ESMA 6835: Computing with R

Dr. Wolfgang Rolke

August 12, 2018

Contents

Resma3.RData (Ver 3.1)

1 Getting Started

1.1 Installation and Updating

1.1.1 Base R

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.

1.1.2 RStudio

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

1.1.3 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)
```

1.2 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.

1.2.1 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). Everything between the three dashes (which are needed) is YAML code. 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 have a file called blank.Rmd which already has everything as I (usually) want it. All I need to do is rename it and put it in the right folder.

1.2.2 Basic R Markdown Syntax

Markdown has simple keyboard shortcuts for many basic editing features. For example, #, ##, ### are for chapter and section headers. Subscripts are done with beginning and ending ~, so for X_1 you need to type X~1~. For superscripts use X^1^.

For a complete list of the basic syntax go to https://rmarkdown.rstudio.com/articles_intro.html or to https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

1.2.3 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}
“‘

Here ‘ is a back tick, on Windows keyboards found on the upper left key, below ~.

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] 0.1938
```

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 *eval*uating). This is useful if the code takes along time to run, or if you want to show code that is actually faulty, or for any number of other reasons.

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 normal 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)

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.

b. inline code.

here is a bit of text:

and the mean was 0.1938.

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

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

1.2.4 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.

1.2.4.1 HTML, Latex(Pd), Word, PowerPoint etc.

One of the great features of Markdown is that its syntax is independent of the eventual document format, so the same markdown file can immediately produce an HTML file of a pdf or or...

In this class we will only use the HTML format, which is the easiest.

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.

1.2.5 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-p

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

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

Gender	Age
Male	20
Male	21
Female	19

I have written my own kable routine which improves a bit on the basic version:

```
kable.nice <- function (x,
      do.row.names = TRUE,
      col.names = NA, font.size = 15)
{
  library(tidyverse)
  library(kableExtra)
  kable(x, row.names = do.row.names,
        col.names = col.names) %>%
```

```
        kable_styling(bootstrap_options = "striped",
            full_width = FALSE,
            font_size = font.size)
}
kable.nice(df)
```

	Gender	Age
1	Male	20
2	Male	21
3	Female	19

which I am sure you agree is nice! You can use it yourself, just copy paste the function code into an R chunk of your document.

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>
```

It 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!

So what do you do if you don't know yet what the output will be, or if you want your routine to produce nice output either way? The solution is this: 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 , then the text and finally 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")
# this figures out what the output format is
  if(output.format == "latex")
    out <- paste0("\\textcolor{red}{", txt, "}")
  else
    out <- paste0("<font color='red'>", txt, "</font>")
  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.

1.2.6 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 word processor 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.

Latex has a HUGE list of symbols for just about anything. A nice list of common symbols is found on https://artofproblemsolving.com/wiki/index.php/LaTeX:Symbols. Often when I need one I don't remember I just google it. For example, say I want to use the symbol for the real numbers: $\mathbf{R}$. So I google "latex real numbers symbol", and the first document tells me the code is \mathbf{R}!

Latex code is usually used in two ways: as part of a sentence or as stand-alone. In the first case use a single dollar sign at the beginning and the end. For example the code

We want to integrate the function

```
$f(x)=\exp(-x^2)$
```

will display as

We want to integrate the function

$f(x) = \exp(-x^2)$.

Exercise

Does anyone know how I displayed the code and not the formula the first time?

The other way to display math in latex is via

1.2.6.1 Multiline math

say you want the following in your document:

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

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

```
##  $$
##  \begin{aligned}
##  &E[X] = \int_{-\infty}^{\infty} xf(x) dx=\\
##  &\int_{0}^{1} x dx    = \frac12 x^2 |_0^1 = \frac12
##  \end{aligned}
##  $$
```

so a multiline expression starts and ends with double dollar signs.

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!

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

1.3 History, Basic Usage

1.3.1 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 lead to

1.3.2 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.

1.3.3 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. Considering that it is only about three years old it has taken the R world by storm!

1.3.4 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] 136
```

shows there are 136 of them. Two you might want to remember are

```
options()["digits"]
```

```
## $digits
## [1] 4
```

```
pi
```

```
## [1] 3.142
```

```
options(digits=5)
pi
```

```
## [1] 3.1416
```

```
options(digits=15)
pi
```

```
## [1] 3.14159265358979
```

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)
```

Changing one or more of these options is something one needs to do on occasion inside a function. If so you likely want to reset them to the defaults before leaving the function. Say we have a program that prints results to the screen and we want all the numbers to have 4 digits. Then we can use

```
# At start of function:
default.options <- options()
options(digits=4)
# now do stuff
# then at the end of the program
options(default.options)
```

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. If you do need to create a .RData file (maybe to send stuff to someone else) use

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

Notice the use of the backslash as a separator for folders. On Windows systems one usually uses forward slashes ("\") but these have a special use in R as the "escape character". One alternative is to use

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

but / is generally preferred.

1.3.6 Assignments

The basic assignment character in R is <-

```
x <- 1
x
```

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

Note think of an arrow pointing left

Note = also works:

```
x = 2
x
```

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

but mainly for reasons of backward compatibility <- is generally recommended.

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.

1.3.7 Help!

1.3.7.1 Help inside of R

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

```
?mean
```

This opens a page in the Help pane with all sorts of information regarding the mean command.

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.legends = 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
```

1.3.7.2 Help from Outside of R

R has one of the most active and friendly user communities anywhere. So if you run into a problem do this:

1.3.7.3 Google!

Most of the problems you have, someone else already had. They ended up asking for help on the internet, and someone replied. So all you need to do is find that reply. Start with a google search. Typically you want to include R in your search terms. If at first you don't succeed, try, try again!

1.3.7.4 Stackoverflow

Many times your google search will lead you to stackoverflow. This is a website for people working on computer coding problems to help each other. It is not specific to R but the R community is using it to discuss and solve R questions.

If you can not find a solution to your problem, eventually you might want to post a question on stackoverflow. To do so you have to become a member, which is free.

Here are some guidelines for posting questions:

- did you really try hard enough to find an answer on your own? It is very embarrassing when you post a question and the reply is a link to a simple Google search that has the answer already.
- try and eliminate anything that does not have to do with your problem.

- produce a simple and short example of your problem. Anything over 5 or 10 lines of code is almost certainly to long.

1.3.7.5 ls, rm

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

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

```
## [1] "%!in%"    "acorn"    "Age"      "agesex"   "agesexUS"
```

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 ^ (called "caret") in front of sa means that *ls* will list all objects whose name starts with sa. This is what is called a *regular expression*. These are general (non R) rules for specifying patters. 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)

}
```

- the things in oldstuff (like f and x) are the names I generally use for stuff that I expect to need only for a short time.
- 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.

1.3.8 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.

2 Base R

2.1 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 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
```

The real and the imaginary parts are chosen with

```
Re(v)
```

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

```
Im(v)
```

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

Two other standard functions for complex numbers are

- complex conjugate

$$z = x + iy$$
$$\bar{z} = x - iy$$

```
v
```

```
## [1] 1-1i
```

```
Conj(v)
```

```
## [1] 1+1i
```

- Modulus:

$$z = x + iy$$
$$\text{Modulus} = \sqrt{x^2 + y^2}$$

```
Mod(1+1i)
```

```
## [1] 1.414
```

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\""
```

2.1.1 Vectors

the basic data unit of R is a vector. One can create a vector with the *c*ombine 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] "2019-07-15 14:31:18 -04"
```

```
Sys.Date()
```

```
## [1] "2019-07-15"
```

2.1.2 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

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

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

```r
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!

Exercise

Before running these in R, try and think about the answer:

What is the result of

```r
c(1, FALSE)
c("A", FALSE)
c(1L, FALSE)
-1 < FALSE
1 == "1"
```

2.1.3 Dates

The default format is yyyy-mm-dd:

```r
mydates <- as.Date(c("2018-01-01", "2018-06-13"))
mydates
```

```
## [1] "2018-01-01" "2018-06-13"
```

```r
mydates[2]-mydates[1]
```

```
## Time difference of 163 days
```

2.1.4 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:

```r
students
```

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

Let's count how many of each we have:

```r
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"
##  [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 list 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
```

Exercise

```
ltrs <-  c("a", "b", "b", "c", "c", "c")
f1 <- factor(ltrs)
f1
```

```
## [1] a b b c c c
## Levels: a b c
```

```
levels(f1) <- c("c", "b", "a")
```

What does f1 now look like?

2.2 Data Formats

2.2.1 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

2.2.2 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, x)
```

```
##      x y x
## [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)
x
```

```
##      [,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.

Sometimes a matrix is created by a function and given row and column names automatically, but we don't want it to have them. They can be removed with

```
rownames(x) <- NULL
x
```

```
##      Height Age.1 Age.2
## [1,]      1     3     5
## [2,]      2     4     6
```

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
```

Why does this work? dim is an attribute of an object, by changing this attribute we are changing the object.

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 behavior is called *recycling* (in computer science, not just R)

In either case R gives a warning. Except in rare cases you should try to avoid these things, they are usually the consequence of bad programming.

2.2.3 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)
xyz <- table(x,y,z)
xyz
```

```
## , , z = a
##
##    y
## x   F M
##   1 2 1
##   2 0 0
##   3 0 1
##
## , , z = b
##
##    y
## x   F M
##   1 1 4
##   2 1 2
##   3 0 3
##
## , , z = c
##
##    y
## x   F M
##   1 1 1
##   2 2 1
##   3 0 0
```

The commands for arrays are similar to those for matrices:

```
dim(xyz)
```

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

```
dimnames(xyz)
```

```
## $x
## [1] "1" "2" "3"
##
## $y
## [1] "F" "M"
##
## $z
## [1] "a" "b" "c"
```

2.2.4 Data Frames

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
```

This type of data is very common in Statistics, and data frames have been the standard data type from its beginning. In general when you get data from a source like the internet it will be as a data frame.

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"
```

Say we want to add another column (variable) to the data frame:

```
xyz[[4]] <- (1:6)^2
colnames(xyz)[4] <- "squares"
xyz
```

```
##   a b     c squares
## a 1 a  TRUE       1
## b 2 a  TRUE       4
## c 3 b FALSE       9
## d 4 c  TRUE      16
## e 5 a FALSE      25
## f 6 c  TRUE      36
```

If we want to get rid of a column:

```
xyz[[4]] <- NULL
```

There is a strange default behavior of data frames: they turn strings into factors:

```
df <- data.frame(x=1:5, y=letters[1:5])
str(df)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ x: int  1 2 3 4 5
##  $ y: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
```

You can prevent this from happening, though:

```
df <- data.frame(x = 1:5,
                 y = letters[1:5],
                 stringsAsFactors = FALSE)
str(df)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ x: int  1 2 3 4 5
##  $ y: chr  "a" "b" "c" "d" ...
```

In fact, this is what I want to happen almost all the time. So I change that option globally whenever I run R or RStudio. I will show you how to do that at some point.

If a data frame is all of the same type we can use *as.matrix* to turn it into a matrix.

Exercise

what does as.matrix() do when it is applied to a data frame with columns of different types?

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

2.2.5 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
```

Often we want to use a list inside a function to record various values. So we need to create an "empty" list of a certain length:

```
x <- as.list(1:3)
x
```

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

and if we run out of space:

```
x[[4]] <- "a"
x
```

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

Internally R stores all data as lists. In fact anything can be an element of a list:

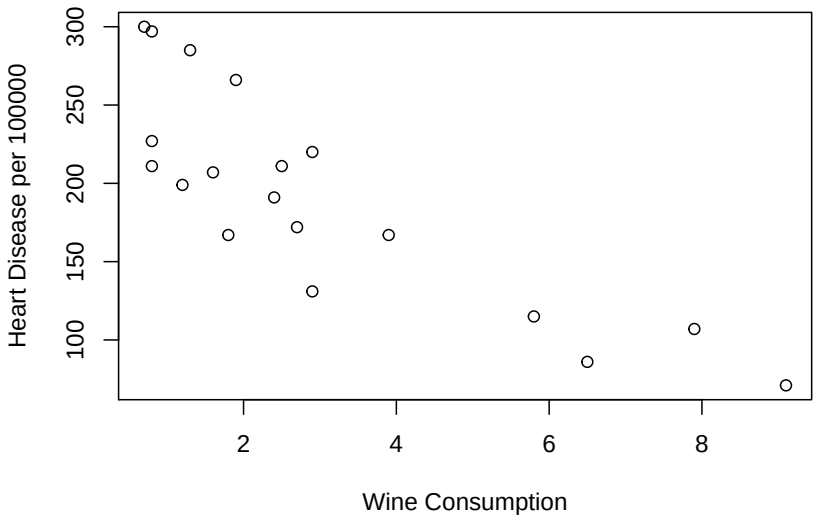
2.2.5.1 Example

In Statistics we often have data of the form (x, y) and we want to find a *linear model*. For example consider the data on *wine consumption and life expectancy*:

```
kable.nice(wine)
```

	Country	Wine.Consumption	Heart.Disease.Deaths
1	Australia	2.5	211
2	Austria	3.9	167
3	Belgium	2.9	131
4	Canada	2.4	191
5	Denmark	2.9	220
6	Finland	0.8	297
7	France	9.1	71
8	Iceland	0.8	211
9	Ireland	0.7	300
10	Italy	7.9	107
11	Netherlands	1.8	167
12	New Zealand	1.9	266
13	Norway	0.8	227
14	Spain	6.5	86
15	Sweden	1.6	207
16	Switzerland	5.8	115
17	United Kingdom	1.3	285
18	United States	1.2	199
19	Germany	2.7	172

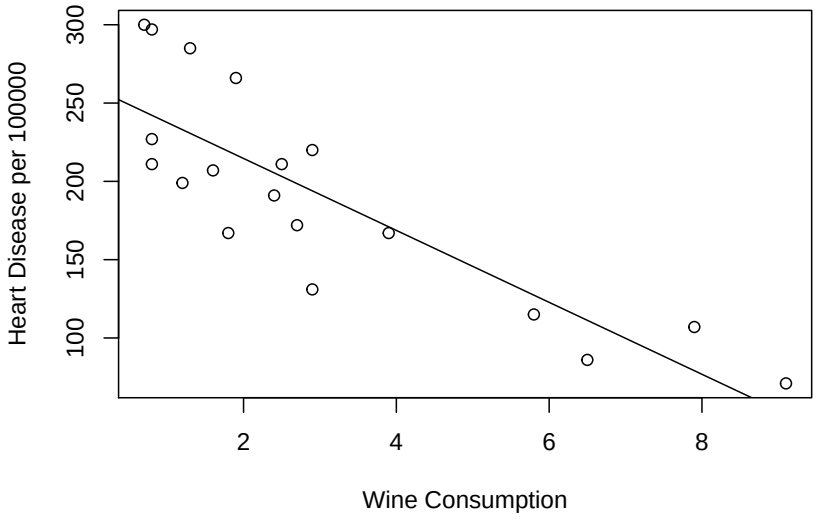
```
attach(wine)
plot(Wine.Consumption, Heart.Disease.Deaths,
      xlab = "Wine Consumption",
      ylab = "Heart Disease per 100000")
```



Note The data set wine and the routine splot are part of a large collection of data sets and routines that I use in many of my courses. You can download the file at http://academic.uprm.edu/wrolke/Resma3/Resma3.RData.

Let's say we want to fit a linear model:

```
fit <- lm(Heart.Disease.Deaths~Wine.Consumption)
plot(Wine.Consumption, Heart.Disease.Deaths,
      xlab = "Wine Consumption",
      ylab = "Heart Disease per 100000")
abline(fit)
```



Let's say we want to save the data and the fit for future use:

```
wine.all <- list(data=wine, fit=fit)
length(wine.all)
```

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

data frames are lists, with the additional requirement that each column has the same length. But a "column" need not be what you think:

```
z <- list(fit, fit, fit)
length(z)
```

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

```
df <- data.frame(x=1:3, y=letters[1:3])
df$z <- z
dim(df)
```

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

This works, but I don't recommend it. A list is likely a better option here.

Example

Say we have the following data set: in each of 10 experiments 5 measurements in five different locations coded as A-E were taken. The result was stored as a list, with each set of measurements an element:

```
results
```

```
## $`Experiment 1`
##     A     B     C     D     E
## 124.8  94.4 135.2 111.2  90.9
##
## $`Experiment 2`
##     A     B     C     D     E
##  83.4  76.7  78.7  68.7 123.1
##
## $`Experiment 3`
##     A     B     C     D     E
## 116.6  95.5 105.3  92.5 148.8
##
## $`Experiment 4`
##     A     B     C     D     E
##  84.1  98.9 105.0 112.4  96.5
##
## $`Experiment 5`
##     A     B     C     D     E
##  55.5  74.7 107.2  99.8  81.2
##
## $`Experiment 6`
```

```
##     A     B     C     D     E
##  97.7  83.7 104.8  71.5 107.3
##
## $`Experiment 7`
##     A     B     C     D     E
## 105.0 101.3 100.4 105.1  87.0
##
## $`Experiment 8`
##     A     B     C     D     E
##  97.6 113.3 122.0 102.9  97.6
##
## $`Experiment 9`
##     A     B     C     D     E
##  81.8  71.2  84.1 125.1 115.4
##
## $`Experiment 10`
##     A     B     C     D     E
##  95.6  91.5  91.6 119.9  94.5
```

Now we want to find the means by locations. One way would be to loop over the list elements, but much easier is:

```
Reduce(`+`, results)/length(results)
```

```
##      A      B      C      D      E
##  94.21  90.12 103.43 100.91 104.23
```

2.3 Generating Objects

2.3.1 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 regular sequences use ":" (read "to")

```
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]  0.0000  0.5263  1.0526  1.5789  2.1053  2.6316  3.1579  3.6842
##  [9]  4.2105  4.7368  5.2632  5.7895  6.3158  6.8421  7.3684  7.8947
## [17]  8.4211  8.9474  9.4737 10.0000
```

- 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 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" "C" "C" "C"
```

```
rep(1:3, rep(5, 3))
```

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

Exercise

What does this do?

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

- 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 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
```

- 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.

- outer

This calculates the outer product of two vectors. It creates a matrix of length(x) x length(y) where $z_{ij} = f(x_i, y_j)$ for some function f

```
x <- 1:3
y <- 1:5
outer(x, y, "*")
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

You can use any function you like:

```
outer(x, y, function(x,y) {x^2/y})
```

```
##      [,1] [,2]   [,3] [,4] [,5]
## [1,]    1  0.5 0.3333 0.25  0.2
## [2,]    4  2.0 1.3333 1.00  0.8
## [3,]    9  4.5 3.0000 2.25  1.8
```

If it is actual multiplication you want there is also a short hand:

```
x %o% y
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

Want to practice your multiplication tables?

```
x <- 1:10
names(x) <- x
x %o% x
```

```
##     1  2  3  4  5  6  7  8  9  10
## 1   1  2  3  4  5  6  7  8  9  10
## 2   2  4  6  8 10 12 14 16 18  20
## 3   3  6  9 12 15 18 21 24 27  30
## 4   4  8 12 16 20 24 28 32 36  40
```

```
## 5   5 10 15 20 25 30 35 40 45  50
## 6   6 12 18 24 30 36 42 48 54  60
## 7   7 14 21 28 35 42 49 56 63  70
## 8   8 16 24 32 40 48 56 64 72  80
## 9   9 18 27 36 45 54 63 72 81  90
## 10 10 20 30 40 50 60 70 80 90 100
```

Exercise

What (if anything) does this do?

```
(1:3 %o% 1:3) %o% 1:2
```

2.3.2 Specialty Data

Some common objects are easy to create:

```
letters
```

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

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

2.3.3 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)))
```

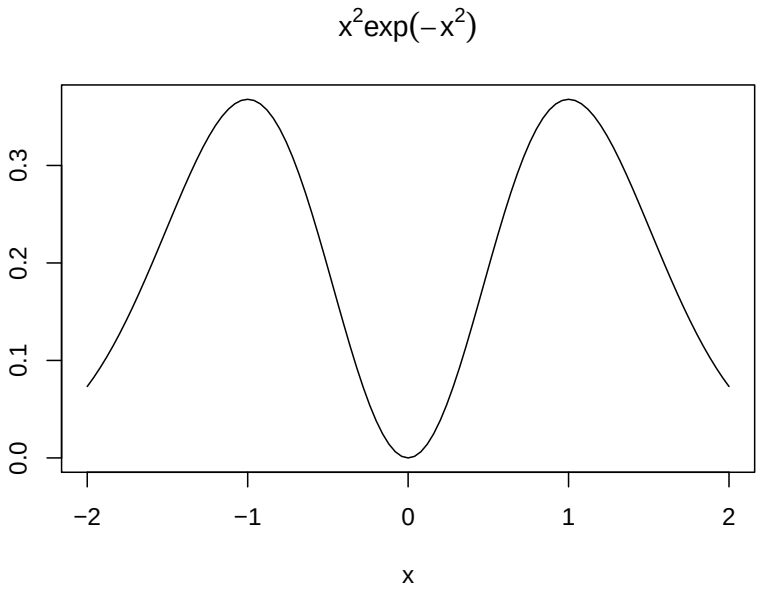
```
eval(exp1)
```

```
## [1] 0.5749
```

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 = "")
```



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] 0.1916
```

```
D(exp1, "y")
```

```
## -(x/(y + exp(z))^2)
```

```
eval(D(exp1, "y"))
```

```
## [1] -0.1102
```

```
D(exp1, "z")
```

```
## -(x * exp(z)/(y + exp(z))^2)
```

```
eval(D(exp1, "z"))
```

```
## [1] -0.2995
```

Exercise

Use the D function to find $\frac{d^2}{dx^2}\frac{x^2}{1+x^3}\big|_{x=0}$

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.

2.4 Sub-setting / Data Wrangling

2.4.1 Vectors

Consider the following vector:

```
x
```

```
##   A   B   C   D   E   F   G   H   I   J
## 9.0 6.4 7.4 6.1 9.0 2.9 1.9 8.9 5.0 8.8
```

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

```
x[3]
```

```
##   C
## 7.4
```

```
x[1:3]
```

```
##   A   B   C
## 9.0 6.4 7.4
```

```
x[c(1, 3, 8)]
```

```
##   A   C   H
## 9.0 7.4 8.9
```

```
x[-3]
```

```
##   A   B   D   E   F   G   H   I   J
## 9.0 6.4 6.1 9.0 2.9 1.9 8.9 5.0 8.8
```

```
x[-c(1, 2, 5)]
```

```
##   C   D   F   G   H   I   J
## 7.4 6.1 2.9 1.9 8.9 5.0 8.8
```

if a vector has names they can be used as well:

```
x["C"]
```

```
##   C
## 7.4
```

```
x[c("A","D")]
```

```
##   A   D
## 9.0 6.1
```

There are also strange things one can do and sometimes get away with:

```r
x <- 1:10
names(x) <- letters[1:10]
x
```

```
##  a  b  c  d  e  f  g  h  i  j
##  1  2  3  4  5  6  7  8  9 10
```

```r
x[0]
```

```
## named integer(0)
```

```r
x[3.1]
```

```
## c
## 3
```

```r
x[3.6]
```

```
## c
## 3
```

```r
x[a]
```

```
## a
## 1
```

Another way to subset a vector is with logical conditions:

```r
x[x > 4]
```

```
##  e  f  g  h  i  j
##  5  6  7  8  9 10
```

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

```
## e f
## 5 6
```

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

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

```
##  a  b  c  d  e  f  g  h  i  j
##  0  2  3  4  5  6  7  8  9 10
```

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

```r
Gender <- sample(c("Male", "Female"),
                 size = 10,
                 replace = TRUE)
Gender
```

```
##  [1] "Male"   "Female" "Male"   "Female" "Male"   "Male"   "Female"
##  [8] "Male"   "Female" "Female"
```

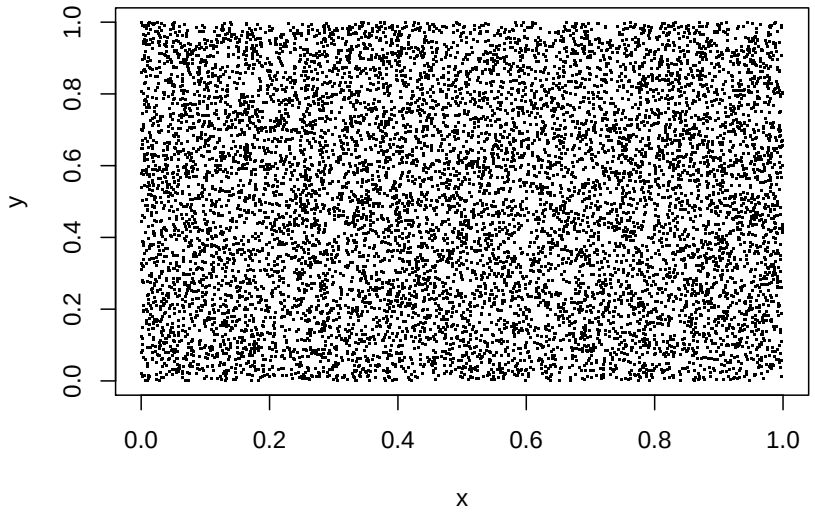
```
GenderCode <- rep(0, length(Gender))
GenderCode[Gender=="Male"] <- 1
GenderCode
```

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

Exercise

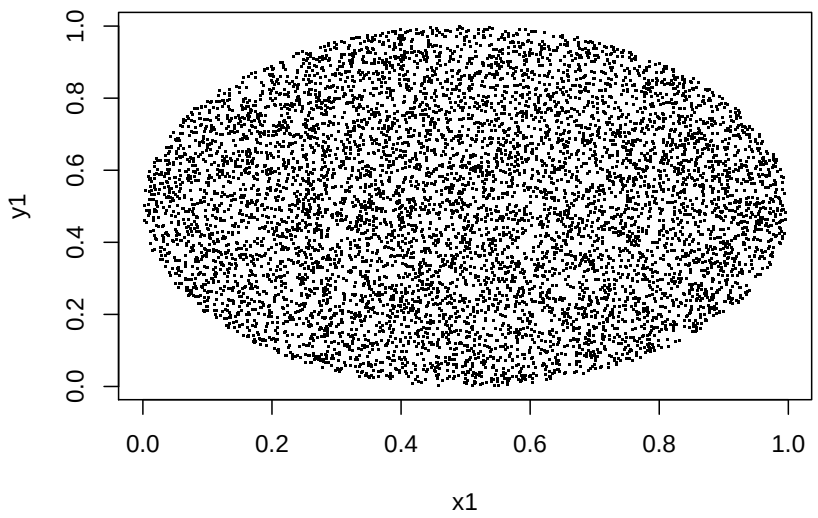
Say we have vectors x and y with the coordinates of points:

```
x <- runif(10000)
y <- runif(10000)
plot(x, y, pch=".")
```



subset x and y in such a way that only points in the circle are left:

```
plot(x1, y1, pch=".")
```



2.4.2 Matrices and Data Frames

Consider the following data frame:

```
students
```

```
##    Age GPA Gender
## 1   23 3.1 Female
## 2   20 3.2   Male
## 3   21 2.1 Female
## 4   20 2.1   Male
## 5   24 2.3 Female
## 6   18 2.9   Male
## 7   20 2.3   Male
## 8   22 3.9   Male
## 9   20 2.6   Male
## 10  21 3.2   Male
```

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

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

```
## [1] Male
## Levels: Female Male
```

There are a variety of ways to do sub-setting:

```
students[, 1]
```

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

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

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

```
students$Age
```

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

And yet another way to do this:

```
attach(students)
```

```
Age
```

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

Exercise

What does this do?

```
x <- 1:10
x[]
```

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

```
students[, 1]
```

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

```
students[1]
```

```
##    Age
## 1   23
## 2   20
## 3   21
## 4   20
## 5   24
## 6   18
## 7   20
## 8   22
## 9   20
## 10  21
```

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

```
##  [1] 23 20 21 20 24 18 20 22 20 21
```

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   23
## 2   20
## 3   21
## 4   20
## 5   24
## 6   18
## 7   20
## 8   22
## 9   20
## 10  21
```

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

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

```
## [1] 23 20 21
```

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

```
##    Age GPA Gender
## 1   23 3.1 Female
## 3   21 2.1 Female
## 4   20 2.1   Male
## 5   24 2.3 Female
## 6   18 2.9   Male
## 7   20 2.3   Male
## 8   22 3.9   Male
## 9   20 2.6   Male
## 10  21 3.2   Male
```

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

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

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

```
##    Age GPA Gender
## 1   23 3.1 Female
## 3   21 2.1 Female
## 5   24 2.3 Female
## 8   22 3.9   Male
## 10  21 3.2   Male
```

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

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

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

is fine but

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

does not work.

Exercise

Subset students so that only females over 21 with a GPA of at least 3.0 are left.

```
##   Age GPA Gender
## 8  22 3.9   Male
```

2.4.3 Lists

Sub-setting of lists is very similar to data frames:

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

```
## $First
## [1] 1 2 3 4 5
##
## $Second
## [1] "A" "B" "C" "D" "E" "F" "G" "H"
##
## $Third
## [1] 20 21 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
```

This can be quite confusing. Here is useful memory device

If list x is a train, x[[5]] is the content of car 5, whereas x[4:5] is the train consisting of cars 4 and 5

2.4.4 Partial Matching

There is one important difference between $ and [[]], the first one allows *partial matching*:

```
x <- list(inside=1:3, outside=5:10)
x$o
```

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

```
x[[o]]
```

```
## Error in eval(expr, envir, enclos): object 'o' not found
```

I don't however recommend to make use of this feature unless necessary.

2.4.5 Useful logic commands

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

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

```
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: 1.234e-16"
```

2.4.6 *subset* command

Finally there is a command that was written for sub-setting:

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

```
##    Age GPA Gender
## 1   23 3.1 Female
## 3   21 2.1 Female
## 5   24 2.3 Female
## 8   22 3.9   Male
## 10  21 3.2   Male
```

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

```
##    Age GPA Gender
## 8   22 3.9   Male
## 10  21 3.2   Male
```

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

```
##    Gender
## 1  Female
## 3  Female
## 5  Female
## 8    Male
## 10   Male
```

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

```
## [1] Female Female Female Male   Male
## Levels: Female Male
```

Notice that this last one results in a vector.

Exercise

The data set *upr* (part of *Resma3.RData*) has the application information provided to the University of all students that were eventually accepted between 2003 and 2013. Here are the first three students:

```
head(upr, 3)
```

```
##      ID.Code Year Gender Program.Code Highschool.GPA Aptitud.Verbal
## 1 00C2B4EF77 2005      M          502           3.97            647
## 2 00D66CF1BF 2003      M          502           3.80            597
## 3 00AB6118EB 2004      M         1203           4.00            567
##   Aptitud.Matem Aprov.Ingles Aprov.Matem Aprov.Espanol IGS Freshmen.GPA
## 1           621          626         672           551 342         3.67
## 2           726          618         718           575 343         2.75
## 3           691          424         616           609 342         3.62
##   Graduated Year.Grad. Grad..GPA Class.Facultad
## 1        Si       2012      3.33           INGE
## 2        No         NA        NA           INGE
## 3        No         NA        NA       CIENCIAS
```

How many female students applied in either 2010 or 2011, had a high school GPA of at least 3.0 and a freshman GPA between 3.0 and 3.5?

2.4.7 *order* command

if we need to sort a vector we have the sort command:

```
sort(Age)
```

```
##  [1] 18 20 20 20 20 21 21 22 23 24
```

but sometime we want to sort one vector by the order of another:

```
students[order(Age), ]
```

```
##    Age GPA Gender
## 6   18 2.9   Male
## 2   20 3.2   Male
## 4   20 2.1   Male
## 7   20 2.3   Male
## 9   20 2.6   Male
## 3   21 2.1 Female
## 10  21 3.2   Male
## 8   22 3.9   Male
## 1   23 3.1 Female
## 5   24 2.3 Female
```

Example Look-up Tables

Say we want to write a function which many times needs to calculate $\log(n!)$. We soon run into the following problem:

```
log(factorial(175))
```

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

despite the fact that this is not a really big number (it is 732.3394). The problem is that internally R uses the gamma function to calculate factorials, and $\Gamma(175)$ is larger than what R can handle.

There is of course a simple solution:

$$\log(n!) = \log(\prod_{i=1}^{n} i) = \sum_{i=1}^{n} \log i$$

but this turns out to be quite slow. Here is a better solution: we will make use of **Sterling's formula**:

$$n! \sim n^n e^{-n} \sqrt{2\pi n}$$
$$\log(n!) \sim n \log(n) - n + \frac{1}{2} \log(2\pi n)$$

say we will need to find $\log(n!)$ for n ranging from 0 to 500, then we can create a look-up table. For small values of n we use the exact formula, then the approximation:

```
logfac <- 0:500
names(logfac) <- logfac
for(n in 0:50)
  logfac[n+1] <- log(factorial(n))
for(n in 51:500)
  logfac[n+1] <- n*log(n)-n+0.5*log(2*pi*n)
```

and now we can find various values very easy:

```
logfac[c("2", 5, 30, 301, 30)]
```

```
##         2         5        30       301        30
##    0.6931    4.7875   74.6582 1420.6127   74.6582
```

Exercise

why did I use

logfac[c("2", 5, 30, 301, 30)]

and not

logfac[c(2, 5, 30, 301, 30)]

2.5 Generating Random Variates

2.5.1 Random Numbers

Everything starts with generating X_1, X_2, .. 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,] 75 98 92  41 37
## [2,] 56 32 97 100 86
## [3,] 77 15  2  59 89
## [4,] 64 57 49  36  1
```

2.5.2 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, .. that **appear** to be iid U[0,1], so-called *pseudo-random numbers*. In R we have the function *runif*:

```
runif(5)
```

```
## [1] 0.0586 0.5099 0.4658 0.4694 0.3597
```

or

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

```
## [1] 71.3 11.6 78.4 64.2 80.5
```

If we want to choose from a finite set we have

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

```
## [1] "f" "y" "t" "d" "z"
```

if no number is given it yields a random permutation:

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

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

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
## 102 228 328 225 117
```

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

Exercise

How can we randomly select 20 rows of the **upr** data set?

Exercise

A very useful technic in Statistics is called the *Bootstrap*. To use it one needs to find (many) Bootstrap samples. These are observations from the original data set, chosen at random and with repetition, as many as the original data set had. For example if the data is

```
##  [1] 13 20 26 30 31 34 35 37 38 48
```

Bootstrap samples might be

```
##  [1] 13 20 20 26 31 38 38 38 48 48
```

```
##  [1] 13 20 20 30 31 31 35 35 37 48
```

```
##  [1] 13 20 34 34 34 35 37 38 48 48
```

How can we find Bootstrap samples of the **upr** data set?

2.5.3 Standard Probability Distributions

Not surprisingly 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

Note we will use the term *density* for both discrete and continuous random variables.

Example Poisson distribution

We have $X \sim \text{Pois}(\lambda)$ if

$$P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda};\ x = 0, 1, ...$$

```
options(digits=4)
x <- c(0, 8, 12, 20)
# density
dpois(x, lambda=10)
```

```
## [1] 0.0000454 0.1125990 0.0947803 0.0018661
```

```
10^x/factorial(x)*exp(-10)
```

```
## [1] 0.0000454 0.1125990 0.0947803 0.0018661
```

```
# cumulative distribution function
ppois(x, 10)
```

```
## [1] 0.0000454 0.3328197 0.7915565 0.9984117
```

```
# random generation
rpois(5, 10)
```

```
## [1]  9 10 11  8  9
```

```
# quantiles
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.

Exercise

Generate 10000 variates from a Binomial distribution with n=10, p=0.25 and compare the relative frequencies with the theoretical probabilities.

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

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

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

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

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

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

```
##  [1] 3 2 1 4 3 1 8 2 3 2
```

if you want to find the probabilities or cdf:

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

```
##  [1] 0.1440 0.2400 0.4000 0.0864 0.1440 0.4000 0.0112 0.2400 0.1440 0.2400
```

```
round(0.4*(1-0.4)^(x-1), 4)
```

```
##  [1] 0.1440 0.2400 0.4000 0.0864 0.1440 0.4000 0.0112 0.2400 0.1440 0.2400
```

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

$$f(x;\alpha,\beta) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta};\ x > 0$$

but R uses

$$f^*(x;\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x};\ x > 0$$

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

```
## [1] 0.06608
```

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

```
## [1] 0.06608
```

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

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

```
## [1] 0.06608
```

Exercise

Consider a *normal mixture model*, say

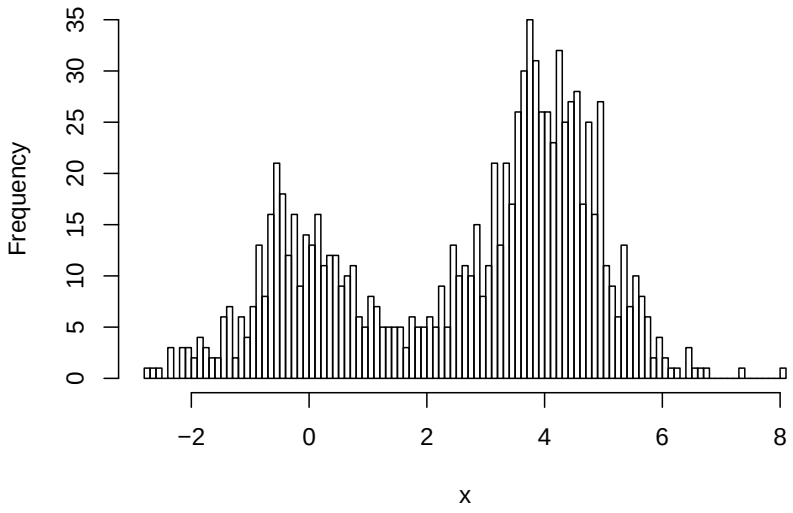
$$f(x,\alpha) = \alpha\phi(x,0,1) + (1-\alpha)\phi(x,4,1)$$

where ϕ is the normal density

$$\phi(x,\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$$

How could we generate 1000 variates from f if (say) $\alpha = 0.3$? This is what it should look like:

```
hist(x, breaks=100, main="")
```



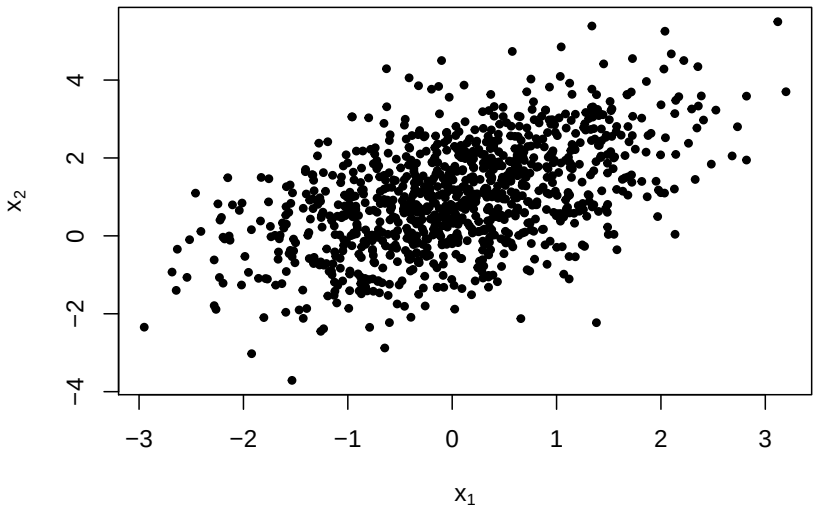
2.5.4 Other Variates

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,
     pch=20,
     xlab = expression(x[1]),
     ylab = expression(x[2]))
```



sigma is the variance-covariance matrix, so in the above we have

$$\rho = Cor(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X)\,Var(Y)}} = \frac{0.8}{\sqrt{1 * 2}} = 0.566$$

```
round(c(var(x[, 1]),
        var(x[, 2]),
        cor(x[, 1], x[, 2])), 3)
```

```
## [1] 1.013 1.917 0.515
```

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

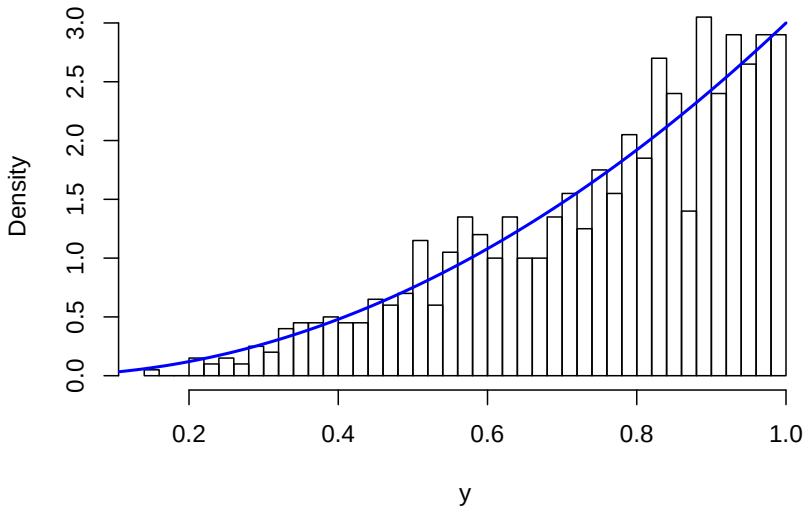
```
rpit <- function (n, fun, A, B)
{
  f <- function(x) eval(parse(text=fun))
  m <- min(2 * n, 1000)
  x <- seq(A, B, length = m)
  y <- f(x)
  z <- (x[2] - x[1])/6 * cumsum((y[-1] + 4 * y[-2] + y[-3]))
  z <- z/max(z)
  y <- c(0, z)
```

```
  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:

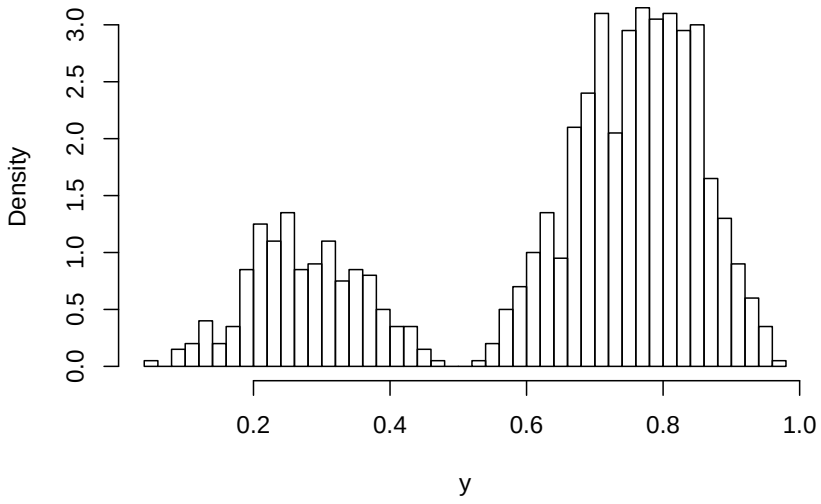
```
y <- rpit(1000, "x^2", 0, 1)
hist(y, 50, freq=FALSE, main="")
curve(3*x^2, 0, 1,
      col = "blue",
      lwd = 2,
      add = TRUE)
```



notice that for *rpit* the function doesn't even have to be normalized (aka integrate to 1).

or a bit more complicated:

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

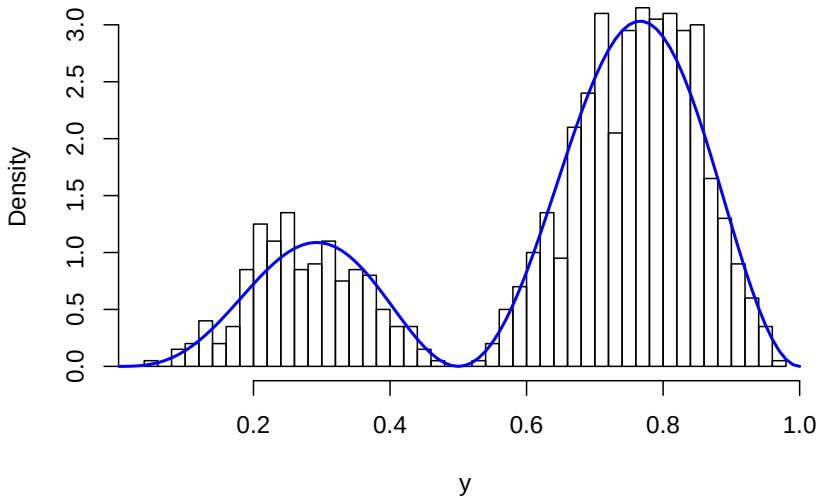


How about adding the density curve? For that we do need to normalize the function, that is we need to make sure that

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

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,
      col = "blue",
      lwd = 2,
      add = TRUE)
```



Exercise

generate 1000 variates from a Beta (1.5, 3) distribution, draw the histogram with 50 bins and add the density curve.

Hint: here you can use base R routines.

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

2.6 Writing Functions

2.6.1 General Information

In R/RStudio you have several ways to write your own functions:

- In the R console type

```
myfun <- function(x) {
  out <- x^2
  out
}
```

- 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 to *myfun*, 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('../some.folder/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 most code editors have many useful features not available in RStudio.

If *myfun* is open in RStudio there are some useful keyboard shortcuts. If the curser 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

2.6.2 Testing

As always you can test whether an object is a function:

```
x <- 1
f <- function(x) x
is.function(x)
```

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

```
is.function(f)
```

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

The *get* function takes a character string and returns a function (if it exists)

```
get("f")(4)
```

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

```
get("g")(4)
```

```
## Error in get("g"): object 'g' not found
```

2.6.3 Arguments

There are several ways to specify arguments in a function:

```
calc.power <- function(x, y, n=2) x^n + y^n
```

here n has a *default value*, x and y do not.

if the arguments are not named they are matched in order:

```
calc.power(2, 3)
```

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

or they can be explicitly named in any order:

```
calc.power(y=2, x=3)
```

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

```
calc.power(n=3, 2, 3)
```

```
## [1] 35
```

This however is not recommend as it can be very confusing.

R does partial matching of arguments:

```
f <- function(first, second) {
  first + second
}
f(fi=1, s=3)
```

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

but this is not a good programming style.

Default arguments can be defined in terms of others:

```
f <- function(first, second=2*first) {
  first + second
}
f(1)
```

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

If an argument does not have a default it can be tested for

```
f <- function(first, second) {
  if(!missing(second))
      out <- first + second
  else out <- first
```

```
  out
}
f(1)
```

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

```
f(1, s=3)
```

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

There is a special argument ..., used to pass arguments on to other functions:

```
f <- function(x, which, ...) {
  f1 <- function(x, mult) mult*x
  f2 <- function(x, pow) x^pow
  if(which==1)
    out <- f1(x, ...)
  else
    out <- f2(x, ...)
  out
}
f(1:3, 1, mult=2)
```

```
## [1] 2 4 6
```

```
f(1:3, 2, pow=3)
```

```
## [1]  1  8 27
```

This is one of the most useful programming structures in R!

Note this example also shows that in R functions can call other functions. In many computer programs there are so called *sub-routines*, in R this concept does not exist, functions are just functions.

Functions can even call themselves:

```
f <- function() {
  cat("A")
  if(sample(1:5, 1)>1) f()
  cat("\n")

}
f()
```

```
## AAAAAA
```

this is called *recursion* and is a very powerful programming technique, although for reasons of memory management not as useful in R as in other languages.

2.6.4 Lazy Evaluation

R uses a concept called *lazy evaluation*. This means that an argument is not evaluated until it is used:

```
f <- function(first, second) {
  if(first<10)
    out <- first
  else
    out <- first + second
  out
}
f(5, "A")
```

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

```
f(11, "A")
```

```
## Error in first + second: non-numeric argument to binary operator
```

This can be a source of computer bugs. One can override this behavior with the *force* command:

```
f <- function(first, second) {
  force(first+second)
  if(first<10)
    out <- first
  else
    out <- first + second
  out
}
f(5, "A")
```

```
## Error in first + second: non-numeric argument to binary operator
```

Note there is another simple way to accomplish the same thing: just use a statement like

test <- first+second

but force makes it clearer that the purpose here is to make sure first and second are of the correct type.

2.6.5 Return Values

A function can either return nothing or exactly one thing. It will automatically return the last object evaluated:

```
f <- function(x) {
  x^2
}
f(1:3)
```

```
## [1] 1 4 9
```

however, it is better programming style to have an explicit return object:

```
f <- function(x) {
  out <- x^2
  out
}
f(1:3)
```

```
## [1] 1 4 9
```

There is another way to specify what is returned:

```
f <- function(x) {
  return(x^2)
}
f(1:3)
```

```
## [1] 1 4 9
```

but this is usually used to return something early in the program:

```
f <- function(x) {
  if(!any(is.numeric(x)))
    return("Works only for numeric!")
  out <- sum(x^2)
  out
}
f(1:3)
```

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

```
f(letters[1:3])
```

```
## [1] "Works only for numeric!"
```

If you want to return more than one item use a list:

```
f <- function(x) {
  sq <- x^2
  sm <- sum(x)
  list(sq=sq, sum=sm)
}
f(1:3)
```

```
## $sq
## [1] 1 4 9
##
## $sum
## [1] 6
```

2.6.6 *on.exit*

on.exit is a routine that you use inside a function and that gets called and executed whenever the function terminates.

The advantage of *on.exit* is that is gets called when the function exits, regardless of whether an error was thrown. This means that its main use is for cleaning up after risky behavior. Risky, in this context, usually means accessing resources outside of R (that consequently cannot be guaranteed to work).

Common examples include connecting to databases or files (where the connection must be closed when you are finished, even if there was an error), or saving a plot to a file (where the graphics device must be closed afterwards)

Exercise

What does this function return?

```
f <- function(x=y) {
  y <- 10
  x
}
f()
```

2.6.7 Basic Programming Structures in R

R has all the standard programming structures:

2.6.7.1 Conditionals (if-else)

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

```
## [1] 0.6931    NaN
```

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

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

```
## [1] -1.099
```

```
centre(x, "median")
```

```
## [1] 0.2245
```

```
centre(x, "trimmed")
```

```
## [1] 0.001319
```

special R construct: ifelse

```
x <- sample(1:10, size=7, replace = TRUE)
x
```

```
## [1] 10 10  2  7  9  4  4
```

```
ifelse(x<5, "Yes", "No")
```

```
## [1] "No"  "No"  "Yes" "No"  "No"  "Yes" "Yes"
```

Exercise

What is the difference between these two functions:

```
f1 <- function(x) {
  if(x<10) return(0)
  x
}
f2 <- function(x) {
  ifelse(x<10, 0, x)
}
```

2.6.7.2 Loops

there are three standard loops in R:

- **for** loop

```
y <- rep(0, 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]  1  3  6 10 15 21 28 36 45 55
```

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]>40) y[i] <- (-1)
}
y
```

```
##  [1]  1  3  6 10 15 21 28 36 -1 -1
```

You can nest loops:

```
A <- matrix(0, 4, 4)
for(i in 1:4) {
  for(j in 1:4)
    A[i, j] <- i*j
}
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

- **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 length 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] 3
```

- **repeat** loop

similar to while loop, except that the 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] 3
```

Notice that a while and repeat loop could in principle run forever. I often 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
```

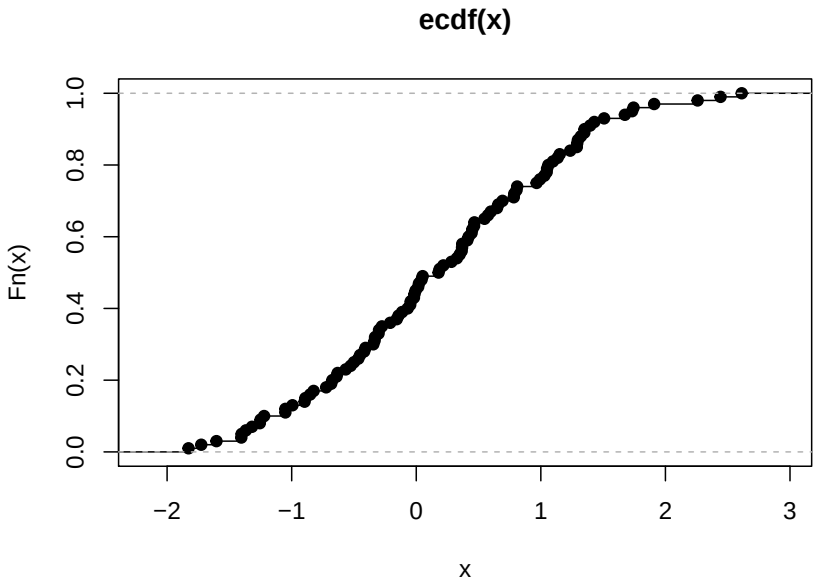
```
## [1] 40
```

2.7 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.

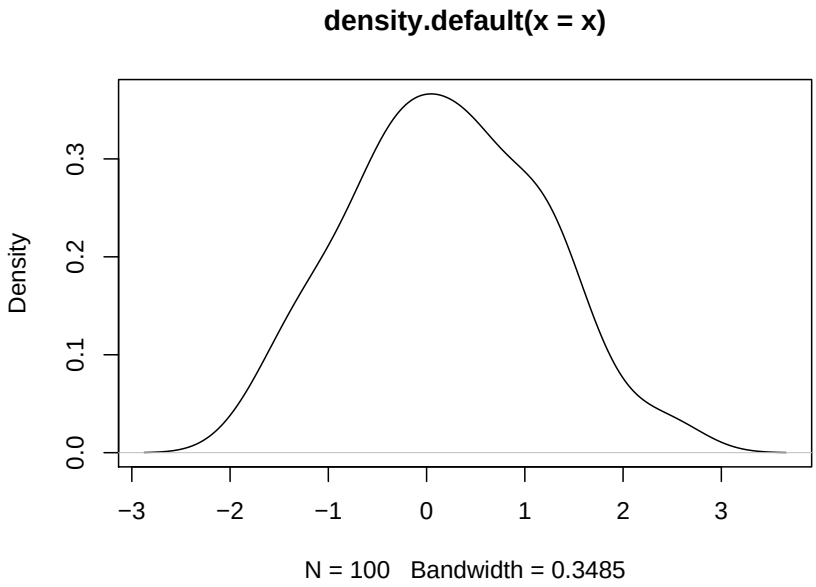
We start with the following:

```
x <- rnorm(100)
# Expirical Distribution Function
a <- ecdf(x)
plot(a)
```




```
# Non parametric density estimate
a <- density(x)
```

```
plot(a, type="l")
```



so although 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"
```

so when *plot* starts to run it examines the class property of the argument and does the corresponding graph.

There are many different plot functions (or *methods*)

```
methods(plot)
```

```
##  [1] plot,ANY-method                plot,color-method
##  [3] plot.aareg*                    plot.acf*
##  [5] plot.ACF*                      plot.agnes*
##  [7] plot.areg*                     plot.areg.boot*
##  [9] plot.aregImpute*               plot.augPred*
## [11] plot.biVar*                    plot.cld*
## [13] plot.clusGap*                  plot.compareFits*
```

```
## [15] plot.confint.glht*             plot.correspondence*
## [17] plot.cox.zph*                  plot.curveRep*
## [19] plot.data.frame*               plot.decomposed.ts*
## [21] plot.default                   plot.dendrogram*
## [23] plot.density*                  plot.describe*
## [25] plot.diana*                    plot.drawPlot*
## [27] plot.ecdf                      plot.factor*
## [29] plot.formula*                  plot.function
## [31] plot.gbayes*                   plot.ggplot*
## [33] plot.glht*                     plot.gls*
## [35] plot.gtable*                   plot.hcl_palettes*
## [37] plot.hclust*                   plot.histogram*
## [39] plot.HoltWinters*              plot.intervals.lmList*
## [41] plot.isoreg*                   plot.lda*
## [43] plot.lm*                       plot.lme*
## [45] plot.lmList*                   plot.mca*
## [47] plot.medpolish*                plot.mlm*
## [49] plot.mona*                     plot.nffGroupedData*
## [51] plot.nfnGroupedData*           plot.nls*
## [53] plot.nmGroupedData*            plot.partition*
## [55] plot.pdMat*                    plot.ppr*
## [57] plot.prcomp*                   plot.princomp*
## [59] plot.profile*                  plot.profile.nls*
## [61] plot.Quantile2*                plot.R6*
## [63] plot.ranef.lme*                plot.ranef.lmList*
## [65] plot.raster*                   plot.regsubsets*
## [67] plot.ridgelm*                  plot.rm.boot*
## [69] plot.rpart*                    plot.shingle*
## [71] plot.silhouette*               plot.simulate.lme*
## [73] plot.spec*                     plot.spline*
## [75] plot.stepfun                   plot.stl*
## [77] plot.summary.formula.response* plot.summary.formula.reverse*
## [79] plot.summaryM*                 plot.summaryP*
## [81] plot.summaryS*                 plot.Surv*
## [83] plot.survfit*                  plot.table*
## [85] plot.transcan*                 plot.trellis*
## [87] plot.ts                        plot.tskernel*
## [89] plot.TukeyHSD*                 plot.varclus*
## [91] plot.Variogram*                plot.xyVector*
## [93] plot.zoo*
## see '?methods' for accessing help and source code
```

so when we call plot with an object of class density, it in turn calls the function plot.density.

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

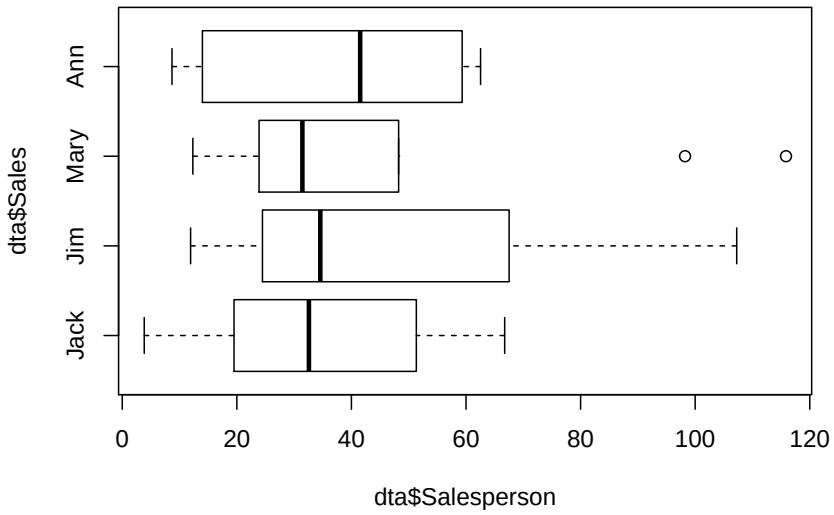
- gives the number of sales, their mean and standard deviation for each salesperson.
- does a boxplot of the sales, grouped by salesperson

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
23.62	Mary
12.32	Mary
21.20	Jim
59.33	Ann
23.63	Jim

Here is the non-object oriented solution:

```
report <- function(dta) {
  salespersons <- unique(dta$Salesperson)
  tbl <- matrix(0, length(salespersons), 3)
  rownames(tbl) <- salespersons
  colnames(tbl) <- c("Sales", "Mean", "SD")
  for(i in seq_along(salespersons)) {
      df <- dta[dta$Salesperson==salespersons[i], ]
      tbl[i, 1] <- nrow(df)
      tbl[i, 2] <- round(mean(df$Sales), 2)
      tbl[i, 3] <- round(sd(df$Sales), 2)
  }
  boxplot(dta$Sales~dta$Salesperson,
          horizontal=TRUE)
  return(kable.nice(tbl))
}
report(sales.data)
```



	Sales	Mean	SD
Mary	10	45.41	34.28
Jim	11	48.13	29.47
Ann	9	37.20	23.59
Jack	10	35.05	19.96

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

```
as.sales <- function(x) {
  class(x) <- c("sales", "data.frame")
  return(x)
}
```

Next we have to define the methods:

```
stats <- function(x) UseMethod("stats")
stats.sales <- function(dta) {
  salespersons <- unique(dta$Salesperson)
  tbl <- matrix(0, length(salespersons), 3)
  rownames(tbl) <- salespersons
  colnames(tbl) <- c("Sales", "Mean", "SD")
  for(i in seq_along(salespersons)) {
      df <- dta[dta$Salesperson==salespersons[i], ]
      tbl[i, 1] <- nrow(df)
      tbl[i, 2] <- round(mean(df$Sales), 2)
      tbl[i, 3] <- round(sd(df$Sales), 2)
  }
```

```
  return(kable.nice(tbl))
}
```

"plot" already exists, so we don't need the UseMethod part:

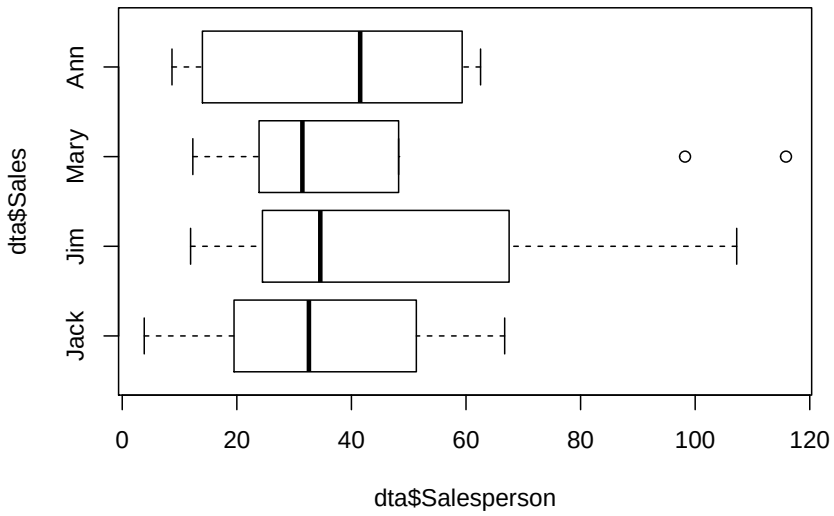
```
plot.sales <- function(dta)
  boxplot(dta$Sales~dta$Salesperson, horizontal=TRUE)
```

and now we can run

```
sales.data <- as.sales(sales.data)
# assign class sales to data frame
stats(sales.data)
```

	Sales	Mean	SD
Mary	10	45.41	34.28
Jim	11	48.13	29.47
Ann	9	37.20	23.59
Jack	10	35.05	19.96

```
plot(sales.data)
```

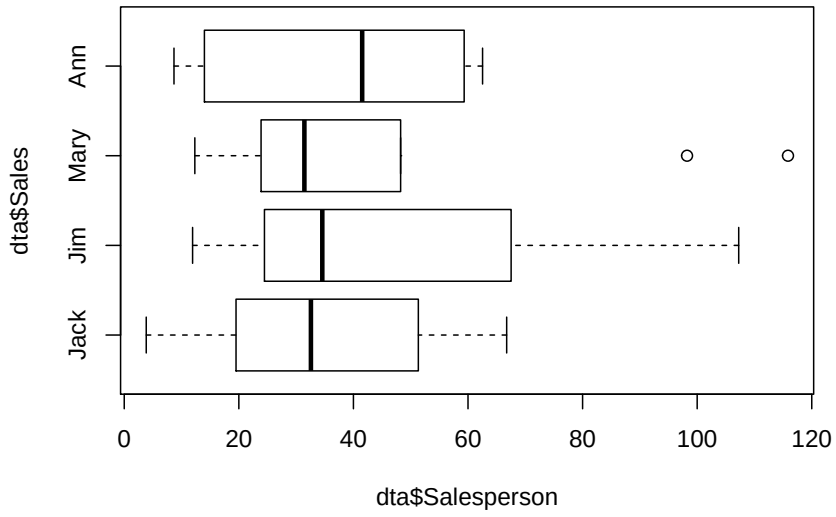


So far not much has been gained. But let's say that sometimes we also have information on the whether the sales person was on the morning or on the afternoon shift, and we want to include this in our report. One great feature of object-oriented programming is *inheritance*, that is we can define a new 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
1	23.62	Mary	Afternoon
2	12.32	Mary	Afternoon
3	21.20	Jim	Morning
4	59.33	Ann	Morning
5	23.63	Jim	Afternoon

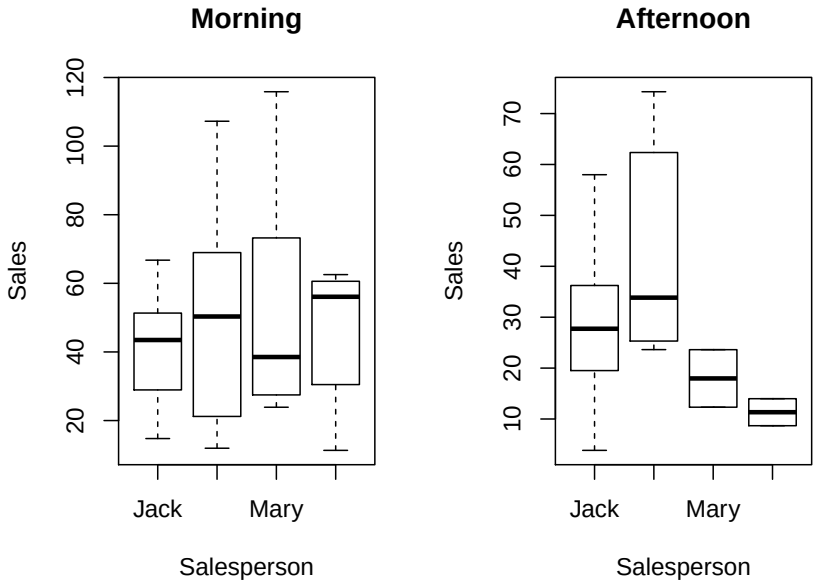
```
class(sales.time.data) <- c("salestime",
                            "sales",
                            "data.frame")
plot(sales.time.data)
```



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.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)
```



Note that we already used inheritance in the definition of the sales class:

```
class(x) <- c("sales", "data.frame")
```

so that the data remains as a data frame. If we had used

```
class(x) <- "sales"
```

sales.data would have been turned into a list.

---

generally every class has at least three methods:

- print
- summary (stats)
- plot

Example

Let's return to the empirical distribution function discussed earlier. It is defined as follows: Let $x_1, .., x_n$ be a sample from some probability density f. Then the empirical distribution function $\hat{F}(x)$ is defined by

$$\hat{F}(x) = \frac{\text{number of } x_i \leq x}{n}$$

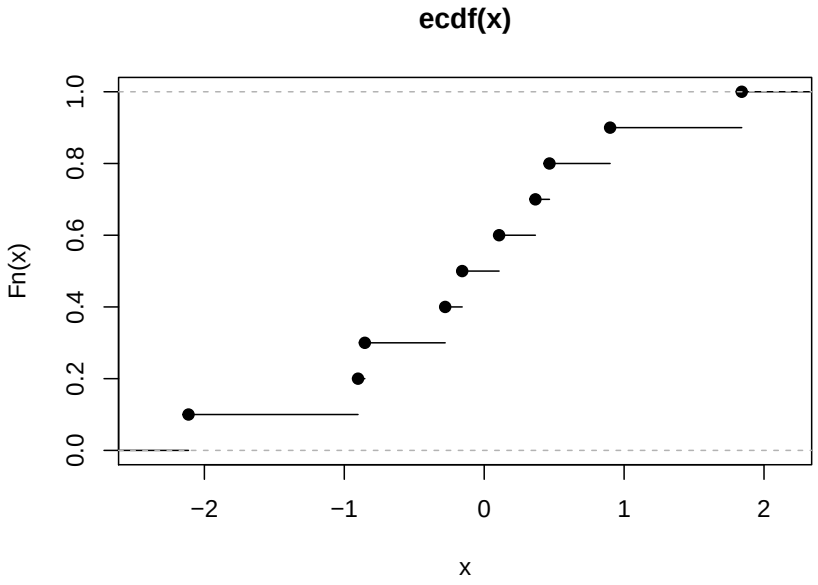
Let's have a closer look at the ecdf object:

```
x <- sort(rnorm(10))
y <- ecdf(x)
class(y)
```

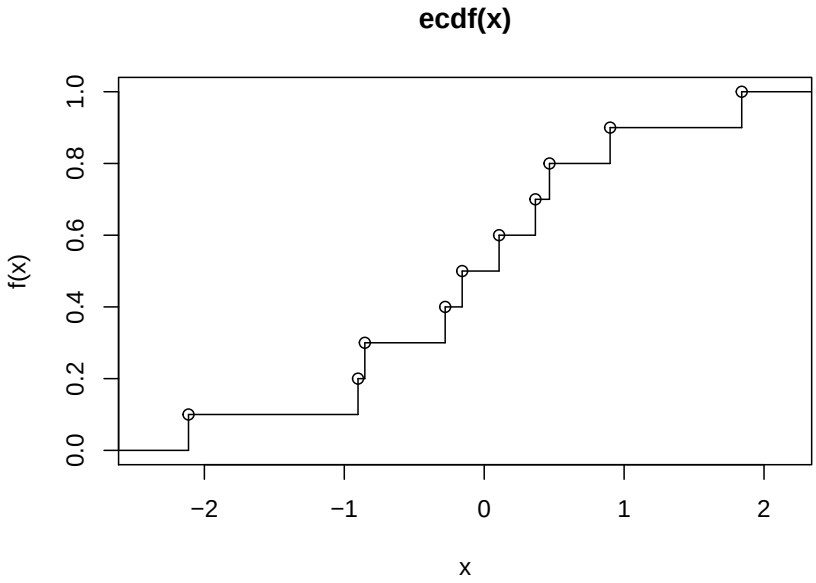
```
## [1] "ecdf"     "stepfun"  "function"
```

so the classes of an ecdf object are ecdf, stepfun and function. Let's see what that means:

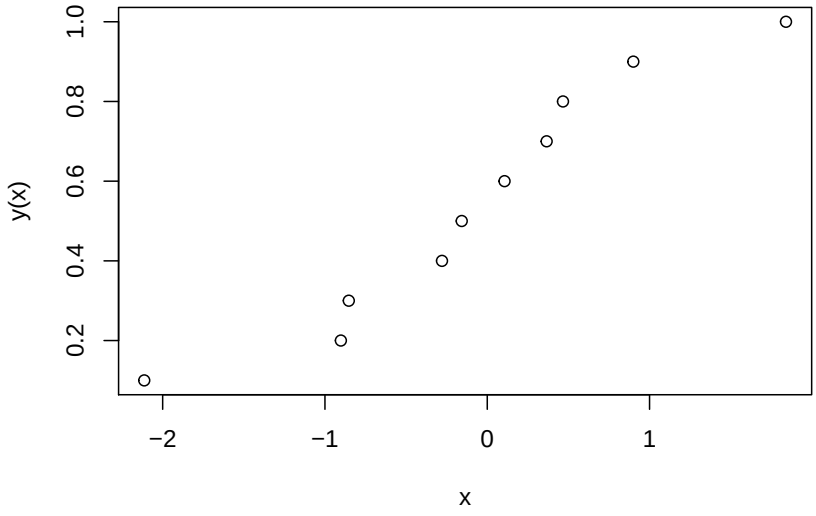
```
plot(y)
```




```
class(y) <- c("stepfun", "function")
plot(y)
```




```
class(y) <- "function"
plot(x, y(x))
```

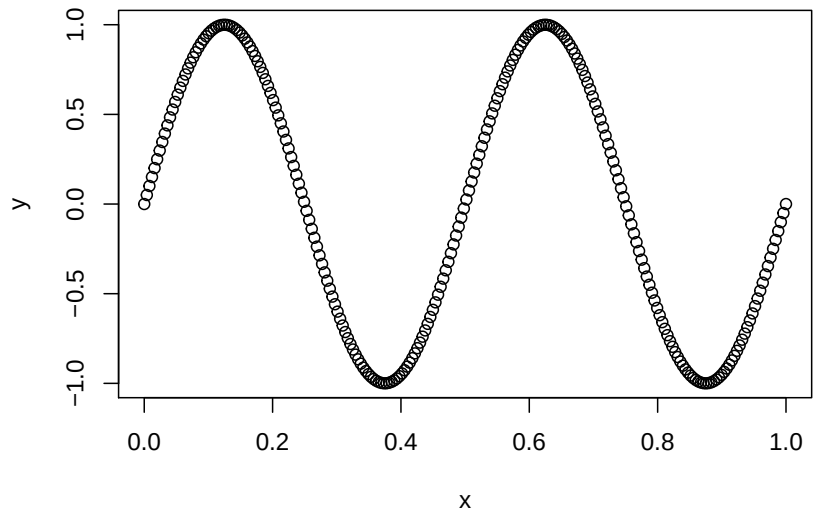


Example

Let's say we have a data frame with an x and a y column, which are the coordinates of some function. We want the plot command to graph that function:

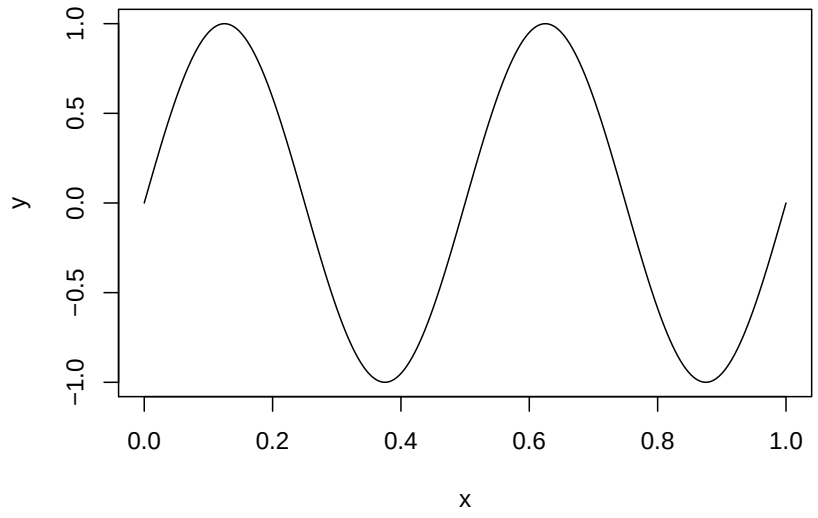
```
x <- seq(0, 1, length=250)
y <- sin(4*pi*x)
df <- data.frame(x=x, y=y)
```

```
plot(df)
```



We can do this by defining a new class *fn*:

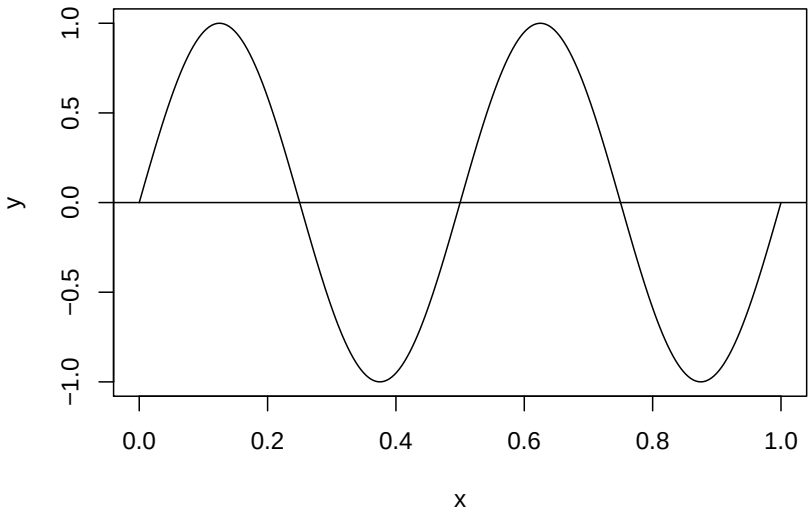
```
class(df) <- c("fn", "data.frame")
plot.fn <- function(df) {
  plot(df[, 1], df[, 2], type="l",
       xlab="x", ylab="y")
}
plot(df)
```



Exercise

Say for all the functions that cross the horizontal line y=0 we want to add that line to the graph. Use OOP!

```
plot(df)
```



2.8 Vector Arithmetic

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

```
options(digits=4)
```

```
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] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

```
x^2*exp(-(x/10)^2)
```

```
##  [1]  0.990  3.843  8.225 13.634 19.470 25.116 30.019 33.747 36.034 36.788
```

```
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
```

but this can also fail!

2.8.1 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 element wise multiplication. If we want to do actual matrix multiplication we have

```
x %*% y # an infix operator!
```

```
##      [,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
```

Note so careful when using t as the name of an object!

As you know (I hope?) from linear algebra any (non-singular square) matrix A can be written in the form

$$A = UDU^{-1}$$

where D is a diagonal matrix. One use of this is we can then easily find

$$\begin{aligned} A^2 &= AA = \\ UDU^{-1}UDU^{-1} &= \\ UDDU^{-1} &= \\ UD^2U^{-1} \end{aligned}$$

and with induction we get

$$A^n = UD^nU^{-1}$$

Example Say

```
A <- matrix(1:9/10, 3, 3)
z <- eigen(A)
z
```

```
## eigen() decomposition
## $values
## [1]  1.612e+00 -1.117e-01  5.239e-17
##
## $vectors
##         [,1]    [,2]    [,3]
## [1,] -0.4645 -0.8829  0.4082
## [2,] -0.5708 -0.2395 -0.8165
## [3,] -0.6770  0.4039  0.4082
```

```
z$vectors %*% diag(z$values^2) %*% solve(z$vectors)
```

```
##      [,1] [,2] [,3]
## [1,] 0.30 0.66 1.02
## [2,] 0.36 0.81 1.26
## [3,] 0.42 0.96 1.50
```

```
A %*% A
```

```
##      [,1] [,2] [,3]
## [1,] 0.30 0.66 1.02
## [2,] 0.36 0.81 1.26
## [3,] 0.42 0.96 1.50
```

```
z$vectors %*% diag(z$values^10) %*% solve(z$vectors)
```

```
##       [,1]  [,2]  [,3]
## [1,] 13.25 30.00 46.75
## [2,] 16.28 36.86 57.45
## [3,] 19.31 43.72 68.14
```

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

2.8.2 Vectorization

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 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] 0.3333
```

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] 0.3333
```

```
fA(2)
```

```
## [1] 2.667
```

but not

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

```
## Error in integrate(f, 0, A): length(upper) == 1 is not TRUE
```

so we need to "vectorize":

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

```
## [1] 0.3333 2.6667
```

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
  y
}
fA.vec(c(1, 2))
```

```
## [1] 0.3333 2.6667
```

Once you have a function that does something for one value, vectorizing it is (usually) very quick and almost always worthwhile!

2.8.3 *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] 0.1414 0.1743
```

Here is an alternative:

- *apply*

```
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] 0.1402 0.1748
```

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 neval
##  sim1(50) 597.6 603.4 643.1  606.7 611.9 965.9    10
```

```
microbenchmark(sim2(50), times = 10)
```

```
## Unit: milliseconds
##       expr   min    lq  mean median    uq   max neval
##  sim2(50) 441.6 445.7 466.5  448.7 458.5 613.7    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] -0.01575
##
## $B
## [1] 0.2297
##
## $C
## [1] 0.1962
```

as we see the result is again a list. Notice that this goes against the way R is generally written: the resulting object could be type-converted to the simpler vector but is not.

Often we want it to be a vector. We could use *unlist*, or

```
sapply(x, mean)
```

```
##         A         B         C
## -0.01575  0.22970  0.19617
```

- *tapply* for vectors

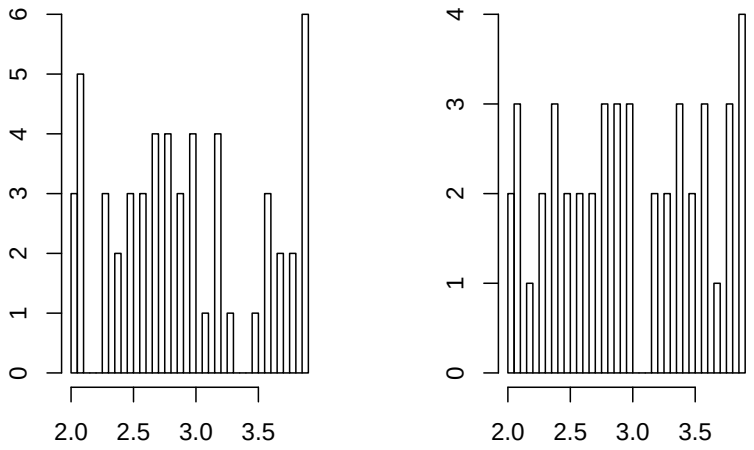
apply a function to the numbers in one vector, grouped by the values in another (categorical) vector:

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

```
## Female   Male
##  2.928  2.993
```

Here is another less obvious example:

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



2.9 Working with Characters

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 a bit of work, but there is a nice routine in the *Hmisc* package that helps:

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

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

Resma3 has the data set *agesexUS*, which has the breakdown of the US population by gender and age according to the 2000 US Census. We are going to work with the names of the states (plus DC and PR) for a bit:

```
states <- agesexUS$State
head(states)
```

```
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
## [6] "Colorado"
```

to find out how long a string is use

```
nchar(states)
```

```
##  [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
```

What state has the longest name?

```
states[which.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] "Ala" "Ala" "Ari" "Ark" "Cal" "Col" "Con" "Del" "Dis" "Flo" "Geo"
## [12] "Haw" "Ida" "Ill" "Ind" "Iow" "Kan" "Ken" "Lou" "Mai" "Mar" "Mas"
## [23] "Mic" "Min" "Mis" "Mis" "Mon" "Neb" "Nev" "New" "New" "New" "New"
## [34] "Nor" "Nor" "Ohi" "Okl" "Ore" "Pen" "Rho" "Sou" "Sou" "Ten" "Tex"
## [45] "Uta" "Ver" "Vir" "Was" "Wes" "Wis" "Wyo" "Pue"
```

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

```
abbreviate(states)[1:6]
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##     "Albm"     "Alsk"     "Arzn"     "Arkn"     "Clfr"     "Clrd"
```

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)[1:6]
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##   "Alabam"   "Alaska"   "Arizon"   "Arknss"   "Calfrn"   "Colord"
```

Notice that this keeps the full strings as names.

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

```
substring(states,
          first = nchar(states)-2,
          last = nchar(states))[1:6]
```

```
## [1] "ama" "ska" "ona" "sas" "nia" "ado"
```

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 directly:

```
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"      "basic.stats.Rmd" "bayes.Rmd"
## [5] "blank.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"
```

the syntax [pP] matches either p or P. This is an example of a *regular expression*, which we will discuss shortly.

Exercise

Find all the states whose name consist of two or more words (like Puerto Rico)

```
##  [1] "District of Columbia" "New Hampshire"        "New Jersey"
##  [4] "New Mexico"           "New York"             "North Carolina"
##  [7] "North Dakota"         "Rhode Island"         "South Carolina"
## [10] "South Dakota"         "West Virginia"        "Puerto Rico"
```

We can also use the function *tolower*, which turns all the letters into lower case:

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

```
## [1] "alabama"  "alaska"   "arizona"  "arkansas"
```

```
grep("p", tolower(states), value = TRUE)
```

```
## [1] "mississippi"   "new hampshire" "pennsylvania"  "puerto rico"
```

but now all the letters are lower case.

Of course there is also a *toupper* function.

We alreadu used *grep*. We also have the *grepl* function, which which does the same but instead of the location it returns TRUE if the string contains the pattern:

```
states[1:6]
```

```
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
## [6] "Colorado"
```

```
grepl(pattern = "s", states)[1:6]
```

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

Exercise

```
cat(y, "\n")
```

```
t e n d a f t w o d z t o t y l d o i m a n i i r m r k p z m g f h g y q c v h z c v e
```

How many a' are in this string?

You can get the vector into R as follows: copy it and in R run

```
# Windows
x <- scan("clipboard", what="char")
# Mac
x <- scan(pipe("pbpaste"), what="char")
```

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:

```
f <- function(x) {
  ifelse(x[1] > 0, length(x), 0)
# if there is no a, x is -1, so we get 0
}
num_a <- sapply(positions_a, f)
num_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")
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, f))
}
num_vowels
```

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

2.9.1 *paste, paste0* commands

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"
```

This last one (no space between) is needed often enough it has its own command:

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

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

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"
```

Exercise

write a routine that generates a licence plate in Puerto Rico at random. For example

```
license.plate()
```

```
## [1] "WMC964"
```

---

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"
```

Exercise

Here is Abraham Lincolns famous Gettyburg address:

```
cat(gettysburg)
```

```
 Four score and seven years ago our fathers brought forth on this continent, a new natio

Now we are engaged in a great civil war, testing whether that nation, or any nation so c

But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallo
```

How many times did Lincoln use the word "people"?

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

2.9.2 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
- 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

```
cat(txt)
```

```
## In 2017 there where 17 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[grepl("\\d", txt)]
```

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

Here d stands for digits. The backslash in front is the standard regex syntax, but backslashes in R have special meanings, so we need another one in front. This second one is called an *escape character*, it tells R to treat the backslash as such, and not as a special character.

Say we want to replace the spaces in a sentence with the underscore. The regex symbol space is s:

```
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. It matches everything between the brackets:

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

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

A caret in front is NOT:

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

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

Exercise

What command would replace all komma's, dot's and semicolons in the gettysburg address with an ampersant (@)?

```
 Four score and seven years ago our fathers brought forth on this continent@ a new natio

Now we are engaged in a great civil war@ testing whether that nation@ or any nation so c

But@ in a larger sense@ we can not dedicate -- we can not consecrate -- we can not hallo
```

2.9.3 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
[[:digit:]] 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[grepl("[[:digit:]]", txt)])
```

```
## [1] 2017   17
```

2.9.4 Some Examples

2.9.4.1 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, " "))
wrds
```

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

Next we need to turn each word around. To do that we split each word into individual letters, reverse them and paste them back together:

```
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"
```

2.9.4.2 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")
```

\n is the newline character, so each line of the webpage will be an element of the vector.

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(unlist(gregexpr("@", txt))>0)
```

```
## [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 an html table, which was read in as a single string. We can see that immediately after each email address is the text , which is the html tag to end a link. Let's split up the text according to the tag:

```
txt <- unlist(strsplit(paste(txt, collapse=""), "</a>"))
txt[1:2]
```

```
## [1] "          <table id=\"people\" cellspacing='0' border='0' cellpadding='2px'><tr>
## [2] "</td><td><a href=\"mailto:robert.acar@upr.edu\"> robert.acar@upr.edu"
```

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*118=236, and each address appeared twice in the table. So we got all of them.

Finally we need to extract the email address 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] "olgamary.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.vidaurraza@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"      "rodrigo.leon@upr.edu"
##  [95] "alibeth.luna@upr.edu"           "eddie.mendez@upr.edu"
##  [97] "robert.medina@upr.edu"          "luis.mestre2@upr.edu"
##  [99] "kevin.molina1@upr.edu"          "bayron.morales@upr.edu"
## [101] "ana.moreno@upr.edu"             "didier.murillo@upr.edu"
## [103] "bernnie.murillo@upr.edu"        "daniel.ovalle@upr.edu"
## [105] "felix.pabon@upr.edu"            "cristian.perdomo@upr.edu"
## [107] "jessenia.quintero@upr.edu"      "eric.rivera8@upr.edu"
## [109] "daniel.rocha@upr.edu"           "william.rueda@upr.edu"
## [111] "jose.santos7@upr.edu"           "deiver.suarez@upr.edu"
## [113] "maria.torres65@upr.edu"         "ana.trujillo2@upr.edu"
## [115] "juan.valera@upr.edu"            "raul.valerio@upr.edu"
## [117] "diana.vargas1@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

2.9.4.3 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 0'1 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] "11101"   "1001010" "1111"    "111000"  "100010"
```

```
summary(x)
```

```
## N = 1000
## Mode = 1010111
## % 0's = 43.7
```

2.10 Data Input/Output, Transferring R Objects

2.10.1 Printing Info to Screen

The basic functions to display information on the screen are

```
options(digits=4)
x <- rnorm(5, 100, 30)
print(x)
```

```
## [1]  61.73  43.28  96.99 115.66  74.89
```

```
cat(x)
```

```
## 61.73 43.28 96.99 115.7 74.89
```

Both of these have certain advantages:

- with print you can easily control the number of digits:

```
print(x, digits=6)
```

```
## [1]  61.7320  43.2771  96.9860 115.6570  74.8856
```

- with cat you can easily mix text and numeric:

```
cat("The mean is ", round(mean(x), 1), "\n")
```

```
## The mean is  78.5
```

The "\n" (**n**ewline) is needed so that the cursor moves to the next line. This also sometimes referred as a *carriage return*.

Another advantage of cat is that one can have different rounding for different numbers:

```
x <- 1100; y <- 0.00123334
print(c(x, y), 4)
```

```
## [1] 1.100e+03 1.233e-03
```

```
cat(x, "  ", round(y, 4), "\n")
```

```
## 1100    0.0012
```

Notice that in the case of print R switches to scientific notation. This default behavior can be changed with

```
options(scipen=999)
print(c(x, y), 4)
```

```
## [1] 1100.000000    0.001233
```

```
options(scipen=0)
print(c(x, y), 4)
```

```
## [1] 1.100e+03 1.233e-03
```

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

```
x <- 1100; pval <- 0.00123334
```

```
c(x, pval)
```

```
## [1] 1.100e+03 1.233e-03
```

```
sprintf("%.3f", c(x, pval))
```

```
## [1] "1100.000" "0.001"
```

2.10.2 Reading in a Vector

Often the easiest thing to do is to use copy-paste the data and then simply *scan* it into R:

```
x <- scan("clipboard")
```

Note: if you are using a Mac you need to use

```
x <- scan(pipe("pbpaste"))
```

- 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="char".

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 a 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 behavior 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")
```

If it is a webpage use

```
x <- scan(url("http://somesite.html"))
```

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.

Example: A non-standard input format.

Consider the file at

```
x <- scan("http://academic.uprm.edu/wrolke/esma6835/sales.txt")
```

```
for(i in 1:5) cat(x[i],"\n")
```

```
    152 278,11     202 998,05      89 060,44
  1 803 360,69      24 608,24      49 004,89
    812 679,54     186 289,80      95 946,42
    171 266,69     208 691,32      28 503,93
     56 646,34      41 287,10      15 483,96
```

these are sales data for some store. We want to find the mean sales amount. So we need to read the data into R, but there are some issues:

- the data delimits the decimals European-style, using a comma instead of a period.
- for easier readability the million and the thousand are separated by a space.

so the first number really is 152278.11.

How can we do this? To start we need to read the data as a single character string:

```
x <- paste0()
  scan("http://academic.uprm.edu/wrolke/esma6835/sales.txt",
   sep="\n"), collapse="")
```

Let's see what we have, at least at the beginning:

```
substring(x, 1, 100)
```

```
## [1] "    152 278,11     202 998,05      89 060,44  1 803 360,69      24 608,24      4
```

Next we can replace the , with .:

```
x <- gsub(",", "\\.", x)
substring(x, 1, 100)
```

```
## [1] "    152 278.11     202 998.05      89 060.44  1 803 360.69      24 608.24      4
```

notice the \\. This is needed because . is a special character in R, it actually needs to be escaped twice!

Next notice that the numbers are always separated by at least two spaces, so we can split them up with

```
x <- strsplit(x, "  ")[[1]]
x[1:10]
```

```
##  [1] ""             ""             "152 278.11"   ""
##  [5] " 202 998.05"  ""             ""             "89 060.44"
##  [9] "1 803 360.69" ""
```

Now we can remove any spaces:

```
x <- gsub(" ", "", x)
x[1:10]
```

```
##  [1] ""           ""           "152278.11"  ""           "202998.05"
##  [6] ""           ""           "89060.44"   "1803360.69" ""
```

and get rid of the "":

```
x <- x[x!=""]
x[1:10]
```

```
##  [1] "152278.11"  "202998.05"  "89060.44"   "1803360.69" "24608.24"
##  [6] "49004.89"   "812679.54"  "186289.80"  "95946.42"   "171266.69"
```

Almost done:

```
x <- as.numeric(x)
mean(x)
```

```
## [1] 198450
```

2.10.3 Importing a Data Frame

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*:

ID	Age	GPA	Gender
37416	24	3.3	Female
44884	25	3.9	Female
16139	18	3.2	Male
13116	20	2.7	Female
39257	24	2.9	Female
94248	21	3.7	Female
96640	21	3.3	Female
28566	21	3.6	Female
49054	23	3.5	Male
43416	18	3.2	Male

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.

2.10.4 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")
```

2.10.5 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.

2.10.6 Packages

there are a number of packages written to help with data I/O. We will discuss some of them later.

2.10.7 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] "_book"           "_bookdown.yml"   "_bookdown_files" "_main.log"
## [5] "_main.Rmd"       "_main.tex"
```

2.11 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.

2.11.1 Barcharts

Say we have the data on the number of freshman, sophomores etc at some college:

```
head(students)
```

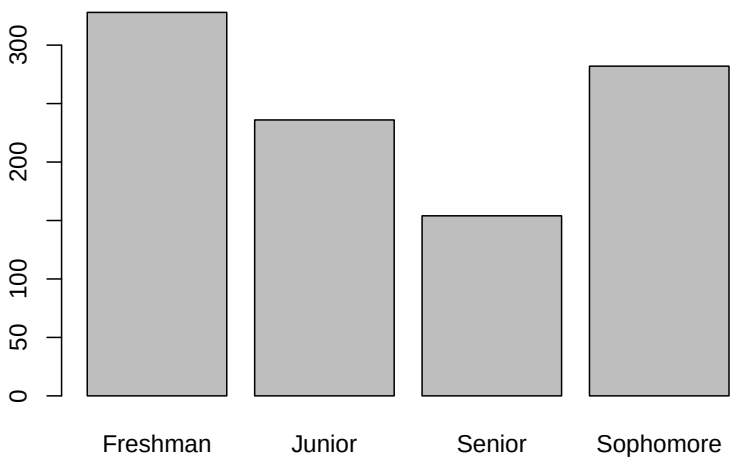
```
## [1] "Freshman" "Freshman" "Senior"   "Freshman" "Freshman" "Freshman"
```

So here we have a *categorical* variable. One popular graph for this type of data is the *pie chart*,

BUT repeat after me: pie charts are evil!!

A MUCH better alternative are bar charts:

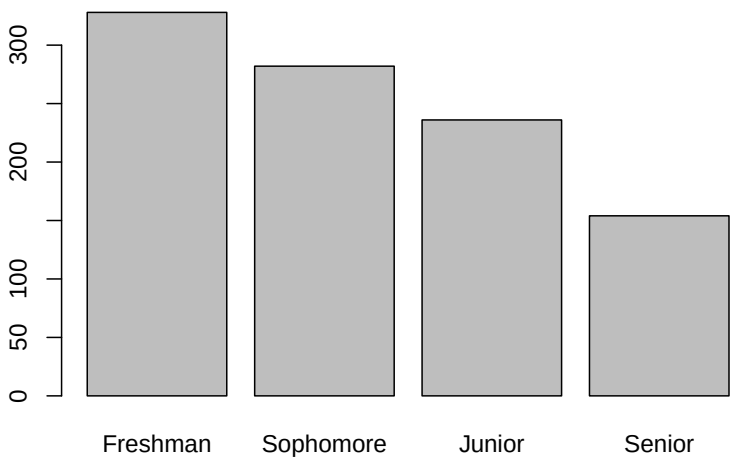
```
barplot(table(students))
```



Note that the argument to bar plot has to be a table.

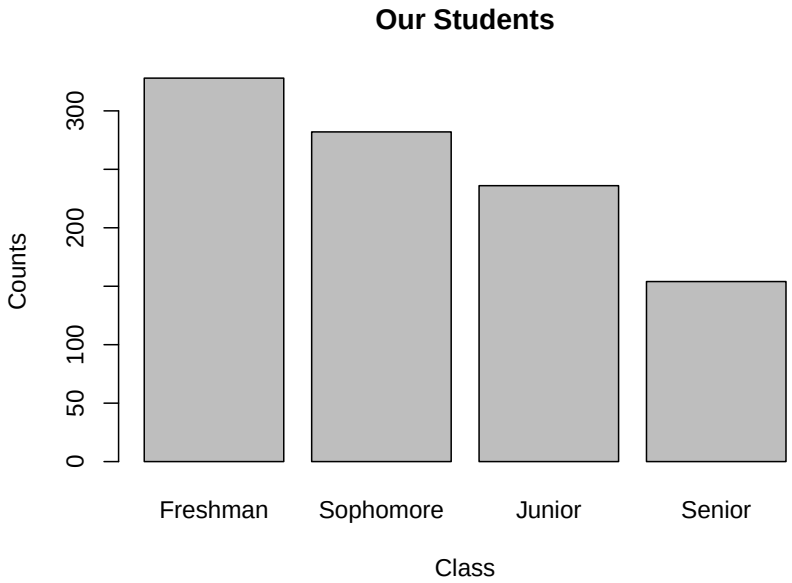
But this is not quite right, Sophomores should come second. In general, when we have a categorical variable which has an ordering we should turn it into a factor:

```
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:

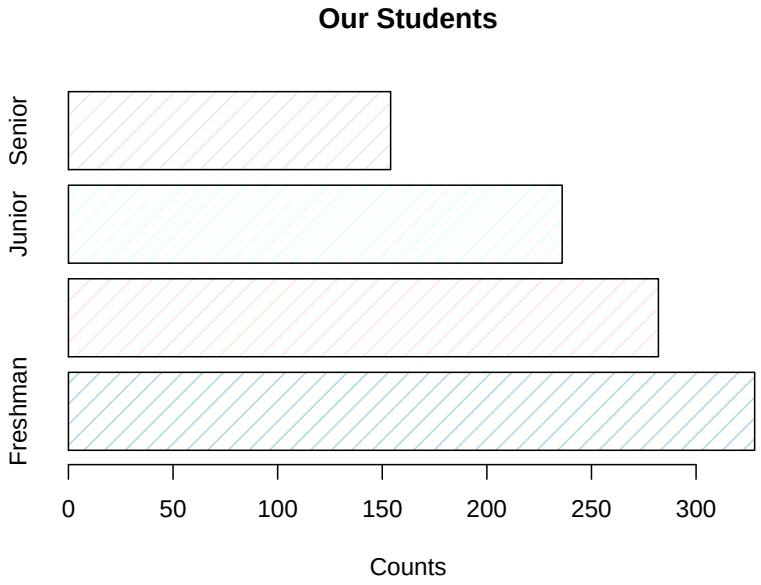
```
barplot(tbl.students.fac,
        main = "Our Students",
        xlab = "Class",
        ylab = "Counts")
```



and then there are arguments specific to the graph:

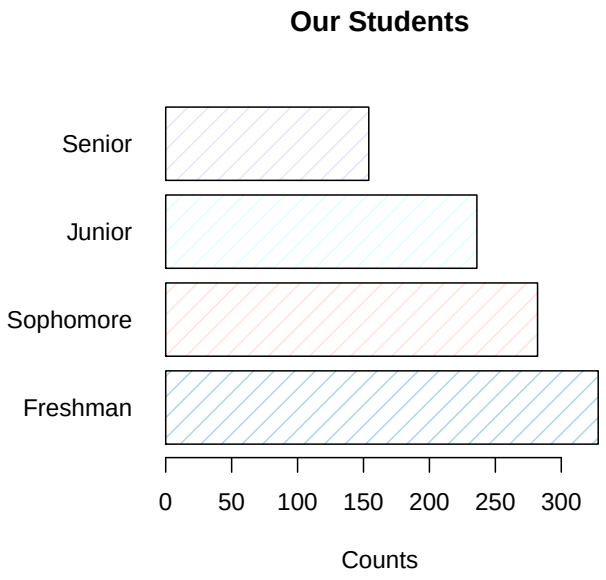
```
barplot(tbl.students.fac,
        main = "Our Students",
```

```
        xlab = "Counts",
        ylab = "",
        horiz = TRUE,
        density = 10,
        col = c("lightblue", "mistyrose",
                "lightcyan", "lavender"))
```



Notice that we can't see all the y labels. It would be better to have them horizontal as well. But we also need to make space for them:

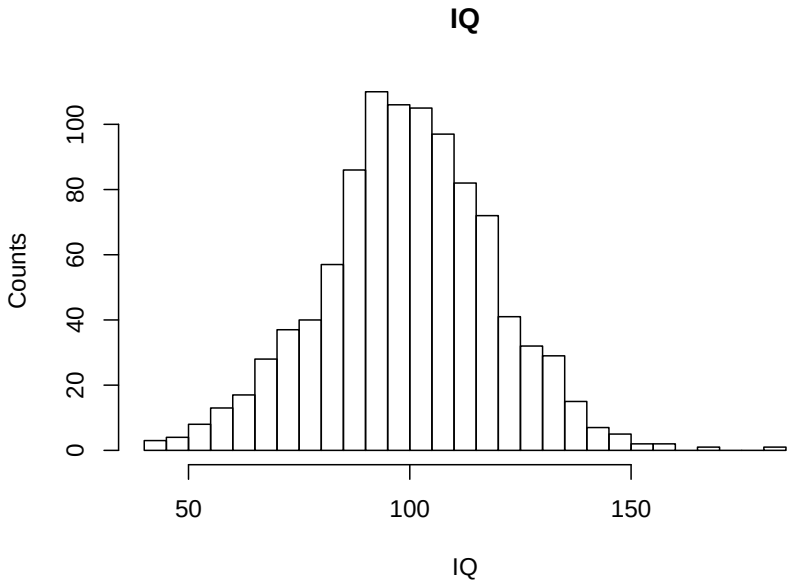
```
par(mai=c(1,2,1,1))
barplot(tbl.students.fac,
        main = "Our Students",
        xlab = "Counts",
        ylab = "",
        horiz = TRUE,
        density = 10,
        las = 1,
        col = c("lightblue", "mistyrose",
                "lightcyan", "lavender"))
```



Here I used the *par* command to set some parameters for how the graph is drawn. There is a very long list of such parameters, see the par help file.

2.11.2 Histogram

```
x <- rnorm(1000, 100, 20)
hist(x,
     breaks = 50,
     main = "IQ",
     xlab = "IQ",
     ylab = "Counts")
```



The hist command is useful for its side effect of counting:

```
range(x)
```

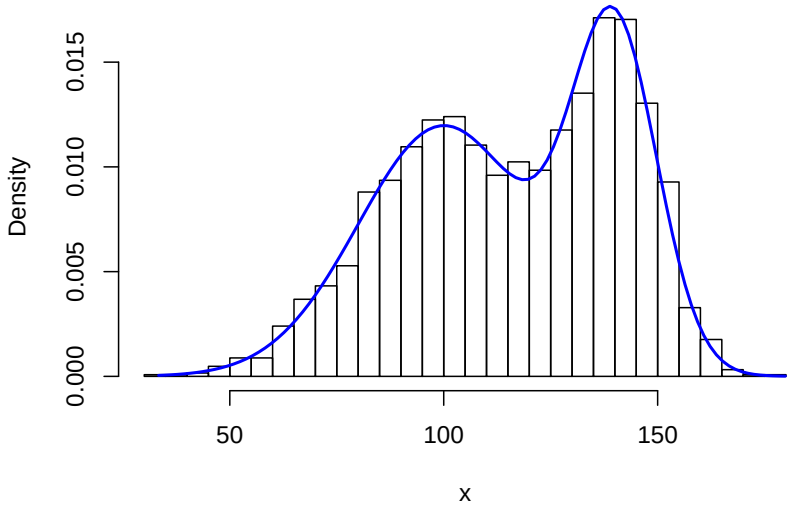
```
## [1]  40.0 181.2
```

```
hist(x,
  breaks = c(0, 50, 80, 100, 120, 150, 250),
  plot = FALSE)$counts
```

```
## [1]   7 143 359 356 129   6
```

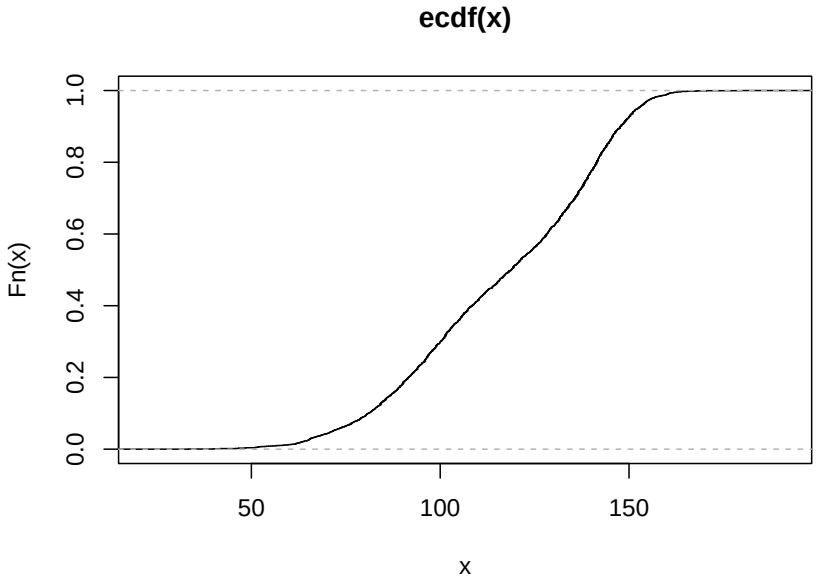
Often we want to compare a histogram with a theoretical curve, say a probability density. Then we can use the curve command with the argument add=TRUE:

```
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)
```



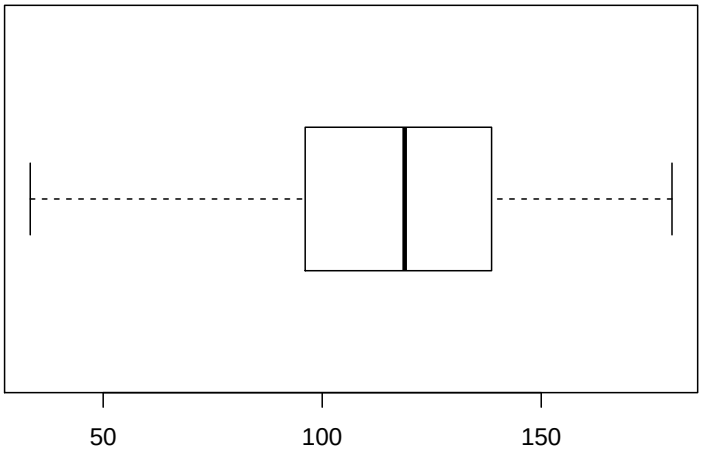
2.11.3 Empirical Distribution Function

```
plot(ecdf(x))
```

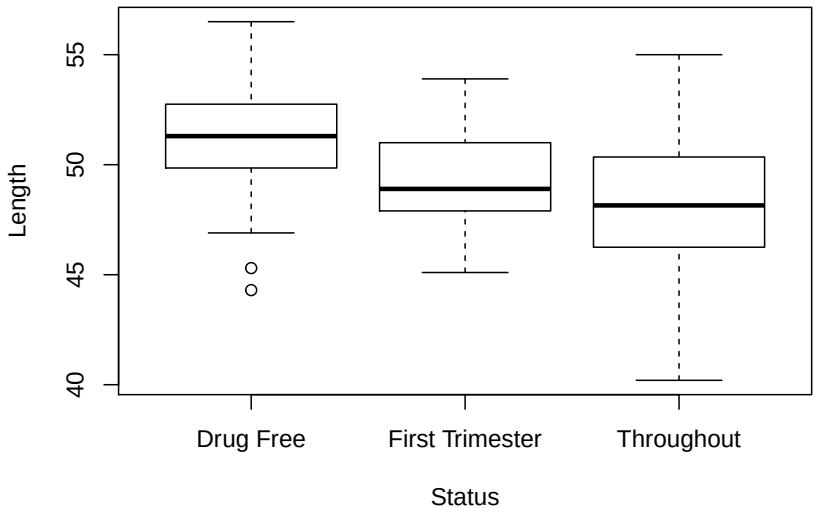


2.11.4 Boxplot

```
boxplot(x, horizontal = TRUE)
```

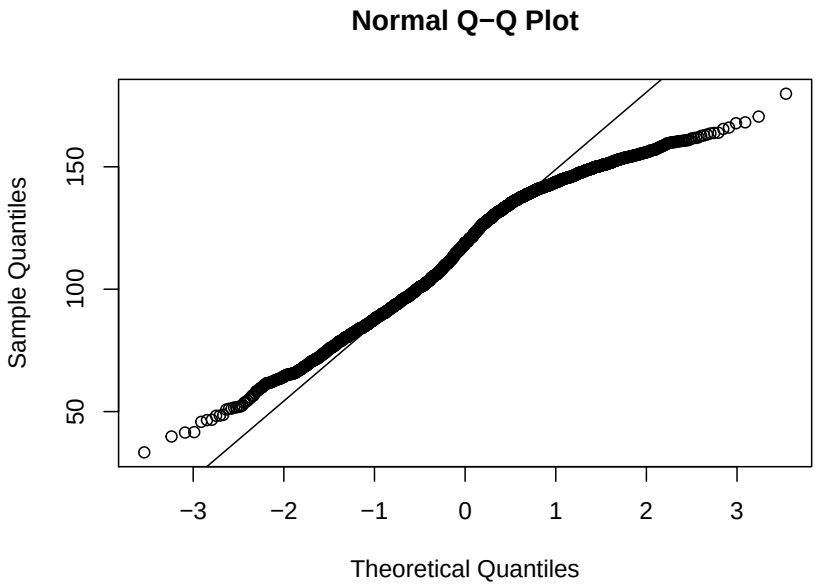



```
boxplot(Length~Status, data=mothers)
```



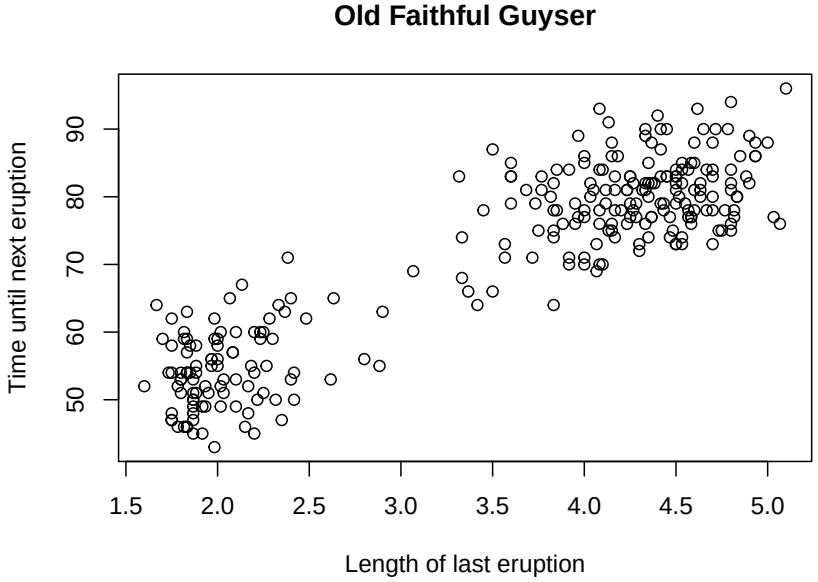
2.11.5 Normal Probability Plot

```
qqnorm(x)
qqline(x)
```



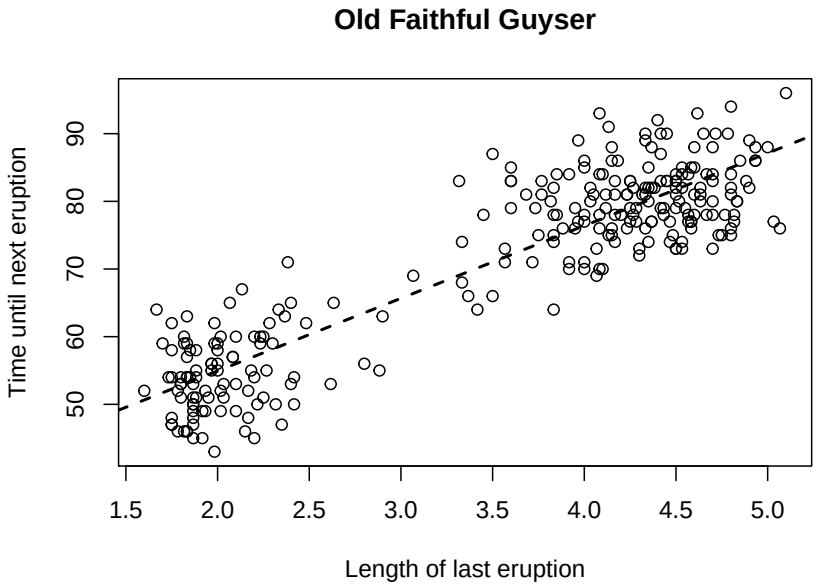
2.11.6 Scatterplot

```
plot(faithful$Eruptions,
     faithful$Waiting.Time,
     xlab = "Length of last eruption",
     ylab = "Time until next eruption",
     main = "Old Faithful Guyser")
```



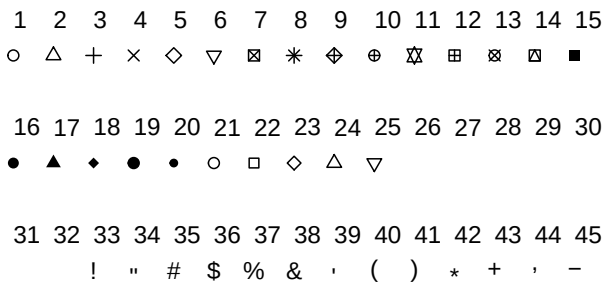
Let's add the least squares regression fit to the graph:

```r
plot(faithful$Eruptions,
     faithful$Waiting.Time,
     xlab = "Length of last eruption",
     ylab = "Time until next eruption",
     main = "Old Faithful Guyser")
abline(lm(Waiting.Time~Eruptions,
          data=faithful), lwd=2, lty=2)
```



There are a lot of different plotting symbols:

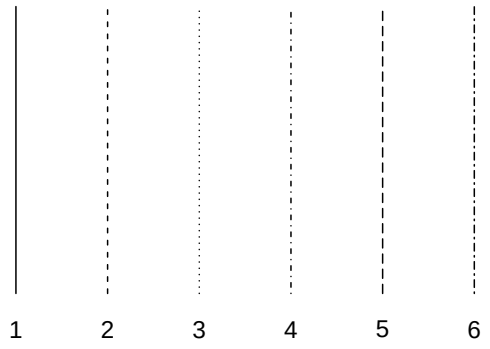
```r
plot(0:16, rep(0, 17), ylim=c(0,1),
     type="n", xlab="", ylab="", axes=F)
for(i in 1:15) {
  for(k in 1:3) {
    points(i, c(0.8, 0.5, 0.2)[k], pch=i+(k-1)*15)
    text(i, c(0.9, 0.6, 0.3)[k], label=i+(k-1)*15, adj = 0)
  }
}
```



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)
}
```



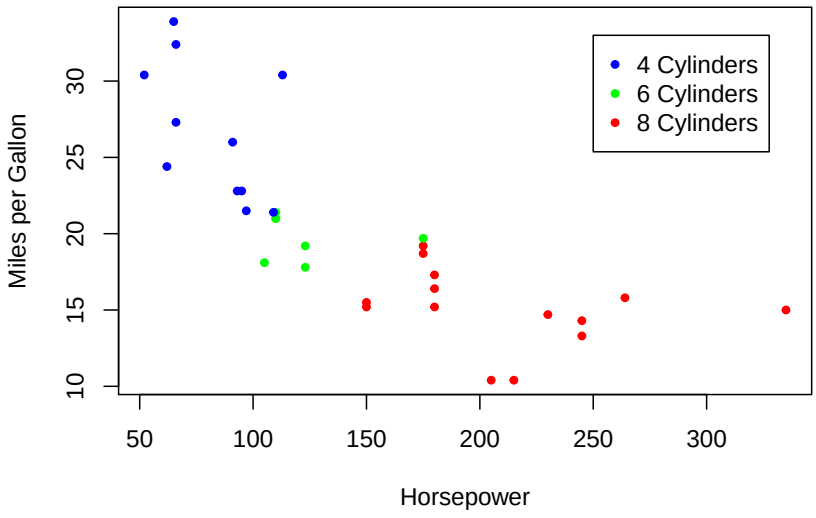
Consider the data set *mtcars*, which is part of base R. It has data on 32 different cars:

```
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
```

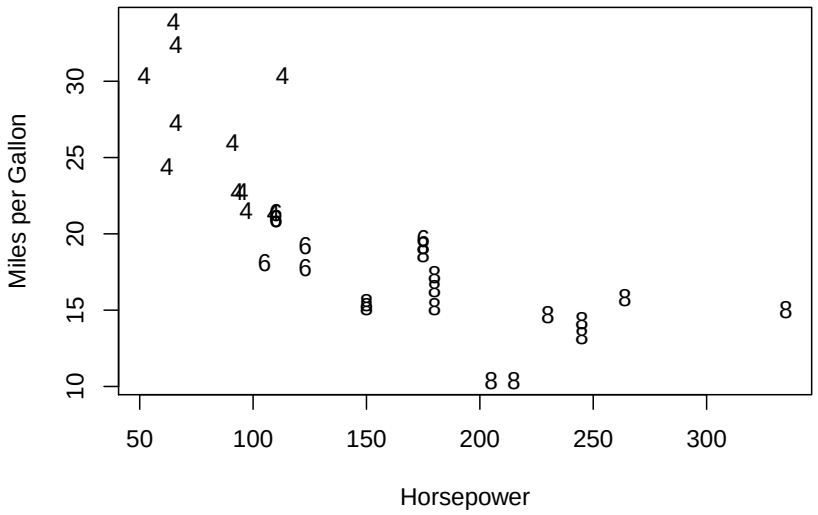
say we wish to study the relationship of mpg (Miles per Gallon) and hp (Horsepower) but also include information on the number of cylinders (either 4, 6 or 8):

```
cols <- rep("blue", 32)
cols[mtcars$cyl==6] <- "green"
cols[mtcars$cyl==8] <- "red"
plot(mtcars$hp, mtcars$mpg,
     xlab = "Horsepower",
     ylab = "Miles per Gallon",
     col = cols, pch = 20)
legend(250, 33,
       paste(c(4, 6, 8), "Cylinders"),
       pch = 20,
       col = c("blue", "green", "red"))
```



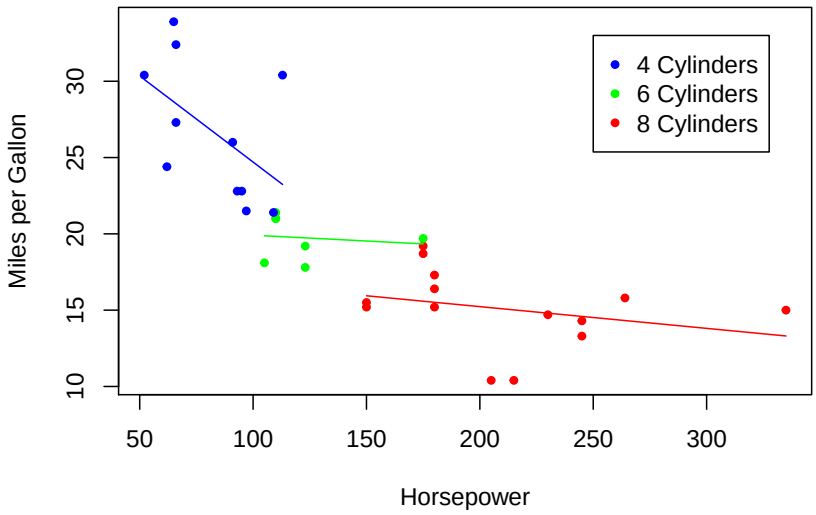
Exercise

Do the graph again, but use the cylinder numbers as plotting symbols:



Let's add the least squares regression lines for each cylinder separately. We could use the *abline(fit)* command, but notice that the hp ranges for the three cylinder types are quite different. Using abline would draw the fit all the way across the graph. Here is a better solution:

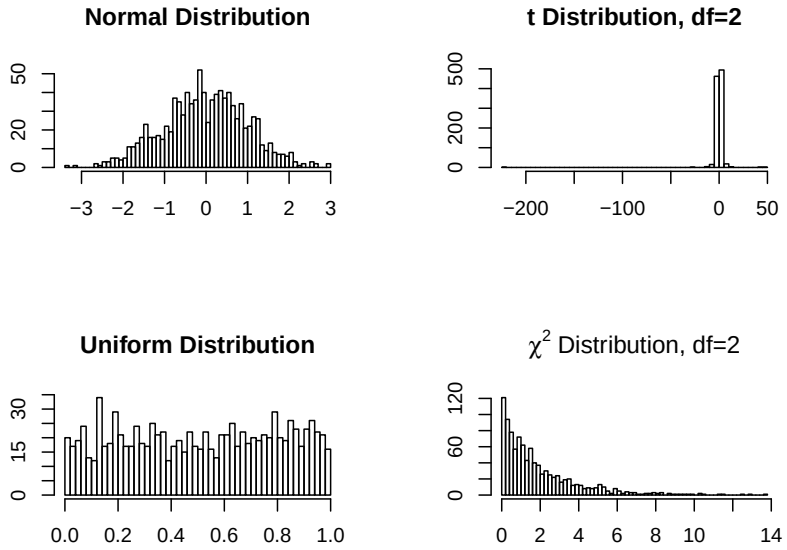
```
cls <- c("blue", "green", "red")
plot(mtcars$hp, mtcars$mpg,
     xlab = "Horsepower",
     ylab = "Miles per Gallon",
     col = cols, pch = 20)
legend(250, 33,
       paste(c(4, 6, 8), "Cylinders"),
       pch = 20,
       col = cls)
for(i in c(4, 6, 8)) {
  fit <- lm(mpg[cyl==i]~hp[cyl==i],
            data=mtcars)
  x <- range(mtcars$hp[mtcars$cyl==i])
  y <- coef(fit)[1] + coef(fit)[2]*x
  segments(x[1], y[1], x[2], y[2],
           col=cls[(i-2)/2])
}
```



2.11.7 Multiple Graphs

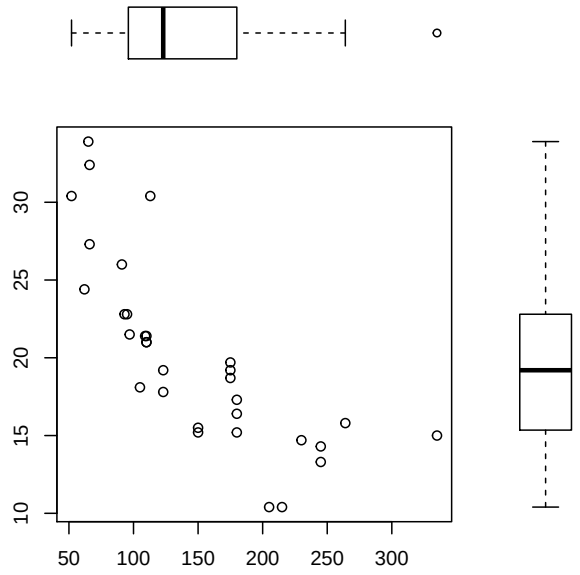
sometimes we want to combine several graphs into one

```
par(mfrow=c(2, 2))
hist(rnorm(1000), 50,
     main="Normal Distribution",
     ylab="", xlab="")
hist(rt(1000, 2), 50,
     main="t Distribution, df=2",
     ylab="", xlab="")
hist(runif(1000), 50,
     main="Uniform Distribution",
     ylab="", xlab="")
hist(rchisq(1000, 2), 50,
     main = expression(paste(chi^2,
            " Distribution, df=2")),
     ylab="", xlab="")
```



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(mtcars$hp, mtcars$mpg)
par(mar = c(0, 3, 1, 1))
boxplot(mtcars$hp,
        axes = FALSE, horizontal = TRUE)
par(mar = c(3, 0, 1, 1))
boxplot(mtcars$mpg, axes = FALSE)
```



2.11.8 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.

So I might have the following function to do the graph above:

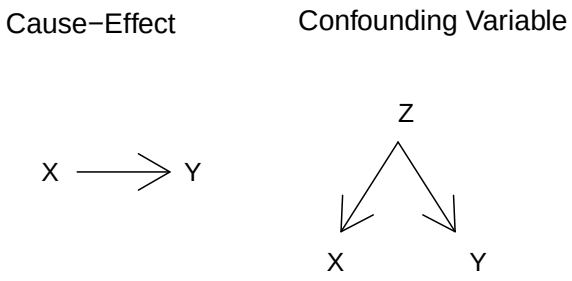
```
figure1 <- function() {
  layout(matrix(c(2, 0, 1, 3), 2, 2,
                byrow = TRUE),
         c(3, 1), c(1, 3), TRUE)
  par(mar = c(3, 3, 1, 1))
  plot(hp, mpg)
  par(mar = c(0, 3, 1, 1))
  boxplot(hp, axes = FALSE, horizontal = TRUE)
  par(mar = c(3, 0, 1, 1))
  boxplot(mpg, axes = FALSE)

}
```

Note these days I often do this with Rmarkdown instead.

2.11.9 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:



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)
```

People have done some very funny things with this. Run this code in the console (you need to install the library *cowsay* first)

```
library(cowsay)
say("Hello world!", by = "cow")
```

2.11.10 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.

2.12 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

$$Y = \beta_0 + \beta_1 x + \epsilon$$

Sometimes 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 two predictors you can

- fit an *additive* model of the form

$$Y = \beta_0 + \beta_1 x + \beta_2 z + \epsilon$$

with

```
lm(y ~ x + z)
```

- fit a model with an *interaction* term

$$Y = \beta_0 + \beta_1 x + \beta_2 z + \beta_3 x \times z + \epsilon$$

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

$$Y_i = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i,j=1}^{n} \beta_{ij} x_i x_j + + \ \ ... \ \ + \beta_{1..n} x_1 \times .. \times x_n + \epsilon$$

```
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)
```

if there are more than a few predictors it is usually easier to generate a matrix of predictors:

```
X <- cbind(x1, x2, x3, x4)
lm(y ~ X)
```

2.12.0.1 Case Study: House Prices

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:

```
summary(lm(Price ~ Sqfeet +
             Feature + Corner + Tax,
           data=albuquerquehouseprice))
```

```
##
## Call:
## lm(formula = Price ~ Sqfeet + Feature + Corner + Tax, data = albuquerquehouseprice)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -541.9  -73.7  -12.9   66.7  617.7
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  76.6016    60.7064    1.26    0.210
## Sqfeet        0.2668     0.0617    4.33  3.5e-05
## Feature      13.6326    13.2901    1.03    0.307
## Corner      -89.0334    42.2067   -2.11    0.037
## Tax           0.6619     0.1088    6.08  2.1e-08
##
## Residual standard error: 171 on 102 degrees of freedom
##   (10 observations deleted due to missingness)
## Multiple R-squared:  0.809,  Adjusted R-squared:  0.802
## F-statistic:  108 on 4 and 102 DF,  p-value: <2e-16
```

- additive model, Sqfeet and Features

```
summary(lm(Price ~ Sqfeet + Feature,
           data=albuquerquehouseprice))
```

```
##
## Call:
## lm(formula = Price ~ Sqfeet + Feature, data = albuquerquehouseprice)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1005.4   -99.1    -3.2    75.9   782.0
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -1.559     67.296   -0.02    0.982
## Sqfeet         0.584      0.039   14.98   <2e-16
## Feature       27.786     14.534    1.91    0.058
##
## Residual standard error: 202 on 114 degrees of freedom
## Multiple R-squared:  0.723,  Adjusted R-squared:  0.718
## F-statistic:  148 on 2 and 114 DF,  p-value: <2e-16
```

- model with interaction, Sqfeet and Features

```
summary(lm(Price ~ Sqfeet * Feature,
           data=albuquerquehouseprice))
```

```
##
## Call:
## lm(formula = Price ~ Sqfeet * Feature, data = albuquerquehouseprice)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -987.9  -98.8   -0.5   85.0  834.6
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    396.3758   172.4254    2.30   0.0234
## Sqfeet           0.3203     0.1124    2.85   0.0052
## Feature        -75.5710    43.7668   -1.73   0.0870
## Sqfeet:Feature   0.0658     0.0264    2.50   0.0140
##
## Residual standard error: 198 on 113 degrees of freedom
## Multiple R-squared:  0.737,  Adjusted R-squared:  0.73
## F-statistic:  106 on 3 and 113 DF,  p-value: <2e-16
```

- model with pairwise interactions:

```
summary(lm(Price ~ (Sqfeet + Feature +
                      Corner +  Tax)^2,
           data=albuquerquehouseprice))
```

```
##
## Call:
## lm(formula = Price ~ (Sqfeet + Feature + Corner + Tax)^2, data = albuquerquehousepric
##
## Residuals:
```

```
##    Min     1Q Median     3Q    Max
## -383.9  -82.0   -2.7   56.1  644.3
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2.64e+02   1.88e+02    1.40   0.1639
## Sqfeet          2.20e-01   2.01e-01    1.10   0.2759
## Feature        -3.65e+01   4.60e+01   -0.79   0.4297
## Corner          4.37e+02   1.56e+02    2.80   0.0061
## Tax             3.18e-01   3.20e-01    0.99   0.3235
## Sqfeet:Feature  4.62e-02   4.44e-02    1.04   0.3011
## Sqfeet:Corner  -3.96e-01   1.35e-01   -2.94   0.0041
## Sqfeet:Tax      1.11e-04   1.06e-04    1.04   0.2997
## Feature:Corner -1.88e+01   3.63e+01   -0.52   0.6062
## Feature:Tax    -3.49e-02   6.53e-02   -0.53   0.5939
## Corner:Tax      2.51e-01   3.33e-01    0.75   0.4529
##
## Residual standard error: 157 on 96 degrees of freedom
##   (10 observations deleted due to missingness)
## Multiple R-squared:  0.849,  Adjusted R-squared:  0.833
## F-statistic:   54 on 10 and 96 DF,  p-value: <2e-16
```

- model with all possible terms:

```
summary(lm(Price ~ (Sqfeet + Feature +
                      Corner +  Tax)^4,
           data=albuquerquehouseprice))
```

```
##
## Call:
## lm(formula = Price ~ (Sqfeet + Feature + Corner + Tax)^4, data = albuquerquehousepric
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -379.6  -80.1    4.7   55.2  650.8
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                4.81e+02   3.68e+02    1.31     0.19
## Sqfeet                     4.32e-02   2.89e-01    0.15     0.88
## Feature                   -8.41e+01   9.76e+01   -0.86     0.39
## Corner                     2.49e+03   2.26e+03    1.10     0.27
## Tax                        2.61e-01   5.25e-01    0.50     0.62
## Sqfeet:Feature             8.54e-02   6.73e-02    1.27     0.21
## Sqfeet:Corner             -1.94e+00   1.59e+00   -1.22     0.23
## Sqfeet:Tax                 1.93e-04   2.55e-04    0.76     0.45
## Feature:Corner            -7.82e+02   6.42e+02   -1.22     0.23
## Feature:Tax               -3.02e-02   1.35e-01   -0.22     0.82
```

```
## Corner:Tax                -2.41e+00   2.48e+00   -0.97     0.33
## Sqfeet:Feature:Corner      5.43e-01   4.33e-01    1.26     0.21
## Sqfeet:Feature:Tax        -1.47e-05   5.93e-05   -0.25     0.80
## Sqfeet:Corner:Tax          1.93e-03   1.46e-03    1.32     0.19
## Feature:Corner:Tax         9.28e-01   7.04e-01    1.32     0.19
## Sqfeet:Feature:Corner:Tax -6.34e-04   3.96e-04   -1.60     0.11
##
## Residual standard error: 153 on 91 degrees of freedom
##   (10 observations deleted due to missingness)
## Multiple R-squared:  0.863,  Adjusted R-squared:  0.841
## F-statistic: 38.3 on 15 and 91 DF,  p-value: <2e-16
```

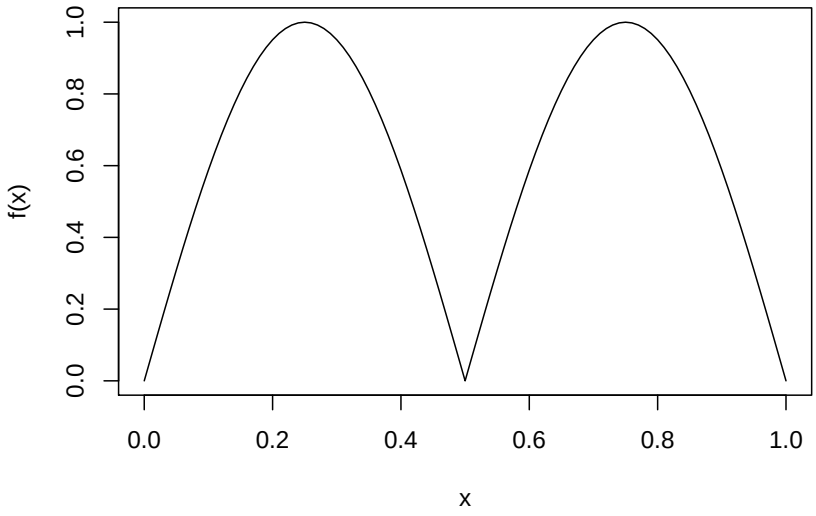
2.13 Numerical Methods

R has a number of routines useful for numerical analysis

2.13.1 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)
```

```
## 0.6366 with absolute error < 7e-15
```

the routine returns a list, usually we only want the value of the integral, so

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

```
## [1] 0.6366
```

integrate allows us to pass additional arguments to f with the ... convention:

```
integrate(f, 0, 1, a=1.4)$value
```

```
## [1] 0.6118
```

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] 0.6118
```

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] 0.6366
```

- 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] 0.6366
```

or any other standard numerical integration formula, see for example Numerical Integration Formulas

2.13.2 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] 5.441
```

Is this right? Let's check:

$$\exp(2(-x^2+xy-y^2)/3) = 2\pi\sqrt{1-(1/2)^2}\left[\frac{1}{2\pi\sqrt{1-(1/2)^2}}\exp\left\{-\frac{1}{2(1-(1/2)^2}\left(x^2-2xy(1/2)+y^2\right)\right\}\right]$$

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] 5.441
```

2.13.3 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] -0.3431
```

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] -10.67
```

Example

The Taylor polynomial of order k of a function f at a point x=a is defined by

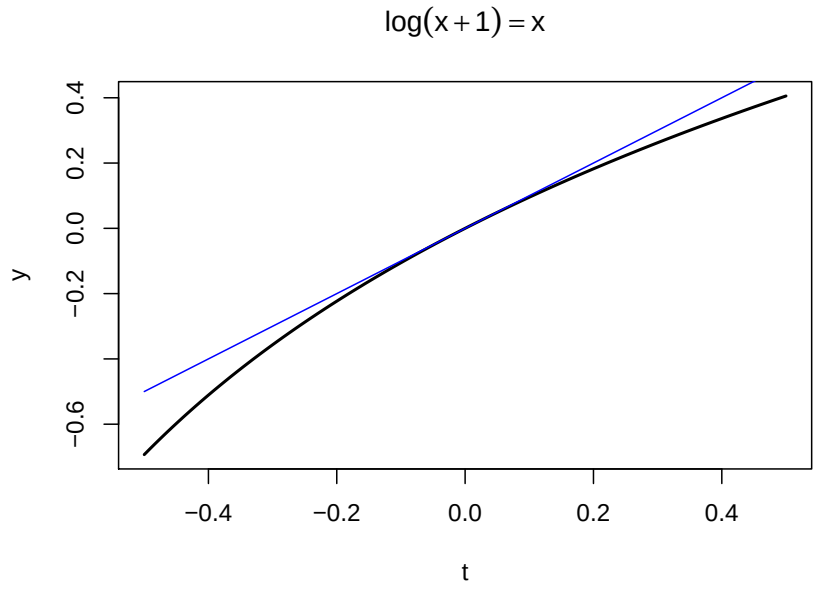
$$T(x;a,k) = \sum_{i=0}^{k} \frac{d^i f}{dx^i}(a)(x-a)^i$$

Let's write a routine that draws the function and it's Taylor polynomial of order k:

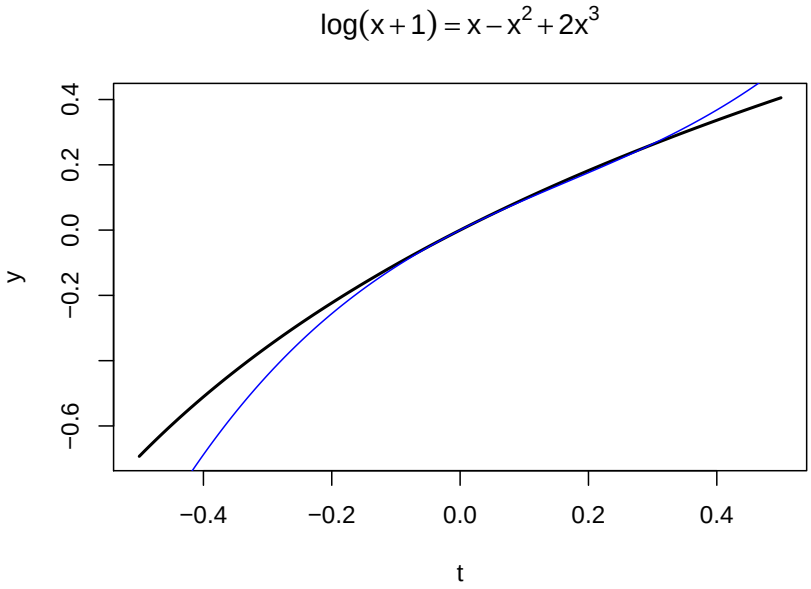
```
taylor <- function(f, a, k=1, from, to) {
  expr <- parse(text=f)
# x and y coordinates of function
  x <- seq(from, to, length=250)
  y <- eval(expr)
  t <- x
# x value of interest a
  x <- a
  z <- eval(expr)
  taylor.coefficients <- rep(z, k+1)
# create text version of Taylor polynomial
  x.text <- 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, "", paste0(abs(tmp),"*")),
          x.text,
          ifelse(i==1, "", paste0("^", i)))
     }
     nosgn <- FALSE
  }
  plot(t, y, type="l", lwd=2,
       main=parse(text=ttl))
  lines(t, z, col="blue")

}
taylor("log(x+1)", a=0, k=1, -0.5, 0.5)
```

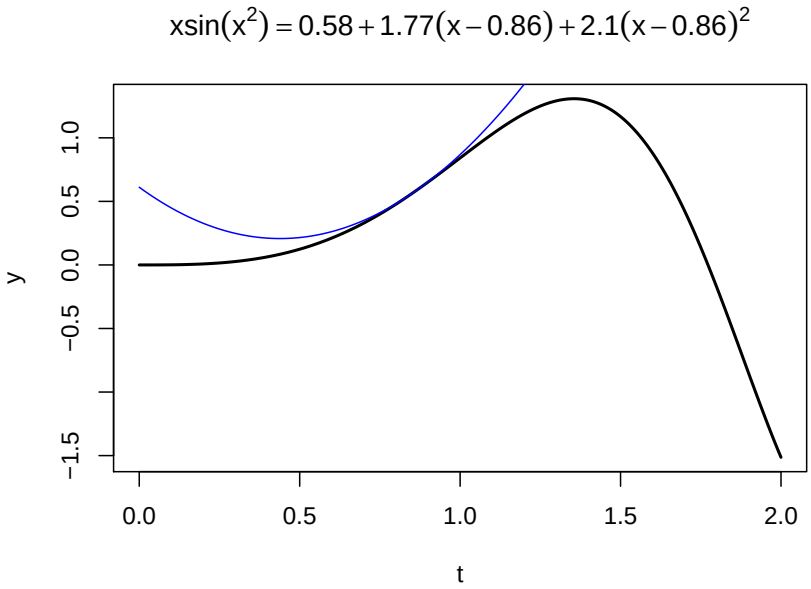



```
taylor("log(x+1)", a=0, k=3, -0.5, 0.5)
```

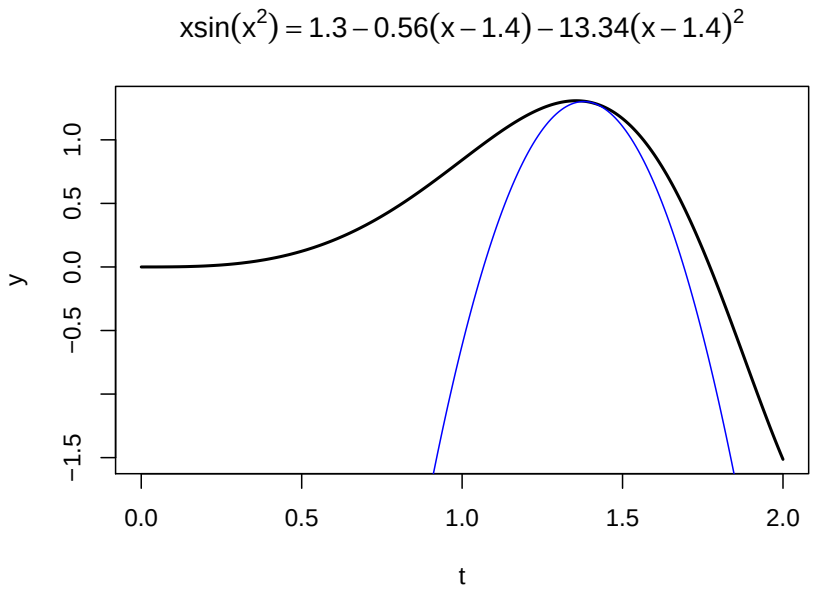


Here is another example

```
taylor("x*sin(x^2)", a=0.86, k=2, from=0, to=2)
```

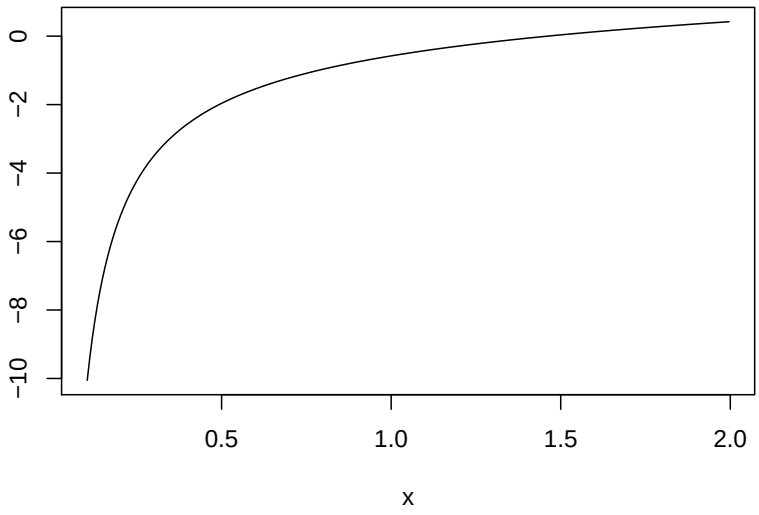



```
taylor("x*sin(x^2)", a=1.4, k=2, from=0, to=2)
```

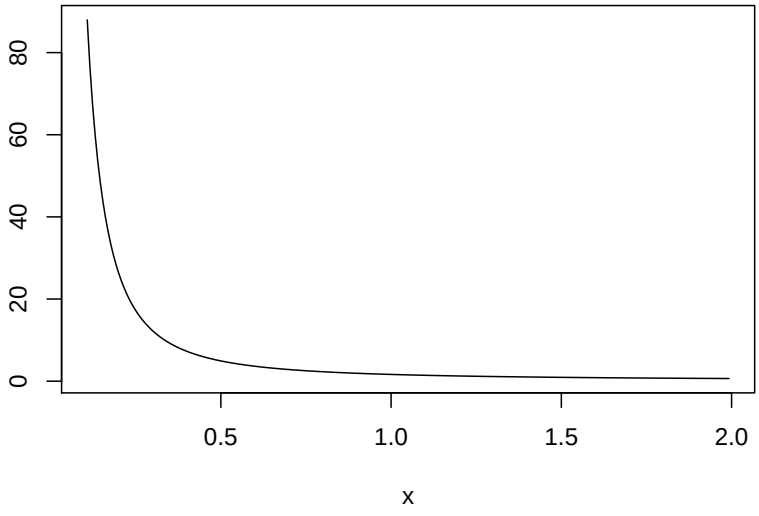


The D function only works on functions that R recognizes. For other cases you might have to write your own:

```
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="")
```

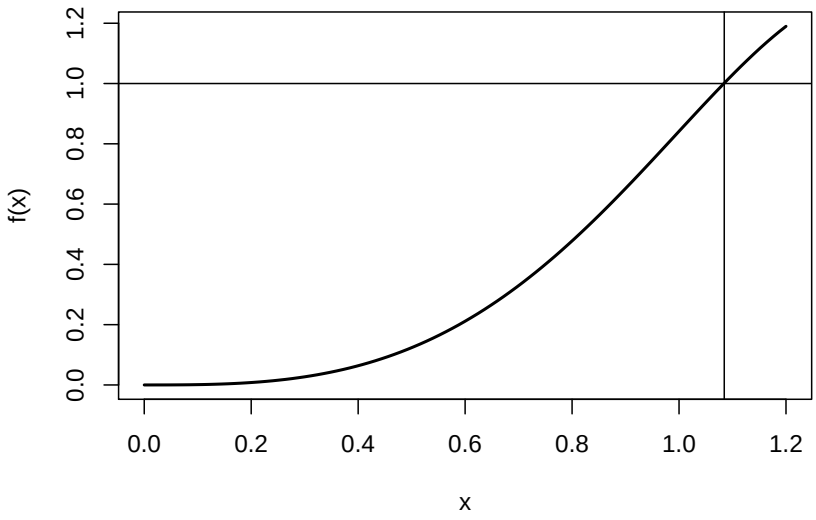


2.13.4 Root Finding

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:

- 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[which.min(abs(y-1))]
abline(v=y0)
```

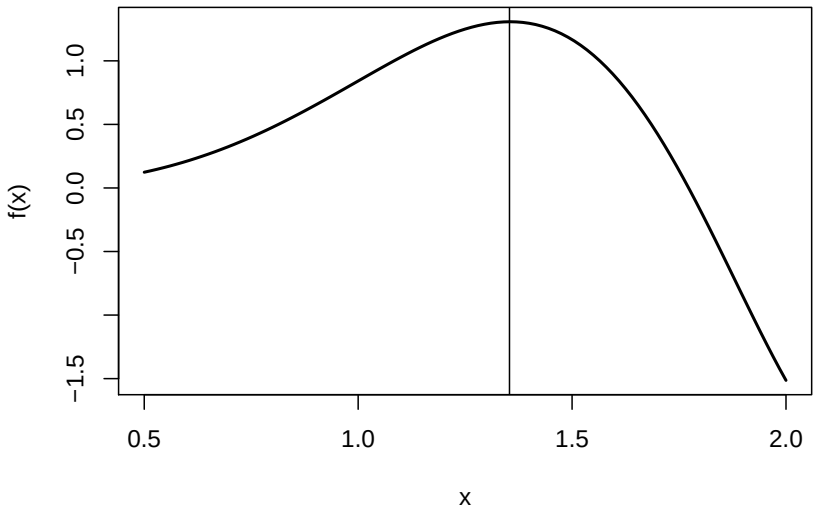



```
y0
```

```
## [1] 1.085
```

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[which.min(abs(y.prime))]
abline(v=x0)
```




```
c(x0, f(x0))
```

```
## [1] 1.354 1.308
```

An alternative is to use *optimize*. By default it finds minima, so

```
optimize(f, c(0.5, 2), maximum = TRUE)
```

```
## $maximum
## [1] 1.355
##
## $objective
## [1] 1.308
```

- *uniroot* command

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] 1.084
```

- *polyroot* command

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]  0.8774+0.7449i -0.7549-0.0000i -1.0000+0.0000i  0.8774-0.7449i
```

this gives 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] -0.7549 -1.0000
```

2.13.5 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] -1
##
## $estimate
## [1] 1 2
##
## $gradient
## [1]  1.521e-07 -7.605e-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] -1
##
## $estimate
## [1] 1 2
```

```
##
## $gradient
## [1]  1.521e-07 -7.605e-08
##
## $hessian
##            [,1]       [,2]
## [1,]  2.000e+00 -1.665e-08
## [2,] -1.665e-08  2.000e+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.

Example

A very important object is Statistics is the *log-likelihood function*. Say we have observations $x_1, .., x_n$ from some density $f(x; \theta)$, where theta is some parameter (vector). Then the log-likelihood function is defined by

$$l(\theta) = \sum_{i=1}^{n} \log f(x_i; \theta)$$

A standard way to estimate θ is using the method of *maximum likelihood*, which finds the maximum of the likelihood function.

Say $X \sim$ Bernoulli(θ), then

$$\begin{aligned} f(x; \theta) &= \theta^x (1-\theta)^{1-x} \\ \log f(x; \theta) &= x \log \theta + (1-x) \log(1-\theta) \\ l(\theta) &= \sum_{i=1}^{n} x \log \theta + (1-x) \log(1-\theta) \end{aligned}$$

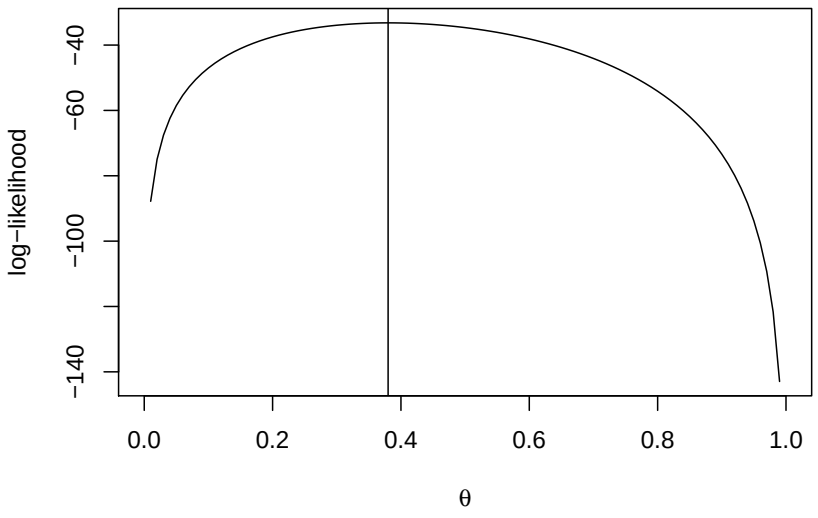
Here is an example of what this might look like:

```
x <- sample(c(0, 1), size=50,
            replace=TRUE, prob = c(2, 1))
ll <- function(theta, x) {
  y <- length(theta)
  for(i in seq_along(theta))
    y[i] <- sum(x*log(theta[i]) +
                (1-x)*log(1-theta[i]))
```

```
  y
}
theta <- seq(0, 1, 0.01)
plot(theta, ll(theta, x=x),
     type="l",
     xlab=expression(theta),
     ylab="log-likelihood")
fn <- function(z) -ll(z, x=x)
# needs function to minimize
tmp <- optim(0.5, fn, lower=0.1, upper=0.9)
unlist(tmp)
```

```
##                                                 par
##                                "0.380000338209983"
##                                               value
##                                "33.2032063282175"
##                                   counts.function
##                                                 "6"
##                                   counts.gradient
##                                                 "6"
##                                       convergence
##                                                 "0"
##                                             message
## "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

```
abline(v=tmp$par)
```



Say $X \sim$ Normal(μ, σ), then

```
x <- rnorm(50, 5, 2)
ll <- function(theta, x) {
  -sum(log(dnorm(x, theta[1], theta[2])))
}
tmp <- optim(c(0, 1), ll, lower=c(-Inf, 0),
             upper=c(Inf, Inf), x=x)
tmp$par
```

```
## [1] 5.345 2.113
```

2.14 Environments and Libraries

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"             "wine"
##  [3] "houseprice"             "gasoline"
##  [5] "elusage"                "mothers"
##  [7] "draft"                  "newcomb"
##  [9] "package:parallel"       "mtcars"
## [11] "package:RCurl"          "package:bitops"
## [13] "package:combinat"       "package:leaps"
## [15] "package:bootstrap"      "package:maxLik"
## [17] "package:miscTools"      "package:modelr"
## [19] "package:gapminder"      "package:nycflights13"
## [21] "package:lubridate"      "package:tidytext"
## [23] "package:gutenbergr"     "package:babynames"
## [25] "package:bindrcpp"       "package:magrittr"
## [27] "package:pdftools"       "package:data.table"
## [29] "package:rio"            "package:gpuR"
## [31] "package:testlib"        "package:roxygen2"
## [33] "package:usethis"        "package:devtools"
## [35] "package:pryr"           "package:maps"
## [37] "package:Hmisc"          "package:Formula"
## [39] "package:survival"       "package:lattice"
## [41] "package:microbenchmark" "package:mvtnorm"
## [43] "package:random"         "wine"
## [45] "package:kableExtra"     "package:forcats"
## [47] "package:stringr"        "package:dplyr"
## [49] "package:purrr"          "package:readr"
## [51] "package:tidyr"          "package:tibble"
## [53] "package:tidyverse"      "package:knitr"
```

```
## [55] "package:bookdown"       "tools:rstudio"
## [57] "package:stats"          "package:graphics"
## [59] "package:grDevices"      "package:utils"
## [61] "package:datasets"       ".MyEnv"
## [63] "package:moodler"        "package:wolfr"
## [65] "package:shiny"          "package:Rcpp"
## [67] "package:grid"           "package:ggplot2"
## [69] "package:methods"        "Autoloads"
## [71] "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/wine/houseprice/gasoline/elusage/mothers/draft/newcomb/package:parall
```

This has the following effect. Say you type

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

```
## [1] 0.5682
```

What has R just done? First it created an object called "x", and stored it in the folder ".GlobalEnv". We can check:

```
ls()
```

```
##   [1] "%!in%"                "%+b%"
##   [3] "a"                    "A"
##   [5] "acorn"                "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"          "calc.power"
##  [35] "calcium"              "cancersurvival"
##  [37] "cardeaths"            "cateyes"
##  [39] "cats"                 "causeofdeath"
##  [41] "centre"               "change.order"
##  [43] "check.packages"       "cheese"
```

```
##  [45] "chi.gof.test"          "chi.ind.test"
##  [47] "chromatography"        "church"
##  [49] "ci.mean.sim"           "classsurvey"
##  [51] "cleanup"               "clouds"
##  [53] "cls"                   "clt.illustration"
##  [55] "cols"                  "company"
##  [57] "cont.table"            "correlation"
##  [59] "counter"               "countries"
##  [61] "crop"                  "cuckoo"
##  [63] "culture"               "d"
##  [65] "decimal.2.binary"      "df"
##  [67] "df1"                   "diamond"
##  [69] "dlr"                   "dlr.predict"
##  [71] "doGraph"               "doTbl"
##  [73] "double.integral"       "dp"
##  [75] "draft"                 "draw.cause.effect"
##  [77] "draw.hist"             "dropcourse"
##  [79] "drownings"             "drugaddiction"
##  [81] "elusage"               "esh"
##  [83] "esh1"                  "ethanol"
##  [85] "euros"                 "examscores"
##  [87] "exp1"                  "f"
##  [89] "f.double.prime"        "f.prime"
##  [91] "f1"                    "fA"
##  [93] "fA.vec"                "fabricwear"
##  [95] "faithful"              "fAvec"
##  [97] "fermentation"          "fiber"
##  [99] "file_list"             "filmcoatings"
## [101] "find.data.set"         "fish"
## [103] "fit"                   "fivenumber"
## [105] "flammability"          "flplot"
## [107] "fn"                    "fontcolor"
## [109] "foods"                 "forbes"
## [111] "friday13"              "galton"
## [113] "gasoline"              "gehan"
## [115] "gen.cont.table.data"   "Gender"
## [117] "GenderCode"            "gestation"
## [119] "get.moodle.data"       "getfun"
## [121] "gettysburg"            "getx"
## [123] "ggMarginal"            "ggQQ"
## [125] "golfscores"            "GPA"
## [127] "graph.out"             "gunsandmurder"
## [129] "h"                     "headache"
## [131] "hearingaid"            "highways"
## [133] "homework"              "horsekicks"
## [135] "hostility"             "hotdogs"
```

```
## [137] "houseprice"           "hpd.beta"
## [139] "hplot"                "hubble"
## [141] "hyptest.helper"       "i"
## [143] "I"                    "ibayesprop"
## [145] "ID"                   "idataio"
## [147] "igetdata"             "ihist"
## [149] "ihplot"               "info"
## [151] "inormal"              "instructor"
## [153] "intplot"              "iplot"
## [155] "iplotone"             "iq"
## [157] "iqandsize"            "is.binary"
## [159] "isCat"                "isplot"
## [161] "isubset"              "isummary"
## [163] "itemplate"            "j"
## [165] "k"                    "kable.nice"
## [167] "kbins"                "krun"
## [169] "kruskalwallis"        "kyphosis"
## [171] "larvea"               "leukemia"
## [173] "license.plate"        "ll"
## [175] "lobster"              "logfac"
## [177] "longjump"             "ltrs"
## [179] "lunatics"             "lung.cancer"
## [181] "lvls"                 "mallows"
## [183] "mannwhitney"          "mcat"
## [185] "mendel"               "mid"
## [187] "mines"                "mlr"
## [189] "mlr.predict"          "mothers"
## [191] "moths"                "mplot"
## [193] "multiple.graphs"      "mydates"
## [195] "mylist"               "n"
## [197] "nested.models.test"   "new.yml"
## [199] "newcomb"              "normal.check"
## [201] "normal.ex"            "nplot"
## [203] "num_a"                "num_vowels"
## [205] "olympics"             "one.sample.prop"
## [207] "one.sample.t"         "one.sample.wilcoxon"
## [209] "one.time.setup"       "oneway"
## [211] "painters"             "pcbtrout"
## [213] "pearson.cor"          "plot.fn"
## [215] "plot.fn0"             "plot.sales"
## [217] "plot.salestime"       "pop1950x2010"
## [219] "popular"              "popularvote"
## [221] "population"           "positions"
## [223] "positions_a"          "predictors"
## [225] "print.binary"         "prop.ps"
## [227] "pval"                 "race.table"
```

```
## [229] "ratings"               "report"
## [231] "results"               "rev.wrds"
## [233] "rice"                  "rock.oil"
## [235] "rocks"                 "rogaine"
## [237] "rpit"                  "run.app"
## [239] "salaries"              "sales.data"
## [241] "sales.time.data"       "salesperson"
## [243] "sc"                    "seatbelt"
## [245] "setup.rmd"             "sexratio"
## [247] "shoesales"             "shuttle"
## [249] "sim1"                  "sim2"
## [251] "singer"                "skulls"
## [253] "sleep"                 "slr"
## [255] "slr.predict"           "smoking"
## [257] "smokingandjob"         "solder"
## [259] "splot"                 "st.y"
## [261] "stamps"                "stat.table"
## [263] "states"                "stats"
## [265] "stats.sales"           "stories"
## [267] "student.levels"        "students"
## [269] "students.fac"          "students.ord"
## [271] "studentsurvey"         "studyhabits"
## [273] "sugar"                 "summary"
## [275] "summary.binary"        "sv"
## [277] "t.ps"                  "taylor"
## [279] "tbl.students.fac"      "test.mean.sim"
## [281] "test_sim"              "testscores"
## [283] "theta"                 "titanic"
## [285] "tmp"                   "tms"
## [287] "trout"                 "tukey"
## [289] "tv"                    "twoway"
## [291] "txt"                   "u"
## [293] "upr"                   "us.population.2010"
## [295] "usa"                   "usmap"
## [297] "ustemperature"         "v"
## [299] "vowels"                "wilcoxon"
## [301] "wine"                  "wine.all"
## [303] "world.mortality.2017"  "worldpopulation"
## [305] "worldseries"           "wrds"
## [307] "wrinccensus"           "wrincsample"
## [309] "x"                     "x0"
## [311] "x1"                    "xyz"
## [313] "y"                     "y.2.prime"
## [315] "y.prime"               "y0"
## [317] "y1"                    "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 "wine", 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")
```

```
## character(0)
```

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: 0x0000021382ba86a8>
```

```
where("mean")
```

```
## <environment: namespace: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)
mean(x)
```

```
## [1] 0.5682
```

```
rm(x)
mean(x)
```

```
## [1] 5.5
```

```
search()[1:3]
```

```
## [1] ".GlobalEnv" "my.data"    "wine"
```

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)
```

2.14.1 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: 0x00000213c3259a40>
##
## $parent
## <environment: 0x0000021382ba86a8>
##
## $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: 0x0000021382ba86a8>
```

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: 0x00000213bdce2918>
##
## $parent
## <environment: 0x00000213bdce25d0>
##
## $objects
##
## $ran.in
## <environment: 0x00000213bdce25d0>
##
## $parent
## <environment: 0x0000021382ba86a8>
##
## $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] "%!in%"                  "%+b%"
```

```
##   [3] "a"                     "A"
##   [5] "acorn"                 "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"           "calc.power"
##  [35] "calcium"               "cancersurvival"
##  [37] "cardeaths"             "cateyes"
##  [39] "cats"                  "causeofdeath"
##  [41] "centre"                "change.order"
##  [43] "check.packages"        "cheese"
##  [45] "chi.gof.test"          "chi.ind.test"
##  [47] "chromatography"        "church"
##  [49] "ci.mean.sim"           "classsurvey"
##  [51] "cleanup"               "clouds"
##  [53] "cls"                   "clt.illustration"
##  [55] "cols"                  "company"
##  [57] "cont.table"            "correlation"
##  [59] "counter"               "countries"
##  [61] "crop"                  "cuckoo"
##  [63] "culture"               "d"
##  [65] "decimal.2.binary"      "df"
##  [67] "df1"                   "diamond"
##  [69] "dlr"                   "dlr.predict"
##  [71] "doGraph"               "doTbl"
##  [73] "double.integral"       "dp"
##  [75] "draft"                 "draw.cause.effect"
##  [77] "draw.hist"             "dropcourse"
##  [79] "drownings"             "drugaddiction"
##  [81] "elusage"               "esh"
##  [83] "esh1"                  "ethanol"
##  [85] "euros"                 "examscores"
##  [87] "exp1"                  "f"
##  [89] "f.double.prime"        "f.prime"
##  [91] "f1"                    "fA"
##  [93] "fA.vec"                "fabricwear"
```

```
##  [95] "faithful"              "fAvec"
##  [97] "fermentation"          "fiber"
##  [99] "file_list"             "filmcoatings"
## [101] "find.data.set"         "fish"
## [103] "fit"                   "fivenumber"
## [105] "flammability"          "flplot"
## [107] "fn"                    "fontcolor"
## [109] "foods"                 "forbes"
## [111] "friday13"              "galton"
## [113] "gasoline"              "gehan"
## [115] "gen.cont.table.data"   "Gender"
## [117] "GenderCode"            "gestation"
## [119] "get.moodle.data"       "getfun"
## [121] "gettysburg"            "getx"
## [123] "ggMarginal"            "ggQQ"
## [125] "golfscores"            "GPA"
## [127] "graph.out"             "gunsandmurder"
## [129] "h"                     "headache"
## [131] "hearingaid"            "highways"
## [133] "homework"              "horsekicks"
## [135] "hostility"             "hotdogs"
## [137] "houseprice"            "hpd.beta"
## [139] "hplot"                 "hubble"
## [141] "hyptest.helper"        "i"
## [143] "I"                     "ibayesprop"
## [145] "ID"                    "idataio"
## [147] "igetdata"              "ihist"
## [149] "ihplot"                "info"
## [151] "inormal"               "instructor"
## [153] "intplot"               "iplot"
## [155] "iplotone"              "iq"
## [157] "iqandsize"             "is.binary"
## [159] "isCat"                 "isplot"
## [161] "isubset"               "isummary"
## [163] "itemplate"             "j"
## [165] "k"                     "kable.nice"
## [167] "kbins"                 "krun"
## [169] "kruskalwallis"         "kyphosis"
## [171] "larvea"                "leukemia"
## [173] "license.plate"         "ll"
## [175] "lobster"               "logfac"
## [177] "longjump"              "ltrs"
## [179] "lunatics"              "lung.cancer"
## [181] "lvls"                  "mallows"
## [183] "mannwhitney"           "mcat"
## [185] "mendel"                "mid"
```

```
## [187] "mines"                "mlr"
## [189] "mlr.predict"          "mothers"
## [191] "moths"                "mplot"
## [193] "multiple.graphs"      "my.data"
## [195] "mydates"              "mylist"
## [197] "n"                    "nested.models.test"
## [199] "new.yml"              "newcomb"
## [201] "normal.check"         "normal.ex"
## [203] "nplot"                "num_a"
## [205] "num_vowels"           "olympics"
## [207] "one.sample.prop"      "one.sample.t"
## [209] "one.sample.wilcoxon"  "one.time.setup"
## [211] "oneway"               "painters"
## [213] "pcbtrout"             "pearson.cor"
## [215] "plot.fn"              "plot.fn0"
## [217] "plot.sales"           "plot.salestime"
## [219] "pop1950x2010"         "popular"
## [221] "popularvote"          "population"
## [223] "positions"            "positions_a"
## [225] "predictors"           "print.binary"
## [227] "prop.ps"              "pval"
## [229] "race.table"           "ratings"
## [231] "report"               "results"
## [233] "rev.wrds"             "rice"
## [235] "rock.oil"             "rocks"
## [237] "rogaine"              "rpit"
## [239] "run.app"              "salaries"
## [241] "sales.data"           "sales.time.data"
## [243] "salesperson"          "sc"
## [245] "seatbelt"             "setup.rmd"
## [247] "sexratio"             "shoesales"
## [249] "show.env"             "shuttle"
## [251] "sim1"                 "sim2"
## [253] "singer"               "skulls"
## [255] "sleep"                "slr"
## [257] "slr.predict"          "smoking"
## [259] "smokingandjob"        "solder"
## [261] "splot"                "st.y"
## [263] "stamps"               "stat.table"
## [265] "states"               "stats"
## [267] "stats.sales"          "stories"
## [269] "student.levels"       "students"
## [271] "students.fac"         "students.ord"
## [273] "studentsurvey"        "studyhabits"
## [275] "sugar"                "summary"
## [277] "summary.binary"       "sv"
```

```
## [279] "t.ps"                    "taylor"
## [281] "tbl.students.fac"        "test.mean.sim"
## [283] "test_sim"                "testscores"
## [285] "theta"                   "titanic"
## [287] "tmp"                     "tms"
## [289] "trout"                   "tukey"
## [291] "tv"                      "twoway"
## [293] "txt"                     "u"
## [295] "upr"                     "us.population.2010"
## [297] "usa"                     "usmap"
## [299] "ustemperature"           "v"
## [301] "vowels"                  "wilcoxon"
## [303] "wine"                    "wine.all"
## [305] "world.mortality.2017"    "worldpopulation"
## [307] "worldseries"             "wrds"
## [309] "wrinccensus"             "wrincsample"
## [311] "x0"                      "x1"
## [313] "xyz"                     "y"
## [315] "y.2.prime"               "y.prime"
## [317] "y0"                      "y1"
## [319] "z"
```

```
f()
ls()
```

```
##   [1] "%!in%"                   "%+b%"
##   [3] "a"                       "A"
##   [5] "acorn"                   "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"             "calc.power"
##  [35] "calcium"                 "cancersurvival"
##  [37] "cardeaths"               "cateyes"
##  [39] "cats"                    "causeofdeath"
##  [41] "centre"                  "change.order"
##  [43] "check.packages"          "cheese"
```

```
##  [45] "chi.gof.test"          "chi.ind.test"
##  [47] "chromatography"        "church"
##  [49] "ci.mean.sim"           "classsurvey"
##  [51] "cleanup"               "clouds"
##  [53] "cls"                   "clt.illustration"
##  [55] "cols"                  "company"
##  [57] "cont.table"            "correlation"
##  [59] "counter"               "countries"
##  [61] "crop"                  "cuckoo"
##  [63] "culture"               "d"
##  [65] "decimal.2.binary"      "df"
##  [67] "df1"                   "diamond"
##  [69] "dlr"                   "dlr.predict"
##  [71] "doGraph"               "doTbl"
##  [73] "double.integral"       "dp"
##  [75] "draft"                 "draw.cause.effect"
##  [77] "draw.hist"             "dropcourse"
##  [79] "drownings"             "drugaddiction"
##  [81] "elusage"               "esh"
##  [83] "esh1"                  "ethanol"
##  [85] "euros"                 "examscores"
##  [87] "exp1"                  "f"
##  [89] "f.double.prime"        "f.prime"
##  [91] "f1"                    "fA"
##  [93] "fA.vec"                "fabricwear"
##  [95] "faithful"              "fAvec"
##  [97] "fermentation"          "fiber"
##  [99] "file_list"             "filmcoatings"
## [101] "find.data.set"         "fish"
## [103] "fit"                   "fivenumber"
## [105] "flammability"          "flplot"
## [107] "fn"                    "fontcolor"
## [109] "foods"                 "forbes"
## [111] "friday13"              "galton"
## [113] "gasoline"              "gehan"
## [115] "gen.cont.table.data"   "Gender"
## [117] "GenderCode"            "gestation"
## [119] "get.moodle.data"       "getfun"
## [121] "gettysburg"            "getx"
## [123] "ggMarginal"            "ggQQ"
## [125] "golfscores"            "GPA"
## [127] "graph.out"             "gunsandmurder"
## [129] "h"                     "headache"
## [131] "hearingaid"            "highways"
## [133] "homework"              "horsekicks"
## [135] "hostility"             "hotdogs"
```

```
## [137] "houseprice"            "hpd.beta"
## [139] "hplot"                 "hubble"
## [141] "hyptest.helper"        "i"
## [143] "I"                     "ibayesprop"
## [145] "ID"                    "idataio"
## [147] "igetdata"              "ihist"
## [149] "ihplot"                "info"
## [151] "inormal"               "instructor"
## [153] "intplot"               "iplot"
## [155] "iplotone"              "iq"
## [157] "iqandsize"             "is.binary"
## [159] "isCat"                 "isplot"
## [161] "isubset"               "isummary"
## [163] "itemplate"             "j"
## [165] "k"                     "kable.nice"
## [167] "kbins"                 "krun"
## [169] "kruskalwallis"         "kyphosis"
## [171] "larvea"                "leukemia"
## [173] "license.plate"         "ll"
## [175] "lobster"               "logfac"
## [177] "longjump"              "ltrs"
## [179] "lunatics"              "lung.cancer"
## [181] "lvls"                  "mallows"
## [183] "mannwhitney"           "mcat"
## [185] "mendel"                "mid"
## [187] "mines"                 "mlr"
## [189] "mlr.predict"           "mothers"
## [191] "moths"                 "mplot"
## [193] "multiple.graphs"       "my.data"
## [195] "mydates"               "mylist"
## [197] "n"                     "nested.models.test"
## [199] "new.yml"               "newcomb"
## [201] "normal.check"          "normal.ex"
## [203] "nplot"                 "num_a"
## [205] "num_vowels"            "olympics"
## [207] "one.sample.prop"       "one.sample.t"
## [209] "one.sample.wilcoxon"   "one.time.setup"
## [211] "oneway"                "painters"
## [213] "pcbtrout"              "pearson.cor"
## [215] "plot.fn"               "plot.fn0"
## [217] "plot.sales"            "plot.salestime"
## [219] "pop1950x2010"          "popular"
## [221] "popularvote"           "population"
## [223] "positions"             "positions_a"
## [225] "predictors"            "print.binary"
## [227] "prop.ps"               "pval"
```

```
## [229] "race.table"            "ratings"
## [231] "report"                "results"
## [233] "rev.wrds"              "rice"
## [235] "rock.oil"              "rocks"
## [237] "rogaine"               "rpit"
## [239] "run.app"               "salaries"
## [241] "sales.data"            "sales.time.data"
## [243] "salesperson"           "sc"
## [245] "seatbelt"              "setup.rmd"
## [247] "sexratio"              "shoesales"
## [249] "show.env"              "shuttle"
## [251] "sim1"                  "sim2"
## [253] "singer"                "skulls"
## [255] "sleep"                 "slr"
## [257] "slr.predict"           "smoking"
## [259] "smokingandjob"         "solder"
## [261] "splot"                 "st.y"
## [263] "stamps"                "stat.table"
## [265] "states"                "stats"
## [267] "stats.sales"           "stories"
## [269] "student.levels"        "students"
## [271] "students.fac"          "students.ord"
## [273] "studentsurvey"         "studyhabits"
## [275] "sugar"                 "summary"
## [277] "summary.binary"        "sv"
## [279] "t.ps"                  "taylor"
## [281] "tbl.students.fac"      "test.mean.sim"
## [283] "test_sim"              "testscores"
## [285] "theta"                 "titanic"
## [287] "tmp"                   "tms"
## [289] "trout"                 "tukey"
## [291] "tv"                    "twoway"
## [293] "txt"                   "u"
## [295] "upr"                   "us.population.2010"
## [297] "usa"                   "usmap"
## [299] "ustemperature"         "v"
## [301] "vowels"                "wilcoxon"
## [303] "wine"                  "wine.all"
## [305] "world.mortality.2017"  "worldpopulation"
## [307] "worldseries"           "wrds"
## [309] "wrinccensus"           "wrincsample"
## [311] "x0"                    "x1"
## [313] "xyz"                   "y"
## [315] "y.2.prime"             "y.prime"
## [317] "y0"                    "y1"
## [319] "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"   "wine"
```

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"             "wine"
##  [3] "houseprice"             "gasoline"
##  [5] "elusage"                "mothers"
##  [7] "draft"                  "newcomb"
##  [9] "package:parallel"       "mtcars"
## [11] "package:RCurl"          "package:bitops"
## [13] "package:combinat"       "package:leaps"
## [15] "package:bootstrap"      "package:maxLik"
## [17] "package:miscTools"      "package:modelr"
## [19] "package:gapminder"      "package:nycflights13"
## [21] "package:lubridate"      "package:tidytext"
## [23] "package:gutenbergr"     "package:babynames"
## [25] "package:bindrcpp"       "package:magrittr"
## [27] "package:pdftools"       "package:data.table"
```

```
## [29] "package:rio"             "package:gpuR"
## [31] "package:testlib"         "package:roxygen2"
## [33] "package:usethis"         "package:devtools"
## [35] "package:pryr"            "package:maps"
## [37] "package:Hmisc"           "package:Formula"
## [39] "package:survival"        "package:lattice"
## [41] "package:microbenchmark" "package:mvtnorm"
## [43] "package:random"          "wine"
## [45] "package:kableExtra"      "package:forcats"
## [47] "package:stringr"         "package:dplyr"
## [49] "package:purrr"           "package:readr"
## [51] "package:tidyr"           "package:tibble"
## [53] "package:tidyverse"       "package:knitr"
## [55] "package:bookdown"        "tools:rstudio"
## [57] "package:stats"           "package:graphics"
## [59] "package:grDevices"       "package:utils"
## [61] "package:datasets"        ".MyEnv"
## [63] "package:moodler"         "package:wolfr"
## [65] "package:shiny"           "package:Rcpp"
## [67] "package:grid"            "package:ggplot2"
## [69] "package:methods"         "Autoloads"
## [71] "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 elses. 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.

2.14.2 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://cran.r-project.org/web/packages/. Currently there are over 14500!

In fact, that is a 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 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 you 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)
```

Note sometimes after a major upgrade this fails, and you have to update each package one by one. The last time this happened was after the upgrade from Ver 3.4.0 to 3.5.0.

2.14.3 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 (>= 3.5.0)
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 (>= 3.5.0)
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")
```

Note if you now look into the folder C:/R/library there will be a folder testlib, which is this library.

Let's check:

```
library(testlib)
search()[1:4]
```

```
## [1] ".GlobalEnv" "wine"       "houseprice" "gasoline"
```

```
ls(2)
```

```
## [1] "Country"              "Heart.Disease.Deaths" "Wine.Consumption"
```

```
f1(2)
```

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

```
f2(2)
```

```
## [1] 1.414
```

And that's it!

Now there will be two folders with the name testlib:

- the one we just created
- another one in the default library folder. On Windows machines that is usually ../R/library and on Macs /Library/Frameworks/R.framework/Resources/library .

These two are NOT the same and only the second one is an actual R library. In essence the *install* command takes the first folder and turns it into a library that it puts in the place where R can find 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 folder with library files
# ' @export
# ' @examples
# ' make.library("moodlr", folder="c:/files")

make.library <- function (name, folder)
{
    library(devtools)
    library(roxygen2)
    olddir <- getwd()
    setwd(folder) # go where you need to be
    document()  # make lib
    setwd("..")
    install(name)
    setwd(olddir) # 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(name="wolfr",
    folder="c:/wolfgang/R/mylibs")
```

Note that ultimately a library is a folder. You can send someone a library by sending them the folder (usually as a compressed zip file)

2.15 Customizing R

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.

2.15.1 .First and .Rprofile

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(). You can however with

```
ls(all.names = TRUE)
```

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 what that is by running

```
dir("~")
```

On machines running Win10 it is usually the C:/Users/YourName/Documents folder.

Exactly in what folder the .Rprofile should be in is bit of a mystery, it depends not just on your operating system but even on its version. You might need to do a bit of trial and error!

So here is what (part of) mine looks like:

options(show.signif.stars=FALSE) # for p-values
options(stringsAsFactors=FALSE) # a classic source of errors
library(ggplot2)
library(wolfr) # Two of my own
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 every time I start RStudio, but different working directories might have different First.R's (or none)

2.15.2 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 Dropbox folder on one computer and then go to another, as soon as the new file is uploaded I got it there as well.
- availability: I can get to my files anyplace, anytime.
- deal with submissions of students, avoids email

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 Rproj folder.

Now of course Dropbox won't save anything automatically, so don't forget to do it regularly yourself!

3 Extending R, Packages

3.1 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.

3.1.1 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

3.1.2 Grammar of Graphics

In 2005 Wilkinson, Anand, and Grossman published the book "The Grammar of Graphics". In it they laid 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

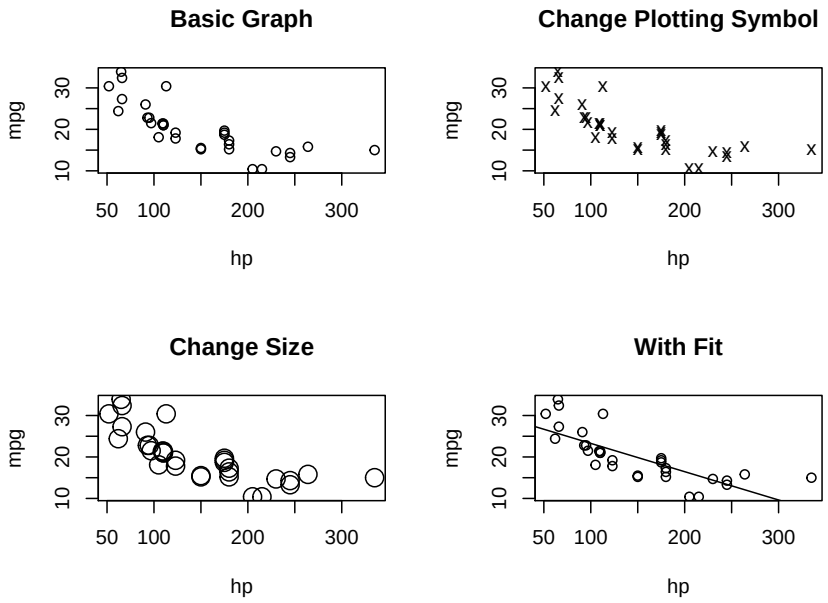
so it is not only about language.

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))
```

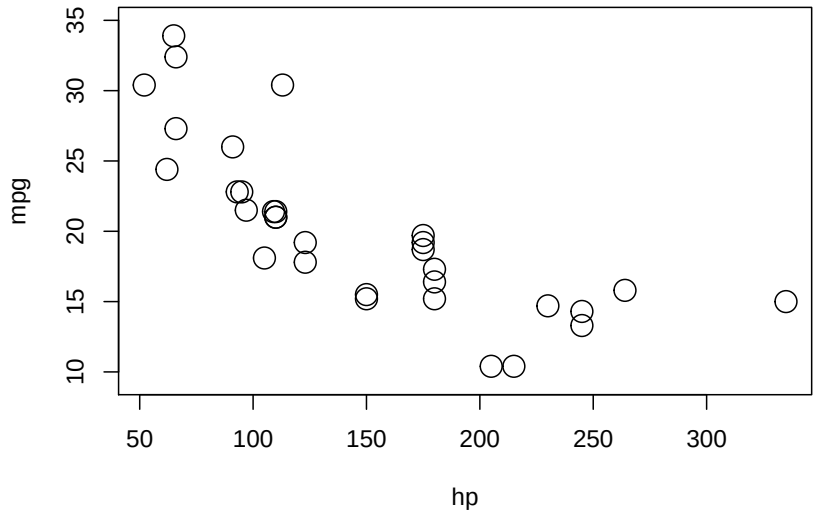


The basic idea of the grammar of graphs is to separate out the parts of the graphs: there is the basic layout, there is the data that goes into it, there is the way in which the data is displayed. Finally there are annotations, here the titles, and possibly more things added, such as a fitted line. In 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 lasagna: 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:

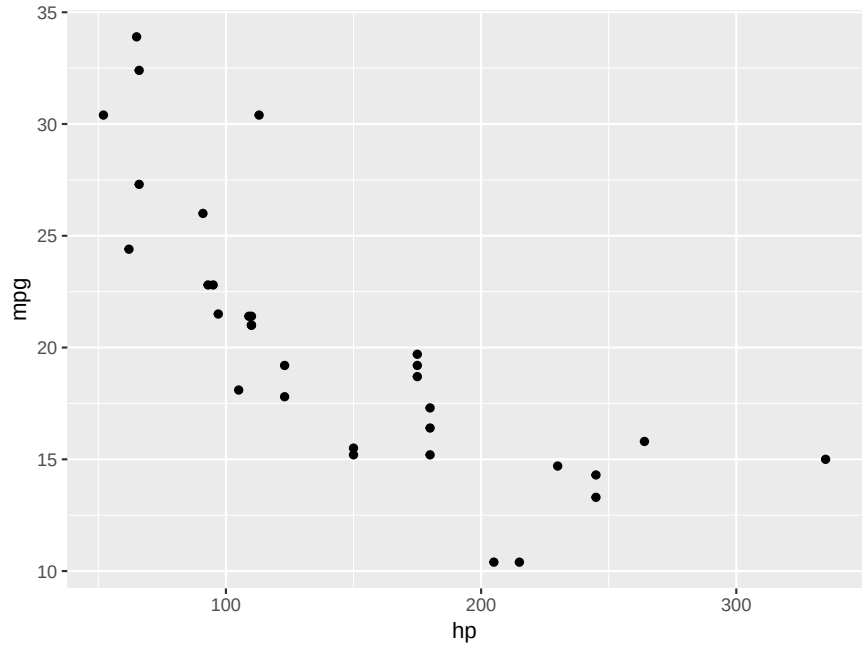
```
plot(hp, mpg, cex=2, ylim=range(mpg)+c(-1, 1))
```



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()
```



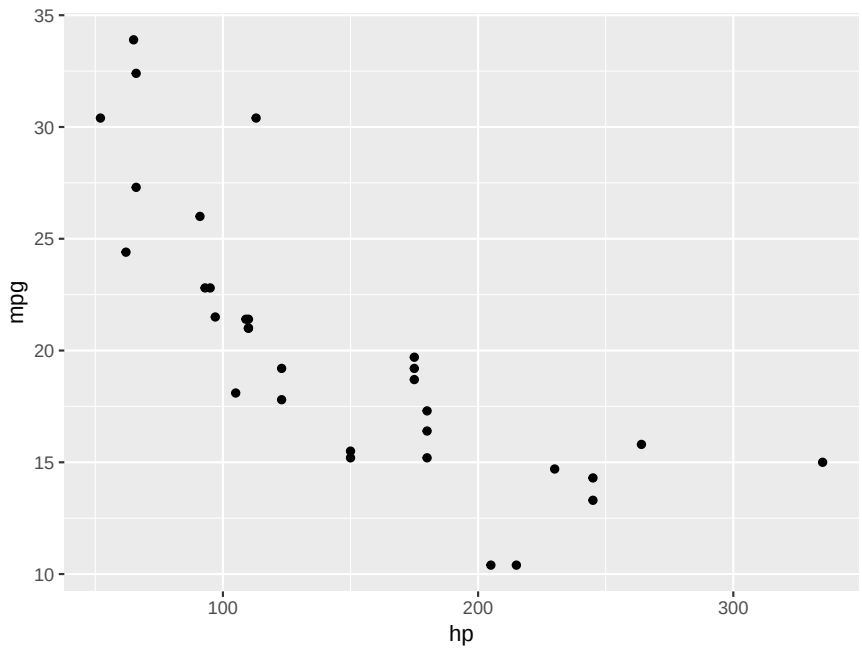
this has the following logic:

- *ggplot* sets up the graph

- it's first argument is the data set (which has to be a dataframe)
- *aes* is the *aestetic 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*

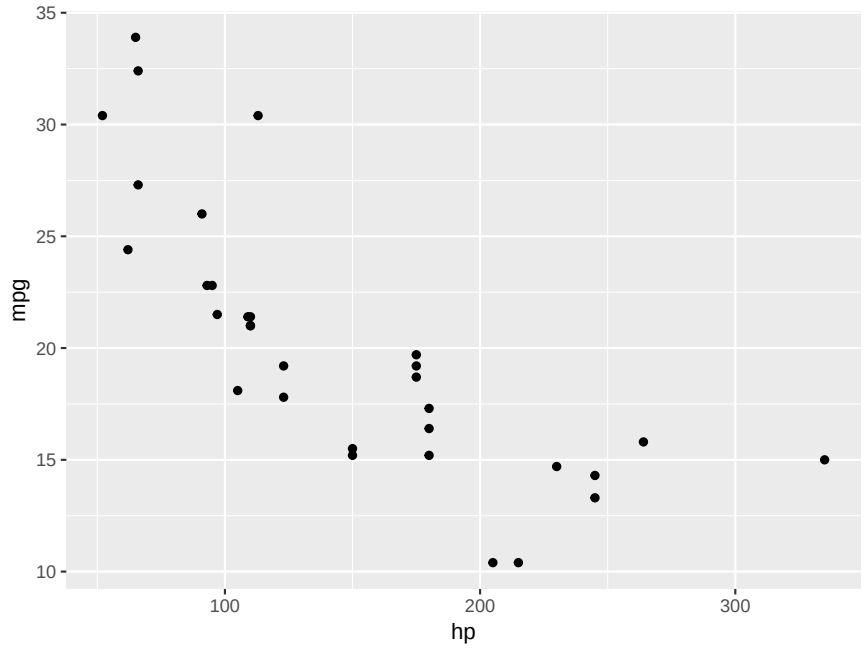
```
qplot(hp, mpg, data=mtcars)
```



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))
```

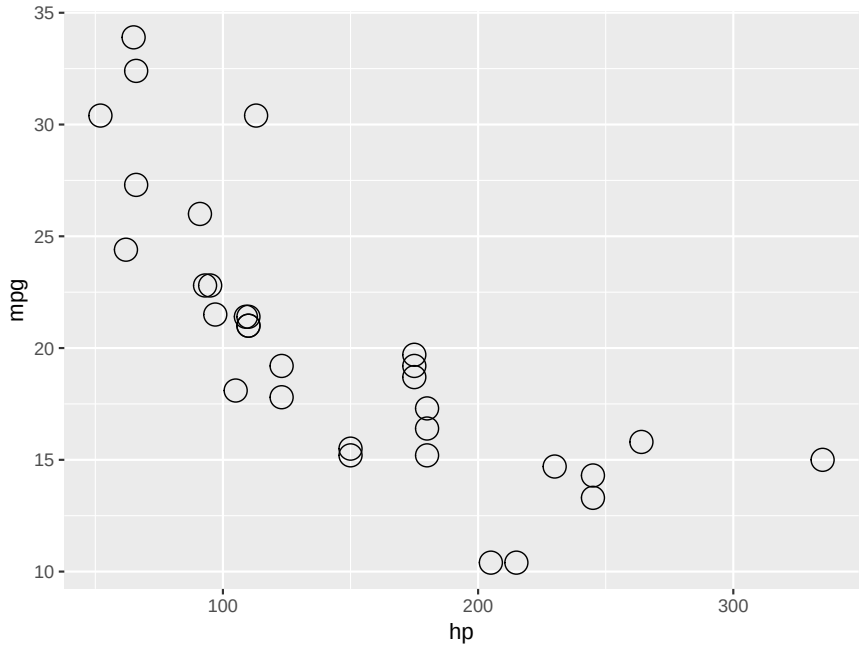


again it seems to do the same thing, but there is a big difference:

- 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.

How about the problem with the graph above, where we had to increase the y margin?

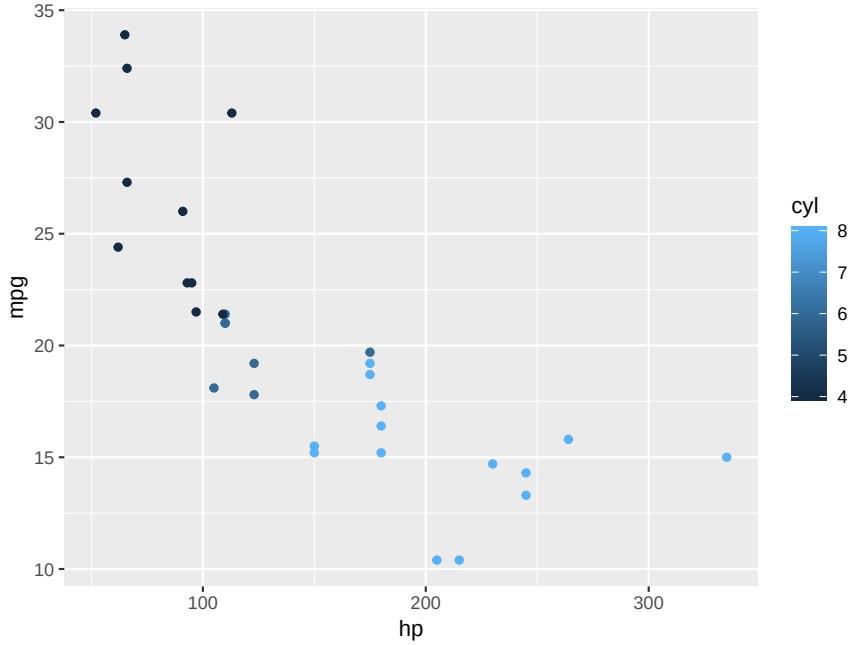
```
ggplot(mtcars, aes(hp, mpg)) +
  geom_point(shape=1, size=5)
```



so we see that here this is done automatically.

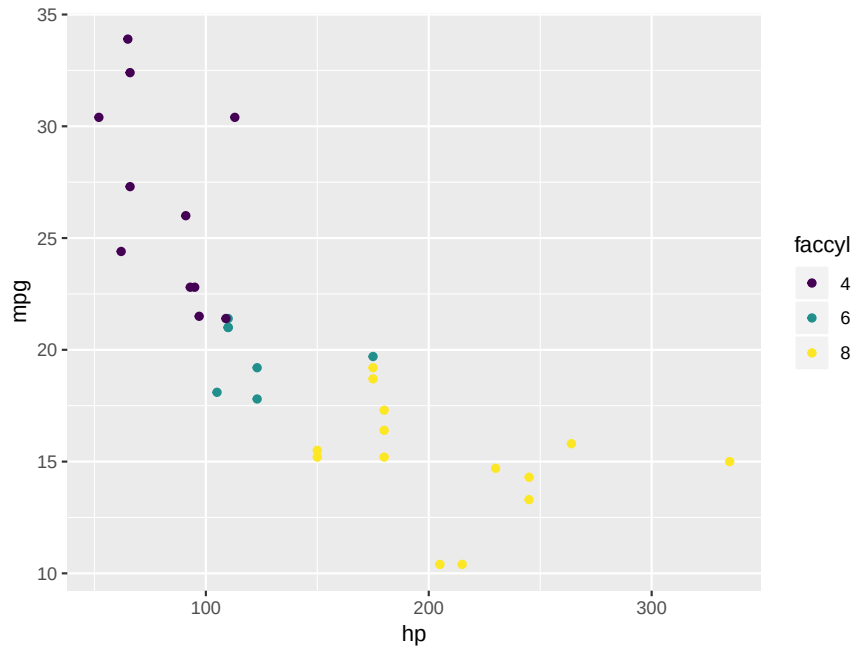
Let's say we want to identify the cars by the number of cylinders:

```
ggplot(mtcars, aes(hp, mpg, color=cyl)) +
  geom_point()
```



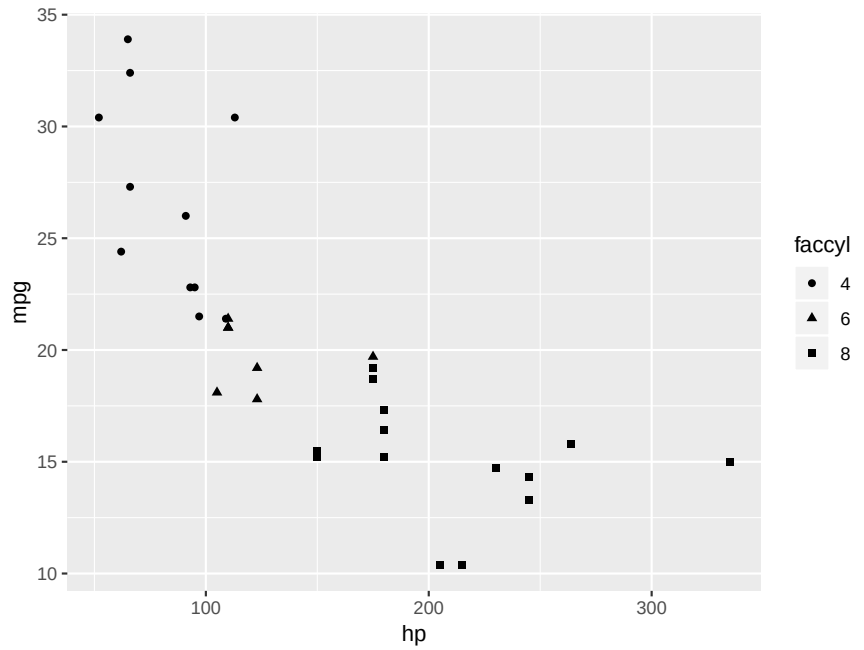
Notice that the legend is a continuous color scale. This is because the variable cyl has values 4, 6, and 8, and so is identified by R as a numeric variable. In reality it is categorical (ever seen a car with 1.7 cylinders?), and so we should change that:

```
mtcars$faccyl <- factor(cyl,
                        levels = c(4, 6, 8),
                        ordered = TRUE)
ggplot(mtcars, aes(hp, mpg, color=faccyl)) +
  geom_point()
```



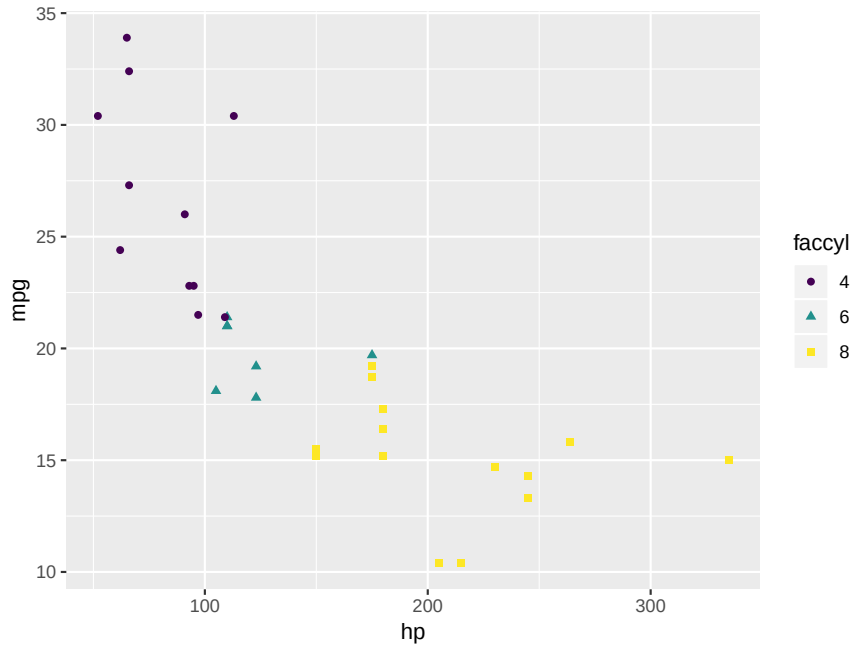
we can also change the shape of the plotting symbols:

```
ggplot(mtcars, aes(hp, mpg, shape=faccyl)) +
  geom_point()
```



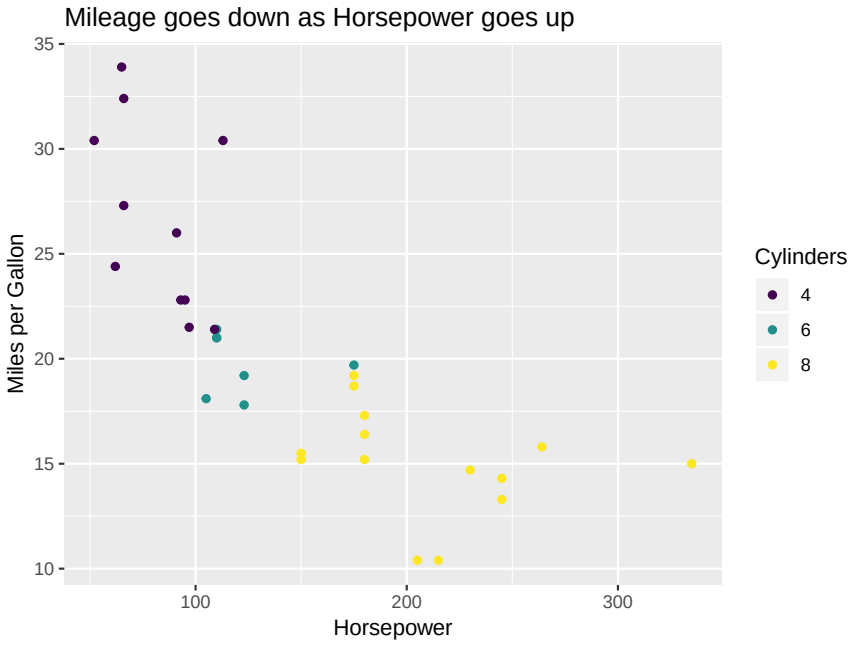
or both:

```
ggplot(mtcars, aes(hp, mpg, shape=faccyl, color=faccyl)) +
  geom_point()
```



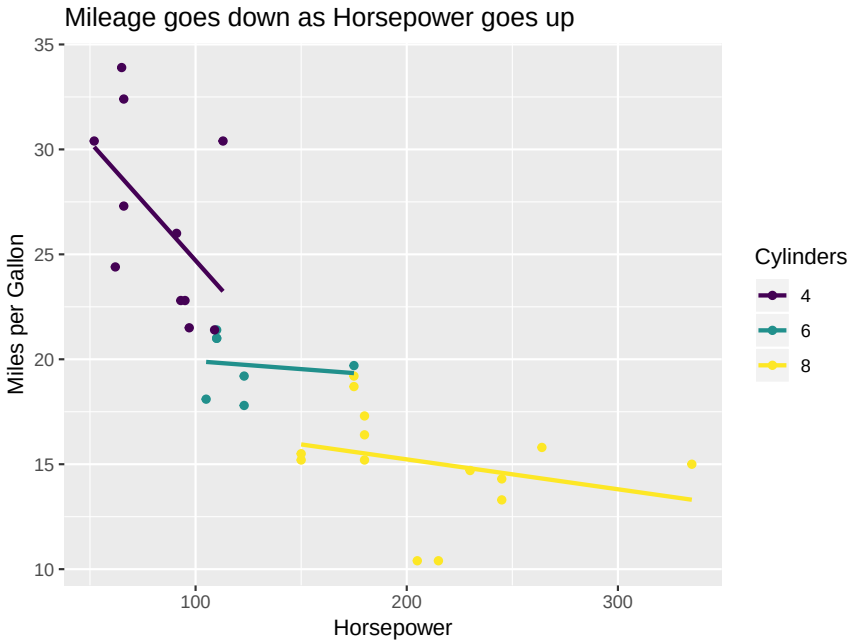
let's pretty up the graph a bit with some labels and a title. We will be playing around with this graph for a while, so I will save some intermediate versions:

```
plt1 <- ggplot(mtcars, aes(hp, mpg, color=faccyl)) +
  geom_point()
plt2 <- plt1 +
  labs(x = "Horsepower",
       y = "Miles per Gallon",
       color = "Cylinders",
       title = "Mileage goes down as Horsepower goes up")
plt2
```



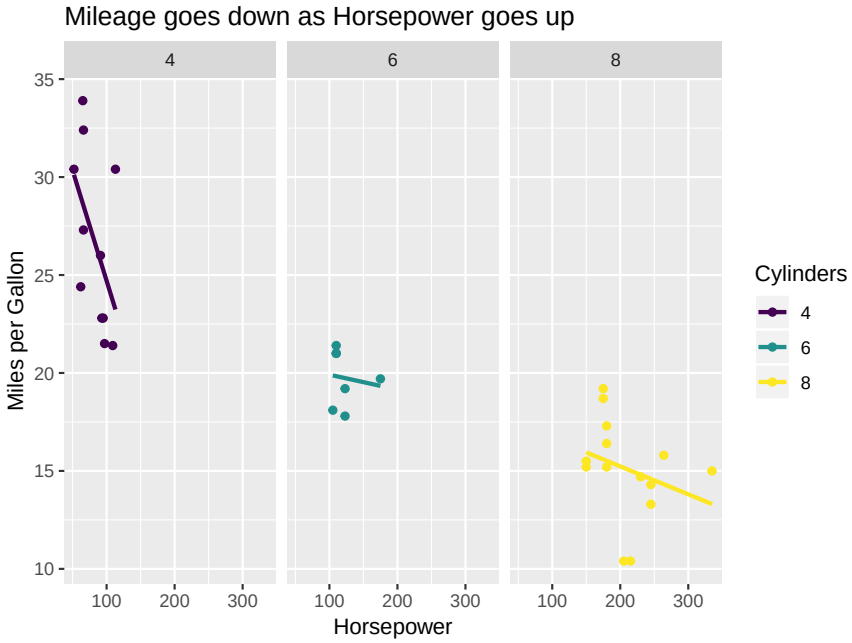
Say we want to add the least squares regression lines for cars with the same number of cylinders:

```
plt3 <- plt2 +
  geom_smooth(method = "lm", se = FALSE)
plt3
```



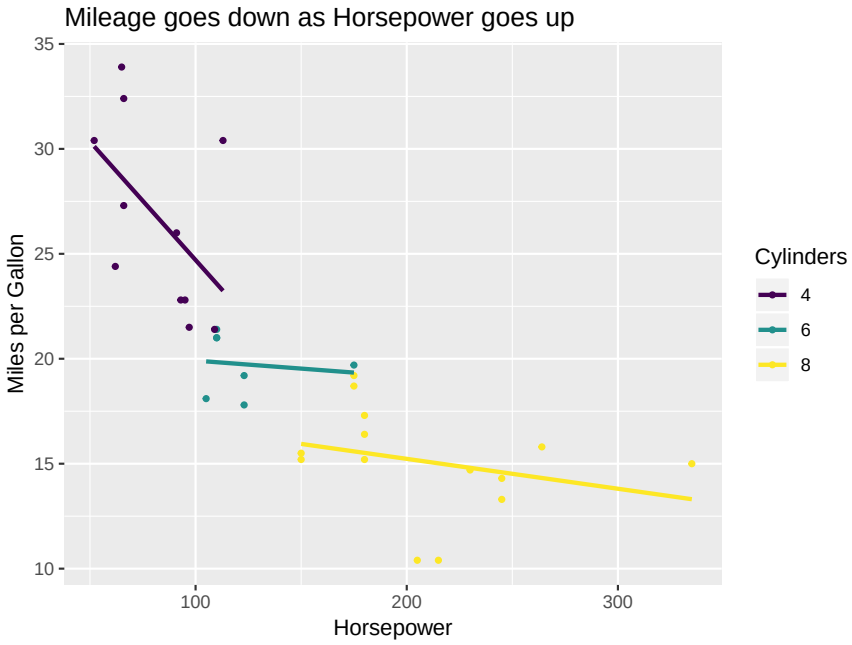
There is another way to include a categorical variable in a scatterplot. The idea is to do several graphs, one for each value of the categorical variable. These are called *facets*:

```
plt3 +
  facet_wrap(~cyl)
```



The use of facets also allows us to include two categorical variables:

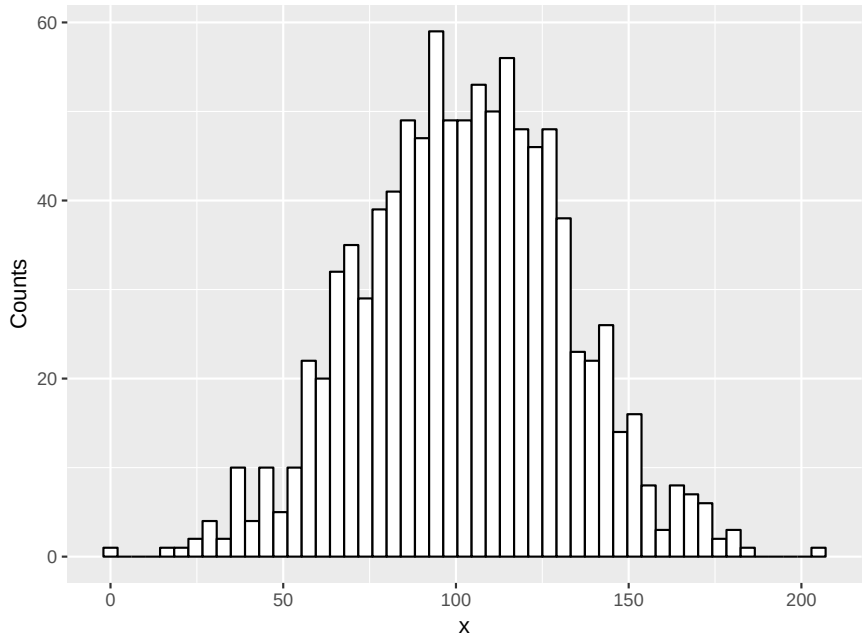
```
mtcars$facgear <-
  factor(gear, levels = 3:5, ordered = TRUE)
ggplot(mtcars, aes(hp, mpg, color=faccyl)) +
           geom_point(size = 1) +
  labs(x = "Horsepower",
       y = "Miles per Gallon",
       color = "Cylinders",
       title = "Mileage goes down as Horsepower goes up") +
  geom_smooth(method = "lm", se = FALSE)
```



This is almost a bit to much, with just 32 data points there is not really enough for such a split.

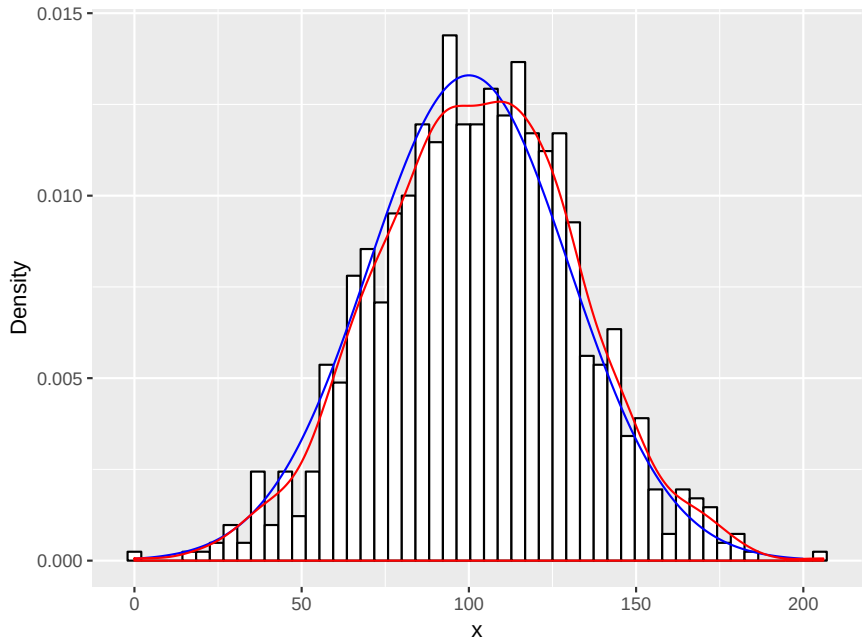
3.1.3 Histograms

```
x <- rnorm(1000, 100, 30)
df3 <- data.frame(x = x)
bw <- diff(range(x))/50 # use about 50 bins
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:

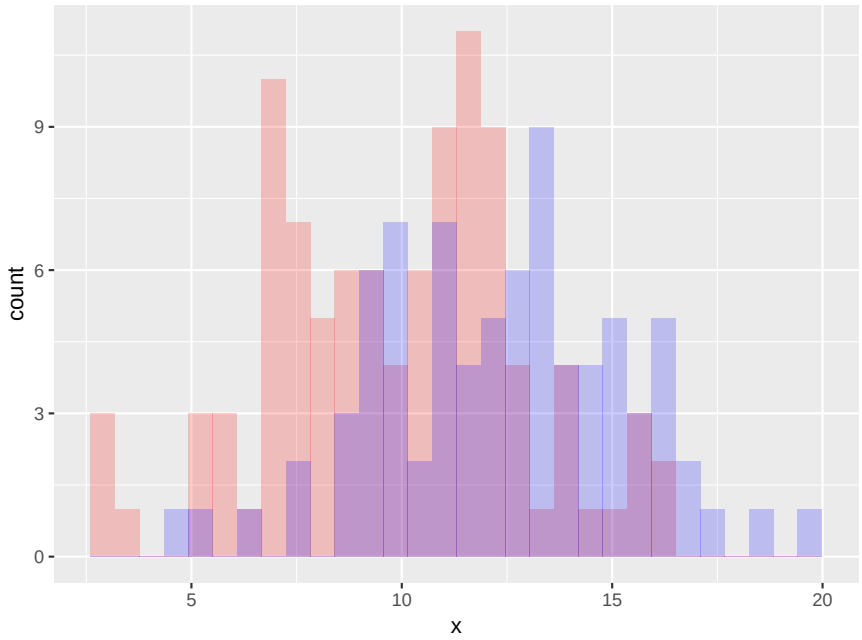
```
x <- seq(0, 200, length=250)
df4 <- data.frame(x=x, y=dnorm(x, 100, 30))
ggplot(df3, aes(x)) +
  geom_histogram(aes(y = ..density..),
        color = "black",
        fill = "white",
        binwidth = bw) +
  labs(x = "x", y = "Density") +
  geom_line(data = df4, aes(x, y),
            colour = "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 overlaid on the other:

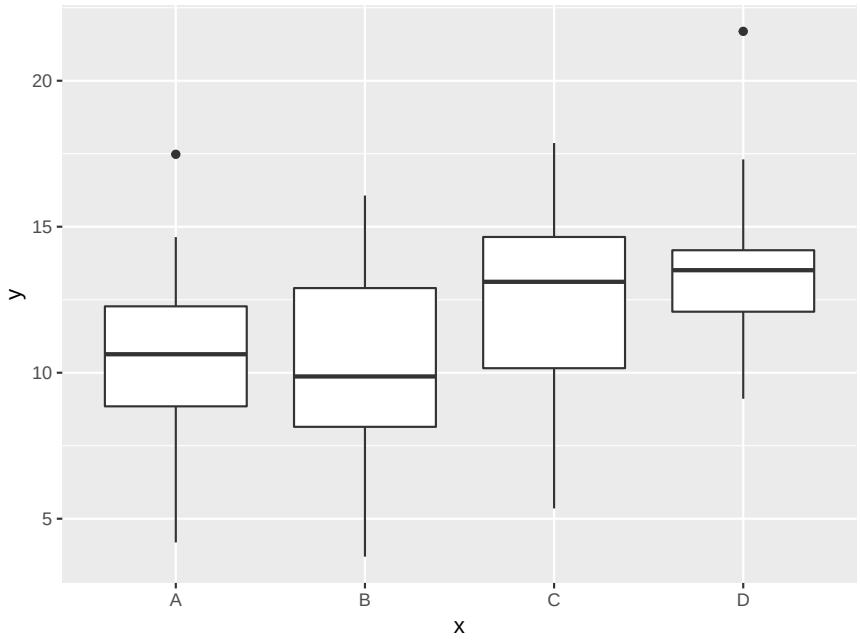
```
df5 <- data.frame(
  x = c(rnorm(100, 10, 3), rnorm(80, 12, 3)),
  y = c(rep(1, 100), rep(2, 80)))
ggplot(df5, aes(x=x)) +
    geom_histogram(data = subset(df5, y == 1),
        fill = "red", alpha = 0.2) +
    geom_histogram(data = subset(df5, y == 2),
        fill = "blue", alpha = 0.2)
```



Notice the use of alpha. In general this "lightens" the color so we can see "behind".

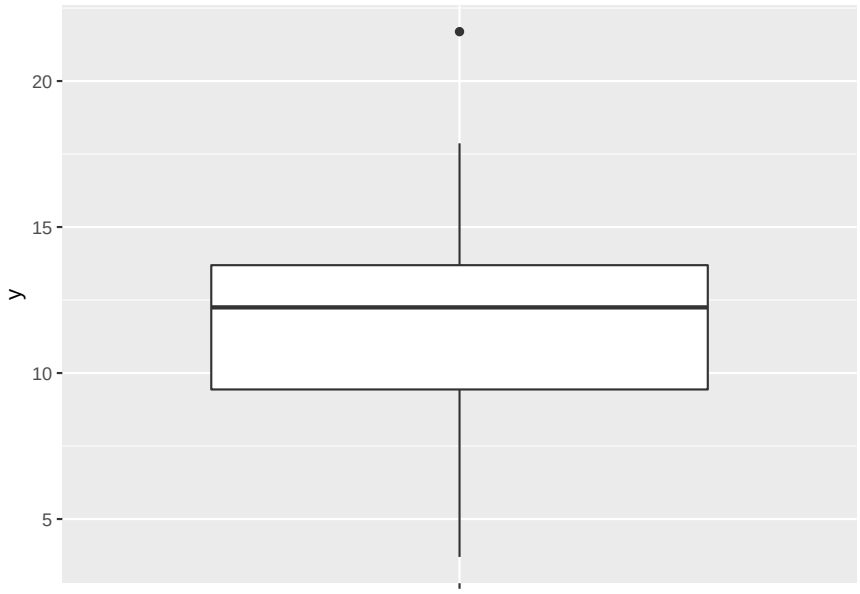
3.1.4 Boxplots

```
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:

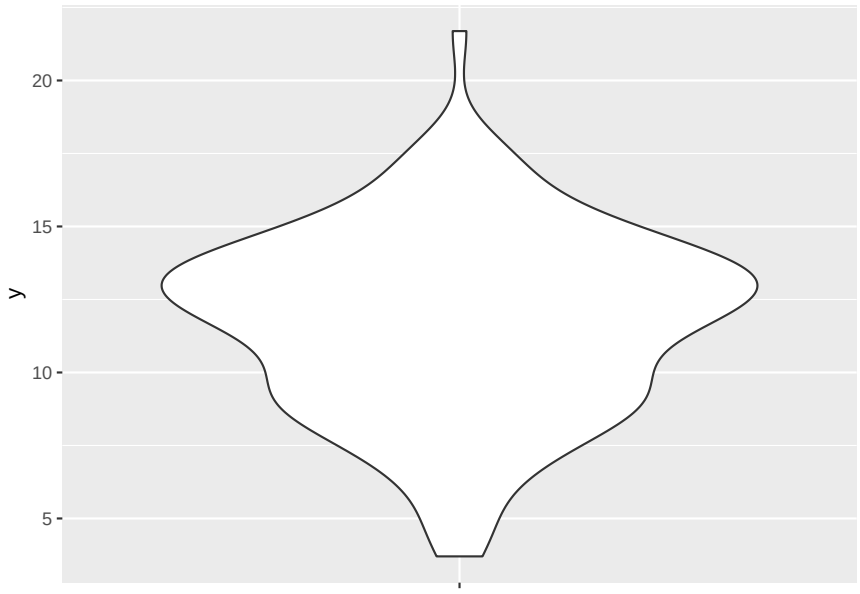
```
ggplot(df6, aes(x="", y)) +
  geom_boxplot() +
  xlab("")
```



There is a modern version of this graph called a violin plot:

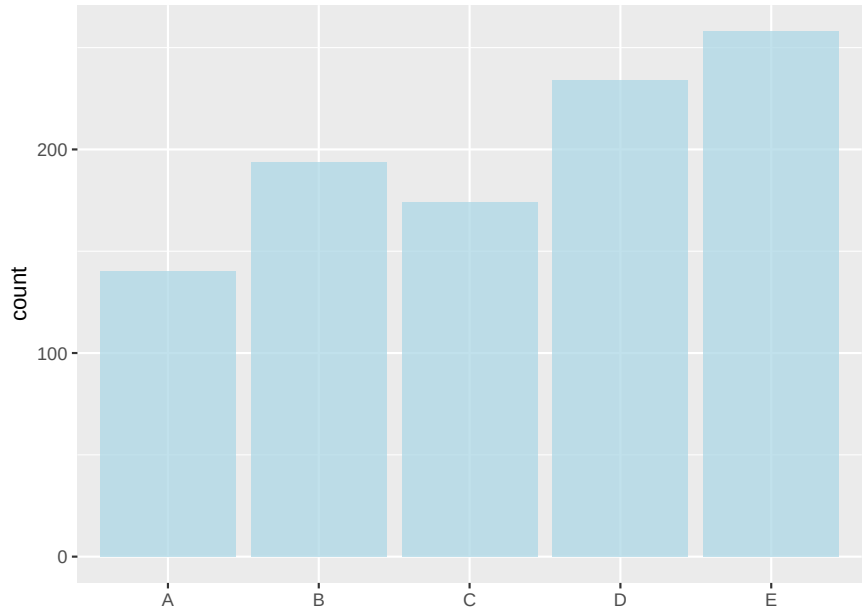
```
ggplot(df6, aes(x="", y)) +
  geom_violin() +
```

```
  xlab("")
```



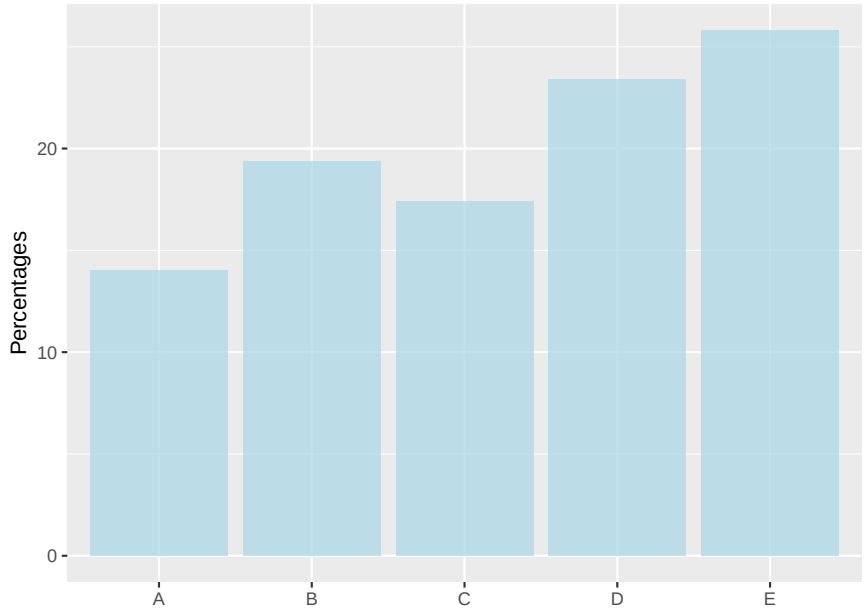
3.1.5 Barcharts

```
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:

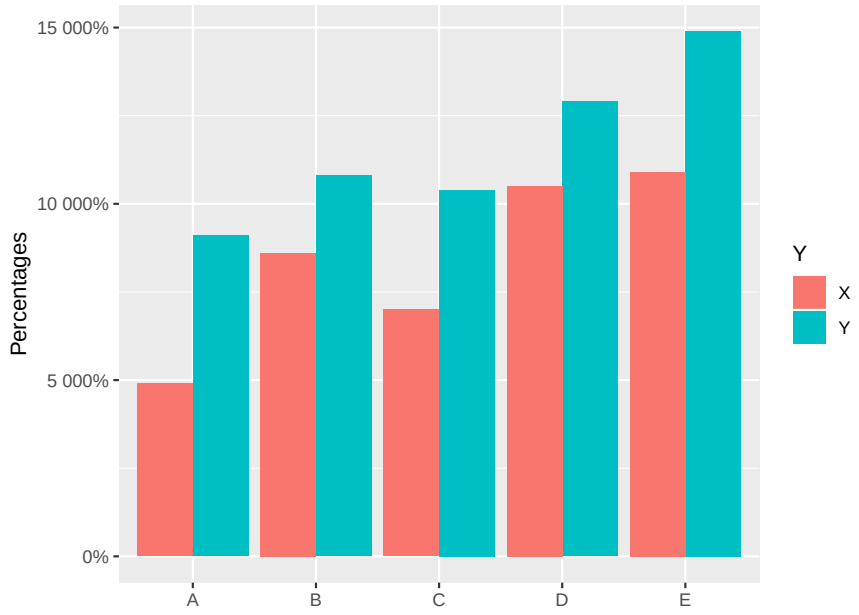
```
ggplot(df7, aes(x=x)) +
  geom_bar(aes(y=100*(..count..)/sum(..count..)),
      alpha = 0.75,
      fill = "lightblue") +
  labs(x="", y="Percentages")
```



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.

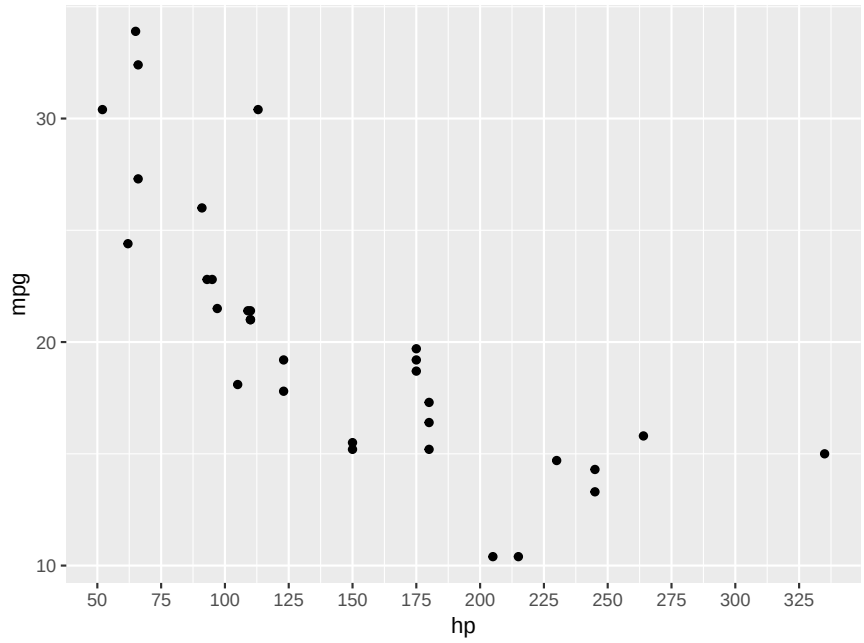
```
df7$y <- sample(c("X", "Y"),
                size = 1000,
                replace = TRUE,
                prob = 2:3)
ggplot(df7, aes(x=x, fill = y)) +
  geom_bar(position = "dodge") +
    scale_y_continuous(labels=scales::percent) +
    labs(x="", y="Percentages", fill="Y")
```



3.1.6 Axis Ticks and Legend Keys

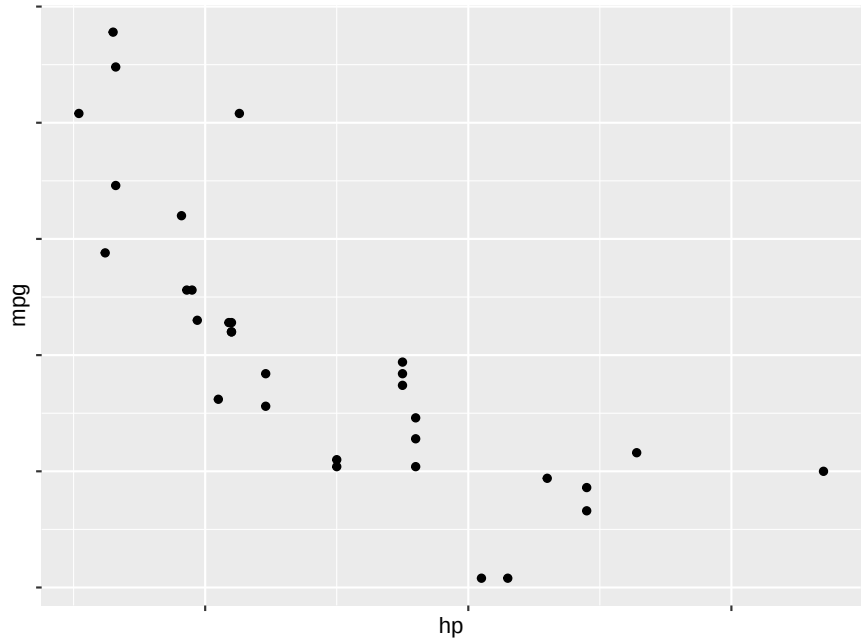
Let's return to the basic plot of mpg by hp. Let's say we want to change the axis tick marks:

```
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))
```



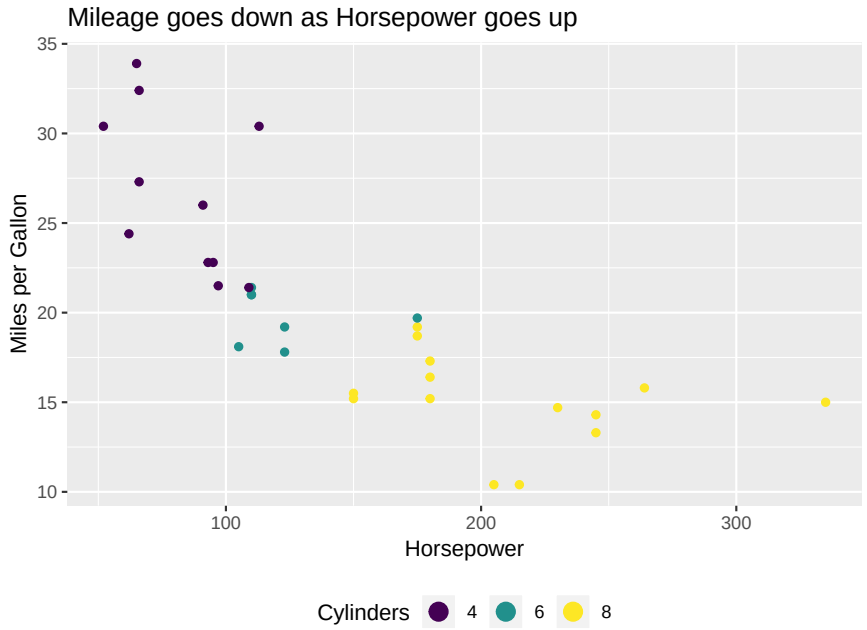
sometimes we want to do graphs without any tick labels. This is useful for example for maps and also for confidential data, so the viewer sees the relationship but can't tell the sizes:

```
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 the appearance of the legend. Recall that the basic graph is in *plt2*. Then

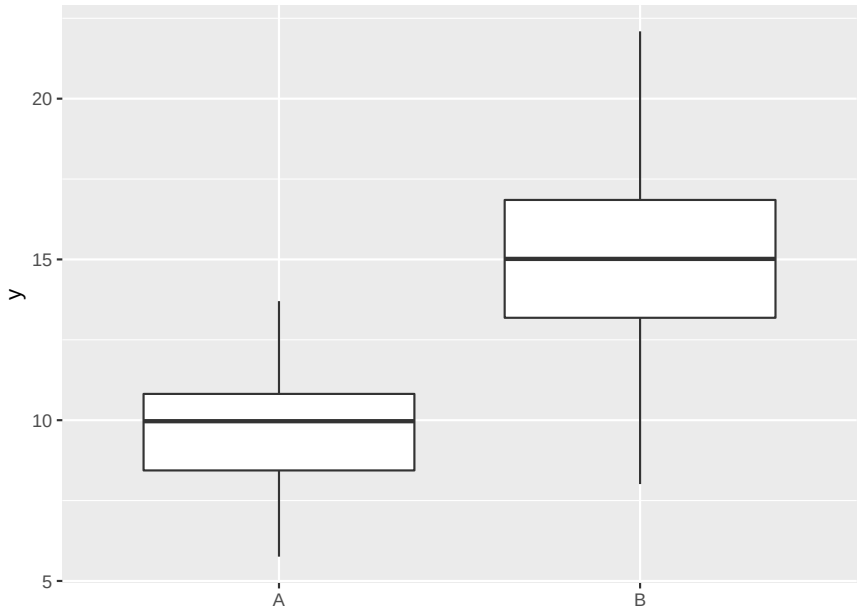
```
plt2 +
  theme(legend.position = "bottom") +
  guides(color=guide_legend(nrow = 1,
                            override.aes = list(size=4)))
```



3.1.7 Special Symbols in Labels etc.

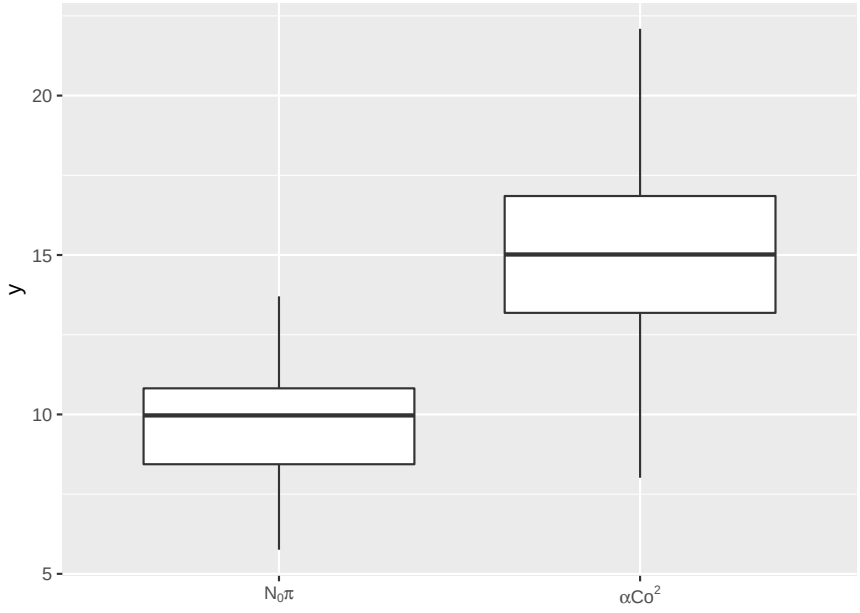
Consider this boxplot:

```
plt <- ggplot(df, aes(x, y)) +
 geom_boxplot() +
  labs(x="")
plt
```



Now it turns out that this are measurements of some chemical concentrations, and the name of A is really $N_0\pi$ and of B it is αCo^2. It would be nice to use these in the graph:

```
plt +
 scale_x_discrete(labels =
         c(expression(paste("N"[0], pi)),
           expression(paste(alpha, "Co"^2))))
```



Unfortunately the exact syntax differs when changing labels, annotations, titles etc, and it can sometimes take a bit of work to find the right one.

3.1.8 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 web page 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.with=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!

3.2 C++ with Rcpp

3.2.1 Basics

For this section you will need the Rcpp and the microbenchmark packages.

```
library(Rcpp)
library(microbenchmark)
```

C++ is probably the most widely used general programming language today. Actually, about half the code of base R is written in C++! (the rest is about half and half R and Fortran)

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)
}
tmpR <- round(as.numeric(summary(microbenchmark(dist2(x, y)))["mean"]), 2)
tmpR
```

```
## [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
```

Note: in the R markdown document the above chunk starts like this:

"'{r engine='Rcpp'}

this tells R markdown to treat this chunk as Rcpp code and compile it accordingly.

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;
}
```

```
tmpRcpp <- round(as.numeric(summary(microbenchmark(dist4(x, y)))["mean"]), 2)
tmpRcpp
```

```
## [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 NA and NA 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 NA $*10^{-3} \times 10^6/3600 =$ NA hours, but the Cpp routine takes only NA 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.
  the most common data types are *int*, *double*, *char* and *bool*.
- in C++ (almost) every line ends with a ;
- vectors start with index 0, not 1
- the for loop is for(int i=0; i<n; ++i)
- the repeat loop is called a *do* loop and has the format

```
do {
# do stuff
} while( condition );
```

- whereas R does a lot of type conversion, C++ does none. This can lead to occasional strange behavior:

```
# include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double mydiv(int n, int m) {
  return n/m;
}
```

```
mydiv(5, 2)
```

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

so the result is 2, not 2.5. The reason is that when we divide two integers, in C++ the result is always an integer.

The get the correct result we need to do the type conversion ourselves:

```
# include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double mydiv(int n, int m) {
```

```
  return n/double(m);
}
```

```
mydiv(5, 2)
```

```
## [1] 2.5
```

To be linked to R the C++ routine has to start with with these lines:

```
# include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]

}
```

The easiest way to get started is to just use RStudio - File - New File - C++ File.

If the C++ routine takes as its argument a single value it can be defined in the usual C++ way, as in the *mydiv* routine. If we want to use a vector as an argument or the return object we need to use a special variable type called NumericVector. It is just what it says it is.

3.2.2 Debugging

Generally you should only turn fairly short code into C++, so debugging is not to 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 0.953316
## 1 0.982095
## 2 0.531598
## 3 0.841642
## 4 0.618781
```

```
## [1] 0.9533 0.9821 0.5316 0.8416 0.6188
```

3.2.3 Sugar

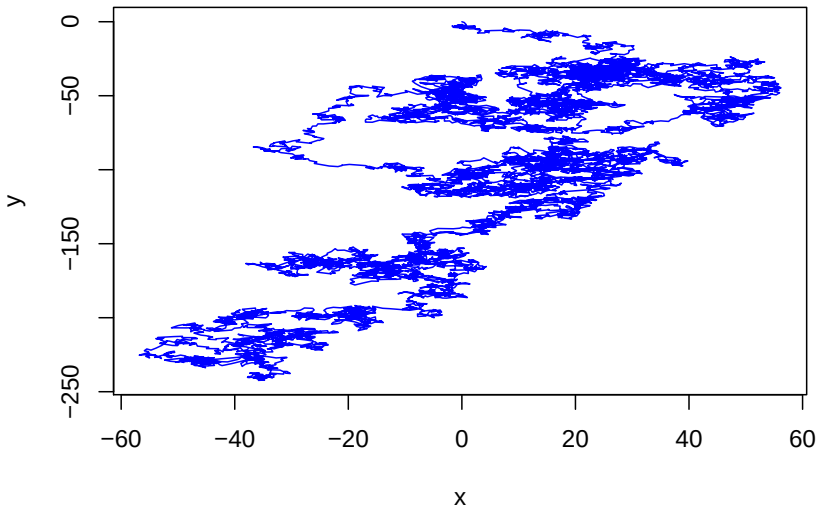
Not only is Rcpp vectorized, many standard R functions have been ported to Rcpp. For example, say we want to write a routine that simulates Brownian motion in R^2. That is, a stochastic process that moves as follows: if at time t it is at (x_0, y_0), then at time $t + \delta$ it is at $(x_0 + \delta X, y_0 + \delta Y)$ 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!

3.2.3.1 *Example* Fibonacci Numbers and the Golden Ratio

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

$$\lim_{k\to\infty} \frac{n_k}{n_{k-1}}$$

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] 1.618
```

The actual value of the golden ratio is of course $\frac{1+\sqrt{5}}{2} = 1.618...$

3.2.4 Using existing 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!

3.3 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.

3.3.1 Using multiple processors

There are a number of packages that help here. I will discuss

```
library(parallel)
```

If you don't know how many processors (called cores) you computer has your 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
```

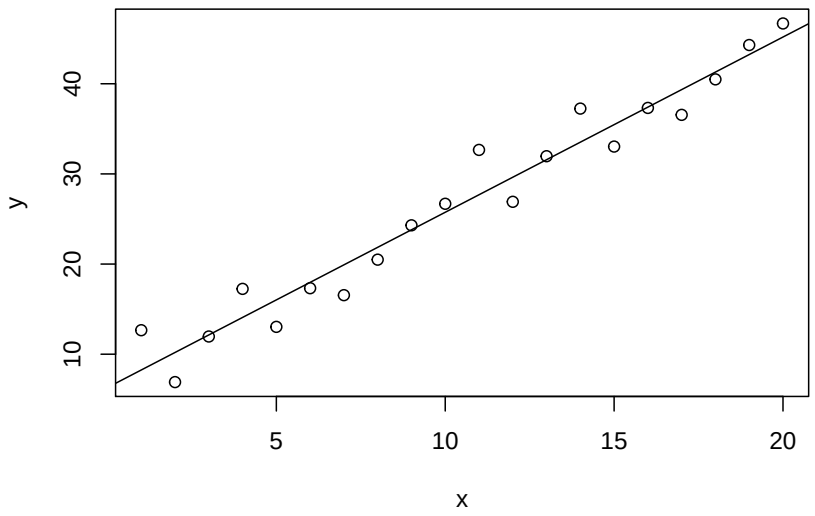
of them.

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 $y = \beta_0 + \beta_1 x + \epsilon$, where $\epsilon \sim N(0, \sigma)$. The parameters β_0, β_1 and σ are estimated by minimizing the least squares criterion

$$L(\beta_0, \beta_1) = \sum_{i=1}^{n} (y - \beta_0 - \beta_1 x)^2$$

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
##       6.303       1.944
```

Now a simulation will be fix some numbers n, β_0, β_1 and σ, 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
##      30
```

so this takes almost 30 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
```

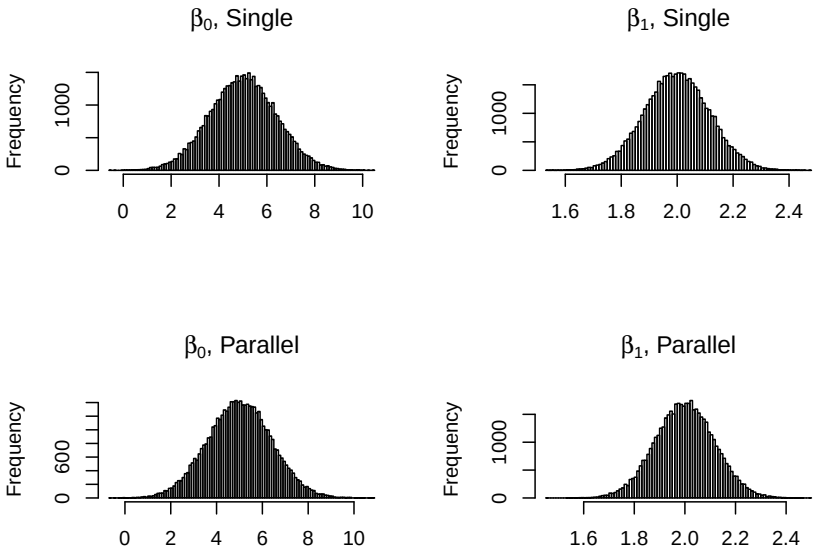
```
## elapsed
##      11
```

and so this took only about 11 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
##    0.32    0.08    0.48
```

```
tm <- proc.time()
a <- parRapply(cl, A, max)
proc.time()-tm
```

```
##    user  system elapsed
##    0.03    0.02    0.06
```

In general the easiest case of parallizing 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:

```
cat("Exam 1: ", grades$Exam_1, "\n")
```

```
## Exam 1:  64 80 74 77 65 62 75 68 62 64
```

```
cat("Exam 2: ", grades$Exam_2, "\n")
```

```
## Exam 2:  73 78 70 65 67 69 69 58 58 69 71 77 62 79 73
```

```
#...
cat("Exam 50: ", grades$Exam_50, "\n")
```

```
## Exam 50:  73 69 67 71 73
```

Now we want to find the minimum , mean, standard deviation and maximum of each exam. We can do that with

```
grade.summary <- function(x) round(c(min(x),
                                     mean(x),
                                     sd(x),
                                     max(x)), 1)
z <- lapply(grades, grade.summary)
z[1:2]
```

```
## $Exam_1
## [1] 62.0 69.1  6.8 80.0
##
## $Exam_2
## [1] 58.0 69.2  6.5 79.0
```

and to run this in parallel:

```
z <- parLapply(cl, grades, grade.summary)
z[1:2]
```

```
## $Exam_1
## [1] 62.0 69.1  6.8 80.0
##
## $Exam_2
## [1] 58.0 69.2  6.5 79.0
```

When you are done with the parallel calculations

```
stopCluster(cl)
```

3.3.2 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 were 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 pug'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 pug programming you need to get the *spur* 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 spur 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 spur we have to turn the matrix into a spur 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 matrix and convector objects. These methods include basic arithmetic functions %*%, +, -, *, /, t, , cross prod, crosspatch, col Means, col Sums, row Mean, and row Sums.

Math functions include sin, sin, sing, cos, cos, cosh, tan, tan, tang, log, log10, exp, abs, max, admin. Additional operations include some linear algebra routines such as cob(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.

3.4 Input/Output revisited

Libraries used in this section:

```
library(rio)
library(readr)
```

```
library(data.table)
library(pdftools)
```

There are a number of packages that help with data I/O. Some are specialized for certain data types, others are more general.

3.4.1 *rio* library

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

```
import("c:/tmpdata/mytestdata.csv")
```

3.4.1.1 Example

```
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 -1.543 -0.497 c
## 2  0.466 -0.225 b
## 3 -0.246  1.123 b
```

```
dir.create("c:/tmpdata")
export(xyz, "c:/tmpdata/mytestdata.csv")
rm(xyz)
```

```
head(import("c:/tmpdata/mytestdata.csv"), 3)
```

```
##        x      y z
## 1 -1.543 -0.497 c
## 2  0.466 -0.225 b
## 3 -0.246  1.123 b
```

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.

3.4.1.2 *readr* library

This package is specific to rectangular data, such as data from an Excel spreadsheet. It's main advantage is it's 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 -1.543 -0.497 c
## 2  0.466 -0.225 b
```

```
(proc.time()-tm)[3]
```

```
## elapsed
##    1.89
```

```
tm <- proc.time()
head(import("c:/tmpdata/mytestdata.csv"), 2)
```

```
##        x      y z
## 1 -1.543 -0.497 c
## 2  0.466 -0.225 b
```

```
(proc.time()-tm)[3]
```

```
## elapsed
##    0.06
```

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.

3.4.2 *data.table* library

Similar to *read.table* but faster and more convenient. The command is called *fread*:

```
tm <- proc.time()
head(fread("c:/tmpdata/mytestdata.csv"), 2)
```

```
##         x      y z
## 1: -1.543 -0.497 c
## 2:  0.466 -0.225 b
```

```
(proc.time()-tm)[3]
```

```
## elapsed
##    0.06
```

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*.

3.4.3 *pdftools* library

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. The problem is that pdf files are not plain text and so can not be read directly by R. Here we can use the package *pdftools*.

3.4.3.1 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-Bookle
```

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"
kable.nice(dta, do.row.names = FALSE)
```

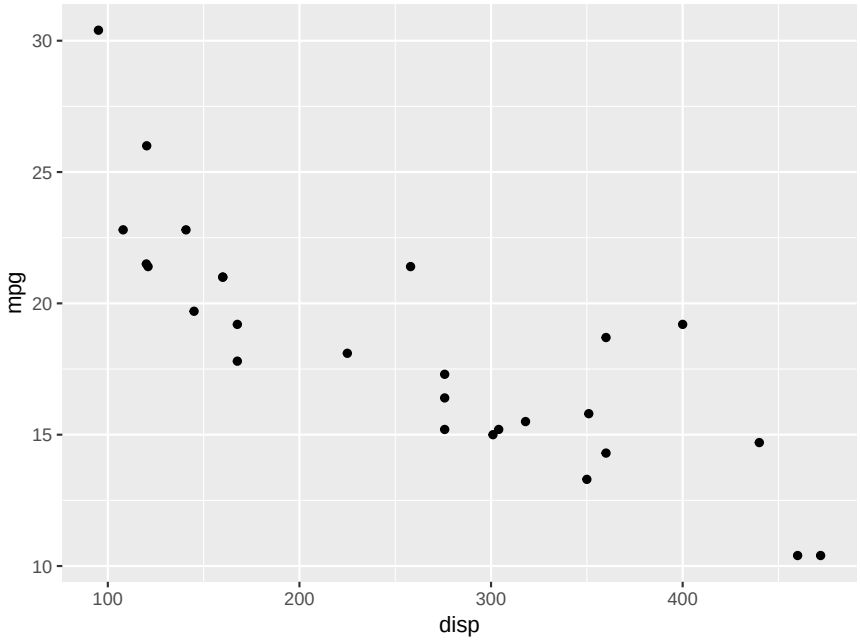
Country	Life Expectancy
China, Hong Kong SAR	153
China, Macao SAR	152
Japan	151
Switzerland	150
Spain	149
Singapore	148
Italy	148
Australia	147
Iceland	146
Australia/New Zealand	146
France	145
Israel	144
Sweden	144
Canada	143
Norway	142
Republic of Korea	141
Martinique	140
Netherlands	139
New Zealand	139
Luxembourg	138
United Kingdom	136
Austria	136
Ireland	135
Finland	134
Guadeloupe	134
Channel Islands	133
Greece	133
Portugal	133
Belgium	133
Slovenia	132
Germany	132
Malta	131
Denmark	130
Cyprus	129
Réunion	128
Mayotte	127
China, Taiwan Province of China	126
Puerto Rico	126
French Guiana	126
Cuba	125
United States Virgin Islands	125

3.5 The pipe, dplyr, tibbles, tidyverse

3.5.1 The pipe

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.

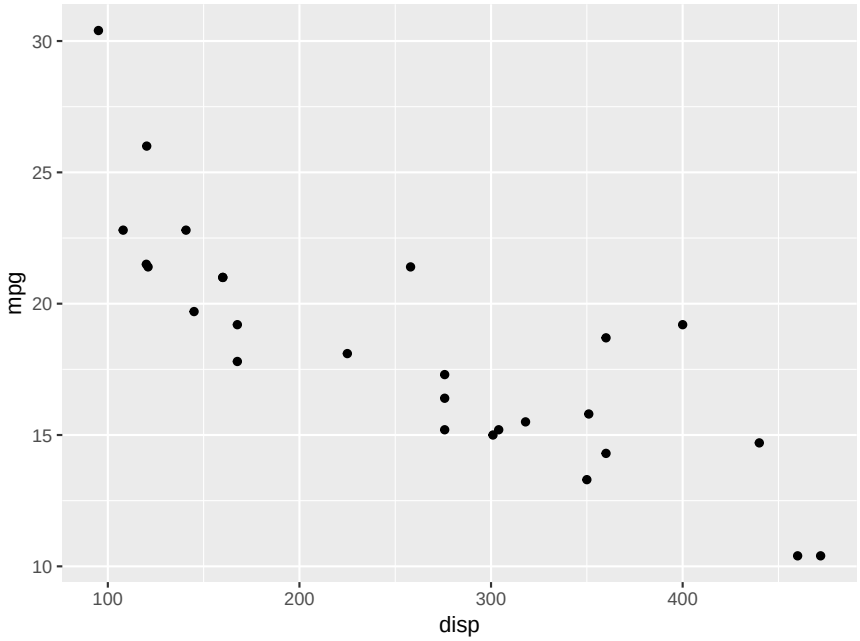
The basic package to use piping is

```
library(magrittr)
```

invented by Stefan Milton. The basic operator is %>%. The same as above can be done with

```
mtcars %>%
  subset(x=., hp>70) %>% #R knows what hp is!
```

```
  ggplot(data=., aes(disp, mpg)) +
     geom_point()
```



In principle the pipe can always be used in this way:

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

```
## [1] 0.2012
```

```
x %>%
  mean()
```

```
## [1] 0.2012
```

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.

Exercise

Consider the following operation:

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
round(exp(diff(log(x))), 1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

write the same using the pipe

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

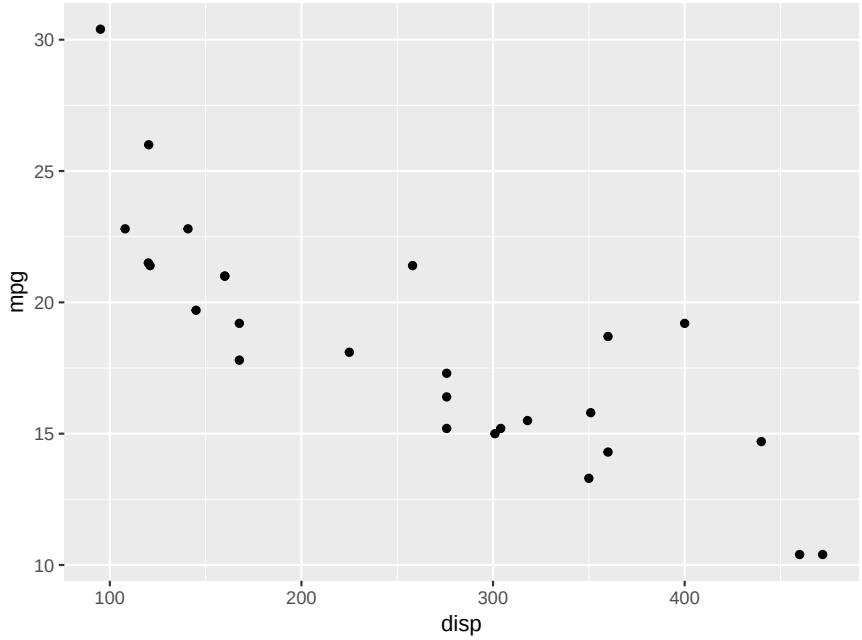
one to think from left to right in the logical order of functions rather than from inside to outside in an ugly large nested statement of functions.

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.

3.5.2 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 help with 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 faccyl
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <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
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1 6
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2 8
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1 6
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4 8
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2 4
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2 4
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4 6
## # ... with 22 more rows, and 1 more variable: facgear <ord>
```

so we have all relevant information about the data set: its size (32x11), the variables and their formats, and the beginning 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.

3.5.3 *dplyr* library

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)

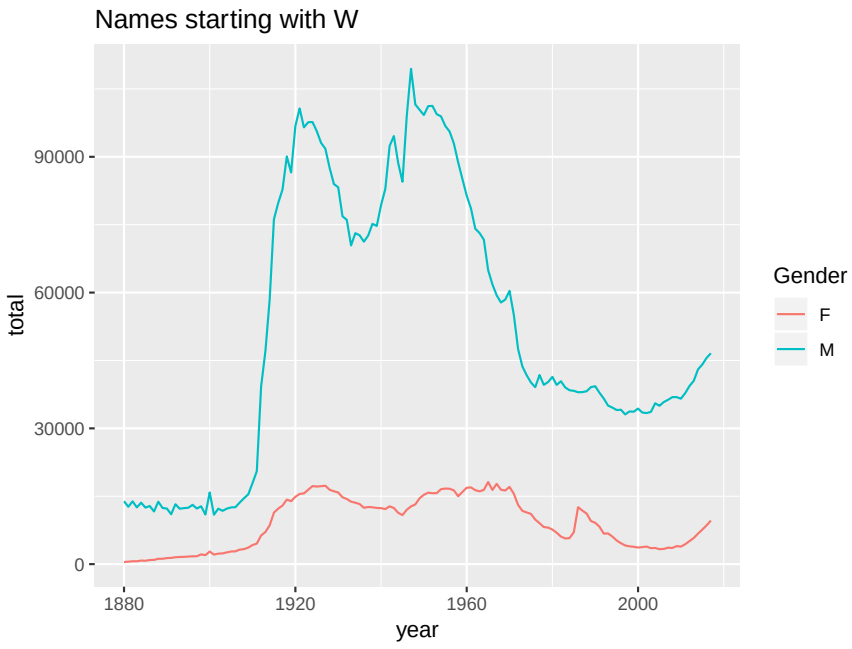
3.5.3.1 Example: babynames

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:

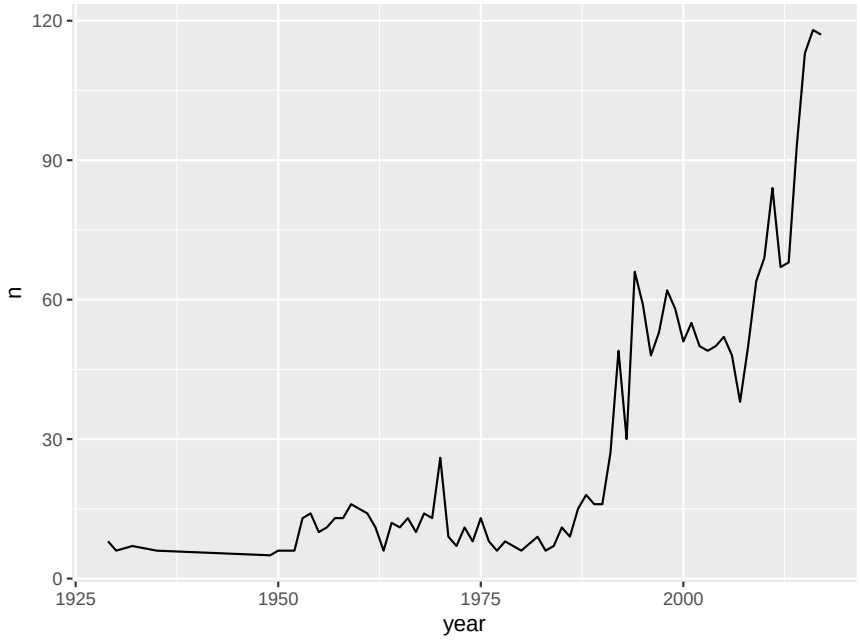
- take the names
- pick out all of those that start with "W"
- separate the genders
- find the total for each year
- do the line graph

```
library(babynames)
babynames %>%
    filter(name %>% substr(1, 1) %>% equals("W")) %>%
    group_by(year, sex) %>%
    summarise(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 %>%                # take babynames
    filter(sex=="F") %>%     # then pick girls only
    group_by(year) %>%       # then separate the years
    mutate(M=max(n)) %>%     # then find most often used name
    filter(n==M) %>%         # then pick only those rows
    ungroup() %>%            # then join data back together
    select(name) %>%         # then select names only
    table() %>%              # then count how often each happened
    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      5
```

```
## 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: 71 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 68 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: 71 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 68 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: 33,098 x 2
##   name    freq
##   <chr>  <dbl>
## 1 Emma     554
## 2 Olivia   533
## 3 Noah     532
## 4 Liam     498
## 5 Sophia   472
## # ... with 3.309e+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.

3.5.4 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> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <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  carb
##   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1        21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 2        21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3 3        22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## # ... with 29 more rows, and 2 more variables: 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)
```

3.6 Interactive web applications with shiny

```
library(shiny)
```

3.6.1 Basic shiny apps

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 parts:

- *ui* contains information about the layout of the page (**u**ser **i**nterface).
- *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:

3.6.1.1 Example 1

```
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 = "WolfgangRolke",
          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

3.6.1.2 Example 2

```
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 textInput, 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:

3.6.1.3 Example 3

```
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'",
            textInput("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:

3.6.1.4 Example 4

```
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:

3.6.1.5 Example 5

```
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:

3.6.1.6 Example 6

```
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)
```

3.6.2 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 is 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.

Uploading an app to shinyapps should be very simple: run the app and click on the Publish button in the upper right corner. I have however had occasional problems with this method. If it does not work run the following command from the console:

```
shinyapps::deployApp("C:\\Users\\...\\name_of-app",
                     account="your_account_name")
```

For more information on how to get up and running with shinyapps.io read https://shiny.rstudio.com/articles/shinyapps.html

```
library(magrittr)
library(dplyr)
```

3.7 Character Manipulation with *stringr*

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(str_pad("Great!", width = 7), width = 8, 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 ar
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.
```

3.7.0.1 Example: 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.

3.7.0.2 Example: Dracula vs The Time Machine

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 catalog:

```
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 *tidyr*.

We begin by joining the two books together, with a new column identifying the book:

```
library(tidyr)
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:

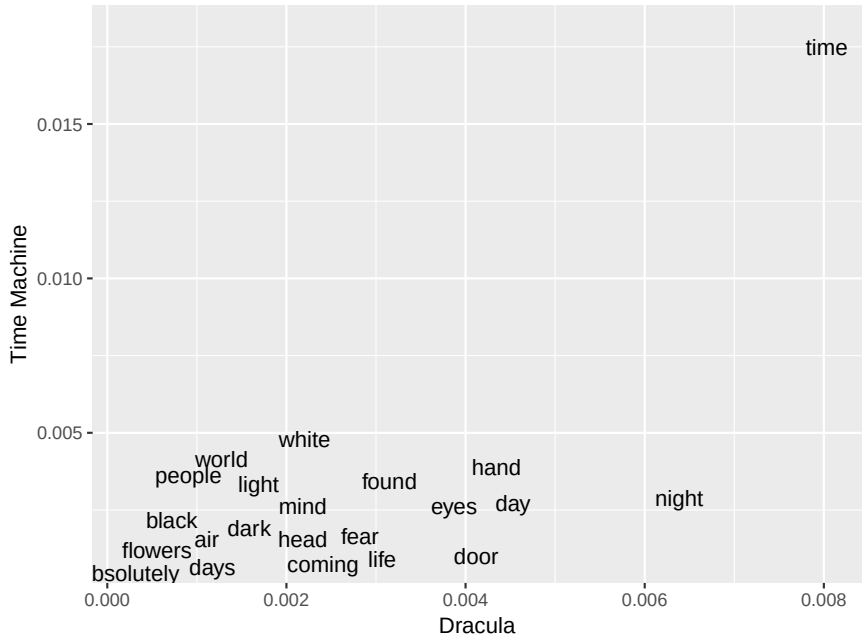
```
freqs %>%
  mutate(word=str_extract(word, "[a-z']+")) %>%
        # take out things like , etc
  count(book, word) %>%
  group_by(book) %>%
  mutate(prop=n/sum(n)) %>% # take into account
                  # different lengths of the books
  ungroup() %>%
  filter(n>10) %>%  # consider only words 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)
```



3.8 Dates with *lubridate*

```
library(tidyverse)
library(lubridate)
```

For a more detailed discussion see Dates and Times Made Easy with lubridate.

3.8.1 Time and date basics

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 calender 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!

3.8.2 Create a date object

to get today's time and date:

```
today()
```

```
## [1] "2019-07-15"
```

```
now()
```

```
## [1] "2019-07-15 14:32:16 -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 division and %% for modulo:

```
12.34 %% 2
```

```
## [1] 0.34
```

```
12.34 %/% 2
```

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

```
2 * (12.34 %/% 2) + 12.34 %% 2
```

```
## [1] 12.34
```

with this we find

```
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
```

3.8.3 Time spans

lubridate has a number of functions to do arithmetic with dates. For example, my age is

```
today()
```

```
## [1] "2019-07-15"
```

```
my.age <- today() - ymd(19610602)
as.duration(my.age)
```

```
## [1] "1834012800s (~58.12 years)"
```

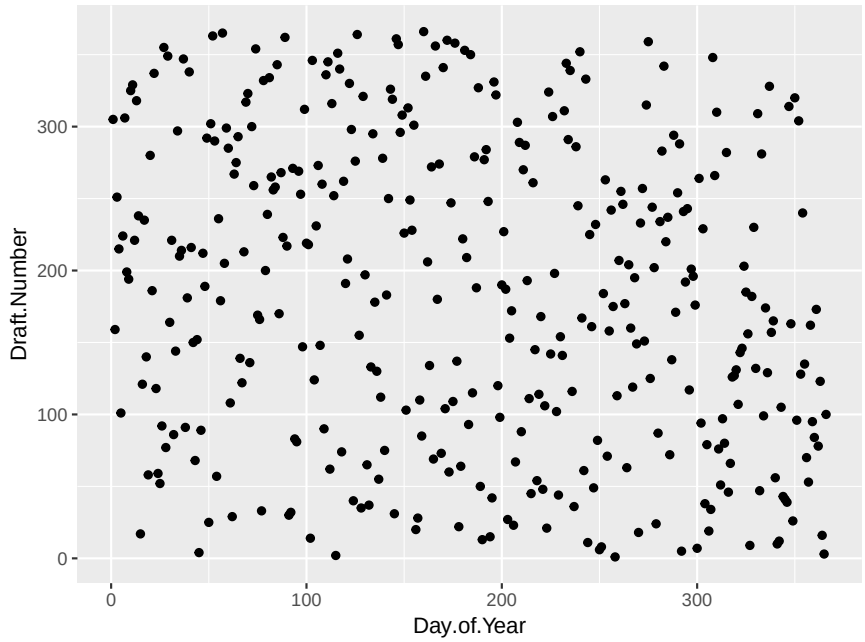
3.9 Factors with *forcats*

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)
```

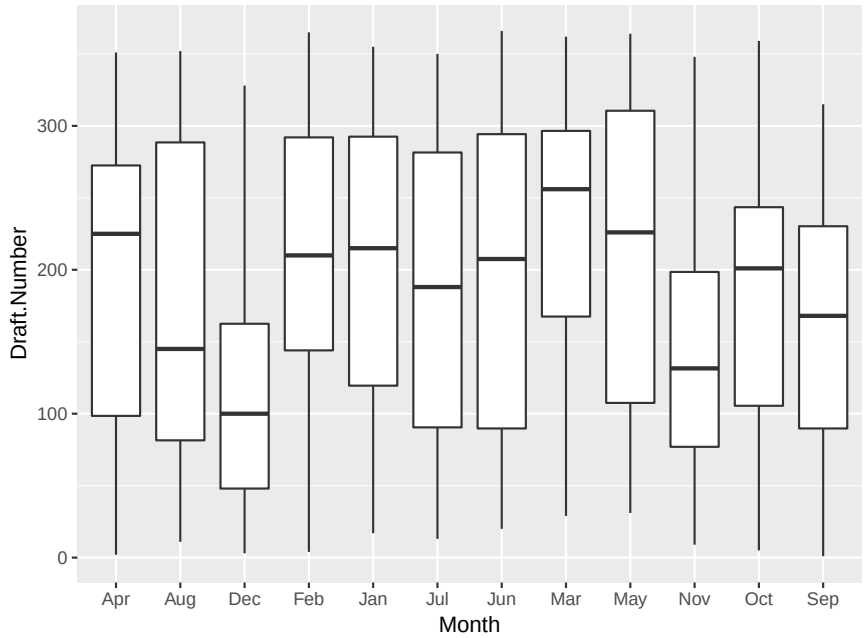
Let's remind ourselves of the base R commands first. Consider the data set *draft*, with the results of the 1970s military draft:

```
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:

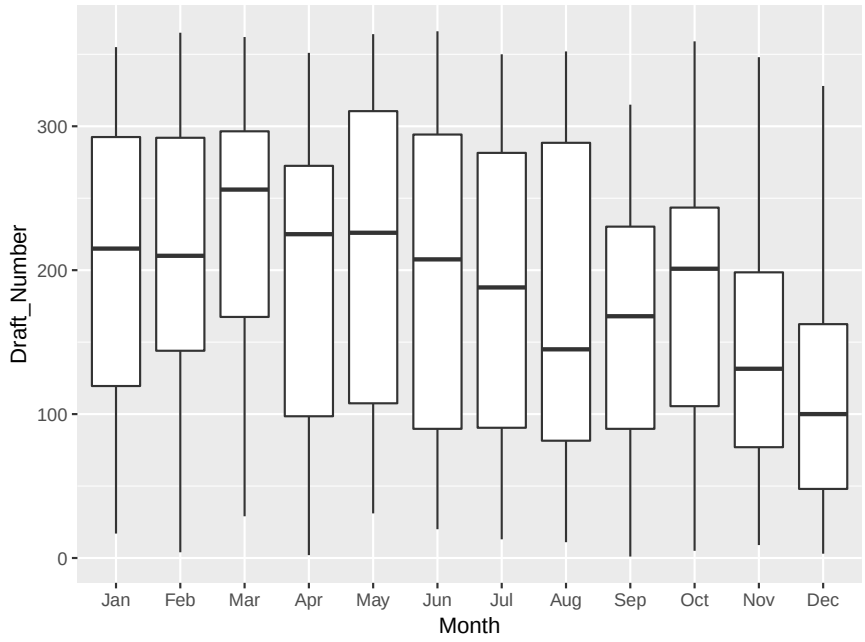
```
draft %>%
  ggplot(aes(Month, Draft.Number)) +
     geom_boxplot()
```



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$Month,
                    levels = lvls,
                    ordered = TRUE)
df <- data.frame(Month=Month.fac,
                 Draft_Number=draft$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

```
lvls <- unique(draft$Month)
```

The forcats 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 ma~    26 White $8000 to~ Ind,near ~ Protes~ Southe~      12
##  2  2000 Divorced     48 White $8000 to~ Not str r~ Protes~ Baptis~      NA
##  3  2000 Widowed      67 White Not appl~ Independe~ Protes~ No den~       2
##  4  2000 Never ma~    39 White Not appl~ Ind,near ~ Orthod~ Not ap~       4
```

```
##  5  2000 Divorced     25 White Not appl~ Not str d~ None    Not ap~       1
##  6  2000 Married      25 White $20000 -~ Strong de~ Protes~ Southe~      NA
##  7  2000 Never ma~    36 White $25000 o~ Not str r~ Christ~ Not ap~       3
##  8  2000 Divorced     44 White $7000 to~ Ind,near ~ Protes~ Luther~      NA
##  9  2000 Married      44 White $25000 o~ Not str d~ Protes~ Other         0
## 10  2000 Married      47 White $25000 o~ Strong re~ Protes~ Southe~       3
## # ... with 21,473 more rows
```

which has the results of the General Social Survey (http://gss.norc.org). This is a survey in the US done by the University of Chicago. We will use it to illustrate forcats.

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:

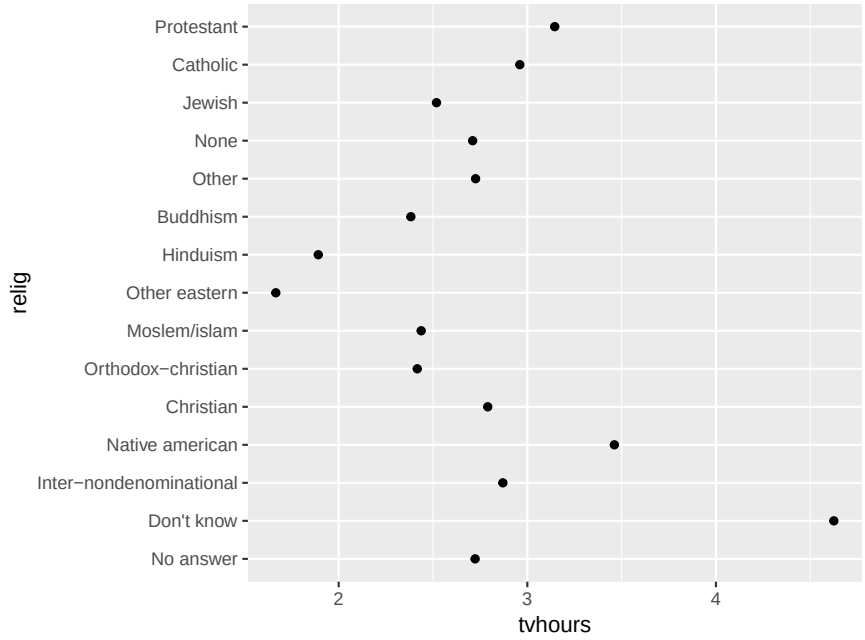
```
levels(gss_cat$race)
```

```
## [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:

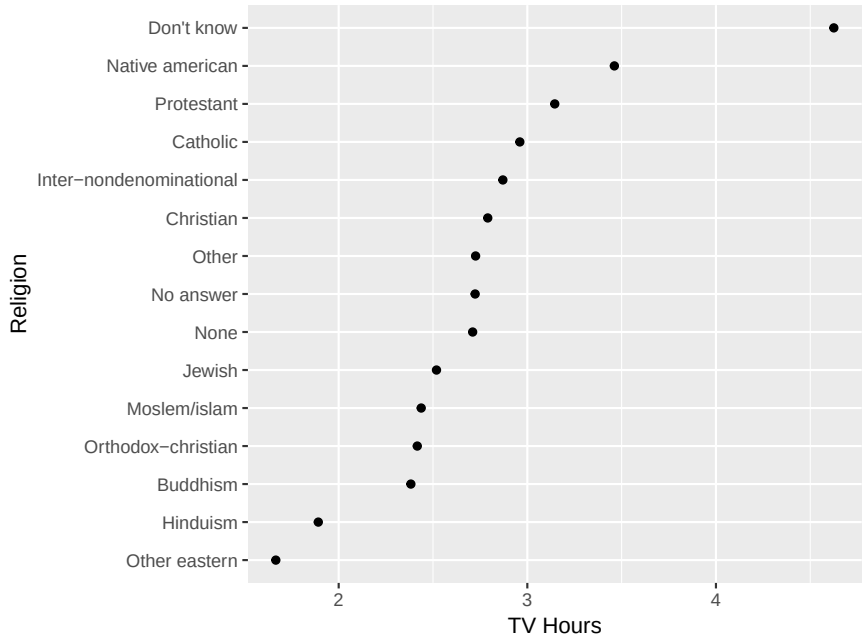
```
gss_cat %>%
  group_by(relig) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
```

```
  ) ->
  tv.relig
tv.relig %>%
  ggplot(aes(tvhours, relig)) +
  geom_point()
```



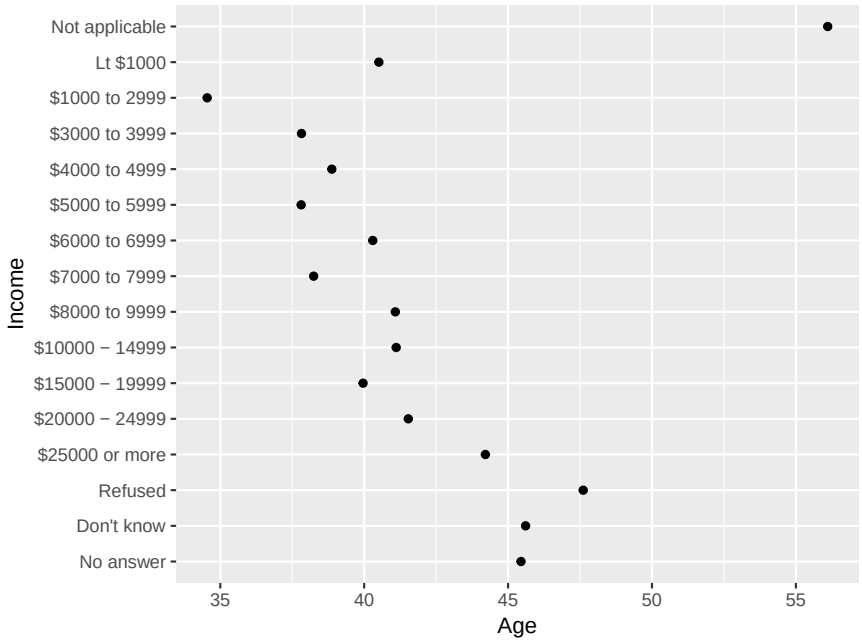
This graph is hard to read, mainly because there is no ordering. But unlike Month the variable itself doesn't have any either. So maybe we should order by size:

```
tv.relig %>%
  ggplot(aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point() +
  labs(x="TV Hours",
       y="Religion")
```



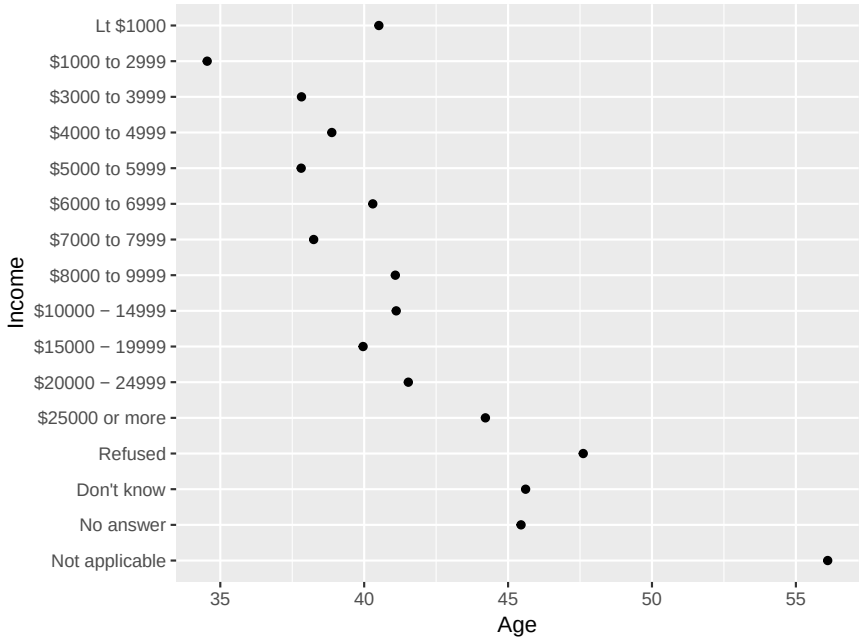
Let's see how income varies with age:

```
gss_cat %>%
  group_by(rincome) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    n = n()
  ) ->
  rincome
rincome %>%
  ggplot(aes(age, rincome)) +
  geom_point() +
  labs(x="Age", y="Income")
```



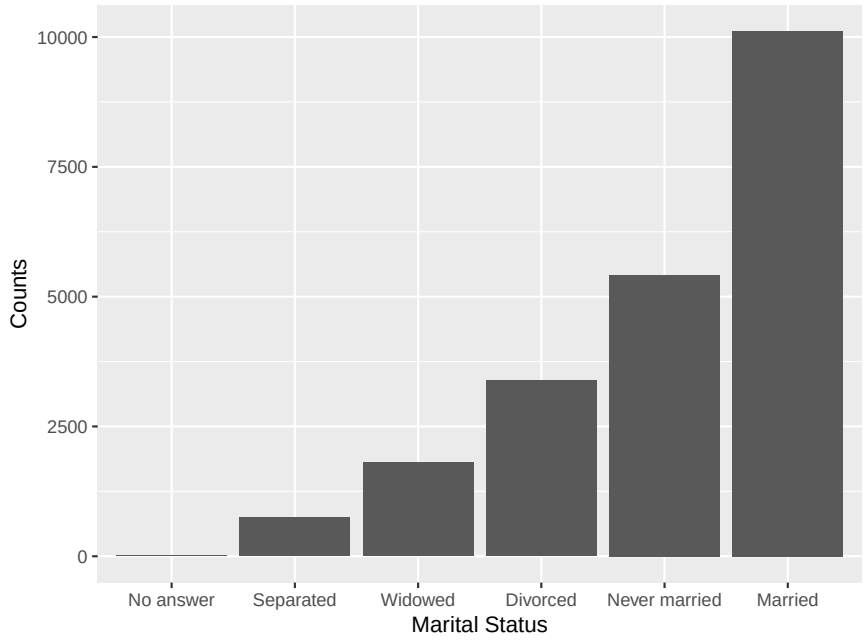
What ordering makes the most sense here? There are two types of levels: those with actual numbers, and those like "Not applicable". We should probably separate them.

```
rincome %>%
  ggplot(aes(age, fct_relevel(rincome,
          c("Not applicable", "Refuse", "No answer")))) +
  geom_point()+
  labs(x="Age", y="Income")
```



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() +
     labs(x="Marital Status",
          y="Counts")
```



3.10 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/WolfgangRolke/Computing-with-R.

3.11 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 message 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 substantial change to one of these files, repeat the Commit-Push steps to upload the file to github.

4 Statistics with R

4.1 Basic Statistics

In this section we will use R to analyze a number of data sets. I will assume that you are familiar with basic concepts such as confidence intervals and hypothesis testing. If you are not you can read up on them on the web page of my course ESMA 3101: Introduction to Statistics I.

4.1.1 Basic Summaries and Graphs

We have talked about the **upr admissions** data before. Here are some simple things to do when looking at this kind of data:

- Tables

```
Gender <- table(upr$Gender)
names(Gender) <- c("Female", "Male")
Percentage <- round(Gender/sum(Gender)*100, 1)
cbind(Gender, Percentage)
```

```
##        Gender Percentage
## Female  11487       48.5
## Male    12179       51.5
```

- Contingency Tables

```
table(upr$Year, upr$Gender)
```

```
##
##           F    M
##   2003 1102 1151
##   2004 1040 1118
##   2005 1162 1138
##   2006 1137 1098
##   2007 1208 1256
##   2008 1219 1219
##   2009 1180 1237
##   2010  958 1073
```

```
##   2011  853  919
##   2012  769  979
##   2013  859  991
```

or percentages based on grand total:

```
round(table(upr$Year, upr$Gender)/length(upr$Year)*100, 1)
```

```
##
##          F   M
##   2003 4.7 4.9
##   2004 4.4 4.7
##   2005 4.9 4.8
##   2006 4.8 4.6
##   2007 5.1 5.3
##   2008 5.2 5.2
##   2009 5.0 5.2
##   2010 4.0 4.5
##   2011 3.6 3.9
##   2012 3.2 4.1
##   2013 3.6 4.2
```

but often more interesting are percentages based on row totals

```
tmp <- table(upr$Year, upr$Gender)
apply(tmp, 1, sum)
```

```
## 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013
## 2253 2158 2300 2235 2464 2438 2417 2031 1772 1748 1850
```

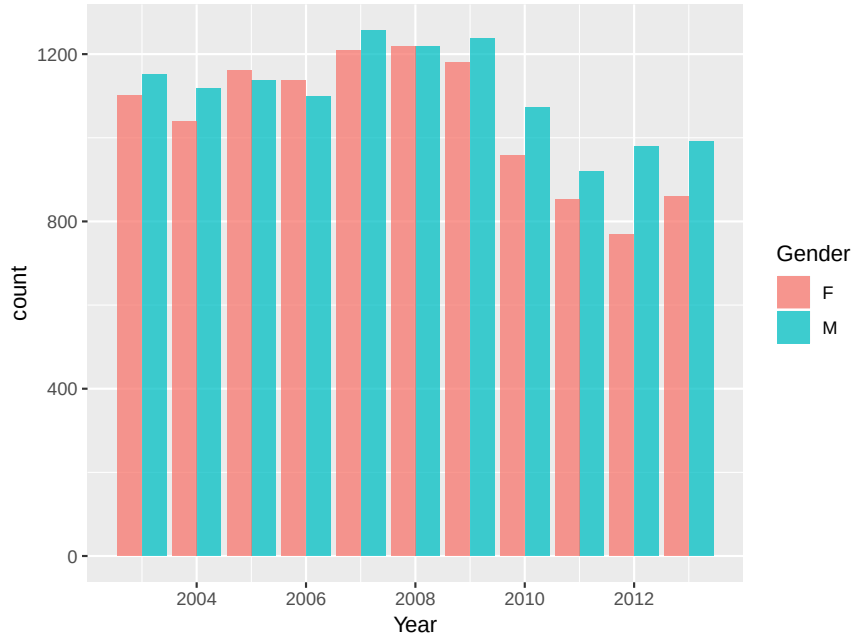
```
tmp <- round(tmp/apply(tmp, 1, sum)*100, 1)
tmp
```

```
##
##           F    M
##   2003 48.9 51.1
##   2004 48.2 51.8
##   2005 50.5 49.5
##   2006 50.9 49.1
##   2007 49.0 51.0
##   2008 50.0 50.0
##   2009 48.8 51.2
##   2010 47.2 52.8
##   2011 48.1 51.9
##   2012 44.0 56.0
##   2013 46.4 53.6
```

or column totals

- Bar Charts

```
ggplot(upr, aes(x=Year, fill=Gender)) +
  geom_bar(position="dodge", alpha=0.75)
```

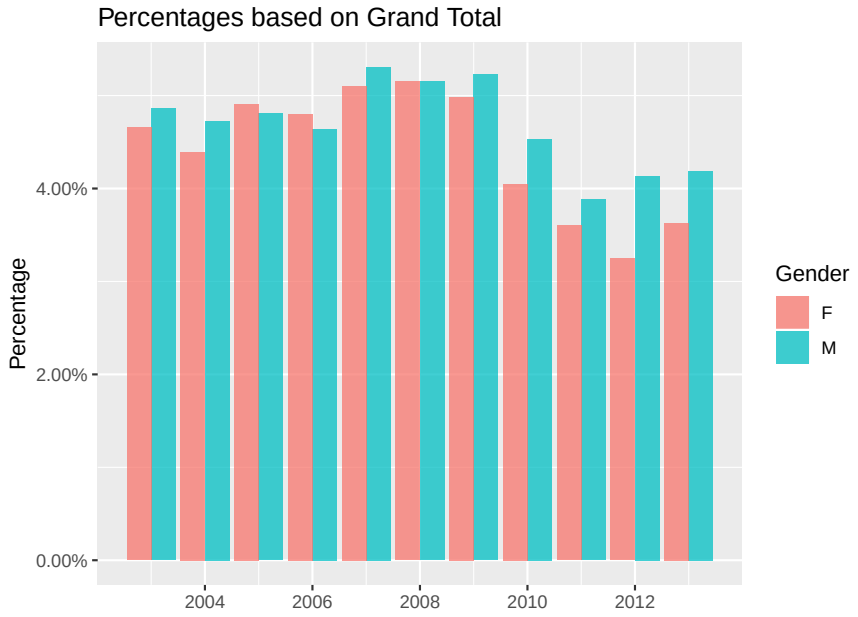



```
    labs(x="", y="Counts")
```

```
## $x
## [1] ""
##
## $y
## [1] "Counts"
##
## attr(,"class")
## [1] "labels"
```

or based on grand total percentages:

```
ggplot(upr, aes(x=Year, fill=Gender)) +
  geom_bar(aes(y = (..count..)/sum(..count..)),
           position="dodge", alpha=0.75) +
  scale_y_continuous(labels=scales::percent) +
    labs(x="", y="Percentage",
         title="Percentages based on Grand Total")
```



4.1.2 Numerical Summaries

```
round(mean(upr$Freshmen.GPA, na.rm=TRUE), 3)
```

```
## [1] 2.733
```

```
round(median(upr$Freshmen.GPA, na.rm=TRUE), 3)
```

```
## [1] 2.83
```

```
round(sd(upr$Freshmen.GPA, na.rm=TRUE), 3) # Standard Deviation
```

```
## [1] 0.779
```

```
round(quantile(upr$Freshmen.GPA,
               probs = c(0.1, 0.25, 0.75, 0.9),
               na.rm=TRUE), 3) # Quantiles and Quartiles
```

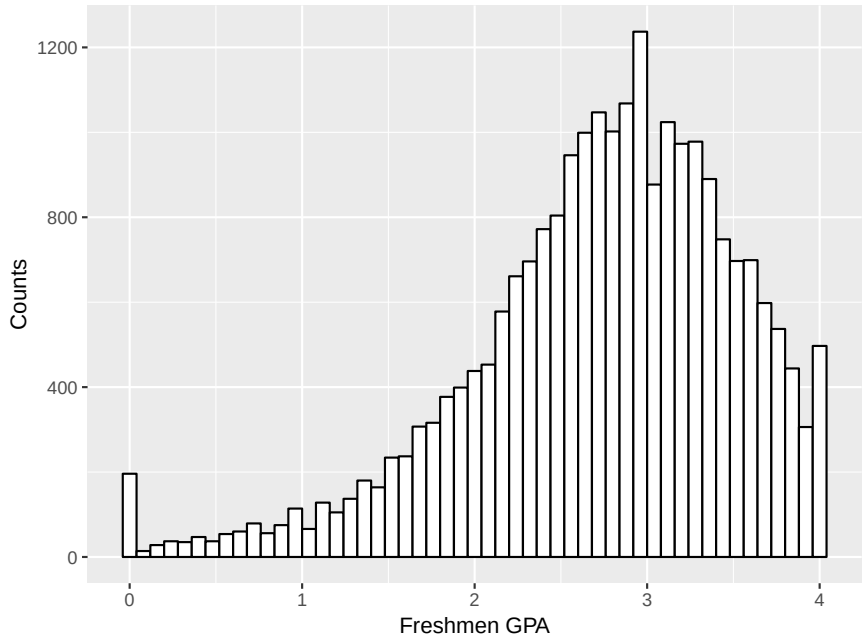
```
##   10%   25%   75%   90%
## 1.71 2.32 3.28 3.65
```

Notice also that I have rounded all the answers. Proper rounding is an important thing to do!

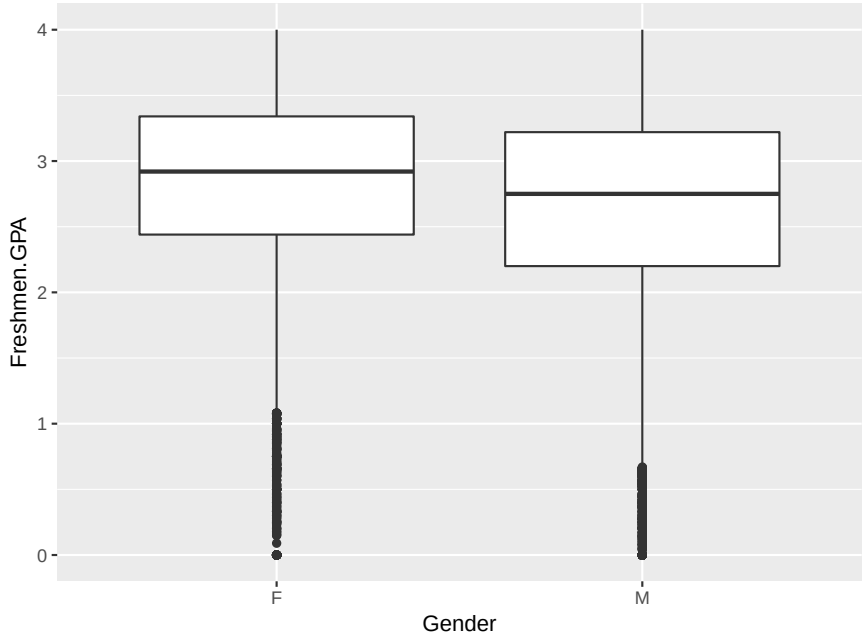
4.1.3 Histogram and Boxplot

```
bw <- 4/50
# use about 50 bins
ggplot(upr, aes(Freshmen.GPA)) +
  geom_histogram(color = "black",
```

```
                 fill = "white",
                 binwidth = bw) +
  labs(x = "Freshmen GPA", y = "Counts")
```




```
ggplot(upr, aes(Gender,Freshmen.GPA)) +
  geom_boxplot()
```

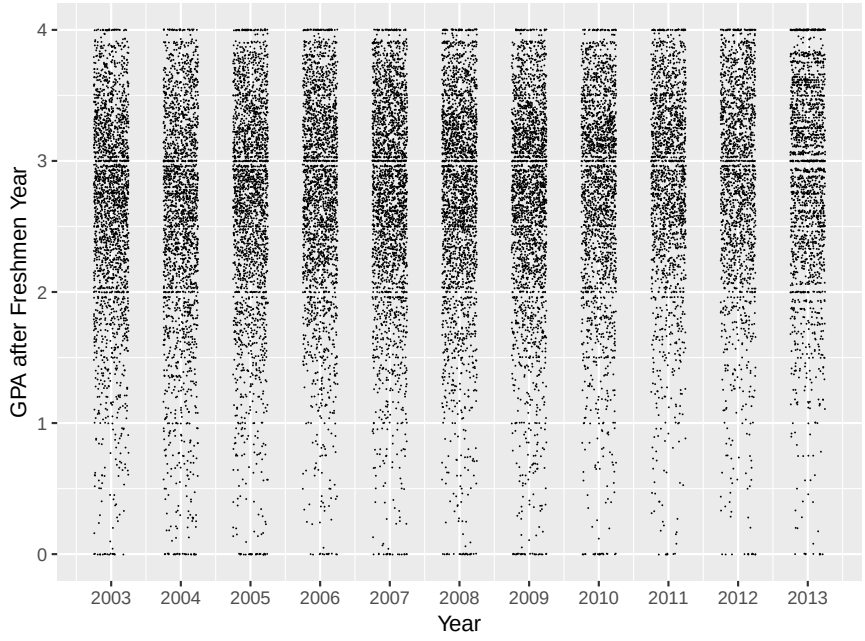


4.1.4 Two Quantitative Variables

```
round(cor(upr$Year, upr$Freshmen.GPA,
    use="complete.obs"), 3)
```

```
## [1] 0.097
```

```
ggplot(upr, aes(Year, Freshmen.GPA)) +
  geom_jitter(shape=16, size=1/10, width = 0.25) +
  scale_x_continuous(breaks = 2003:2013) +
  labs(x="Year", y="GPA after Freshmen Year")
```



4.1.5 Inference for the Mean

The basic R command for inference for a population mean is *t.test*:

4.1.5.1 Example: Simon Newcomb's Measurements of the Speed of Light

Simon Newcomb made a series of measurements of the speed of light between July and September 1880. He measured the time in seconds that a light signal took to pass from his laboratory on the Potomac River to a mirror at the base of the Washington Monument and back, a total distance of 7400m. His first measurement was 0.000024828 seconds, or 24,828 nanoseconds (109 nanoseconds = 1 second).

We want to find a 95% confidence interval the speed of light.

```
attach(newcomb)
t.test(Measurement)
```

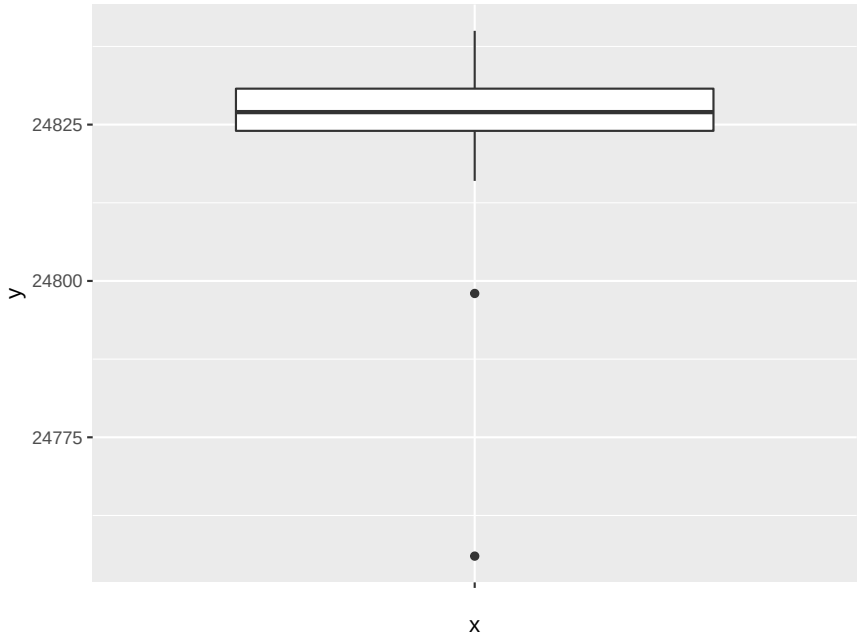
```
##
##  One Sample t-test
##
## data:  Measurement
## t = 18770, df = 65, p-value <2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  24824 24829
## sample estimates:
## mean of x
##     24826
```

The assumptions for this method are:

- data comes from a normal distribution
- or data set is large enough

Let's check:

```
y <- Measurement
dta <- data.frame(y = y,
                  x = rep(" ", length(y)))
 ggplot(aes(x, y), data = dta) +
   geom_boxplot()
```



It seems there is at least one serious outlier. This should not happen if the data came from a normal distribution.

We could proceed in one of two ways:

- eliminate outlier

```
x <- Measurement[Measurement>24775]
t.test(x)$conf.int
```

```
## [1] 24826 24829
## attr(,"conf.level")
## [1] 0.95
```

- use the median:

```
median(Measurement)
```

```
## [1] 24827
```

but now we need to find a confidence interval for the median. That can be done with the *non-parametric Wilcoxon Rank Sum* method:

```
wilcox.test(newcomb$Measurement, conf.int = TRUE)
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  newcomb$Measurement
## V = 2211, p-value = 2e-12
## alternative hypothesis: true location is not equal to 0
## 95 percent confidence interval:
##  24826 24829
## sample estimates:
## (pseudo)median
##          24827
```

4.1.5.2 Example: Resting Period of Monarch Butterflies

Some Monarch butterflies fly early in the day, others somewhat later. After the flight they have to rest for a short period. It has been theorized that the resting period (RIP) of butterflies flying early in the morning is shorter because this is a thermoregulatory mechanism, and it is cooler in the mornings. The mean RIP of all Monarch butterflies is 133 sec. Test the theory at the 10% level.

Research by Anson Lui, Resting period of early and late flying Monarch butterflies Danaeus plexippus, 1997

1. Parameter: mean μ

2. Method: 1-sample t

3. Assumptions: normal data or large sample

4. $\alpha = 0.1$

5. $H_0 : \mu = 133$ (RIP is the same for early morning flying butterflies as all others)

6. $H_0 : \mu < 133$ (RIP is the shorter for early morning flying butterflies)

7.

```
t.test(butterflies$RIP.sec.,
       mu=133,
       alternative = "less")$p.value
```

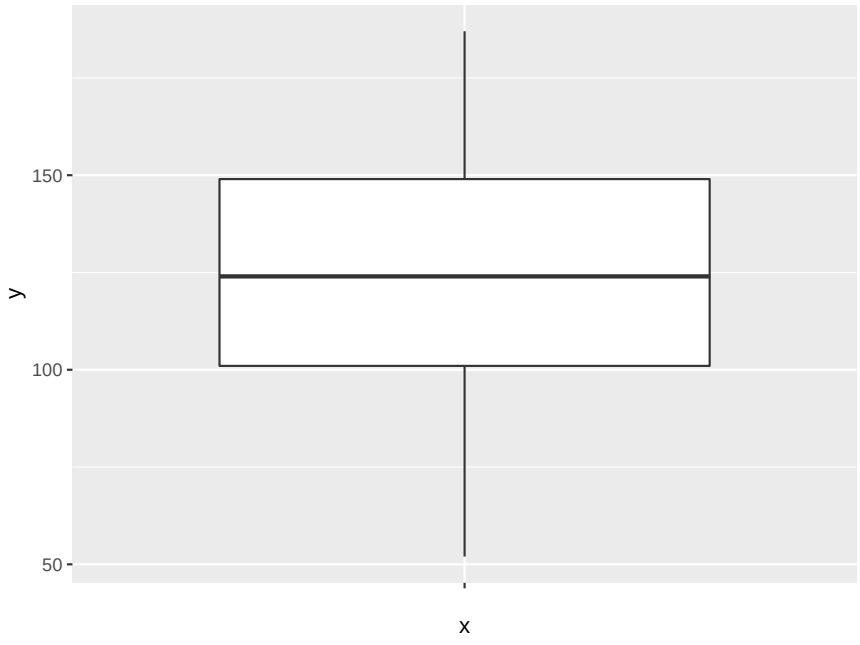
```
## [1] 0.05584
```

8. $p = 0.0558 < \alpha = 0.1$, so we reject the null hypothesis

9. It appears the resting time is somewhat shorter, but the conclusion is not a strong one.

Checking the assumption:

```
y <- butterflies$RIP.sec.
dta <- data.frame(y = y,
                  x = rep(" ", length(y)))
 ggplot(aes(x, y), data = dta) +
   geom_boxplot()
```



looks good.

4.1.6 Inference for Proportions

The R routine for inference for a proportion (or a probability or a percentage) is *binom.test*. This implements a method by Clopper and Pearson (1934). This method is exact and has no

assumptions.

Note The formula discussed in many introductory statistic courses for the confidence interval is

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

where $\hat{p}$ is the proportion of success. This leads to confidence intervals that are now known to be quite wrong, and so this method should not be used anymore. The same is true for the corresponding hypothesis test. This method (actually a slight improvement due to Wilson (1927)) is implemented in R by *prop.test*.

4.1.6.1 Example: Jon Kerrichs Coin

The South African Jon Kerrich spent some time in a German prisoner of war camp during world war I. He used his time to flip a coin 10000 times, resulting in 5067 heads.

Test at the 5% level of significance whether 5067 heads in 10000 flips are compatible with a fair coin.

1. Parameter: proportion π
2. Method: exact binomial
3. Assumptions: None
4. $\alpha = 0.05$
5. $H_0 : \pi = 0.5$ (50% of flips result in "Heads", coin is fair)
6. $H_a : \pi \neq 0.5$ (coin is not fair)
7.

```
binom.test(x = 5067, n = 10000)$p.value
```

```
## [1] 0.1835
```

8. $p = 0.1835 > \alpha = 0.05$, so we fail to reject the null hypothesis.
9. it appears Jon Kerrich's coin was indeed fair.

4.1.6.2 Example: Sample Size for Polling

Say some polling institute wants to conduct a poll for the next election for president. They will then find a 95% confidence interval and they want this interval to have an error of 3 percentage points (aka ± 0.03). What sample size do they need?

In American politics the two parties are always very close, so in a poll with n people about n/2 will vote for one or the other party. Let's do a little trial and error:

```
n <- 100
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.1017
```

Now that is to large, so

```
n <- 200
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.07134
```

```
n <- 400
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.05009
```

```
n <- 800
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.03522
```

```
n <- 1200
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.02868
```

```
n <- 1100
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.02997
```

```
n <- 1050
diff(as.numeric(binom.test(n/2, n)$conf.int)/2)
```

```
## [1] 0.03068
```

There is something quite remarkable about this result!

4.1.7 Correlation

4.1.7.1 Example: UPR Admissions data

What are the correlations between the various variables?

```
head(upr, 2)
```

```
##      ID.Code Year Gender Program.Code Highschool.GPA Aptitud.Verbal
```

```
## 1 00C2B4EF77 2005      M          502           3.97            647
## 2 00D66CF1BF 2003      M          502           3.80            597
##   Aptitud.Matem Aprov.Ingles Aprov.Matem Aprov.Espanol IGS Freshmen.GPA
## 1           621          626         672           551 342         3.67
## 2           726          618         718           575 343         2.75
##   Graduated Year.Grad. Grad..GPA Class.Facultad
## 1        Si       2012      3.33           INGE
## 2        No         NA        NA           INGE
```

Let's take out the those variables that are not useful for predicting success, either because we don't have their value at the time of the admissions process (Freshmen.GPA) or for legal reasons (Gender)

```
x <- upr[, -c(1:4, 11:16)]
head(x, 2)
```

```
##   Highschool.GPA Aptitud.Verbal Aptitud.Matem Aprov.Ingles Aprov.Matem
## 1           3.97            647           621          626         672
## 2           3.80            597           726          618         718
##   Aprov.Espanol
## 1           551
## 2           575
```

```
round(cor(x, use = "complete.obs") ,3)
```

```
##                Highschool.GPA Aptitud.Verbal Aptitud.Matem Aprov.Ingles
## Highschool.GPA          1.000          0.176         0.156        0.049
## Aptitud.Verbal          0.176          1.000         0.461        0.513
## Aptitud.Matem           0.156          0.461         1.000        0.456
## Aprov.Ingles            0.049          0.513         0.456        1.000
## Aprov.Matem             0.216          0.474         0.819        0.481
## Aprov.Espanol           0.247          0.602         0.389        0.428
##                Aprov.Matem Aprov.Espanol
## Highschool.GPA       0.216         0.247
## Aptitud.Verbal       0.474         0.602
## Aptitud.Matem        0.819         0.389
## Aprov.Ingles         0.481         0.428
## Aprov.Matem          1.000         0.404
## Aprov.Espanol        0.404         1.000
```

somewhat surprising, Highschool.GPA is not highly correlated with any of the tests. The highest correlation is between the two math tests (0.819)

4.1.7.2 Example: The 1970's Military Draft

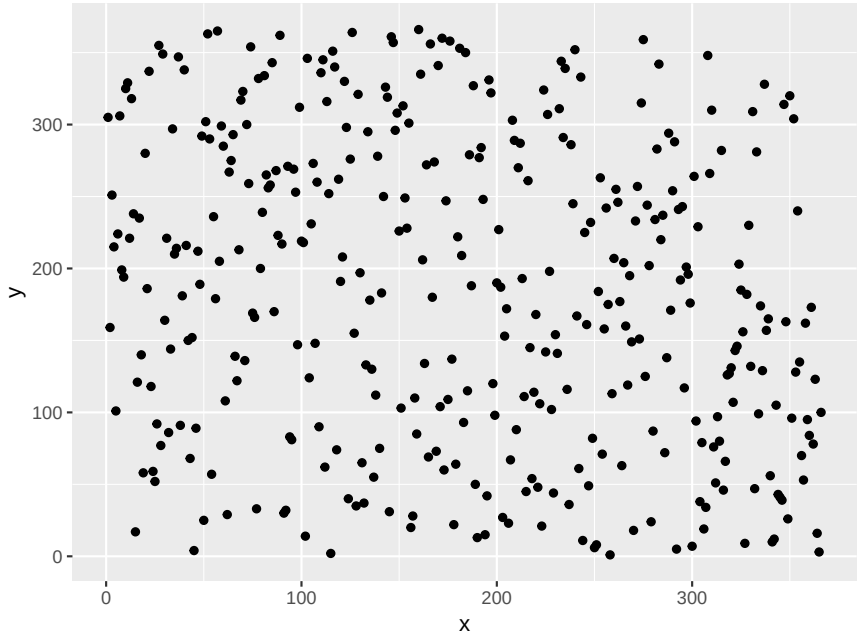
In 1970, Congress instituted a random selection process for the military draft. All 366 possible birth dates were placed in plastic capsules in a rotating drum and were selected one by one. The first date drawn from the drum received draft number one and eligible men born on that date were drafted first. In a truly random lottery there should be no relationship between

the date and the draft number.

Question: **was the draft was really "random"?**

Here we have two quantitative variables, so we start with the scatterplot:

```
attach(draft)
ggplot(aes(x, y),
       data=data.frame(x=Day.of.Year,
                       y=Draft.Number)) +
  geom_point()
```



and this does not look like there is a problem with independence.

However:

1) Parameter: Pearson's correlation coefficient ρ
2) Method: Test for Pearson's correlation coefficient ρ
3) Assumptions: relationship is linear and that there are no outliers.
4) $\alpha = 0.05$

5) $H_0 : \rho = 0$ (no relationship between Day of Year and Draft Number)
6) $H_a : \rho \neq 0$ (some relationship between Day of Year and Draft Number)
7)

```
cor.test(Draft.Number, Day.of.Year)$p.value
```

```
## [1] 1.264e-05
```

8) $p = 0.0000 < \alpha = 0.05$, so we reject the null hypothesis,

9) There is a statistically significant relationship between Day of Year and Draft Number.

4.1.8 Categorical Data Analysis - Tests for Independence

4.1.8.1 Example: Drownings in Los Angeles

Data is from O'Carroll PW, Alkon E, Weiss B. Drowning mortality in Los Angeles County, 1976 to 1984, JAMA, 1988 Jul 15;260(3):380-3.

Drowning is the fourth leading cause of unintentional injury death in Los Angeles County. They examined data collected by the Los Angeles County Coroner's Office on drownings that occurred in the county from 1976 through 1984. There were 1587 drownings (1130 males and 457 females) during this nine-year period

```
drownings
```

```
##                          Male Female
## Private Swimming Pool   488    219
## Bathtub                 115    132
## Ocean                   231     40
## Freshwater bodies       155     19
## Hottubs                  16     15
## Reservoirs               32      2
## Other Pools              46     14
## Pails, basins, toilets    7      4
## Other                    40     12
```

Here we have two categorical variables (Method of Drowning and Gender), both categorical. We want to know whether the variables are independent. The most popular method of analysis for this type of problem is **Pearson's chi square test of independence**. It is done with the command *chisq.test* and it has the assumption of no expected counts less than 5.

1. Parameters of interest: measure of association

2. Method of analysis: chi-square test of independence

3. Assumptions of Method: all expected counts greater than 5

4. Type I error probability α=0.05

5. H_0: Classifications are independent = there is no difference in the method of drowning between men and women.

6. H_a: Classifications are dependent = there is some difference in the method of drowning between men and women.

7.

```
chisq.test(drownings)
```

```
##
##  Pearson's Chi-squared test
##
## data:  drownings
## X-squared = 144, df = 8, p-value <2e-16
```

8. $p = 0 < \alpha$=0.05, we reject the null hypothesis, there is a statistically significant difference between men and women and where they drown.

Let's see whether there is a problem with the assumptions:

```
chisq.test(drownings)$expected
```

```
##                            Male  Female
## Private Swimming Pool  503.409 203.591
## Bathtub                175.873  71.127
## Ocean                  192.962  78.038
## Freshwater bodies      123.894  50.106
## Hottubs                 22.073   8.927
## Reservoirs              24.209   9.791
## Other Pools             42.722  17.278
## Pails, basins, toilets   7.832   3.168
## Other                   37.026  14.974
```

and we see that the expected counts of Pails, basins, toilets and Female is 3.2. In real life this would be considered ok, but it would also be easy to fix:

```
newmale <- c(drownings[1:7, 1], 7+40)
newfemale <- c(drownings[1:7, 2], 4+12)
newdrown <- cbind(newmale, newfemale)
newdrown
```

```
##                        newmale newfemale
## Private Swimming Pool      488       219
## Bathtub                    115       132
## Ocean                      231        40
## Freshwater bodies          155        19
## Hottubs                     16        15
## Reservoirs                  32         2
## Other Pools                 46        14
##                             47        16
```

```
out <- chisq.test(newdrown)
out$expected
```

```
##                        newmale newfemale
## Private Swimming Pool   503.41   203.591
```

```
## Bathtub                175.87    71.127
## Ocean                  192.96    78.038
## Freshwater bodies      123.89    50.106
## Hottubs                 22.07     8.927
## Reservoirs              24.21     9.791
## Other Pools             42.72    17.278
##                         44.86    18.142
```

```
out$p.value
```

```
## [1] 8.519e-28
```

4.1.9 Comparing the Means of Several Populations - ANOVA

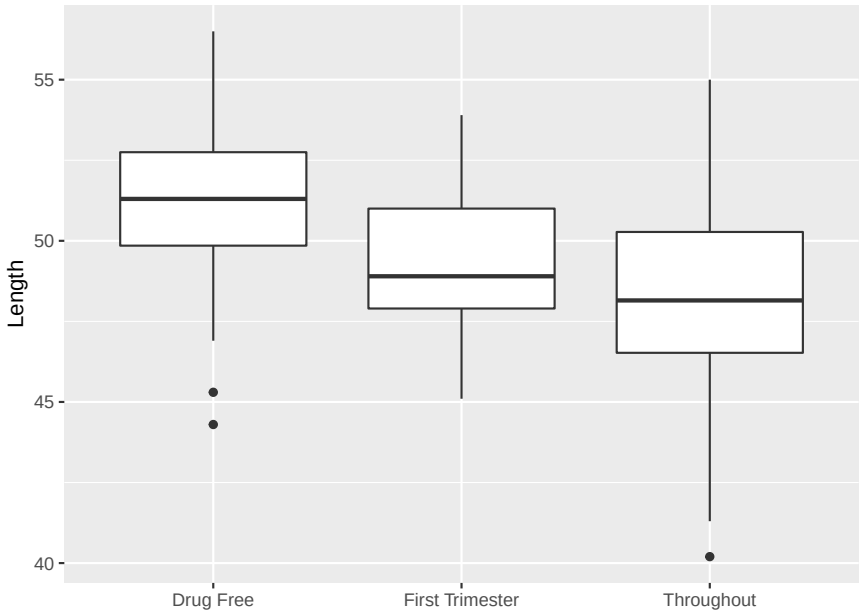
4.1.9.1 Example: Mothers Cocaine Use and Babies Health

Chasnoff and others obtained several measures and responses for newborn babies whose mothers were classified by degree of cocaine use.

The study was conducted in the Perinatal Center for Chemical Dependence at Northwestern University Medical School. The measurement given here is the length of the newborn.

Source: Cocaine abuse during pregnancy: correlation between prenatal care and perinatal outcome
Authors: SN MacGregor, LG Keith, JA Bachicha, and IJ Chasnoff
Obstetrics and Gynecology 1989;74:882-885

```
attach(mothers)
dta <- data.frame(y = Length,
                  x = Status)
 ggplot(aes(x, y), data = dta) +
   geom_boxplot() +
   labs(x="", y="Length")
```




```
out <- matrix(0, 3, 3)
colnames(out) <- c("Size", "Mean", "SD")
rownames(out) <- unique(Status)
out[, 1] <- tapply(Length, Status, length)
out[, 2] <- round(tapply(Length, Status, mean), 2)
out[, 3] <- round(tapply(Length, Status, sd), 2)
out
```

```
##                 Size Mean  SD
## Drug Free         39 51.1 2.9
## First Trimester   19 49.3 2.5
## Throughout        36 48.0 3.6
```

The standard method for this problem is called **ANOVA** (Analysis of Variance) and is run with the *aov* command.

1. Parameters of interest: group means
2. Method of analysis: ANOVA
3. Assumptions of Method: residuals have a normal distribution, groups have equal variance
4. Type I error probability α=0.05
5. Null hypothesis H_0: $\mu_1 = \mu_2 = \mu_3$ (groups have the same means)
6. Alternative hypothesis H_a: $\mu_i \neq \mu_j$ (at least two groups have different means)
7.

```
fit <- aov(Length~Status)
summary(fit)
```

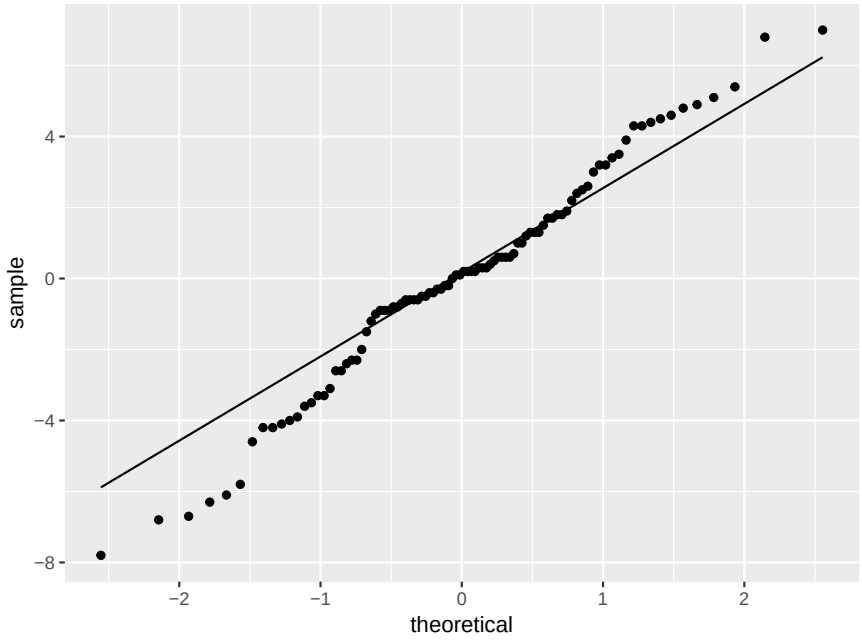
```
##              Df Sum Sq Mean Sq F value  Pr(>F)
## Status        2    181    90.7    9.32 0.00021
## Residuals    91    886     9.7
```

8. $0.000 < 0.05$, there is some evidence that the group means are not the same, the babies whose mothers used cocaine tend to be a little shorter (less healthy?)

In step 3 we have the assumptions

a. residuals have a normal distribution

```
ggplot(data.frame(x=fit$res), aes(sample=x)) +
  stat_qq() +
  stat_qq_line()
```



looks fine

b. groups have equal variance

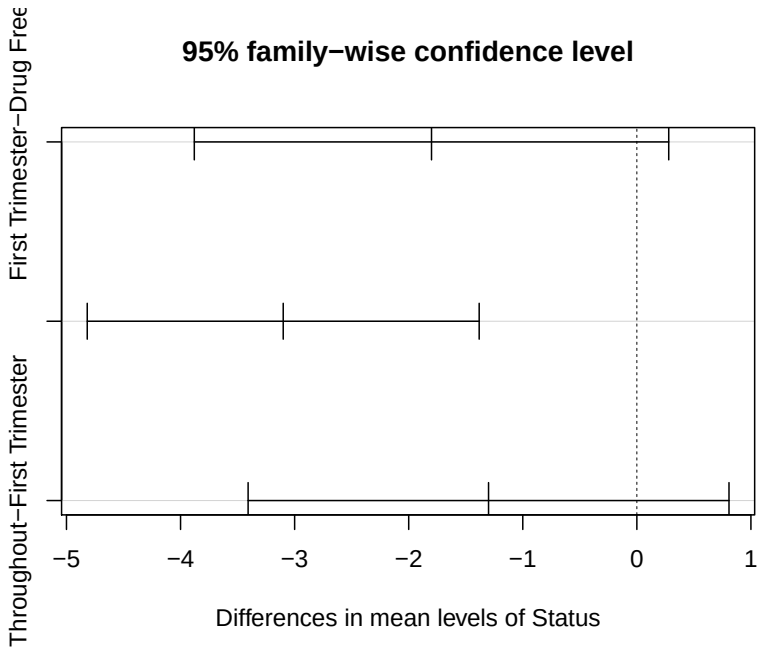
The box plot shows that the variances are about the same.

Often if the null of no difference is rejected, one wants to go a step further and do a pairwise comparison:

- is Drug Free different from First Trimester?
- is First Trimester different from Throughout?

There are a number of methods known for this problem, a popular one is by **Tukey**:

```
tuk <- TukeyHSD(fit)
plot(tuk)
```



this draws confidence intervals for the difference in means of all pairs. If an interval does not contain 0, the corresponding pair is statistically significantly different.

Here that is the case only for Drug Free - Throughout, so the other two pairs are not statistically significantly different. Remember, however that *failing to reject* H_0 is NOT the same as *accepting* H_0. The fact that those pairs are not statistically significantly different is almost certainly due to a lack of sample size.

4.1.10 Regression

4.1.10.1 Example: Predicting the Usage of Electricity

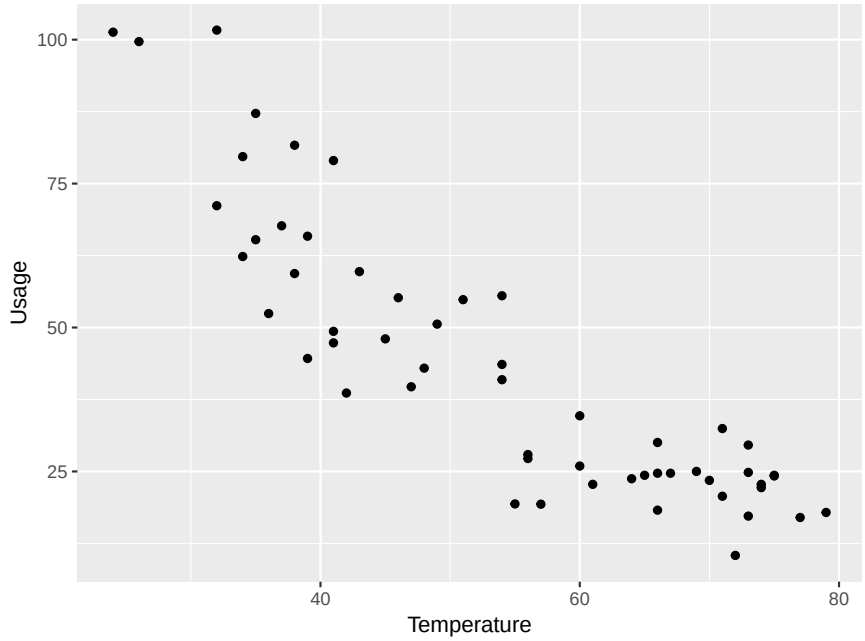
In Westchester County, north of New York City, Consolidated Edison bills residential customers for electricity on a monthly basis. The company wants to predict residential usage, in order to plan purchases of fuel and budget revenue flow. The data includes information on usage (in kilowatt-hours per day) and average monthly temperature for 55 consecutive months for an all-electric home. Data on consumption of electricity and the temperature in Westchester County, NY.

```
attach(elusage)
head(elusage)
```

```
##   Month Year Usage Temperature
## 1     8 1989 24.83          73
## 2     9 1989 24.69          67
## 3    10 1989 19.31          57
```

```
## 4    11 1989 59.71          43
## 5    12 1989 99.67          26
## 6     1 1990 49.33          41
```

```
ggplot(aes(Temperature, Usage), data=elusage) +
  geom_point()
```



We want to find a *model* (aka a function) f with

$$\text{Usage} = f(\text{Temperature}) + \epsilon$$

where ϵ is some random variable, often with the assumption $\epsilon \sim N(0, \sigma)$.

1. Linear Model

$$\text{Usage} = \beta_0 + \beta_1 \text{ Temperature} + \epsilon$$

```
fit <- lm(Usage~Temperature, data=elusage)
summary(fit)
```

```
##
## Call:
## lm(formula = Usage ~ Temperature, data = elusage)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -22.305  -8.163   0.559   7.723  28.611
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) 116.7162     5.5649    21.0   <2e-16
## Temperature  -1.3646     0.0994   -13.7   <2e-16
##
## Residual standard error: 11.4 on 53 degrees of freedom
## Multiple R-squared:  0.78,   Adjusted R-squared:  0.776
## F-statistic:  188 on 1 and 53 DF,  p-value: <2e-16
```

gives the model as

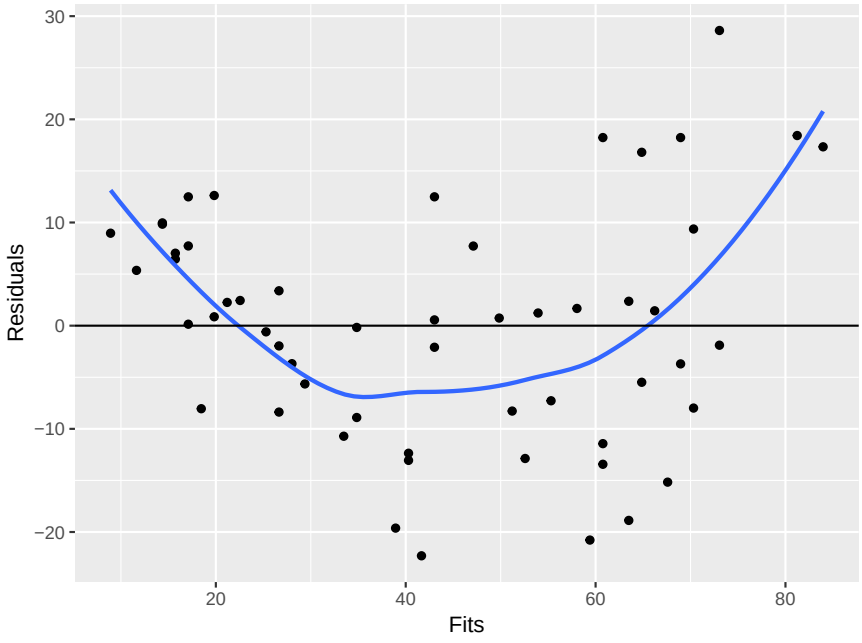
$$\text{Usage} = 116.7 - 1.36\ \text{Temperature} + \epsilon$$

How do we know that this is a good model? The main diagnostic is the *Residual vs Fits plot*:

```
Fits <- 116.7 - 1.36*Temperature
Residuals <- Usage-Temperature
```

actually, they are already part of the fit object:

```
ggplot(aes(Fits, Residuals),
       data=data.frame(Fits=fitted(fit),
                       Residuals=residuals(fit))) +
 geom_point() +
 geom_smooth(se=FALSE) +
 geom_abline(slope = 0)
```



the idea is that if the model is good, the residuals and the fitted values should be independent, and so this graph should not show any pattern. Adding the non-parametric fit and a horizontal line shows that this is not the case here.

- polynomial model

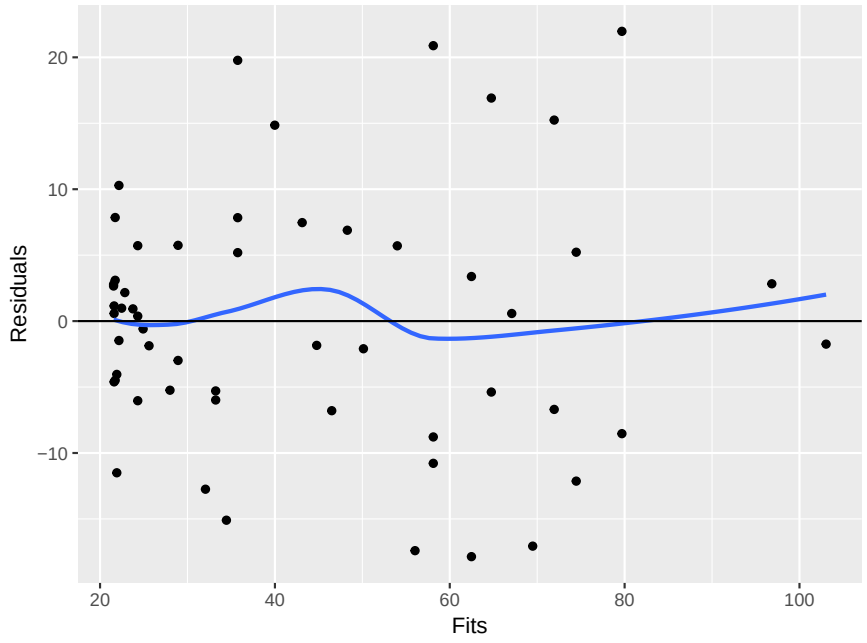
Let's add a quadratic term to the model

$$\text{Usage} = \beta_0 + \beta_1 \text{ Temperature} + \beta_2 \text{ Temperature}^2 + \epsilon$$

```
T2 <- Temperature^2
quad.fit <- lm(Usage~Temperature + T2,
               data=elusage)
summary(quad.fit)
```

```
##
## Call:
## lm(formula = Usage ~ Temperature + T2, data = elusage)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.859  -5.681   0.376   5.465  21.970
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 196.71532   17.48757   11.25  1.5e-15
## Temperature  -4.64046    0.69491   -6.68  1.6e-08
## T2            0.03073    0.00647    4.75  1.6e-05
##
## Residual standard error: 9.57 on 52 degrees of freedom
## Multiple R-squared:  0.847,  Adjusted R-squared:  0.841
## F-statistic:  144 on 2 and 52 DF,  p-value: <2e-16
```

```
ggplot(aes(Fits, Residuals),
       data=data.frame(Fits=fitted(quad.fit),
                       Residuals=residuals(quad.fit))) +
 geom_point() +
 geom_smooth(se=FALSE) +
 geom_abline(slope = 0)
```



and that is much better. If it were not one can continue and add even higher order terms. There will always be a polynomial model that give a good fit.

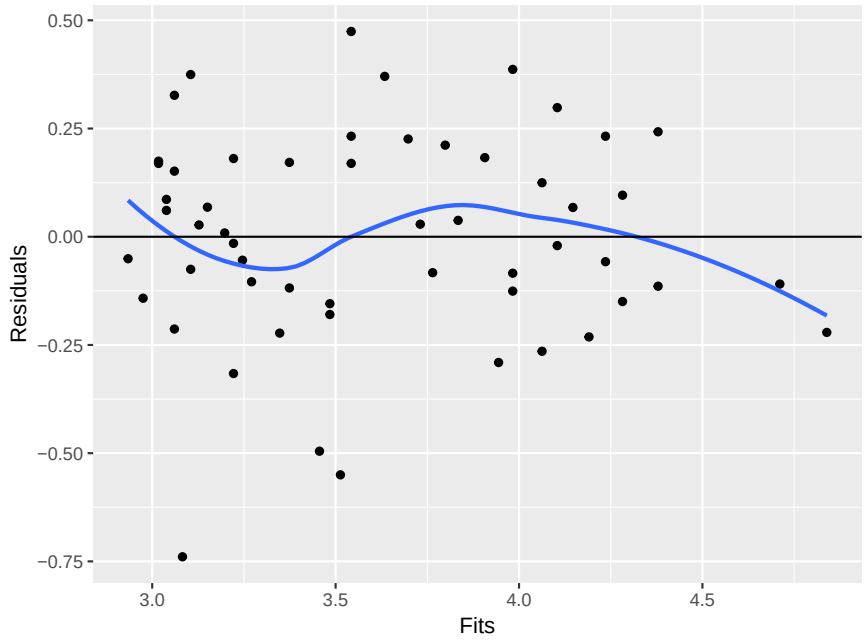
- Transformations

here we use some transformation (functions) of the data, for example log:

```
log.usage <- log(Usage)
log.temp <- log(Temperature)
log.fit <- lm(log.usage~log.temp)
summary(fit)
```

```
##
## Call:
## lm(formula = Usage ~ Temperature, data = elusage)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -22.305  -8.163   0.559   7.723  28.611
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 116.7162     5.5649    21.0   <2e-16
## Temperature  -1.3646     0.0994   -13.7   <2e-16
##
## Residual standard error: 11.4 on 53 degrees of freedom
## Multiple R-squared:  0.78,   Adjusted R-squared:  0.776
## F-statistic:  188 on 1 and 53 DF,  p-value: <2e-16
```

```
ggplot(aes(Fits, Residuals),
       data=data.frame(Fits=fitted(log.fit),
                       Residuals=residuals(log.fit))) +
 geom_point() +
 geom_smooth(se=FALSE) +
 geom_abline(slope = 0)
```



and that is also not to bad. Other standard choices for transformations are square root and inverse, but in principle any function might work.

How do we choose among these models? A standard measure of the quality of the fit is the **Coefficient of Determination**. It is defined as

$$R^2 = \text{cor}(\text{Observed Values, Predicted Values})^2 100\%$$

the better a model is, the more correlated it's fitted values and the observed values should be, so if we have a choice of two model, the one with the higher R^2 is better.

Here we find

```
# Linear Model
round(100*unlist(summary(fit)$r.squared), 2)
```

```
## [1] 78.05
```

```
# Quadratic Model
round(100*unlist(summary(quad.fit)$r.squared), 2)
```

```
## [1] 84.69
```

```
# Log Transfrom Model
round(100*unlist(summary(log.fit)$r.squared), 2)
```

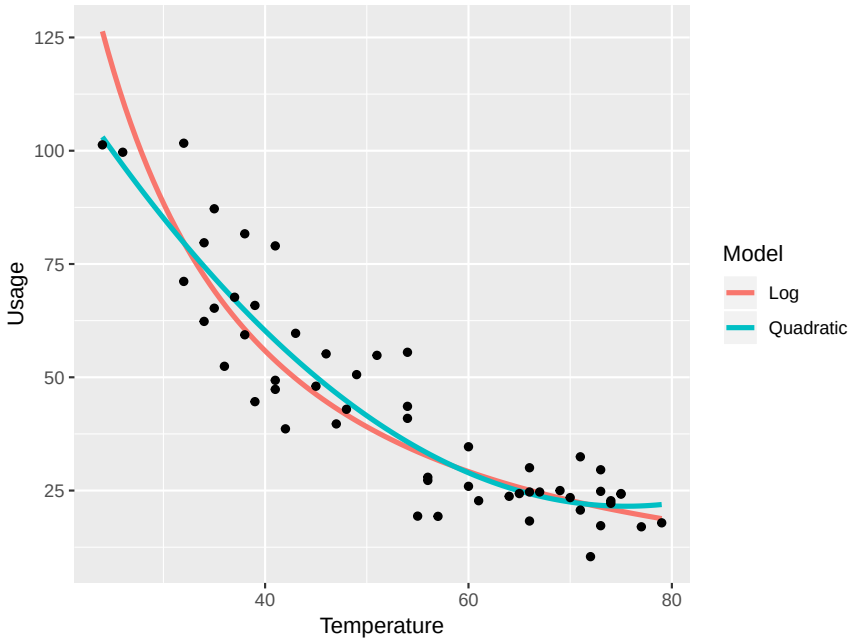
```
## [1] 81.12
```

but we need to be careful here: the linear model is a special case of the quadratic model, and so it's R^2 can never be smaller. There are other ways to choose between such *nested models*, for example the F test, but here this is not an issue because the linear model is bad anyway.

Now the R^2 of the quadratic model is 84.7% and that of the log transform model is 81.1%, so the quadratic one is better.

Finally let's have a look what those models look like:

```
x <- seq(min(Temperature), max(Temperature), length=100)
y.quad <- predict(quad.fit,
                  newdata=data.frame(Temperature=x,
                                     T2 = x^2))
y.log <- exp(predict(log.fit,
                 newdata=data.frame(log.temp=log(x))))
dta <- data.frame(x=c(x, x),
                  y=c(y.quad, y.log),
                  Model=rep(c("Quadratic", "Log"),
                            each=100))
ggplot(aes(x, y, color=Model), data=dta) +
  geom_line(size=1.2) +
  geom_point(data=elusage,
             aes(Temperature, Usage),
             inherit.aes = FALSE) +
  xlab("Temperature") +
  ylab("Usage")
```



4.2 Multiple Predictors

4.2.1 ANOVA

4.2.1.1 Case Study: Gasoline Type and Mileage

In an experiment to study gas mileage four different blends of gasoline are tested in each of three makes of automobiles. The cars are driven a fixed distance to determine the mpg (miles per gallon) The experiment is repeated three times for each blend-automobile combination. (Taken from Lyman Ott)

Note that the interest here is indifferent gasoline blends, automobile is a blocking variable, so this is a randomized block design.

Gasoline is numbers, but these are just codes for different blends, so it is a categorical variable or factor.

```
gasoline$Gasoline <- factor(gasoline$Gasoline,
                            levels = c(1, 3, 2, 4),
                            ordered = TRUE)
gasoline <- as.data.frame(gasoline)
```

```
attach(gasoline)
head(gasoline)
```

```
##    MPG Gasoline Automobile
## 1 22.7        1          A
## 2 22.4        1          A
## 3 22.9        1          A
## 4 21.5        2          A
```

```
## 5 21.8        2          A
## 6 21.6        2          A
```

Here is an interesting calculation:

```
table(Gasoline, Automobile)
```

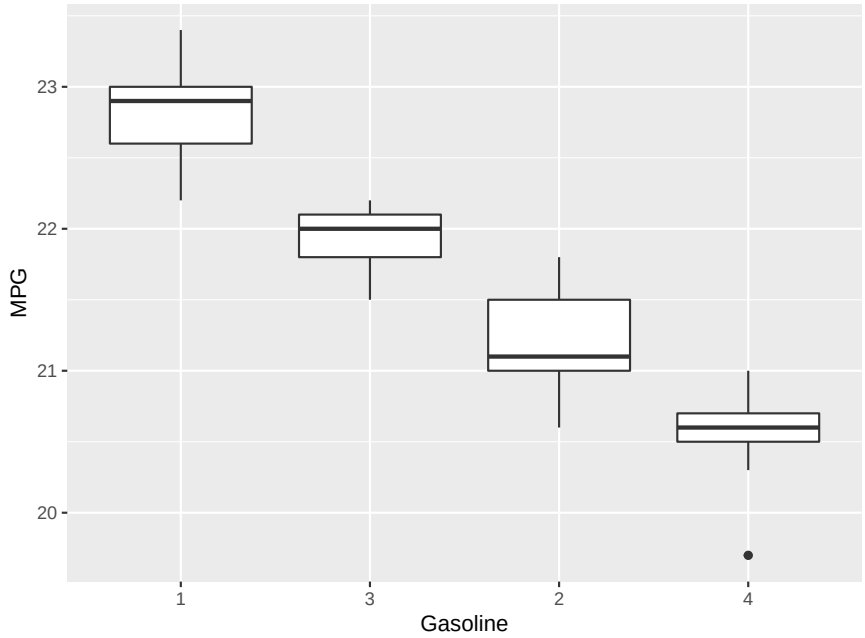
```
##          Automobile
## Gasoline A B C
##        1 3 3 3
##        3 3 3 3
##        2 3 3 3
##        4 3 3 3
```

This shows us two things:

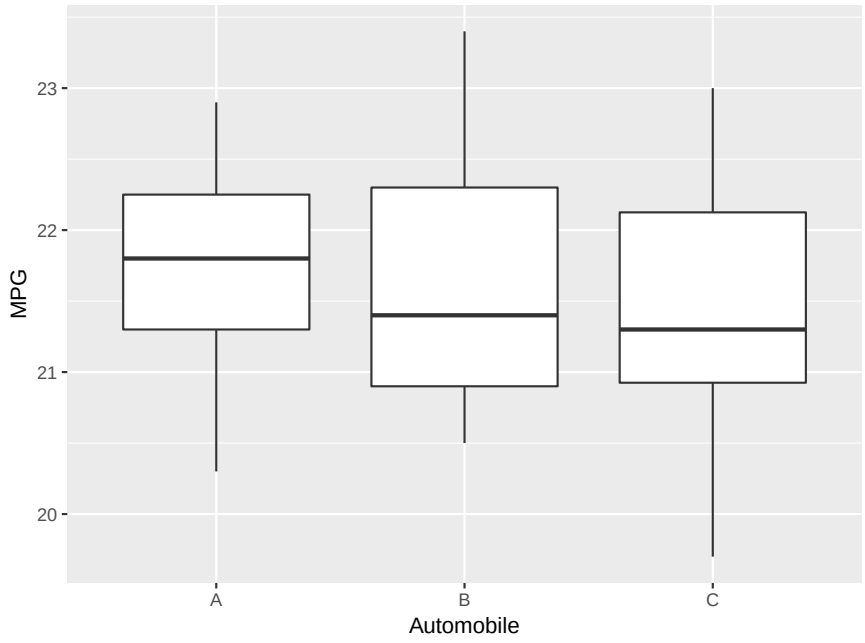
1. we have *repeated measurements* (several observations per factor-level combination)
2. we have a *balanced design* (the same number of repetitions in each factor-level combination)

This second feature used to be quite important because the calculations in a balanced design are much simpler. Nowadays with fast computers this is not important anymore. There are still good reasons why you want to design your experiment to have a balanced design if possible, though!

```
ggplot(gasoline, aes(Gasoline, MPG)) +
  geom_boxplot()
```




```
ggplot(gasoline, aes(Automobile, MPG)) +
  geom_boxplot()
```



the boxplots suggest a difference between blends but not between automobiles.

The summary statistics are

```
round(tapply(MPG, Gasoline, mean), 1)
```

```
##    1    3    2    4
## 22.8 21.9 21.2 20.5
```

```
round(tapply(MPG, Gasoline, sd), 2)
```

```
##    1    3    2    4
## 0.36 0.25 0.37 0.36
```

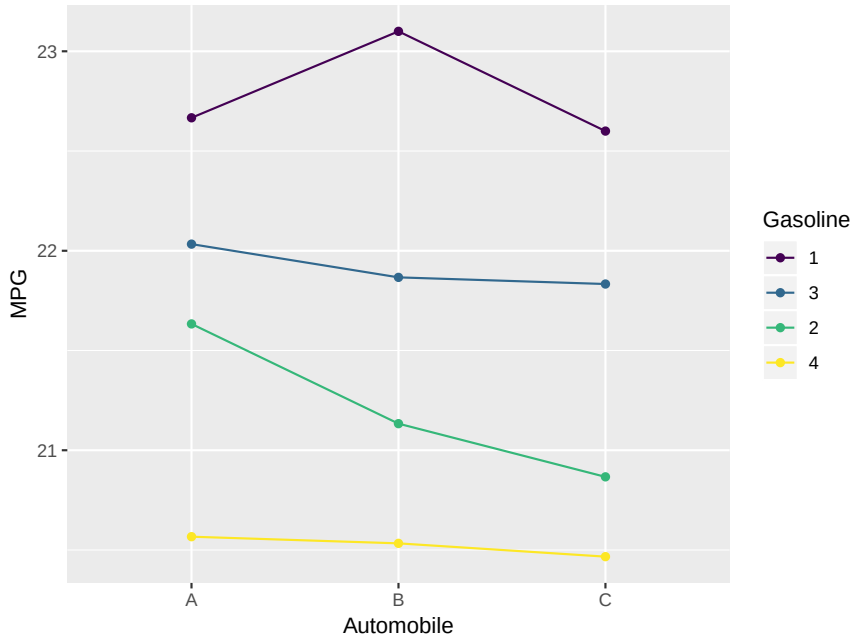
Interaction:

In the case of two or more predictors (factors) we also need to worry about relationships between them. This is called *interaction*.

One way to study this issue is as follows:

- find the means of MPG for each factor-level combination
- plot these points with one factor on the x axis
- connect the points according to the other factor

```
ggplot(data = gasoline,
       aes(Automobile, MPG,
           colour = Gasoline, group=Gasoline)) +
      stat_summary(fun.y=mean, geom="point")+
      stat_summary(fun.y=mean, geom="line")
```



if there is no interaction, the line segments should be fairly parallel. This is the case here, so there is no indication of interaction.

We have **repeated measurements** (3 per factor-level combination), so we can test also test for this:

```
aov(MPG~Gasoline*Automobile, data=gasoline)
```

```
## Call:
##    aov(formula = MPG ~ Gasoline * Automobile, data = gasoline)
##
## Terms:
##                 Gasoline Automobile Gasoline:Automobile Residuals
## Sum of Squares    25.405      0.527                0.909     2.247
## Deg. of Freedom        3          2                    6        24
##
## Residual standard error: 0.306
## Estimated effects may be unbalanced
```

1) Parameters of interest: Interaction

2) Method of analysis: ANOVA

3) Assumptions of Method: residuals have a normal distribution, groups have equal variance

4) Type I error probability α=0.05

5) Null hypothesis H_0 : no interaction

6) Alternative hypothesis H_a: some interaction

7) p value = 0.1854
8) 0.1854 > 0.05, there is no evidence of interaction.

So we will now proceed without the interaction term:

```
aov(MPG~Gasoline+Automobile, data=gasoline)
```

```
## Call:
##    aov(formula = MPG ~ Gasoline + Automobile, data = gasoline)
##
## Terms:
##                   Gasoline Automobile Residuals
## Sum of Squares    25.405      0.527     3.156
## Deg. of Freedom        3          2        30
##
## Residual standard error: 0.3243
## Estimated effects may be unbalanced
```

the assumptions are the same as those of oneway anova, they are fine here.

Now let's test for the factors:

Test for Factor Gasoline:

1) Parameters of interest: means of gasoline groups

2) Method of analysis: ANOVA

3) Assumptions of Method: residuals have a normal distribution, groups have equal variance

4) Type I error probability $\alpha = 0.05$

5) Null hypothesis $H_0 : \mu_1 = .. = \mu_4$ (Gasoline groups have the same means)

6) Alternative hypothesis H_a: $\mu_i \neq \mu_j$ (Gasoline groups have different means)

7) p value=0.000

8) 0.000<0.05, there is some evidence of differences in gasoline blends

Test for Factor Automobile is not really needed because this is a *blocking variable*.

Notice that if we included the interaction the p-value for Automobile was 0.08, without the interaction it is 0.1. One advantage of being able to fit an additive model is that often it makes the conclusions stronger.

4.2.2 Regression

4.2.2.1 Case Study: House Prices

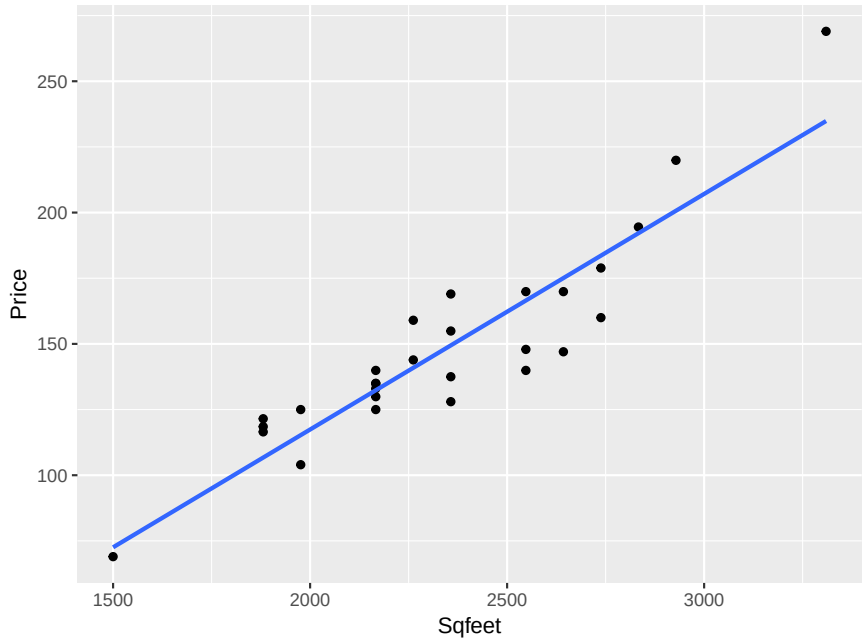
Prices of residencies located 30 miles south of a large metropolitan area with several possible predictor variables.

```
attach(houseprice)
houseprice
```

```
##    Price Sqfeet Floors Bedrooms Baths
## 1   69.0   1500      1        2   1.0
## 3  118.5   1881      1        2   2.0
## 4  104.0   1976      1        3   2.0
## 5  116.5   1881      1        3   2.0
## 6  121.5   1881      1        3   2.0
## 7  125.0   1976      1        3   2.0
## 8  128.0   2357      2        3   2.5
## 9  129.9   2167      1        3   1.7
## 10 133.0   2167      2        3   2.5
## 11 135.0   2167      2        3   2.5
## 12 137.5   2357      2        3   2.5
## 13 139.9   2167      1        3   2.0
## 14 143.9   2262      2        3   2.5
## 15 147.9   2548      2        3   2.5
## 16 154.9   2357      2        3   2.5
## 17 160.0   2738      2        3   2.0
## 18 169.0   2357      1        3   2.0
## 19 169.9   2643      1        3   2.0
## 20 125.0   2167      1        4   2.0
## 21 134.9   2167      1        4   2.0
## 22 139.9   2548      1        4   2.0
## 23 147.0   2643      1        4   2.0
## 24 159.0   2262      1        4   2.0
## 25 169.9   2548      2        4   3.0
## 26 178.9   2738      1        4   2.0
## 27 194.5   2833      2        4   3.0
## 28 219.9   2929      1        4   2.5
## 29 269.0   3310      2        4   3.0
```

Let's go through the list of predictors one by one:

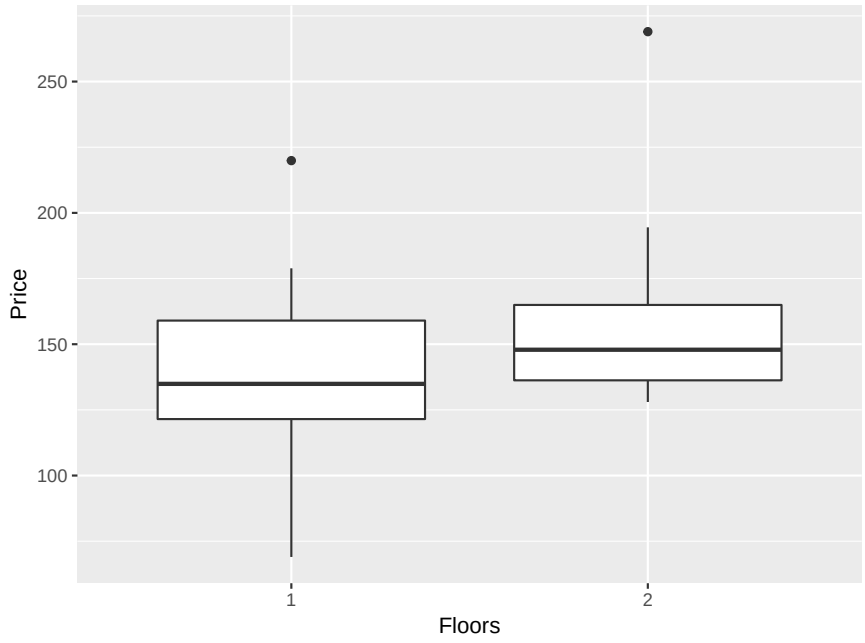
```
ggplot(houseprice, aes(Sqfeet, Price)) +
  geom_point() +
  geom_smooth(method = "lm", se=FALSE)
```




```
cor(Price, Sqfeet)
```

```
## [1] 0.9152
```

```
ggplot(houseprice, aes(factor(Floors), Price)) +
  geom_boxplot() +
  labs(x="Floors")
```

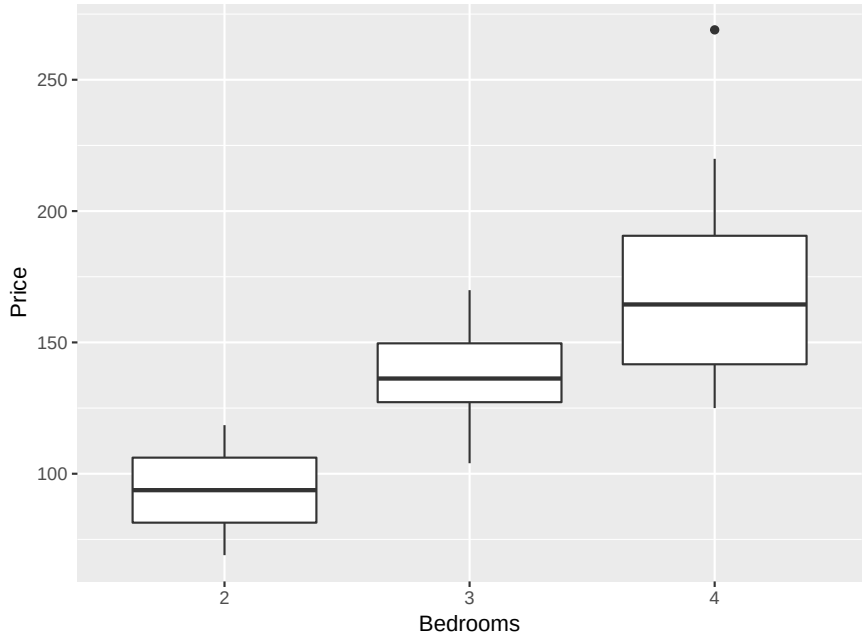



```
cor(Price, Floors)
```

```
## [1] 0.2911
```

Note we used the boxplot here although Floors is a quantitative predictor. If the predictor has only a few different values (2 here!) this is often a better choice.

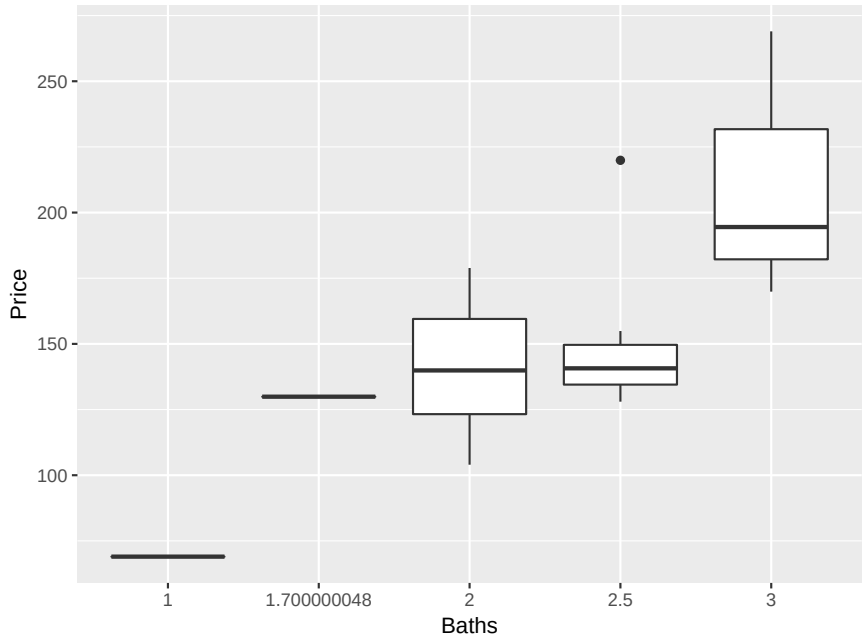
```
ggplot(houseprice, aes(factor(Bedrooms), Price)) +
  geom_boxplot() +
  labs(x="Bedrooms")
```




```
cor(Price, Bedrooms)
```

```
## [1] 0.6045
```

```
ggplot(houseprice, aes(factor(Baths), Price)) +
  geom_boxplot() +
 labs(x="Baths")
```




```
cor(Price, Baths)
```

```
## [1] 0.6526
```

Now to run the regression:

```
fit <- lm(Price~., data=houseprice)
summary(fit)
```

```
##
## Call:
## lm(formula = Price ~ ., data = houseprice)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -23.02  -5.94   1.86   5.95  30.95
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -67.6198    17.7082   -3.82  0.00088
## Sqfeet        0.0857     0.0108    7.97  4.6e-08
## Floors      -26.4931     9.4895   -2.79  0.01036
## Bedrooms     -9.2862     6.8298   -1.36  0.18712
## Baths        37.3807    12.2644    3.05  0.00571
##
## Residual standard error: 13.7 on 23 degrees of freedom
## Multiple R-squared:  0.886,  Adjusted R-squared:  0.866
## F-statistic: 44.8 on 4 and 23 DF,  p-value: 1.56e-10
```

Notice that there is something very strange about this model!

```
round(cor(houseprice[,-1]), 3)
```

```
##          Sqfeet Floors Bedrooms Baths
## Sqfeet    1.000  0.370    0.652 0.628
## Floors    0.370  1.000   -0.018 0.743
## Bedrooms  0.652 -0.018    1.000 0.415
## Baths     0.628  0.743    0.415 1.000
```

The highest correlation between predictors is r=0.743 (Floors-Baths)

4.2.3 Variable Selection

Generally the fewer predictor we have in a model, the easier it is to understand/use, so

can we eliminate any of our predictors without making the model (stat. signif.) worse?

There are several things one can think of:

Choose based on R^2

but we already know this will always lead to the model with all predictors, for the same reason that a cubic model always has an R^2 at least as high as the quadratic model.

Note:

Price by Sqfeet, Floors and Bedrooms: R^2=80.1%
Price by Floors, Bedrooms and Baths: R^2=68.4%
Price by Sqfeet, Bedrooms and Baths: R^2=83.5%
Price by Sqfeet, Floors, Bedrooms and Baths: R^2=88.2%

so the model with all 4 has a higher R^2 than any of the models with just 3,

but this will always be so, even if one of the predictors is completely useless.

Choose based on Hypothesis Tests

In the output of the fit object above we see that the p_value of Bedrooms = 0.187121 > 0.05, so eliminate Bedrooms.

This sounds like a good idea AND IT IS WIDELY USED IN REAL LIFE, but it turns out to be a bad one ! The reason why is bit hard to explain, though.

What we need is new idea:

Best Subset Regression and Mallow's C_p

We will find ALL possible models and calculate Mallow's C_p statistic for each. The model with the lowest C_p is best.

```
library(leaps)
leaps(houseprice[, -1], Price, method="Cp", nbest=1)
```

```
## $which
##      1     2     3     4
```

```
## 1 TRUE FALSE FALSE FALSE
## 2 TRUE FALSE FALSE  TRUE
## 3 TRUE  TRUE FALSE  TRUE
## 4 TRUE  TRUE  TRUE  TRUE
##
## $label
## [1] "(Intercept)" "1"           "2"           "3"           "4"
##
## $size
## [1] 2 3 4 5
##
## $Cp
## [1] 8.834 8.812 4.849 5.000
```

so the best model has Cp=4.85 and uses Sqfeet, Floors and Baths.

To find the model we rerun lm, now without Bedrooms:

```
fit <- lm(Price~. - Bedrooms, data=houseprice)
summary(fit)
```

```
##
## Call:
## lm(formula = Price ~ . - Bedrooms, data = houseprice)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -25.94  -7.11   2.12   5.66  33.35
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -73.92954   17.38880   -4.25  0.00028
## Sqfeet        0.07772    0.00917    8.48  1.1e-08
## Floors      -19.67975    8.19979   -2.40  0.02451
## Baths        30.57922   11.39408    2.68  0.01298
##
## Residual standard error: 13.9 on 24 degrees of freedom
## Multiple R-squared:  0.877,  Adjusted R-squared:  0.862
## F-statistic: 57.1 on 3 and 24 DF,  p-value: 4.5e-11
```

4.3 Parameter Estimation

4.3.1 Maximum Likelihood

In this chapter we will study the problem of parameter estimation. In its most general form this is as follows: we have a sample $X_1, .., X_n$ from some probability density $f(x; \theta)$. Here both x and θ 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 θ based on the data $X_1, .., X_n$, that is a function (called a *statistic*) $T(X_1, .., X_n)$ such that in some sense $T(X_1, .., X_n) \approx \theta$.

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

$$f(x_1, .., x_n; \theta) = \prod_{i=1}^{n} f(x_i, \theta)$$

and the log-likelihood function is defined by

$$l(\theta) = \sum_{i=1}^{n} \log f(x_i, \theta)$$

the estimate of θ is then found by maximizing the function l. Let's call this $\hat{\theta}$.

4.3.2 Large Sample Theory of MLEs and Confidence Intervals

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) θ_i as

$$\hat{\theta} \pm z_{\alpha/2}\sqrt{I_{ii}^{-1}}$$

where z_α is the $(1 - \alpha)100\%$ quantile of the standard normal distribution. In R this is found with

```
qnorm(1-0.05/2)
```

```
## [1] 1.96
```

if $\alpha = 0.05$.

4.3.2.1 Example: Inference for mean of normal distribution

$X_1, .., X_n \sim N(\mu, \sigma)$, σ known.

$$
\begin{aligned}
&f(x;\mu) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \\
&l(\mu) = \sum \log f(x_i, \mu) = \\
&n\log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}\sum(x_i-\mu)^2 \\
&\frac{d}{d\mu} l(\mu) = \frac{1}{\sigma^2}\sum(x_i-\mu) = 0 \\
&\hat{\mu} = \frac{1}{n}\sum x_i
\end{aligned}
$$

Here we have only one parameter (μ), 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 σ).

4.3.2.2 Example: Beta distribution

Say we have $X_1, .., X_n \sim B(\alpha, \alpha)$.

Now

$$f(x;\alpha) = \frac{\Gamma(2\alpha)}{\Gamma(\alpha)^2}[x(1-x)]^{\alpha-1}$$

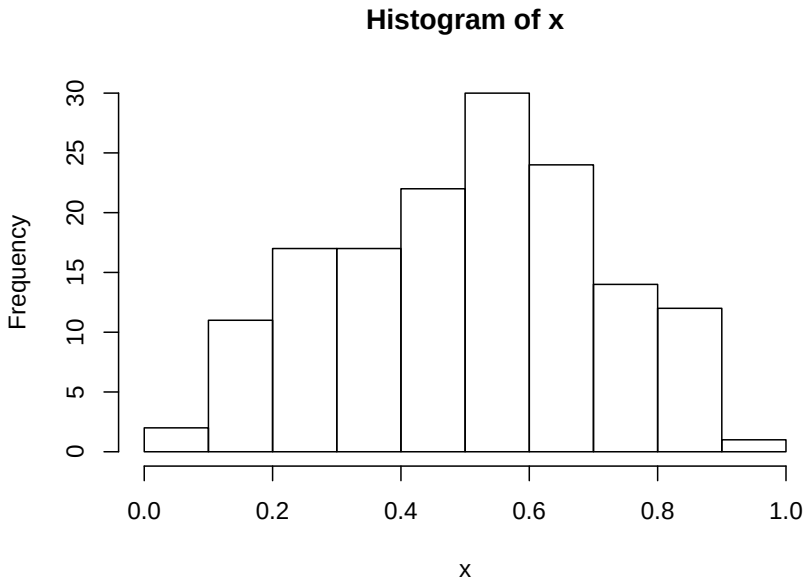
where $\Gamma(x)$ is the gamma function, defined by

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$$

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:

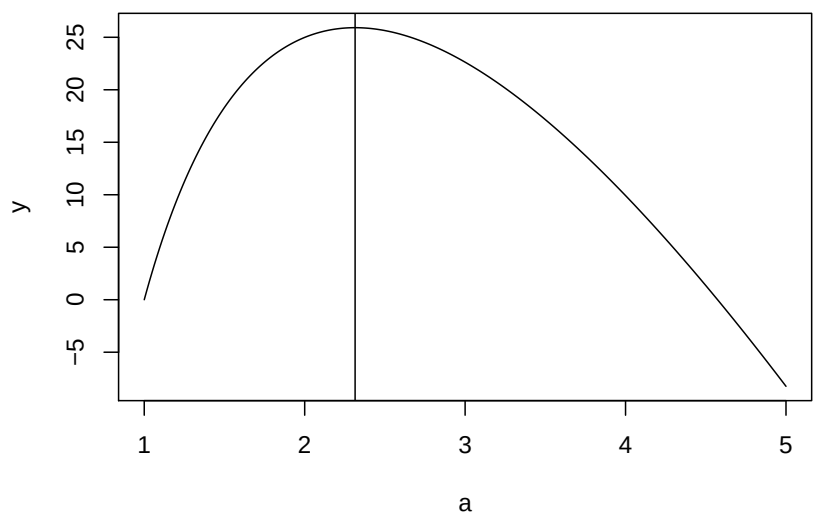
```
set.seed(111)
n <- 150
x <- rbeta(n, 2.5, 2.5)
hist(x, 10)
```



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] 2.314
```

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 $\frac{1}{n}\sum H \to I$. 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)
```

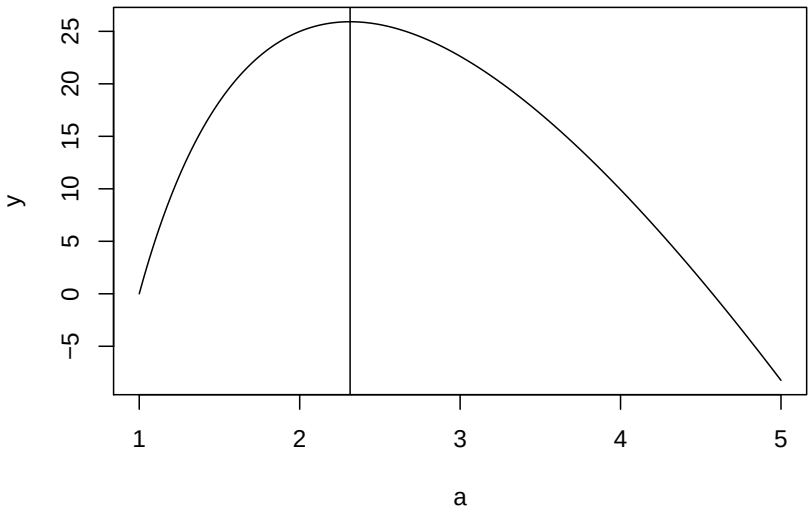
```
## [1] 1.838 2.791
```

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)
```



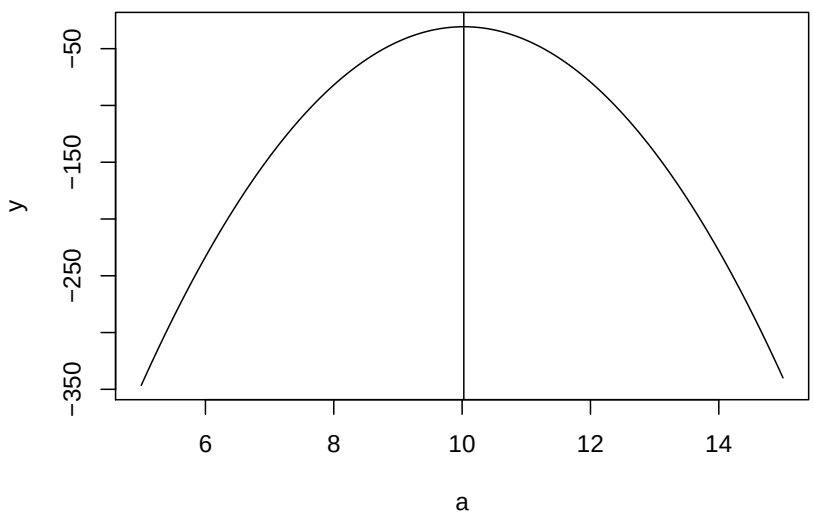

```
## $mle
## [1] 2.314
##
## $ci
## Lower Upper
## 1.838 2.791
```

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] 10.027  9.635 10.419
```

```
mle.est(f = function(a) {dnorm(x, a)},
        param = 10,
        rg = c(5, 15),
        do.graph = TRUE)
```




```
## $mle
## [1] 10.03
##
## $ci
##  Lower  Upper
##  9.635 10.419
```

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] 5.587 1.676
##
## $ci
##       Lower Upper
## mu    5.355 5.819
## sigma 1.512 1.840
```

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)
```

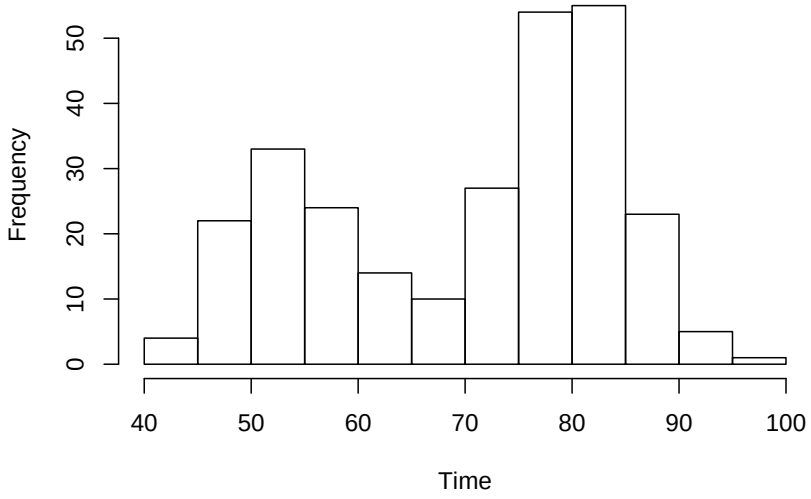
```
## $mle
## [1] 2.349 3.731
```

```
##
## $ci
##       Lower Upper
## alpha 1.913 2.785
## beta  3.011 4.451
```

4.3.2.3 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] 54.05 80.05
```

```
c(sd(Time[Time<65]), sd(Time[Time>65]))
```

```
## [1] 5.365 5.867
```

```
sum(Time<65)/length(Time)
```

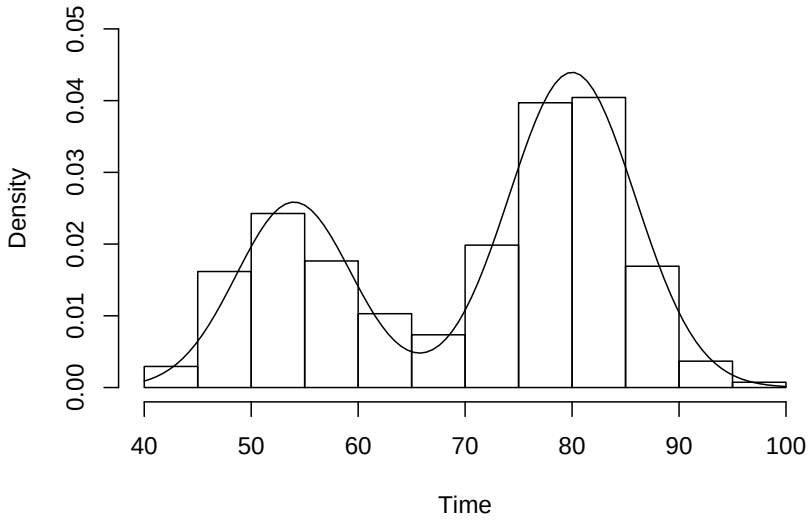
```
## [1] 0.3456
```

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 to bad!

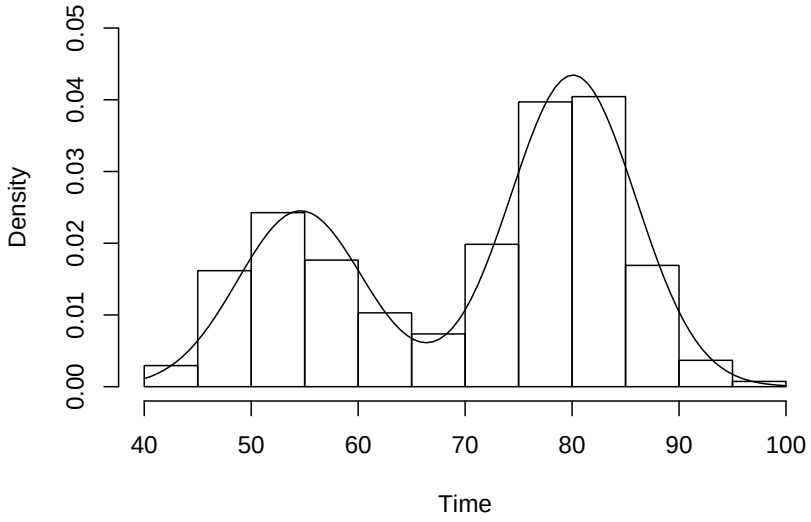
Can we do better? How about fitting for the parameters?

```
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]  0.3609 54.6149  5.8712 80.0911  5.8677
##
## $ci
##        Lower  Upper
## [1,]  0.2998  0.422
## [2,] 53.2433 55.986
## [3,]  4.8176  6.925
## [4,] 79.1024 81.080
## [5,]  5.0819  6.654
```

and this looks like

```r
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.

4.3.3 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 where 1582 deaths. During the period September 20 to October 31 there where 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 cause 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 π. Apparently before the storm we had $\pi = 83.5/3500000$. If we denote the probability to die due to Maria by μ, 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] 9.862
##
## $estimate
## [1] 2.379e-05 5.842e-06
##
## $gradient
## [1] 0 0
##
## $hessian
##      [,1] [,2]
## [1,] -Inf -Inf
## [2,] -Inf -Inf
##
## $code
## [1] 1
##
## $iterations
## [1] 1
```

Oops, 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] 9.707
##
## $estimate
## [1] 83.26 19.57
##
## $gradient
## [1] -3.840e-12 -7.261e-12
##
## $hessian
##        [,1]   [,2]
## [1,] 0.6365 0.4084
## [2,] 0.4084 0.4084
```

```
##
## $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:

$$\log\{\text{dpois}(x,\lambda)\} =$$
$$\log\left\{\frac{\lambda^x}{x!}e^{-\lambda}\right\} =$$
$$x\log(\lambda) - \log(x!) - \lambda =$$

```
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:

Total de Defunciones por Mes

Mes	2015	2016	2017	2018
Jan	2744	2742	2894	2821
Feb	2403	2592	2315	2448
Mar	2427	2458	2494	2643
Apr	2259	2241	2392	2218
May	2340	2312	2390	1892
Jun	2145	2355	2369	0
Jul	2382	2456	2367	0
Aug	2272	2427	2321	0
Sep	2258	2367	2928	0
Oct	2393	2357	3040	0
Nov	2268	2484	2671	0
Dec	2516	2854	2820	0
Total	28407	29645	31001	12022

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] 79.04
```

```
deaths.after/72 # Daily Deaths after Maria
```

```
## [1] 97.61
```

```
round(72*(deaths.after/72 - deaths.before/231)) # point estimate for total deaths due t
```

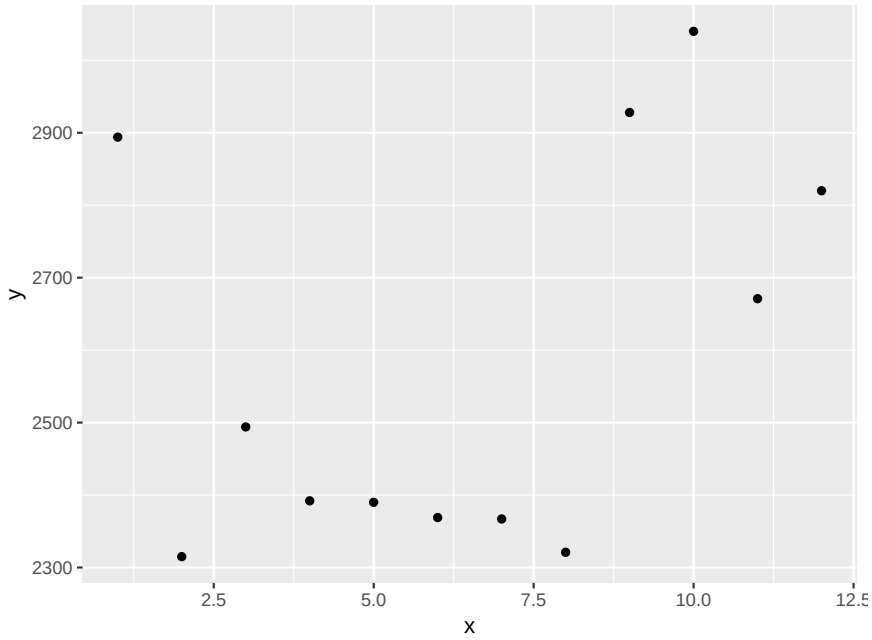
```
## [1] 1337
```

```
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()
```



4.3.4 Packages for estimation

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 (2 free parameter(s))
## Estimate(s): 83.18 19.65
```

In general these just provide wrappers for the routines mentioned above.

4.4 Bayesian Statistics

4.4.1 Prior and Posterior Distribution

Say we have a sample $\mathbf{X} = (X_1, .., X_n)$, iid from some probability density $f(.|\theta)$, and we want to do some inference on the parameter θ.

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

$$m(\mathbf{x}) = \int .. \int f(\mathbf{x}|\theta)\pi(\theta)d\theta$$

4.4.1.1 Example: Normal mean, normal prior

$X \sim N(\mu, \sigma)$ independent, σ known, $\mu \sim N(a, b)$.

Now

$$m(x) = \int_{-\infty}^{\infty} f(x|\mu)\pi(\mu)d\mu =$$
$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \frac{1}{\sqrt{2\pi b^2}} e^{-\frac{1}{2b^2}(\mu-a)^2} d\mu$$

Note

$$(x-\mu)^2/\sigma^2 + (\mu-a)^2/b^2 =$$
$$x^2/\sigma^2 - 2x\mu/\sigma^2 + \mu^2/\sigma^2 + \mu^2/b^2 - 2a\mu/b^2 + a^2/b^2$$
$$(1/\sigma^2 + 1/b^2)\mu^2 - 2(x/\sigma^2 + a/b^2)\mu + K_1 =$$
$$(1/\sigma^2 + 1/b^2)\left(\mu^2 - 2\frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}\mu\right) + K_2 =$$
$$\frac{(\mu-d)^2}{c^2} + K_3$$

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.

$$f(\theta|\mathbf{x}) = f(\mathbf{x}|\theta)\pi(\theta)/m(\mathbf{x}) =$$
$$K_6 e^{-\frac{1}{2c^2}(\mu-d)^2}$$

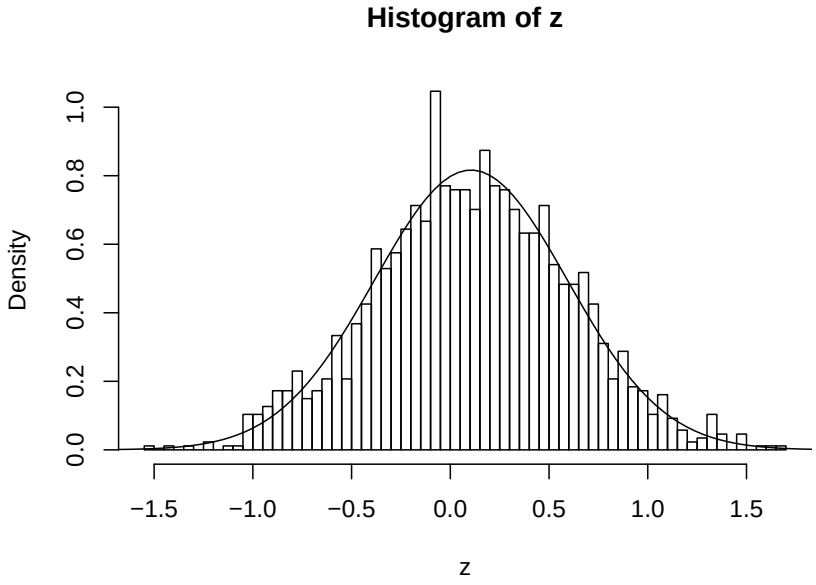
Notice that we don't need to worry about what exactly K_6 is, because the posterior will be a proper probability density, so K_6 will be what it has to be!

So we found

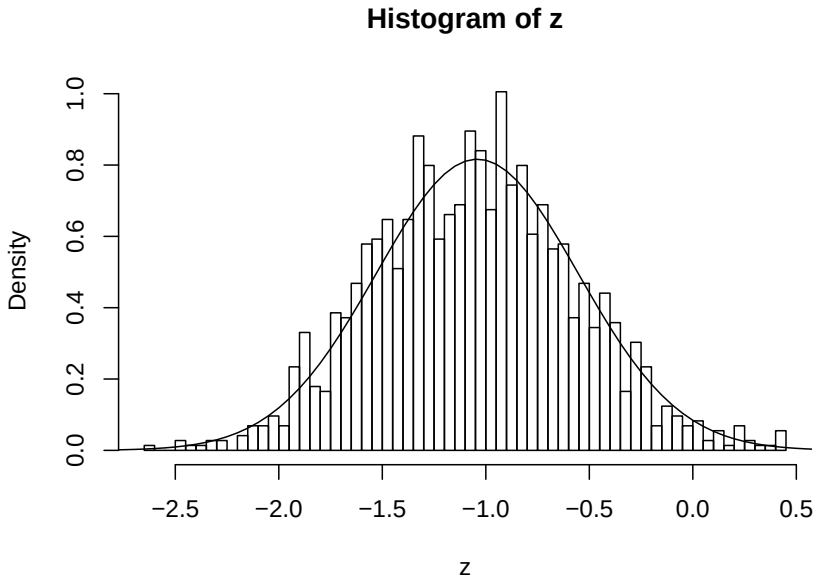
$$\mu|X = x \sim N\left(\frac{x/\sigma^2 + a/b^2}{1/\sigma^2 + 1/b^2}, 1/\sqrt{1/\sigma^2 + 1/b^2}\right)$$

Let's so 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)
```



4.4.1.2 Example: Binomial proportion, Uniform prior

$X \sim Bin(n, p)$, $p \sim U[0, 1]$

$$m(x) = \int_{-\infty}^{\infty} f(x|\mu)\pi(\mu)d\mu =$$
$$\int_0^1 \binom{n}{x} p^x (1-p)^{n-x} 1 dp =$$
$$K_1 \int_0^1 p^{(x+1)-1}(1-p)^{(n-x+1)-1} dp = K_2$$

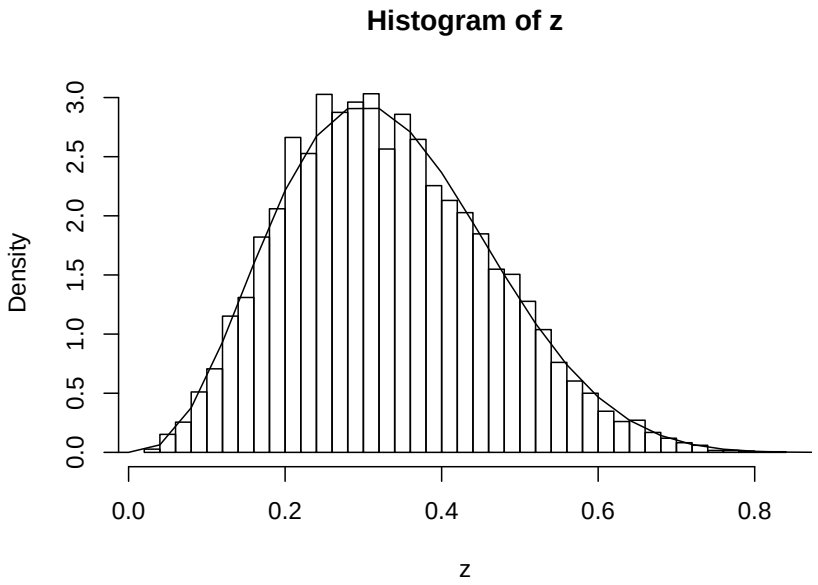
because this is (up to a constant) a Beta density which will integrate to 1. So

$$f(\theta|\mathbf{x}) = f(\mathbf{x}|\theta)\pi(\theta)/m(\mathbf{x}) =$$
$$K_3 p^{(x+1)-1}(1-p)^{(n-x+1)-1}$$

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)
```



4.4.2 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:

4.4.2.1 Example: Normal mean, normal prior

say the following is a sample $X_1, .., X_n$ from a normal with standard deviation $\sigma = 2.3$:

```
dta.norm
```

```
##  [1] 2.2 2.3 2.5 5.0 5.0 5.0 5.0 5.1 5.3 5.8 6.0 6.1 6.1 6.1 6.2 6.5 6.6
## [18] 7.1 9.0 9.6
```

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] 5.607
```

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] 0.5069
```

```
round(qnorm(c(0.025, 0.975), d, cc), 2)
```

```
## [1] 4.61 6.60
```

Note: the standard frequentist solution would be

```
round(mean(dta.norm), 2)
```

```
## [1] 5.62
```

```
round(mean(dta.norm)+c(-1, 1)*qt(0.975, 12)*2.3/sqrt(20), 2)
```

```
## [1] 4.50 6.75
```

4.4.2.2 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
```

4.4.2.3 Example: Normal mean, Gamma prior

let's say that μ 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 μ 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] 5.55
```

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] 4.56 6.54
```

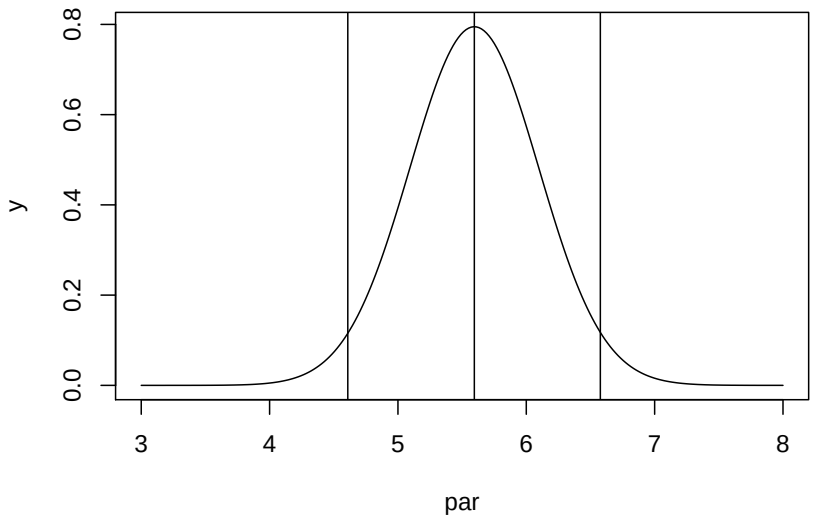
Let's generalize all this and write a routine that will find a point estimate and a $(1-\alpha)100\%$ credible interval for any problem with one parameter:

```
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)
}
```

4.4.2.4 Example: Normal mean, normal prior

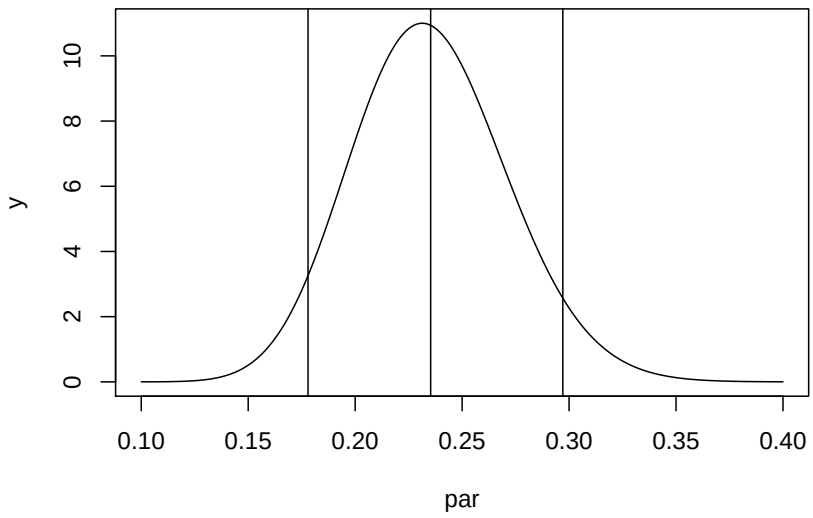
```
df <- function(x, par) dnorm(x, par, 2.3/sqrt(20))
prior <- function(par) dnorm(par, 5, 2.3)
round(bayes.credint(mean(dta.norm), df=df, prior=prior,
      lower=3, upper=8, Show=T), 2)
```




```
## [1] 5.60 4.61 6.58
```

4.4.2.5 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
```

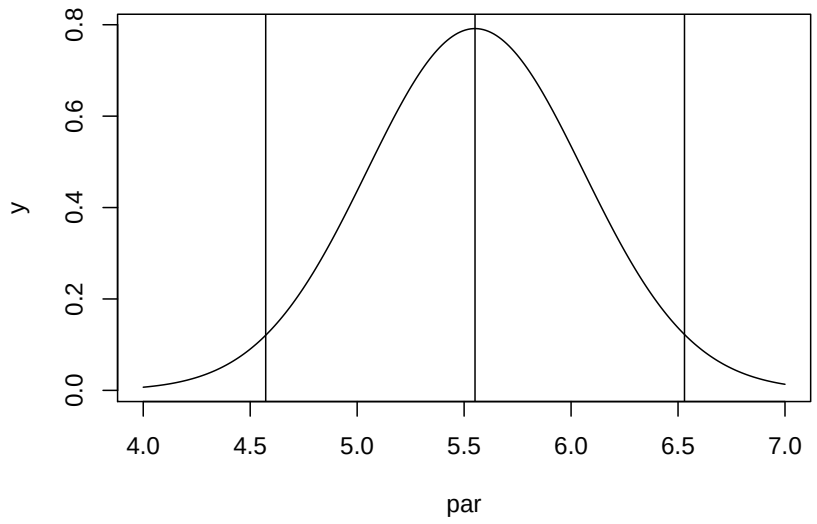
which is the same before:

```
round(qbeta(c(0.05, 0.95), 31 + 1, 134 - 31 + 1)*100, 1)
```

```
## [1] 17.8 29.7
```

4.4.2.6 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)
```



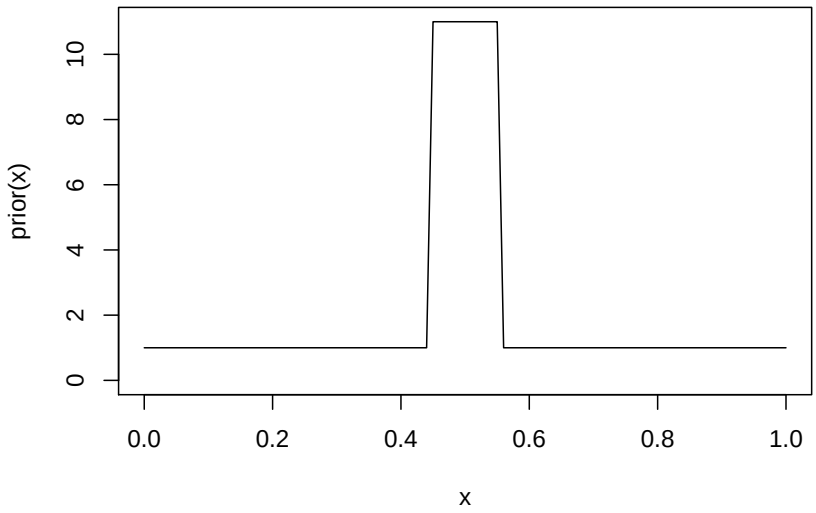

```
## [1] 5.55 4.57 6.53
```

4.4.2.7 Example: Binomial proportion, Lincoln's hat prior

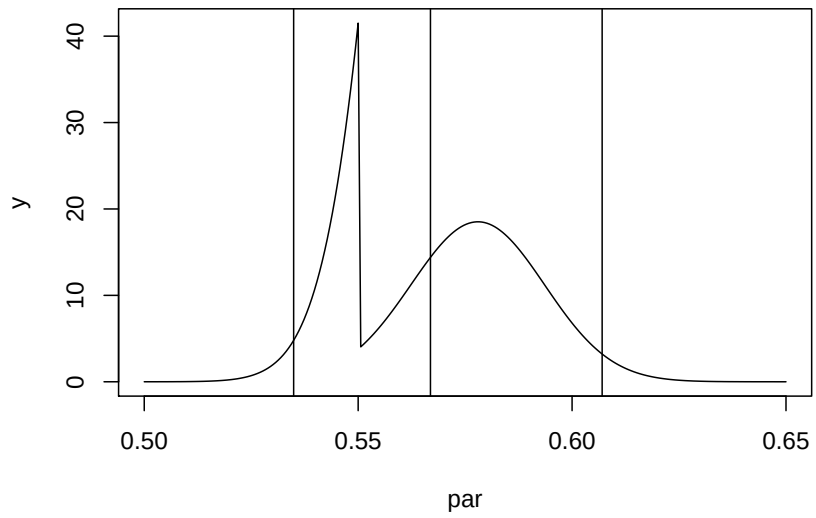
Say we pick a coin from our pocket. We flip it 1000 time and get 578 heads. We want to find a 95% credible interval for the proportion of heads.

What would be good prior here? We might reason as follows: on the one had we are quite sure that indeed 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)
```



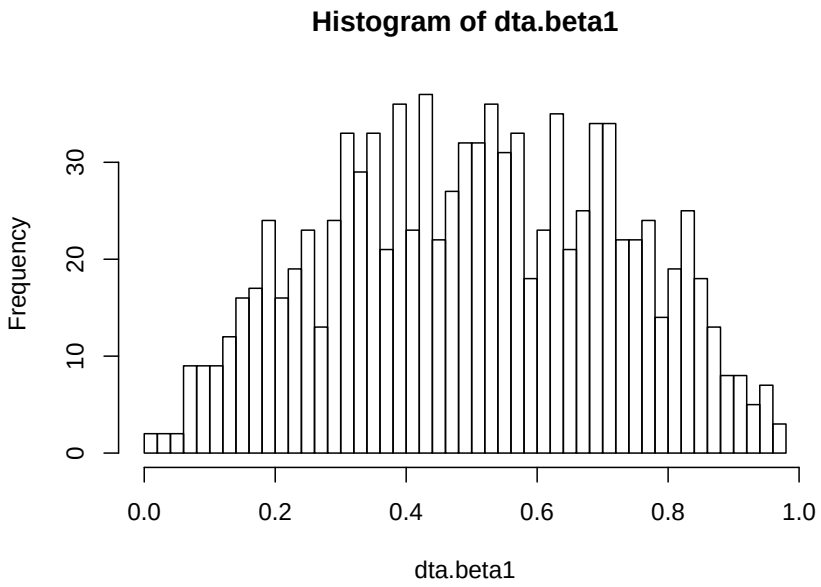

```
## [1] 0.567 0.535 0.607
```

So, have we just solved the Bayesian estimation problem for one parameter?

4.4.2.8 Example: Beta density, Gamma prior

consider the following sample:

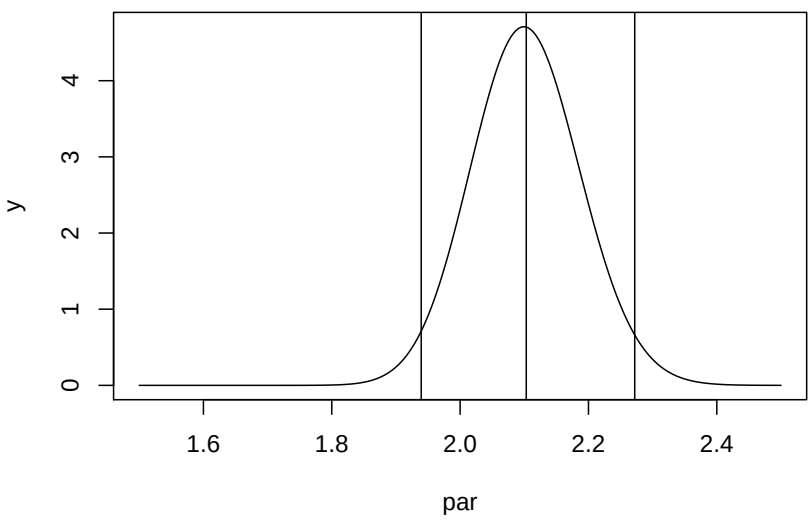
```
dta.beta1 <- round(rbeta(1000, 2, 2), 3)
hist(dta.beta1, 50)
```



Let's say we know that this is from a Beta(a, a) distribution and we want to estimate a. As a prior we want to use 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] 2.10 1.94 2.27
```

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)
```

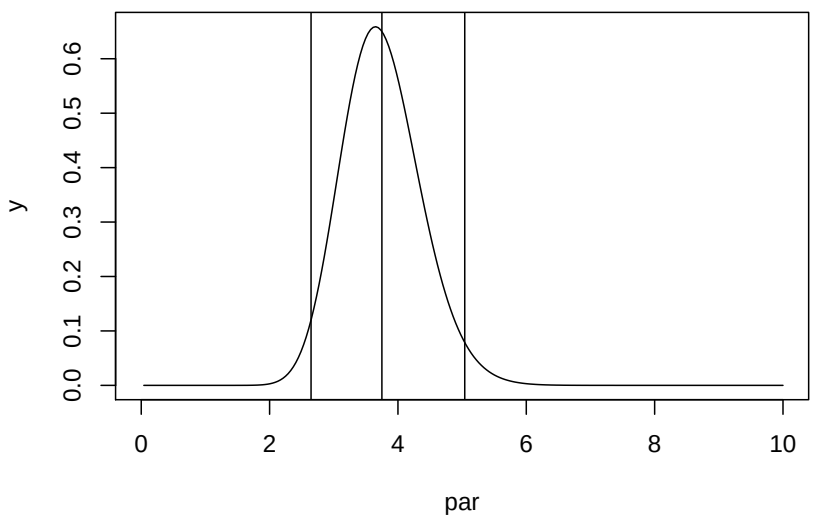
```
## Error in integrate(posterior.density, lower = lower, upper = upper, x = x): non-finit
```

Why does this not work? The problem is that the values is $\prod_{i=1}^{n} \text{dbeta}(x_i, a, a)$ 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 practice 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 completely 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] 3.750 2.646 5.041
```