $42 \times 19.3 = 810.6$  deaths caused by Maria in this time period.

Can we find a 95% confidence interval? To start, the number of people who die on any one day is a Binomial random variable with  $n=3500000$  (the population of Puerto Rico) and success(!!) parameter  $\pi$ . Apparently before the storm we had  $\pi = 83.5/3500000$ . If we denote the probability to die due to Maria by  $\mu$ , we find the probability model

$$f(x, y) = \text{dbinom}(1587, 19 \times 3500000, \pi) \text{dbinom}(4319, 42 \times 3500000, \pi + \mu)$$

Let's see:

```

N <- 3500000
f <- function(a) -log(dbinom(1582, 19*N, a[1])) -
 log(dbinom(4319, 42*N, a[1]+a[2]))
nlm(f, c(1582/19/3500000, (4319/42-1582/19)/3350000), hessian = TRUE)

$minimum
[1] 9.8624618888344
##
$estimate
[1] 2.37894736842105e-05 5.84184341450642e-06
##
$gradient
[1] 0 0
##
$hessian
[,1] [,2]
[1,] -Inf -Inf
[2,] -Inf -Inf
##
$code
[1] 1
##
$iterations
[1] 1

```

Ooops, that didn't work. The problem is that the numbers for calculating the Hessian matrix become so small that it can not be done.

What to do? First we can try to use the usual Poisson approximation to the Binomial:

```

f <- function(a)
 -log(dpois(1582, 19*a[1])) - log(dpois(4319, 42*(a[1]+a[2])))
res <- nlm(f, c(80, 20), hessian = TRUE)
res

$minimum
[1] 9.70656119758024
##
$estimate
[1] 83.2631585522970 19.5701747934333
##
$gradient
[1] -3.84016456559495e-12 -7.26148584016244e-12
##
$hessian
[,1] [,2]
[1,] 0.636508273851911 0.408387037811490
[2,] 0.408387037811490 0.408412332540153

```

```

$code
[1] 1

$iterations
[1] 10
```

and now

```
round(42*(res$estimate[2] +
c(-1, 1)*qnorm(1-0.05/2)*sqrt(solve(res$hessian)[2, 2])))
```

```
[1] 607 1037
```

An even better solution is to do a bit of math:

$$\begin{aligned}\log \{dpois(x, \lambda)\} &= \\ \log \left\{ \frac{\lambda^x}{x!} e^{-\lambda} \right\} &= \\ x \log(\lambda) - \log(x!) - \lambda &= \end{aligned}$$

```
f <- function(a)
-1582*log(19*a[1]) + 19*a[1] -
4319*log(42*(a[1]+a[2])) + 42*(a[1]+a[2])
res <- nlm(f, c(20, 80), hessian = TRUE)
round(42*(res$estimate[2] +
c(-1, 1)*qnorm(1-0.05/2)*sqrt(solve(res$hessian)[2, 2])))
```

```
[1] 607 1037
```

By the way, in the paper we used a somewhat different solution based on the *profile likelihood*. In this case the answers are quite similar.

The paper is here

UPDATE: After a long legal fight the Department of Health on June 1st 2018 finally updated the numbers:

Notice how in general the number of deaths is much higher in the winter than in the summer. So it may be best to just use the data from February to November:

```
deaths.before <- 2315+2494+2392+2390+2369+2367+2321+2928-1317
deaths.after <- 1317+3040+2671
deaths.before/231 #Daily Deaths before Maria

[1] 79.04329004329
deaths.after/72 #Daily Deaths after Maria

[1] 97.6111111111111
```

Total de Defunciones por Mes

Mes	2015	2016	2017	2018
<b>Jan</b>	2744	2742	2894	2821
<b>Feb</b>	2403	2592	2315	2448
<b>Mar</b>	2427	2458	2494	2643
<b>Apr</b>	2259	2241	2392	2218
<b>May</b>	2340	2312	2390	1892
<b>Jun</b>	2145	2355	2369	0
<b>Jul</b>	2382	2456	2367	0
<b>Aug</b>	2272	2427	2321	0
<b>Sep</b>	2258	2367	2928	0
<b>Oct</b>	2393	2357	3040	0
<b>Nov</b>	2268	2484	2671	0
<b>Dec</b>	2516	2854	2820	0
<b>Total</b>	<b>28407</b>	<b>29645</b>	<b>31001</b>	<b>12022</b>

Figure 2:

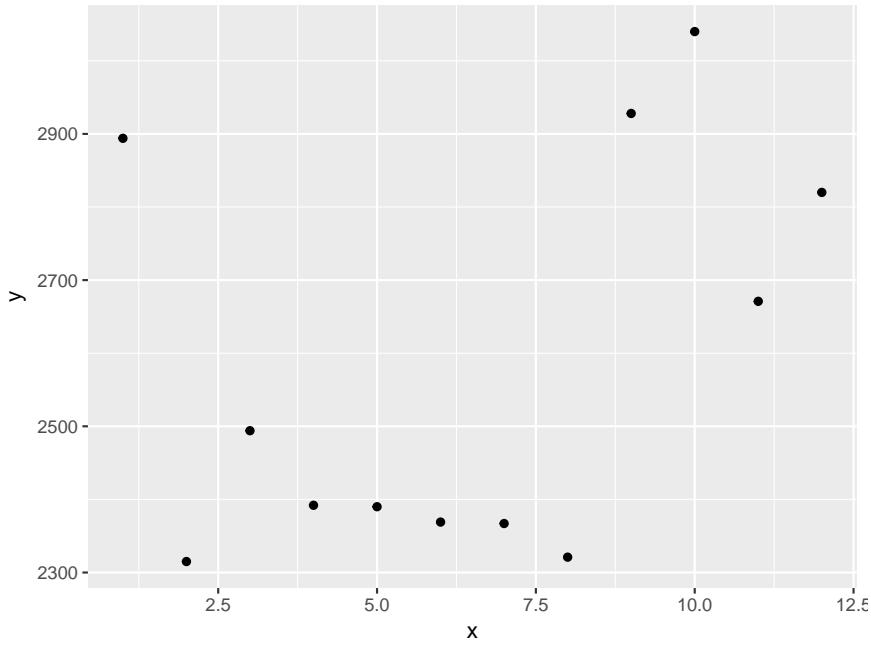
```

round(72*(deaths.after/72 - deaths.before/231)) #point estimate for total deaths due to
[1] 1337

f <- function(a)
 -deaths.before*log(231*a[1]) + 231*a[1] -
 deaths.after*log(72*(a[1]+a[2])) + 72*(a[1]+a[2])
res <- nlm(f, c(20, 80), hessian = TRUE)
round(72*(res$estimate[2] +
 c(-1, 1)*qnorm(1-0.05/2)*sqrt(solve(res$hessian)[2, 2])))
[1] 1153 1521

Months <- factor(unique(draft$Month), ordered=TRUE)
Deaths <- c(2894, 2315, 2494, 2392, 2390, 2369, 2367,
 2321, 2928, 3040, 2671, 2820)
ggplot(data=data.frame(x=1:12, y=Deaths), aes(x, y)) +
 geom_point()

```



### 30.1 Packages

There are a number of packages available for maximum likelihood fitting:

```
library(maxLik)
x <- c(1582, 4319)
f <- function(param) {
 x[1]*log(19*param[1]) - 19*param[1] +
 x[2]*log(42*(param[1]+param[2])) - 42*(param[1]+param[2])
}
maxLik(logLik=f, start=c(20, 80))

Maximum Likelihood estimation
Newton-Raphson maximisation, 21 iterations
Return code 2: successive function values within tolerance limit
Log-Likelihood: 41906.1106539303 (2 free parameter(s))
Estimate(s): 83.1785379497624 19.6523730145212
```

In general these just provide wrappers for the routines mentioned above.

## 31 The Bootstrap

The idea of the Bootstrap is rather strange: say we have some data from some distribution and we want to use it to estimate some parameter  $\theta$ . We have a formula (a *statistic*)  $T(x_1, \dots, x_n)$ . What is the standard error in this estimate? That is, what is  $sd[T(X_1, \dots, X_n)]$ ?

Sometimes we can do this mathematically: Let's assume that the  $X_i$  are *iid* (independent

and identically distributed) and we are interested in the mean. Let's write  $\mathbf{X} = (X_1, \dots, X_n)$ , then

$$\begin{aligned}
\theta &= E[X] \\
T\mathbf{X} &= \frac{1}{n} \sum_{i=1}^n x_i \\
E[T\mathbf{X}] &= E\left[\frac{1}{n} \sum_{i=1}^n X_i\right] = \frac{1}{n} \sum_{i=1}^n E[X_i] = \frac{1}{n} n\theta = \theta \\
Var[T\mathbf{X}] &= E\left[\left(\frac{1}{n} \sum_{i=1}^n X_i - \theta\right)^2\right] = \\
&\frac{1}{n^2} E\left[\left(\sum_{i=1}^n X_i - n\theta\right)^2\right] = \\
&\frac{1}{n^2} E\left[\left(\sum_{i=1}^n (X_i - \theta)\right)^2\right] = \\
&\frac{1}{n^2} E\left[\sum_{i,j=1}^n (X_i - \theta)(X_j - \theta)\right] = \\
&\frac{1}{n^2} \left[ \sum_{i=1}^n E(X_i - \theta)^2 + \sum_{i,j=1, i \neq j}^n E(X_i - \theta)(X_j - \theta) \right] = \\
&\frac{1}{n^2} [nE(X_1 - \theta)^2 + 0] = \frac{1}{n} Var[X_1]
\end{aligned}$$

because  $E(X_i - \theta)^2 = E(X_1 - \theta)^2$  (identically distributed) and  $E(X_i - \theta)(X_j - \theta) = E(X_i - \theta)E(X_j - \theta) = 0$  because of independence.

But let's say that instead of the mean we want to estimate  $\theta$  with the median. Now what is the  $sd[\text{median}(x_1, \dots, x_n)]$ ? This can still be done analytically, but is already much more complicated.

It would of course be easy if we could simulate from the distribution:

```

sim.theta <- function(B=1e4, n, mu=0, sig=1) {
 x <- matrix(rnorm(B*n, mu, sig), B, n)
 xbar <- apply(x, 1, mean)
 med <- apply(x, 1, median)
 round(c(sig/sqrt(n), sd(xbar), sd(med)), 3)
}

sim.theta(n=25)

[1] 0.200 0.198 0.251

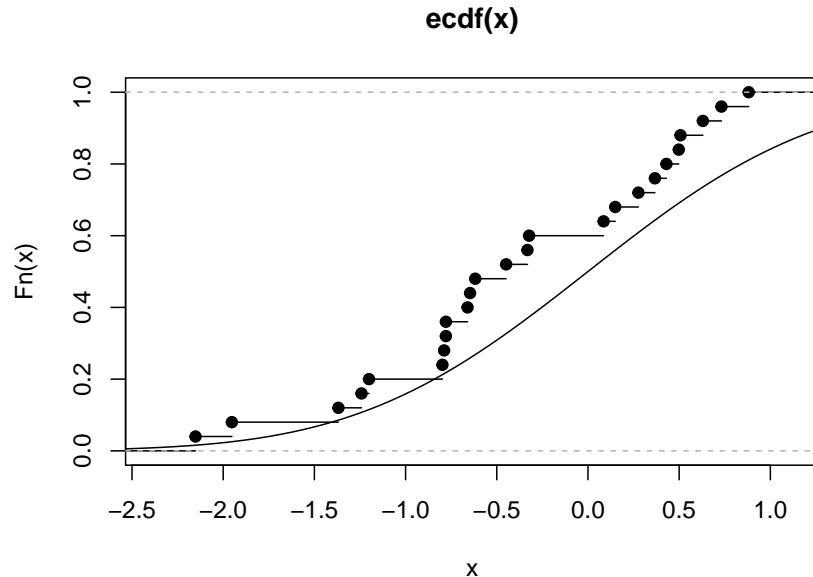
```

But what do we do if didn't know that the data comes from the normal distribution? Then we can't simulate from  $F$ . We can, however simulate from the next best thing, namely the empirical distribution function  $edf \hat{F}$ . This is defined as:

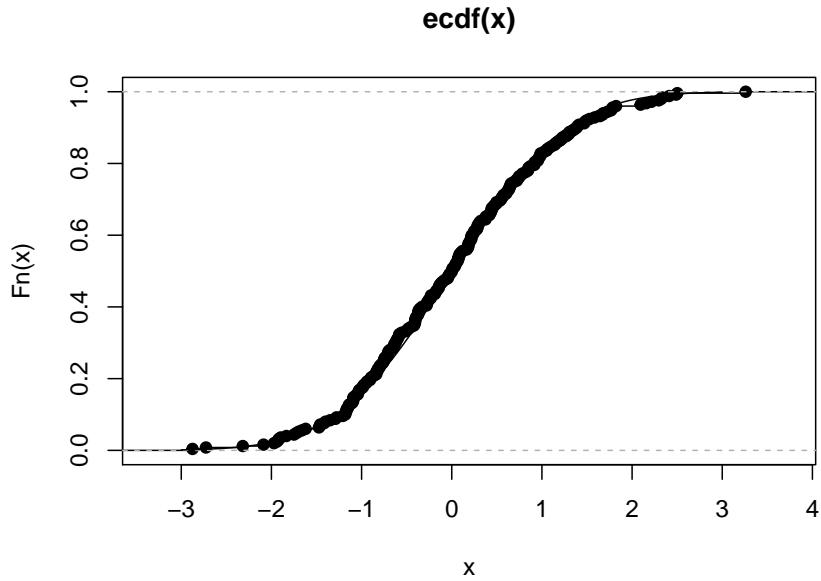
$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^n I_{(-\infty, x]}(X_i) = \frac{\#X_i \leq x}{n}$$

Here are two examples:

```
x <- rnorm(25)
plot(ecdf(x))
curve(pnorm(x), -3, 3, add=T)
```



```
x <- rnorm(250)
plot(ecdf(x))
curve(pnorm(x), -3, 3, add = TRUE)
```



There is a famous theorem in probability theory (Glivenko-Cantelli) that says that the empirical distribution function converges to the true distribution function uniformly.

How does one simulate from  $\hat{F}$ ? It means to *resample* from the data, that is randomly select numbers from  $x$  *with replacement* such that each observation has an equal chance of getting picked:

```
x <- sort(round(rnorm(10, 10, 3), 1))
x

[1] 4.5 7.9 11.2 11.3 11.6 11.7 11.7 12.1 15.3 17.6
sort(sample(x, size=10, replace=T))

[1] 4.5 4.5 4.5 7.9 11.2 11.6 11.7 11.7 15.3 17.6
sort(sample(x, size=10, replace=T))

[1] 4.5 7.9 11.3 11.6 11.7 11.7 12.1 12.1 15.3 17.6
sort(sample(x, size=10, replace=T))

[1] 7.9 7.9 11.2 11.3 11.6 11.7 11.7 15.3 15.3 17.6
```

Now the Bootstrap estimate of standard error is simply the sample standard deviation of the estimates of  $B$  such bootstrap samples:

```
x <- rnorm(25, 0, 1)
B <- 500
z <- matrix(0, B, 2)
for(i in 1:B) {
 z[i, 1] <- mean(sample(x, size=25, replace=T))
 z[i, 2] <- median(sample(x, size=25, replace=T))
```

```

}
round(c(1/sqrt(25), apply(z, 2, sd)), 3)

```

```
[1] 0.200 0.220 0.331
```

There is also a package that we can use:

```

library(bootstrap)
sd(bootstrap(x, 500, mean)$thetastar)

```

```
[1] 0.20210474639317
```

```
sd(bootstrap(x, 500, median)$thetastar)
```

```
[1] 0.327668224726312
```

### 31.0.1 Example

the *skewness* of a distribution is a measure of it's lack of symmetry. It is defined by

$$\gamma_1 = E \left[ \left( \frac{X - \mu}{\sigma} \right)^3 \right]$$

a standard estimator of  $\gamma_1$  is

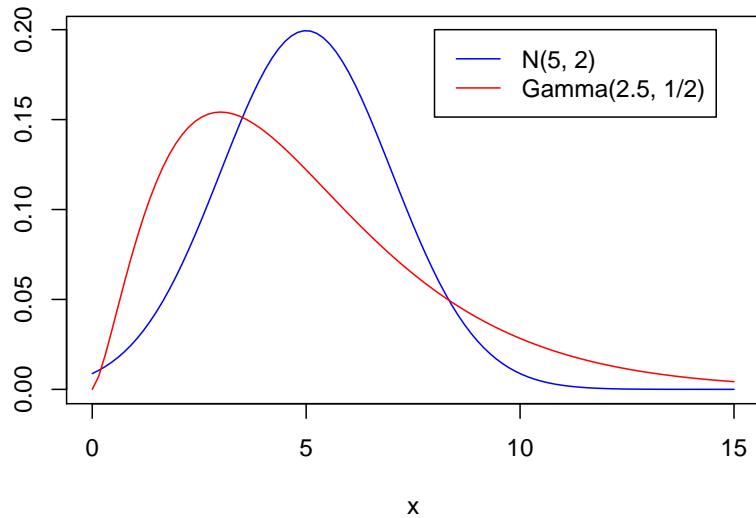
$$\hat{\gamma}_1 = \frac{\frac{1}{n} \sum (x_i - \bar{x})^3}{[\frac{1}{n-1} \sum (x_i - \bar{x})^2]^{3/2}} = \frac{\frac{1}{n} \sum (x_i - \bar{x})^3}{[\text{sd}(x)]^{3/2}}$$

What is the standard error in this estimate? Doing this analytically would be quite an exercise, but:

```

curve(dnorm(x, 5, 2), 0, 15, col="blue", ylab="")
legend(8, 0.2, c("N(5, 2)", "Gamma(2.5, 1/2)",
 lty=c(1, 1), col=c("blue", "red"))
curve(dgamma(x, 2.5, 1/2), 0, 15, add=TRUE, col="red")

```



```

T.fun <- function(x) mean((x-mean(x))^3)/sd(x)^(3/2)
x <- rnorm(250, 5, 2)
sd(bootstrap(x, 500, T.fun)$thetastar)

[1] 0.323731221295088

x <- rgamma(250, 2.5, 1/2)
sd(bootstrap(x, 500, T.fun)$thetastar)

[1] 1.26898969532169

```

## 31.1 Confidence Intervals

There are two standard techniques for using the Bootstrap to find confidence intervals:

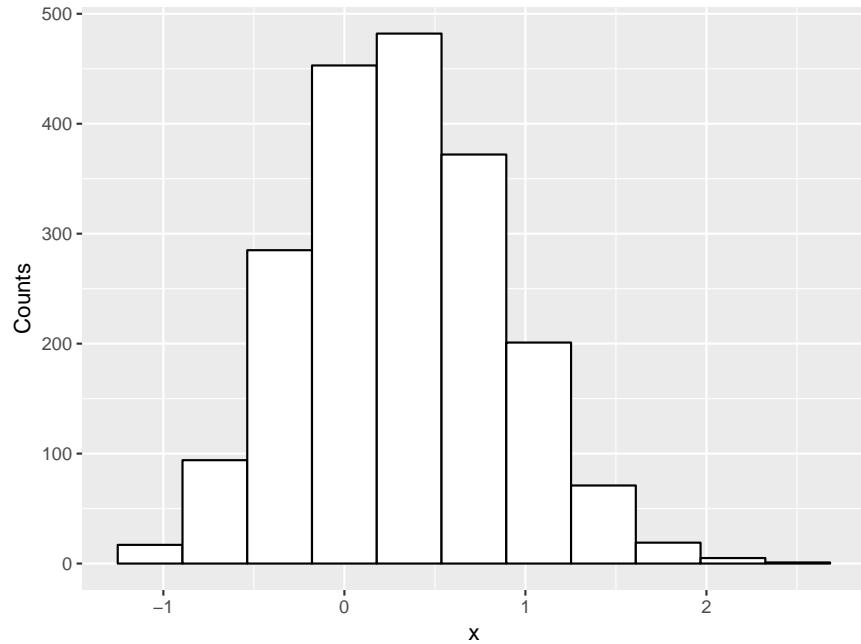
### 31.1.1 Normal Theory

Let's continue the discussion of the skewness, and put a 95% confidence interval on the estimates:

```

x.normal <- rnorm(250, 5, 2)
T.fun <- function(x) mean((x-mean(x))^3)/sd(x)^(3/2)
thetastar.normal <- bootstrap(x.normal, 2000, T.fun)$thetastar
df <- data.frame(x = thetastar.normal)
bw <- diff(range(x))/50
ggplot(df, aes(x)) +
 geom_histogram(color = "black", fill = "white", binwidth = bw) +
 labs(x="x", y="Counts")

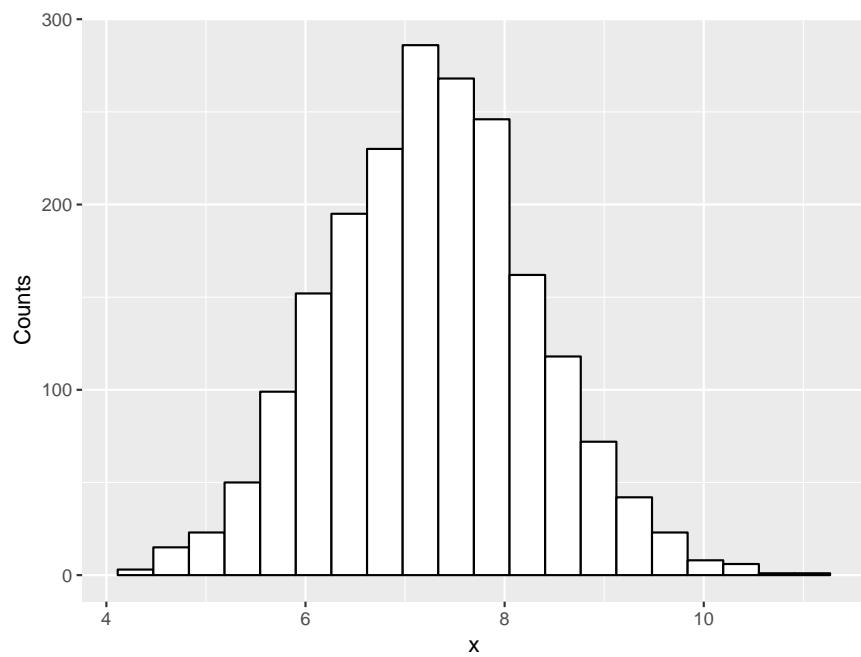
```



```

x.gamma <- rgamma(250, 2.5, 1/2)
thetastar.gamma <- bootstrap(x.gamma, 2000, T.fun)$thetastar
df <- data.frame(x = thetastar.gamma)
bw <- diff(range(x))/50
ggplot(df, aes(x)) +
 geom_histogram(color = "black",
 fill = "white",
 binwidth = bw) +
 labs(x="x", y="Counts")

```



Note that I increased the number of Bootstrap samples to 2000, which is standard when calculating confidence intervals.

We can see that the bootstrap estimates are reasonably normally distributed, so we can find the confidence interval with

```
T.fun(x.normal) + c(-1, 1)*qnorm(0.975)*sd(thetastar.normal)

[1] -0.773590347812249 1.389957432317396

T.fun(x.gamma) + c(-1, 1)*qnorm(0.975)*sd(thetastar.gamma)

[1] 5.38963829661169 9.45082989042207
```

Notice that here there is no  $/\sqrt{n}$ , because  $sd(\theta_{\text{star}})$  is already the standard deviation of the estimator, not of an individual observation.

### 31.1.2 Percentile Intervals

An alternative way to find confidence intervals is by estimating the population quantiles of the bootstrap sample with the sample quantiles:

```
sort(thetastar.normal)[2000*c(0.025, 0.975)]

[1] -0.665956368140892 1.426486296370276

sort(thetastar.gamma)[2000*c(0.025, 0.975)]

[1] 5.29729526958885 9.33286146388310
```

### 31.1.3 More Advanced Intervals

There are a number of ways to improve the performance of bootstrap based confidence intervals. One of the more popular ones is called *nonparametric bias-corrected and accelerated (BCa) intervals*. The package *bootstrap* has the routine *bcanon*. The intervals are found via the percentile method but the percentile are found with

$$\begin{aligned}\alpha_1 &= \Phi \left( \widehat{z}_0 + \frac{\widehat{z}_0 + z_\alpha}{1 - \hat{a}(\widehat{z}_0 + z_\alpha)} \right) \\ \alpha_2 &= \Phi \left( \widehat{z}_0 + \frac{\widehat{z}_0 + z_{1-\alpha}}{1 - \hat{a}(\widehat{z}_0 + z_{1-\alpha})} \right)\end{aligned}$$

here

- $\Phi$  is the standard normal cdf
- $\alpha$  is the desired confidence level
- $\widehat{z}_0$  is a bias-correction factor

- $\hat{a}$  is called the acceleration

```
bcanon(x.normal, 2000, T.fun, alpha=c(0.025, 0.975))$conf
```

```
alpha bca point
[1,] 0.025 -0.564418428578087
[2,] 0.975 1.758385638686377
```

```
bcanon(x.gamma, 2000, T.fun, alpha=c(0.025, 0.975))$conf
```

```
alpha bca point
[1,] 0.025 5.71849337200989
[2,] 0.975 9.89511910399530
```

## 32 Curve Fitting

### 32.1 Parametric Models

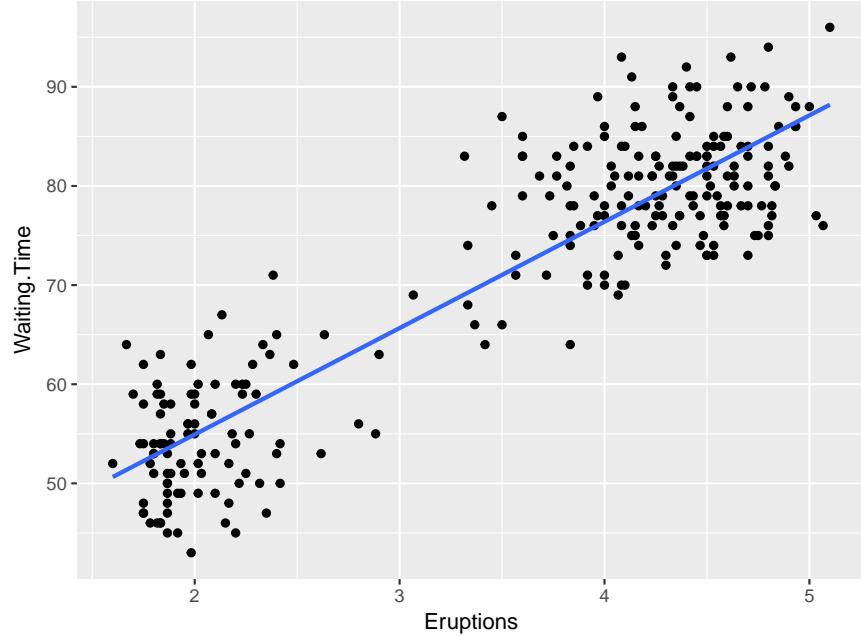
Let's have another look at the Old Faithful data. Now we will consider both variables, the length of the eruptions and the waiting time until the next one.

```
head(faithful)
```

```
Eruptions Waiting.Time
1 3.600 79
2 1.800 54
3 3.333 74
4 2.283 62
5 4.533 85
6 2.883 55
```

Both variables are continuous and we are interested in their relationship, so we start with the scatterplot:

```
ggplot(data=faithful, aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_smooth(method="lm", se=FALSE)
```



Here we have added the *least squares regression line*, which is found by minimizing the least squares criterion

$$LS(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2$$

using R we find

```
fit.lm <- lm(Waiting.Time ~ Eruptions, data=faithful)
summary(fit.lm)

##
Call:
lm(formula = Waiting.Time ~ Eruptions, data = faithful)
##
Residuals:
Min 1Q Median 3Q
-12.079608092047 -4.483102547619 0.212248466006 3.924627047929
Max
15.971858094279
##
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 33.474397022754 1.154873514655 28.98534 < 2.22e-16
Eruptions 10.729641395134 0.314753405846 34.08904 < 2.22e-16
##
Residual standard error: 5.91400943342 on 270 degrees of freedom
Multiple R-squared: 0.811460760973, Adjusted R-squared: 0.810762467495
F-statistic: 1162.06263796 on 1 and 270 DF, p-value: < 2.220446049e-16
```

and so the best fit line is given by

$$\text{Waiting Time} = 33.5 + 10.7 \text{ Eruption}$$

Let's say we want to estimate the waiting time until the next eruption if the last one lasted 2.5 minutes. Actually we want to find a 95% confidence interval:

```
predict(fit.lm,
 newdata = list(Eruptions=2.5),
 interval = "prediction")

fit lwr upr
1 60.2985005105873 48.617630637816 71.9793703833585
```

the argument interval = “prediction” indicates that we want to predict an individual response, as opposed to the mean response, which would be with interval = “confidence”.

Let's do this again, but this time with the bootstrap:

```
library(bootstrap)
fun <- function(x) {
 fit <- lm(Waiting.Time ~ Eruptions, data=faithful[x,])
 predict(fit, newdata=list(Eruptions=2.5))
}
int <- bcanon(x = 1:dim(faithful)[1],
 n = 2000,
 theta = fun,
 alpha = c(0.025, 0.975))$conf
int

alpha bca point
[1,] 0.025 59.4121195076264
[2,] 0.975 61.2448834555296
```

but this is quite different from the interval above!

Notice that this interval is much shorter ( $61.2 - 59.4 = 1.8$  vs  $72.0 - 48.6 = 13.4$ ), so maybe these are the *confidence* intervals (aka for the mean response)? Let's see:

```
predict(fit.lm,
 newdata = list(Eruptions=2.5),
 interval = "confidence")

fit lwr upr
1 60.2985005105873 59.3641026499698 61.2328983712047
```

that seems to be the case!

How can we get the prediction interval using the bootstrap? We will need to do a bit of math. First here are the formulas for the standard errors:

$$se_c = \hat{\sigma} \sqrt{\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}$$

$$se_p = \hat{\sigma} \sqrt{1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}$$

so we see they look quite similar, except for the  $1+$  term in  $se_p$ . So let's try to use one to solve for the other:

$$\frac{se_c^2}{\hat{\sigma}^2} = \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}$$

$$1 + \frac{se_c^2}{\hat{\sigma}^2} = 1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}$$

$$se_c^2 + \hat{\sigma}^2 = \hat{\sigma}^2 \left( 1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right)$$

$$se_p = \hat{\sigma} \sqrt{1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}} = \sqrt{se_c^2 + \hat{\sigma}^2}$$

but what is  $\hat{\sigma}$ ? It is the standard deviation of the residuals, so

```
se_c <- diff(int[, 2])/qnorm(0.975)/2
sigma <- sd(fit.lm$residuals)
se_p <- sqrt(se_c^2 + sigma^2)
predict(fit.lm, newdata=list(Eruptions=2.5)) +
 c(-1, 1)*qnorm(0.975)*se_p
```

```
[1] 48.6924268492098 71.9045741719648
```

and that fits well with the result above.

So far we have fit a linear model. Is there something to be gained by fitting a higher order polynomial?

Doing so is easy, for the quadratic model we can just run

```
fit.quad <- lm(Waiting.Time ~ Eruptions + I(Eruptions^2),
 data = faithful)
summary(fit.quad)

##
Call:
lm(formula = Waiting.Time ~ Eruptions + I(Eruptions^2), data = faithful)
##
Residuals:
Min 1Q Median 3Q
-12.647076932716 -4.090420923394 0.285504708309 3.520364380888
Max
14.678734527617
##
```

```

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 17.200021274124 4.495761406223 3.82583 0.00016209
Eruptions 22.213151226457 3.086160376183 7.19767 6.1183e-12
I(Eruptions^2) -1.766201851260 0.472300444581 -3.73957 0.00022525

Residual standard error: 5.77673764873 on 269 degrees of freedom
Multiple R-squared: 0.820777913631, Adjusted R-squared: 0.819445407412
F-statistic: 615.965540964 on 2 and 269 DF, p-value: < 2.220446049e-16

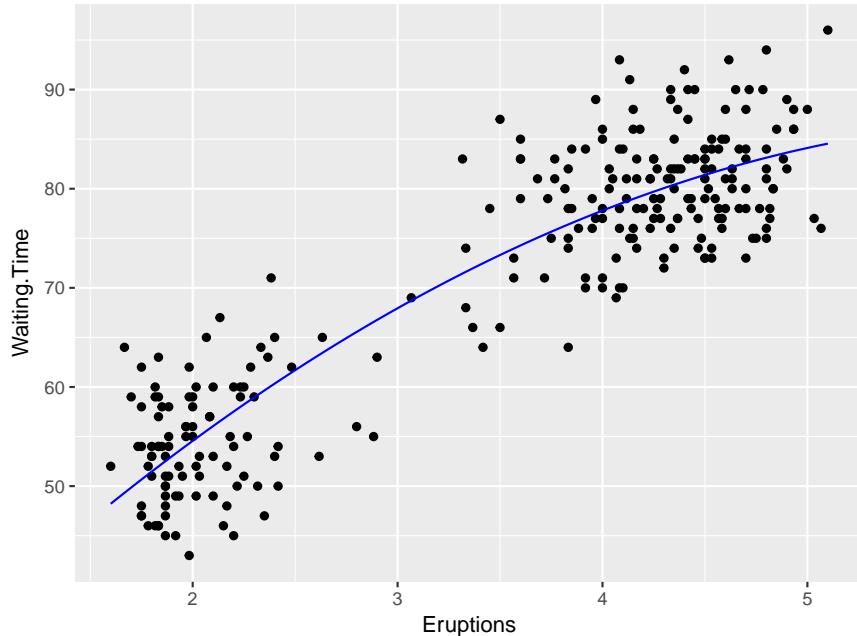
```

which looks like this:

```

x <- seq(from = min(faithful$Eruptions),
 to = max(faithful$Eruptions),
 length = 100)
y <- coef(fit.quad)[1] +
 coef(fit.quad)[2]*x +
 coef(fit.quad)[3]*x^2
ggplot(data = faithful,
 aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_line(aes(x,y),
 data = data.frame(x=x, y=y),
 color = "blue")

```



Is this a better model than the linear one? We can actually test for this:

```
anova(fit.quad, fit.lm)
```

```
Analysis of Variance Table
```

```


Model 1: Waiting.Time ~ Eruptions + I(Eruptions^2)
Model 2: Waiting.Time ~ Eruptions
Res.Df RSS Df Sum of Sq F Pr(>F)
1 269 8976.717724940
2 270 9443.387046217 -1 -466.6693212767 13.9844 0.00022525

```

This does the so called F test. It tests the null hypothesis of no difference between the models. The p value is 0.00022, so we would reject the null and conclude that the quadratic model is better.

But why stop there? But before we go on we should make one change: our largest x value is about 5,  $x^2 = 25$ ,  $x^3 = 125$  etc. These numbers keep going quite rapidly. Standardizing them should help. In fact, R has a nice routine for us

```

fit.quad <- lm(Waiting.Time ~ poly(Eruptions, 2), data=faithful)
fit.cube <- lm(Waiting.Time ~ poly(Eruptions, 3), data=faithful)
anova(fit.cube, fit.quad)

```

```

Analysis of Variance Table
##
Model 1: Waiting.Time ~ poly(Eruptions, 3)
Model 2: Waiting.Time ~ poly(Eruptions, 2)
Res.Df RSS Df Sum of Sq F Pr(>F)
1 268 8656.627086468
2 269 8976.717724940 -1 -320.0906384724 9.90967 0.0018299

fit.4 <- lm(Waiting.Time ~ poly(Eruptions, 4), data=faithful)
anova(fit.4, fit.cube)

```

```

Analysis of Variance Table
##
Model 1: Waiting.Time ~ poly(Eruptions, 4)
Model 2: Waiting.Time ~ poly(Eruptions, 3)
Res.Df RSS Df Sum of Sq F Pr(>F)
1 267 8413.382397482
2 268 8656.627086468 -1 -243.2446889857 7.71941 0.0058507

fit.5 <- lm(Waiting.Time ~ poly(Eruptions, 5), data=faithful)
anova(fit.5, fit.4)

```

```

Analysis of Variance Table
##
Model 1: Waiting.Time ~ poly(Eruptions, 5)
Model 2: Waiting.Time ~ poly(Eruptions, 4)
Res.Df RSS Df Sum of Sq F Pr(>F)
1 266 8391.682662873
2 267 8413.382397482 -1 -21.69973460923 0.68784 0.40764

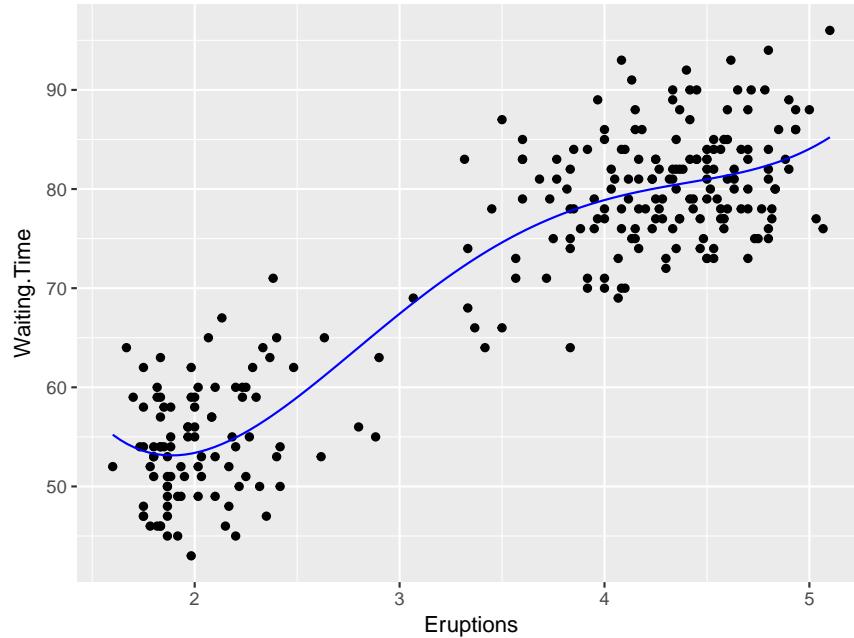
```

and it seems we are done, the power 5 model is NOT stat. significantly better than the power

4 model.

Again, what does this look like?

```
y <- predict(fit.4, newdata=list(Eruptions=x))
ggplot(data=faithful, aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_line(aes(x,y), data=data.frame(x=x, y=y), color="blue")
```



Here is a different solution: there are a number of measures of how well a curve fits a set of data. The best known is the *coefficient of determination*:

$$R^2 = \text{cor}(\text{Observed Values}, \text{Predicted Values})^2 \times 100\%$$

It is not helpful for us in this problem, though, because a higher order polynomial can never have a worse fit, and therefore a smaller  $R^2$ . Instead we can use *Mallow's Cp*. It is sort of  $R^2$  with a penalty for the number of terms used.

We can do the following: let's define a polynomial of large enough degree so that it clearly overfits the data. Then we run all the possible regressions with any combination of the powers. We find the Cp statistic for each of these models and pick the best (aka the lowest Cp):

```
library(leaps)
x <- faithful$Eruptions
x <- (x-mean(x))/sd(x)
x2 <- x^2
x3 <- x^3
x4 <- x^4
x5 <- x^5
x6 <- x^6
X <- cbind(x, x2, x3, x4, x5, x6)
```

```

colnames(X) <- paste0("deg=", 1:6)
z <- leaps(X, faithful$Waiting.Time)
z

$which
1 2 3 4 5 6
1 TRUE FALSE FALSE FALSE FALSE FALSE
1 FALSE FALSE TRUE FALSE FALSE FALSE
1 FALSE FALSE FALSE FALSE TRUE FALSE
1 FALSE FALSE FALSE TRUE FALSE FALSE
1 FALSE FALSE FALSE FALSE FALSE TRUE
1 FALSE TRUE FALSE FALSE FALSE FALSE
2 TRUE TRUE FALSE FALSE FALSE FALSE
2 TRUE FALSE FALSE TRUE FALSE FALSE
2 TRUE FALSE TRUE FALSE FALSE FALSE
2 TRUE FALSE FALSE FALSE TRUE FALSE
2 TRUE FALSE FALSE FALSE FALSE TRUE
2 FALSE FALSE TRUE FALSE TRUE FALSE
2 FALSE TRUE TRUE FALSE FALSE FALSE
2 FALSE FALSE TRUE TRUE FALSE FALSE
2 FALSE FALSE TRUE FALSE FALSE TRUE
2 FALSE TRUE FALSE FALSE TRUE FALSE
3 TRUE TRUE FALSE TRUE FALSE FALSE
3 TRUE TRUE FALSE FALSE FALSE TRUE
3 TRUE FALSE FALSE TRUE FALSE TRUE
3 TRUE TRUE FALSE FALSE TRUE FALSE
3 TRUE TRUE TRUE FALSE FALSE FALSE
3 TRUE FALSE FALSE FALSE TRUE TRUE
3 TRUE FALSE TRUE TRUE FALSE FALSE
3 TRUE FALSE TRUE FALSE FALSE TRUE
3 TRUE FALSE FALSE FALSE TRUE TRUE
3 TRUE FALSE TRUE FALSE TRUE FALSE
4 TRUE TRUE TRUE TRUE FALSE FALSE
4 TRUE TRUE FALSE TRUE TRUE FALSE
4 TRUE TRUE TRUE FALSE FALSE TRUE
4 TRUE TRUE FALSE FALSE TRUE TRUE
4 TRUE FALSE TRUE TRUE FALSE FALSE
4 TRUE FALSE FALSE FALSE TRUE TRUE
4 TRUE FALSE TRUE TRUE TRUE FALSE
4 TRUE FALSE FALSE FALSE TRUE TRUE
4 TRUE FALSE TRUE FALSE TRUE TRUE
5 TRUE TRUE TRUE TRUE TRUE FALSE
5 TRUE TRUE TRUE FALSE TRUE TRUE
5 TRUE TRUE TRUE TRUE FALSE TRUE
5 TRUE TRUE FALSE TRUE TRUE TRUE

```

```

5 TRUE FALSE TRUE TRUE TRUE TRUE
5 FALSE TRUE TRUE TRUE TRUE TRUE
6 TRUE TRUE TRUE TRUE TRUE TRUE
##
$label
[1] "(Intercept)" "1" "2" "3" "4"
[6] "5" "6"
##
$size
[1] 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5
[36] 5 6 6 6 6 6 6 6 7
##
$Cp
[1] 30.30329671553761 181.31135585137963 360.77310129516195
[4] 666.98775951550454 707.63445883300562 729.07673353558266
[7] 17.56187011388050 27.24664811196124 27.80285873978755
[10] 30.08012621472619 31.29770625626276 101.79682640162946
[13] 165.94323170001650 180.99568564846896 181.89011539900724
[16] 340.29867817315903 3.02723302259864 3.10261318588891
[19] 6.66291040431571 8.86324302785891 9.45065766046690
[22] 17.90416792089604 17.98578642894842 23.27890937528412
[25] 24.11501693143759 25.38670376131705 3.76690052143709
[28] 4.25873240461425 4.43513176107751 4.85879386327775
[31] 4.89270687710246 8.36622104621517 8.65542367280034
[34] 10.86298748590076 19.79075974069474 25.23668899970721
[37] 5.08143646705634 5.13435841001456 5.70577807426417
[40] 6.20533790703360 7.71284046153744 41.27801664017420
[43] 7.000000000000000

I <- c(1:length(z$Cp))[z$Cp==min(z$Cp)]
I

[1] 17
z$Cp[I]

[1] 3.02723302259864
c(1:6)[z$which[I,]]

[1] 1 2 4

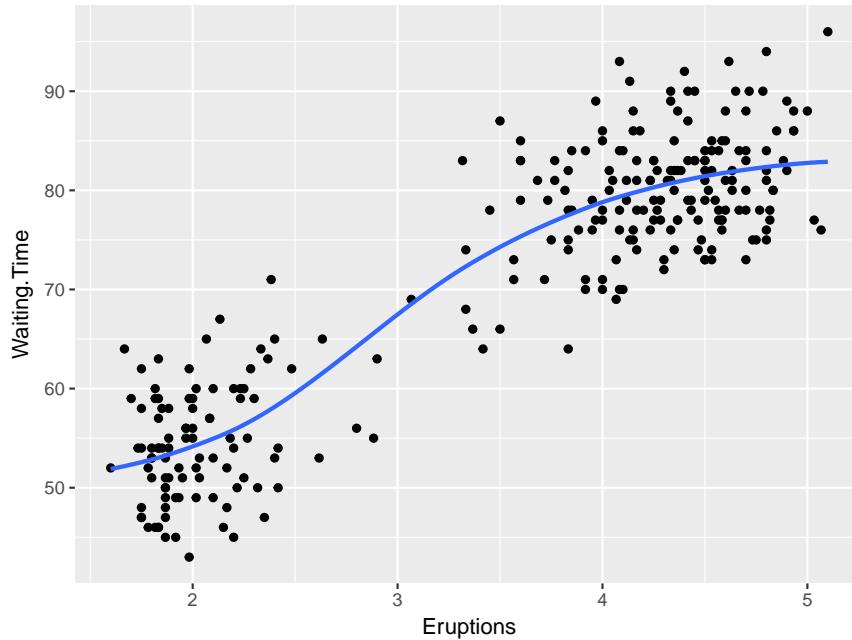
```

so this chooses a model with highest degree 4. This is the same as before, which is nice. This does not however have to happen, the two solutions can sometimes yield different models.

## 32.2 Nonparametric Regression

Consider the following graph:

```
ggplot(data=faithful, aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_smooth(se=FALSE)
```



what is this curve? It is nonparametric regression fit, that is there is no functional form specified. There are a number of ways to fit such a curve, the one used here is called *loess*. We can do the fit ourselves:

```
fit.loess <- loess(Waiting.Time ~ Eruptions, data=faithful)
summary(fit.loess)
```

```
Length Class Mode
n 1 -none- numeric
fitted 272 -none- numeric
residuals 272 -none- numeric
enp 1 -none- numeric
s 1 -none- numeric
one.delta 1 -none- numeric
two.delta 1 -none- numeric
trace.hat 1 -none- numeric
divisor 1 -none- numeric
robust 272 -none- numeric
pars 10 -none- list
kd 5 -none- list
call 3 -none- call
terms 3 terms call
xnames 1 -none- character
x 272 -none- numeric
y 272 -none- numeric
```

```
weights 272 -none- numeric
```

One thing we have lost is an understanding of the model, because now there is none! This does not matter, though, if our goal is prediction:

```
tmp <- predict(fit.loess, newdata=list(Eruptions=2.5), se=TRUE)
tmp
```

```
$fit
Eruptions
59.5469609332631

$se.fit
Eruptions
0.843008339275481

$residual.scale
[1] 5.64254609329652

$df
[1] 267.614212478678
```

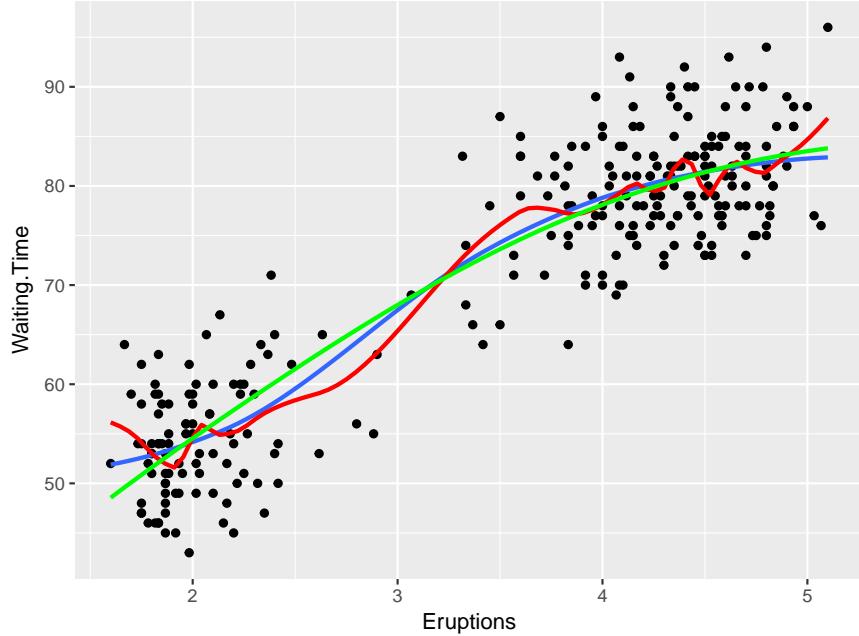
as with the lm command, the standard error is for a confidence interval, if we want a prediction interval we need to find

```
sigma <- sd(fit.loess$residuals)
se_p <- sqrt(tmp$se.fit^2 + sigma^2)
tmp$fit + c(-1, 1)*qnorm(0.975)*se_p

[1] 48.4420978531998 70.6518240133262
```

All nonparametric regression methods have a *tuning parameter*, that is a way to adjust the smoothness of the curve. In the case of the loess method it is called *span*. The default is 0.75 but we can change that:

```
ggplot(data=faithful, aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_smooth(se=FALSE) +
 geom_smooth(se=FALSE, span=0.2, color="red") +
 geom_smooth(se=FALSE, span=2, color="green")
```



so a smaller value of span leads to a more rugged curve.

Is there a way to find an *optimal* span? One common idea is called *cross-validation*:

- set aside some part of the data (called the *evaluation* data)
- do the fit with the rest (called the *training* data)
- use the model to predict for the evaluation data
- compare the true responses with the predicted ones
- repeat many times
- find the average deviation
- repeat until you have the span that yields the smallest such deviation

How much of the data should be used for training and for evaluation? There are a number of common answers:

- leave-one-out cross-validation: just one observation for evaluation
- k-fold cross-validation: k observations in evaluation set (often k=10)
- bootstrap cross-validation: choose a bootstrap sample, use it for training, evaluate on observations not in bootstrap sample. (this is often called the 0.632 rule because the probability for any one observation to be in the bootstrap sample is asymptotically  $1 - e^{-1} = 0.623$ )

Let's write a routine that finds the best span using 10-fold cross validation:

```

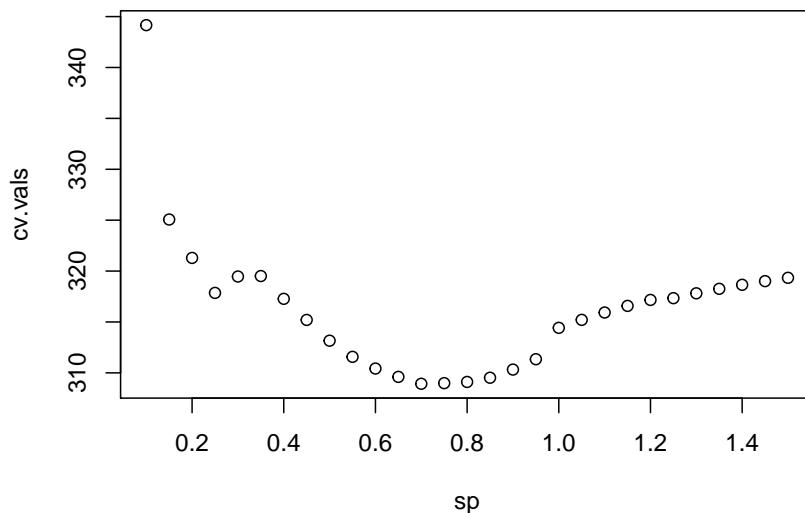
cv <- function(y, x, span, k=10) {
 n <- length(x)
 m <- (n-n%%k)/k
 dev <- rep(0, m+1)
 for(i in 1:m) {
 L <- ((i-1)*k+1):(i*k)
 fit <- loess(y[-L] ~ x[-L], span=span)
 yhat <- predict(fit, newdata=x[L])
 dev[i] <- sum((y[L]-yhat)^2, na.rm = TRUE)
 }
 if(m*10<n) {
 L <- (10*m):n
 fit <- loess(y[-L] ~ x[-L], span=span)
 yhat <- predict(fit, newdata=x[L])
 dev[i+1] <- sum((y[L]-yhat)^2, na.rm = TRUE)
 }
 else dev <- dev[1:m]
 mean(dev)
}

```

```

sp <- seq(0.1, 1.5, 0.05)
cv.vals <- 0*sp
for(i in seq_along(sp))
 cv.vals[i] <- cv(faithful$Waiting.Time, faithful$Eruptions,
 span=sp[i])
plot(sp, cv.vals)

```



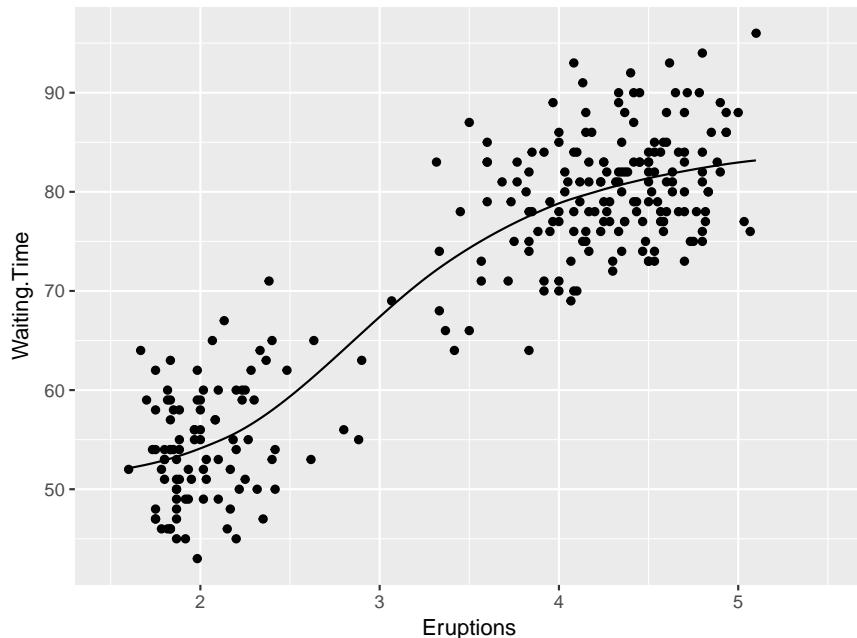
```

sp <- sp[cv.vals==min(cv.vals)]
sp

[1] 0.7

fit <- loess(Waiting.Time ~ Eruptions, data=faithful,
 span=sp)
x <- seq(1.5, 5.5, length=250)
y <- predict(fit, newdata=x)
ggplot(data=faithful, aes(x=Eruptions, y=Waiting.Time)) +
 geom_point() +
 geom_line(data=data.frame(x=x, y=y), aes(x,y))

```



Finally, is the parametric power 4 model from above better or worse than this non-parametric one? We can no longer use the F test or Mallow's Cp because those work on *nested* models (one is a special case of the other). Also notice that the summary of the loess fit does not give an  $R^2$ , and in fact there is none.

Actually, there is no generally agreed upon way to decide this question! In some way, the two approaches are not comparable.

## 33 Bayesian Analysis

### 33.1 Prior and Posterior Distribution

Say we have a sample  $\mathbf{X} = (X_1, \dots, X_n)$ , iid from some probability density  $f(\cdot | \theta)$ , and we want to do some inference on the parameter  $\theta$ .

A Bayesian analysis begins by specifying a *prior* distribution  $\pi(\theta)$ . Then one uses Bayes' formula to calculate the *posterior distribution*:

$$f(\theta|\mathbf{x}) = f(\mathbf{x}|\theta)\pi(\theta)/m(\mathbf{x})$$

where  $m(\mathbf{x})$  is the marginal distribution

$$m(\mathbf{x}) = \int \dots \int f(\mathbf{x}|\theta)\pi(\theta)d\theta$$

### 33.1.1 Example: Normal mean, normal prior

$X \sim N(\mu, \sigma)$  independent,  $\sigma$  known,  $\mu \sim N(a, b)$ .

Now

$$\begin{aligned} m(x) &= \int_{-\infty}^{\infty} f(x|\mu)\pi(\mu)d\mu = \\ &\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \frac{1}{\sqrt{2\pi b^2}} e^{-\frac{1}{2b^2}(\mu-a)^2} d\mu \end{aligned}$$

Note

$$\begin{aligned} (x - \mu)^2/\sigma^2 + (\mu - a)^2/b^2 &= \\ x^2/\sigma^2 - 2x\mu/\sigma^2 + \mu^2/\sigma^2 + \mu^2/b^2 - 2a\mu/b^2 + a^2/b^2 &= \\ (1/\sigma^2 + 1/b^2)\mu^2 - 2(x/\sigma^2 + a/b^2)\mu + K_1 &= \\ (1/\sigma^2 + 1/b^2) \left( \mu^2 - 2\frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}\mu \right) + K_2 &= \\ \frac{(\mu - d)^2}{c^2} + K_3 & \end{aligned}$$

where  $d = \frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}$  and  $c = 1/\sqrt{1/\sigma^2 + 1/b^2}$

therefore

$$m(x) = K_4 \int_{-\infty}^{\infty} e^{-\frac{1}{2c^2}(\mu-d)^2} d\mu = K_5$$

because the integrand is a normal density with mean  $d$  and standard deviation  $c$ , so it will integrate to 1 as long as the constants are correct.

$$\begin{aligned} f(\theta|\mathbf{x}) &= f(\mathbf{x}|\theta)\pi(\theta)/m(\mathbf{x}) = \\ K_6 e^{-\frac{1}{2c^2}(\mu-d)^2} & \end{aligned}$$

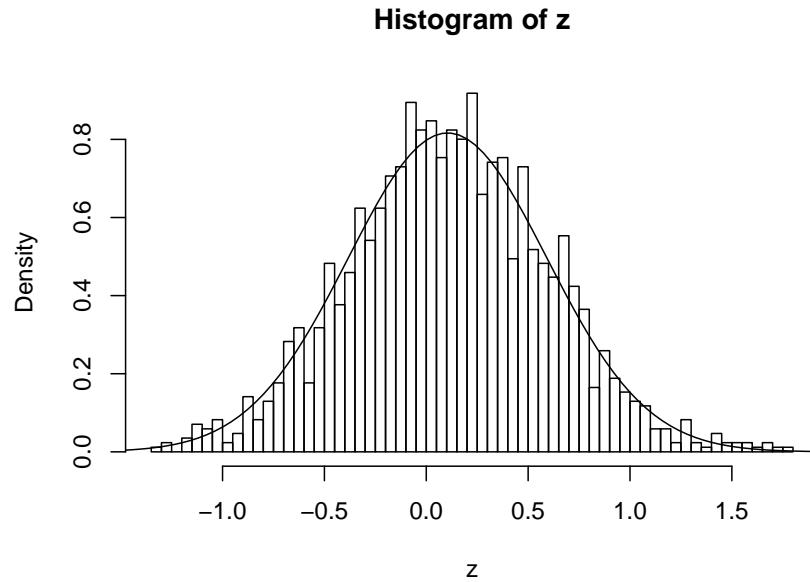
Notice that we don't need to worry about what exactly  $K_6$  is, because the posterior will be a proper probability density, so  $K_6$  will be what it has to be!

So we found

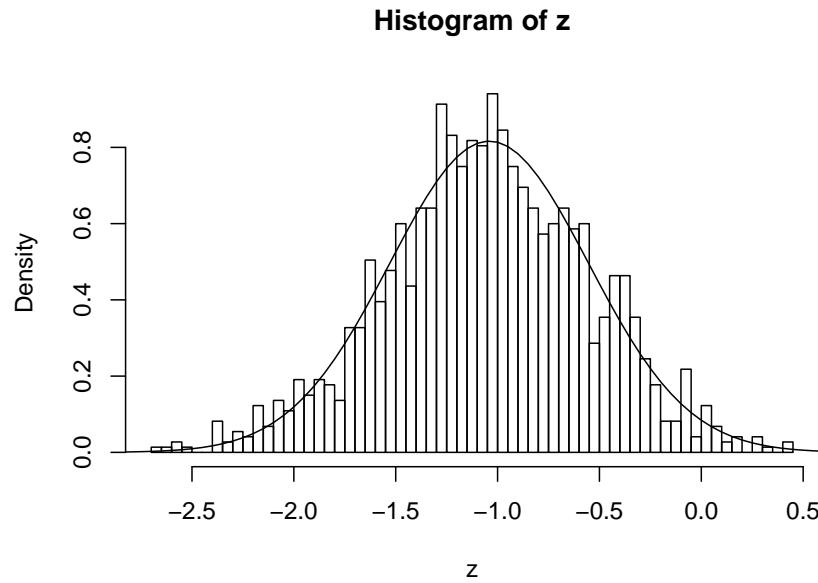
$$\mu|X = x \sim N\left(\frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}, 1/\sqrt{1/\sigma^2 + 1/b^2}\right)$$

Let's do a little simulation to see whether we got this right:

```
a <- 0.2 #just as an example
b <- 2.3
sigma <- 0.5
mu <- rnorm(1e5, a, b)
x <- rnorm(1e5, mu, sigma)
x0 <- 0.1
cc <- 1/sqrt(1/sigma^2 + 1/b^2)
d <- (x0/sigma^2+a/b^2)/(1/sigma^2 + 1/b^2)
z <- mu[x>x0-0.05 & x<x0+0.05]
hist(z, 50, freq=FALSE)
curve(dnorm(x, d, cc), -2, 2, add=TRUE)
```



```
x0 <- (-1.1)
d <- (x0/sigma^2+a/b^2)/(1/sigma^2 + 1/b^2)
z <- mu[x>x0-0.05 & x<x0+0.05]
hist(z, 50, freq=FALSE)
curve(dnorm(x,d, cc), -3, 2, add=TRUE)
```



### 33.1.2 Example: Binomial proportion, Uniform prior

$X \sim \text{Bin}(n, p)$ ,  $p \sim U[0, 1]$

$$\begin{aligned} m(x) &= \int_{-\infty}^{\infty} f(x|\mu)\pi(\mu)d\mu = \\ &\int_0^1 \binom{n}{x} p^x (1-p)^{n-x} 1 dp = \\ &K_1 \int_0^1 p^{(x+1)-1} (1-p)^{(n-x+1)-1} dp = K_2 \end{aligned}$$

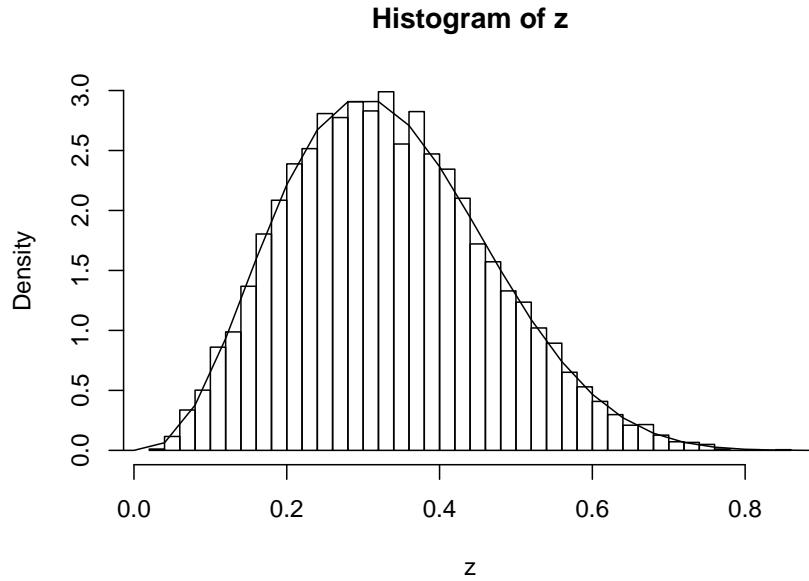
because this is (up to a constant) a Beta density which will integrate to 1. So

$$\begin{aligned} f(\theta|x) &= f(x|\theta)\pi(\theta)/m(x) = \\ &K_3 p^{(x+1)-1} (1-p)^{(n-x+1)-1} \end{aligned}$$

and we find

$$p|X=x \sim \text{Beta}(x+1, n-x+1)$$

```
n <- 10
p <- runif(1e5)
x <- rbinom(1e5, n, p)
x0 <- 3
z <- p[x==x0]
hist(z, 50, freq=FALSE)
curve(dbeta(x, x0+1, n-x0+1), -2, 2, add=T)
```



## 33.2 Inference

In a Bayesian analysis any inference is done from the posterior distribution. For example, point estimates can be found as the mean, median, mode or any other measure of central tendency:

### 33.2.1 Example: Normal mean, normal prior

say the following is a sample  $X_1, \dots, X_n$  from a normal with standard deviation  $\sigma = 2.3$ :

```
dta.norm
```

```
[1] 0.0 1.9 3.4 3.4 4.1 4.1 4.4 4.8 5.3 5.8 6.3 6.4 6.6 6.7 6.9 7.4 7.7
[18] 8.2 8.5 9.8
```

if we decide to base our analysis on the sample mean we have  $\bar{X} \sim N(\mu, \sigma/\sqrt{n})$ . Now if we use the posterior mean we find

$$E[\mu|X = x] = \frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}$$

now we need to decide what a and b to use. If we have some prior information we can use that. Say we expect a priori that  $\mu = 5$ , and of course we know  $\sigma = 2.3$ , then we could use  $a = 5$  and  $b = 3$ :

```
d <- (mean(dta.norm)/(2.3^2/20) + 5/3^2)/(1/(2.3^2/20) + 1/3^2)
d
```

```
[1] 5.56829834313778
```

If we wanted to find a 95% interval estimate (now called a credible interval) we can find it directly from the posterior distribution:

```
cc <- 1/sqrt(1/(2.3^2/20) + 1/3^2)
cc

[1] 0.506900944081795
round(qnorm(c(0.025, 0.975), d, cc), 2)

[1] 4.57 6.56
```

Note: the standard frequentist solution would be

```
round(mean(dta.norm), 2)

[1] 5.58
round(mean(dta.norm)+c(-1, 1)*qt(0.975, 12)*2.3/sqrt(20), 2)

[1] 4.46 6.71
```

### 33.2.2 Example: Binomial proportion, uniform prior

Say in a class with 134 students 31 received an A. Find a 90% credible interval for the true percentage of students who get an A in this class.

```
round(qbeta(c(0.05, 0.95), 31 + 1, 134 - 31 + 1)*100, 1)

[1] 17.8 29.7
```

### 33.2.3 Example: Normal mean, Gamma prior

let's say that  $\mu$  is a physical quantity, like the mean amount of money paid on sales. In that case it makes more sense to use a prior that forces  $\mu$  to be non-negative. For example we could use  $\mu \sim \text{Gamma}(\alpha, \beta)$ . However, now we need to find

$$m(x) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \frac{1}{\Gamma(\alpha)\beta^\alpha} \mu^{\alpha-1} e^{-\mu/\beta} d\mu$$

and this integral does not exist. We will have to use numerical methods instead. Let's again find a point estimate based on the posterior mean. As prior we will use  $\mu \sim \text{Gamma}(5, 1)$

```
fmu <- function(mu)
 dnorm(mean(dta.norm), mu, 2.3/sqrt(20)) *
 dgamma(mu, 5, 1)
mx <- integrate(fmu, lower=0, upper=Inf)$value
posterior.density <- function(mu) fmu(mu)/mx
posterior.mean <-
 integrate(
 function(mu) {mu*posterior.density(mu)},
```

```

lower = 0,
upper = Inf)$value
round(posterior.mean, 2)

```

```
[1] 5.51
```

how about a 95% credible interval? This we need to solve the equations

$$F(\mu|\mathbf{x}) = 0.025, F(\mu|\mathbf{x}) = 0.975$$

where  $F$  is the posterior distribution function. Again we need to work numerically. We can use a simple bisection algorithm:

```

pF <- function(t) integrate(posterior.density,
 lower=3, upper=t)$value
cc <- (1-0.95)/2
l <- 3
h <- posterior.mean
repeat {
 m <- (l+h)/2
 if(pF(m)<cc) l <- m
 else h <- m
 if(h-l<m/1000) break
}
left.endpoint <- m
h <- 8
l <- posterior.mean
repeat {
 m <- (l+h)/2
 if(pF(m)<1-cc) l <- m
 else h <- m
 if(h-l<m/1000) break
}
right.endpoint <- m
round(c(left.endpoint, right.endpoint), 2)

```

```
[1] 4.52 6.51
```

Let's generalize all this and write a routine that will find a point estimate and a  $(1 - \alpha)100\%$  credible interval for any problem with one parameter:

```

bayes.credint <- function(x, df, prior, conf.level=0.95, acc=0.001,
 lower, upper, Show=TRUE) {
 if(any(c(missing(lower), missing(upper)))) {
 cat("Need to give lower and upper boundary\n")
 }
 posterior.density <- function(par, x) {
 y <- 0*seq_along(par)
 for(i in seq_along((par)))

```

```

y[i] <- df(x, par[i])*prior(par[i])/mx
y
}
mx <- 1
mx <- integrate(posterior.density,
 lower=lower, upper=upper, x=x)$value
if(Show) {
 par <- seq(lower, upper, length=250)
 y <- posterior.density(par, x)
 plot(par, y, type="l")
}
f.expectation <- function(par, x) par*posterior.density(par, x)
parhat <- integrate(f.expectation,
 lower=lower, upper=upper, x=x)$value
if(Show) abline(v=parhat)
pF <- function(t, x) integrate(posterior.density,
 lower=lower, upper=t, x=x)$value
cc <- (1-conf.level)/2
l <- lower
h <- parhat
repeat {
 m <- (l+h)/2
 if(pF(m, x)<cc) l <- m
 else h <- m
 if(h-l<acc*m) break
}
left.endpoint <- m
h <- upper
l <- parhat
repeat {
 m <- (l+h)/2
 if(pF(m, x)<1-cc) l <- m
 else h <- m
 if(h-l<acc*m) break
}
right.endpoint <- m
if(Show) abline(v=c(left.endpoint, right.endpoint))
c(parhat, left.endpoint, right.endpoint)
}

```

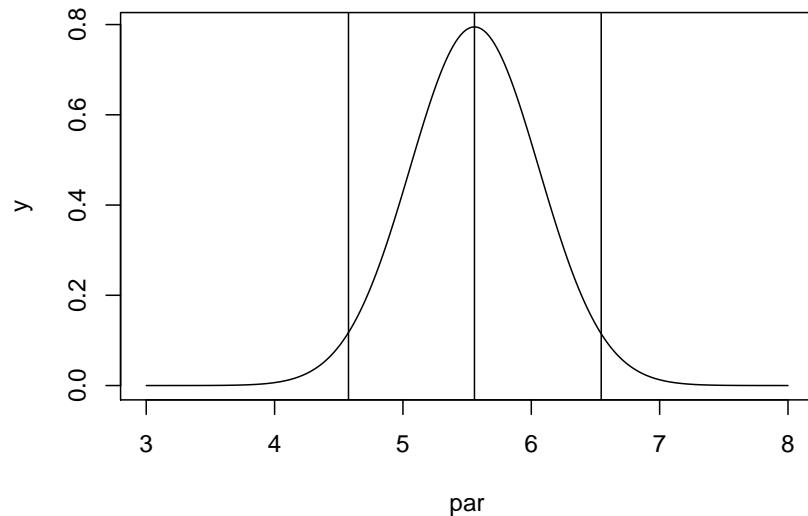
### 33.2.4 Example: Normal mean, normal prior

```

df <- function(x, par) dnorm(x, par, 2.3/sqrt(20))
prior <- function(par) dnorm(par, 5, 2.3)
round(bayes.credint(mean(dta.norm), df=df, prior=prior,

```

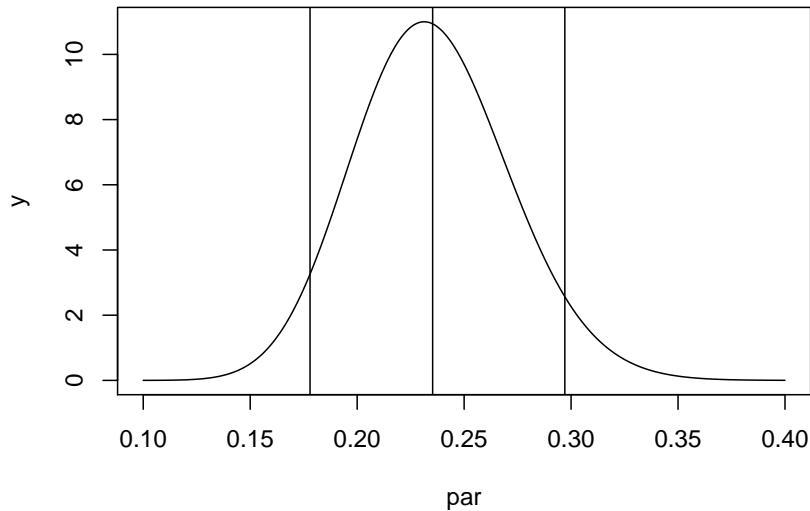
```
lower=3, upper=8, Show=T), 2)
```



```
[1] 5.56 4.58 6.54
```

### 33.2.5 Example Binomial proportion, uniform prior

```
df <- function(x, par) dbinom(x, 134, par)
prior <- function(par) dunif(par)
round(100*bayes.credint(x=31, df=df, prior=prior, acc=0.0001,
conf.level = 0.9, lower=0.1, upper=0.4, Show=T), 1)
```



```
[1] 23.5 17.8 29.7
```

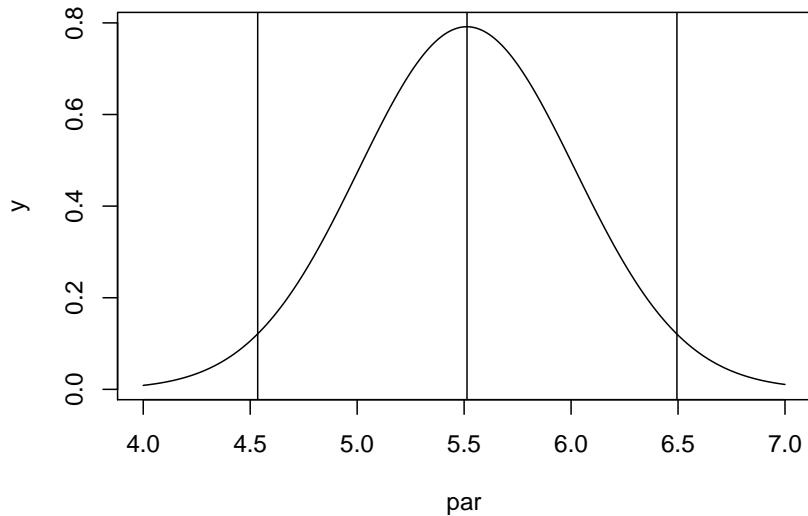
which is the same before:

```
round(qbeta(c(0.05, 0.95), 31 + 1, 134 - 31 + 1)*100, 1)
```

```
[1] 17.8 29.7
```

### 33.2.6 Example: Normal mean, Gamma prior

```
df <- function(x, par) dnorm(x, par, 2.3/sqrt(20))
prior <- function(par) dgamma(par, 5, 1)
round(bayes.credint(mean(dta.norm), df=df, prior=prior,
lower=4, upper=7, Show=TRUE), 2)
```



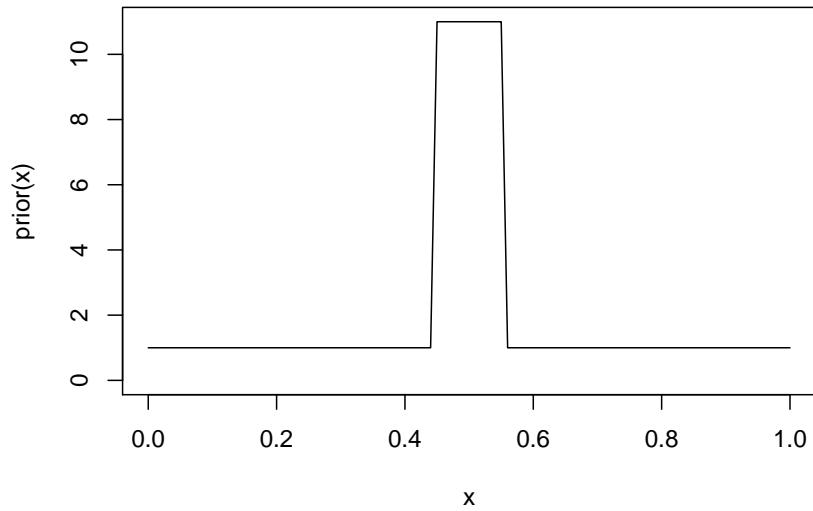
```
[1] 5.51 4.54 6.49
```

### 33.2.7 Example: Binomial proportion, Lincoln's hat prior

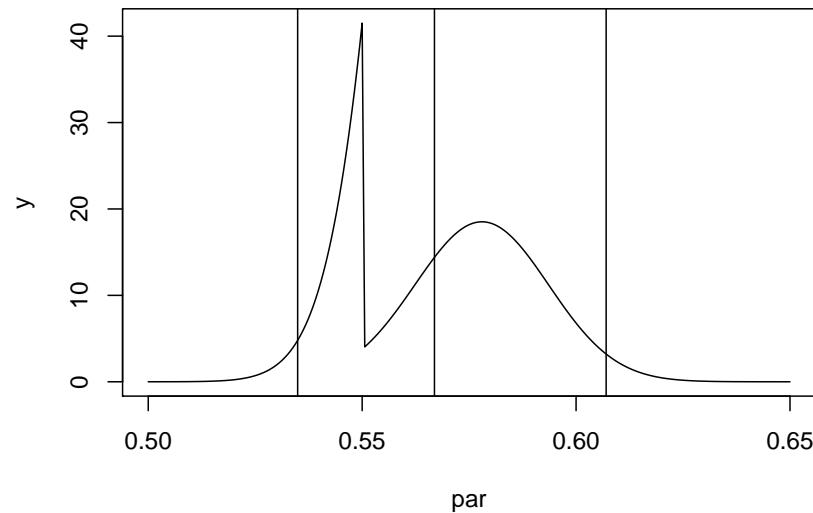
Say we pick a coin from our pocket. We flip it 1000 time and get 578 heads. We want to find a 95% credible interval for the proportion of heads.

What would be good prior here? We might reason as follows: on the one had we are quite sure that instead it is an “almost” fair coin. On the other hand if it is not a fair coin we really don’t know how unfair it might be. We can encode this in the *Lincoln’s hat* prior:

```
prior <- function(x) dunif(x) + dunif(x, 0.45, 0.55)
curve(prior, 0, 1, ylim=c(0, 11))
```



```
df <- function(x, par) dbinom(x, 1000, par)
round(bayes.credint(x=578, df=df, prior=prior, acc=0.0001,
lower=0.5, upper=0.65, Show=TRUE), 3)
```



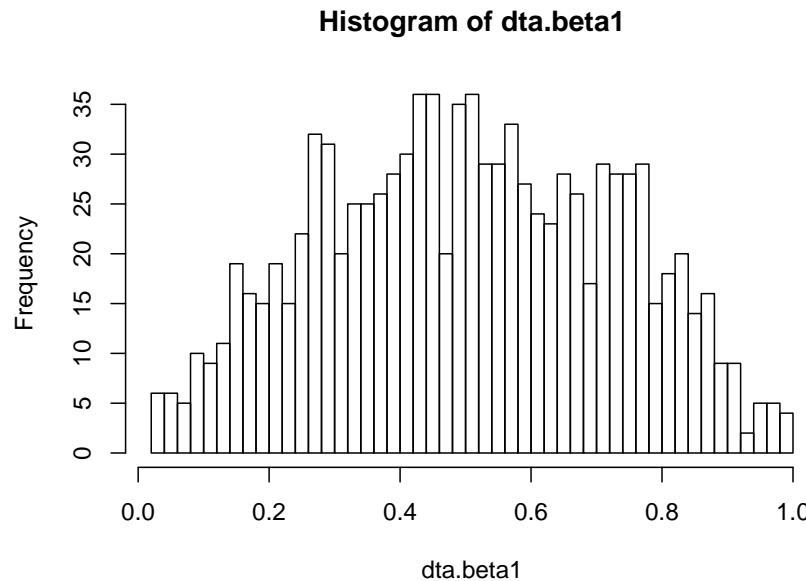
```
[1] 0.567 0.535 0.607
```

So, have we just solved the Bayesian estimation problem for one parameter?

### 33.2.8 Example: Beta density, Gamma prior

consider the following sample:

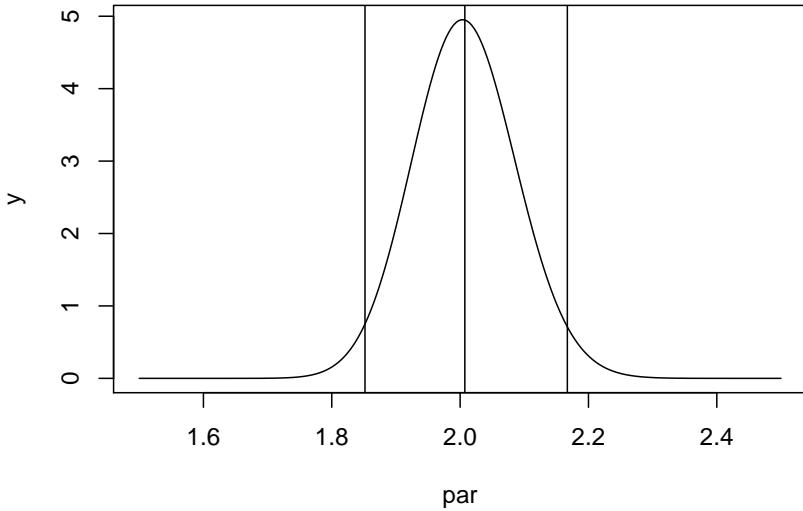
```
dta.beta1 <- round(rbeta(1000, 2, 2), 3)
hist(dta.beta1, 50)
```



Let's say we know that this is from a  $\text{Beta}(a, a)$  distribution and we want to estimate  $a$ . As a prior we want to use  $\text{Gamma}(2, 1)$

Now what is df? Because this is an independent sample we find  $df(x, a) = \prod_{i=1}^n \text{dbeta}(x_i, a, a)$ , so

```
df <- function(x, par) prod(dbeta(x, par, par))
prior <- function(par) dgamma(par, 2, 1)
round(bayes.credint(dta.beta1, df=df, prior=prior,
 lower=1.5, upper=2.5, Show=TRUE), 2)
```



```
[1] 2.01 1.85 2.17
```

so far, so good. But now

```
dta.beta2 <- round(rbeta(10000, 2, 2), 3)
bayes.credint(dta.beta2, df=df, prior=prior,
 lower=1.5, upper=2.5, Show=TRUE)
```

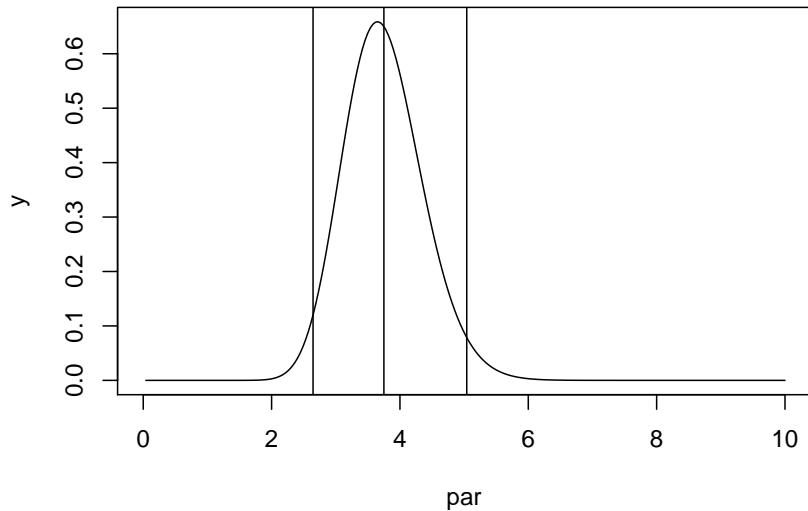
```
Error in integrate(posterior.density, lower = lower, upper = upper, x = x): non-finite
```

Why does this not work? The problem is that the values in  $\prod_{i=1}^n \text{dbeta}(x_i, a, a)$  get so small that R can't handle them anymore!

Occasionally one can avoid this problem by immediately choosing a *statistic*  $T(x)$ , aka a function of the data, so that  $T(X)$  has a distribution that avoids the product. That of course is just what we did above by going to  $\bar{X}$  in the case of the normal! In fact, it is also what we did in the case of the Binomial, because we replace the actual data (a sequence of Bernoulli trials) with their sum.

Can we generalize this to more than one parameter? In principle yes, but in practise no, at least not if the number of parameters is much more than 3 or 4. The main problem is the calculation of the marginal  $m(x)$ , because numerical integration in higher-dimensional spaces is very difficult. In that case a completley different approach is used, namely sampling from the posterior distribution using so called MCMC (Markov Chain Monte Carlo) algorithms.

```
df <- function(x, par) dpois(sum(x), 10*par)
prior <- function(par) 1/sqrt(par)
x <- rpois(10, 5)
bayes.credint(x=37, df=df, prior=prior, lower=0, upper=10, Show=T)
```



```
[1] 3.74999999999835 2.64587402343634 5.04089355468619
```

## 34 Classification

### 34.1 Example: Fisher's Iris data set

One of the most famous data sets in all of statistics is Fisher's iris data. This is data on four types of iris flowers together with information on the length and width of their sepals and petals. The problem is to use this information to predict the type of iris.

For pictures of the flowers and more info go to [https://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](https://en.wikipedia.org/wiki/Iris_flower_data_set).

In essence what we have here is a regression problem where the response variable is categorical. This type of problem is called “classification”.

The data is part of base R:

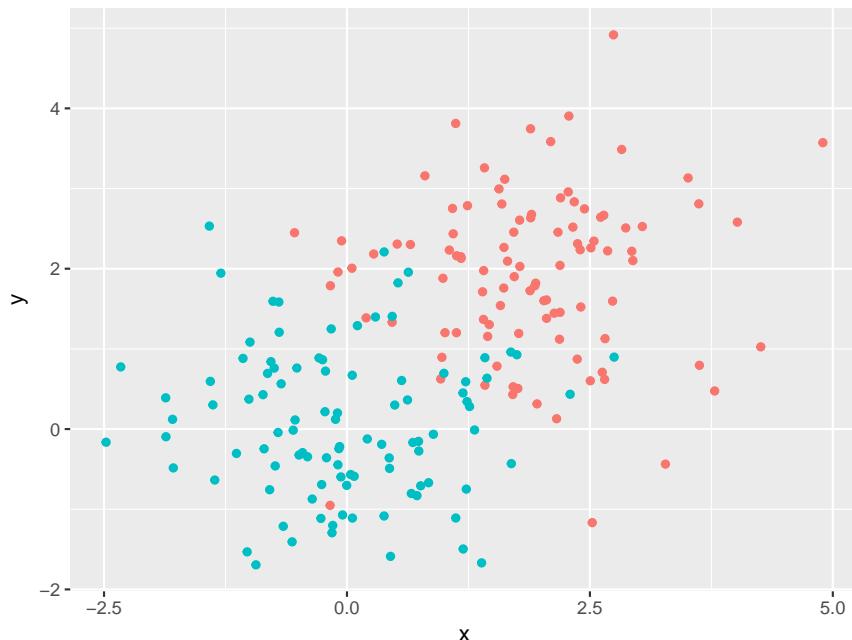
```
head(iris)
```

```
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
```

Before we consider the iris data, let's look at a couple of artificial examples:

1. Two groups, simple separation:

```
example.data.1 <- function(mu=2, n=100) {
 x <- rnorm(2*n)
 y <- rnorm(2*n)
 x[1:n] <- x[1:n] + mu
 y[1:n] <- y[1:n] + mu
 data.frame(x = x,
 y = y,
 z1 = rep(c("Blue", "Red"), each=n),
 z2 = rep(0:1, each=n))
}
do.graph1 <- function(dta) {
 ggplot() +
 geom_point(data = dta,
 aes(x, y, color=z1)) +
 theme(legend.position="none")
}
do.graph1(example.data.1())
```



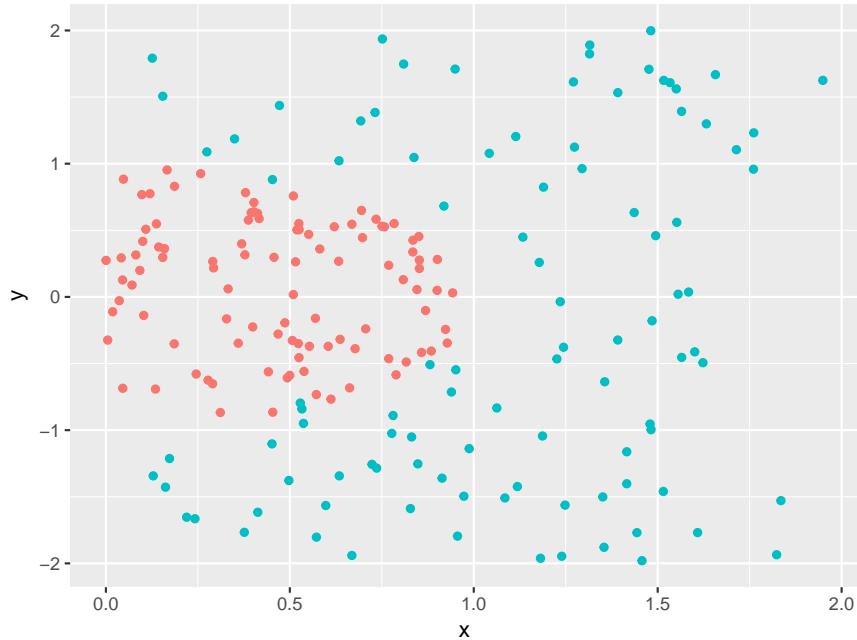
2. two groups, complicated separation:

```
example.data.2 <- function(n=100) {
 x <- cbind(runif(1000), runif(1000, -1, 1))
 x <- x[x[, 1]^2 + x[, 2]^2 < 1,]
 x <- x[1:n,]
 y <- cbind(runif(1000, 0, 2),
 runif(1000, -2, 2))
 y <- y[y[, 1]^2 + y[, 2]^2 > 0.9,]
```

```

y <- y[1:n ,]
data.frame(x = c(x[, 1], y = y[, 1]),
 y = c(x[, 2], y = y[, 2]),
 z1 = rep(c("Blue", "Red"), each=n),
 z2 = rep(0:1, each=n))
}
do.graph1(example.data.2())

```

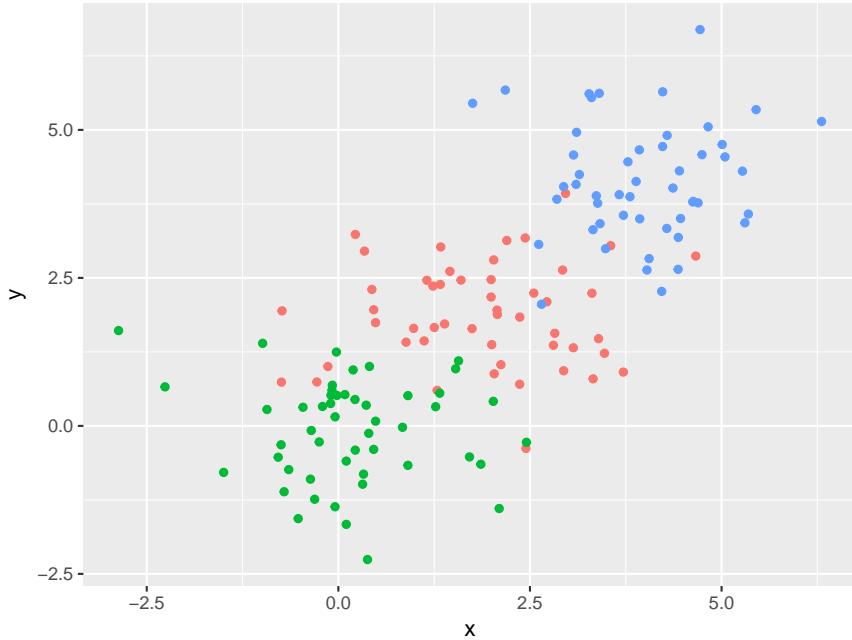


1. Three groups:

```

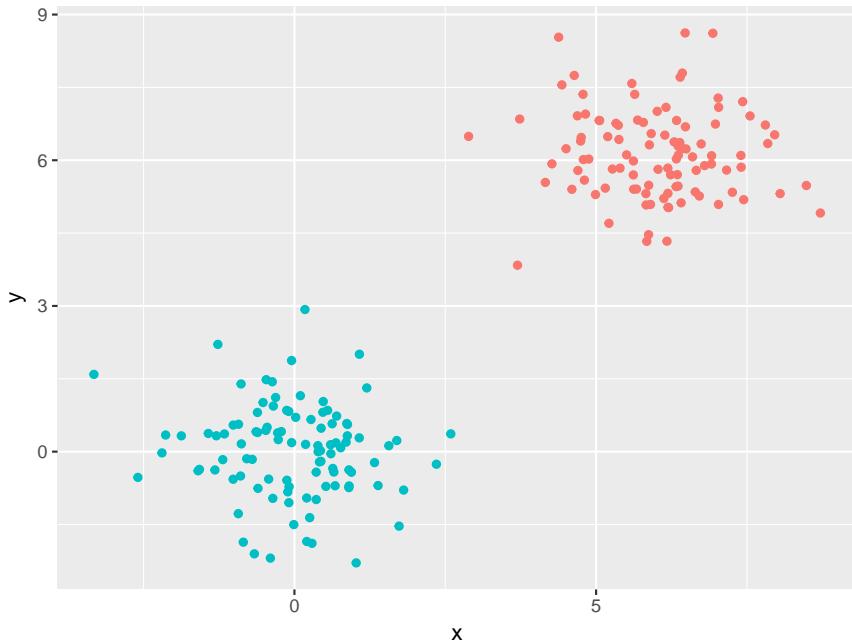
example.data.3 <- function(mu=2, n=50) {
 x <- rnorm(3*n)
 y <- rnorm(3*n)
 x[1:n] <- x[1:n] + mu
 y[1:n] <- y[1:n] + mu
 x[1:n+n] <- x[1:n+n] + 2*mu
 y[1:n+n] <- y[1:n+n] + 2*mu
 data.frame(x = x,
 y = y,
 z1 = rep(c("Blue", "Red", "Green"),
 each=n),
 z2 = rep(0:2, each=n))
}
do.graph1(example.data.3())

```



Let's start with the first example. If we redraw it with a larger mu

```
set.seed(111)
dta <- example.data.1(mu=6)
do.graph1(dta)
```



it is clear that in this case there should exist a line that completely separates the two groups. How can we find that line? Actually, we can use linear regression:

```
fit <- lm(z2 ~ x+y, data=dta)
cf <- round(coef(fit), 3)
```

```
cf
```

```
(Intercept) x y
0.978 -0.070 -0.087
```

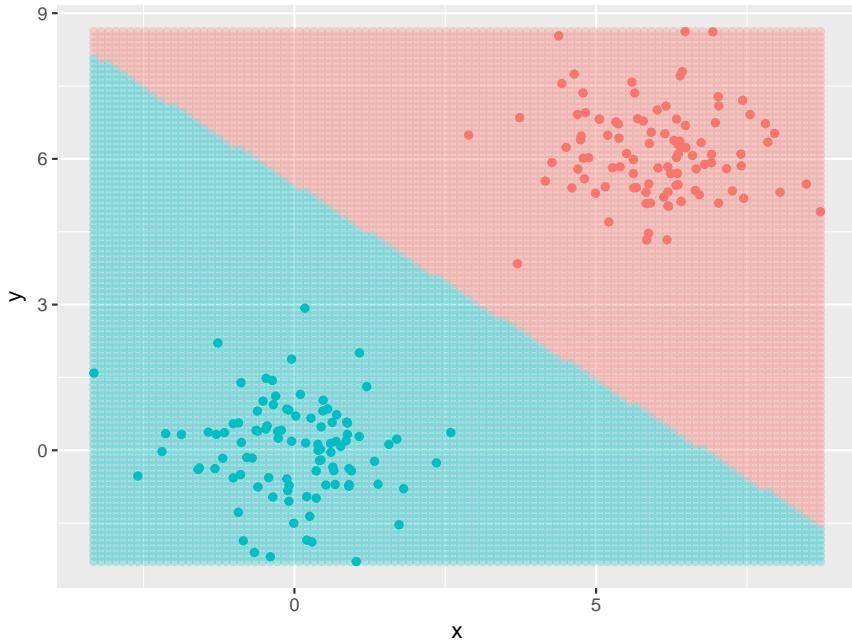
Now for each point  $(x, y)$  in the grid this predicts  $\hat{z}$ . Of course this is regular regression, so  $\hat{z}$  will be a continuous variable. We coded the data using 0 and 1, so logically a value of  $\hat{z}$  close to 0 should be marked as blue and a value close to 1 as red. We can therefore predict Blue if

$$0.978 - 0.070x - 0.087y < 0.5$$

and as Red otherwise.

Let's see what that looks like. To do so we create a fine grid of points and predict the color for each:

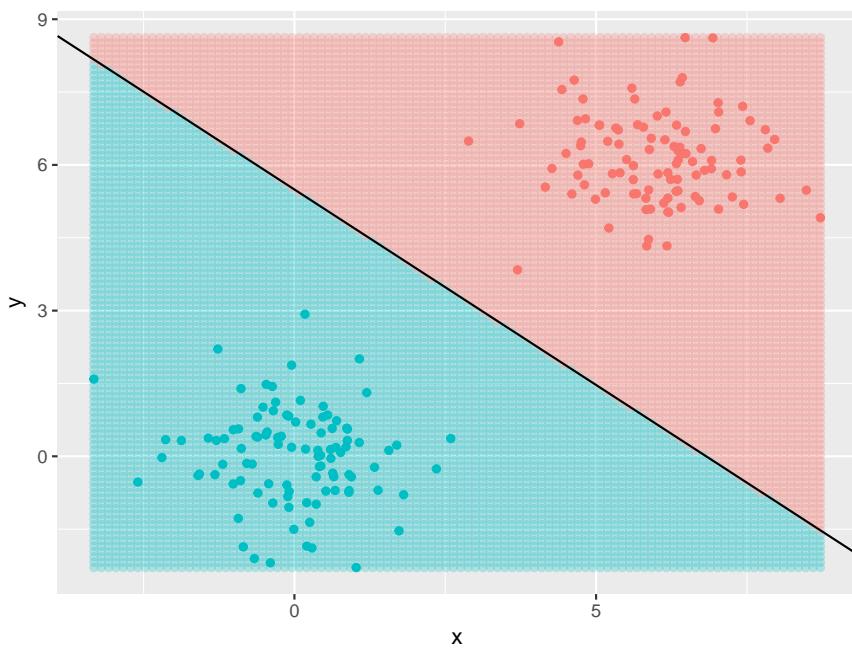
```
do.predict <- function(dta, fit) {
 x.grid <- seq(min(dta$x), max(dta$x), length=100)
 y.grid <- seq(min(dta$y), max(dta$y), length=100)
 xy.grid <- expand.grid(x.grid, y.grid)
 dta.predict <- data.frame(x = xy.grid[, 1],
 y = xy.grid[, 2])
 zhat <- predict(fit, newdata = dta.predict)
 if(length(table(dta$z1))==2)
 zhat <- ifelse(zhat<0.5, "Blue", "Red")
 else {
 tmp <- ifelse(zhat<2/3, "Red", "Blue")
 tmp[zhat>4/3] <- "Green"
 zhat <- tmp
 }
 dta.predict$zhat <- zhat
 dta.predict
}
do.graph2 <- function(dta, fit) {
 dta.predict <- do.predict(dta, fit)
 ggplot() +
 geom_point(data = dta,
 aes(x, y, color=z1)) +
 theme(legend.position="none") +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25) +
 theme(legend.position="none")
}
do.graph2(dta, fit)
```



Of course we can calculate the *decision boundary* explicitly:

$$0.978 - 0.070x - 0.087y = 0.5 \\ y = -0.805x + 5.494$$

```
do.graph2(dta, fit) +
 geom_abline(intercept = 5.494,
 slope = -0.805)
```

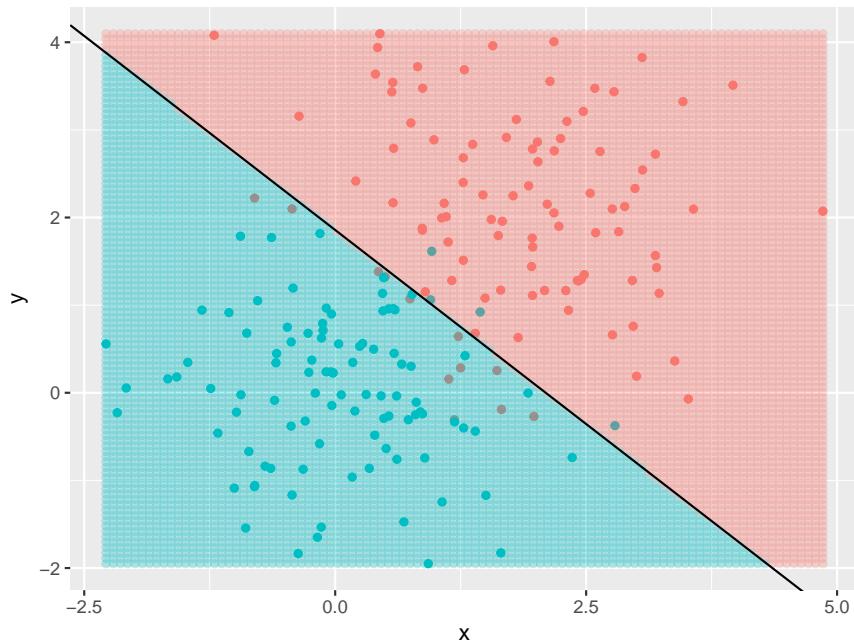


Of course if the points overlap this won't work quite so well:

```

dta <- example.data.1()
fit <- lm(z2 ~ x+y, data=dta)
do.graph2(dta, fit) +
 geom_abline(
 intercept = (0.5-coef(fit)[1])/coef(fit)[3] ,
 slope = -coef(fit)[2]/coef(fit)[3])

```

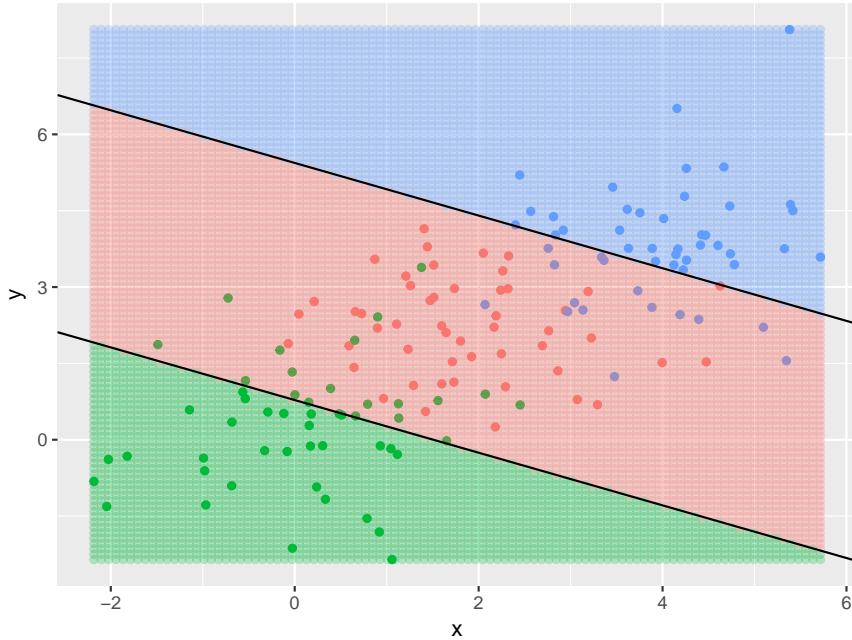


In the case of three groups we need two decision boundaries:

```

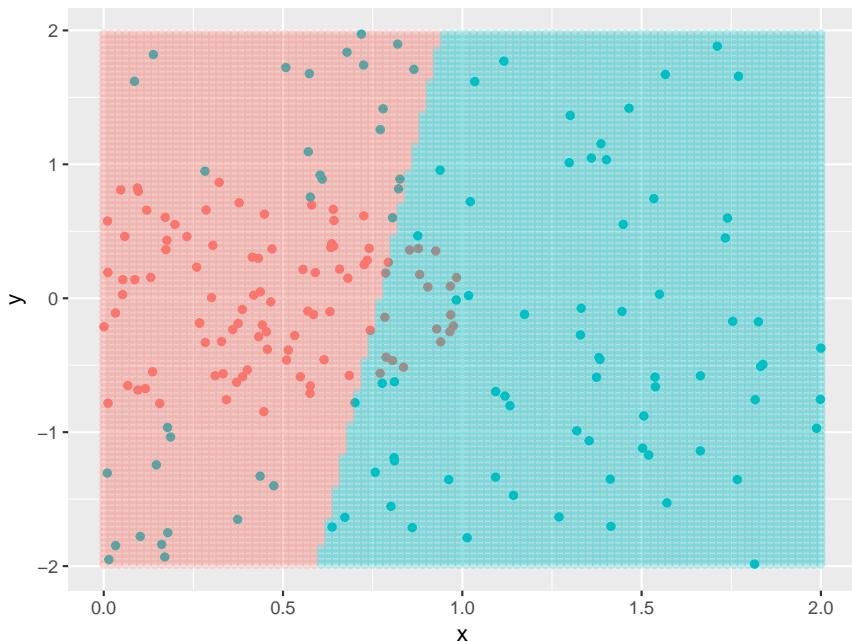
dta <- example.data.3()
fit <- lm(z2 ~ x+y, data=dta)
do.graph2(dta, fit) +
 geom_abline(
 intercept = (2/3-coef(fit)[1])/coef(fit)[3] ,
 slope = -coef(fit)[2]/coef(fit)[3]) +
 geom_abline(
 intercept = (4/3-coef(fit)[1])/coef(fit)[3] ,
 slope = -coef(fit)[2]/coef(fit)[3])

```



How about the second case?

```
set.seed(111)
dta <- example.data.2()
fit <- lm(z2~x+y, data=dta)
do.graph2(dta, fit)
```



That does not work, which is obvious because we don't have a linear decision boundary. But how about a quadratic one? Let's see

```

dta$x2 <- dta$x^2
dta$y2 <- dta$y^2
dta$xy <- dta$x*dta$y
dta <- dta[, c(3, 4, 1, 5, 2, 6, 7)]
fit <- lm(z2 ~ .-z1, data=dta)
a <- round(coef(fit), 4)
a

(Intercept) x x2 y y2 xy
-0.0719 0.3767 0.0724 -0.0593 0.2595 0.0449

```

and now

$$0.5 = a_1 + a_2x + a_3x^2 + a_4y + a_5y^2 + a_6xy$$

$$(a_1 - 0.5 + a_2x + a_3x^2) + (a_4 + a_6x)y + a_5y^2 = 0$$

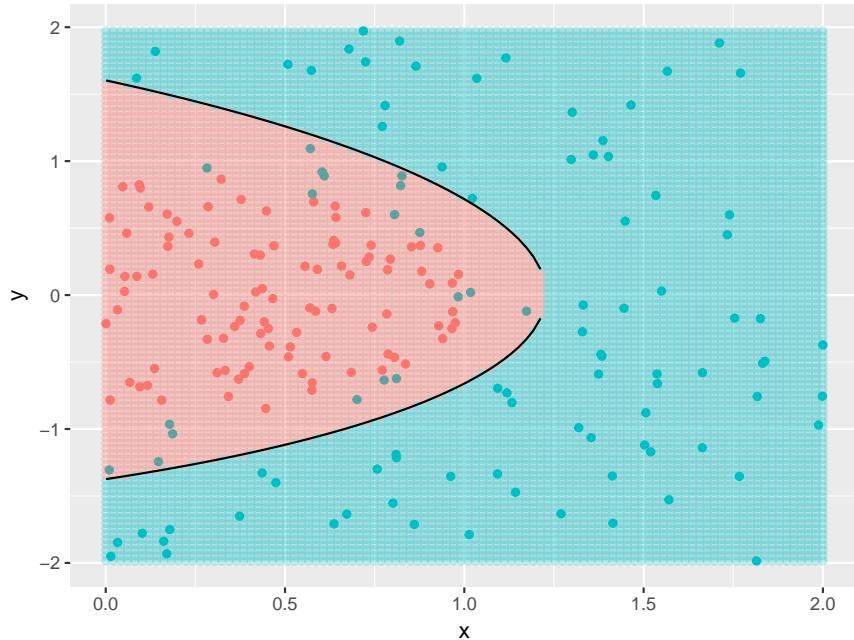
$$y = \frac{-(a_4 + a_6x) \pm \sqrt{(a_4 + a_6x)^2 - 4a_5(a_1 - 0.5 + a_2x + a_3x^2)}}{2a_5}$$

```

x.grid <- seq(min(dta$x), max(dta$x), length=100)
y.grid <- seq(min(dta$y), max(dta$y), length=100)
xy.grid <- expand.grid(x.grid, y.grid)
dta.predict <- data.frame(x = xy.grid[, 1],
 y = xy.grid[, 2])
dta.predict$x2 <- dta.predict$x^2
dta.predict$y2 <- dta.predict$y^2
dta.predict$xy <- dta.predict$x*dta.predict$y
dta.predict$z1 <- rep("Blue",
 length(dta.predict$x))
zhat1 <- predict(fit, newdata = dta.predict)
zhat <- ifelse(zhat1<0.5, "Blue", "Red")
dta.predict$zhat <- zhat
x <- seq(min(dta$x), max(dta$x), length=100)
y1 <- (-(a[4]+a[6]*x) -
 sqrt((a[4]+a[6]*x)^2 -
 4*a[5]*(a[1]-0.5+a[2]*x+a[3]*x^2)))
)/2/a[5]
y2 <- (-(a[4]+a[6]*x) +
 sqrt((a[4]+a[6]*x)^2 -
 4*a[5]*(a[1]-0.5+a[2]*x+a[3]*x^2)))
)/2/a[5]
dta.line.1 <- data.frame(x = x, y = y1)
dta.line.2 <- data.frame(x = x, y = y2)
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25) +

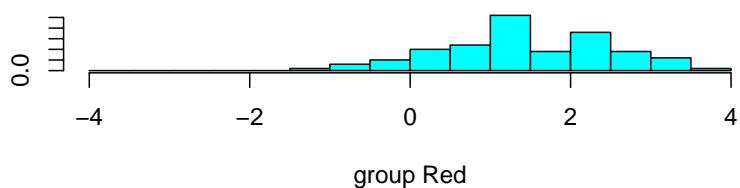
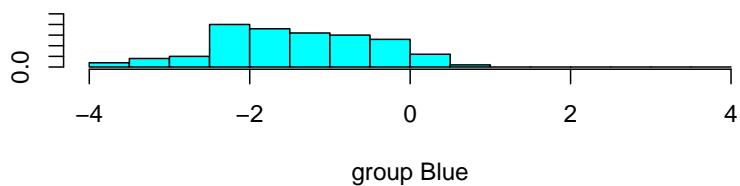
```

```
geom_line(aes(x, y), data=dta.line.1) +
 geom_line(aes(x, y), data=dta.line.2)
```



The two solutions described above are usually called Fisher's linear (LDA) and quadratic (QDA) discriminants. They can be fitted directly with the *lda* function in the *MASS* package:

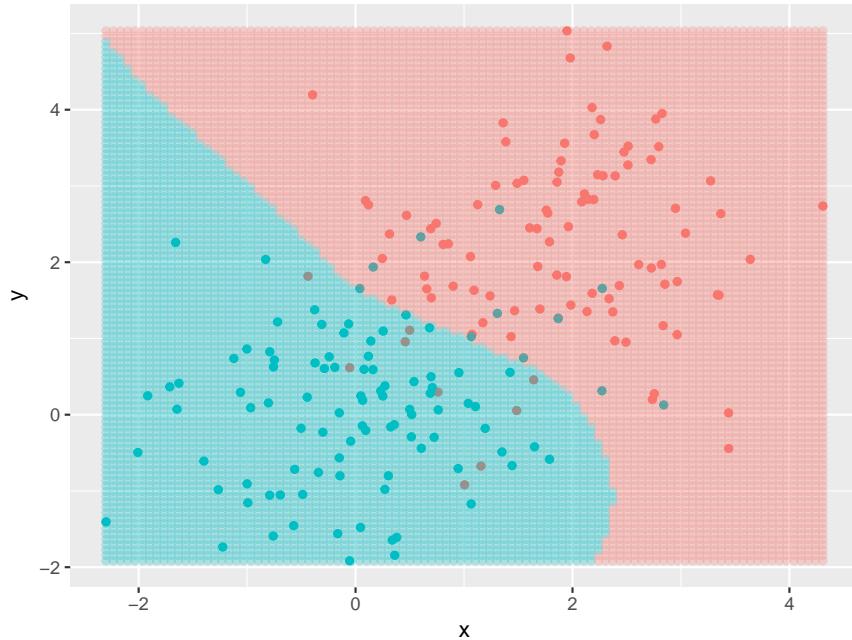
```
library(MASS)
dta <- example.data.1()
fit <- lda(z1~x+y, data=dta)
plot(fit)
```



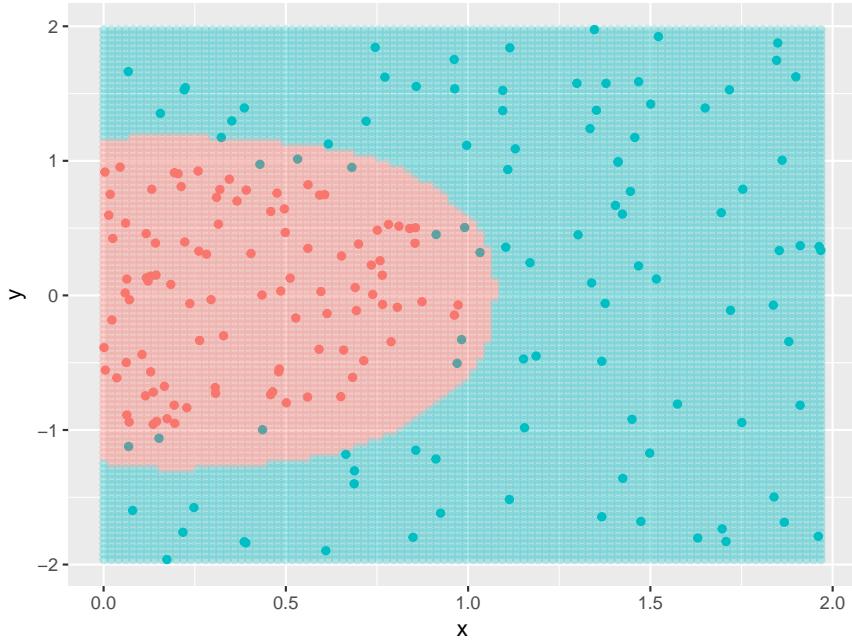
### 34.1.1 Non-parametric Method

As before we can replace the least squares solution with a non-parametric one, say loess:

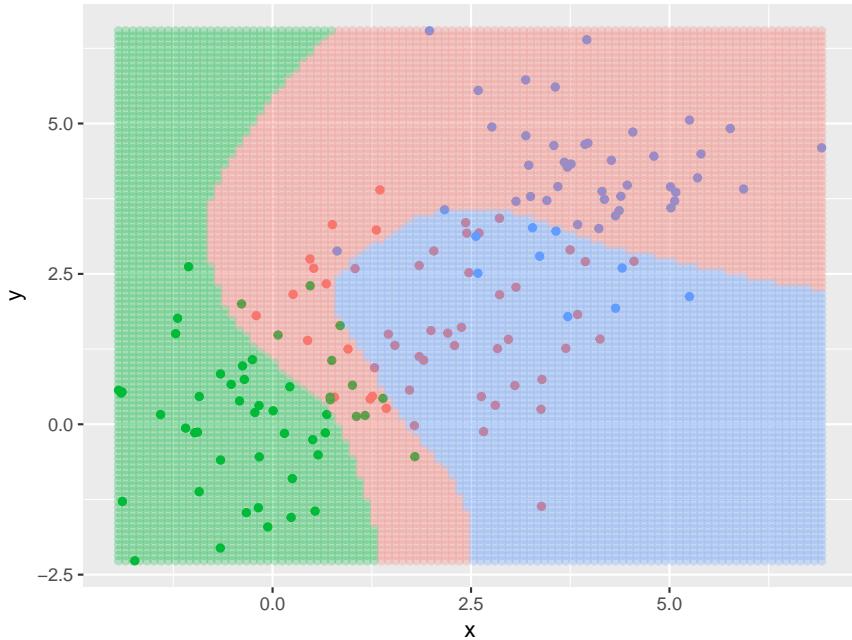
```
dta <- example.data.1()
fit <- loess(z2 ~ x+y, data=dta)
do.graph2(dta, fit)
```



```
dta <- example.data.2()
fit <- loess(z2 ~ x+y, data=dta)
do.graph2(dta, fit)
```



```
dta <- example.data.3()
fit <- loess(z2 ~ x+y, data=dta)
do.graph2(dta, fit)
```



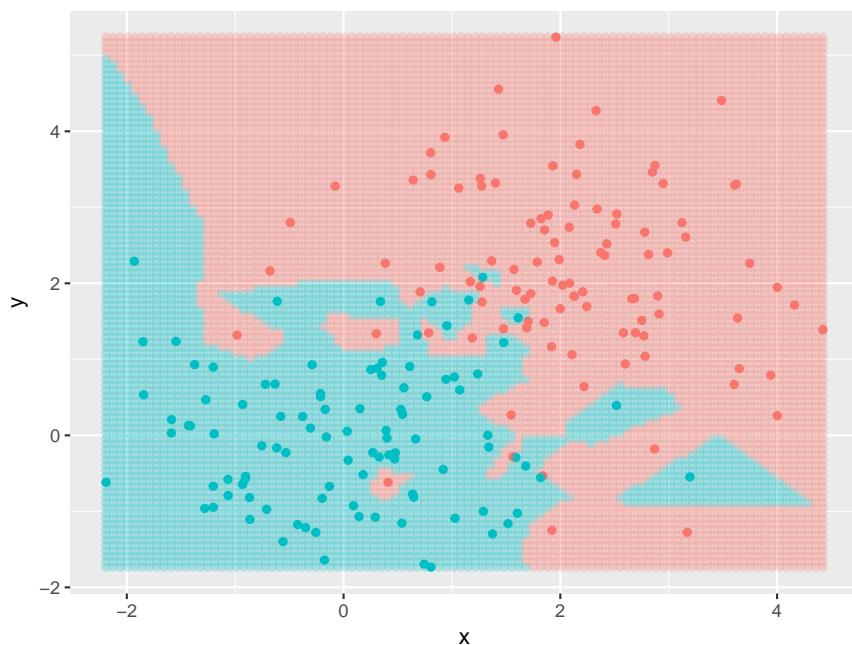
### 34.1.2 k-nearest neighbor

Here is another, completely different way: for a new point  $(x_1, x_2)$  find the point in the dataset that is closest, and classify  $(x_1, x_2)$  the same! This is called the 1-nearest neighbor classifier. It is implemented by the function *knn*, part of the *class* library:

```

library(class)
dta <- example.data.1()
x.grid <- seq(min(dta$x), max(dta$x), length=100)
y.grid <- seq(min(dta$y), max(dta$y), length=100)
xy.grid <- expand.grid(x.grid, y.grid)
dta.predict <- data.frame(x = xy.grid[, 1],
 y = xy.grid[, 2])
zhat <- knn(dta[, 1:2], xy.grid, cl=dta$z1, k=1)
dta.predict$zhat <- zhat
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25)

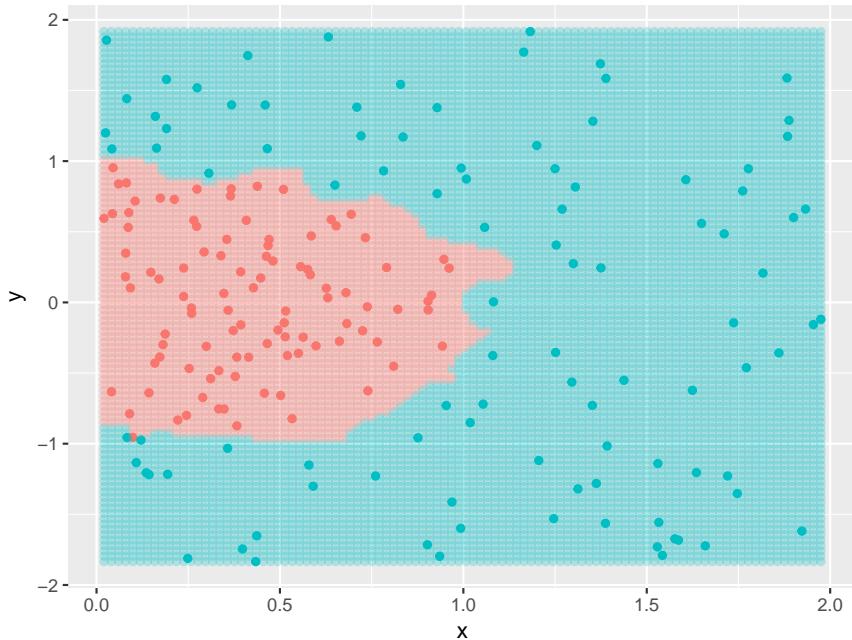
```



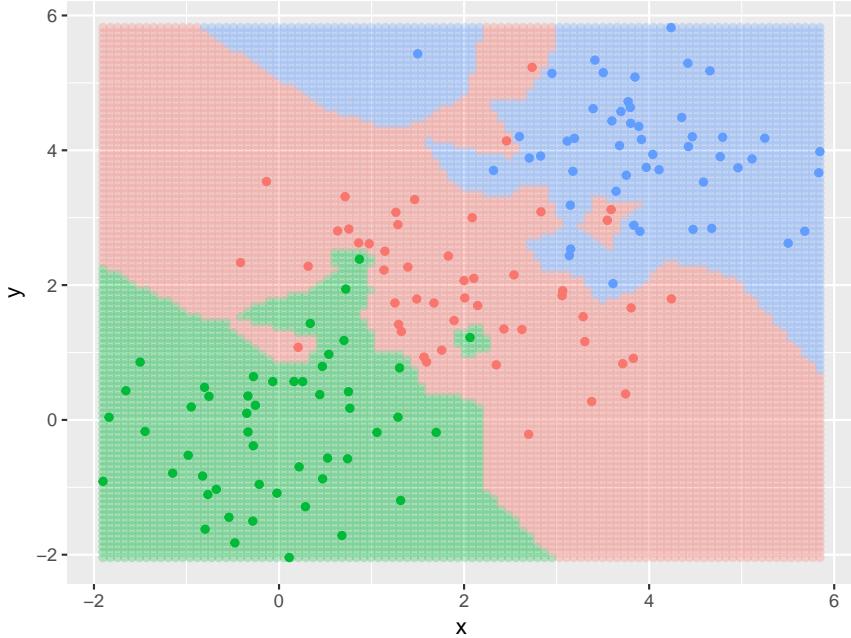
```

dta <- example.data.2()
x.grid <- seq(min(dta$x), max(dta$x), length=100)
y.grid <- seq(min(dta$y), max(dta$y), length=100)
xy.grid <- expand.grid(x.grid, y.grid)
dta.predict <- data.frame(x = xy.grid[, 1],
 y = xy.grid[, 2])
zhat <- knn(dta[, 1:2], xy.grid, cl=dta$z1, k=1)
dta.predict$zhat <- zhat
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25)

```

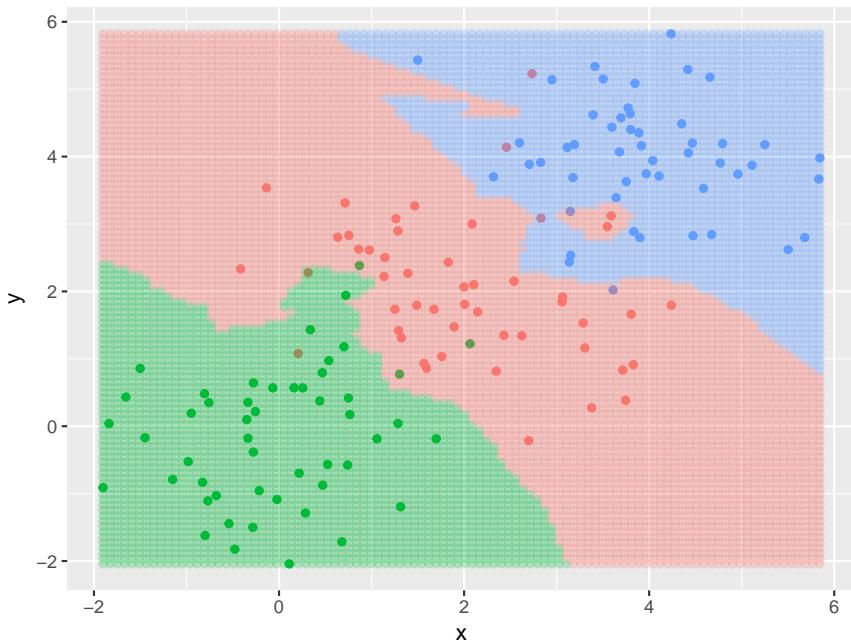


```
dta <- example.data.3()
x.grid <- seq(min(dta$x), max(dta$x), length=100)
y.grid <- seq(min(dta$y), max(dta$y), length=100)
xy.grid <- expand.grid(x.grid, y.grid)
dta.predict <- data.frame(x = xy.grid[, 1],
 y = xy.grid[, 2])
zhat <- knn(dta[, 1:2], xy.grid, cl=dta$z1, k=1)
dta.predict$zhat <- zhat
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25)
```



We can “smooth” the result by considering more than one neighbor. In that case the prediction is done by majority vote: if two out of three are red, prediction is red:

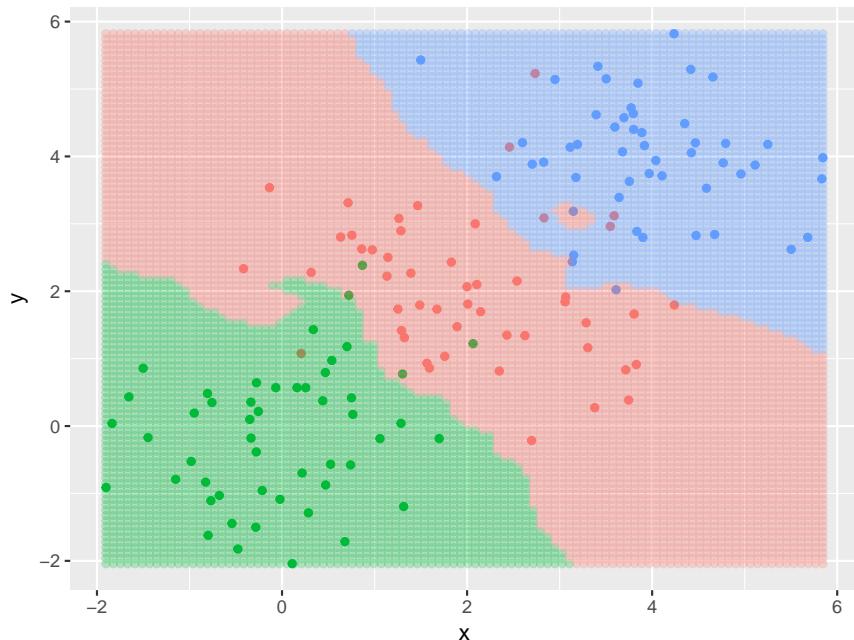
```
zhat <- knn(dta[, 1:2], xy.grid, cl=dta$z1, k=3)
dta.predict$zhat <- zhat
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25)
```



```

zhat <- knn(dta[, 1:2], xy.grid, cl=dta$z1, k=5)
dta.predict$zhat <- zhat
do.graph1(dta) +
 geom_point(data = dta.predict,
 aes(x, y, color=zhat),
 alpha=0.25)

```



Clearly, we should use an odd  $k$ .

### 34.1.3 Neural Networks

Another popular approach is to fit a *neural network*. This is a large topic, and we won't have time for any detailed discussion. Here is just one example:

```

library(nnet)
dta <- example.data.2()
fit <- nnet(x = cbind(dtax, dtay),
 y = dta$z2,
 size = 2)

weights: 9
initial value 50.416977
iter 10 value 25.047802
iter 20 value 18.012011
iter 30 value 11.595345
iter 40 value 9.653016
iter 50 value 9.090629
iter 60 value 8.298239

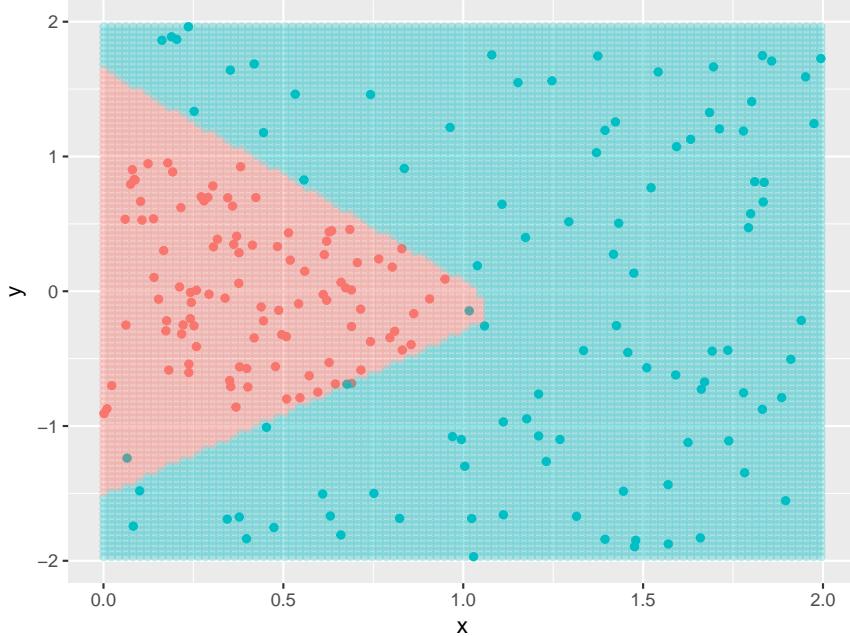
```

```

iter 70 value 6.679673
iter 80 value 4.019072
iter 90 value 3.752627
iter 100 value 2.674251
final value 2.674251
stopped after 100 iterations

do.graph2(dta, fit)

```



#### 34.1.4 Tree based methods

Yet another idea is as follows. Let's define a “distance” between the groups. Now let's take one variable (say  $x$ ) and for some number  $a$  split the data set into two parts  $x < a$  and  $x > a$ . For each  $a$  we get a distance in the two splits. Find  $a$  such that this distance is maximized. Do the same for all the variables. Use the best overall split.

Next we repeat the same procedure within each of the splits defined before. In this way we get smaller and smaller data sets which are more and more homogeneous (aka have the same group).

If we kept on going eventually each observation would be its own split, but instead we will stop at some point before.

There are a number of methods for tree based classification. Here we will use the library *rpart*, which implements recursive partitioning.

Let's start with an example

```

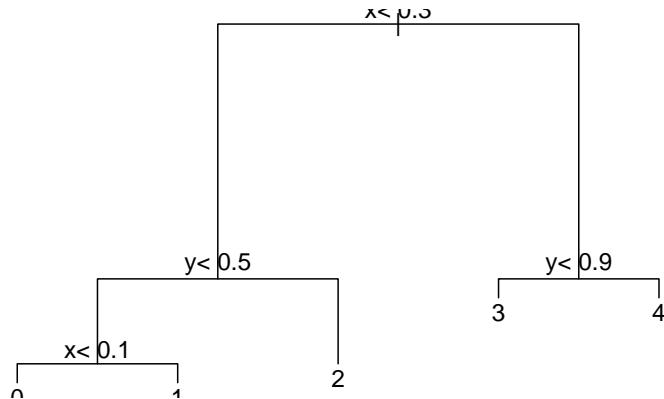
library(rpart)
x <- rep(c(0.05, 0.15, 0.05, 0.15, 0.45, 0.45), 10)
y <- rep(c(0.25, 0.25, 0.75, 0.75, 0.85, 0.95), 10)

```

```

z <- rep(c(0, 1, 2, 2, 3, 4), 10)
fit <- rpart(z ~ x + y)
plot(fit)
text(fit)

```



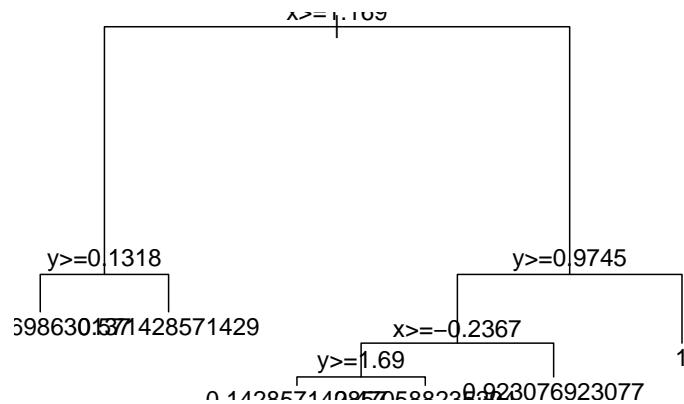
so the first split is for  $x < 0.3$ . For those observations with  $x < 0.3$  the next split is for  $y < 0.5$  etc.

Here is what this looks like for our examples:

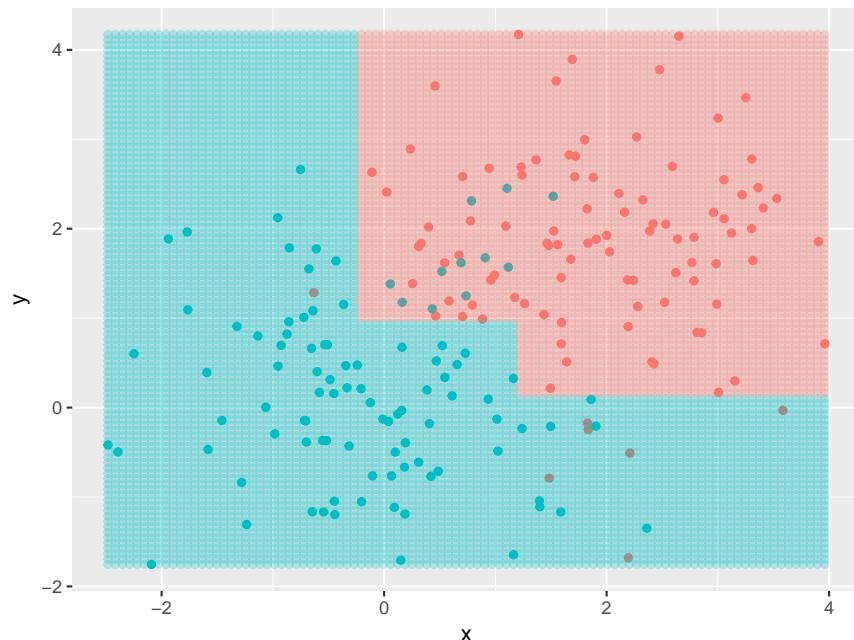
```

dta <- example.data.1()
fit <- rpart(z2 ~ x + y, data=dta)
plot(fit)
text(fit)

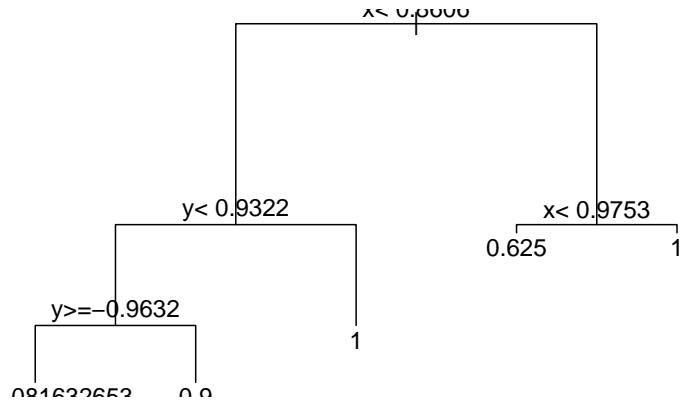
```



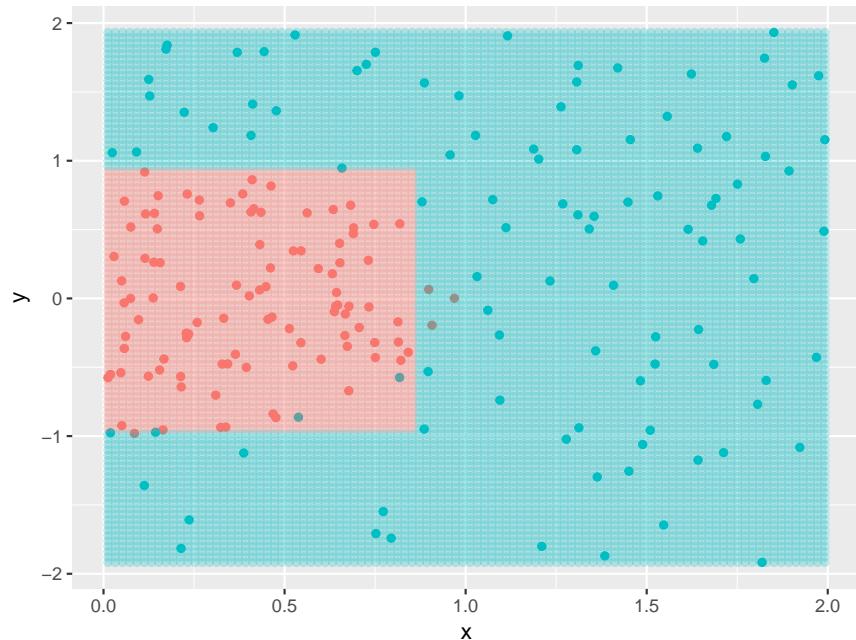
```
do.graph2(dta, fit)
```



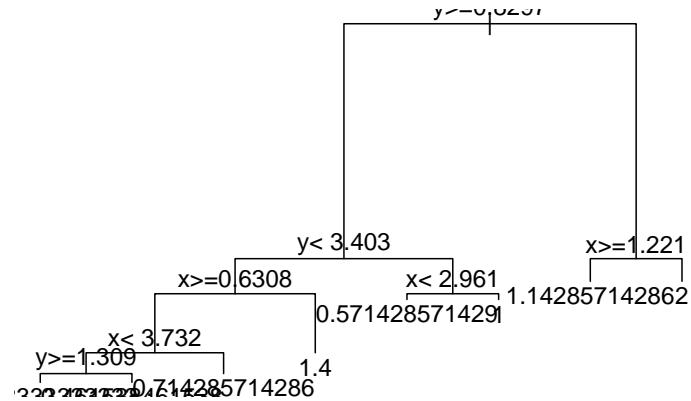
```
dta <- example.data.2()
fit <- rpart(z2 ~ x + y, data=dta)
plot(fit)
text(fit)
```



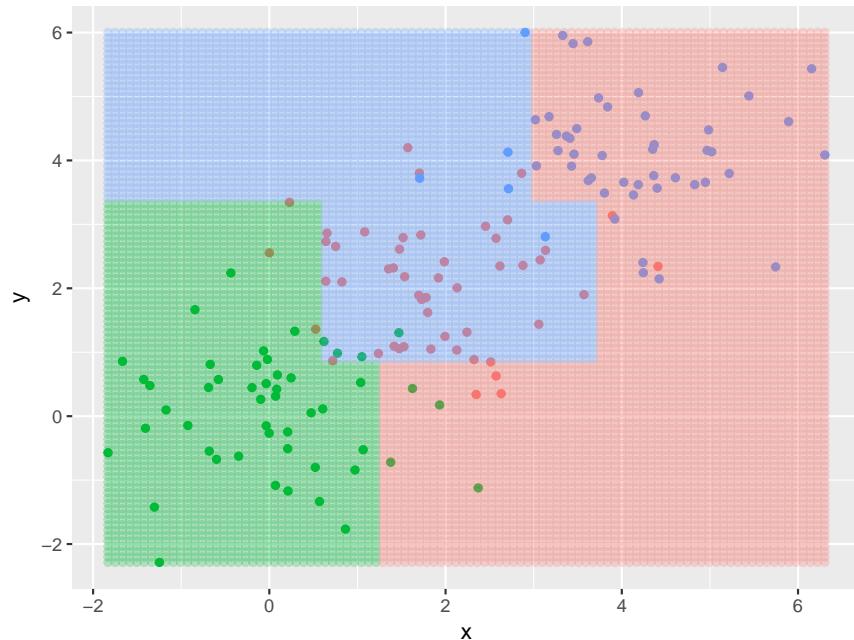
```
do.graph2(dta, fit)
```



```
dta <- example.data.3()
fit <- rpart(z2 ~ x + y, data=dta)
plot(fit)
text(fit)
```



```
do.graph2(dta, fit)
```



Tree based methods using advanced methods such as bagging, boosting, random forests etc. can be very effective, they are however often difficult to interpret.

### 34.2 Misclassification Rate

How can we choose among all of those methods? The most common measure of how good a classifier is is the *misclassification rate*, that is how many observations would have been

classified wrong. Let's find this rate for the various methods:

```
dta <- example.data.1(n=1000)
```

- LDA

```
fit <- lm(z2~x+y, data=dta)
zhat <- predict(fit)
zhat <- ifelse(zhat<0.5, "Blue", "Red")
tbl <- table(dta$z1, zhat)
tbl

zhat
Blue Red
Blue 915 85
Red 77 923
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc
```

```
[1] 0.081
```

- Loess

```
fit <- loess(z2~x+y, data=dta)
zhat <- predict(fit)
zhat <- ifelse(zhat<0.5, "Blue", "Red")
tbl <- table(dta$z1, zhat)
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc
```

```
[1] 0.0865
```

- 1-nearest neighbor

```
zhat <- knn(dta[, 1:2], dta[, 1:2], cl=dta$z1, k=1)
tbl <- table(dta$z1, zhat)
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc
```

```
[1] 0
```

- 3-nearest neighbor

```
zhat <- knn(dta[, 1:2], dta[, 1:2], cl=dta$z1, k=3)
tbl <- table(dta$z1, zhat)
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc
```

```
[1] 0.061
```

- neural network

```

fit <- nnet(x = cbind(dtax, dtay),
 y = dta$z2,
 size = 2)

weights: 9
initial value 493.332245
iter 10 value 124.544901
iter 20 value 121.398654
iter 30 value 120.905300
iter 40 value 120.489294
iter 50 value 119.182951
iter 60 value 119.069161
final value 119.069015
converged

zhat <- predict(fit)
zhat <- ifelse(zhat<0.5, "Blue", "Red")
tbl <- table(dta$z1, zhat)
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc

[1] 0.082

• trees

fit <- rpart(z2 ~ x + y, data=dta)
zhat <- predict(fit)
zhat <- ifelse(zhat<0.5, "Blue", "Red")
tbl <- table(dta$z1, zhat)
msc <- (tbl[1, 2]+tbl[2, 1])/sum(tbl)
msc

[1] 0.077

```

So 1-nearest neighbor wins, with misc=0!

Or does it? The problem is that we evaluated the methods on the same data set that we used for fitting (or “training”). In the case of 1-nearest neighbor that means we never get this wrong. But if we then applied the fit to a new data set we likely would do much worse. This is true to a lesser extent for all the methods, and in real life we should always use different data sets for training and for evaluation. If only one data set is available we can again use the idea of cross-validation.

### 34.3 Fisher’s iris data

Let’s apply all of these methods to Fisher’s iris data

- LDA

```

species.code <- rep(0, 150)
species.code[51:100] <- 1
species.code[101:150] <- 2
iris1 <- iris
iris1$Species <- species.code
fit <- lm(Species~., data=iris1)
tmp <- predict(fit)
zhat <- ifelse(tmp<2/3, 0, 1)
zhat[tmp>4/3] <- 2
tbl <- table(species.code, zhat)
(150-sum(diag(tbl)))/150

[1] 0.06666666666666667

• Loess

fit <- loess(Species~., data=iris1)
tmp <- predict(fit)
zhat <- ifelse(tmp<2/3, 0, 1)
zhat[tmp>4/3] <- 2
tbl <- table(species.code, zhat)
(150-sum(diag(tbl)))/150

[1] 0.09333333333333333

• 5-nearest neighbor

zhat <- knn(iris[, 1:4],
 iris[, 1:4],
 cl = iris$Species, k=5)
tbl <- table(iris$Species, zhat)
(150-sum(diag(tbl)))/150

[1] 0.03333333333333333

• neural net

targets <- class.ind(c(rep(0, 50), rep(1, 50), rep(2, 50)))
fit <- nnet(x = iris1[, 1:4],
 y = targets,
 size = 2)

weights: 19
initial value 118.302238
iter 10 value 57.506863
iter 20 value 49.125181
iter 30 value 39.155967
iter 40 value 35.030671
iter 50 value 33.583669
iter 60 value 33.119886

```

```

iter 70 value 31.986357
iter 80 value 30.216743
iter 90 value 29.781578
iter 100 value 29.774892
final value 29.774892
stopped after 100 iterations

tmp <- round(predict(fit))
zhat <- ifelse(tmp[, 1]==1, 0, 1)
zhat[tmp[, 3]==1] <- 2
tbl <- table(species.code, zhat)
(150-sum(diag(tbl)))/150

[1] 0.02
• trees

fit <- rpart(Species ~ ., data=iris1)
tmp <- predict(fit)
zhat <- ifelse(tmp<2/3, 0, 1)
zhat[tmp>4/3] <- 2
tbl <- table(species.code, zhat)
(150-sum(diag(tbl)))/150

[1] 0.04

```