Introduction to R and Basics

About Intro to R Programming

What can you expect?

Content The aim of this course is, broadly speaking, to teach you how to use R for data management, data visualisation, and for implementing statistical methods. In other words, this course is about statistical programming. Using R as an example we will study many generic concepts used in programming, so you will also learn about programming in general. This should help you learn other programming languages (like C, C++, Java, ...) more easily.

Furthermore it aims to give you a basic understanding of the concepts underlying statistical computing.

This course does *not* focus on the statistical methods already implemented in R (like `lm` or `glm`) — these functions will be covered in the *Data Analysis* course next semester.

The only way of learning how to program is by getting "hands-on experience". Thus most of the lectures are nothing other than labs where we look at simple examples which illustrate the different aspects of statistical programming in R.

Format The course consists of 10 self-guided lectures available online and 10 labs (see Moodle page for details). Each week there will be a pdf file on Moodle with a lesson you should go through on your own, which includes videos explaining the main concepts you will need to know to successfully perform the tasks in the labs, notes and some exercises. The videos were originally designed for an online module taught by the school and therefore may sometimes refer to modules of the online program or concepts we will not actually cover within this module. However, this will be apparent from the video and in some cases there will be comments in the notes underlying this. Notice that the videos are included as an additional resource to support your learning!

Website Class announcements, handouts, and R code will be available on Moodle

`http://moodle2.gla.ac.uk/course/view.php?id=3431`

Software We will use GNU R, which you can obtain for free from

`http://cran.r-project.org/`

We will use RStudio Desktop, an integrated development environment (IDE), which makes working with R a bit more comfortable. RStudio is also free software and can be obtained from

`http://rstudio.org/download/desktop`

What do we expect from you?

Attendance Attendance to labs is strongly encouraged. Programming is not always easy, but by attending the labs, as well as going through the online lectures, you will find it easy to keep up with the course.

Assessment The course is assessed by continuous assessment only.

- There will be two take-home assignment sheets (contributing 7.5% each to the overall grade).
- There will be two in-lab assessments ("class-tests"). One in week 6, the other in week 10.

See the Moodle page for provisional due dates for the assignments and the dates of the class tests.

References

You might find the following books helpful:

- Dalgaard, P. (2002). Introductory Statistics with R. Springer.
- Braun, W. J. and Murdoch, D. J. (2007). A First Course in Statistical Programming with R. Cambridge University Press.
- Venables, W. and Ripley, B. D. (2002). Modern Applied Statistics with S. Fourth Edition, Springer.
- Venables, W. and Ripley, B. D. (2001). S Programming. Springer.
- Chambers, J. M. (2008) Software for Data Analysis. Programming in R. Springer.
- Murrell, P. (2005). R Graphics. Chapman & Hall/CRC.

There are excellent references available on the web, such as

- R Development Core Team (2014). An Introduction to R. `http://cran.r-project.org/doc/manuals/R-intro.html`

Statistical software packages

Overview

Without any doubt, Microsoft Excel is (unfortunately) the most widely used statistical software, even though this is largely due to its wide availability rather than its suitability.[1]

One of the oldest statistical software packages is BMDP, which was developed in 1961 at the University of California in Los Angeles. Even though it is still sold, it hardly used today.

Minitab was developed in the 1970s at Pennsylvania State University and is mostly used for teaching and was used at Glasgow for levels 1 and 2 until 2012.

SPSS (originally "Statistical Package for the Social Sciences") is mostly used in Social Sciences and uses a graphical user interface (GUI) akin to Minitab, though historically, both were command-line programs. SPSS has been taken over by IBM and was for a short time marketed by IBM as PASW (Predictive Analytics Software).

The SAS system dominates the commercial market. Compared to other statistical software packages SAS has very sophisticated data management functions (like a full SQL engine) and typically allows for analysing more complex data than Minitab or SPSS. A graphical user interface ("Enterprise Guide") is available. The pharmaceutical industry uses almost exclusively SAS.

S is a statistical programming language developed by John M. Chambers and others in the late 1970s and early 1980s at Bell Labs. According to John Chambers, the aim of the software was "to turn ideas into software, quickly and faithfully." The S engine was licensed to and finally purchased by Insightful (now acquired by TIBCO), which sells a value-added version called S-Plus (now marketed as Spotfire S+) containing a graphical user interface. S-Plus used to dominate the high-end market (academic and industrial research).

There are many other statistical software packages (some of which are niche products): GenStat, GLIM, Stata, WinBugs, BayesX, etc.

R

GNU[2] R was originally written by two researchers at the University of Auckland (New Zealand), Ross Ihaka and Robert Gentleman, but is now maintained by the R Core Team. R is an implementation of the S

[1] For a critical review of Excel's statistical functions, see e.g. `http://www.forecastingprinciples.com/paperpdf/McCullough.pdf` or `http://www.stat.uni-muenchen.de/~knuesel/elv/excelxp.pdf`

[2] for more information about the GNU project, see `http://www.gnu.org/`

programming language, which is in many respects superior to the original S system. R is a free software, you can obtain it for free, and the source code of R is freely available, so (if you want) you can study how R works internally and modify it as you like. R is extensible and a large selection of extensions packages can be obtained from CRAN[3]. R is a dynamically typed[4], object-oriented[5], interpreted[6] programming language.

R has largely replaced S-Plus in the academic world and is increasingly being adopted in industry. The New York Times recently dedicated an article to R.[7] The New York Times also uses R to create some of the charts used online and in the printed newspaper.

Why use a command-line program?

R is a command-line program, i.e. you control R by typing commands. Whilst command-line programs were ubiquitous until the mid-1990s, most software nowadays uses menu-driven graphical user interfaces (GUI), like e.g. Word or Minitab. Why should we then use an "old-fashioned" command-line program in the 21st century?

Menu-driven software works very well when used for a limited set of tasks. However, if one includes the R packages on CRAN, R can carry out several hundred thousands of different tasks. These can simply not be arranged in a menu in any meaningful way. In addition, most menu-driven software is very inflexible: you can often only use in the way the authors have designed it to be used.

Command-line based programs are much more flexible, you can do things no one has done or even thought of before by writing your own programs. You can write them from scratch or re-use some of the functions already provided. Some menu-driven software (like e.g. Minitab) offer the option of using macros. However these macros are often very clumsy and less elegant than say R code.

Furthermore, programming languages are very similar. If you know how to program in R, you will find learning other languages like C, C++, C#, Java, Javascript, PHP or Python much easier.

We will use an integrated development environment, called RStudio, which makes working with R more comfortable.[8]

How to use RStudio

To start RStudio in the lab open the *Maths & Stats* folder on the Desktop and then double-click the RStudio icon. This is extremely important to make sure RStudio does not function slowly in the labs. Below is a screenshot of RStudio.

[3] `http://cran.r-project.org`

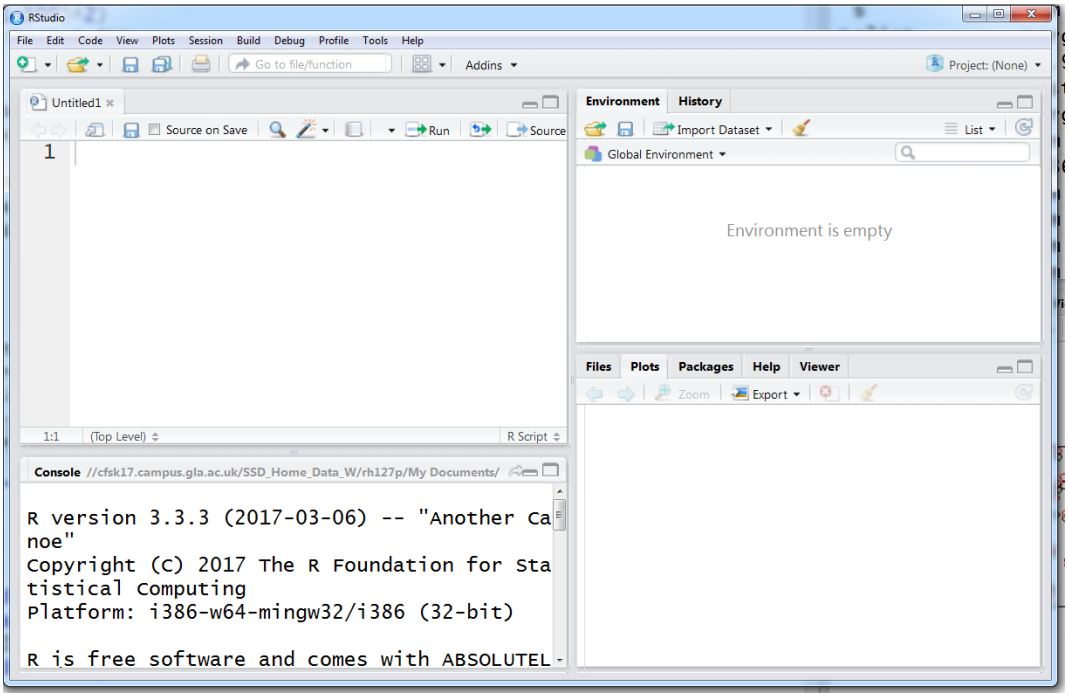
[4] You do not have to declare variables before using them. See e.g. `http://en.wikipedia.org/wiki/Type_system`.

[5] Depending on their type, data constructs can behave differently. For example, the R function `summary` for example performs different calculations for a data set and a linear model fit. See also `http://en.wikipedia.org/wiki/Object_oriented_programming`.

[6] In an intrepreted language program code is executed step-by-step, without prior translation to machine code ("compilation") See e.g. `http://en.wikipedia.org/wiki/Interpreted_language`.

[7] see `http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html`

[8] RStudio is free software and can be obtained from `http://www.rstudio.org/`.



After starting RStudio it is best to start with either creating a new R script (by clicking on *File* > *New File* > *R Script* or clicking on the left-most button) or opening an existing R script (by clicking on *File* > *Open File* or clicking on the second button from the left).

You can type R commands directly into the R console running at the bottom-left of RStudio. However it is typically better to type the R commands into the editor at the top-left. The commands can then be submitted to R by highlighting the commands and clicking on the *Run* button. Pressing `Ctrl-Enter` also submits the current selection or, if no text is selected, the current line of code.

The top-right has a list of the current objects in the workspace as well as a second tab with a history of commands used in the past.

The bottom-right contains four tabs: one showing the files in the current working directory, one showing the plots drawn so far, one showing all available extension packages and finally (and most importantly) one showing the R help.

The script editor and the R console have a number of useful features, most of which are common to many integrated development environments (IDEs):

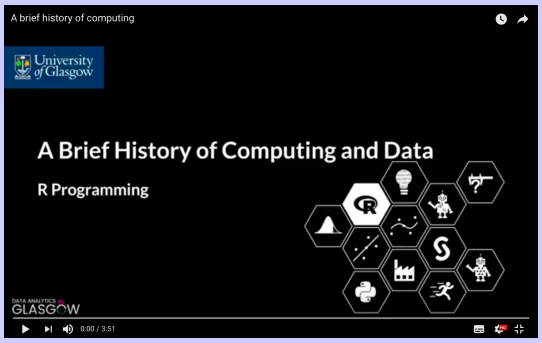
Syntax highlighting The colour of the text is changed automatically according to the R syntax rules. This makes the code easier to read.

Matching brackets When typing a closing bracket (`)`, `]`, `}`) the corresponding opening bracket is highlighted. This helps determining the correct number and positioning of closing brackets.

Auto completion If you start typing a variable name (or the name of an argument of a function) pressing `Tab` will automatically complete the name if it is unique or show a context menu with all possible completions. For functions the context menu also shows an explanation of each argument.

Help for current command Pressing `F1` after having typed the name of a function (e.g. `sort`) opens the help file for this function.

A brief history of computing



A Brief History of Computing and Data

https://youtu.be/mjDbSsKkVdc

Duration: 3m51s

Computers have become powerful

The advances in computational power and data storage are the drivers behind the ascent of Data Science and Analytics. In this section we look at two simple examples illustrating how powerful desktop computers are nowadays. Don't worry too much about the details of the R code at this stage.

Matrix multiplication Suppose we want to multiply two matrices, each having 1,000 rows and columns. How long would it take a "human computer"? Suppose that we can add or multiply two numbers in a second (which is rather optimistic), i.e. we manage to do 1 floating point operation per second. The resulting matrix has 1,000 × 1,000 entries, i.e. we must compute 1,000,000 numbers, each of which is a sum of 1,000 products, i.e. we need to carry out $2 \cdot 10^9$ additions and multiplications, i.e. it would take us $2 \cdot 10^9$ seconds, i.e. more than 32 years (working 24/7). Let's see how long R takes to do this (results are from a mid-range Core i5 processor).

```
n <- 1e3
A <- matrix(runif(n^2),nrow=n)          # Create a 1000x1000 matrix with random numbers.
B <- matrix(runif(n^2),nrow=n)          # Another 1000x1000 matrix with random numbers.
system.time(C <- A%*%B)                 # Time how long it takes to multiply them.
```

```
##    user  system elapsed
##   1.169   0.012   1.311
                                        # The third figure is the elapsed time in seconds.
```

So, this has only taken less than 1/10th of a second.

Sorting Suppose we have a data vector of 1,000,000 values. If we printed 5 values per row and 80 rows per page, the numbers would fill 2,500 pages. How long will it take to sort these values?

```
n <- 1e6
x <- runif(n)                           # Create a vector with 1000000 random values.
system.time(sort(x))                    # Time how long it takes to sort them.
```

```
##    user  system elapsed
##   0.117   0.013   0.140
                                        # The third figure is the elapsed time in seconds.
```

Again, we obtained the result in much less than a second.

Computers make mistakes

The enormous arithmetic power of modern computers can lead to the wrong impression that computers are infallible black boxes, which, regardless of what tasks we set them, obtain the right answer. Computers only have a finite precision and do not have the oversight most of us have and take for granted.

The following examples should act as a warning not to blindly trust a computer.

Is addition commutative? All of us know that $10^{20} - 10^{20} + 1 = 1$. Using R we obtain the same result:

```
10^20-10^20+1
```

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

Of course $1 + 10^{20} - 10^{20} = 1$ as well. According to R, however,

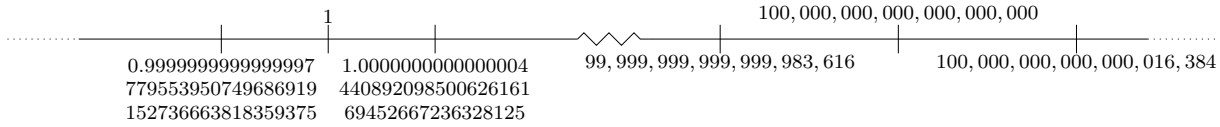
```
1+10^20-10^20
```

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

None of us would have made this mistake, as we would see at once that the sum of 10^{20} and -10^{20} is 0, thus the answer must be 1. The computer processes this sum from left to right, and for the computer $1+10^{20} \approx 10^{20}$, as 1 is very small compared to 10^{20}. In fact the next smallest number a computer can represent is $99,999,999,999,999,983,616$, which is much further away from $10^{20} - 1$ than 10^{20} itself. Subtracting 10^{20} then yields the wrong result 0. In other words, addition is not necessarily commutative on a computer, so the order of the terms might matter.

A computer only has finite precision, so we cannot represent arbitrarily large numbers, and there are "gaps" between the numbers. Like most other software, R uses IEEE 754 double precision floating point numbers. Floating point numbers are the computer implementation of scientific notation (like "$3 \cdot 10^{-9}$"), i.e. the significant and the exponent are stored separately. Storing the exponent separately makes the decimal point "float". The largest number that can be represented is $2^{1024} \approx 1.7977 \cdot 10^{308}$, which is large enough for most purposes. If a computation results in a value larger than this, arithmetic overflow occurs. In the past, this typically caused the program to abort. However, in IEEE 754 arithmetic and thus in R, the result is simply set to `-Inf` or `+Inf`.

The problem causing the computer to get the wrong result are however the "gaps" between the numbers: between each number and next smaller (or larger) number there is a gap of about $2 \cdot 10^{-16}$ times the number. And for 10^{20} this "gap" is larger than 1: see the figure below.



Note that in our example (and in many other situations) we can ensure that this problem does not occur by making the computer carry out the operations in a certain order.

More simple arithmetic You are used to rounding errors from your calculators. For example both on a computer and on a calculator $\frac{5}{6} - \frac{1}{6} \cdot 5$ is not 0:

```
5/6 - 1/6 * 5
```

```
## [1] 1.110223e-16
```

A similar, but more surprising example is that $0.1 + 0.1 + 0.1 - 0.3$ is not 0 on a computer:

```
0.1+0.1+0.1-0.3
```

```
## [1] 5.551115e-17
```

The result is almost (but only almost) 0. Again, we would have expected the computer to get this right.

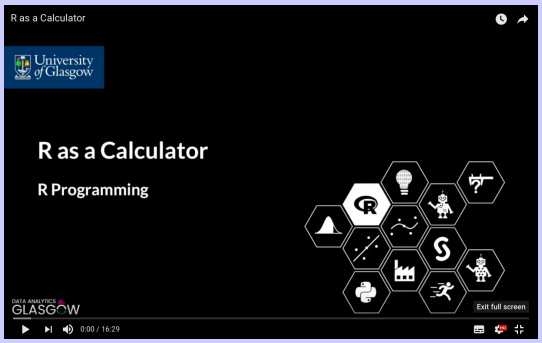
Almost all modern computers (as opposed to calculators) internally use a binary system instead of the decimal system we were taught at school. And in binary numbers $0.1 + 0.1 + 0.1 - 0.3$ is $0.000110011\ldots + 0.000110011\ldots + 0.000110011\ldots - 0.01001001\ldots$. As neither 0.1, nor 0.3 have a finite representation in a binary system a rounding error occurs.

To quote from the book *The Elements of Programming Style* by Kernighan and Plauger: "10.0 times 0.1 is hardly ever 1.0".

Rounding errors for a single computation are typically very small. However computers often carry out a long series of calculations, and typically rounding errors do not cancel out, but accumulate. Thus a complex computation can be subject to a significant error.

R as a calculator

Basic arithmetic operators



R as a calculator

`https://youtu.be/Gib3Wk2FFi8`

Duration: 16m29s

This section gives an overview over the basic arithmetic operators and functions in R. The following table contains the basic arithmetic operators available in R.

Operator	Meaning	Example	Result
`+`	Addition	`3+2`	`5`
`-`	Subtraction	`3-2`	`1`
`*`	Scalar multiplication	`3*2`	`6`
`/`	Division	`5/2`	`2.5`
`%/%`	Integer division	`5%/%2`	`2`
`%%`	Remainder after integer division	`5%%2`	`1`
`^` or `**`	Power	`5^2`	`25`

If an R expression contains more than one operator, we need to know in which order R evaluates the expression. This is known as *operator precedence* in Computer Science. For example, does

```
2 / 3 * 2
```

compute $\frac{2}{3 \cdot 2} = \frac{1}{3}$ or $\frac{2}{3} \cdot 2 = \frac{4}{3}$?

R uses the following rules:

- R first evaluates `^` and `**`, then the sign `-` (*not* difference), then `%/%` or `%%`, then `*` or `/`, and finally `+` or `-` (difference, *not* sign).
- In case of ties (operators of same precedence) the expressions are evaluated from the left to the right.

Thus in the above example `2/3*2` computes $\frac{2}{3} \cdot 2$.

Use parenthesis to get R to perform calculations in a different order. For example, in order to calculate $\frac{2}{3 \cdot 2}$, you have to use

```
2 / (3 * 2)
```

Example 1 *To compute* $\left(\frac{2}{3}\right)^{\frac{1}{4}}$ *we have to use*

```
(2/3)^(1/4)
```

```
## [1] 0.903602
```

If we omit the parentheses and enter

```
2/3^1/4
```

```
## [1] 0.1666667
```

R computes $\frac{2}{\frac{3^1}{4}} = \frac{1}{6}$.

Task 1 *Use R to compute* $3 + \frac{4}{5}$, $\frac{3+4}{5}$, *and* $27^{1/3}$.

Mathematical functions and constants

A large choice of mathematical functions is available in R, such as `abs`, `sign`, `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `gamma`, `beta`, etc.

The variable `pi` contains the value of π. You can generate the constant e using `exp(1)`.

IEEE 754 special values

R supports the IEEE 754 special values `Inf`, `-Inf`, and `NaN`, so you can carry out very limited computations on $\mathbb{R} \cup \{-\infty, +\infty\}$. These special values allow for mitigating some of the problems caused by numerical underflow (number rounded to zero) and overflow (number larger than the largest number which can represented by the computer).

```
1 / 0
```

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

for example gives `Inf`, whereas

```
1 / Inf
```

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

gives `0`. If you ask R to compute

```
Inf / Inf
```

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

it will return `NaN` (not a number): it cannot tell what the result is. Expressions like

```
sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced
```

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

give a warning and the result is `NaN`. R can handle complex numbers, just use

```
sqrt(-1+0i)
```

```
## [1] 0+1i
```

Note that a `NaN` (not a number) is *not* the same as `NA` (missing value, '`not available`').

Example 2 *`log(0)` returns `-Inf`, as* $\lim_{x \searrow 0} \log(x) = -\infty$. *`Inf-10` returns `Inf`, as* $\lim_{x \to +\infty} x - 10 = +\infty$. *However `Inf - Inf` is `NaN`, as the limit is ambiguous. Similarly, `sqrt(Inf) / Inf` is `NaN`. R evaluates `sqrt(Inf)` first, which is `Inf`. `Inf/Inf` is `NaN`.*

Variables and assignments

In all the above examples R simply returned a value. If we want to reuse the value, we need to assign the value to a variable. Variables are a little bit like the memory function of your calculator, except that you can use as many different variables as you like. The default assignment operator in R is `<-`. To store the result of `2/3 * 2` in a variable called `a`, we would use:

```
a <- 2/3 * 2
```

You can also use the more common assignent operator `=` instead of `<-` in most (but not all) circumstances. Assignments can be made in both directions, so you could also use

```
2/3 * 2 -> a
```

to store the number $\frac{4}{3}$ in the variable `a`, though `->` is used very rarely.

Variable names are case-sensitive, i.e. you can define both `a` and `A`, and `a` and `A` can hold different values. Historically, most R users only use lowercase letters and separate words using dots, e.g. `two.words`, though underscores have become increasingly popular in variable and function names.

If we want to print the value of the variable `a`, we just enter its name:

```
a
```

```
## [1] 1.333333
```

This is equivalent to

```
print(a)
```

```
## [1] 1.333333
```

which is what needs to be used inside control structures and functions (we will come back to this later).

You can define new variables using the values stored in other variables, as in

```
b <- a / 5
```

Note that changing `a` to something else will not automatically change `b`, so in the above example

```
a <- 10
b <- a / 5
a <- a + 30
b
```

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

`b` will stay `2`, even though `a` will be `40`.

In case you forgot to use a variable, the last expression you computed is stored in `.Last.value`.

The use of variables is an important tool in programming. Variables ensure that even code performing complex operations remains legible.

You can list all variables in the current workspace using

```
ls()
```

```
## [1] "a" "A" "b" "B" "C" "n" "x"
```

Alternatively, local objects are shown in the *Enrivonment* tab of RStudio.

Task 2 *In the video we have considered the example of taking out a loan of £9,000 for 20 years with an annual interest rate of 15%. The yearly repayment can be shown to be*

$$P = L \cdot \frac{1-v}{v(1-v^n)}.$$

We have used the following R code to calculate the yearly payment

```
n <- 20                                 # term of the loan
loan <- 9000                            # amount
interest.rate <- 0.15                   # effective annual interest rate
v <- 1 / (1+interest.rate)              # effective annual discount factor
payment <- loan * (1-v) / (v*(1-v^n))   # yearly payments
payment
```

```
## [1] 1437.853
```

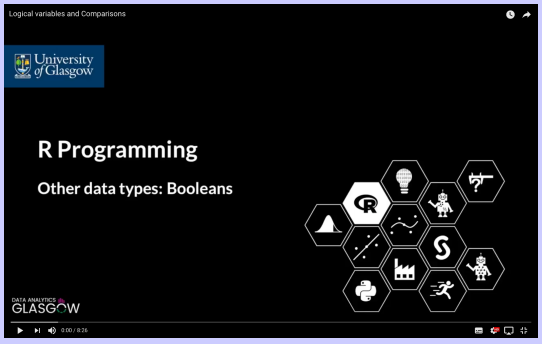
Of your yearly payments of £1437.86, how much is interest and how much is used for paying back the loan? The interest you pay in the first year is $L \cdot i$. The remainder of the first payment $(P - L \cdot i)$ is thus used for paying back the loan. More generally, one can show that the payment in year k can be decomposed into

$$P = \underbrace{P \cdot \alpha_k}_{\text{capital repayment}} + \underbrace{P \cdot (1-\alpha_k)}_{\text{interest}}$$

with $\alpha_k = v^{n+1-k}$.

Compute how much of the 10th payment is used for paying back the loan $(P \cdot \alpha_{10})$ and how much is interest $(P \cdot (1-\alpha_{10}))$.

Logical variables and comparisons



Other data types: Booleans

https://youtu.be/Xrm1cp-WSLM

Duration: 6m48s

Logical variables

A logical variable can only hold the two values `TRUE` and `FALSE`. Logical variables are sometimes called Boolean variables, after George Boole (1815–1864), an English mathematician, logician and philosopher. R

has the following three binary operators `!` (negation), `&` (logical AND) and `|` (logical OR).

`expr1`	`expr2`	`!expr1` (NOT)	`expr1 & expr2` (AND)	`expr1 | expr2` (OR)
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE

Example 3

```
a <- TRUE
b <- FALSE
c <- a & !b
c
```

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

In this example `c` *is* `TRUE`*, because* `!b` *is* `TRUE` *and* `TRUE & TRUE` *is* `TRUE`*.*

In R, `&` has higher precedence than `|`. So, in the absence of parentheses, `&` is evaluated before `|`. For example, `TRUE | FALSE & FALSE` is treated by R as `TRUE | (FALSE & FALSE)`, which is `TRUE`. We have to use parentheses to calculate `(TRUE | FALSE) & FALSE`, which is `FALSE`.

Task 3 *Consider three logical variables*

```
a <- TRUE
b <- FALSE
c <- TRUE
```

Without using R determine the values of . . .

```
!a & b
a | !b
!(a | !b)
(a & b) | c
```

R also allows using the shorthands `T` instead of `TRUE` and `F` instead of `FALSE`. It is however *not* recommended to use these shorthands. Whereas `TRUE` and `FALSE` are reserved keywords, which cannot be overwritten, `T` and `F` are just global variables set to `TRUE` and `FALSE`, respectively. This means they can be masked by local variables. Nothing (in R) prevents you from setting

```
T <- FALSE
F <- TRUE
```

and `T` and `F` have become the exact opposite of what they are meant to be! Of course, few users would do this deliberately, but it is not inconceivable that you happen to define a variable `T` or `F`, which can, depending on its values, have exactly the same effect.

R also has "lazy" operators `&&` and `||`. In contrast to `&` and `|` they will only evaluate the arguments until the result has become clear. On the other hand, `&` and `|` will always evaluate all arguments. `expr1&&expr2` will evaluate `expr2` only if `expr1` is `TRUE` (otherwise the result is guaranteed to be `FALSE` no matter what `expr2` is). Similarly, `expr1||expr2` will evaluate `expr2` only if `expr1` is `FALSE` (otherwise the result is guaranteed to be `TRUE` no matter what `expr2` is). `&&` and `||` can be helpful in conditional `if` statements. You should *not* use `&&` and `||` for vectors of length greater than 1.

Comparison operators

The comparison operators in R are `==` (testing for exact equality), `!=` ("not exactly equal") `<`, `<=`, `>`, and `>=`.

The comparison operators return a logical value (i.e. `TRUE` or `FALSE`), so you can use the operators `!`, `&` and `|` to combine them to more complex expressions.

Example 4 *Consider a variable* `x` *set to the number 2.*

```
x <- 2
```

We can then test whether it is negative or less than or equal to 3.

```
x < 0
```

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

```
x <= 3
```

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

If we want to test whether `x` *is in the unit interval we can use*

```
x>0 & x<1
```

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

or, equivalently,

```
!(x<=0 | x>=1)
```

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

Due to rounding and representation errors, you do not want to use `==` to compare non-integers. For example, despite $0.3 - 2 \times 0.1 = 0.1$, R yields

```
0.3 - 2 * 0.1 == 0.1
```

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

because the expression on the left-hand side is not exactly 0.1. We see this by subtracting 0.1 from the left-hand side (which should then be exactly zero, but isn't)

```
0.3-2*0.1 - 0.1
```

```
## [1] -2.775558e-17
```

For non-integers we really only want to test whether they are "nearly equal", we can do so by comparing the absolute difference to a small number (say 10^{-8}).

```
abs(0.3-2*0.1 - 0.1) < 1e-8
```

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

or use the built-in function `all.equal`.

```
isTRUE(all.equal(0.3-2*0.1, 0.1))
```

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

Task 4 *Create a variable* `x` *storing the fraction* $\frac{355}{113}$. *Use R to test*

- *whether this fraction is less than* π *(*`pi` *in R),*
- *whether this fraction is between 3 and 4, and*
- *whether this fraction is within* $\pm 10^{-6}$ *of* π.

Answers to the tasks

Task 1 You can use the following code.

```
3 + 4 / 5                  # No parentheses necessary
```

```
## [1] 3.8
```

```
(3 + 4) / 5                # Parentheses needed
```

```
## [1] 1.4
```

```
27^(1/3)                   # Parentheses needed
```

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

Task 2 To compute how the payment is split in year 10 we can use the code below.

```
n <- 20                                # term of the loan
loan <- 9000                           # amount
interest.rate <- 0.15                  # effective annual interest rate
v <- 1 / (1+interest.rate)             # effective annual discount factor
payment <- loan * (1-v) / (v*(1-v^n))  # yearly payments
k <- 10                                # set k to 10 years
alpha10 <- v^(n+1-k)                   # split factor
capital10 <- payment * alpha10         # capital repaymment
capital10
```

```
## [1] 309.0568
```

```
interest10 <- payment * (1-alpha10)    # interest part
interest10
```

```
## [1] 1128.796
```

So even after ten years the largest part of the repayment is interest!

Task 3 If we use R to work out the answers we obtain

```
!a & b
```

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

```
a | !b
```

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

```
!(a | !b)
```

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

```
(a & b) | c
```

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

Task 4 We can use the following R code.

```
x <- 355 / 113
x < pi
```

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

```
x>3 & x<4
```

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

```r
abs(x-pi) < 1e-6
```

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

Structures and Operations

Vectors in R

Vectors

https://youtu.be/pWNNWw-G51I

Duration: 6m58s

So far we have only looked at scalar variables, i.e. variables containing only one value. In most programming languages such scalar variables are the basic data types. R however has not scalar variables, all variables are vectors. A variable storing a number is for example just a numeric vector of length 1.

Defining vectors

The simplest way to create vectors in R is to use the function `c`:

```
a <- c(1, 4, 2)
a
```

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

We can use the function `c` as well to concatenate ("stick together") two vectors:

```
a <- c(1, 4, 2)
b <- c(5, 9, 13)
c <- c(a, b)
c
```

```
## [1]  1  4  2  5  9 13
```

Task 1 *Create a vector 'x' that contains the values* $\mathbf{x} = (1, 3, 2, 5)$ *and a vector 'y' which contains the values* $\mathbf{y} = (1, 0)$. *Finally, concatenate 'x' and 'y' and store the result as 'z'. Print 'z'.*

Naming entries

If the entries of the vectors correspond to quantities with natural names it is a good idea to associated names with the entries of the vector. This can be done using

```
names(a) <- c("first", "second", "third")
a
```

```
##  first second  third
##      1      4      2
```

Alternatively we can set the names when creating the vector using `c`

```r
a <- c(first=1, second=4, third=2)
```

Accessing elements

You can use square brackets to access a single element of a vector. `x[i]` returns the `i`-th element of the vector `x`. Using the vector `a` from above,

```r
a[3]
```

```
## third
##     2
```

You can use the same notation to change elements of a vector:

```r
a[3] <- 10
a
```

```
##  first second  third
##      1      4     10
```

If the element you want to change is beyond the last element, the vector will be extended, such that the `i`-th element is the last element.

```r
a[7] <- 99
a
```

```
##  first second  third
##      1      4     10     NA     NA     NA     99
```

If the entries of the vector are named, you can also use the names for accessing elements.

```r
a["third"]
```

```
## third
##    10
```

Subsetting vectors

You can use square brackets not only to access a single element of a vector, but also to subset a vector. There are three ways of specifying subsets of vectors:

You can use a vector specifying the indices to be returned:

```r
a <- c(1, 4, 9, 16)
a[c(1,2,3)]
```

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

You can use a vector specifying the indices to be removed (as negative numbers):

```r
a[-4]
```

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

You can use a logical vector specifying the elements to be returned:

```r
a[c(TRUE, TRUE, TRUE, FALSE)]
```

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

We can exploit the latter when we want to subset a vector based on its values. Suppose you want to keep all elements in `a` that are divisible by 2:

```
a[a%%2==0]
```

```
## [1]  4 16
```

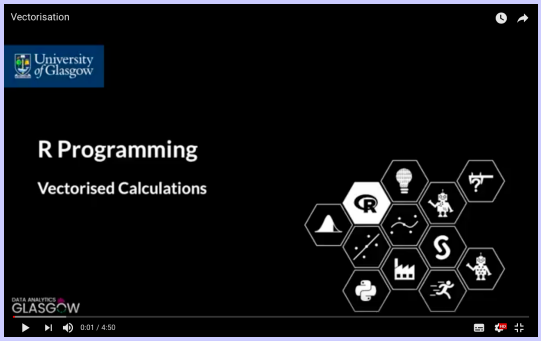
Why does this work? `a\%\%2==0` returns a logical vector of length 4, indicating which elements of `a` are divisible by 2. These elements are then selected from `a`.

Task 2 *Consider the vector 'x' defined as*

```
x <- c(1, 5, 9, 3, 8)
```

Use all three of the above methods to extract the first, third and fifth entry.

Vectorised calculations



Vectorised calculations

`https://youtu.be/dILiux93ueA`

Duration: 4m50s

Vectors can be used in arithmetic expressions using the arithmetic operators and the mathematical and statistical functions we have seen when we used R as a calculator. In this case the computations are carried out element-wise.

For example,

```
a <- c(1, 2, 3, 4)
b <- c(2, 0, 1, 3)
c <- 2 * a + b
c
```

```
## [1]  4  4  7 11
```

The third entry of the result, 7, is obtained by taking twice the third entry of the vector `a` and adding it to third entry of the vector `b`, i.e. $c_3 = 2 \times a_3 + b_3 = 2 \times 3 + 1 = 7$.

Recycling rules

If vectors of different length are used in an arithmetic expression, the shorter vector(s) are repeated ("recycled") until they match the length of the longest vector.

```
a <- c(1, 2, 3, 4)
b <- c(2, 0)
a * b
```

```
## [1] 2 0 6 0
```

R has thus "recycled" the vector `b` once.

If the length of the longest vectors is not a multiple of the length of the shorter vector(s), R will produce a warning. For example,

```
a <- c(1, 2, 3, 4)
b <- c(2, 0, 1)
a * b
```

```
## Warning in a * b: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 0 3 8
```

The vector `b` is shorter than the vector `a`. However, if `b` is recycled once, it would have 6 elements, making it longer than `a`. Thus R produces a warning. Such a warning is almost always a sign that you have made a mistake.

Task 3 *Consider the following two vectors 'x' and 'y' and their sum*

```
x <- c(1, 2, 9, 4, 5, 6)
y <- c(0, 2)
x+y
```

```
## [1] 1 4 9 6 5 8
```

Explain how R has calculated the result.

Sequences and patterned vectors

Sequences

R has built-in functions to create simple sequences and patterned vectors:

The operator `:` can be used for creating basic sequences.

```
2:5
```

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

If the first argument is larger then the second argument, then the sequence will be decreasing.

```
1:0
```

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

`seq(from, to, by)` creates a sequence from `from` to `to` using `by` as increment.

```
seq(1, 2, by=0.2)
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

`seq(from, to, length.out=n)` creates a sequence of length `n` from `from` to `to`.

```
seq(3, 5, length.out=5)
```

```
## [1] 3.0 3.5 4.0 4.5 5.0
```

Repeats

`rep(x, times=n)` repeats the vector `x` `n` times.

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

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

`rep(x, each=n)` repeats each element of the vector `x` `n` times.

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

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

Task 4 *Create each of the following vectors using ':', 'seq' and 'rep'.*

```
2 3 4 5 6
2 4 6
1.00 1.25 1.50 1.75 2.00
3 3 4 4 5 5
2 3 4 2 3 4
```

Useful functions for vectors

`numeric(n)` creates a numeric vector of length `n` (containing 0's).

`length(x)` returns the length (numebr of elements) of the vector `x`.

`unique(x)` returns the unique elements of `x`.

`rev(x)` reverses the vector `x`, i.e. returns $(x_n \ldots, x_1)$

Sorting

The function `sort(x)` sorts the vector `x`. The function `order(x)` returns the permutation requird to sort the vector `x` Consider the vector

```
x <- c(11, 7, 3, 9, 4)
```

We can sort `x` using

```
sort(x)
```

```
## [1]  3  4  7  9 11
```

So what does the funtion order do?

```
p <- order(x)
p
```

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

The first entry of the result `p` is 3. The third entry of `x` is the smallest: if we want to sort `x` we have to put the third entry first, or, in other words, the first entry of the sorted vector would be the third entry of `x`. The second entry of `p` is 5, because the second smallest entry of `x` is its fith entry.

We can obtain the sorted vector by applying the permutation obtained from `order` to `x`

```
x[p]
```

```
## [1]  3  4  7  9 11
```

This trick can be used to sort an entire dataset by one column.

If we want to sort the vector in descending order, we have to append the argument `decreasing=TRUE`.

```
sort(x, decreasing=TRUE)
```

```
## [1] 11  9  7  4  3
```

Scalar summary functions

R has a large selection of functions that compute a scalar function of a vector. Their names are self-explanatory: `min`, `max`, `sum`, `prod`, `mean`, `median`, `sd`, `var`, ...

Task 5 *Use R to calculate the sums* $\sum_{i=1}^{10} i$ and $\sum_{i=1}^{100} i^2$.

Other data types in R

Character strings

So far we have considered vectors which were either numerical or logical. In this section we will look at character strings.

Character strings in R can be defined using double or single quotes.

```
string <- "A string in single quotes"
another.string <- 'This time defined using single quotes'
```

Strings can be printed using `print` (or in the console just the variable name).

```
string
```

```
## [1] "A string in single quotes"
```

Use the function `cat` to print the string as it is

```
cat(string)
```

```
## A string in single quotes
```

R has a variety of built-in functions for manipulating strings. It is however easier to use the functions from the package stringr.

For example, two strings can be concatenated using the function `str_c` (the equivalent base R function is `paste`).

```
library(stringr)
str_c("Two strings", "put together", sep=" - ")
```

```
## [1] "Two strings - put together"
```

Note that you cannot use character vectors for any sort of arithmetic. For example

```
"120" + "5"
```

```
## Error in "120" + "5": non-numeric argument to binary operator
```

You are unlikely to do this deliberately, but you might come across if R treats a supposedly numerical variable as a character string because of at least one non-numerical entry.

Comparisons between character strings use lexicographic ordering, i.e. they are compared based on where they would be in a dictionary. For example

```
"apple">"pear"
```

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

However character strings containing numbers are compared in an unexpected way

```
"120" > "5"
```

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

In lexicographic ordering "5" comes after "120" (as only the first digit "1" matters).

Factors

Factors are a variant on that theme and designed for use with categorical variables. The main difference between a factor and a character vector is that factors are only allowed to take a pre-defined set of values ("levels").

Any vector can be converted to a factor using the function `factor`. It has the additional (optional) arguments `levels` and `labels` which can be used to set the labels printed when a vector is displayed.

```
x <- c(1, 4, 2, 4, 1, 3, 1, 2, 4)
X <- factor(x, levels=1:5, labels=c("one", "two", "three", "four", "five"))
X
```

```
## [1] one   four  two   four  one   three one   two   four
## Levels: one two three four five
```

In our example, the vector `X` is only allowed to take the values `"one"`, `"two"`, `"three"`, `"four"` and `"five"` (the level `"five"` is currently not being used). Thus we can set for example

```
X[1] <- "five"
X
```

```
## [1] five  four  two   four  one   three one   two   four
## Levels: one two three four five
```

We cannot set the first entry to `"six"`. `"six"` is not in the set of allowed labels.

```
X[1] <- "six"
```

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

```
X
```

```
## [1] <NA>  four  two   four  one   three one   two   four
## Levels: one two three four five
```

In order to be able to set the first entry to `"six"` we need to first expand the set of levels.

```
levels(X) <- c(levels(X), "six")
X[1] <- "six"
X
```

```
## [1] six   four  two   four  one   three one   two   four
## Levels: one two three four five six
```

To turn `X` back into its original numerical format we can use the function `unclass`.

```
unclass(X)
```

```
## [1] 6 4 2 4 1 3 1 2 4
## attr(,"levels")
## [1] "one"   "two"   "three" "four"  "five"  "six"
```

Converting between data types

You can convert between different data types by using `as.<target-datatype>`. For example you can convert

```
x <- pi                 # x is numeric
x <- as.character(x)    # now x is a character string
x <- as.numeric(x)      # x is numeric again (but we lost some digits)
```

Often R will convert between different types automatically. The most common data types are

Data type	Conversion function	Description
numeric	`as.numeric`	Floating point numbers
integer	`as.integer`	Integer numbers
logical	`as.logical`	`TRUE` or `FALSE`
character	`as.character`	Character string
factor	`as.factor`	Factor

Missing values

R uses the special value `NA` to indicate that a value is missing. It is different from `NaN`, which is "not a number", i.e. a value for which the calculations have gone wrong.

You cat set a value to `NA` by simply assigning it the value `NA`.

```
x <- 1:4
x[4] <- NA
x
```

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

Calculations (arithmetic, summary functions, etc.) involving `NA`s have the result set to `NA` as well: the result depends on the value that is not available.

```
mean(x)
```

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

If you want R to omit the missing values you can either use

```
mean(na.omit(x))
```

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

or

```
mean(x, na.rm=TRUE)
```

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

The former is more generic as not every function support the additional argument of `na.rm=TRUE`.

Use the function `is.na` to test whether a value is missing.

```
is.na(x)
```

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

You cannot use `==` to test whether a value is missing

```
x==NA
```

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

The comparison just results in `NA`.

Matrices

Matrices

`https://youtu.be/slxsFdCNfkk`

Duration: 6m45s

Matrices in R

Matrices are the two-dimensional generalisation of vectors. The main difference between a vector and a matrix is that a vector has a single index, whereas a matrix has two indices: row and column.

Internally, R stores matrices in column-major mode, i.e. the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

is stored as

`1 2 3 4 5 6 7 8 9`

i.e. R internally stacks the columns on top of each other, which is known as "column-major mode". If you had stored the matrix $\mathbf{A}$ is a two-dimensional array in C or Java, it would be stored in what is called "row-major mode", i.e. the rows would be stacked on top of each other.

Creating matrices

There are essentially three ways of creating a matrix in R.

Using the internal representation

The first one consists of using the internal representation of matrices as vectors. If we want to create a matrix

$$\mathbf{B} = \begin{bmatrix} 0 & 2 & 9 \\ 7 & 4 & 6 \end{bmatrix}$$

we can use the command `matrix`.

```
B <- matrix(c(0, 7, 2, 4, 9, 6), nrow=2)
```

Alternatively, you could specify the number of columns using `ncol=3`.

The function `matrix` can also be used to create "empty" matrices. `matrix(a, nrow, ncol)` creates a `nrow`×`ncol` matrix in which every entry is set to `a`.

Row-wise build-up

Another option is to build up the matrix row-wise using the function `rbind`.

```
B <- rbind(c(0, 2, 9),
           c(7, 4, 6))
```

We can also use `rbind` to add a row to an existing matrix.

```
rbind(B, c(1, 2, 9))
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
## [3,]    1    2    9
```

If a vector given to `rbind` is shorter than the other rows, it is "recycled" using the same rules as used for vector arithmetic. For example, to add a row of 0's to the matrix $\mathbf{B}$ we can use

```
rbind(B, 0)
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
## [3,]    0    0    0
```

Column-wise build-up

The third option consists of using `rbind`'s sibling `cbind`. `cbind` adds a column to a matrix and can be used to build up a matrix column-wise. Thus we can create the matrix $\mathbf{B}$ using

```
B <- cbind(c(0, 7), c(2, 4), c(9, 6))
```

Task 6 *Use all three methods fom above to create the matrix*

$$\mathbf{M} = \begin{bmatrix} 9 & 2 & 4 \\ 3 & -2 & 7 \\ 4 & 8 & -1 \end{bmatrix}$$

Dimensions of a matrix

To find out the dimensions of a matrix you can use the three functions `nrow`, `ncol` and `dim`.

```
nrow(B)
```

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

```
ncol(B)
```

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

```
dim(B)
```

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

The function `length` returns the number of entries of a matrix ($2 \times 3 = 6$ in our case)

```
length(B)
```

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

Diagonal matrices

Diagonal matrices have a special role in Linear Algebra and thus in Statistics. For this reason R has a function dedicated to diagonal matrices: `diag`.

```
E <- diag(c(1 ,4 , 2))
E
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    4    0
## [3,]    0    0    2
```

`diag` can also be used to access the diagonal of an existing matrix. The matrix does not even need to be diagonal for this. You can change the second element of the diagonal of the matrix `E` to `5` using

```
diag(E)[2] <- 5
E
```

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

Task 7 *Create in R the diagonal matrix*

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & -9 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Naming rows and columns

When working with data matrices it is a good idea to name at least the columns of a matrix. It is much better to refer to variables in a matrix by their name rather than the column index in which they are stored.

Rows and columns can be named using the functions `rownames` and `colnames`.

```
colnames(B) <- c("First column", "Second column", "Third column")
rownames(B) <- c("First row", "Second row")
B
```

```
##            First column Second column Third column
## First row             0             2            9
## Second row            7             4            6
```

Accessing elements and submatrices

Single entries of matrices can be accessed using square brackets. To extract B_{23}, i.e. the value in the second row and third column of $\mathbf{B}$ we can use

```
B[2,3]
```

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

Similarly, we can set B_{23} to `-1` using

```
B[2,3] <- -1
```

Though this is not recommended, we could have also used the internal vector representation and extract the sixth element of the internal representation

```
B[6]
```

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

You can access arbitrary submatrices by specifying the rows and columns you wish to access. You can do so by using any combination of the three methods used for vectors. To extract the first row and first and second column of **B** you can use any of the following lines (. . . and there are many other ways of doing so):

```
B[1, 1:2]
```

```
##  First column Second column
##             0             2
```

```
B[-2, 1:2]
```

```
##  First column Second column
##             0             2
```

```
B[c(TRUE, FALSE), -3]
```

```
##  First column Second column
##             0             2
```

Each line returns the vector $[0, 2]$

Because the output object has only one row it is returned as a vector (instead of a matrix). In complex programmes in which the result is then expected to be matrix this can sometimes cause problems. In this case you can use the additional argument `drop=FALSE`:

```
B[1, 1:2, drop=FALSE]
```

```
##           First column Second column
## First row            0             2
```

If we only want to subset rows and columns the elector for the other dimension can be left empty. To access the first row of the matrix **B** use

```
B[1,]
```

```
##  First column Second column  Third column
##             0             2             9
```

To access the third column of **B** use

```
B[, 3]
```

```
##  First row Second row
##          9         -1
```

To replace the third column of **B** with the numbers $(1, 8)$ you can use

```
B[, 3] <- c(1, 8)
```

Furthermore, logical expressions can be used to subset matrices in the same way as they are used to subset vectors. Suppose you want to set all entries larger than 5 to 6.

```
B[B > 5] <- 6
B
```

```
##            First column Second column Third column
## First row             0             2            1
```

```
## Second row            6             4             6
```

Task 8 *Using the matrix* $\mathbf{M}$ *from Task 6:*

- *extract the first row,*
- *set the top-right entry to 0,*
- *add 1 to the last column.*

Matrix multiplication and linear algebra

Matrix multiplication

The basic arithmetic operators can be applied to matrices in the same way as they can be applied to vectors. They are interpreted element-wise. Most importantly, `*` performs element-wise multiplication. In order to perform matrix multiplication you need to use the operator `%*%`.

To compute the matrix product

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 9 \\ 7 & 4 & -1 \end{bmatrix}$$

in R, you can use:

```
A <- matrix(c(1, 0, -1, -1, 1, 0), ncol=2)
B <- matrix(c(0, 7, 2, 4, 9, -1), ncol=3)
A %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   -7   -2   10
## [2,]    7    4   -1
## [3,]    0   -2   -9
```

Note that matrix multiplication is generally not commutative (unless $\mathbf{A}$ and $\mathbf{B}$ are symmetric), thus `A\%*\%B` is not the same as `B\%*\%A`.

The transpose $\mathbf{A}^\top$ of a matrix $\mathbf{A}$ can be computed using the function `t(A)`. The cross product $\mathbf{A}^\top\mathbf{A}$ can be computed using the function `crossprod(A)`.

Matrix inverse and linear systems of equations

The function `solve` computes the matrix inverse. For example,

```
C <- B%*%t(B)              # Create a square matrix C=BB'
C.inv <- solve(C)          # Compute the inverse of C
C.inv%*%C                  # Check whether C.inv is indeed the inverse
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
C%*%C.inv
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

The function `solve` can be used not only for inverting a matrix, but also for solving a (non degenerate) system of linear equations. `solve(A, b)` solves the system of equations $\mathbf{Az} = \mathbf{b}$ for $\mathbf{z}$.

To solve the system of equations

$$\begin{aligned} 5z_2 &+ z_3 &= 7 \\ 7z_2 &- z_3 &= 5 \\ 11z_1 + z_2 &+ z_3 &= 14 \end{aligned}$$

you can use

```
A <- rbind(c( 0, 5,  1),
           c( 0, 7, -1),
           c(11, 1,  1))
b <- c(7, 5, 14)
z <- solve(A, b)
z
```

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

We can check the answer by computing $\mathbf{Az}$, which should then be $\mathbf{b}$.

```
A%*%z
```

```
##      [,1]
## [1,]    7
## [2,]    5
## [3,]   14
```

You could as well first compute the inverse of $\mathbf{A}$ and then compute $\mathbf{A}^{-1}\mathbf{b}$, i.e. use

```
z <- solve(A)%*%b
```

but this is slightly less efficient than the above code.

Linear systems of equations play a key role in Data Analytics and Statistics. For example, you will learn that the least-squares estimate of the coefficients in a linear regression model can be found by

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y},$$

which we can rewrite as solving the system of linear equations

$$\underbrace{(\mathbf{X}^\top\mathbf{X})}_{=\mathbf{A}}\underbrace{\boldsymbol{\beta}}_{=\mathbf{z}} = \underbrace{\mathbf{X}^\top\mathbf{y}}_{=\mathbf{b}}$$

Matrix decomposition

In certain special cases the linear system of equations $\mathbf{Az} = \mathbf{b}$ can be solved more easily by decomposing the matrix $\mathbf{A}$. The key idea is to rewrite the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{A} = \mathbf{BC}.$$

We can then solve $\mathbf{Az} = \mathbf{b}$ by solving $\mathbf{BCz} = \mathbf{b}$. We can do this by first solving $\mathbf{Bv} = \mathbf{b}$ for $\mathbf{v}$ and then solving $\mathbf{Cz} = \mathbf{v}$ for $\mathbf{z}$. Then $\mathbf{Az} = \mathbf{BCz} = \mathbf{Bv} = \mathbf{b}$, i.e. $\mathbf{z}$ is indeed a solution to the linear system of equations.

The trick here is that the matrices $\mathbf{B}$ and $\mathbf{C}$ are of a form for which solving the associated system of equations is much simpler than the original one.

Choleski decomposition

We look here at one specific type of decomposition, although many others are commonly used in linear algebra, namely the *Choleski* decomposition. If the matrix $\mathbf{A}$ is symmetric and positive-definite then the *Choleski* decomposition can be computed as

$$\mathbf{A} = \mathbf{LL}^{\mathrm{T}},$$

where $\mathbf{L}$ is lower-diagonal. The Choleski decomposition can be computed using the function `chol` in R. Let's consider the matrix $\mathbf{A}$ so defined

```
A <- matrix(c(1,3,3,13),nrow=2)
A
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    3   13
```

and its Choleski decomposition is

```
L <- t(chol(A))
L
```

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

We can now verify that indeed $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$

```
L%*%t(L)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    3   13
```

Eigenvectors and eigenvalues

The function `eigen` can be used to compute the eigenvectors and eigenvalues of a square matrix. For example,

```
A.eig <- eigen(A)
A.eig
```

```
## eigen() decomposition
## $values
## [1] 13.7082039  0.2917961
##
## $vectors
##            [,1]       [,2]
## [1,] 0.2297529 -0.9732490
## [2,] 0.9732490  0.2297529
```

We can now numerically verify the spectral decomposition $\mathbf{A} = \mathbf{\Gamma}\mathbf{\Lambda}\mathbf{\Gamma}^{\mathrm{T}}$, where $\mathbf{\Gamma}$ is the matrix of eigenvectors and $\mathbf{\Lambda}$ is the diagonal matrix containing the eigenvalues.

```
A.eig$vectors %*% diag(A.eig$values) %*% t(A.eig$vectors)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    3   13
```

which is identical to $\mathbf{A}$.

Solutions to the tasks

Task 1

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

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

Task 2 We can use the following R code, but there are many other equally good answers

```
x <- c(1,5,9,3,8)
x[c(1,3,5)]
```

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

```
x[-c(2,4)]
```

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

```
x[c(TRUE,FALSE,TRUE,FALSE,TRUE)]
```

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

Task 3 The recycling rule means that R treats `x+y` as `x+yyy` where

```
yyy <- c(0,2,0,2,0,2)
```

Indeed

```
x <- c(1,2,9,4,5,6)
x+yyy
```

```
## [1] 1 4 9 6 5 8
```

Task 4

```
2:6
```

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

```
seq(2,6,by=2)
```

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

```
seq(1,2,length.out=5)
```

```
## [1] 1.00 1.25 1.50 1.75 2.00
```

```
rep(3:5,each=2)
```

```
## [1] 3 3 4 4 5 5
```

```
rep(2:4,2)
```

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

Task 5 We can calculate $\sum_{i=1}^{10} i$ using

```
sum(1:10)
```

```
## [1] 55
```

We can calculate $\sum_{i=1}^{100} i^2$ using

```
sum((1:100)^2)
```

```
## [1] 338350
```

Note that we need to put the power of 2 inside the parenthesis. If we had used

```
sum(1:100)^2
```

```
## [1] 25502500
```

we would have calculated $\left(\sum_{i=1}^{100} i\right)^2$. We also have to put `1:100` inside parentheses. If we had used

```
sum(1:100^2)
```

```
## [1] 50005000
```

we would have calculated $\sum_{i=1}^{100^2} i$.

Task 6 Using the internal representation

```
M <- matrix(c(9,3,4,2,-2,8,4,7,-1),ncol=3)
```

Using `rbind`

```
M <- rbind(c(9,3,4),c(2,-2,8),c(4,7,-1))
```

Using `cbind`

```
M <- cbind(c(9,3,4),c(2,-2,8),c(4,7,-1))
```

Task 7 You can use the following R code

```
diag(c(4,7,-9,4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4    0    0    0
## [2,]    0    7    0    0
## [3,]    0    0   -9    0
## [4,]    0    0    0    4
```

Task 8 You can use the following R code

```
M[1,]
```

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

```
M[1,3] <- 0
M[,3] <- M[,3] + 1
```

Data Management

Lists

Lists

https://youtu.be/-MumGJIIrTI

Duration: 6m37s

Creating lists

You can create a list using the `list` command.

```
example1 <- list(1:3, c(TRUE,FALSE), 7)
example1
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] 7
```

Lists can be nested within each other.

```
example2 <- list(list(pi, 1:2), list("some text", 1:3))
example2
```

```
## [[1]]
## [[1]][[1]]
## [1] 3.141593
##
## [[1]][[2]]
## [1] 1 2
##
##
## [[2]]
## [[2]][[1]]
## [1] "some text"
##
## [[2]][[2]]
## [1] 1 2 3
```

Just like vectors, lists can be concatenated using the function `c`.

```
c(example1, example2)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] 7
##
## [[4]]
## [[4]][[1]]
## [1] 3.141593
##
## [[4]][[2]]
## [1] 1 2
##
##
## [[5]]
## [[5]][[1]]
## [1] "some text"
##
## [[5]][[2]]
## [1] 1 2 3
```

The first three entries of the resulting list come from `example1`, the remaining fourth and fifth entry from `example2`.

It is good programming style to name the elements of a list (if possible). This can be done using `names`:

```
names(example1) <- c("a", "b", "c")
example1
```

```
## $a
## [1] 1 2 3
##
## $b
## [1]  TRUE FALSE
##
## $c
## [1] 7
```

Alternatively, you can specify the names directly in the `list` command:

```
example1 <- list(a=1:3, b=c(TRUE,FALSE), c=7)
```

Accessing elements

Elements of a list can be accessed using either double square brackets or, if the list has names, `$`. The following three lines of R code all return the first entry of the list `example1` (using either its position or its name).

```
example1[[1]]
```

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

```
example1[["a"]]
```

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

```
example1$a
```

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

if you want to subset a list (in the sense of extracting more than one entry) use single square brackets:

```
example1[1:2]
```

```
## $a
## [1] 1 2 3
##
## $b
## [1]  TRUE FALSE
```

If we extract an individual entry using single square brackets we obtain a list containing that entry rather than the entry itself.

```
example1[1]
```

```
## $a
## [1] 1 2 3
```

Lists are everywhere ...

Most R functions return lists. Take the `lm` function as an example. The function `lm` fits a linear regression model, which is essentially a straight line fit to the data. First of all we fit a linear model:

```
fit <- lm(dist~speed, data=cars)          # Fit a linear model
fit                                       # Print the model
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

The output object `fit` is in fact a list.

```
is.list(fit)
```

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

Let's look at its entries

```
names(fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

So, if we for example want to retrieve the regression coefficients we can use

```
fit$coefficients
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

Actually, we do not have to fully spell out the name of the entry after the `$`, we only have to use the first few characters until the name is uniquely determined. So we could have also used

```
fit$coef
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

We could have not used

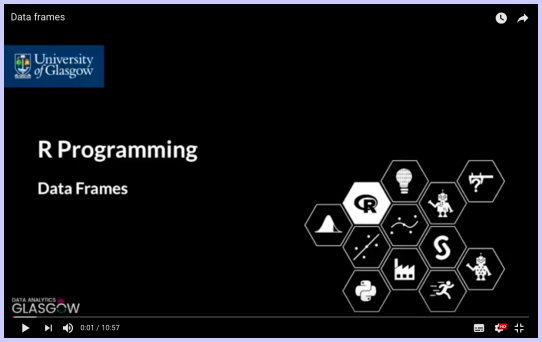
```
fit$c
```

```
## NULL
```

as there is more than one element with a name starting with a "c" (`"call"` and `"coefficients"`).

Data frames

Data frame basics



Data frames

`https://youtu.be/SSei0Sf_nu0`

Duration: 10m57s

Last week we have learned how to create matrices. In principle, we could use matrices to store data sets. However, matrices (and vectors) have one important constraint: all entries of a vector or a matrix have to be of the same data type. Many data sets we work with have both numerical and factor variables, so we would need to store our data using different data types for the different columns. We could in theory use lists to manage such data. However, lists do not enforce the constraint that each variable has the same number of observations, so if we used lists our data would soon get messy.

Let's look at an example illustrating this limitation of matrices.

Consider a matrix `kids` containing age, weight and height of two children.

```
kids <- rbind(c( 4, 15, 101),
              c(11, 28, 132))
colnames(kids) <- c("age", "weight", "height")
rownames(kids) <- c("Sarah", "John")
kids
```

```
##       age weight height
## Sarah   4     15    101
## John   11     28    132
```

We can for example see that John is older than Sarah.

```
kids["John", "age"] > kids["Sarah", "age"]
```

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

Let's now try adding a column gender.

```
kids2 <- cbind(kids, gender=c("f", "m"))
kids2
```

```
##       age  weight height gender
## Sarah "4"  "15"   "101"  "f"
## John  "11" "28"   "132"  "m"
```

Let's see whether John is still older than Sarah.

```
kids2["John", "age"] > kids2["Sarah", "age"]
```

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

Not any more. How come? We have added the variable `gender`, which is a character vector. As all the data in the matrix needs to be of the same data type, R had to convert all the columns, including `age` to character strings, and character strings compare not in the same way as numbers: in a dictionary `11` would be before `4`. We can see from the quotes around the numerical variables that these have been converted to characters (accidental conversion to factors is slightly more difficult to spot, as R prints factors without quotes).

Because of situations like this one it is better to use data frames to store data sets.

Creating data frames

A matrix can be converted to a data frame using `as.data.frame` and we can convert a data frame back to a matrix using `as.matrix`.

```
kids <- as.data.frame(kids)
kids <- cbind(kids, gender=c("f","m"))
kids
```

```
##       age weight height gender
## Sarah   4     15    101      f
## John   11     28    132      m
```

```
kids["John", "age"] > kids["Sarah", "age"]
```

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

A data frame can handle column of different data types, so the numeric column is not converted when adding a character (or factor) column.

Data frames can be created using the function `data.frame`, so we could have created the data set using

```
kids <- data.frame(age=c(4,11), weight=c(15,28), height=c(101,132), gender=c("f", "m"))
rownames(kids) <- c("Sarah", "John")
```

or

```
kids <- rbind(Sarah=data.frame(age=4, weight=15, height=101, gender="f"),
              John=data.frame(age=11, weight=28, height=132, gender="m"))
```

A data frame has row names and column names like a matrix and these can be set in the same way as for matrices. Data frames can also be subset like matrices. We will come back to this later on.

Data frames also behave like lists

Data frames behave not just only like matrices, they also behave like lists (with the columns being the entries). We can use `$` to access columns:

```
kids$age
```

```
## [1]  4 11
```

So for a data frame, the following four lines of R commands are all equivalent

```
kids$age
```

```
## [1]  4 11
```

```
kids[,"age"]
```

```
## [1]  4 11
```

```
kids[,1]
```

```
## [1]  4 11
```

```
kids[["age"]]
```

```
## [1]  4 11
```

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

```
## [1]  4 11
```

As mentioned previously, it is always better to refer to columns by their name rather then their index. The latter is too likely to change as you work with the data.

We can use the same notation to set values in the data frame. Suppose it was John's birthday and we want to change his age to 5. We can use any of the lines below (and there are even more possible ways).

```
kids["John", "age"] <- 5
kids$age[2] <- 5
kids[2,1] <- 5
kids[[2]][1] <- 5
```

Again, the command at the top is probably best, because it is the most "human-readable".

Data manipulation

In this section we now look at various R functions for manipulating data frames. We work with a toy dataset called `chol`, which you can load into R using

```
load(url("http://www.stats.gla.ac.uk/~levers/rp/chol.RData"))
```

The data set contains (simulated) blood fat measurements for a small number of patients.

Adding new columns

Transforming/adding variables

https://youtu.be/XJE9x9qW0zk

Duration: 3m57s

Suppose we want to add to our dataset a new column called `log.hdl.ldl` which contains the logarithm of the ratio of HDL and LDL cholesterol.

Using what we have seen so far we could use

```
chol <- cbind(chol, log.hdl.ldl=log(chol[,"hdl"]/chol[,"ldl"]))
```

or

```
chol[,"log.hdl.ldl"] <- log(chol[,"hdl"]/chol[,"ldl"])
```

or

```
chol$log.hdl.ldl <- log(chol$hdl/chol$ldl)
chol
```

```
##    ldl hdl trig age gender     smoke log.hdl.ldl
## 1  175  25  148  39 female        no -1.94591015
## 2  196  36   92  32 female        no -1.69459572
## 3  139  65   NA  42   male      <NA> -0.76008666
## 4  162  37  139  30 female ex-smoker -1.47667842
## 5  140 117   59  42 female ex-smoker -0.17946849
## 6  147  51  126  65 female ex-smoker -1.05860695
## 7   82  81   NA  57   male        no -0.01227009
## 8  165  63  120  48   male   current -0.96281075
## 9  149  49   NA  32 female        no -1.11212601
## 10  95  54  157  55 female ex-smoker -0.56489285
## 11 169  59   67  48 female        no -1.05236127
## 12 174 117  168  41 female        no -0.39688136
## 13  91  52  146  69 female   current -0.55961579
```

The first two commands work for data frames and matrices whereas the last one only works for data frames.

All of the above commands look slightly messy and use the name of the dataset more than once, which is a potential source of mistakes when you want to change the name of the dataset in the future.

It is generally better to use the functions transform. Its main advantage is that we do not need to put `chol$` everywhere.

```
chol <- transform(chol, log.hdl.ldl=log(hdl/ldl))
```

Removing columns

You can use the subsetting techniques for matrices to remove columns. For example,

```
chol <- chol[,-3]
```

removes the column `trig` (third column). Alternatively, you could use list-style syntax:

```
chol[[3]] <- NULL
```

or

```
chol$trig <- NULL
```

The null object in R is `NULL`. You can test whether an object is null by using `is.null(object)`. `object==NULL` does however *not* work.

Subsetting data sets

Data frames can be subset just like matrices. For example, to remove all patients who have never smoked and store the result in a data frame called `chol.smoked` we can use

```
chol.smoked <- chol[chol$smoke!="no",]
chol.smoked
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## NA  NA  NA  NA   <NA>      <NA>          NA
## 4  162  37  30 female ex-smoker  -1.4766784
## 5  140 117  42 female ex-smoker  -0.1794685
## 6  147  51  65 female ex-smoker  -1.0586070
## 8  165  63  48   male   current  -0.9628107
## 10  95  54  55 female ex-smoker  -0.5648928
## 13  91  52  69 female   current  -0.5596158
```

Because there is a missing value in the smoking status, the new data set starts with a row of missing values. This is due to the fact that `chol$smoke!="no"` has as third entry `NA`.

```
chol$smoke!="no"
```

```
##  [1] FALSE FALSE    NA  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [12] FALSE  TRUE
```

For every `NA` in the condition, R will put a row of `NA`'s at the top of the data set.

We then have to remove the row(s) with missing values, which is best done using the function `na.omit`

```
chol.smoked <- na.omit(chol.smoked)
```

We can subset the data frame in one go when we use the function `subset`. Just like `transform` it provides a cleaner solution because it does not require us to use `chol$` in the condition.

```
chol.smoked <- subset(chol, smoke!="no")
chol.smoked
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## 4  162  37  30 female ex-smoker  -1.4766784
## 5  140 117  42 female ex-smoker  -0.1794685
## 6  147  51  65 female ex-smoker  -1.0586070
## 8  165  63  48   male   current  -0.9628107
## 10  95  54  55 female ex-smoker  -0.5648928
## 13  91  52  69 female   current  -0.5596158
```

Task 1 *Create a data frame* `chol.lowhdl` *containing the data for patients with a HDL cholesterol of less than 40 mg/dl.*

Sorting data sets

The function `order` can be used to sort an entire data set by one column. The data frame chol can be sorted by the age of the patient using

```
permut <- order(chol$age)      # Create the permutation ordering the age
chol <- chol[permut,]          # Apply this permutation to the entire data set
chol
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## 4  162  37  30 female ex-smoker -1.47667842
## 2  196  36  32 female        no -1.69459572
## 9  149  49  32 female        no -1.11212601
## 1  175  25  39 female        no -1.94591015
## 12 174 117  41 female        no -0.39688136
## 3  139  65  42   male      <NA> -0.76008666
## 5  140 117  42 female ex-smoker -0.17946849
## 8  165  63  48   male   current -0.96281075
## 11 169  59  48 female        no -1.05236127
## 10  95  54  55 female ex-smoker -0.56489285
## 7   82  81  57   male        no -0.01227009
## 6  147  51  65 female ex-smoker -1.05860695
## 13  91  52  69 female   current -0.55961579
```

Attaching datasets

To access the column `height` in the data frame `kids` we have to use `kids$height` (or one of the above equivalent statements). Sometimes it would be easier just to refer to it as `height` (without the `kids$`) as we have done inside `transform` or `subset`. This can be done using the function `attach`. After calling

```
attach(kids)
```

we can access the columns of `kids` as if they were variables, i.e. we can use

```
weight / (height/100)^2
```

```
## [1] 14.70444 16.06979
```

instead of

```
kids$weight / (kids$height/100)^2
```

```
## [1] 14.70444 16.06979
```

To undo the effects `attach` simply use

```
detach(kids)
```

However, `attach` behaves in an unexpected way if you try to change any of the variables of a data frame you have already attached. The problem with `attach`'ed data is that it exists in R twice. Once as inside the data frame (where it was before you have called `attach`) and once as a variable in your current environment. R does however *not* link those two. If you change one of them, the other one does *not* get updated automatically.

Suppose we now want to change the weight from kgs to pounds, i.e. divide it by 0.45359237. If you now use

```
weight <- weight / 0.45359237
```

to change the unit, you have changed the unit only for the `attach`'ed variable.

```
weight
```

```
## [1] 33.06934 61.72943
```

```
kids$weight
```

```
## [1] 15 28
```

`weight` contains the weight in pounds, whereas `kids$weight` still contains the "old" weight in kgs.

If we had used

```
kids$weight <- kids$weight / 0.45359237
```

the opposite would have happened. `weight` would have contained the old version (in kgs), and only `kids$weight` would have contained the new version (in pounds). In other words, whatever we do, we will end up with inconsistent data.

Thus, is probably a good idea to avoid using `attach`. If you use `attach` remember this important rule: Never manipulate `attach`'ed data!

The function `with` is a safer alternative to `attach`. We could use

```
with( kids, {
  weight <- weight / 0.45359237
} )
```

to transform the column `weight` from kgs to pounds. In scripts using `with` is actually clearer than `attach`. However in the interactive console, `with` is slightly more awkward to use. Note that in this example it would have been easiest to use `transform`.

```
kids <- transform(kids, weight=weight / 0.45359237)
```

Task 2 *The data frame* `cia` *contains data about almost all countries taken from the CIA World Factbook.*

You can load the data frame into R using

```
load(url("http://www.stats.gla.ac.uk/~levers/rp/cia.RData"))
```

We will use the following columns.

Variable	Content
`Country`	Name of the country
`Continent`	Geographic region the country is located in
`Population`	Population
`Life`	Life expectancy at birth in years
`GDP`	Gross domestic product in USD
`MilitaryExpenditure`	Military expenditure in USD

(a) *Delete all observations for which the population is missing. You might find the functions* `is.na` *or* `complete.cases` *useful. (You can get R to show the help for these functions by entering* `?is.na` *or* `?complete.cases`*).*

(b) *Which countries have a population of less than 10,000 inhabitants?*

(c) *Which countries have a military expenditure of at least 8% of the GDP?*

(d) *Ignoring missing values, what is the combined GDP of all European countries?*

(e) *Create a new column* `GDPPerCapita`*, which contains the per capita GDP (*`GDP` *divided by* `Population`*). Also create a new column* `MilitaryExpPerCapita`*, which contains the per capita military expenditure*

`MilitaryExpenditure` *divided by* `Population`*).*

(f) *Which country has the highest life expectancy?*

(g) *Which ten countries have the highest life expectancy?*

Merging data sets

Often, the data required for a certain analysis is stored in more than one data frame. Many transactional databases are designed using the so-called "third-normal form" design principle, which, despite being optimal from a transactional point of view, requires combining many tables before being able to analyse the data. R has a built-in command `merge`, which allows for merging tables using common keys.

Consider the following example. In a study of 2,287 eighth-grade pupils (aged about 11) a language test score and the verbal IQ were determined. The question of interest is whether the socio-economic status (SES) of the parents and various characteristics of the class influence the language test score and the verbal IQ. The data is stored in two data frames, one containing the data about the children (`children`) and one containing the data about the different classes (`classes`).

The first few rows of `children` are:

```
load(url("http://www.stats.gla.ac.uk/~levers/rp/children_classes.RData"))
head(children)
```

```
##   lang   IQ SES class
## 1   46 15.0  23   180
## 2   45 14.5  10   180
## 3   33  9.5  15   180
## 4   46 11.0  23   180
## 5   20  8.0  10   180
## 6   30  9.5  10   180
```

The first few rows of `classes` are:

```
head(classes)
```

```
##   class size combined
## 1   180   29     TRUE
## 2   280   19     TRUE
## 3  1082   25     TRUE
## 4  1280   31     TRUE
## 5  1580   35     TRUE
## 6  1680   28     TRUE
```

There are good reasons for storing the data in this format. This way less redundant information is stored and the information about each classes is stored exactly once. This makes it easier to change class properties like for example the number of pupils.

However, for the analysis of the data it is necessary that we "copy" all information from the data frame `classes` into the data frame `children`: for every child we need to look up which class it belongs to and copy the information about that class into the row belonging to that child. Of course we cannot simply use `cbind`: the child in the second row of the data frame `children` attended class "180", which is described in the first row of `classes`.

The function `merge` can be used for such a task. By default, `merge` merges datasets using the columns both data frames have in common (in our case `class`)

```
data <- merge(children, classes)
```

It is typically better to explicitly specify which column(s) are to be used for merging the data frames. This avoids that columns that happen to have the same name in both data frames, but are not related are used in the merger. The common key(s) to be used for merging can be specified using the argument `by`, provided they have the same names in both data frames.

```
data <- merge(children, classes, by="class")
```

The arguments `by.x` and `by.y` allow merging data frames using columns which do not have the same name in both data frames (in our case they of course do have the same name).

```
data <- merge(children, classes, by.x="class", by.y="class")
```

For each child `merge` has looked up the information about the class and added it to the row in the resulting data frame `data`.

```
head(data)
```

```
##   class lang   IQ SES size combined
## 1   180   46 15.0  23   29     TRUE
## 2   180   45 14.5  10   29     TRUE
## 3   180   33  9.5  15   29     TRUE
## 4   180   46 11.0  23   29     TRUE
## 5   180   20  8.0  10   29     TRUE
## 6   180   30  9.5  10   29     TRUE
```

If there were children in a class for which there is no entry in `classes` R would by default remove these from the resulting data frame. Similarly, data from classes for which there are no pupils in `children` will also not appear in the resulting table.

If you want the resulting data frame to contain all cases from the first data frame even if there is no matching entry in the second data frame, you need to specify the additional argument (`all.x=TRUE`). If you want the resulting data frame to contain all cases from the second data frame even if there is no matching entry in the first data frame, you need to specify the additional argument (`all.y=TRUE`).

Task 3 *Consider two data frames* `patients` *and* `weights`*, which you can load into R using*

```
load(url("http://www.stats.gla.ac.uk/~levers/rp/patients_weights.RData"))
```

The first few rows of the data sets are

```
head(patients)
```

```
##   PatientID Gender Age   Smoke
## 1         1   male  33      no
## 2         2 female  32      no
## 3         3   male  67      ex
## 4         4   male  36 current
## 5         5 female  47 current
```

```
head(weights)
```

```
##   PatientID Week Weight
## 1         1    1     72
## 2         1    2     74
## 3         1    3     71
## 4         2    1     54
## 5         2    3     54
## 6         3    1     96
```

Merge the data sets such that there is information about the patient for each weighting.

Importing and exporting data from R

IMPORTANT: whenever using Rstudio in the labs and opening or saving R script files, navigate to either the H: or K: drive and open/save from there, but don't go directly or navigate into the Documents or Desktop folder.

Native .RData files

R has an internal binary data format ("`.RData` files"), which can be used to store one or more objects. Typically `.RData` or `.rda` is used as the file extension. The key advantage of using R's internal format is that it stores objects exactly as they are in R. If you load the objects back in you are guaranteed that they are exactly reproduced.

If you want to save a object `x` to a file you can use

```
save(x, file="MyX.RData")
```

If you want to save more than one object (say `x` and `y`) you can use

```
save(x, y, file="MyXY.RData")
```

If you want to save all objects in your workspace to a file (`MyVariables.RData`), you can use

```
save.image(file="MyVariables.RData")
```

To load the data you have saved back into R use

```
load(file="MyVariables.RData")
```

When saving the workspace at the end of a session, R simply stores all objects in your workspace in a file `.RData` in your home directory.

R's internal format is a good idea if you only work with R. However, very few other software products support it.

Path names in Windows typically contain backslashes (\). In R you need to either use a forward slash instead (for example `c:/Users/Ludger/data.RData`) or escape the backslash, i.e. use a double `\\` (for example `c:\\Users\\Ludger\\data.RData`).

Text and CSV files

Reading in data

`https://youtu.be/OWR6DJKpz3A`

Duration: 5m55s

Importing data into R

In most cases data is stored in a table or spreadsheet format. The easiest way of reading external data into R is to use delimited text files. If the data at hand is say an Excel spreadsheet it is typically easier to open it in

Excel first and convert it to a text (or CSV) file in Excel, and only then open the text file in R.

Before reading a text file into R, it is always a good idea to look at the file first using a raw text editor and determine the following information, which is easy for you to determine, but hard for R to guess automatically:

- Column names: Does the first line of the file contain data or does it contain the names of the columns ("variables")?
- Delimiter: What character is used to delimit the columns? This is typically white space, tabulator, `,` or `;`.
- Missing values: Determine how missing values are encoded (if there are any). R uses `NA`, but `*` and `.` are common as well.

Below are the first few lines of the file chol.txt.

```
175 25 148 39 female no
196 36 92 32 female no
139 65 NA 42 male NA
```

We can see that ...

- The first line contains data and *not* the names of the columns.
- The columns are delimited using white space.
- The data set uses `NA` to encode missing values.

Such white space (or tab) delimited files can be read in using the R function `read.table`. It assumes by default that the first line of the file does *not* contain the column names (i.e. the first line already contains data), that white space is used as delimiter, and that missing values are encoded as `NA`. If this is not the case, you need to use the following additional arguments:

- `header`: Use `header=TRUE` if the first line of the file contains the names of the columns.
- `sep`: If the delimiter of the columns is not white space, but another character, you need to use the additional argument `sep`. For comma-separated data, use `sep=","`. The latter is better read in using the function `read.csv`.
- `na.strings`: If missing values are encoded using strings other than "`NA`", you need to use the additional argument `na.strings`. If for example "`*`" is used to denote missing values, you would use `na.strings="*"`. The argument `na.strings` can be a character vector if more than one string is used to denote missing values. You do not need to use this argument if your data set does not contain any missing values.
- `dec`: You can use the additional option `dec` to set the decimal separator (e.g. `dec=','`)

The file `chol.txt` uses no column names, white space as a separator, and uses "`NA`" for missing values. Thus we do not need to use any additional arguments to read it into R.

```
chol <- read.table("chol.txt")
head(chol)
```

```
##    V1  V2  V3 V4     V5        V6
## 1 175  25 148 39 female        no
## 2 196  36  92 32 female        no
## 3 139  65  NA 42   male      <NA>
## 4 162  37 139 30 female ex-smoker
## 5 140 117  59 42 female ex-smoker
## 6 147  51 126 65 female ex-smoker
```

It is always worth looking at the first few lines of the data you have read in to make sure it was read in correctly.

If, like in our example, the data file does not contain variables, it is a good idea to set them right after you have read in the data.

```
colnames(chol) <- c("ldl", "hdl", "trig", "age", "gender", "smoke")
```

R tries to guess of what type each column (variable) is, but might not always get it right. It also is worth checking that each column was read in as the data type you had intended. This can be done using

```
sapply(chol, class)
```

```
##       ldl       hdl      trig       age    gender     smoke
## "integer" "integer" "integer" "integer"  "factor"  "factor"
```

Alternatively we could use

```
str(chol)
```

```
## 'data.frame':    13 obs. of  6 variables:
##  $ ldl   : int  175 196 139 162 140 147 82 165 149 95 ...
##  $ hdl   : int  25 36 65 37 117 51 81 63 49 54 ...
##  $ trig  : int  148 92 NA 139 59 126 NA 120 NA 157 ...
##  $ age   : int  39 32 42 30 42 65 57 48 32 55 ...
##  $ gender: Factor w/ 2 levels "female","male": 1 1 2 1 1 1 2 2 1 1 ...
##  $ smoke : Factor w/ 3 levels "current","ex-smoker",..: 3 3 NA 2 2 2 3 1 3 2 ...
```

If a variable which you had intended to be numeric (or an integer) shows up as a factor it is likely that missing values (or other error codes) were in the data that were not read in correctly.

The file chol.csv contains the same data, however in comma-separated ("CSV") format. The first few lines of this file are

```
ldl,hdl,trig,age,gender,smoke
175,25,148,39,female,no
196,36,92,32,female,no
139,65,.,42,male,.
```

We can see that ...

- The first line of the file are the column names.
- The columns are delimited using commas.
- The missing values are coded using ..

Thus we can read the file into R using

```
chol <- read.table("chol.csv", header=TRUE, sep=",", na.strings=".")
```

or

```
chol <- read.csv("chol.csv", na.strings=".")
```

`read.csv` is a sibling of `read.table` with the main difference that it assumes by default that the data is comma-separated and that the first line contains the variable names. In other words, you do not need to specify `sep=","` and `header=TRUE"` when using `read.csv`.

Task 4 *Read the data files* `cars.csv` *and* `ships.txt` into R.

Exporting data from R

Text files can be used as well to export data from R using the function

```
write.table(chol, file="chol.csv", sep=",", col.names=TRUE, row.names=FALSE)
```

The arguments `col.names` and `row.names` can be used to choose whether the column names and row names should be exported as well. The function `write.csv` can be used instead of the additional argument `sep=","`.

```r
write.csv(chol, file="chol.csv", row.names=FALSE)
```

Other file formats

There are many R packages that allow reading in data stored in file formats used by other software products. In most cases it is best to first convert the file into a text file using the proprietary software the file format corresponds to and then open the exported text file in R. However, there are R packages which allow opening various different file formats.

Package	Functions	Formats that can be read in
`readxl`	`read_excel`	Excel spreadsheets (`.xls` and `.xlsx`)
`xlsx`	`read.xlsx`, `write.xlsx`	Excel spreadsheets (only `.xlsx`)
`foreign`	`read.dta`, `write.dta`	Stata binary files
`foreign`	`read.spss`	SPSS files

Installing R packages

Before you can load an R package with `library(packagename)` you need to install it, either using the RStudio user interface (click on the tab *Packages* in the bottom right pane and then click on `Install` or install the package from command line using the command `install.packages("packagename")`.

Solutions to the tasks

Task 1

```
chol.lowdl <- subset(chol,hdl<40)
chol.lowdl
```

```
##   ldl hdl trig age gender     smoke
## 1 175  25  148  39 female        no
## 2 196  36   92  32 female        no
## 4 162  37  139  30 female ex-smoker
```

Task 2 (a)

```
cia <- subset(cia,!is.na(Population))
```

(b)

```
subset(cia,Population < 1e4)
```

```
##                               Country             Continent Population
## 50                  Christmas Island                  Asia       1402
## 52             Cocos (Keeling) Islands                  Asia        596
## 78  Falkland Islands (Islas Malvinas) Central/South America       3140
## 103           Holy See (Vatican City)                Europe        826
## 156                       Montserrat Central/South America       5097
## 170                             Niue                 Other       1398
## 171                   Norfolk Island                 Other       2141
## 183                  Pitcairn Islands                 Other         48
## 191                  Saint Barthelemy Central/South America       7448
## 192                      Saint Helena                Africa       7637
## 196         Saint Pierre and Miquelon         North America       7051
## 218                          Svalbard                 Other       2116
## 229                           Tokelau                 Other       1416
##      Life       GDP MilitaryExpenditure
## 50     NA        NA                  NA
## 52     NA        NA                  NA
## 78     NA 105100000                  NA
## 103    NA        NA                  NA
## 156 72.76        NA                  NA
## 170    NA  10010000                  NA
## 171    NA        NA                  NA
## 183    NA        NA                  NA
## 191    NA        NA                  NA
## 192 78.44        NA                  NA
## 196 79.07        NA                  NA
## 218    NA        NA                  NA
## 229    NA        NA                  NA
```

(c)

```
subset(cia,MilitaryExpenditure > 0.08*GDP)$Country
```

```
## [1] Iraq         Jordan       Oman         Qatar        Saudi Arabia
## 255 Levels: Afghanistan Akrotiri Albania Algeria American Samoa ... Zimbabwe
```

(d)

```
cia.europe <- subset(cia,Continent == "Europe" & !is.na(GDP))
sum(cia.europe$GDP)
```

```
## [1] 4.111738e+13
```

or alternatively

```
sum(subset(cia,Continent=="Europe")$GDP, na.rm=TRUE)
```

```
## [1] 4.111738e+13
```

(e)

```
cia <- transform(cia,GDPPerCapita=GDP/Population,MilitaryExpPerCapita=MilitaryExpenditure/Population)
```

(f)

```
cia[order(cia$Life,decreasing = TRUE)[1],]
```

```
##     Country Continent Population  Life       GDP MilitaryExpenditure
## 138   Macau      Asia     559846 84.36 2.204e+10                  NA
##     GDPPerCapita MilitaryExpPerCapita
## 138     39367.97                  NA
```

or alternatively

```
cia[which.max(cia$Life),]
```

```
##     Country Continent Population  Life       GDP MilitaryExpenditure
## 138   Macau      Asia     559846 84.36 2.204e+10                  NA
##     GDPPerCapita MilitaryExpPerCapita
## 138     39367.97                  NA
```

(g)

```
cia[order(cia$Life, decreasing = TRUE)[1:10], ]
```

```
##        Country     Continent Population  Life       GDP
## 138      Macau          Asia     559846 84.36 2.204e+10
## 6      Andorra        Europe      83888 82.51        NA
## 118      Japan          Asia  127078679 82.12 4.844e+12
## 206  Singapore          Asia    4657542 81.98 1.545e+11
## 199 San Marino        Europe      30324 81.97 8.500e+08
## 105  Hong Kong          Asia    7055071 81.86 2.238e+11
## 16   Australia         Other   21262641 81.63 1.069e+12
## 43      Canada North America   33487208 81.23 1.564e+12
## 82      France        Europe   64057792 80.98 2.978e+12
## 220     Sweden        Europe    9059651 80.86 5.129e+11
##     MilitaryExpenditure GDPPerCapita MilitaryExpPerCapita
## 138                  NA     39367.97                   NA
## 6                    NA           NA                   NA
## 118          3.8752e+10     38118.12             304.9449
## 206          7.5705e+09     33172.00            1625.4282
## 199                  NA     28030.60                   NA
## 105                  NA     31721.86                   NA
## 16           2.5656e+10     50275.97            1206.6234
## 43           1.7204e+10     46704.40             513.7484
## 82           7.7428e+10     46489.27            1208.7210
## 220          7.6935e+09     56613.66             849.2049
```

Task 3

```
weights.all <- merge(patients,weights,by="PatientID")
head(weights.all)
```

```
##   PatientID Gender Age Smoke Week Weight
## 1         1   male  33    no    1     72
## 2         1   male  33    no    2     74
## 3         1   male  33    no    3     71
## 4         2 female  32    no    1     54
## 5         2 female  32    no    3     54
## 6         3   male  67    ex    1     96
```

Task 4 The first line of the file `cars.csv` contains the variable names and the fields are separated by commas. Missing values are recorded as aterisks.

```
cars <- read.csv("cars.csv",na.strings ="*")
str(cars)
```

```
## 'data.frame':    20 obs. of  5 variables:
##  $ Manufacturer: Factor w/ 10 levels "Cadillac","Chevrolet",..: 2 8 3 2 2 10 3 10 1 6 ...
##  $ Model       : Factor w/ 20 levels "Achieva","Astro",..: 3 1 17 2 9 8 18 12 10 11 ...
##  $ MPG         : int  19 NA 22 NA 25 18 18 25 16 29 ...
##  $ Displacement: num  3.4 2.3 2.5 4.3 2.2 2.8 3 1.8 4.9 1.5 ...
##  $ Horsepower  : int  160 155 100 165 110 178 300 81 200 81 ...
```

or alternatively

```
cars <- read.table("cars.csv", header = TRUE, sep = ",", na.strings = "*")
str(cars)
```

```
## 'data.frame':    20 obs. of  5 variables:
##  $ Manufacturer: Factor w/ 10 levels "Cadillac","Chevrolet",..: 2 8 3 2 2 10 3 10 1 6 ...
##  $ Model       : Factor w/ 20 levels "Achieva","Astro",..: 3 1 17 2 9 8 18 12 10 11 ...
##  $ MPG         : int  19 NA 22 NA 25 18 18 25 16 29 ...
##  $ Displacement: num  3.4 2.3 2.5 4.3 2.2 2.8 3 1.8 4.9 1.5 ...
##  $ Horsepower  : int  160 155 100 165 110 178 300 81 200 81 ...
```

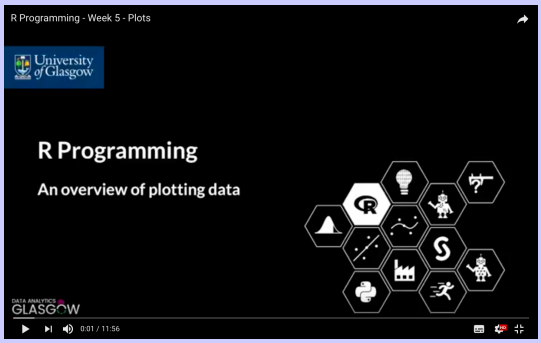
The first line of the file `ships.txt` contains the variable names and the fields are separated by whitespace. Missing values are encoded as dots.

```
ships <- read.table("ships.txt", header= TRUE, na.strings= ".")
str(ships)
```

```
## 'data.frame':    40 obs. of  5 variables:
##  $ type     : Factor w/ 5 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 2 2 ...
##  $ year     : int  60 60 65 65 70 70 75 75 60 60 ...
##  $ period   : int  60 75 60 75 60 75 60 75 60 75 ...
##  $ service  : int  127 63 NA 1095 1512 3353 0 2244 44882 17176 ...
##  $ incidents: int  0 0 3 4 6 18 0 11 39 29 ...
```

R Graphics

Plotting in R



An overview of plotting data

`https://youtu.be/BaZD5uKAPdg`

Duration: 11m56s

The above video gives an overview over the key high-level plotting functions in R. Although in the video some plots are produced using the `magrittr` syntax (which is not covered in the module), below you will find the standard R code to produce all the plots in the video.

R has a built-in sophisticated and customisable plotting engine, which we look at this week.

To get an idea of what R can do, enter

```
demo(graphics)
```

R is widely used for data visualisation: the New York Times has been using R to create some of the graphics in the newspaper and on its website.

Although all details you need are included in these notes, the following references may be useful:

1. Sections 4.2 to 4.4 of Introductory Statistics with R by Dalgaard.
2. Chapter 3 of A First Course in Statistical Programming with R by Murdoch and Brown.
3. Chapter 12 of the Introduction to R Manual by the R Core team.
4. The R Graph Gallery, where examples of plots are included.

The function plot

Specifying the coordinates

The function `plot` is at the heart of R's built-in graphics. It can be used to create two-dimensional plots.

The following four commands all create the same scatter plot of `y` against `x`:

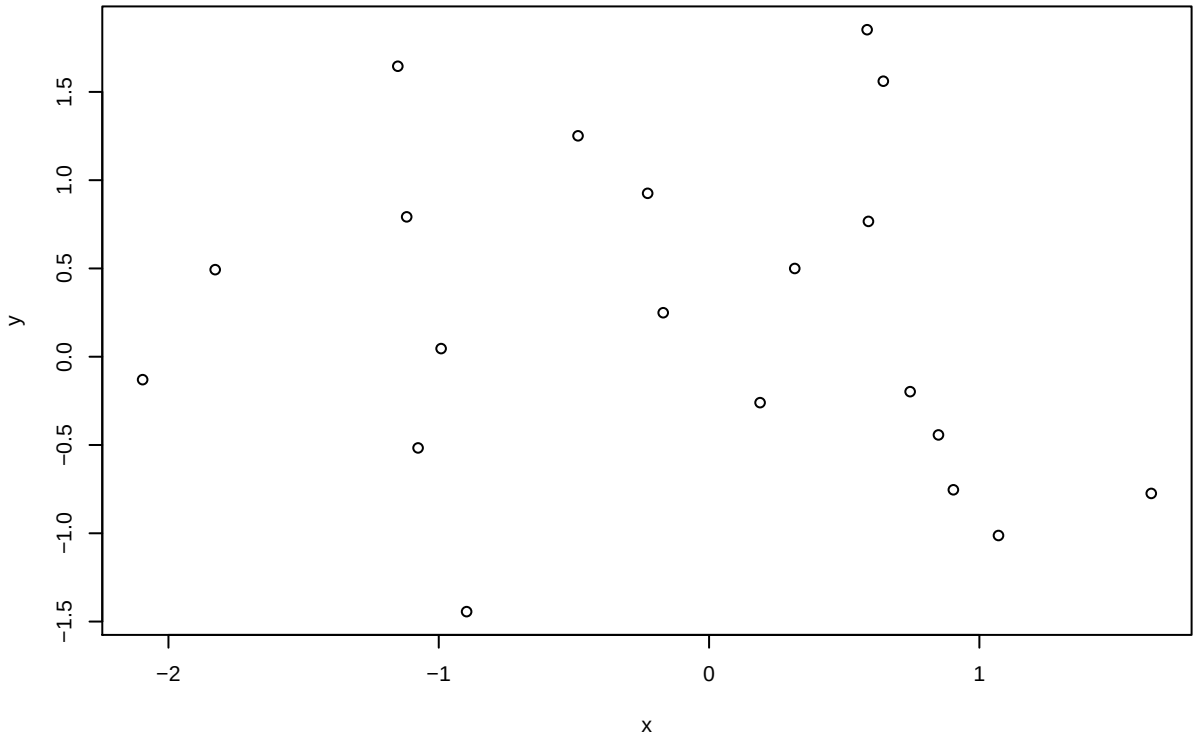
- `plot(x, y, ...)` where `x` and `y` are the vectors containing the coordinates.
- `plot(y~x, ...)` where `x` and `y` are the vectors containing the coordinates, or `plot(y~x, data=data, ...)` where `x` and `y` are column in `data`.
- `plot(M, ...)`, where `M` is a matrix having two columns (x- and y-coordinates). If `M` has more than two columns, R will ignore these.
- `plot(lst, ...)`, where `lst` is a list with (possibly amongst others) an element `x` and an element `y`.

We first look at a simple example illustrating these three ways of calling `plot`. We first create two vectors `x` and `y` containing white noise.

```
n <- 20
x <- rnorm(n)
y <- rnorm(n)
```

In this example, the simplest way of calling `plot` provides two vectors as arguments:

```
plot(x, y)
```



Alternatively we could have used (note that `y` comes first if we use the formula interface).

```
plot(y~x)
```

If we arrange `x` and `y` in matrix (or data frame)

```
data <- cbind(x=x, y=y)
```

we can use

```
plot(data)
```

or

```
plot(y~x, data=data)
```

For the former command the order of the columns in `data` matters.

If we store `x` and `y` in a list

```
lst <- list(x=x, y=y)
```

we can use

```
plot(lst)
```

Customising the plot

`plot` has a wide range of optional arguments. The most important ones are:

- `type`: controls how the data is plotted. Use `p` (default) for points, `l` for a line through the points, `b` for a line together with the points, `n` to set up the plot without actually plotting any points. For more options, see `?plot`.
- `xlab`: the label of the x axis (in quotes). Otherwise, R will use the name of the corresponding variable / column.
- `ylab`: the label of the y axis (in quotes). Otherwise, R will use the name of the corresponding variable / column.
- `main`: the title of the plot (in quotes). We can also set the title using the function `title`.
- `sub`: the subtitle of the plot (in quotes).
- `xlim`: if set to `c(xmin, xmax)` the range of the x axis is from `xmin` to `xmax`. If not specified, R will determine the ranges of the axes automatically.
- `ylim`: if set to `c(ymin, ymax)` the range of the y axis is from `ymin` to `ymax`.
- `log`: controls which axes should use a logarithmic scale (`""` (default), `"x"`, `"y"`, or `"xy"`).
- `pch`: the plotting symbol (`0-14` for open symbols, `15-20` for solid symbols, `21-25` for filled symbols (fill colour can be set using `bg`), or a character in quotes).

- `lty`: the type of line (`0`=blank, `1`=solid (default), `2`=dashed, `3`=dotted, `4`=dot-dash, `5`=long dash, `6`=two dashes).
- `lwd`: the width of the line.
- `col`: the colour (either a number, a name in quotes, `#` followed by the hex triplet in quotes (as used in HTML, e.g. `"#ffff00"` for yellow), or the output of the functions such as `rgb`, `hsv`, `gray`/`grey`, `rainbow`). The function `palette` can be used to change how integers map to colours.
- `cex`: the size of the plotting symbol.

If the arguments `col`, `pch`, and `cex` are set to a single value, this value is applied to all points. If they are set to a vector of the same length as the the data points, a different colour / plotting symbol / size is used for each point.

Example 1 *In this example we will plot the health expenditure data which we looked at in the video. It is contained in the RData file*

```
load(url("http://www.stats.gla.ac.uk/~levers/rp/w5.RData"))
head(health)
```

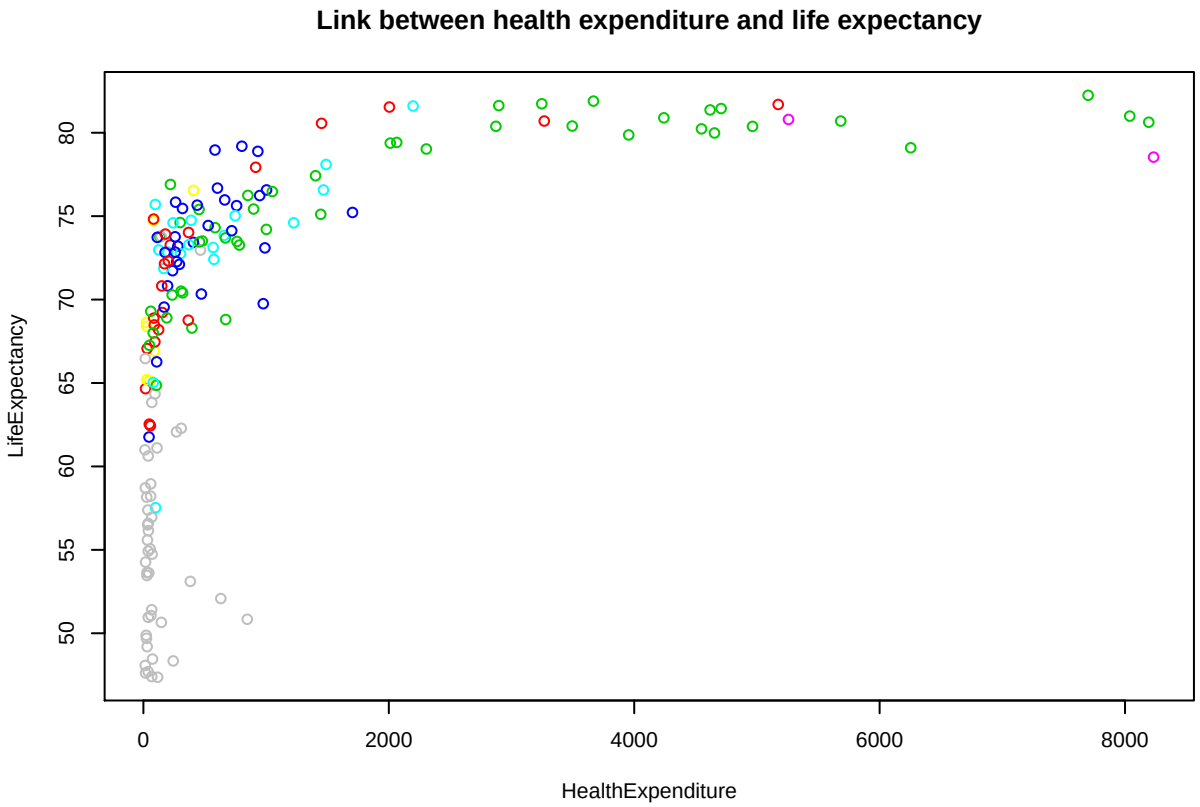
```
##     Country                      Region Year Population LifeExpectancy
## 1   Albania      Europe & Central Asia 2010    3204284       76.90095
## 2   Algeria Middle East & North Africa 2010   35468208       72.85254
## 3    Angola         Sub-Saharan Africa 2010   19081912       50.65366
## 4 Argentina  Latin America & Caribbean 2010   40412376       75.63215
## 5   Armenia      Europe & Central Asia 2010    3092072       73.78356
## 6 Australia        East Asia & Pacific 2010   22065300       81.69512
##   HealthExpenditure
## 1          220.2286
## 2          198.1556
## 3          146.1099
## 4          759.2994
## 5          133.7856
## 6         5173.5024
```

We can create a scatter plot of life expectancy against health expenditure using

```
plot(LifeExpectancy~HealthExpenditure, data=health)
```

We can now use `col` *to use colour to denote the geographical region. However,* `Region` *is a factor, so we first need to convert it to integers (which R accepts as colours). We can do this using the function* `unclass` *– we add 1 to the result, otherwise one group would have their points drawn in black (R's first choice of colour).*

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
title("Link between health expenditure and life expectancy")
```



Next we should add a legend to the plot but we will look at this later on.

Task 1 *The data set* `diamonds` *contains the prices and other attributes of almost 54,000 diamonds.*

```
diamonds <- read.csv("diamonds.csv")
head(diamonds)
```

```
##   carat       cut color clarity depth table price    x    y    z
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
## 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

Create a scatter plot of carat against price, using different colours to denote the different colour and different plotting symbols to denote the different cuts.

Plotting objects

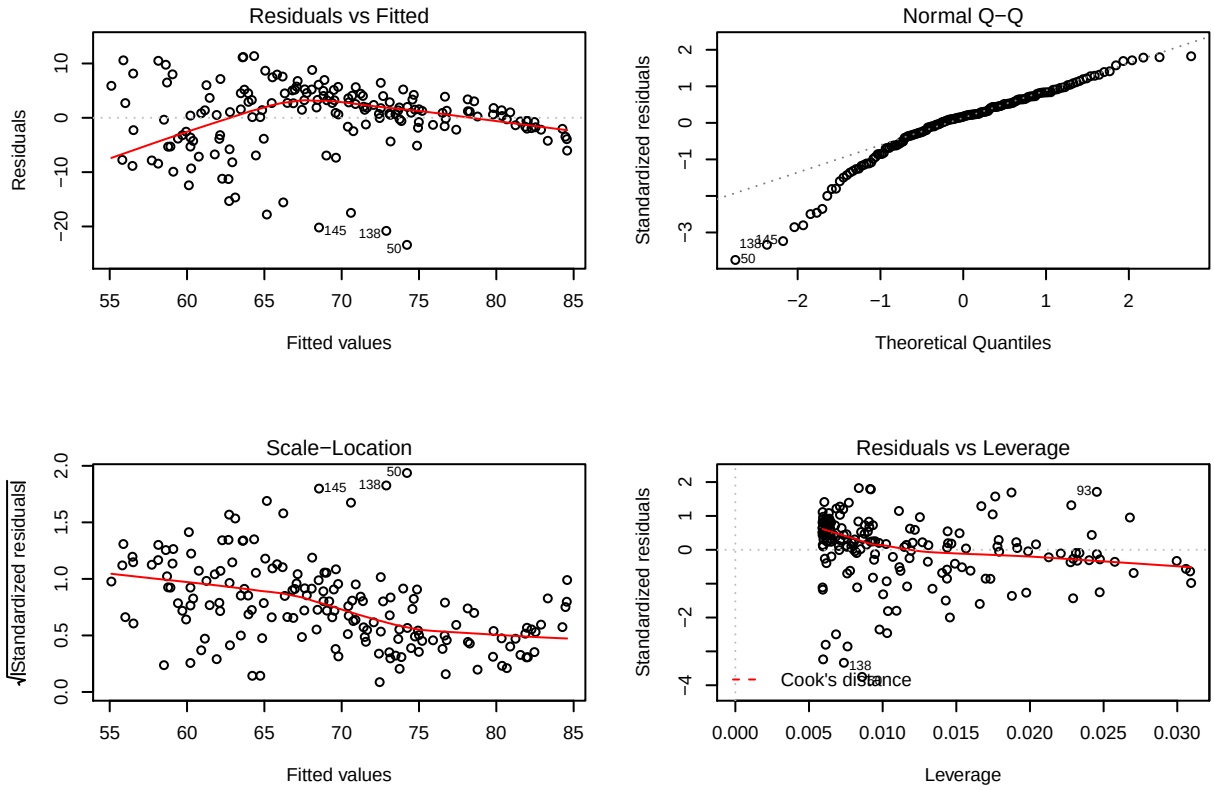
Many R objects (such as model fits) have a `plot` method, which draws a visualisation of or diagnostic check relating to this object.

Example 2 *Suppose we fit a linear regression model to the* `health` *data from the example:*

```
model <- lm(LifeExpectancy ~ log(HealthExpenditure), data=health)
```

The `plot` *method for linear model objects then produces a series of diagnostic plots.*

```
plot(model)
```



You will learn more about linear regression models and also how to interpret these diagnostic plots in other modules, here this is just to demonstrate that the output of the function `plot` *depends on the object plotted.*

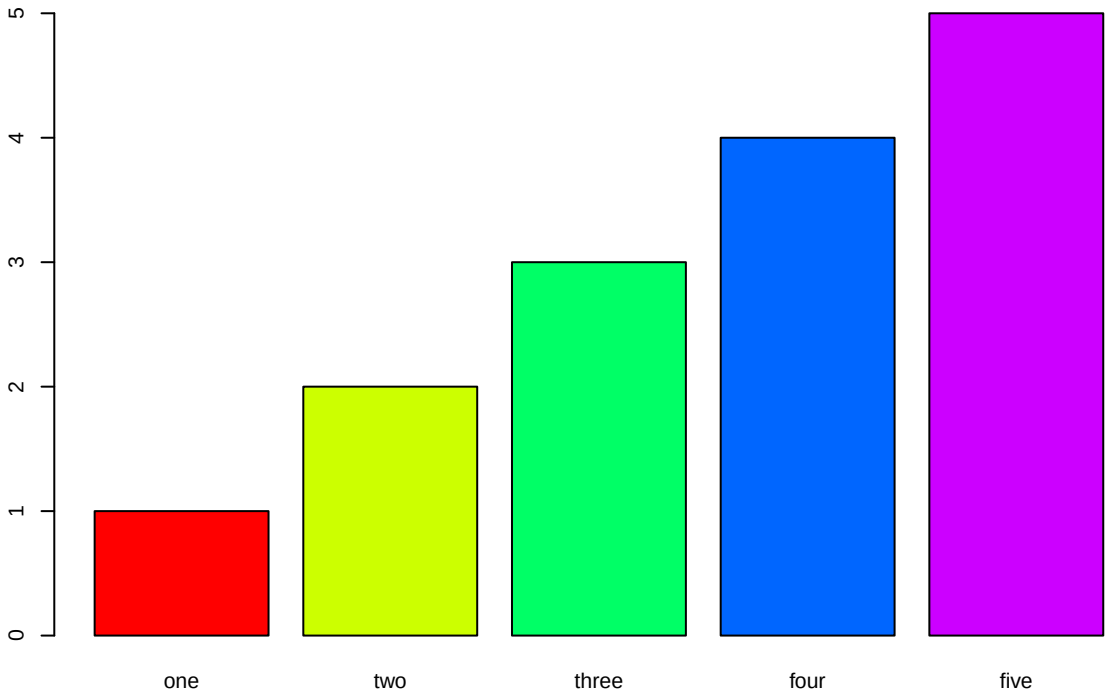
High-level functions for statistical plots

Bar charts and pie charts

The function `barplot(height, ...)` can be used to create bar plots. The vector `height` hereby contains the heights of the bars.

Example 3

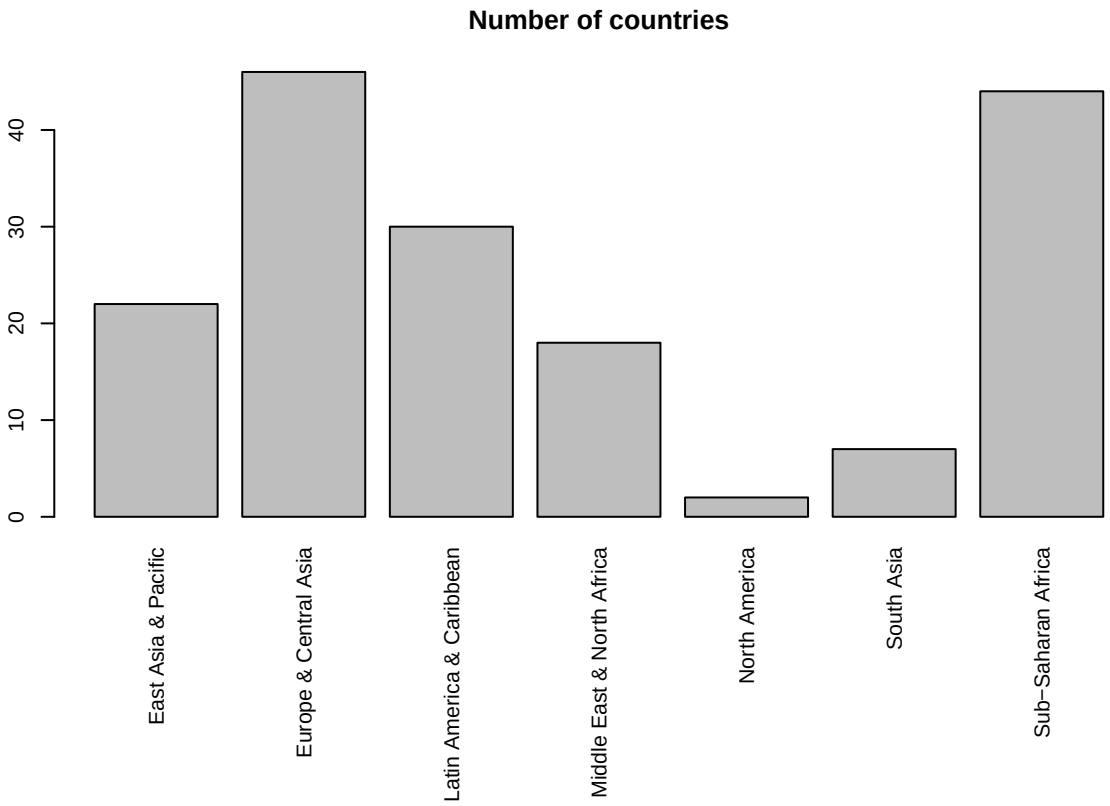
```
height <- 1:5
names(height) = c("one", "two", "three", "four", "five")
barplot(height, col=rainbow(5))
```



If we want to use the function `barplot` to chart frequencies of categorical variables, we first need to tabulate these using the function `table`.

Example 4 *We can create a barchart of the number of countries in each region using*

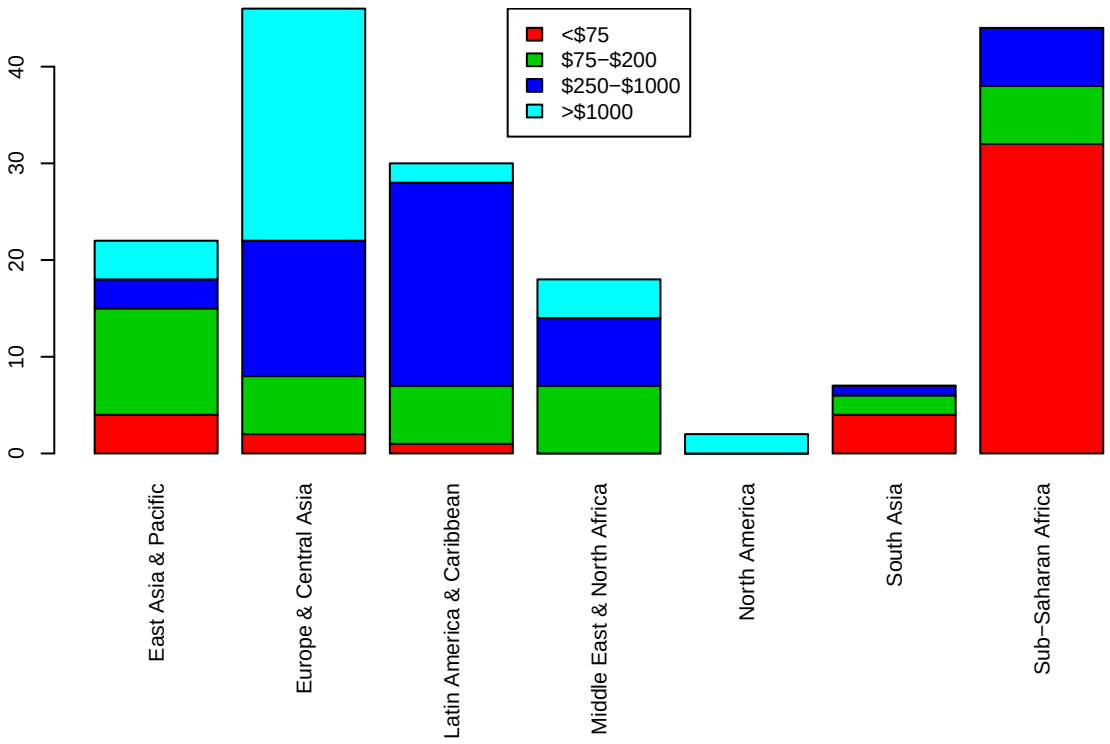
```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))                    # Make space for labels
barplot(table(health$Region))
title("Number of countries")
```



If the argument `height` to `barplot` is a matrix the bars are shown in groups. If we use the additional argument `beside=FALSE`, the bars are stacked.

Example 5 *Let's use this to illustrate the distribution of health expenditure in the different regions. We start by discretising the health expenditure and then plotting the number of countries for each level and region.*

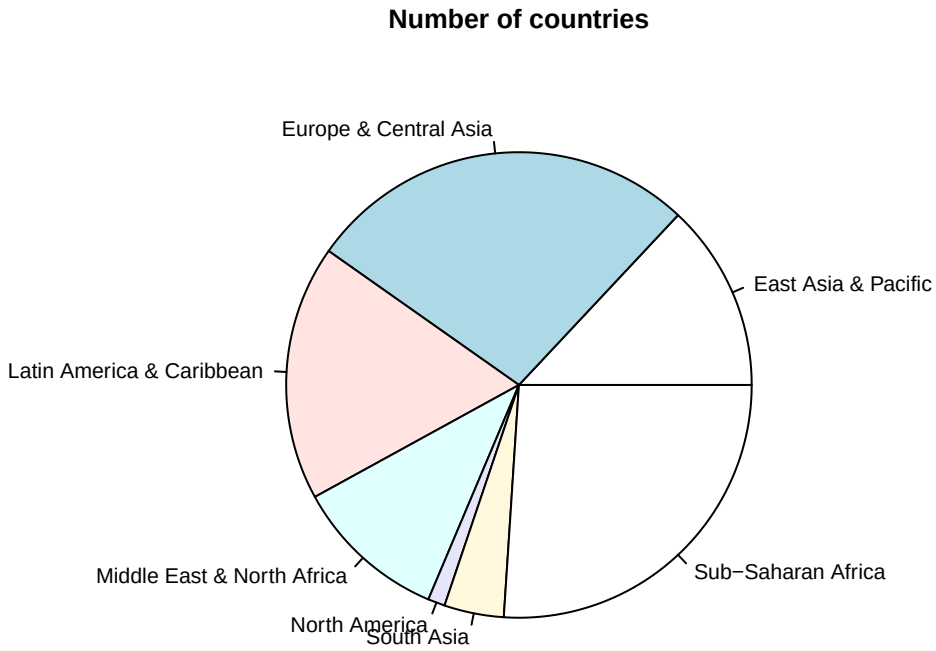
```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))                    # Increase space for labels
# Discretisation of Expenditure
health <- transform(health, DiscExp = cut(HealthExpenditure,breaks =
          c(0,75,250,1000,10000),labels =  c("<$75","$75-$250","$250-$1000","$>1000")))
# Plotting of the barplot
barplot(table(health$DiscExp,health$Region),col=2:5)
# Here  we also add a legend for illustration
legend("top", fill=2:5, c("<$75","$75-$200","$250-$1000",">$1000"))
```



We can use the function `pie(x,...)` to draw a pie chart of the proportions given by the vector `x` (which will be renormalised, if necessary).

Example 6 *We can draw pie chart showing the proportions of countries which comes from each region using the following R code.*

```
pie(table(health$Region))
title("Number of countries")
```



Note that pie charts are not ideal: the human eye is better at judging lengths than angles and pie charts are essentially just about judging angles.

Boxplots

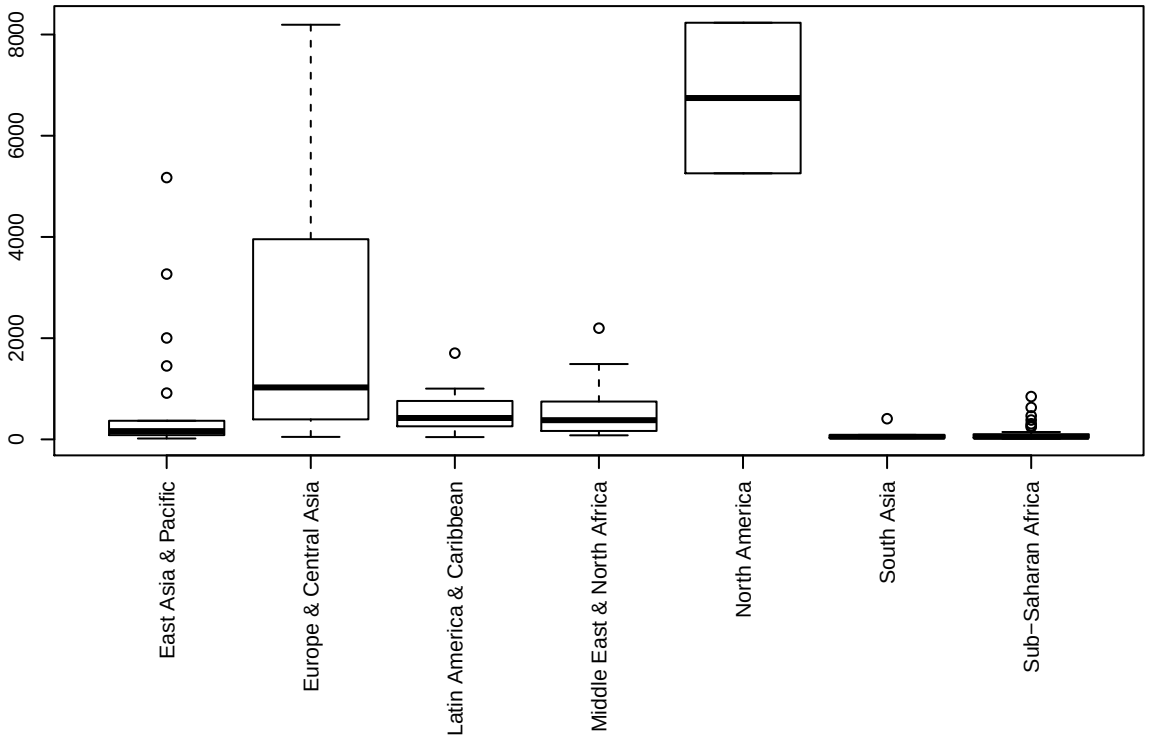
Boxplots can be created using the function `boxplot`.

`boxplot(y, ...)` creates a box plot of the data in the vector `y`. If `y` is a data frame than R will draw one box plot per column (using a common y axis).

If `x` is a categorical variable then `boxplot(y~x, ...)` draws a boxplots separately for each level of `x`.

Example 8 *We draw boxplots showing the distribution of the health expenditure in each region using the following R code.*

```
par(las=3, mar=c(12.1, 4.1, 4.1, 2.1))                    # Increase space for labels
boxplot(health$HealthExpenditure ~ health$Region)
```



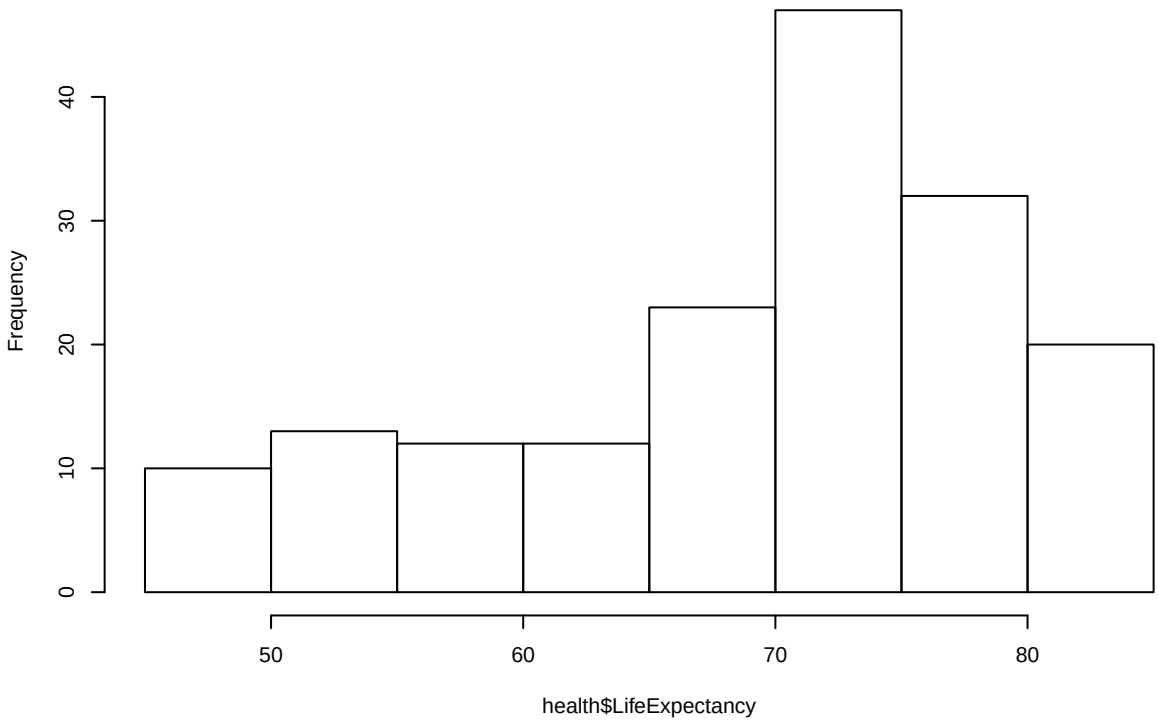
Task 2 *Let's return to the diamonds data from Task 1. Create boxplots of the prices of diamonds as a function of cut and colour.*

Histograms and density plots

Histograms can be created using the function `hist(x, ...)`.

Example 9 *A histogram of health expenditure can be created using*

```
hist(health$LifeExpectancy)
```

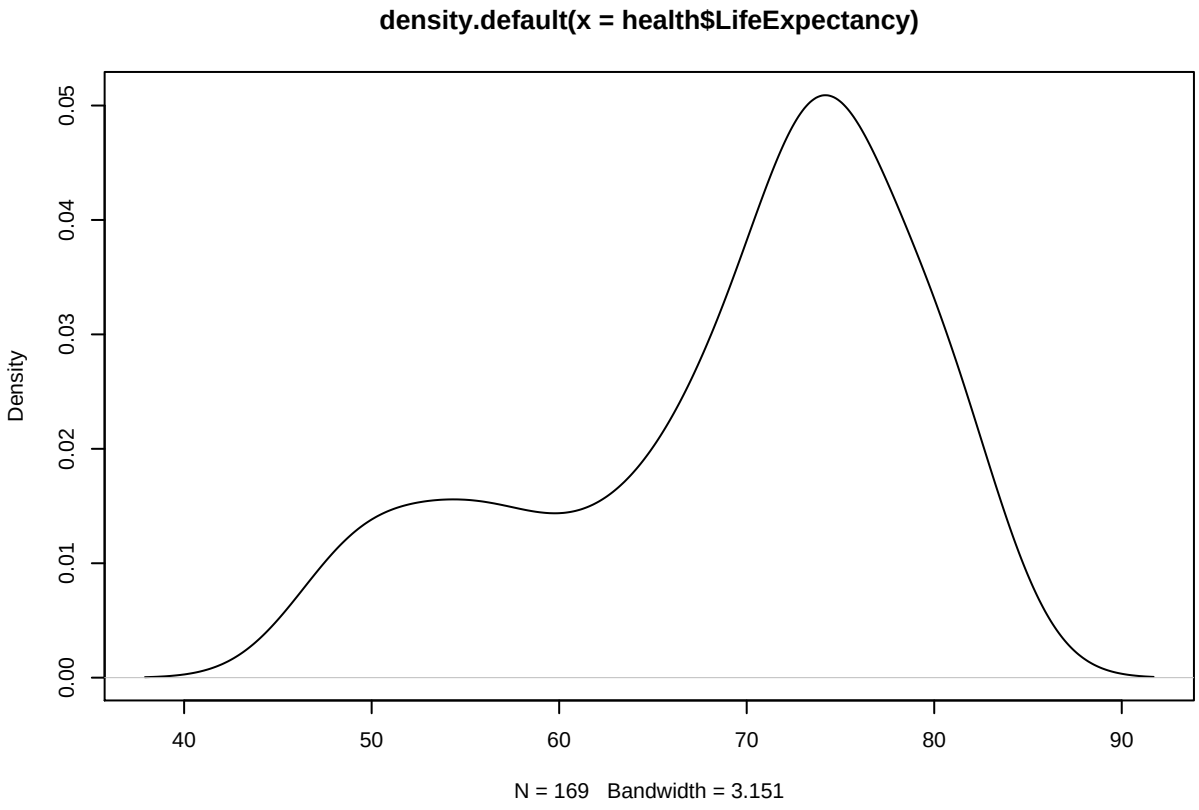


R automatically chooses the breakpoints. You can use the argument `breaks` to override it: `breaks=n` forces R to use `n` breakpoints, alternatively you can set `breaks=vec`, where `vec` is a vector containing the breakpoints to be used.

A histogram is a discrete approximation to the probability density function. You can estimate the probability density function using the function `density`:

Example 10 *A plot of the (estimated) density of health expenditure can be created using*

```
plot(density(health$LifeExpectancy))
```



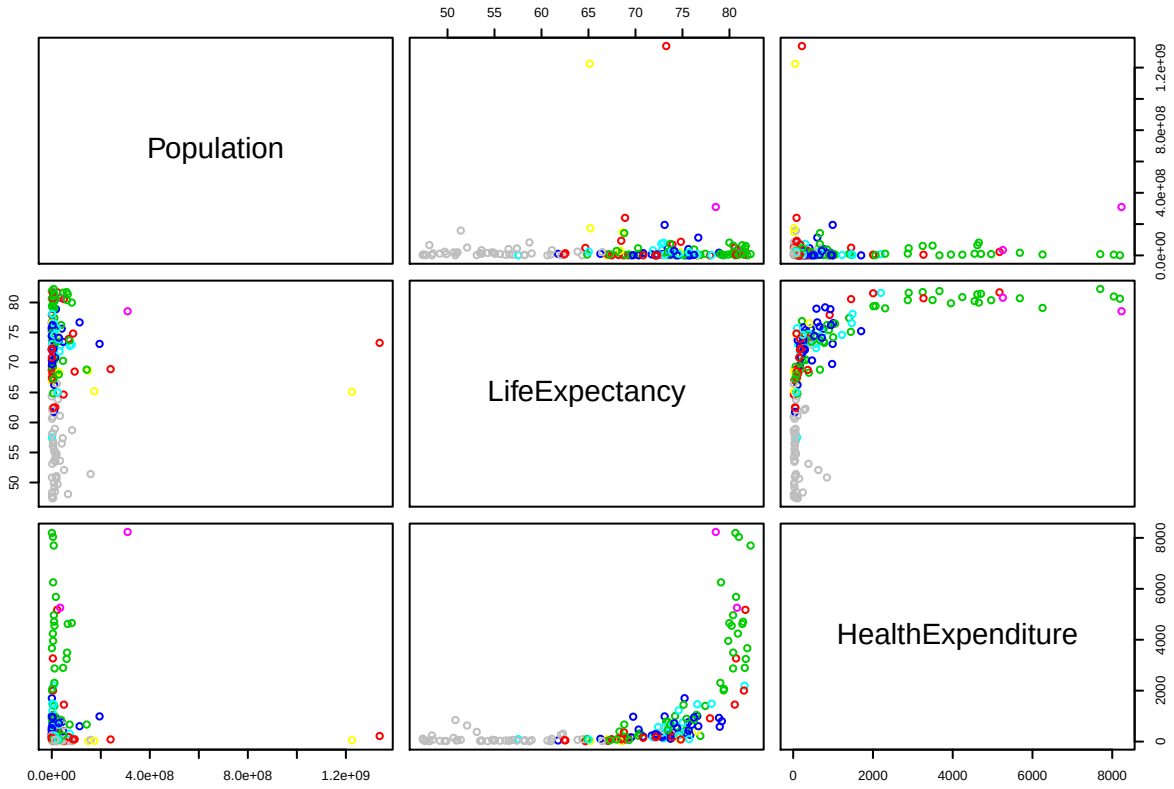
Scatterplot matrices

We can use the function `plot` to create a scatterplot of two continuous variables. Often, datasets contain more than just two continuous variables. The function `pairs` draws a scatterplot matrix. It contains a scatter plot of every variable against every other variable.

Colour (and plotting symbols, ...) can be set in the same way as for plot.

Example 11 *The fourth to sixth column of the health expenditure data contain continuous data, which we can visualise using a scatterplot matrix.*

```
pairs(health[,4:6],
      col=1+unclass(health$Region))
```

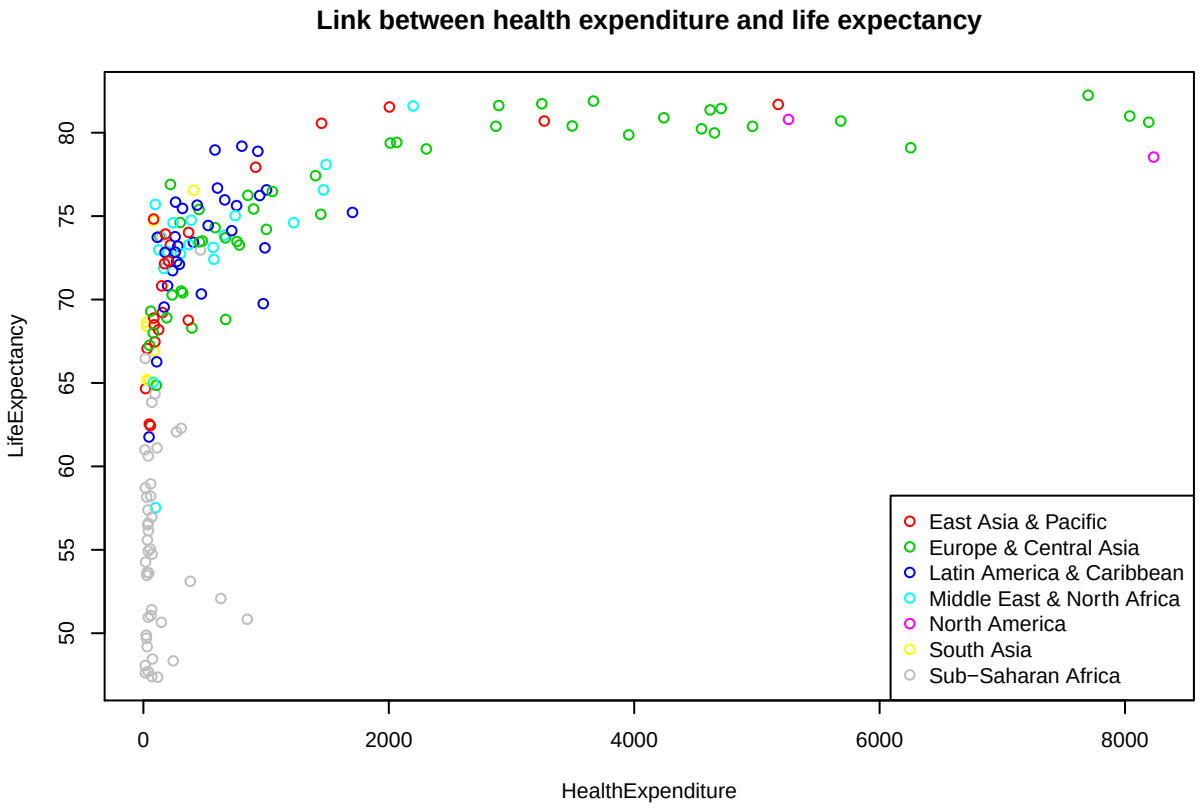


Legends

The function `legend(position, type=values, legend=legend)` can be used to add a legend to a plot. More than one `type=values` expression can be used. `legend` is the vector containing the labels to be used in the legend. `position` can be `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`. Alternatively you can specify the coordinates of the legend.

Example 12 *We will now add a legend to our plot of the health expenditure from Task 1.*

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
legend("bottomright",  pch=1, col=1+1:nlevels(health$Region), legend=levels(health$Region))
title("Link between health expenditure and life expectancy")
```



`pch=1` *is needed to make sure that R uses the standard plotting symbol in the legend.*

Low-level plotting functions

Adding to plots using points and lines

The function `lines` and `points` can be used to add lines or points to an existing plot. `lines` behaves like `plot` using `type="l"`, and `points` behaves like `plot` using `type="p"`, except that the points/lines are added to the active plot, rather than a new plot. `points` and `lines` can be used the same way as `plot` (except for the arguments `title`, `sub`, `xlim`, `ylim`, `log`, `type`, which cannot be used).

Example 13 *Suppose we want to highlight the observations belonging to the UK, the US and Australia in the plot of health expenditure and life expectancy.*

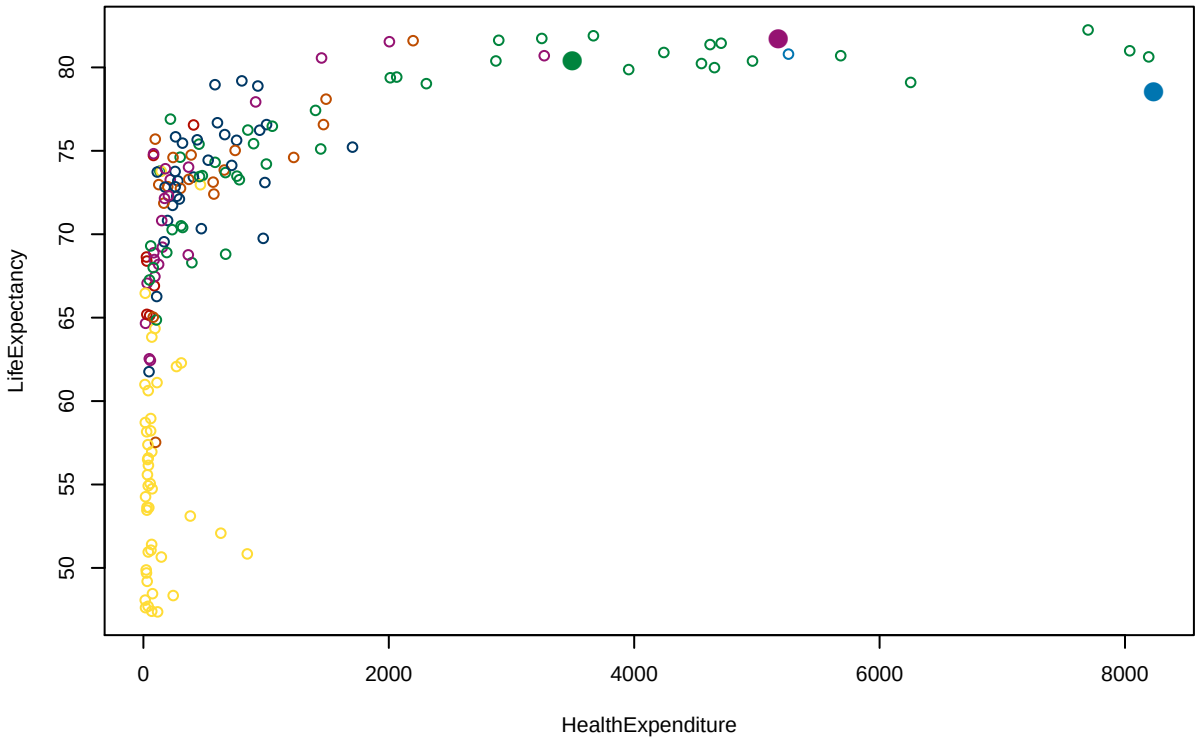
We can extract the data belonging to these three countries . . .

```
health2 <- subset(health, Country == "Australia" | Country == "United Kingdom"
                  | Country == "United States")
health2
```

```
##            Country                Region Year Population LifeExpectancy
## 6        Australia   East Asia & Pacific 2010   22065300       81.69512
## 161 United Kingdom Europe & Central Asia 2010   62262786       80.40244
## 162  United States         North America 2010  309349689       78.54146
##     HealthExpenditure DiscExp
## 6            5173.502  $>1000
## 161          3494.731  $>1000
## 162          8232.875  $>1000
```

... and we can then draw the points for these three countries with a filled circle (`pch=16`*) and twice the size (*`cex=2`*):*

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
points(health2$HealthExpenditure, health2$LifeExpectancy, col=1+unclass(health2$Region),
                        pch=16, cex=2)
```



Task 3 *Consider two vectors* `x` *and* `y` *created using*

```
n <- 1e3
x <- runif(n, 0, 2*pi)                  # x is random uniform from (0,2*pi)
x <- sort(x)                            # Sort entries in x to avoid a mess
y <- sin(x)                             # Set y to the sine of x
y.noisy <- y + .25 * rnorm(n)           # Create noisy version of y
```

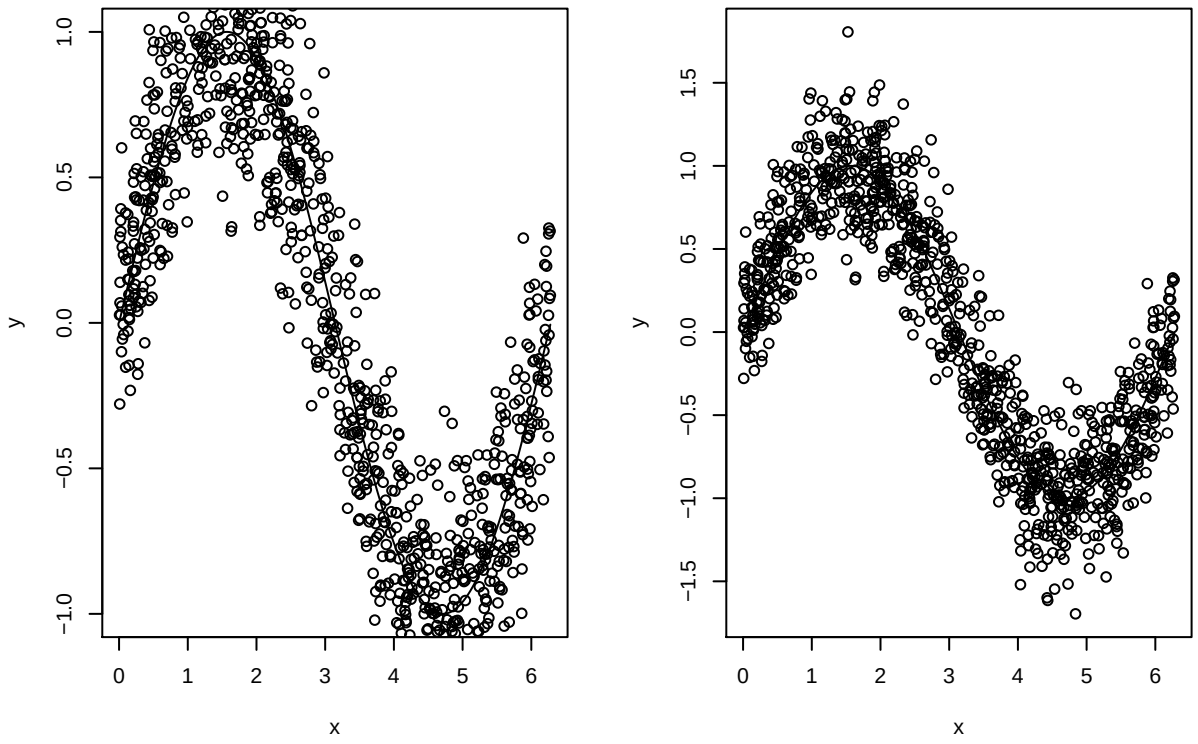
Consider the following two blocks of code:

```
plot(x, y.noisy, ylab="y")
lines(x, y, col=2)
```

and

```
plot(x, y, type="l", col=2)
points(x, y.noisy)
```

The two plots generated are shown below (in arbitrary order).



What is the difference between the two commands? Why do the plots looks different? Which plot comes from which command?

The function `abline` can be used to add a straight line to an existing plot:

- `abline(h=ypos, ...)` draws a horizontal line at `ypos`.
- `abline(v=xpos, ...)` draws a vertical line at `xpos`.
- `abline(a=intercept, b=slope, ...)` draws a line with `intercept` as its intercept and `slope` as its slope.

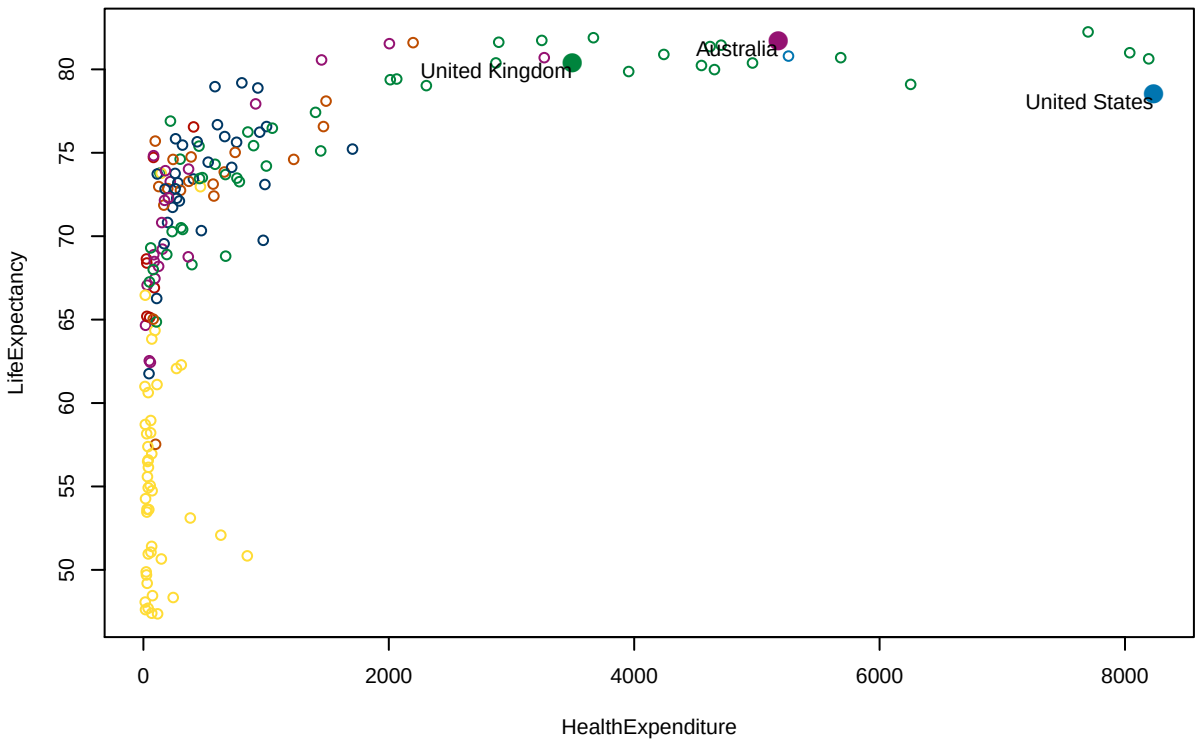
`col`, `lwd` and `lty` can be used as additional arguments.

Adding text

The function `text(x, y, text, ...)` plots the text `text` (one character string or a vector of strings) at the coordinate(s) (`x`, `y`). The optional arguments include `col, cex`, and `adj=c(horiz, vert)`, which sets the horizontal adjustment to `horiz` (`0`: left justified, `0.5` centred, `1`: right justified) and the vertical adjustment to `vert` (`0`: bottom, `0.5` centre, `1`: top). The defauilt is all centered `c(0.5, 0.5)`.

Example 14 *Suppose now want to label the observations belonging to the US, UK and Australia in the plot from Task 13.*

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region))
points(health2$HealthExpenditure, health2$LifeExpectancy, col=1+unclass(health2$Region),
                            cex=2, pch=16)
text(health2$HealthExpenditure, health2$LifeExpectancy, health2$Country, adj=c(1,1))
```



Drawing rectangles and polygons

The functions `rect` and `polygon` can be used to draw filled rectangles and polygons.

`rect(xleft, ybottom, xright, ytop, ...)` draws a rectangle with bottom left corner (`xleft`, `ybottom`) and top right corner (`xright`, `ytop`).

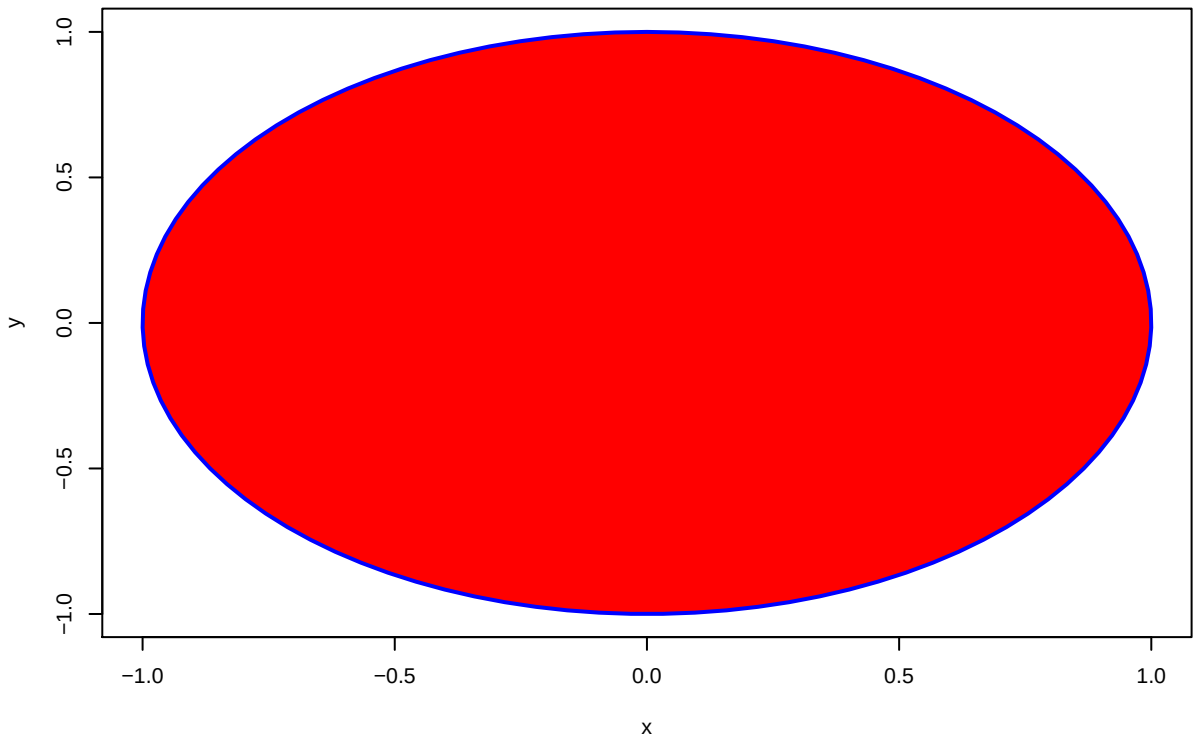
`polygon(x, y, ...)` draws a polygon with vertices in (`x`, `y`).

The following optional arguments can be used:

- `border`: the colour of the outline. Use `border=NA` to draw no outline.
- `lwd` / `lty`: the line width / line type of the outline.
- `col`: the colour used for filling the polygon. Use `col=NA` for a transparent rectangle / polygon (i.e. with no fill).
- `density`: the density of shading lines (defaults to `NULL`, i.e no shading lines, but a solid fill is used).

Example 15 *The following R code draws a red circle of radius 1 having a blue outline. Note the use of* `plot` *with argument* `type=n` *to set up the plot (coordinate axes, etc.).*

```
t <- seq(0, 2*pi, length.out=100)
circle <- cbind(sin(t), cos(t))
plot(circle, type="n", xlab="x", ylab="y")
polygon(circle, col="red", border="blue", lwd=2)
```

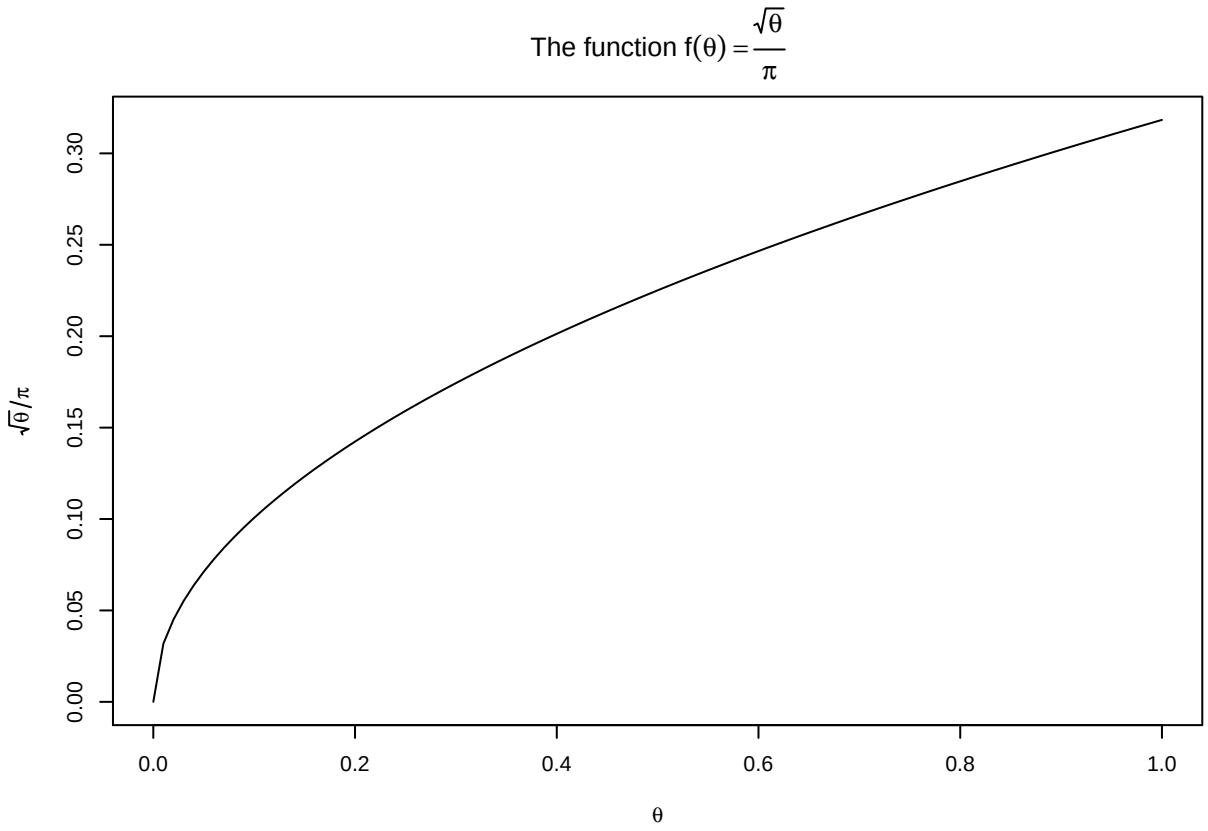


The circle looks more like an oval. In order to obtain a plot with perfectly equal scales (so that the circle looks like a circle) one can use the function `eqscplot` *from the package* `MASS` *instead of* `plot`.

Mathematical notation in plots

The text arguments used to display text in graphics (`xlab`, `ylab`,`main`, `sub`, ...) and the function `text` can be used as well to typeset mathematical formulae in a TeX-like manner. The formulae are not enclosed in quotation marks, but given as an argument to the functions `quote` or `expression`, as the following example shows:

```
theta <- seq(0, 1, by=0.01)
plot(theta, sqrt(theta)/pi, type="l",
     xlab=quote(theta), ylab=quote(sqrt(theta)/pi),
     main=quote("The function "*f(theta)==frac(sqrt(theta),pi)))
```



Note that `*` is used to juxtapose two expressions. `==` gives a = `frac(,)` gives $\frac{x}{y}$. Use the latin transliteration of Greek letters (e.g. `alpha`). See `?plotmath` for details.

3D and image plots

The functions `persp(x, y, M, ...)`, `image(x, y, M, ...)`, `contour(x, y, M, ...)`, and `filled.contour(x, y, M, ...)` can be used to visualise functions $f(x, y)$ of two variable x and y. The functions are also useful when visualising spatial or map data.

The two (optional) arguments and are vectors containing the different values of `x` and `y` respectively. `M` is a matrix containing the values of f, such that

$$M_{ij} = f(x_i, y_j)$$

In other words, these functions need the data in "wide" format.

Example 16 *In this example we will plot the probability density function of the two-dimensional standard normal distribution*

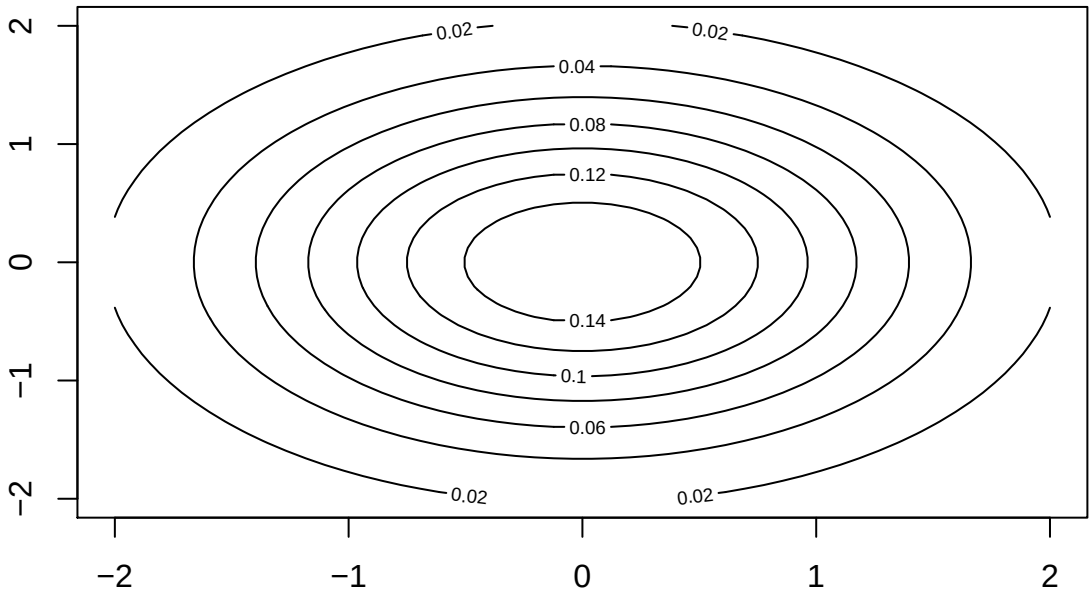
$$f(x, y) = \phi(x) \cdot \phi(y),$$

which is the product of the probability density function of the univariate standard normal distribution (as X and Y are independent).

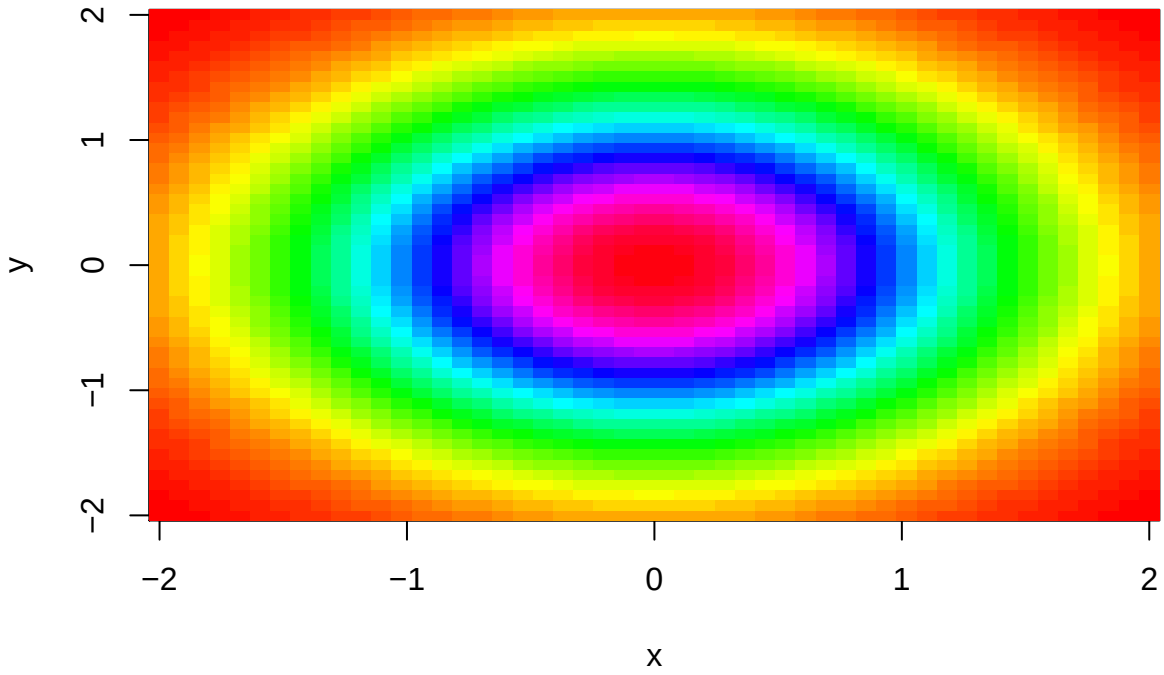
We need to evaluate the function $f(x, y)$ for all combinations of input values x_i and y_j. The following code does this, by using two `for` *cycles (we discuss these next week):*

```
x <- seq(-2, 2, len=50)
y <- seq(-2, 2, len=50)
z <- matrix(nrow=length(x),ncol=length(y))
for (i in 1:length(x))
  for(j in 1:length(y))
    z[i,j] <- dnorm(x[i])*dnorm(y[j])
```

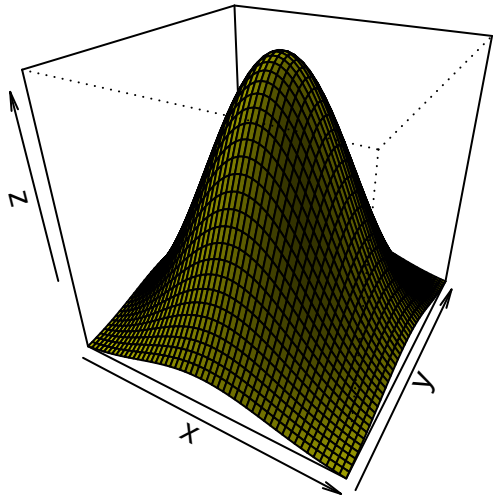
```
contour(x,y,z)
```




```
image(x,y,z, col=rainbow(100))
```



```
persp(x,y,z,theta=30,phi=30,col="yellow",shade=0.5)
```



The arguments `theta` *and* `phi` *of* `persp` *allow for changing the angle from which the surface is viewed.*

Setting plot preferences

The function `par` can be used to customise plots in many ways (see `?par`). We have used `par` already to adjust the margins and to change the orientation of the tickmarks. Another important use of `par` is to put more than one plot onto a figure.

Margins

The margins of the plot can be changed by calling `par(mar=c(bottom, left, top, right))` before plotting.

Arranging plots on a grid

`par(mfrow=c(nrows, ncols))` divides the figure into `nrows` rows and `ncols` columns, which will be used in *row-wise* order, as shown in Figure 1.

`par(mfcol=c(nrows, ncols))` divides the figure into `nrows` rows and `ncols` columns, which will be used in *column-wise* order, as shown in Figure 2.

Task 4 *Create a plot of* $\sin(x)$ *and* $\cos(x)$ *right next to each other for* $x \in (0, 2\pi)$.

Sophisticated arrangement of plots using layout

For more sophisticated arrangements you can use the function `layout`. The first (and possibly only) argument of `layout` is an integer matrix describing how the figure is to be divided. For example,

```
layout.matrix <- rbind(c(1, 1, 2),
                       c(1, 1, 3))
layout(layout.matrix)
```

creates the layout shown in Figure 3.

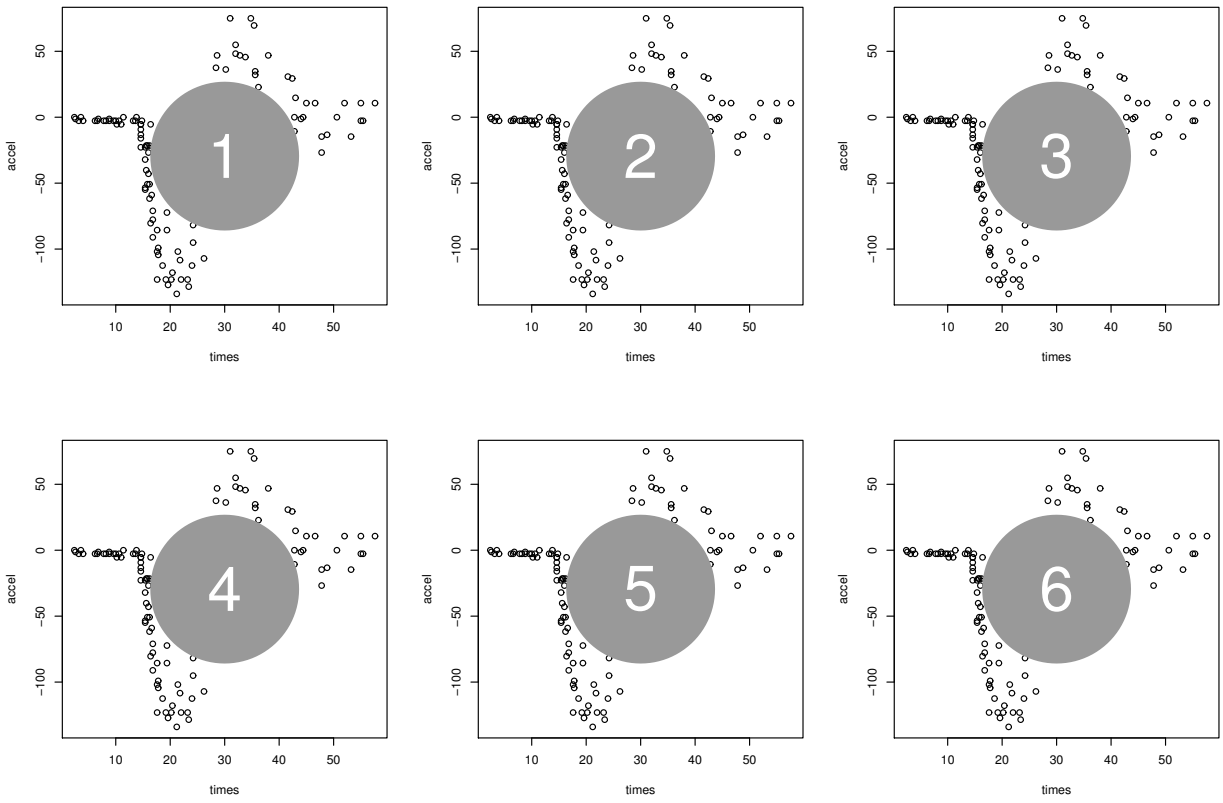


Figure 1: Figure split using `mfrow=c(2,3)`

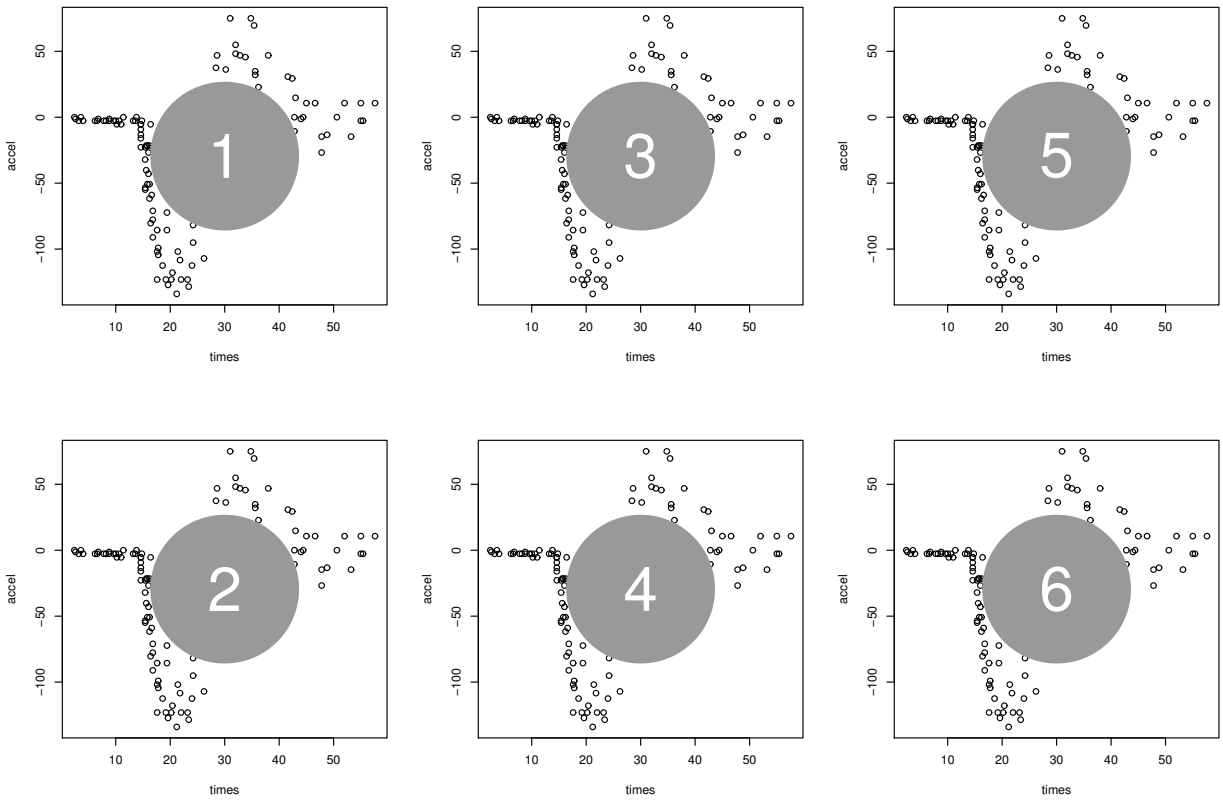


Figure 2: Figure split using `mfcol=c(2,3)`

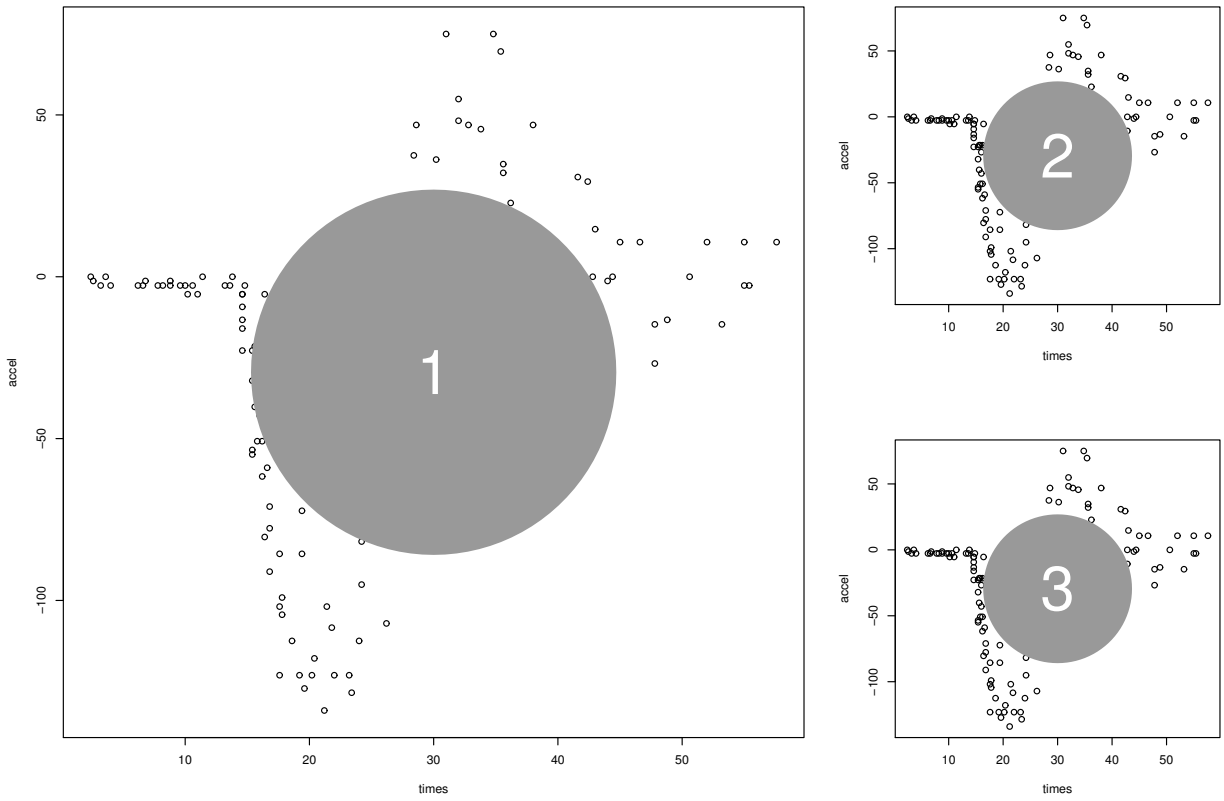


Figure 3: A plot region split using `layout`

The optional arguments `widths` and `heights` can be used to change the widths of the columns and heights of the rows. If they are not specified, each column / row has the same width / height.

Exporting plots

By default R uses the screen as plotting device.

- To copy your graphics from the classical R GUI into another application under Windows, choose *File > Copy to the clipboard* and then *as a Metafile*. You can then paste the graphics into your document (in Word, Powerpoint, ...).
- In RStudio, to copy a plot to the clipboard choose *Export* and then *Copy plot to clipboard ...*. Choosing the *Metafile* option usually gives better quality results. You can also export the plot to a variety of other file formats.

The screen is not the only graphics device under R and, especially if you want to create a large number of plots, it is simpler to use a dedicated graphics device, rather than using the GUI to export the active graphics device. The most important graphics devices are:

- `win.metafile("filename.emf" , width=w, height=h)` creates an enhanced Windows metafile with the dimensions `w`×`h` (not available on Mac or Linux).
- `pdf("filename.pdf", width=w, height=h)` `postscript("filename.ps", width=w, height=h)` create a PDF or PostScript file with the dimensions `w`×`h`.
- `svg("filename.svg", width=w, height=h)` creates an SVG file with the dimensions `w`×`h`.
- `png("filename.png", width=w, height=h)` and `jpeg("filename.jpg", width=w, height=h)` create an PNG / JPEG image of resolution `w`×`h` pixels.

The `w` and `h` arguments are optional.

When using any of the above graphics devices, you have to use `dev.off()` to close the device once you have finished the plot, otherwise the file remains open (and most likely unfinished).

Additional libraries

The following short list includes some R packages which provide advanced plotting functions. These are not examinable and are included here just as a reference.

ggplot2

The package ggplot2 is by far the most popular R package for graphics. It provides a declarative and simple, yet powerful interface for creating sophisticated graphics.

lattice

The package lattice used to be popular before the advent of ggplot. It provides "Trellis"-like high-level plot functions (named after the plotting library for S-Plus) and has inspired some of the functionality of ggplot.

3D plots using rgl

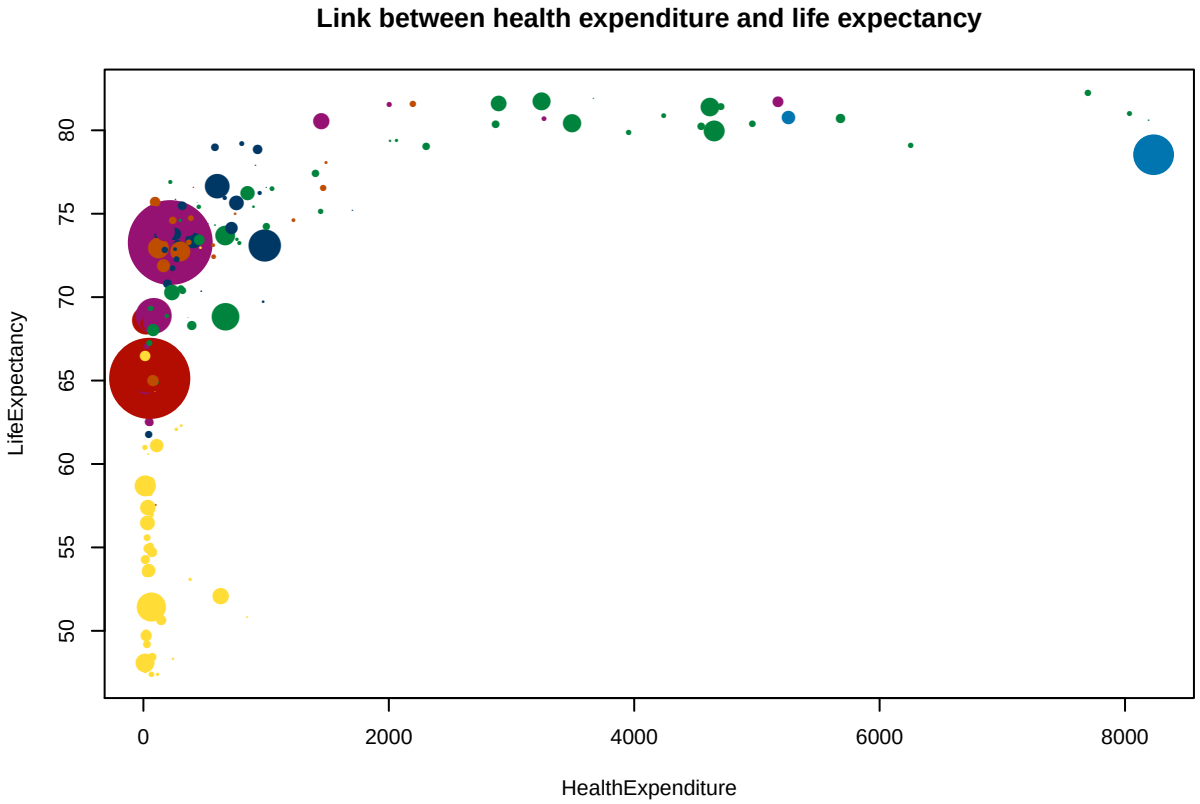
The package rgl allows for creating more sophisticated 3D plots using OpenGL. OpenGL is a cross-platform 3D graphics rendering engine like DirectX under Windows. OpenGL is highly sophisticated: it supports

advanced features like translucent objects, textured surfaces, light effects and even reflective surfaces. The package rgl allows rendering 3D plots through OpenGL. `rgl` plots can be spun using your mouse.

More examples

Example 17 *In the video you saw that it is possible to rescale the points of a plot according to the value of a third variable. Namely, the video included the scatterplot of* `LifeExpectancy` *and* `HealthExpenditure` *of the* `health` *dataset where the size of the points reflected the overall population of the associated country. One possible solution to produce such a plot is by using the* `cex` *option.*

```
plot(LifeExpectancy~HealthExpenditure, data=health, col=1+unclass(Region),
     cex=sqrt(Population)/mean(sqrt(Population)), pch=16)
title("Link between health expenditure and life expectancy")
```



Such plots can be much more easily produced using `ggplot`*, but this goes beyond the scope of the module.*

Example 18 *From the Moodle page you download the dataset* ***hp****, which contains the median house price for all regions of the UK for the periods from 1996 to 2016.*

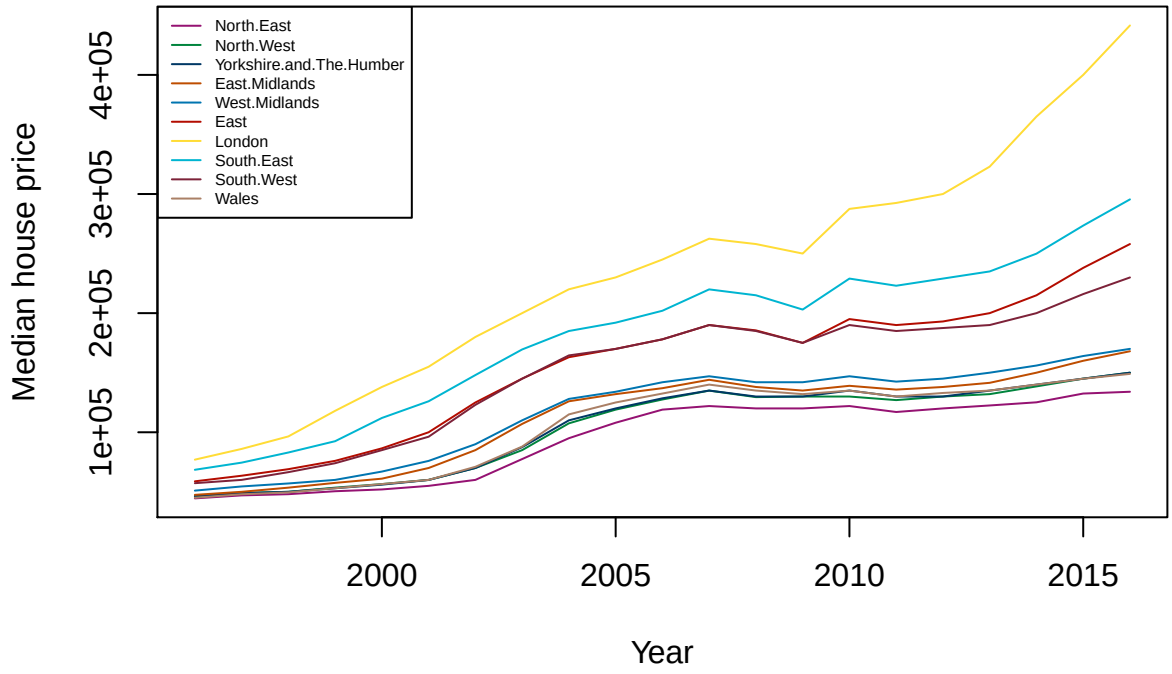
```
hp <- read.csv("hp.csv")
head(hp)
```

```
##   Year North.East North.West Yorkshire.and.The.Humber East.Midlands
## 1 1996      44500      46250                    46500         47450
## 2 1997      46995      49000                    48950         50000
## 3 1998      48000      50000                    50000         53500
## 4 1999      50495      53500                    53000         57500
## 5 2000      52000      56500                    56000         61044
## 6 2001      55000      60000                    59950         70000
```

```
##   West.Midlands  East London South.East South.West Wales
## 1         51000 58795  77000      68500      57250 45000
## 2         54500 63500  86000      74500      60000 47950
## 3         57000 69000  96500      83000      66500 49500
## 4         60000 76000 118000      92500      74000 53000
## 5         67000 86500 138000     112000      85000 56500
## 6         75950 99995 155000     126000      96250 60000
```

The data is in wide format. If we want to plot a line for each region. we first need to set up a plotting region that is large enough. We use the common trick to plot `NULL`*, which does not draw anything (we'll draw the lines later on). We next add the lines correspondong to each region one by one. We will use a loop (we'll cover loops in more detail next week).*

```
plot(NULL, xlim=range(hp$Year), ylim=range(hp[,-1]),
     xlab="Year", ylab="Median house price")
for (i in 2:ncol(hp)) {lines(hp$Year, hp[,i], col=i)}
legend("topleft", lty=1, col=2:ncol(hp), colnames(hp)[-1],cex=0.5)
```



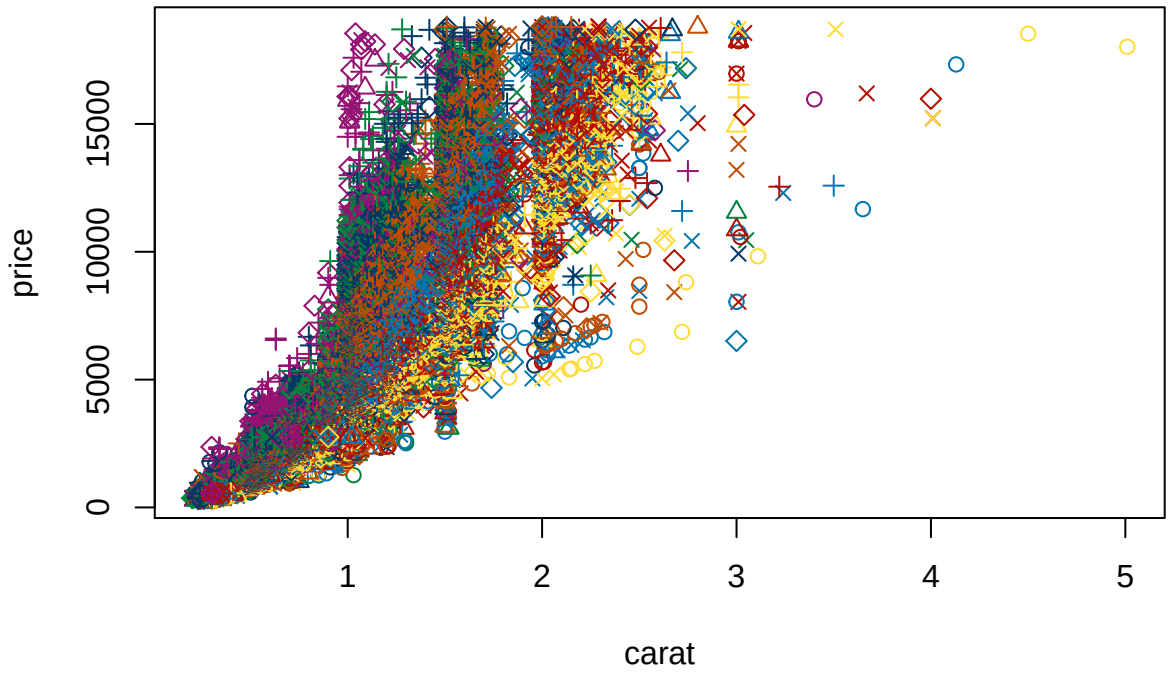
This seemed rather complicated. Luckily enough, there is R function `matplot`*, which plots matrices column by column.*

```
matplot(hp$Year, hp[,-1], type="l", col=2:ncol(hp), lty=1,
        xlab="Year", ylab="Median house price")
legend("topleft", lty=1, col=2:ncol(hp), colnames(hp)[-1])
```

Solutions to the tasks

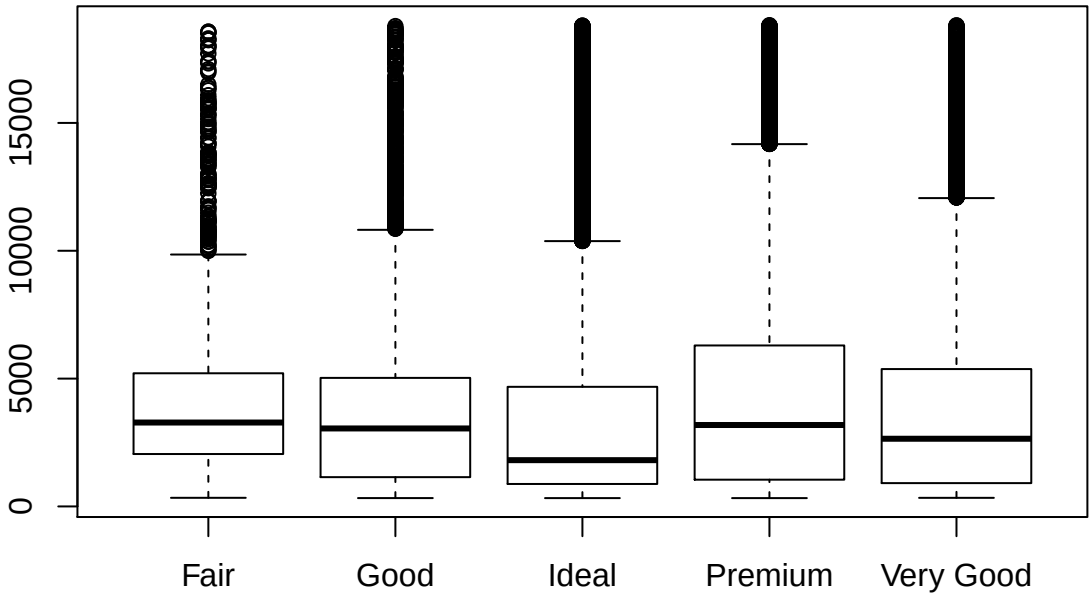
Task 1

```
plot(price~carat, data = diamonds, col=unclass(color)+1, pch=unclass(cut))
```

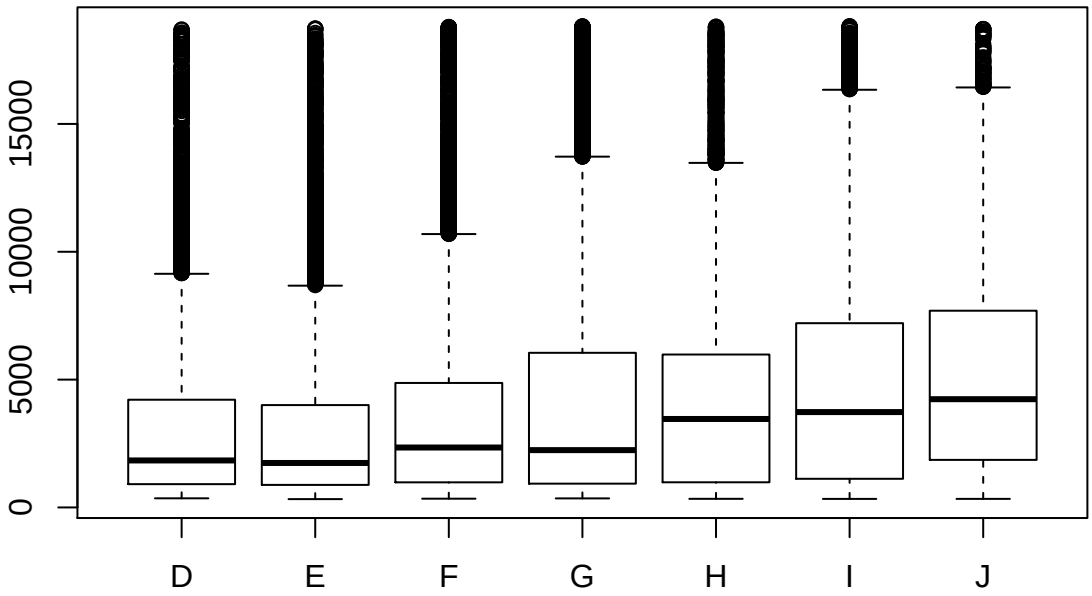


Task 2

```
boxplot(diamonds$price~diamonds$cut)
```



```
boxplot(diamonds$price~diamonds$color)
```



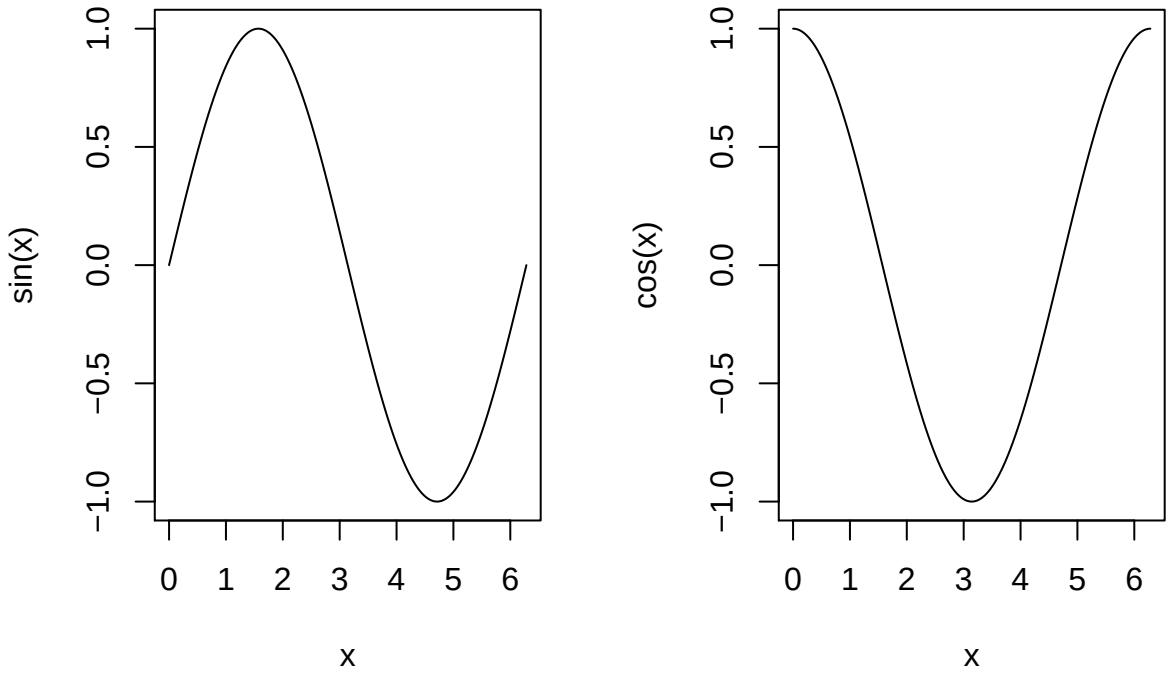
Task 3 The former set of commands first draws the noisy data and then adds the sine curve. The latter set of commands first draws the sine curve and then the noisy data. The noisy data have a greater range than the noise-free data. As `plot` determines the range of the y axis and this cannot be changed by `points` or `lines`*, drawing the noisy points first (as done by the former command) gives a larger range of the y axis that can show all the data. Drawing the noise-free line first leads to a smaller range of the y axis, which is then too small to show all the noisy points.

Thus the left-hand plot comes from the latter command (noise-free line first, then oisy points) and the right-hand plot comes from the former command (noisy points first, then noise-free line).

Another difference is that the former command plots the line above the points, whereas the latter plots the points above the line.

Task 4 In this case it does not matter whether we use `mfrow` or `mfcol` as there are only two plots.

```
x <- seq(0,2*pi,length.out = 1000)
par(mfrow=c(1,2))
plot(x,sin(x),type="l")
plot(x,cos(x),type="l")
```



The function `curve` makes sketching curves easier.

```
par(mfrow=c(1,2))
curve(sin,from=0,to=2*pi)
curve(cos,from=0,to=2*pi)
```

Control Structures

Logical switches: if statements

If statements

https://youtu.be/1Akkbt1aWxA

Duration: 7m14s

We have already made use of logical expressions when subsetting vectors or matrices:

```
x <- rnorm(10)                    # Generate ten realisation from N(0,1)
x[x<0] <- 0                       # Set all negative x to 0
```

This week we learn control structures which let us perform such operations in a less cryptic way.

```
x <- rnorm(10)                    # Generate ten realisation from N(0,1)
x <- ifelse(x<0, 0, x)            # Set all negative x to 0
```

Before covering the function `ifelse` we start with basic `if` statements. With `if` statements R can be programmed to take entirely different actions under different circumstances. `If` statements are in some way yes/no questions: depending on a condition R will either take one specified action or another one.

The basic form of an `if` statement is:

```
if (condition) {
    statement11
    ...
    statement1m
} else {
    statement21
    ...
    statement2n
}
```

`condition` is a logical expression, i.e. a scalar expression that evaluates to either `TRUE` or `FALSE`. If `condition` evaluates to `TRUE` R will run the statements in the first branch (`statement11` to `statement1m`). If however `condition` evaluates to `FALSE` then R will run the statements in the second branch (`statement21` to `statement2n`). The second `else`-part of the `if` statement is optional. If each branch only consists of a single statement the curly brackets can be omitted.

We can use the logical operators `!`, `&` `&&`, `|` and `||` as well as the functions such as `all`, `any` and `xor` in `condition` to combine logical expressions.

Note that the `condition` of an `if` statement *cannot* be a vector (of length > 1). If we want to carry out a conditional operation on a vector, we need to either subset it, use loops or the `ifelse` function.

It is common to indent the content of the two branches. The R interpreter ignores indentation, however properly indented code is much easier to read (for a human).

Example 1 *The if statement in*

```
x <- 2
if (x==2) {
  print("x is 2")
} else {
  print("x is not 2")
}
```

```
## [1] "x is 2"
```

checks whether `x` *is* `2` *and then prints* `"x is 2"` *on the screen.*

Example 2 *In this example we will set* `y` *to* $\sqrt{x}$ *if* x *is non-negative. Otherwise we set it to* $-\sqrt{-x}$.

```
x <- rnorm(1)
if (x>0) {
  x <- sqrt(x)
} else {
  x <- -sqrt(-x)
}
```

This is equivalent to setting `y` *to* `sign(x)*sqrt(abs(x))`.

Task 1 *Create two variables* `x` *and* `y` *containing one random number each. Use an* `if` *statement to set the smaller of the two variables to the value of the larger variable.*

Loops

For loops

For loops

`https://youtu.be/BivoKnOakVQ`

Duration: 8m38s

In the first tutorial we have looked at the so-called "Babylonian Method" for finding $\sqrt{2}$: the sequence defined by

$$x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}}$$

tends to $\sqrt{2}$ as $n \longrightarrow \infty$ (provided $x_0 > 0$).

In order to approximate $\sqrt{2}$ using R we first set `x` to an initial value, say 1,

```
x <- 1
```

and then had to repeatedly (say 10 times) update `x` using the recursive formula from above.

```
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
```

```
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
```

Loops are a way of doing these "repetitive" steps in a more elegant (and flexible) way:

```
x <- 1
for (i in 1:10)
    x <- x/2 + 1/x
```

The general syntax of `for` loops is

```
for (variable in sequence) {
    statement1
    ...
    statementn
}
```

The `for` loop executes the statements in the body of the loop (`statement1` to `statementn`) once for every element of `sequence`: the first time `variable` is set to the first element of `sequence` and the statements in the body are run using that value of `variable`, the second time `variable` is set to the second element of `sequence` and the statements in the body run using `variable` set to that value, and so on.

If we wish to only iterate one statement we can omit the curly brackets.

Example 3 *A simple example illustrating a* `for` *loop is*

```
for (i in 1:3)
    print(i)
```

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

Example 4 *The sequence we iterate over does not need to consist of numbers (though this is very often the case). We can use*

```
for (day in c("M", "Tu", "W", "Th", "F"))
    print(day)
```

```
## [1] "M"
## [1] "Tu"
## [1] "W"
## [1] "Th"
## [1] "F"
```

Example 5 *We can use a* `for` *loop with an* `if` *statement to set all negative values of a vector* `x` *to 0.*

```
x <- rnorm(10)
for (i in seq_along(x))
    if (x[i]<0)
        x[i] <- 0
x
```

```
##  [1] 0.69561056 1.44757849 0.00000000 2.46028127 0.00000000 0.08524304
##  [7] 0.00000000 0.00000000 0.00000000 0.00000000
```

In this loop we want to iterate over the length of the vector `x`*. We could have used* `1:length(x)`*. However, this will not work if the length of* `x` *is 0.* `1:length(x)` *would then return the sequence* $(1, 0)$*, rather than a sequence of length 0 as would be required. The function* `seq_along(x)` *does exactly the same as* `1:length(x)`*, except that it handles the case of a vector of length zero correctly.*

Example 6 *In this example we will use a* `for` *loop to generate a random sample of size 1000 from the model for an auto-correlated time series:*

$$\begin{aligned} X_1 &\sim \mathsf{N}(0,1) \\ X_i | X_{i-1} = x_{i-1} &\sim \mathsf{N}(0.8x_{i-1}, 0.6^2) \end{aligned}$$

One can show that the second line is equivalent to setting $X_i = 0.8 \cdot X_{i-1} + 0.6 \cdot \epsilon_i$*, where* $\epsilon_i \sim \mathsf{N}(0,1)$

We start with creating an empty vector of the required size and setting its first entry to a random number drawn from the $\mathsf{N}(0,1)$ *distribution:*

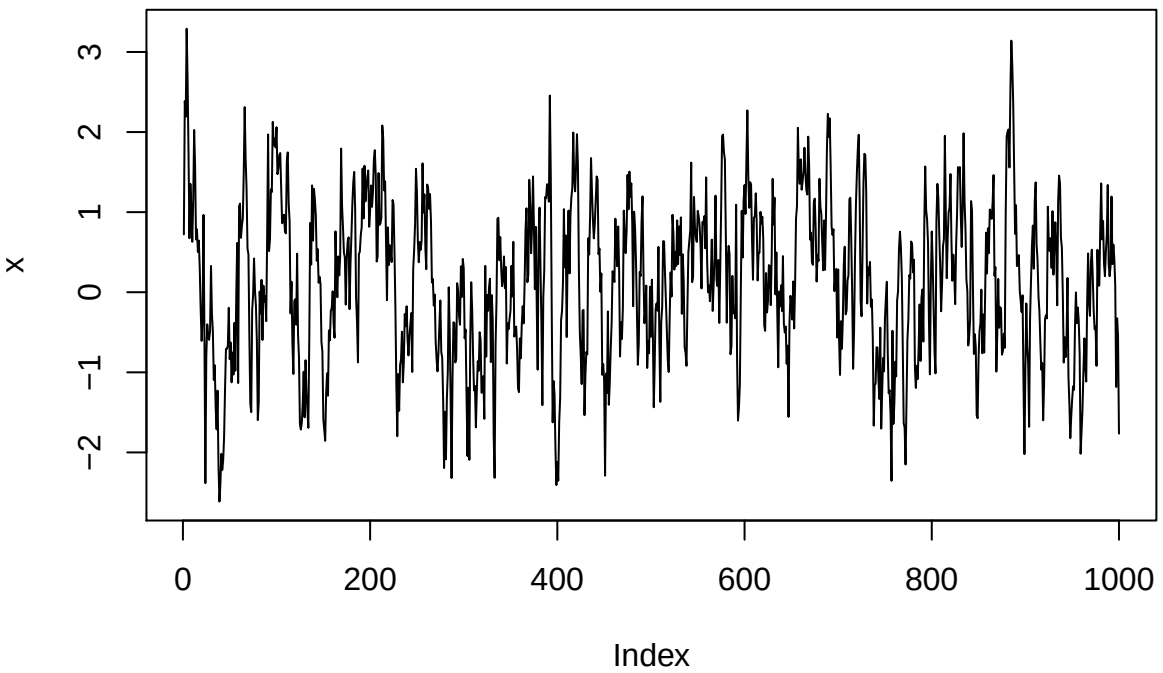
```
n <- 1000
x <- numeric(n)
x[1] <- rnorm(1)
```

We simulate all the remaining entries using a `for` *loop:*

```
for (i in 2:n) {                          # We start the loop at i=2!
    epsilon <- rnorm(1)
    x[i] <- 0.8*x[i-1] + 0.6 * epsilon
}
```

Finally, we can plot the results

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



Task 2 *Create a vector* `x` *containing some missing values using*

```
x <- rnorm(100)                    # Generate white noise
x[sample(100,10)] <- NA            # Sneak in 10 missing values
```

Use a loop to create a vector `y` *which contains the entries of* `x`*, however with missing values replaced by 0. (Remember, you can use the function* `is.na` *to test whether a value is missing.) Can you do the same without*

a loop?

Task 3 *In the setting of the previous task, suppose that rather than setting the missing values to 0 we want to omit them from* `y`*. We can do so by setting*

```
y <- x[!is.na(x)]                    # Only copy non-missing entries
```

Can you modify your loop from Task 2 to achieve the same?

Nested loops

Loops can be nested within each other. Note that you have to use different names for the loop variables in nested loops. In the examples below the outer loop uses `i`, whereas the inner loop uses `j`.

Example 7 *The following simple example illustrate how a nested loop works.*

```
for (i in 1:2)
    for (j in 1:3)
        print(c(i,j))
```

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

The nested loop loops over all combinations of $i \in \{1, 2\}$ *and* $j \in \{1, 2, 3\}$*. The index* `j` *changes fast, whereas the index* `i` *changes slowly.*

Example 8 *When we looked at the standard plotting functions in R we created an image plot of the density of the bivariate normal distribution. We created a matrix* $\mathbf{Z} = (Z_{ij})_{ij}$ *with*

$$Z_{ij} = \phi(x_i) \cdot \phi(y_j)$$

using a nested loop. Let's look at this example again. We start by creating the sequences `x` *and* `y` *and then create an empty matrix to hold the* Z_{ij}*'s.*

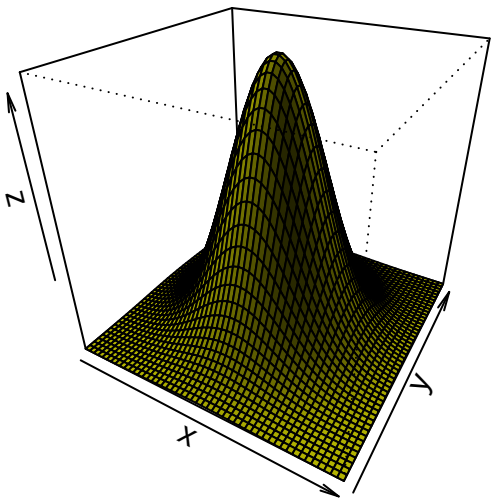
```
x <- seq(from=-3, to=3, length.out=50)      # Create sequence of grid for the x axis
y <- seq(from=-3, to=3, length.out=50)      # Create sequence of grid for the y axis
z <- matrix(nrow=length(x), ncol=length(y)) # Create matrix to store function values
```

In order to set every value of the matrix we need to go through all rows and all columns. Thus we need two loops nested within each other:

```
for (i in seq_along(x))                     # For all rows ...
    for (j in seq_along(y))                 # For all columns ...
        z[i,j] <- dnorm(x[i])*dnorm(y[j])   # Compute f(x,y)
```

Now we can create the plot:

```
persp(x, y, z, theta=30, phi=30, col="yellow", shade=0.5)
```



break and next

A `for`-loop repeats the statements in it a fixed number of times. The `break` statement gives additional flexibility and allows for aborting the loop immediately and before the sequence of indices has been finished. It is typically used inside an `if` statement.

Example 9 *In the motivating example at the beginning of this section we computed* $\sqrt{2}$ *using the Babylonian method, which is based on the iteration* $x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}}$. *We implemented it using a* `for` *loop.*

```
x <- 1
for (i in 1:10)
    x <- x/2 + 1/x
```

After the 10 iterations we obtained $x_{10} = 1.4142135623730949$, *which is quite close to* $\sqrt{2} = 1.4142135623730951$. *Often, we are only interested in the first 8 digits, so we could have stopped the loop earlier, as soon as* `x` *changes by less than say* 10^{-8}. *This can be done using* `break`.

In order to be able to quantify by how much `x` *has changed we need to store the previous value of* `x` *in a variable* `x.old`:

```
x <- 1
for (i in 1:100) {
    x.old <- x                          # Store the old value of x
    x <- x/2 + 1/x                      # Update x
    if (abs(x-x.old)<1e-8)              # Check for convergence
        break
}
```

The `break` *statement will abort the loop as soon as the change in* `x` *is less than* 10^{-8}. *To find out how many iterations were necessary, we can simply print* `i` *after running the loop:*

```
i
```

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

Thus we needed only 5 iterations to obtain $\sqrt{2}$ *to a precision of* $\pm 10^{-8}$.

Task 4 *In Example 9 we have used a variable* `x` *to store the current value of* x *in the recursive sequence. We had to introduce* `x.old` *to store the old value of* `x`, *so that we can compare to the current value in order to check for convergence.*

Instead we could have used a vector `x` *of length 100 and stored the value at the* `i`*-th iteration* x_i *in the* `i`*-th entry of* `x`*. After the end of the loop the required value would then be in* `x[i]`*. Rewrite the loop in that way.*

`next` halts the processing of the current iteration and goes back to the begin of the body of the loop (using the next value of `sequence` in a `for` loop). `next` is the R equivalent of `continue` in C or Java.

Example 10 *The loop*

```
for (x in 1:10) {
    if (x%%2==0)
        next
    print(x)
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
```

prints all the odd numbers: if `x` *is an even number* `x%%2` *is* `0`*, thus* `next` *is called, and thus the remainder of the statements in the body are skipped (before* `x` *is printed) and R continues with the next iteration. Thus R only prints odd numbers.*

In nested loops, `break` and `next` only affect the inner-most loops. R does not support `break`ing outer loops.

while loops

There are occasions when we need to repeatedly perform some operations, but we do not know in advance for how many times. As we have just seen, we can use `break` to stop the loop early. Another option it to use `while` loops (and `repeat` loops, which is the same as `while` loop without a condition).

The syntax of a `while` loop is

```
while (condition) {
    statement1
    ...
    statementn
}
```

The `while` loop checks `condition` each time before executing the first statement in the body of the loop. It executes the loop only if the `condition` evaluates to `TRUE`. As soon as the `condition` evaluates to `FALSE` for the first time, the loop is aborted.

Example 11 *We consider again the Babylonian method for finding* $\sqrt{2}$*. Just like in Example 9, we want to stop iterating as soon as the change in* `x` *is small enough.*

```
x <- 1
x.old <- 0
while(abs(x-x.old)>1e-8) {
    x.old <- x                          # Store the old value of x
    x <- x/2 + 1/x                      # Update x
}
x
```

```
## [1] 1.414214
```

Example 12 *We have seen that we can print the numbers 1 to 3 using a for loop:*

```
for (i in 1:3)
    print(i)
```

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

We can do the same using a `while` *loop, but we have to "manage"* `i` *ourselves.*

```
i <- 1
while (i<=3) {
    print(i)
    i <- i+1
}
```

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

The function ifelse

The function ifelse

https://youtu.be/TylRPCmi6ro

Duration: 6m33s

We have seen that the condition of an `if` statement has to be of length 1. If we want to use a condition that is of length greater than 1, we need to use a loop to go through the vector one-by-one. The function `ifelse` provides a more compact way of doing so. In some way, `ifelse` is the vectorised sibling of `if` statements.

The function

```
result <- ifelse(condition, yes, no)
```

sets the i-th element of the result to the i-th element of `yes` if the i-th element of `condition` is `TRUE`, otherwise it will be set to the i-th element of `no`.

The length of `result` will be the same as the length of `condition`. If `yes` and `no` are shorter than `condition`, `ifelse` will use the standard recycling rules.

Example 13 *We return to the example in which we set the negative entries of a vector* `x` *to 0.*

If we want to use an `if` *statement, we have to use a loop to go through the vector* `x` *one-by-one.*

```
for (i in seq_along(x))
    if (x[i]<0)
        x[i] <- 0
```

Using `ifelse` *simplifies this a lot:*

```
x <- ifelse(x<0, 0, x)
```

Note that the assignment (`<-`) is always *outside* the call to `ifelse`, whereas the assignment is typically *inside* the `if` statement.

Example 14 *Suppose we have two vectors* `x` *and* `y`

```
n <- 5
x <- sample(n)
y <- sample(n)
```

each containing the integers 1 to 5 in a random order.

Suppose you want to the set i-th entry of a new vector `z` *to* $z_i = \max\{x_i, y_i\}$. *You can do this by placing an* `if` *statement inside a* `for` *loop.*

```
z <- numeric(n)                  # Create vector to hold result
for (i in 1:n) {
  if (x[i]>y[i]) {               # If x[i] is larger ...
    z[i] <- x[i]                 # ... set z[i] to  x[i]
  } else {                       # Otherwise (i.e. if x[i] is not larger) ...
    z[i] <- y[i]                 # ... set z[i] to  y[i]
  }
}
```

It is a lot easier to use `ifelse`.

```
z <- ifelse(x>y, x, y)
```

Note that we could have also used subsetting.

```
z <- x                          # Start with a copy of x
select <- y>x                   # Find out for which entries y is larger than x
z[select] <- y[select]          # Set these to y
```

Task 5 *Without running the code in R, determine what* `ifelse` *returns in the code snippet below.*

```
x <- c(1,2,9)
y <- c(2,6,4)
z <- c(3,5,7)
ifelse(x<4, y, z)
```

Task 6 *What does the following loop do?*

```
x <- rnorm(10)                          # Generate some white noise
out <- numeric(length(x))
for (i in seq_along(x)) {
    if (x[i]>0) {
        out[i] <- x[i]
    } else {
        out[i] <- -x[i]
    }
}
```

Rewrite the code so that it uses the function `ifelse`.

Avoiding loops

Avoiding loops

https://youtu.be/P-Z7-dh49D0

Duration: 11m17s

Loops are relatively slow in R. Code usually runs faster and can become more legible when avoiding loops. R's vectorised nature makes this particularly easy.

Example 15 *Suppose we want to calculate the sum of two vectors* `x` *and* `y` *of length 100,000.*

```
n <- 1e5
x <- rnorm(n)
y <- rnorm(n)
```

The easiest and fastest way is to exploit that R can add vectors together using the operator `+`.

```
system.time(z <- x + y)
```

```
##    user  system elapsed
##   0.000   0.001   0.001
```

Using a loop to set the entries `z` *one-by-one is a lot slower:*

```
system.time( {
  z <- numeric(n)                 # Create vector of correct size
  for (i in 1:n)                  # Set entries one-by-one
    z[i] <- x[i]+y[i]
} )
```

```
##    user  system elapsed
##   0.142   0.004   0.149
```

An even less efficient approach would consist of creating a vector `z` *of (initially) zero length, and then appending the newly computed* z_i *one by one.*

```
system.time( {
  z <- c()
  for (i in 1:n)
    z <- c(z, x[i]+y[i])
} )
```

```
##    user  system elapsed
##  28.291   1.558  30.704
```

This is awfully slow. The reason why this approach is so slow is that in every iteration `z` *is replaced by a new vector. Memory for the new vector needs to be allocated, the current vector* `z` *needs to be copied into the new vector* `z`*, and finally the old vector* `z` *needs to be deleted from the memory.*

Our code would be equally slow if we were sloppy when initialising the vector `z` *and create a vector of zero length. This is valid as R increases the size of the vector as needed, but brings with it the same issues of having to repeatedly copy the vector as it is being extended.*

```
system.time( {
  z <- c()                              # Create an empty vector and let R extend it
  for (i in 1:n)                        # Set entries one-by-one
      z[i] <- x[i]+y[i]
})
```

```
##    user  system elapsed
##  16.178   2.046  18.469
```

We will now look at a less straightforward example showcasing how using vector-based operations and subsetting can speed up code (and yield more compact code).

Example 16 *Suppose we compute the vector of increments $d_i = x_{i+1} - x_i$. Our first approach uses a loop.*

```
system.time( {
    n <- length(x)
    d <- numeric(n-1)
    for (i in 1:(n-1))
        d[i] <- x[i+1] - x[i]
} )
```

```
##    user  system elapsed
##   0.133   0.000   0.133
```

We cannot simply set `d` *to the difference of* `x` *and* `x`*, as we subtract x_i from x_{i+1}. Essentially we need to offset the two copies of* `x` *before we subtract them. We can do this using*

```
system.time( {
    n <- length(x)
    d <- x[-1] - x[-n]
} )
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

which is a lot faster. (The video explains this trick in more detail.)

Solutions to the tasks

Task 1 You can use the following R code.

```
x <- rnorm(1)
y <- rnorm(1)
if (x<y) {
    x <- y
} else {
    y <- x
}
```

The code is equivalent to setting

```
x <- y <- max(x,y)
```

Task 2 You can use the following loop

```
x <- rnorm(100)
x[sample(100,10)] <- NA
y <- numeric(length(x))
for (i in seq_along(x))
    if (!is.na(x[i])) {
        y[i] <- x[i]
    } else {
        y[i] <- 0
    }
```

In this example it is a lot easier to use subsetting.

```
y <- x
y[is.na(y)] <- 0
```

Task 3 We create an empty vector `y` and only append the non-missing entries from `x`.

```
y <- c()
for (i in seq_along(x))
    if (!is.na(x[i]))
        y <- c(y, x[i])
```

The loop is a lot slower than the subsetting approach and also less clear to read (for a human).

Task 4 We can store the entire sequence (rather than just the store the current value) using the following R code.

```
n <- 100
x <- numeric(n)
x[1] <-  1
for (i in 2:100) {
    x[i] <- x[i-1]/2 + 1/x[i-1]           # Update x
    if (abs(x[i]-x[i-1])<1e-8)            # Check for convergence
        break
}
x[i]
```

```
## [1] 1.414214
```

Task 5 The call to `ifelse` returns the vector $(2, 6, 7)$. The condition evaluates to (`TRUE`,`TRUE`,`FALSE`), so the result is set to $(y_1, y_2, z_3) = (2, 6, 7)$.

Task 6 The loop stores the modulus ("absolute value") of `x` in `out`. This can be recoded using `ifelse` as

```
out <- ifelse(x>0, x, -x)
```

We could, of course, also have simply set

```
out <- abs(x)
```

Functions

Calling R functions

Calling an R function

`https://youtu.be/IUhA7qpgs6Q`

Duration: 8m35s

So far, we have used many different R functions: `rnorm`, `cbind`, etc. When we called these functions, we usually passed on some information ("arguments") to the function. For example we used the command `rnorm(10)` to create a sequence of white noise of length 10. This section explains the formal rules R uses to match arguments.

Consider the example of the function `dnorm`. According to its help page, its arguments are:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

`dnorm` computes the (logarithm of the) p.d.f. of the $\mathsf{N}(\texttt{mean}, \texttt{sd}^2)$ distribution.

To evaluate the p.d.f. of the $N(0.5, 1)$ distribution at $x = -2$ we can use any of the following lines (and many other variants on the theme):

```
dnorm(-2, mean=0.5, sd=1)
dnorm(-2, 0.5, 1)
dnorm(0.5, 1, x=-2)
dnorm(x=-2, mean=0.5)
```

The reason as to why all the above commands work is that arguments in R are handled in a much more flexible way compared other programming languages such as C or Java.

In R arguments can be given in two different forms:

- Named form: Arguments are given in the form `name=value`, i.e. `dnorm(x=-2, mean=0.5)`. Arguments in named form can be in any order. The name of the argument does not need to be given in full as long as it is unique, i.e. you can use `dnorm(x=-2, m=0.5)`: in this case R will interpret `m` as `mean`, as there are no other arguments starting with the letter "m".
- Positional form: In positional form no `name`s are given. In this case, R matches the arguments by their position. For `dnorm` for example, the first argument is, according to the function definition, `x`, and the second argument is `mean`, thus R interprets `dnorm(-2, 0.5)` as `dnorm(x=-2, mean=0.5)`.

You can also use a mixture of named and positional form. In this case R first matches the named arguments, and then matches the remaining arguments using their position. Let's look at the example `dnorm(0.5, 1, x=-2)` in more detail. R first matches the named arguments, in our case `x=-2`. The remaining arguments of the function `dnorm` are `mean`, `sd`, and `log`. The remaining arguments of our call are `0.5, 1`, so R interprets this as `mean=0.5` and `sd=1`.

Some arguments have default values and only need to be specified if you want to set this argument to a different value. This is typically (but not always) visible in the function declaration, e.g. `sd` of `dnorm` defaults to 1 ("`sd=1`"). Thus if we do not specify `sd`, we implicitly set `sd=1`.

Some functions allow unspecified arguments ("..."); we will come back to this later.

Task 1

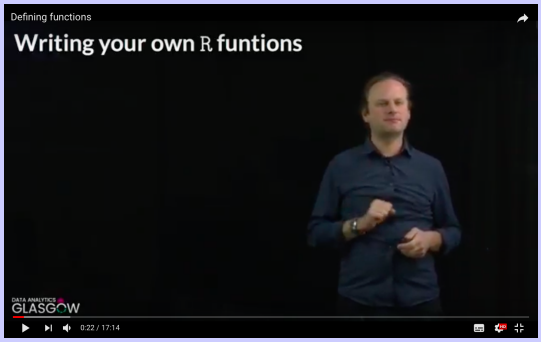
Consider a function taking arguments `a`, `b` *and* `c` *and returning* $(a + 2 \times b)^c$, *defined as set out below.*

```
func <-  function(a, b=0, c=1) {
    (a+2*b)^c
}
```

Rewrite the calls below into named form (i.e. using ***`parameter=value`****) and give the output of* ***`func`*** *without running it in R.*

```
func(1, 2, 3)
func(4)
func(c=3, 2, 1)
```

Defining your own functions



Writing your own R functions

`https://youtu.be/R_h7IntJ6QU`

Duration: 17m14s

Why should I write my own functions?

Functions are one of the most important concepts in programming. Functions allow you to structure complex code.

The idea behind functions is to break a complex problem into blocks of code, each of which is self-contained and performs a specified subtask.

Once we have established that a function performs the specified subtask correctly, we only need to remember how to use the function. We do not need to remember the details of how we implemented the subtask. For example, when you use the function `dnorm` you only need to know that it computes the p.d.f. of the $\mathsf{N}(\mu, \sigma^2)$ distribution, but you do not need to know how it does it.

The three main advantages of using functions are ...

- Functions structure your code, thus making reading and maintaining the code much easier.
- Functions enable the reuse of code and thus help reducing the duplication of code. Duplication of code (i.e. copying and pasting of code from one task to another) is generally considered to be bad programming style. Rather than copying your code you should write a function that implements the common task and then call this function for each task. This approach is not only more efficient and less error-prone, but also avoids the so-called "update anomaly": If you find an error in duplicated code, you have to remember where you copied the code to, and correct every copy of the code. If however you had used a function, you would only need to fix the function.

- Functions can be "unit-tested". We can test whether the function "does what it says on the tin" before we integrate into a more complex project.

Furthermore, using a structured approach also makes implementing complex tasks much easier.

How to write your own functions

The syntax for defining functions in R is

```
function.name <- function(argument1, argument2, ...) {
    statement1
    ...
    statementn
}
```

Example 1

A circle can be drawn in R using the following commands

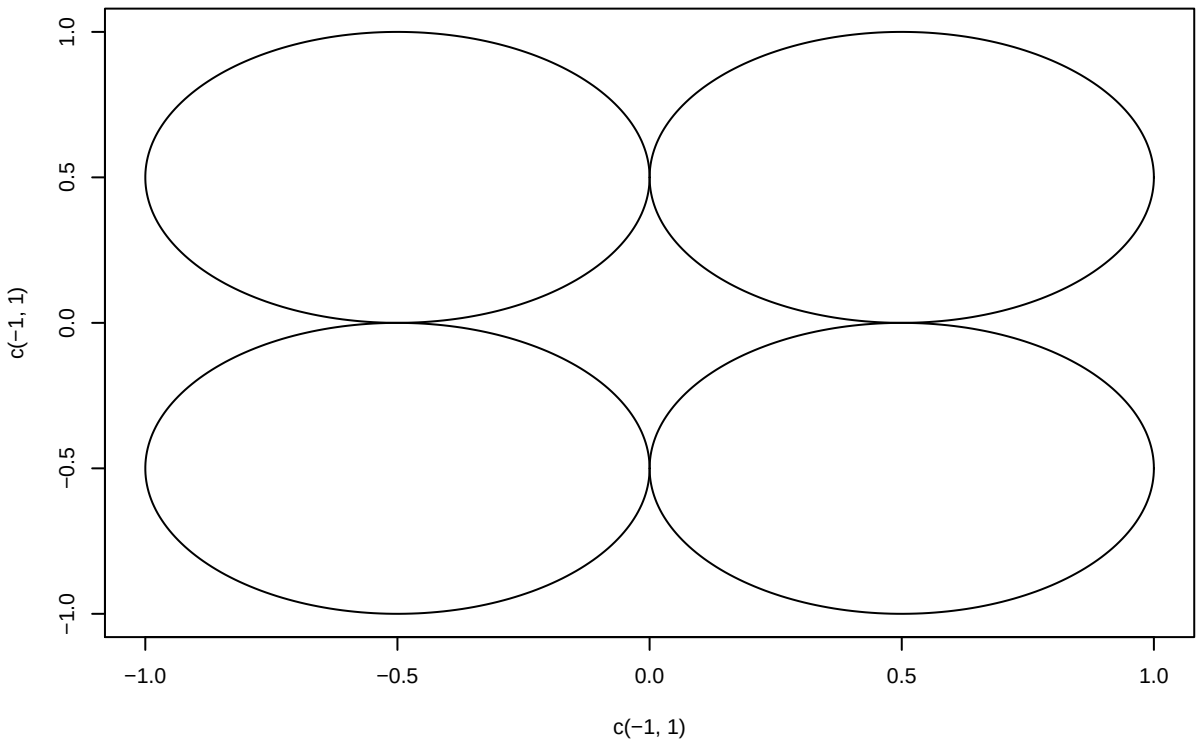
```
plot(c(-1,1), c(-1,1), type="n")    # Set up canvas
x <- 0                              # x-coordinate of centre
y <- 0                              # y-coordinate of centre
r <- 1                              # Radius
t <- seq(0, 2*pi, length=250)
lines(x+r*cos(t), y+r*sin(t))       # Draw circle
```

Suppose you want to draw four circles. Instead of repeating the above commands four times you might want to define a function `circle`*. This function should have three arguments: x-coordinate, y-coordinate, and radius:*

```
circle <- function(x, y, r) {
  t <- seq(0, 2*pi, length=250)
  lines(x+r*cos(t), y+r*sin(t))
}
```

Now we can draw four circles using

```
plot(c(-1,1), c(-1,1), type="n")                # Set up a canvas
circle(-0.5, -0.5, r=0.5)
circle(-0.5,  0.5, r=0.5)
circle( 0.5, -0.5, r=0.5)
circle( 0.5,  0.5, r=0.5)
```



The circles look more like ovals. This is because the scales of the x axis and of the y axis are not equal. If we had used `eqscplot` *from* `MASS` *instead of* `plot`*, the circles would have been perfect circles.*

You can specify default values for some of the arguments using `argument=expression` in the list of arguments. If the user does not provide this argument, the default value (determined by `expression`) is used.

The default value can be a function of other arguments given to the function. Essentially, R evaluates arguments lazily (i.e. only if and when they are needed inside the function body), so when R evaluates `argument` for the first time, it must be able to evaluate `expression`.

Example 2

In our function `circle` *we might want to set the radius by default to 1.*

```
circle <- function(x, y, r=1) {
  t <- seq(0, 2*pi, length=250)
  lines(x+r*cos(t), y+r*sin(t))
}
```

The special argument `...` captures all arguments that are not matched otherwise. It is very useful to pass on additional arguments to a function that you call inside a function without having to worry about the details of these arguments.

Example 3

Suppose we want to use the function `circle` *to draw circles in different colours and using different line types and widths using arguments like* `col`, `lty` *and* `lwd`*. We can then pass on these arguments when we call the* `lines` *command.*

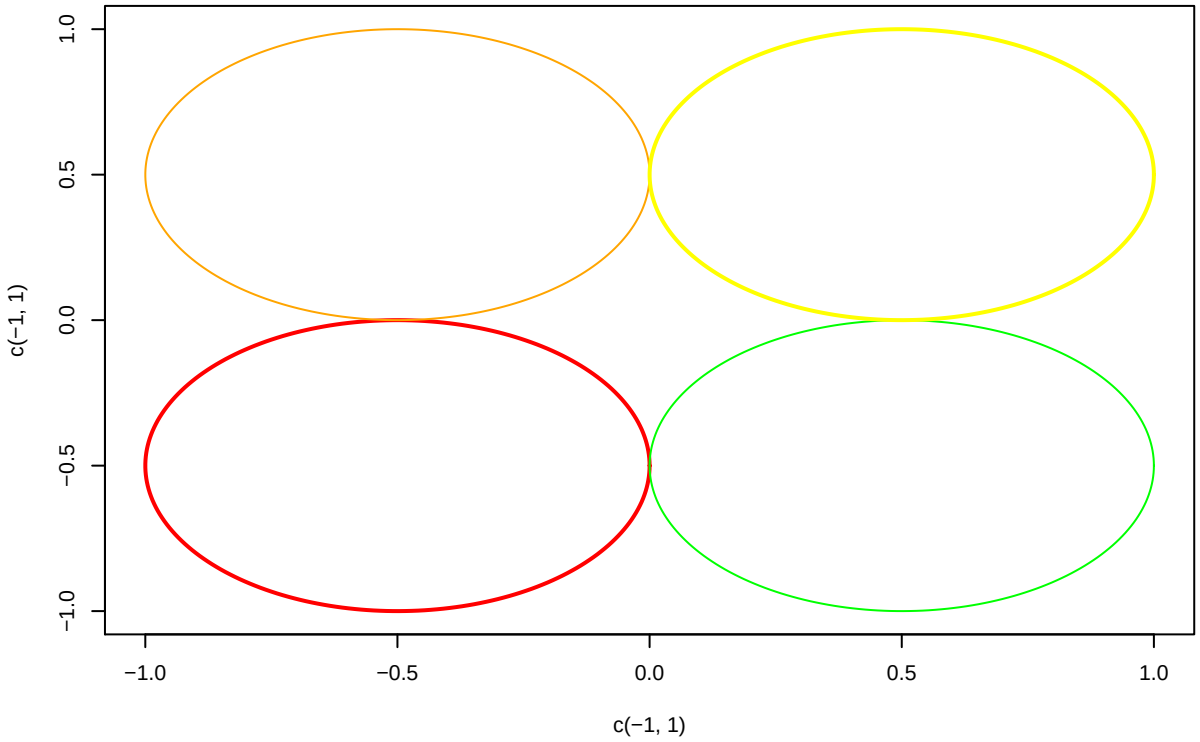
Rather than specifying all possible parmeters that can be used to customise lines, we can simply use the special argument `...`*.*

```
circle <- function(x, y, r=1, ...) {
  t <- seq(0, 2*pi, length=250)
```

```
  lines(x+r*cos(t), y+r*sin(t), ...)
}
```

Now we can draw the circles in different colours and line widths:

```
plot(c(-1,1), c(-1,1), type="n")                    # Set up a canvas
circle(-0.5, -0.5, r=0.5, col="red", lwd=2)
circle(-0.5,  0.5, r=0.5, col="orange")
circle( 0.5, -0.5, r=0.5, col="green")
circle( 0.5,  0.5, r=0.5, col="yellow", lwd=2)
```



Task 2

Set up a "canvas" using

```
plot(c(-3,3), c(-3,3), type="n")
```

and set the variables

```
x <- 0
y <- 0
width <- 4
```

Use the function `rect` *to draw a square with bottom left corner* $(\texttt{x} - \texttt{width}/2, \texttt{y} - \texttt{width}/2)$ *and top right corner* $(\texttt{x} + \texttt{width}/2, \texttt{y} + \texttt{width}/2)$.

Turn (some of) your code into a function `square` *that takes the arguments* `x`, `y` *and* `width` *and that draws a square with bottom left corner* $(\texttt{x} - \texttt{width}/2, \texttt{y} - \texttt{width}/2)$ *and top right corner* $(\texttt{x} + \texttt{width}/2, \texttt{y} + \texttt{width}/2)$. *Additional arguments (such as* `col` *or* `lty`*) provided on to your function* `square` *should be passed on to* `rect`.

Use your function to draw three squares in different colours with centre at $(0, 0)$ *and widths 2, 4, and 6.*

Returning objects

Often we want functions not only to perform certain tasks, but also to return a value (e.g. the result of a calculation). We can return values using the function `return(object)`. The function `return` terminates the function and returns `object`. When we return `object` from the function we can "capture" this value from outside the function and store it in a variable. If we do not assign the returned result from a function to a variable it will be printed on the screen.

If we do not use `return`, then R returns the value of the last statement of the function body.

Example 4

Suppose we want to write a function that computes Stirling's approximation to $n!$*, which is* $\sqrt{2\pi n}n^n \exp(-n)$.

```
stirling <- function(n) {
  approx <- sqrt(2*pi*n) * n^n * exp(-n)
  return(approx)
}
```

Actually, there is no need to use `return` *in the above example, as an R function will always return the value of the last statement. Thus we could have used as well*

```
stirling <- function(n) {
  sqrt(2*pi*n) * n^n * exp(-n)
}
```

We can now calculate the Stirling approximation to $10!$ *using*

```
stirling(10)
```

```
## [1] 3598696
```

As the function `stirling` *now returns a value, we can assign its result to a variable.*

```
exact <- factorial(10)      # Store exact result in exact
exact
```

```
## [1] 3628800
```

```
approx <- stirling(10)      # Store Stirling's approximation in approx
approx
```

```
## [1] 3598696
```

```
exact-approx                # Print the error of the approximation
```

```
## [1] 30104.38
```

Lists are typically used to return more than one value or object.

Example 5

Suppose we want to write a function `stirling.bounds` *that returns the bounds for* $n!$ *obtained from the double inequality*

$$\sqrt{2\pi n}n^n \exp\left(-n + \frac{1}{12n+1}\right) < n! < \sqrt{2\pi n}n^n \exp\left(-n + \frac{1}{12n}\right).$$

The function should return both bounds. We can do this using a list with two entries `lower` *and* `upper`.

```
stirling.bounds <- function(n) {
  approx <- sqrt(2*pi*n) * n^n * exp(-n)
  list(lower=approx * exp(1/(12*n+1)),
```

```
        upper=approx * exp(1/(12*n)))
}
```

We can then calculate the bounds for 10! *using*

```
stirling.bounds(10)
```

```
## $lower
## [1] 3628560
##
## $upper
## [1] 3628810
```

We can calculate the difference between the upper and the lower bound using

```
bounds <- stirling.bounds(10)        # Store result (a list) in variable bounds
bounds$upper-bounds$lower            # Compute difference betweem bounds
```

```
## [1] 249.9094
```

Task 3

Write an R function `all.means` *that returns for a given vector* `x` *the arithmetic mean* $\frac{\sum_{i=1}^{n} x_i}{n}$, *the harmonic mean* $\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$, *and the geometric mean* $\sqrt[n]{\prod_{i=1}^{n} x_i}$. *The function should return a list with the elements* `arithmetic`, `harmonic`, *and* `geometric`.

Inside a function you can use all the control structures we have learned (`if`, loops, ...).

Task 4

Write an R function `quadratic` *which computes the real-valued solutions of the quadratic equation*

$$ax^2 + bx + c = 0$$

Your function should take a, b, *and* c *as arguments and should return the unique solutions to the above quadratic equation.*

- *Your function should start with computing the discriminant* $\Delta = b^2 - 4ac$.
- *If* $\Delta < 0$, *then the quadratic equation has no solution. In this case, return a vector of length 0.*
- *If* $\Delta = 0$, *then the quadratic equation has exactly one ("double") solution,* $-\frac{b}{2a}$. *In this case return this value only.*
- *If* $\Delta > 0$, *then the quadratic equation has two solutions,* $-\frac{b+\sqrt{\Delta}}{2a}$ *and* $-\frac{b-\sqrt{\Delta}}{2a}$. *In this case return a vector of length two containing both solutions.*

What happens if $a = 0$? *Can you also handle this case correctly?*

Scoping

Scoping

`https://youtu.be/vvAeaoB47V8`

Duration: 8m27s

When writing your own functions it is important that you distinguish between variables available locally in the function and variables available in your workspace (or in a parent function, from which your function is being called), i.e. outside your function.

Every variable that exists outside the function can be referenced and used inside the function. However, as soon as you change any of these variables, a local copy will be created: the changes you make to this variable will be only temporary. Once the function has finished running, the changes will be rolled back. Similarly, if you create a new variable, it will only be created temporarily and will be deleted once you exit from the function, unless you return it using the function `return` or by putting the object in the last line of the function body.

In R, all arguments (with the exception of environments) are always passed on to the function by value.

Example 6

Consider the following function.

```
test <- function(a) {
  print(b)
  b <- a
  print(b)
}
```

This function is certainly not best practice, but a valid R function.

```
b <- 19
test(13)
```

```
## [1] 19
## [1] 13
```

```
b
```

```
## [1] 19
```

Whilst the variable `b` *visible inside the function is 13, the variable* `b` *in the workspace has retained the value 19.*

The table below shows how the variables in the workspace and the local variables in the function change as the statements inside the function get run.

Commands entered	*Commands run inside function*	*Variables in workspace*	*Local variables inside function*	*Comment*
`b <- 19`	–	`b=19`	–	–
`test(13)`	`print(b)`	`b=19`	`a=13`	*Prints 19*
	`b <- a`	`b=19`	`a=13, b=13`	*Local* `b` *masks* `b` *from workspace*
	`print(b)`	`b=19`	`a=13, b=13`	*Prints 13*

Commands entered	*Commands run inside function*	*Variables in workspace*	*Local variables inside function*	*Comment*
`print(b)`	–	`b=19`	–	*Prints 19*

The fact that you can access objects available outside functions does not mean you should do this.

Good practice

Functions should be self-contained units, so all inputs should be passed on as arguments and all outputs should be `return`ed. Thus when writing a function, you should use inside the function only

- variables that are passed on to the function as arguments, and
- local variables you have defined inside the function.

You can access variables available only in the workspace, but this is typically (and for good reason) considered to be bad programming style. The following example illustrates why.

Example 7

Suppose you want to write your own function for computing the mean of a vector `x`.

```
x <- 1:10                           # Set x
n <- length(x)                      # Determine length of x
my.mean <- function(x) {
  x.bar <- sum(x) / n               # Compute mean
  x.bar
}
```

When you want to compute the mean of `x`*, you can now run*

```
my.mean(x)
```

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

which returns the correct mean. So, at first sight everything seems fine. However when we run

```
y <- 1:3
my.mean(y)
```

```
## [1] 0.6
```

rather than 3, the correct value. Why did this happen? When looking at the implementation of `my.mean` *we computed the length* `n` *outside the function, thus we assumed that there is a variable* `n` *in the workspace which holds the length of the vector whose mean we want to compute. This was the case for* `x`*, but is not the case for* `y`.

According to the advice given above, we should not have used `n` *in the function without having it first set to the length of* `x` *inside the function. Thus we should have coded the function* `my.mean` *as follows:*

```
my.mean <- function(x) {
  n <- length(x)                    # Determine length of x - now inside function
  x.bar <- sum(x) / n               # Compute mean
  x.bar
}
```

Designing functions

Programming is, just like mathematical proofs, an art.

Plan before you code

When working on more complex programming tasks, take the time to plan before you start coding. Structuring a problem is often the key step to getting things right. Once you have broken down a problem into small, more manageable pieces it is a lot easier to code these up. There are no cook-book recipes that can guarantee you success. Below are a few hints which I found useful.

Keep in mind that your code doesn't just need to work. It also has to be maintainable both by others and future you. If you structure your code in terms of short well-defined (and documented) functions, your code is a lot easier to read, test, maintain and debug.

Often, going through the following steps helps tackling a complex programming problem:

1. Try to understand the problem you are trying to solve. If you don't know where to start, think of a simple special case.
2. Work out which steps are required to solve the problem. When working on a complex problem, think how you could divide the problem into smaller (and hopefully easier) sub-tasks. It is often a good idea to use a "top-down" design: split your main task into at most a dozen sub-steps. If necessary, split these sub-steps again into "sub-sub-steps" etc. until you end up with small enough steps which are straightforward to implement. It might help writing up your steps as "instructions" in plain English, or using flow diagrams or pseudo-code (use whatever works best for you). For complex problems, this step is by far the most difficult.
3. Translate each of these (sub-)steps into code. Make sure you document your functions properly. For each function explain at least what the function does, what arguments it takes, and what it returns.
4. Check your program. Does it work? Try out a few examples. If you can think of difficult special cases, try your program with these. Do not only test the final main function, but also test the functions implementing the sub-steps individually: this way you are not only more likely to find a mistake, it is also easier to locate what has gone wrong. If any of the functions does not do what it should, you need to find the mistake (more on this in the subsection Debugging).

Unit testing

When programming it is important that we make sure that our programmes work as intended. So every time you have written a piece of code (especially a function), you should test it systematically. This way you can ensure that it is indeed working as intended.

It pays off to confront possible issues as early as possible. It is much easier to test small blocks of code, rather than big blocks of code. Thus it is much better to use many small functions than a few big ones. This way, each of the functions can be tested individually, and once a bug is found, it can be identified fairly quickly.

There are in principle two ways how software can be checked:

- "White-box testing": One can inspect and review the source code of the programme. This is typically only useful if the code is checked by someone other than the author of the programme. However, this makes code review expensive and thus it is often only used for safety-critical or security-critical software. It is however also used for statistical programmes used in clinical trials.
- "Black-box testing": Alternatively, one can apply the function to a number of test cases for which the correct output and behavior of the function can be predicted. This form of testing is much simpler. However the test cases used might not cover every possible scenario, so some bugs might remain unnoticed.

In some settings (e.g. statistical analysis of clinical trials) it is not uncommon to have two programmers implement the same tasks independently. When finished, the code produced by the two programmers is then run and the results are compared. This allows spotting mistakes, unless both programmers make the same mistake.

Warnings and Errors

Exceptions are conditions which do not allow continuing the normal flow of execution. R has two major types of exceptions:

Warnings

Warnings are less serious exceptions. R has encountered a problem processing some part of your commands, but can continue processing the other commands. However, the result might not be what you would expect, so you should carefully examine all variables used in the command that triggered the warning. Before returning to the command prompt R will display a short warning message describing the problem that has occurred.

Examples of code that produces warnings are ...

- taking the logarithm of negative numbers,

```
x <- log(-1:1)
```

```
## Warning in log(-1:1): NaNs produced
```

```
x
```

```
## [1]  NaN -Inf    0
```

- trying to use recycling rules when the lengths of the vectors involved are not multiples of each other.

```
x <- 1:3
y <- 4:5
z <- x+y
```

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

```
z
```

```
## [1] 5 7 7
```

Errors

Errors are more serious exceptions. They are that serious that R cannot continue processing your commands and returns to the command prompt showing a brief description of the error.

Examples of code causing errors are ...

- syntax errors like mismatched parentheses

```
sin(x[i)]
```

```
## Error: <text>:1:8: unexpected ')'
## 1: sin(x[i)
##            ^
```

- trying to use a variable that does not exist,

```
print(I.do.not.exist)
```

```
## Error in print(I.do.not.exist): object 'I.do.not.exist' not found
```

- calling a function with the wrong arguments.

```
rnorm(sample.size=10)
```

```
## Error in rnorm(sample.size = 10): unused argument (sample.size = 10)
```

A warning or an error message does not necessarily imply that there is something wrong with this specific line of code. It might well be that you made a mistake earlier on, but somehow R manages to carry on for a while before producing an error or a warning message.

In the subsection Debugging we will look at techniques for identifying the root cause of errors.

Raising warnings and errors in your code

When writing your own R functions, you can (and should) make use of the functions like `warning` and `stop` to handle exceptions (do *not* use `print` for these purposes). `warning` will display a warning message, but not abort the function, whereas `stop` will display an error message and return immediately to the command prompt.

Example 8

Consider the following function `moments`*, which calculates the expected value and standard deviation for a discrete distribution with range space* `x` *and associated probabilities* `p`*.*

```
moments <- function(x, p) {
    e.x <- sum(x*p)
    var.x <- sum((x-e.x)^2*p)
    sd.x <- sqrt(var.x)
    list(e.x=e.x, sd.x=sd.x)
}
```

The calculations above are only sensible if

- `x` *and* `p` *are vectors of the same length.*
- *The entries in* `p` *are non-negative.*
- *The entries in* `p` *sum to one. However in this case we can simply rescale* `p` *so that it sums to 1.*

Things suggests that we should raise an error in the first two scenarios and raise a warning if we need to rescale `p`*.*

```
moments <- function(x, p) {
    if (length(x)!=length(p))
        stop("p and x must be of the same length.")
    if (any(p<0))
        stop("p is a vector probabilities and cannot have negative entries")
    if (abs(sum(p)-1)>1e-8) {                  # Allow for some numerical error
        p <- p/sum(p)
        warning("p has been rescaled so that probabilities sum to 1")
    }
    e.x <- sum(x*p)
    var.x <- sum((x-e.x)^2*p)
    sd.x <- sqrt(var.x)
    list(e.x=e.x, sd.x=sd.x)
}
```

Debugging

Bugs

A bug is a fault in a computer program that will cause the program either to produce an error or a warning message, and/or to produce an incorrect or unexpected result. Whilst it is relatively easy to find bugs that lead to error or warning messages, it is much harder to locate bugs that "just" lead to an incorrect result: often such bugs remain unnoticed for some time.

Bugs can be due to many factors: a faulty design ("semantic error"), an error in the code, numerical problems, or, however not very often, a bug in R (or the operating system) itself.

It is almost impossible (at least for me) to write code that is free from errors. Having a well thought-out design and having structured and commented your code will not only help avoiding mistakes, but also help simplify debugging. Nonetheless, every program will (at least initially) contain bugs. Thus it is important to know how to find bugs and how to fix them.

It is an unfortunate fact of life that it often takes as least as much time (or even more) to debug a program as it takes to write it in the first place. When programming I typically spend around one third of my time on designing the program, one third on actual coding, and another third on testing and debugging.

Identifying and fixing bugs usually involves going through the following steps:

1. Realize that your program does not (always) work as intended.
2. Find out how you can trigger the bug. Without being able to reproduce the bug, you have little chance of being able to locate the bug. Thinking about the conditions under which you can trigger the bug might help you get an idea of where the bug might be. This step can be very difficult, especially if your code uses random numbers (simulation studies, bootstrap, MCMC, etc.)
3. Once you know how to trigger the bug, you have to identify where it is and what is causing the bug. R has special tools for this (more on this in the video).
4. Once you have identified the actual mistake, you "just" have to correct it.
5. Look out for similar bugs. If for example your bug was due to a design fault, it is likely that other parts of the program also turn out to be faulty.

Debugging in RStudio

Debugging in Rstudio

`https://youtu.be/hS482Goj1t8`

Duration: 9m37s

Task 5

The function below is meant to compute the variance $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$ *of a sample* $(x_1,\ldots,x_n)$.

```
compute.variance <- function(x) {
  x.bar <- mean(x)
  variance <- sum(x-x.bar)^2 / n-1
  variance
}
```

Can you fix it?

Loop avoidance functions

R contains several *loop-avoidance functions*. Historically these functions used to be somewhat more efficient than loops, however this is not the case any more (at least in R). However, if used appropriately, they can make the code more compact and easier to read.

The function `apply`

Example 9

Suppose you want to find the maximal values of the four numeric columns of Anderson's `iris` *data set. We can do this using a loop:*

```
iris.maxi <- numeric(4)
for (i in 1:4)
  iris.maxi[i] <- max(iris[,i])
iris.maxi
```

```
## [1] 7.9 4.4 6.9 2.5
```

The `apply` *function allows us to do this in a single line*

```
iris.maxi <- apply(iris[,1:4], 2, max)
iris.maxi
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##          7.9          4.4          6.9          2.5
```

The syntax of the function `apply` is

```
apply(X, MARGIN, FUN, ...)
```

The arguments are:

- `X` is the matrix or data frame (or array in more general) to be used.
- `MARGIN` indicates whether the function should be applied row-wise (`MARGIN=1`) or column-wise (`MARGIN=2`).
- `FUN` is the function to be applied. Additional arguments to the function can be supplied using the `...` argument.

Thus `apply(iris[,1:4], 2, max)` computes the column-wise maxima of the first four columns of the iris dataset.

`apply` is a very versatile function, we can use it for example to remove all observations with missing values from a dataset.

Example 10

To remove all observations with missing values from the `Cars93` *data from the library* `MASS` *we can use*

```
library(MASS)
line.has.na <- apply(is.na(Cars93), 1, any)
cars.no.nas <- Cars93[!line.has.na,]
```

The function `is.na` *returns a matrix indicating whether the corresponding entry of* `cars` *is a missing value. Applying the function* `any` *row-wise to this matrix creates a vector indicating for each row whether it contains missing values.*

Task 6

You probably know that the sum of exponentially distributed random variables has a Gamma distribution: if $X_1, \ldots, X_k \overset{\text{i.i.d.}}{\sim} \mathsf{Expo}(\theta)$*, then* $Y = \sum_{i=1}^{k} X_i \sim \mathsf{Ga}(k, \theta)$*. Exploit this result to draw a sample of size 10 from the* $\mathsf{Ga}(100, 0.5)$ *distribution. Do this both implementing a* `for` *loop and using the function* `apply`*.*

Example 11

Consider a bivariate contingency table on voting patterns in the US

Ethnicity	Obama	Romney	No preference
Caucasian	392	598	230
other ethnicity	484	139	177

We are interested in the question whether the voting pattern is independent of the ethnicity of the voter. This can be investigated using a χ^2*-test. When using* N_{ij} *to denote the observed counts, then the test statistic is defined as*

$$T = \sum_{i,j} \frac{(N_{ij} - E_{ij})^2}{E_{ij}}, \qquad \text{with } E_{ij} = \frac{N_{i\bullet} N_{\bullet j}}{N}$$

where N *is the grand total and* $N_{i\bullet} = \sum_j N_{ij}$ *and* $N_{\bullet j} = \sum_i N_{ij}$ *are the row and column totals.*

```
N <- rbind(c(392, 598, 230), c(484, 139, 177))
colnames(N) <- c("Obama","Romney", "No preference")
rownames(N) <- c("caucasian", "other")
Ni <- apply(N, 1, sum)              # row totals
Nj <- apply(N, 2, sum)              # column totals
total <- sum(N)                     # overall total
E <- Ni %*% t(Nj) / total           # expected counts
statistic <- sum((N-E)^2/E)         # test statistic
statistic
```

```
## [1] 224.8192
```

The `apply` function in R has many siblings. The functions `lapply` and `sapply` for example can be applied to lists, see their help pages for details.

The functions `sweep` and `scale`

Example 12

Suppose we want to centre the first four columns of the `iris` *dataset by subtracting their means. We can do this using a loop.*

```
iris.means <- apply(iris[,1:4], 2, mean)     # Determine means
iris.zeromean <- iris[,1:4]                  # Copy iris dataset
for (i in 1:4)                               # Subtract mean of each column
  iris.zeromean[,i] <- iris.zeromean[,i] - iris.means[i]
```

However, we can avoid the loop by using the function `sweep`:

```
iris.means <- apply(iris[,1:4], 2, mean)
iris.zeromean <- sweep(iris[,1:4], 2, iris.means, "-")
```

The general syntax of the function `sweep` is

```
sweep(x, MARGIN, STATS, FUN, ...)
```

The arguments of `sweep` are

- `x` is the matrix / dataframe (or array in more general) to be used.

- `MARGIN` indicates whether we should "sweep" row-wise (`MARGIN=1`) or column-wise (`MARGIN=2`).
- `STATS` The vector to be "swept" out.
- `FUN` The function (or operator, in quotes) used to "sweep" out `STAT`. `FUN` must have at least two arguments, additional arguments can be passed on to `FUN` using the `...` argument. The default value of `FUN` is `-`.

`sweep(iris[,1:4], iris.means, 1, "-")` thus subtracts `iris.means` from each row of `iris[,1:4]`.

R has a built-in function to standardize data: `scale`. We can subtract the means of each column using

```
scale(iris[,1:4], center=TRUE, scale=FALSE)
```

The argument `center=TRUE` tells R to subtract the mean of each column. The argument `scale=FALSE` tells R not to divide each column by its standard deviation.

The function `by`

Example 13

Suppose you want to compute the average values of the four measurements for the three different species in the `iris` *data set.*

We could use

```
colMeans(iris[iris$Species=="setosa", 1:4])
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
```

```
colMeans(iris[iris$Species=="versicolor", 1:4])
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
```

```
colMeans(iris[iris$Species=="virginica", 1:4])
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

This code however requires us to find out the three species names ahead of time. We could use a `for` *loop and the function* `levels` *to do this automatically.*

```
for (species in levels(iris$Species))
  print(colMeans(iris[iris$Species==species, 1:4]))
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

The function `by` *lets us do such tasks more comfortably*

```
by(iris[,1:4], iris$Species, colMeans)
```

```
## iris$Species: setosa
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
## ------------------------------------------------------------
```

```
## iris$Species: versicolor
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
## ------------------------------------------------------------
## iris$Species: virginica
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

The general syntax of the function `by` is

```
by(data, INDICES, FUN, ...)
```

The arguments of `by` are

- `data` is the matrix or data frame to be used.
- `INDICES` contains the variable (a factor, or coercible to a factor) to be used to group the data.
- `FUN` is the function to be applied to each group. Additional arguments to the function can be supplied using the additional `...` argument.

Just like `apply`, we can use `by` together with user-defined functions.

Example 14

The code below computes separate regressions of the petal length against the sepal length for each species.

```
do.regression <- function(x) {
  lm(Petal.Length ~ Sepal.Length, data=x)
}
by(iris, iris$Species, do.regression)
```

```
## iris$Species: setosa
##
## Call:
## lm(formula = Petal.Length ~ Sepal.Length, data = x)
##
## Coefficients:
##  (Intercept)  Sepal.Length
##       0.8031        0.1316
##
## ------------------------------------------------------------
## iris$Species: versicolor
##
## Call:
## lm(formula = Petal.Length ~ Sepal.Length, data = x)
##
## Coefficients:
##  (Intercept)  Sepal.Length
##       0.1851        0.6865
##
## ------------------------------------------------------------
## iris$Species: virginica
##
## Call:
## lm(formula = Petal.Length ~ Sepal.Length, data = x)
##
## Coefficients:
##  (Intercept)  Sepal.Length
```

```
##       0.6105        0.7501
```

Solutions to the tasks

Task 1

The call

```
func(1,2,3)
```

is equivalent to

```
func(a=1, b=2, c=3)
```

and calculates $(1 + 2 \times 2)^3 = 5^3 = 125$.

The call

```
func(4)
```

is equivalent to

```
func(a=4, b=0, c=1)
```

(default values are used for `b` and `c`) and calculates $(4 + 0 \times 2)^1 = 4^1 = 4$.

The call

```
func(c=3, 2, 1)
```

is equivalent to

```
func(a=2, b=1, c=3)
```

and calculates $(2 + 1 \times 2)^3 = 4^3 = 64$.

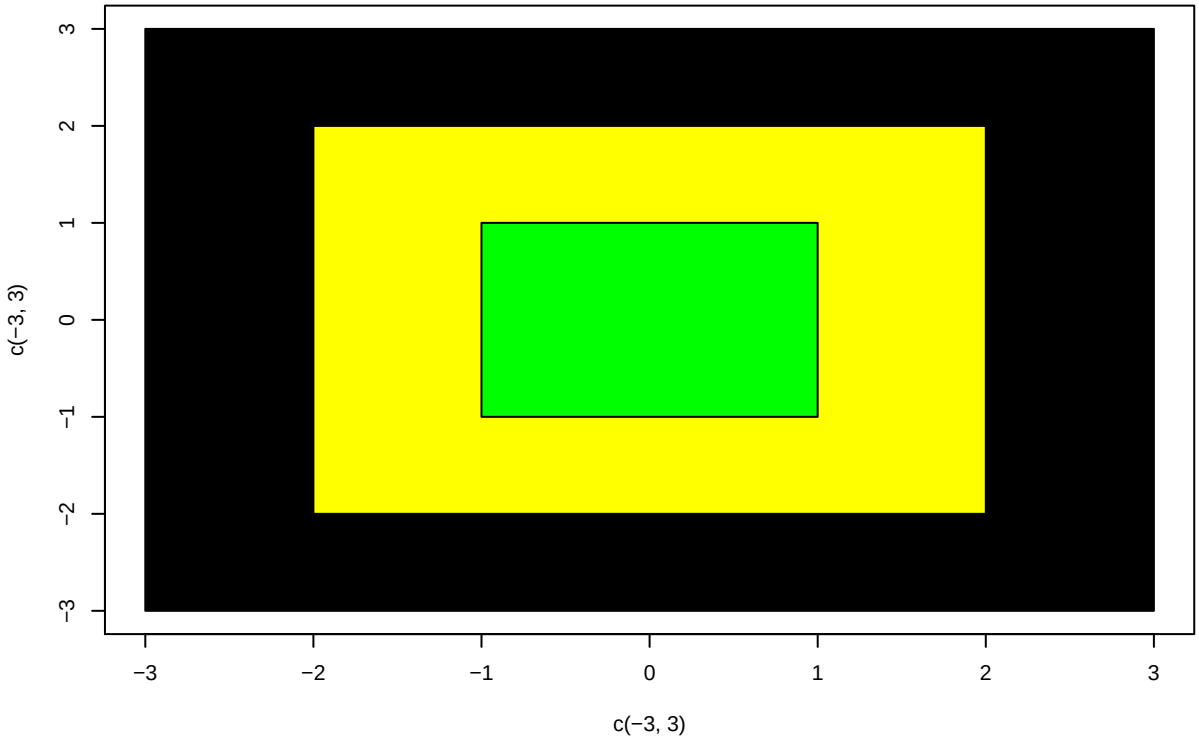
Task 2

We can define the function square as follows.

```
square <- function(x, y, width, ...) {
  rect(x-width/2, y-width/2, x+width/2, y+width/2, ...)
}
```

We can then draw the three rectangles using

```
plot(c(-3,3), c(-3,3), type="n")
square(0, 0, 6, col="black")
square(0, 0, 4, col="yellow")
square(0, 0, 2, col="green")
```



Task 3

We start by defining functions which calculate the harmonic mean and the geometric mean.

```
#' Compute the harmonic mean
#' @param x numeric vector containing the data
#' @return the harmonic mean of x
harmonic.mean <- function(x) {
  n <-length(x)
  n / sum(1/x)
}

#' Compute the geometric mean
#' @param x numeric vector containing the data
#' @return the geometric mean of x
geometric.mean <- function(x) {
  n <- length(x)
  prod(x)^(1/n)
}
```

We then combine these two functions and `mean` into the function `all.means`:

```
#' Compute the arithmetic, geometric and harmoic mean
#' @param x numeric vector containing the data
#' @return a list containing the arithmetic, geometric and harmonic mean of x
all.means <- function(x) {
  n <- length(x)
  list(arithmetic = mean(x),
       harmonic = harmonic.mean(x),
       geometric = geometric.mean(x))
}
```

Task 4

The case $a = 0$ corresponds to a linear equation (as the coefficient in front of the quadratic term is zero), so the solution is given by $-\frac{c}{b}$.

```
#' Solve quadratic equation ax^2 + bx + c = 0
#' @param coefficient of the quadratic term (scalar)
#' @param coefficient of the linear term (scalar)
#' @param coefficient of the intercept term (scalar)
#' @return the real solution(s) as a vector of length 0, 1 or 2.
quadratic <- function(a, b, c) {
  if (abs(a)<1e-10) {              # Linear case (for really small a the quadratic
                                   # formula is unstable, so we use the linear one)
     return(-c/b)
  }
  delta <- b^2 - 4*a*c             # Discriminant
  if (abs(delta)<1e-10) {          # Exactly one solution (allowing for rounding errors)
    return(-b/(2*a))
  }
  if (delta>0) {                   # Two real solutions
    sol1 <- -(b+sqrt(delta)) / (2*a)
    sol2 <- -(b-sqrt(delta)) / (2*a)
    return(c(sol1, sol2))
  }
  return(c())                      # Otherwise no solution
}
```

Task 5

First of all we need to define `n` as the length of `x`. There are also two mistakes in the line `variance <- sum(x-x.bar)^2 / n-1`. As it stands it computes

$$\frac{\left(\sum_{i=1}^{n}(x_i - \bar{x})\right)^2}{n} - 1$$

instead of

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$

Thus this line of code should be `variance <- sum((x-x.bar)^2) / (n-1)`, i.e. the function should be

```
compute.variance <- function(x) {
  x.bar <- mean(x)
  n <- length(x)
  variance <- sum((x-x.bar)^2) / (n-1)
  variance
}
```

Task 6

```
result <- numeric(10)
for (i in 1:10)
  result[i] <- sum(rexp(100, 0.5))
  result
```

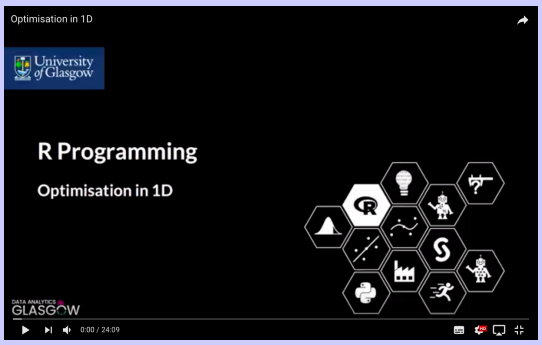
```
##  [1] 191.5792 214.3045 207.3660 205.1126 188.3656 200.2940 193.1433
##  [8] 204.5403 200.8767 206.9854
```

```r
A <- matrix(rexp(10*100, 0.5), ncol=100)
apply(A, 1, sum) #the simulated numbers will not be the same as in the first case
```

```
##   [1] 188.8110 245.0213 186.0111 172.5276 207.9535 231.6238 175.5332
##   [8] 198.5468 160.3511 199.5012
```

Numerical Methods in R

Optimisation in 1D



Optimisation in 1D

`https://youtu.be/blGBEl3f8tY`

Duration: 24m10s

In Statistics and Data Science we often come across optimisation problems. This is typically when we want to estimate parameters. This is often done using a technique called "maximum likelihood": its idea is that we can estimate parameters by maximising the (logarithm) of the joint probability density function or probability mass function.

In simple cases, we can solve optimisation problems in closed form. If the function is differentiable "all" we have to do is solve the equation that the derivative is zero (assuming the maximum does not occur on a boundary). However in many real life scenarios, we have to resort to numerical methods for maximising an objective function.

Example 1

Consider the following function (together with its first two derivatives), which we would like to maximise over θ (the data points x_i are given to us).

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^n \log f(x_i) = -\sum_{i=1}^n x_i + (\theta - 1) \cdot (\sum_{i=1}^n \log x_i) - n \cdot \log \Gamma(\theta) + \text{const} \\ \ell'(\theta) &= (\sum_{i=1}^n \log x_i) - n \cdot (\log \Gamma(\theta))' \\ \ell''(\theta) &= -n \cdot (\log \Gamma(\theta))'' \end{aligned}$$

You may have already noticed that this is the loglikelihood for the shape parameter of the gamma distribution.

We can write an R function to compute the objective function and its first derivatives:

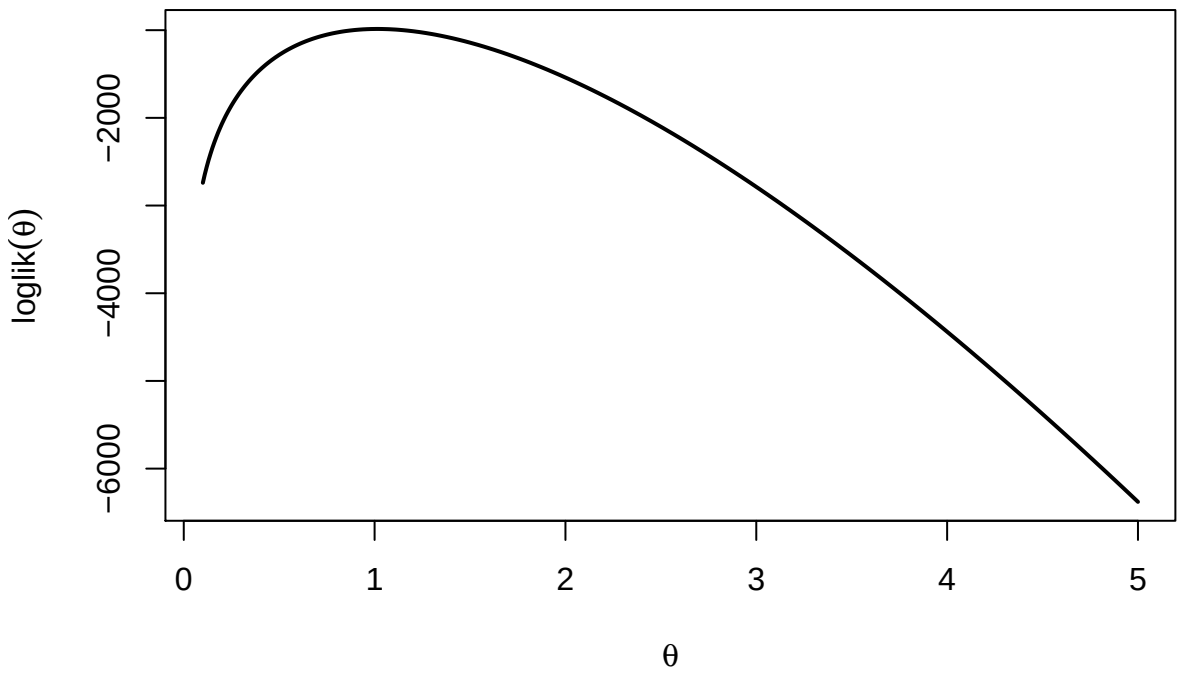
```
loglik <- function(theta, x) {
  -sum(x) + (theta-1)*sum(log(x)) - length(x) * lgamma(theta)
}

loglik.d <- function(theta, x) {
  sum(log(x)) - length(x) * digamma(theta)
}

loglik.dd <- function(theta, x) {
  -length(x) * trigamma(theta)
}
```

We also need some data `x`*, which we will draw from an exponential distribution, which is a special case of the gamma distribution with shape parameter equal to one.*

```
x <- rexp(1e3)
```

To get an idea of how the objective function looks like we can plot it.

```r
theta.range <- 10:500/100
plot(theta.range, loglik(theta.range, x), type="l", lwd=2,
     xlab=expression(theta), ylab=expression(loglik(theta)))
```



We will next introduce Newton's method, a very basic method for maximising / minimising a function. Note that there are two ways of viewing Newton's method. It can be seen as an algorithm for solving a nonlinear equation as well as a method of maximising / minimising a function (i.e. solving that the derivative is zero).

Newton's method

Some background *Consider the quadratic function of a scalar* θ

$$Q(\theta) = c + b \cdot (\theta - \theta_0) + \frac{1}{2} a \cdot (\theta - \theta_0)^2.$$

To find its only local extremum we need to set its derivative to zero, i.e. solve

$$\frac{\partial}{\partial \theta} Q(\theta) = b + a \cdot (\theta - \theta_0) = 0 \qquad \Longleftrightarrow \qquad \theta = \theta_0 - \frac{b}{a}.$$

$\theta_0 - \frac{b}{a}$ *is the global maximum of* $Q(\cdot)$ *if* $a < 0$*. It is the global minimum of* $Q(\cdot)$ *if* $a > 0$*.*

The basic idea of Newton's method is to repeatedly approximate the objective function $\ell(\theta)$ by a quadratic function, or to be more precise by its second order Taylor expansion:

$$\ell(\theta) \approx \ell(\theta_0) + \ell'(\theta_0) \cdot (\theta - \theta_0) + \frac{1}{2} \cdot \ell''(\theta_0) \cdot (\theta - \theta_0)^2,$$

where

$$\ell'(\theta_0) = \frac{\partial \ell}{\partial \theta}(\theta)\Big|_{\theta=\theta_0}, \qquad \ell''(\theta_0) = \frac{\partial^2 \ell}{\partial \theta^2}(\theta)\Big|_{\theta=\theta_0}$$

The minimum / maximum of the quadratic approximation is attained for

$$\theta_0 - \frac{\ell'(\theta_0)}{\ell''(\theta_0)}$$

Newton's method consists of nothing other that repeatedly using this approximation.

1. Initialise $\theta^{(0)}$ to some value.
2. For $h = 1, 2, 3, \ldots$ iterate until convergence $\ldots$ Set

$$\theta^{(h)} = \theta^{(h-1)} - \frac{\ell'(\theta^{(h-1)})}{\ell''(\theta^{(h-1)})}$$

$\theta^{(h)}$ denotes the value θ takes at the end of the h-th iteration.

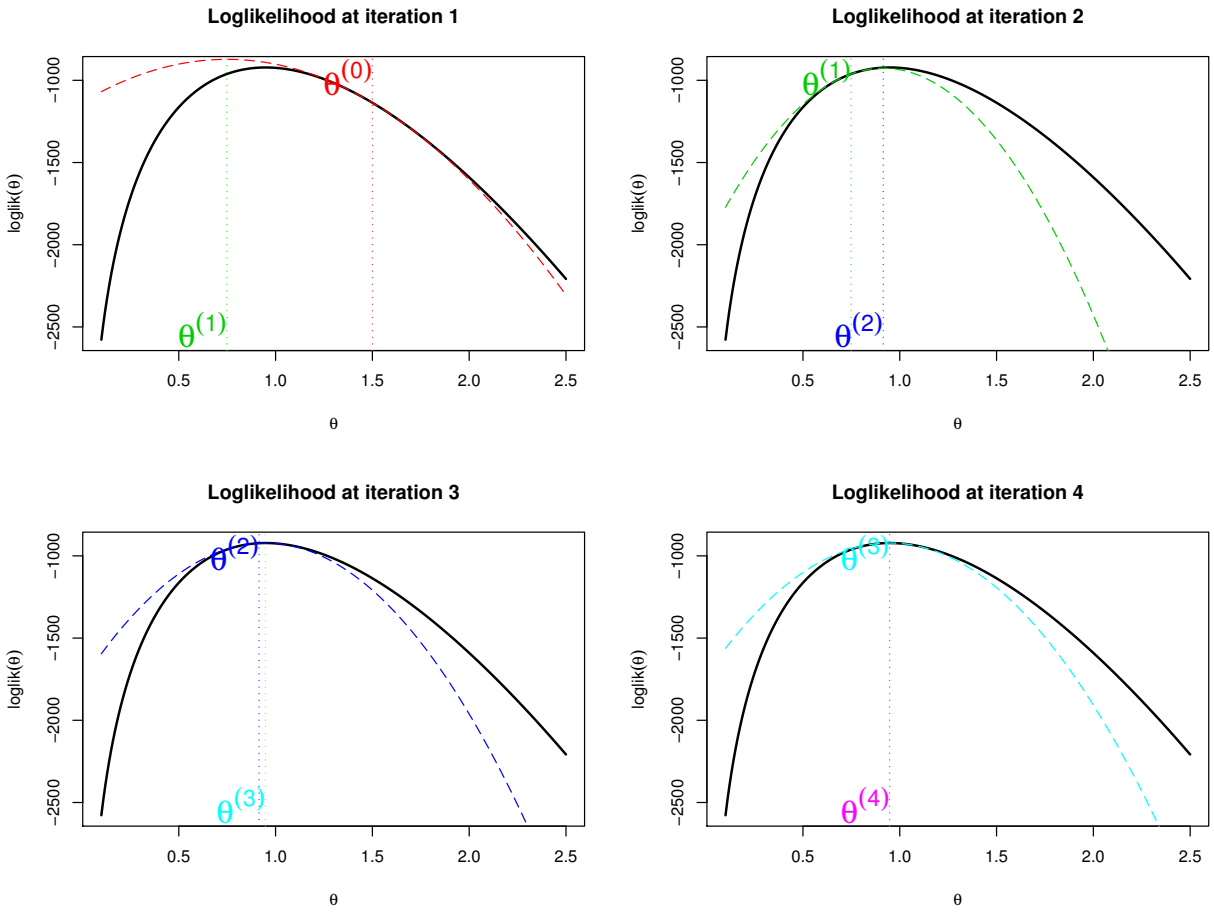


Figure 1: Newton's methods view in terms of maximising a function

Newton's method does not only have a graphical interpretation in terms of the objective function $\ell(\cdot)$, but also in terms of its derivative ("score") $\ell'(\cdot)$: At every iteration we find where the tangent to $\ell'(\cdot)$ in θ^{h-1} intersects the x-axis.

We can implement Newton's method in R using

```
theta <- 1.5
for (h in 1:100)
  theta <- theta - loglik.d(theta, x) / loglik.dd(theta, x)
theta
```

```
## [1] 1.014414
```

The above example carries out a fixed number of iterations. It would be better to check convergence at every iteration and abort if the change in θ is small enough.

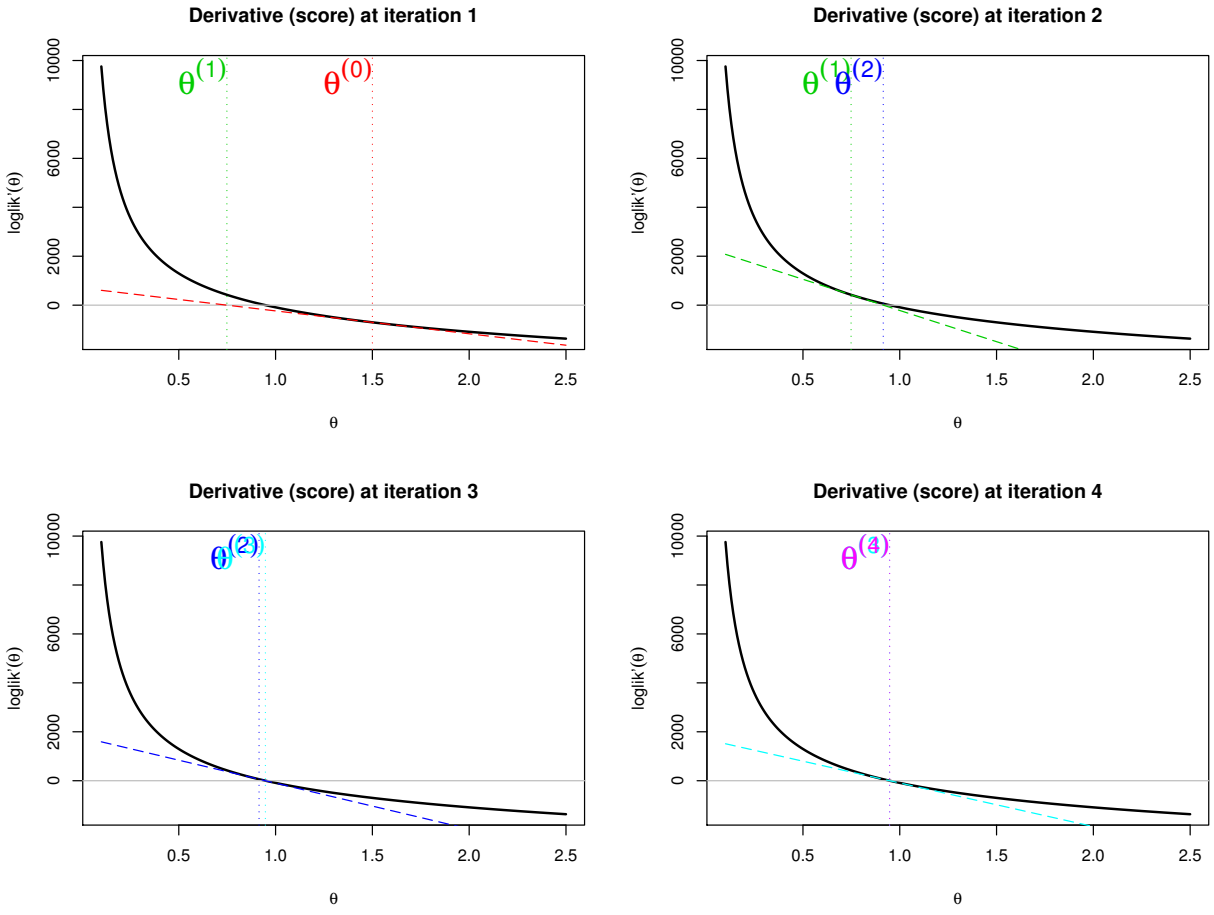


Figure 2: Newton's methods view in terms of solving that the derivative is zero

```r
theta <- 1.5
for (h in 1:100) {
  old.theta <- theta
  theta <- theta - loglik.d(theta, x) / loglik.dd(theta, x)
  if (abs(theta-old.theta)<1e-10)
    break
}
theta
```

```
## [1] 1.014414
```

Convergence of Newton's method

Once suitably close to a maximum / minimum, Newton's method will converge quadratically fast, i.e. the number of correct digits will double at each iteration. However, it is important to note that there is no guarantee that Newton's method will converge to a local extremum.

If we want to maximise a function and start Newton's method at a point where the objection function $\ell(\cdot)$ is convex, Newton's method might converge to a local minimum or diverge. As you can see from the way we derived Newton's method, it does not "know" the difference between a local maximum and a local minimum, largely because it solves $\ell'(\theta) = 0$.

Figure 3 shows an example: Trying to find the maximum of the p.d.f. of the $\mathsf{N}(0, 1)$ distribution with a starting point in its convex part.

It might also happen that Newton's method diverges or, if we are especially unlucky, it might cycle between two or more points.

Even if the objective function is concave, Newton's method is *not* guaranteed to converge to a local maximum. There are additional conditions on the derivative of the loglikelihood. Most simple statistical models (like the one we looked at in this lecture) however have these properties and thus Newton's method will converge for these models.

If the objective function is multimodal, it would be altogether unrealistic to expect Newton's method to converge to a global extremum.

Alternatives

There are many safer (but slower) alternatives for functions of one parameter like successive parabolic interpolation or golden section search.

The basic idea of successive parabolic interpolation is to update triplets of points, so that the objective function is (eventually) largest at the point in the middle. This (eventually) ensures that there is a local maximum somewhere between the point to the left-most and the right-most of the three active points. A new point is added by interpolating the three points with a parabola and computing its maximum.

A combination of these two methods is implemented in R's `optimize` function. The syntax of `optimize` is

```r
optimize(f, interval=c(from, to), ..., maximum=FALSE)
```

- `f` is the function you want to optimize (over its first argument).
- `from` and `to` are the lower and upper bound of the interval over which `f` is to be optimised.
- `...` allows for passing on further arguments to `f`.
- `maximum` indicates whether R should look for a (local) minimum or a (local) maximum.

To maximise the objective function from above over θ we could have also just used

```r
optimize(loglik, interval=c(0.1,10), x, maximum=TRUE)
```

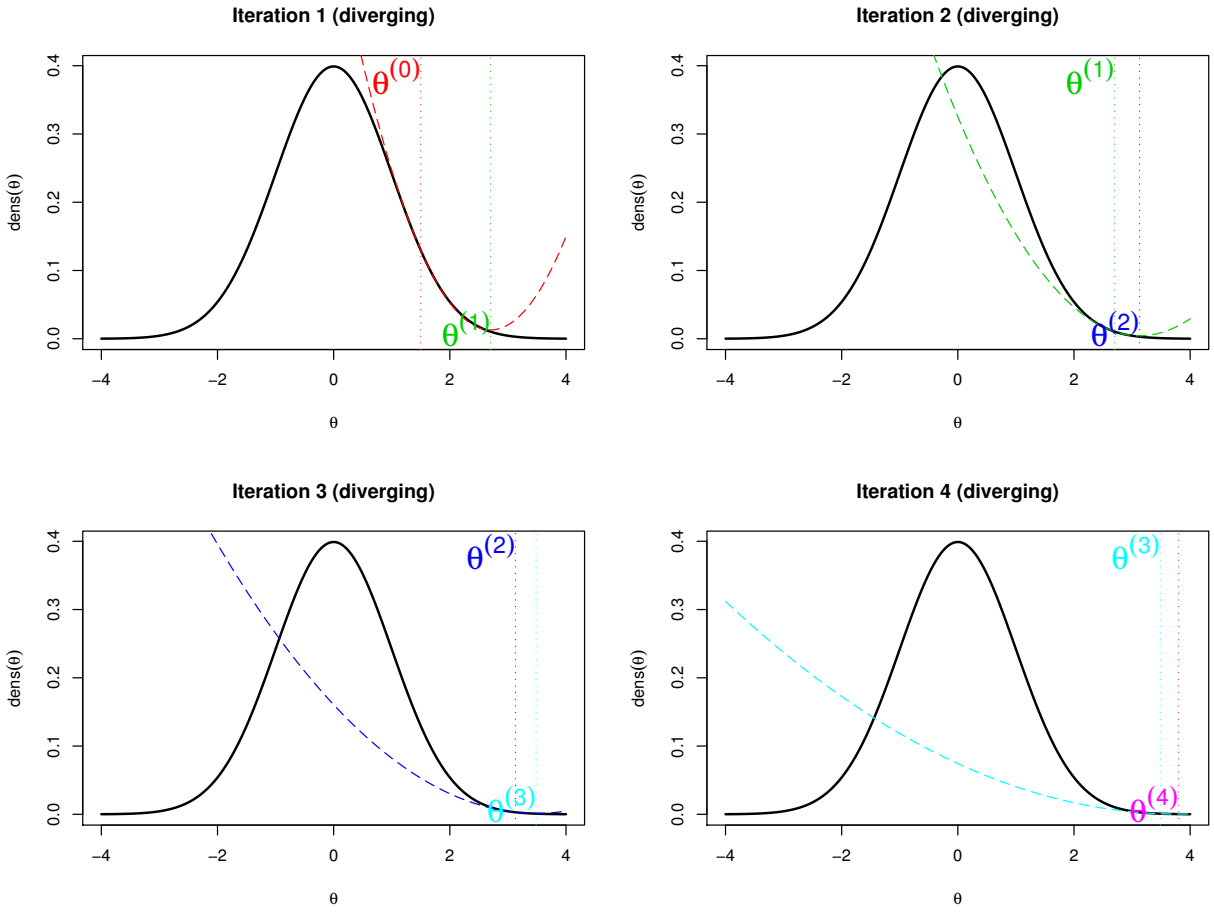


Figure 3: Netwon's methods fails when applied to a Gaussian probability density function

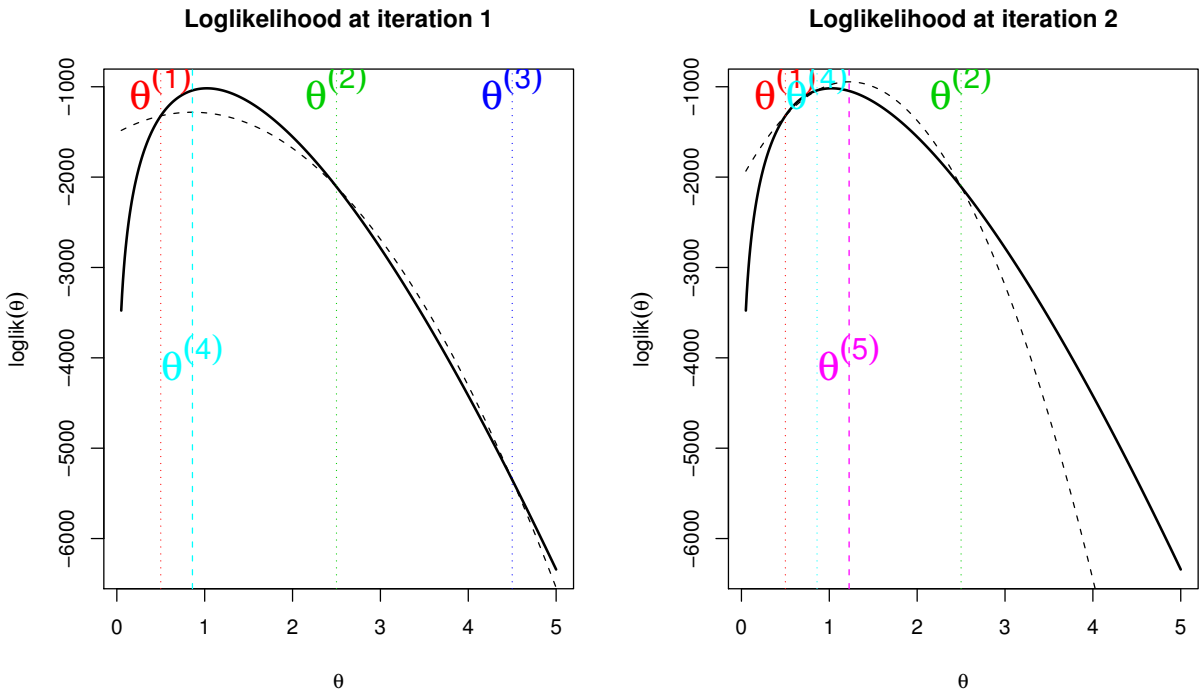


Figure 4: Successive parabolic search

```
## $maximum
## [1] 1.014414
##
## $objective
## [1] -986.0054
```

Another benefit of using `optimize` is that, in contrast to Newton's method, it does not require any derivatives.

Root finding in one dimension

In this section we will study numerical methods for solving a nonlinear equation of the form

$$f(\theta) = 0.$$

Note that an equation of the form $g(\theta) = h(\theta)$ can be written as $g(\theta) - h(\theta) = 0$, thus can be written as $f(\theta) = 0$ with $f(\theta) = g(\theta) - h(\theta)$.

Newton's method for solving a non-linear equation

In the first half of the lecture we have studied how Newton's method can be used to find a local extremum (i.e. a local minimum or maximum) of a function $f(\cdot)$. We have seen that Newton's method for optimising a function $f(\cdot)$ can also be seen as solving the equation that the derivative is 0, i.e. $f'(\theta) = 0$. This suggests that Newton's method can also be used to solve a nonlinear equation.

Suppose we want to solve the linear equation $a + b(\theta - \theta_0) = 0$, then the solution is $\theta = \theta_0 - \frac{a}{b}$, if $b \neq 0$.

Now suppose we want to solve the nonlinear equation $f(\theta) = 0$. We can approximate the function $f(\cdot)$ by its tangent in some point θ_0 (which also happens to be its first order Taylor approximation around θ_0):

$$f(\theta) \approx f(\theta_0) + f'(\theta_0) \cdot (\theta - \theta_0)$$

Thus the solution to the linear equation $f(\theta_0) + f'(\theta_0) \cdot (\theta - \theta_0) = 0$ is an approximate solution to the nonlinear equation $f(\theta) = 0$. The solution to this linear equation is

$$\theta = \theta_0 - \frac{f(\theta_0)}{f'(\theta_0)}$$

Geometrically speaking, θ is the intersection of the tangent to $f(\cdot)$ in θ_0 with the x-axis.

We obtain Newton's method for solving a nonlinear equation by repeatedly using this approximation, which gives the following algorithm:

1. Initialise θ to some value.
2. For $h = 1, 2, 3, \ldots$ iterate until convergence $\ldots$ Set

$$\theta^{(h)} = \theta^{(h-1)} - \frac{f(\theta^{(h-1)})}{f'(\theta^{(h-1)})}.$$

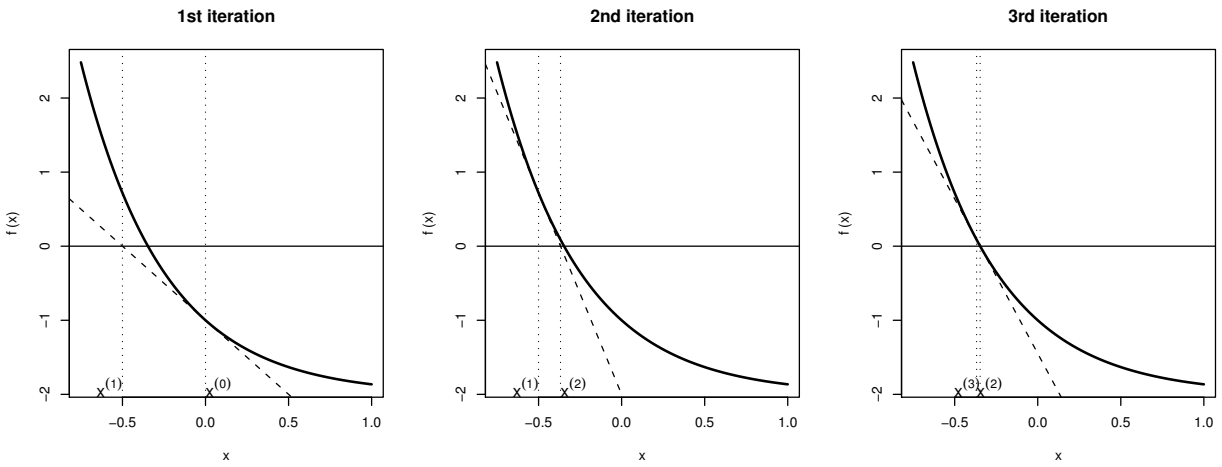


Figure 5: Illustration of Newton's methods for root finding

Example 2

Suppose we want to solve the equation $\exp(-2\theta) = 2$*, i.e.* $\exp(-2\theta) - 2 = 0$.

We start by defining the functions $f(\cdot)$ *and* $f'(\cdot)$.

```
f <- function(theta) {
  exp(-2*theta)-2
}

f.d <- function(theta) {
  -2*exp(-2*theta)
}
```

In its simplest form Newton's method can now be implemented as follows:

```
theta <- 0
for (h in 1:20) {
    theta <- theta - f(theta) / f.d(theta)
}
```

We obtain $\theta = -0.34657\ldots$*, which is nothing other than* $-\frac{1}{2}\log 2$*. The first three iterations of the method are shown below.*

Alternatives to Newton's method

Newton's method converges to a solution of the equation $f(x) = 0$ if started close enough to a solution. In general, there is however no guarantee that Newton's method will converge to a solution of the equation $f(x) = 0$. It can also happen that it diverges or that it cycles between two or more values. Thus it is usually better to use slower, but safer alternatives to Newton's method, such as the bisection method.

The bisection method starts with an interval (a, b), such that $f(a) \cdot f(b) < 0$, i.e. $f(\cdot)$ changes sign between a and b. In other words, if $f(\cdot)$ is continuous, $f(\cdot)$ intersects the x-axis somewhere between x and b, i.e. a solution to the equation $f(x)$ must lie inside the interval (a, b). Now we split the interval into two halves. Now either $f(\cdot)$ intersects the x-axis in the left half or in the right half (or in both[1]). If $f(a) \cdot f\left(\frac{a+b}{2}\right) < 0$ a solution must lie in the interval $\left(a, \frac{a+b}{2}\right)$, otherwise a solution must lie in the interval $\left(\frac{a+b}{2}, b\right)$. Now this interval is split into two halves again, etc. etc.

uniroot in R

The R function `uniroot` makes use of a more powerful version of the bisection method, Brent's method. The syntax of `uniroot` is

```
uniroot(f, interval=c(from, to), ...)
```

- `f` the function whose root $f(\theta) = 0$ we want to find.
- `from` and `to` are the lower and upper end of the interval in which the root is searched.
- Additional arguments (`...`) are passed on to `f`.

In contrast to Newton's method, `uniroot` does not require the derivative of $f(\cdot)$.

Example 3

Let's return to Example 2 and use `uniroot` *instead of Newton's method.*

```
uniroot(f, c(-1,1))
```

```
## $root
## [1] -0.3465738
##
## $f.root
## [1] 7.37133e-07
##
## $iter
## [1] 7
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

[1] In this case it has to intersect the x-axis at least twice in at least one of the intervals

Optimisation in more than one dimension

Netwon's algorithm (and extensions) can be equally defined to optimise functions over more than one parameter. We will not go through the details of the algorithm, but only discuss the functions already implemented in R to do this.

A sophisticated version of Newton's method which can work for multi-parameter optimisation is implemented in R's function `optim`. Its syntax is

```
optim(par, fn, gr, method="BFGS", ...)
```

- `par` are the initial values for the parameters to be optimised over.
- `fn` is the function to be optimised over its first argument.
- `gr` is the gradient (first derivative) of the function (optional - if omitted `optim` will approximate the gradient).
- Additional optional arguments (`...`) are passed on to `fn` and `gr`.

Note that `optim` performs minimisation by default. Use the additional argument `control=list(fnscale=-1)` to perform maximisation.

Example 4

The Rosenbrock function is a quartic function in two dimensions given by

$$f(\boldsymbol{\theta}) = 100(\theta_2 - \theta_1^2)^2 + (1 - \theta_1)^2$$

The Rosenbrock function is a test function frequently used in Numerical Optimisation although one can determine its minimum analytically. It is easy to see that f is the sum of two quadratic functions, so $f(\boldsymbol{\theta}) \geq 0$. The second quadratic is 0 if and only if $\theta_1 = 1$. If $\theta_1 = 1$ the first term is only 0 if $\theta_2 = 1$ as well, thus f has a global minimum at $(1, 1)$.

The first derivative (gradient) of the Rosenbrock function is

$$\mathbf{f}'(\boldsymbol{\theta}) = \left(\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}\right)(\boldsymbol{\theta}) = \left(-400\theta_1(\theta_2 - \theta_1^2) - 2(1 - \theta_1), 200(\theta_2 - \theta_1^2)\right)$$

We start with defining the Rosenbrock function in R. In order to be able to use `optim` *later on, we need to combine θ_1 and θ_2 into a single parameter*

```
rosenbrock <- function(theta, trace=FALSE) {
   if (trace)
     points(t(theta))
   100 * (theta[2] - theta[1]^2)^2 + (1-theta[1])^2
}

rosenbrock.gradient <- function(theta, ...) {
    c(-400*theta[1]*(theta[2]-theta[1]^2) - 2*(1-theta[1]), 200*(theta[2]-theta[1]^2))
}
```

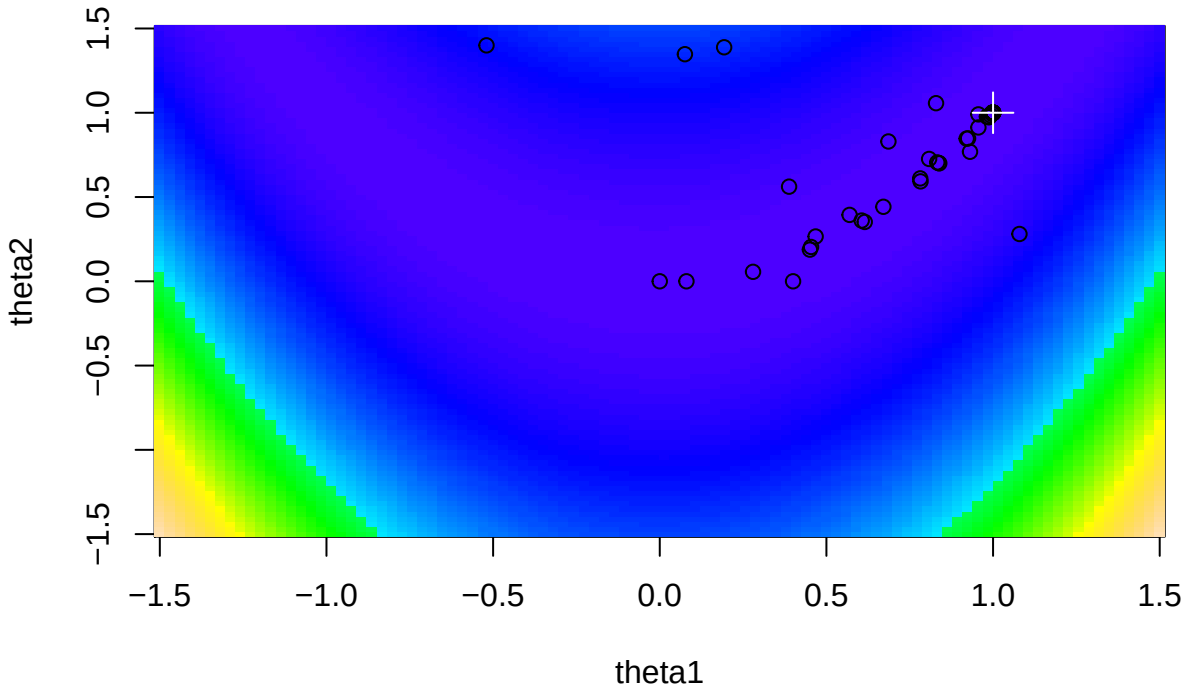
The function has an additional argument `trace`*, which, if set to* `TRUE` *adds a point to a plot, so that we can trace how the algorithm evolves. We next create an image plot of the objective function.*

```
theta1 <- seq(-1.5, 1.5, length.out=100)
theta2 <- seq(-1.5, 1.5, length.out=100)

val <- matrix(nrow=length(theta1), ncol=length(theta2))
for (i in seq_along(theta1))                          # Evaluate function on grid
    for (j in seq_along(theta2))
```

```
        val[i,j] <- rosenbrock(c(theta1[i],theta2[j]))

image(theta1, theta2, val, col=topo.colors(128))       # Draw image

sol <- optim(c(0,0), fn=rosenbrock, gr=rosenbrock.gradient,
             method="BFGS", trace=TRUE)

sol
```

```
## $par
## [1] 1 1
##
## $value
## [1] 2.311966e-18
##
## $counts
## function gradient
##       57       28
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
points(sol$par[1], sol$par[2], cex=2, pch=3, col="white") # Mark minimum found (should be (1,1))
```



`optim` also implements the Nelder-Mead algorithm. This algorithm does not make use of any derivatives, as is suited to objective functions which are not differentiable or not particularly smooth.

Nelder & Mead's method is implemented in `optim` and will be used when setting `method="Nelder-Mead"`.

```
optim(par, fn, method="Nelder-Mead", ...)
```

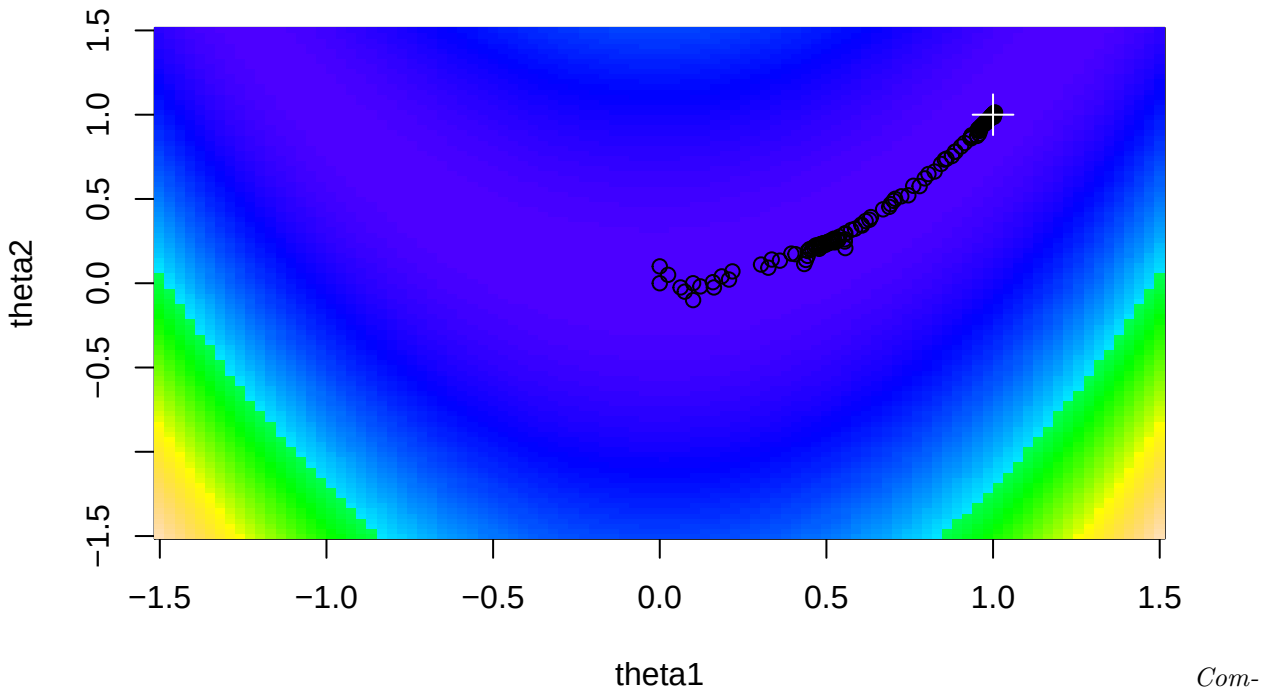
Example 5

In this example we apply the Nelder-Mead method to the Rosenbrock function from Example 4

```
image(theta1, theta2, val, col=topo.colors(128))       # Draw image

sol <- optim(c(0,0), fn=rosenbrock,
             method="Nelder-Mead", trace=TRUE)

sol
```

```
## $par
## [1] 0.9999564 0.9999085
##
## $value
## [1] 3.729052e-09
##
## $counts
## function gradient
##      169       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
points(sol$par[1], sol$par[2], cex=2, pch=3, col="white") # Mark minimum found (should be (1,1))
```



Compared to the BFGS method, the Nelder-Mead algorithm proceeds in much smaller steps.

If the objective function is reasonably smooth and differentiable, it is more efficient and more reliable to use the BFGS method, even if the derivative `gr` is not specified and the gradient needs to be approximated.

A few general remarks on the practical side of local optimisation

- It is usually a good idea to plot the objective function first.
- If there are multiple extrema the solution often depends on the initialisation. Run `optim` with several different starting values (especially if you are not sure whether there are local extrema).
- Be very critical of the results obtained. Optimisation algorithms are largely black boxes which can fail in many ways!
- Rescaling (or using the control option `parscale`) is often helpful. Most optimisation methods work best if each parameter has an equally strong effect.

Global optimisation

All methods considered so far (Newton's method, BFGS, Nelder&Mead, ...) perform (at most) *local* optimisation. However, local extrema are not necessary global extrema. Local extrema can be very far from the true global extremum. In the case of maximum likelihood estimation, getting stuck in a local extremum can yield invalid conclusions.

There are several strategies for global optimisation:

- One simple approach would be to evaluate function on a regular grid. Additionally we could start a local optimiser at each grid point. Unless we know some properties of the objective function, choosing an appropriate grid is difficult (How fine should the grid be?). However, most importantly, this method is computationally prohibitive, especially for high-dimensional parameter spaces.
- A usually more promising approach is to use stochastic optimisation methods. One such method is simulated annealing. (not covered here)

Numerical integration in R

The integral $\int_a^b f(x)\,dx$, which is the area under the function $f(\cdot)$ between a and b, can be approximated by the total area of the trapezoids shown in Figure 6.

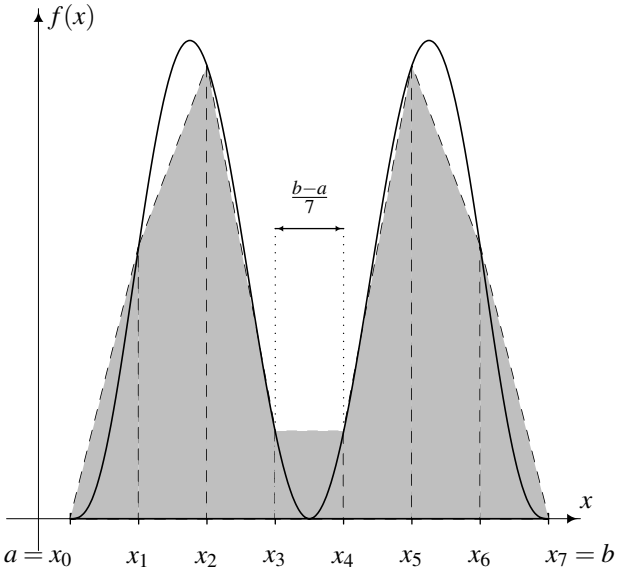


Figure 6: Approximation of an integral using trapezoids

As the number of trapezoids increases and the width of each trapezoid tends to 0, the two areas become identical. If we use n trapezoids of equal width $\frac{b-a}{n}$, the total area of the trapezoids is

$$\sum_{h=1}^{n} \frac{b-a}{n} \cdot \frac{f(x_{h-1}) + f(x_h)}{2} = \frac{b-a}{2n} f(x_0) + \frac{b-a}{n} \sum_{h=1}^{n-1} f(x_h) + \frac{b-a}{2n} f(x_n),$$

where $x_h = a + \frac{h}{n}(b-a)$.

This can be generalised to the following weighted approximation

$$\int_a^b f(x)\, dx \approx \sum_{h=0}^{n} w_h \cdot f(x_h)$$

The trapezoidal rule illustrated above corresponds to one specific choice of (equally spaced) support points $x_0, \ldots x_n$ and weights

$$w_h = \begin{cases} \frac{b-a}{2n} & \text{for } h = 0 \text{ or } h = n \\ \frac{b-a}{n} & \text{otherwise.} \end{cases}$$

It turns out that this choice is not optimal. Gaussian quadrature obtains a more precise approximation by not equally spacing the support points. Gaussian quadrature even obtains the exact result for polynomials of degree up to $2n-1$.

The R function

```
integrate(f=f, lower=a, upper=b, ...)
```

performs an adaptive version of Gaussian quadrature to compute (an approximation to) the one-dimensional integral

$$\int_a^b f(x)\, dx$$

`f` is hereby an R function whose first argument is the variable we want to integrate over. The function `f` must accept vectorised input: If the first argument of `f` is a vector, it has to return a vector of the same length with the i-th element being $f(x_i)$. Additional arguments can be passed on to `f` using the `...` argument of `integrate`. The bounds `a` and `b` may be infinite.

Just like solving equations or maximising functions numerically, numerical integration methods can fail in many ways, so don't trust the results too blindly. Especially, don't expect the methods to work well for functions with singularities, or any function which over the domain of integration cannot be approximated well by a polynomial.

Example 6 *The expected value of the exponential distribution with rate θ is $E(X) = \frac{1}{\theta}$, so if the rate $\theta = 2$ then the expected value should be 0.5. We will now check this using* `integrate`.

```
integrate(function(x) dexp(x, rate=2)*x, lower=0, upper=+Inf)
```

```
## 0.5 with absolute error < 8.6e-06
```

`integrate` only integrates numerically. Software like Maple, Mathematica, Wolfram Alpha or Maxima can also integrate symbolically, i.e. calculate the integral using the same integration rules as you would be using with pen and paper.

`integrate` can only compute univariate integrals. The function `adaptIntegrate` from the package cubature can be used to compute integrals in higher dimension. High-dimension integrals are however typically easier to compute using Monte-Carlo methods (not covered here).