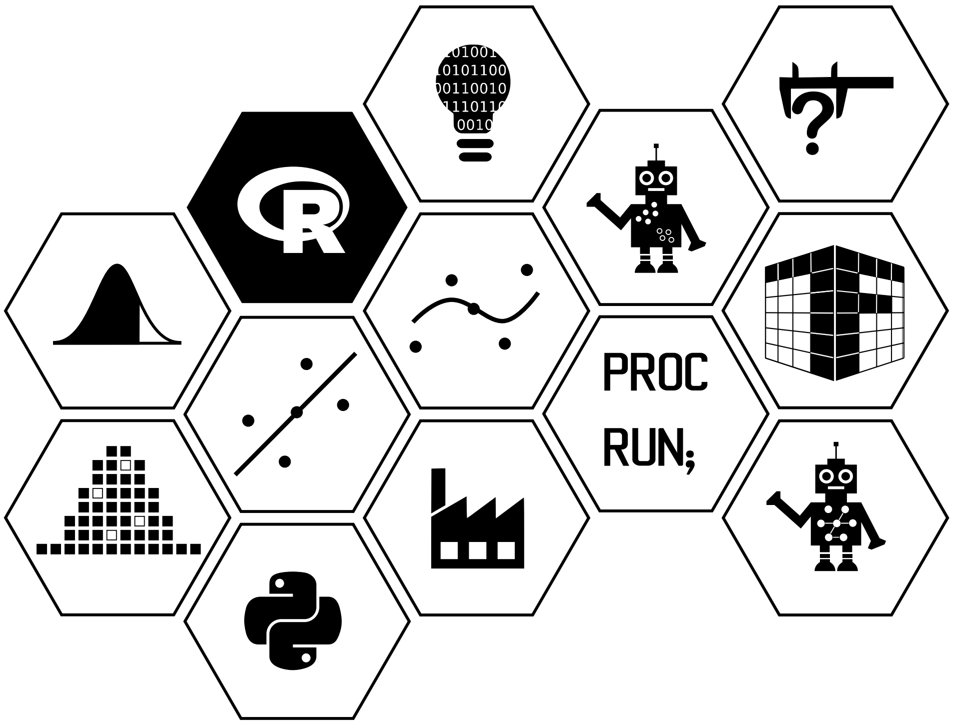
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 1:

Introduction to Computing and R



Installing R

R

You need to have access to R for this course. You can download R for free from CRAN.

R is available for Windows, Mac OS and Linux as well as some less common platforms.

CRAN

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

You can download the standard version of R from CRAN.

Downloading and installing R for Windows

To download the Windows installer of R, just enter the following URL (or click on the clink).

https://cran.r-project.org//bin/windows/base/release.html

This will download the most installer for the most recent version of R. Alternatively, you can go to the main CRAN page, https://cran.r-project.org/, and then click on "Download R for Windows", click on "base" and then on "Download R x.y.z for Windows" (where x.y.z is the current version number of R).

You can then run the installer, accepting all default settings.

Downloading and installing R for Mac

To download the Windows installer for Mac, just enter the following URL (or click on the clink).

https://cran.r-project.org/bin/macosx/

From here, select the most recent version of R and the .pkg file will automatically download. This file will be in the form "R-x.y.z" (where x.y.z is the current version number of R).

Once the file is opened, the installer will open and you can select the default settings.

RStudio

It is recommended that you also download and install RStudio Desktop, a powerful integrated development environment (IDE) for R. RStudio contains a much better code editor. It has, for example, syntax highlighting, i.e. it will automatically display your code in different colours to make it easier and quicker to read the code. Even though other IDEs, such as Visual Studio Code, or Emacs can also be used with R, RStudio is by far the most popular among R users.

RStudio is just a front-end for R, so to be able make use of RStudio, you need to also have R installed.

RStudio Desktop

https://www.rstudio.com/products/rstudio/download/

RStudio Desktop Open Source is available for free from RStudio.

Installing RStudio for Windows/Mac

Go to

https://www.rstudio.com/products/rstudio/download/

and scroll down to the section "All installers", then click on "RStudio-x-y-z.exe" in the first row of the table for a Windows install, or click on "RStudio-x-y-z.dmg" for a macOS install. This should start the download of the RStudio installer.

You can then run the installer, accepting all default settings (for macOS, you will need to drag and drop the application into the applications folder once open).

Supplementary material:
Posit Cloud

Posit also offer a cloud version of RStudio, available at

https://posit.cloud/

With Posit Cloud, you can get access to an R session even if you cannot install R locally (e.g. on a tablet or mobile phone). It can also be extremely useful when working on collaborative projects and when you require more memory or processing power (paid-for plans only). A free plan is available.

A step-by-step tutorial is available at

https://posit.cloud/learn/guide

Supplementary material:
R in google Colab

Google Colab, a Google-hosted version of Jupyter notebooks, also has an R mode, though it is primarily designed for Python. If you have a Google account you can create a new R notebook using the link

https://colab.research.google.com/#create=true&language=r

Colab is free to use and you R code will be run on Google servers. Colab is especially worth looking at if you are used to Jupyter notebooks (You can, of course, also use an R kernel in a local installation of Jupyter). We will use Colab in some of our live sessions for working through tasks.

Posit and RStudio

RStudio became a part of the Posit family of products in 2023. Despite the name change, the features of RStudio have not changed in any way, and indeed there has been increased flexibility in RStudio such as integration with Python code.

R packages

R comes with a default selection of packages, which should cover your "basic needs" in terms of data management, data visualisation and modelling. However, there is a large selection of "add-on" R packages available on CRAN, some of which we will use in this course. You can only use these R packages once you have installed them.

Imagine you want to use an R package called `ggplot2` (which we will use later in this course). In order to be able to use it, you first need to install it. You can do so by entering

```
install.packages("ggplot2")
```

into R. This will download and install the package `ggplot2`, as well as any other packages `ggplot` uses.

Alternatively, you can click on the tab "Packages in the bottom-right panel, and then click on"Install".

You can then enter the name of the package you want to install and click on "Install"

Once you have installed an R package, you can load it using the function `library`

```
library(ggplot2)
```

Now you can draw a pretty plot using `ggplot2`, for example using

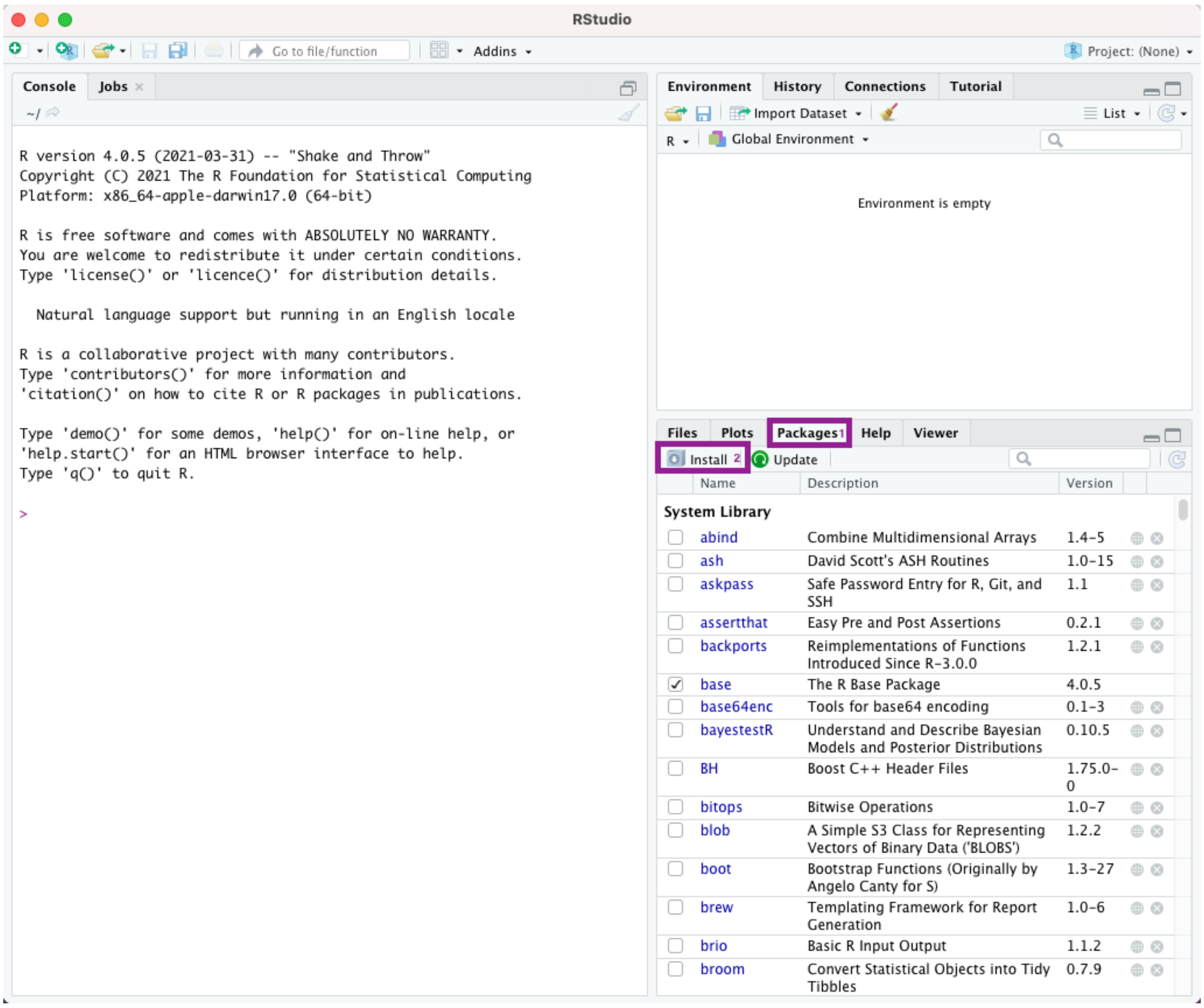
```
ggplot(data=diamonds) + geom_point(aes(carat,price,colour=cut))
```

(You will learn more about plotting using `ggplot2` in Week 6 of the course.)

If you run `library(somepackage)` and obtain the error message

```
Error in library(somepackage) : there is no package called 'somepackage'
```

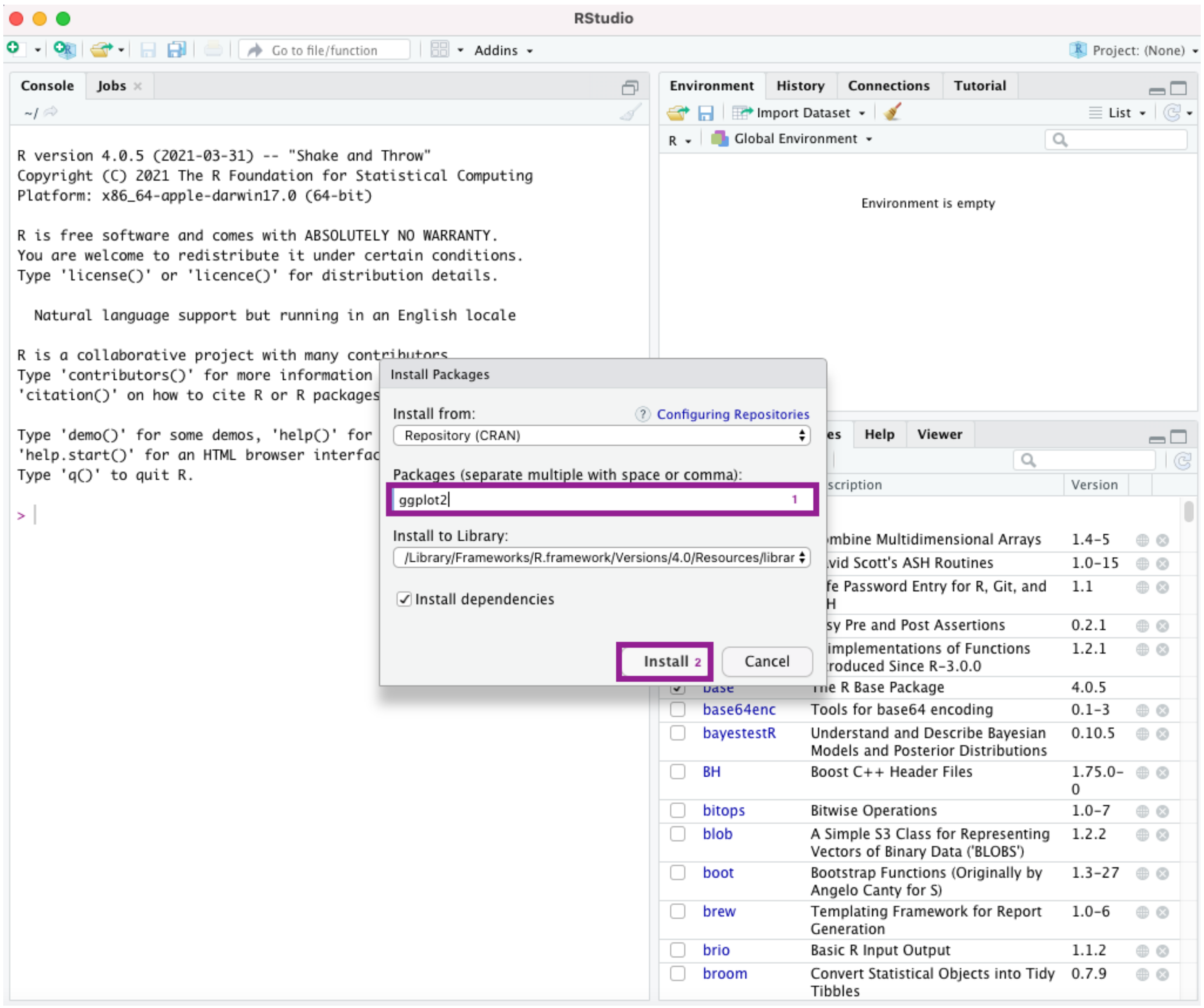
then you do not have this package installed and need to install it.



```
R version 4.0.5 (2021-03-31) -- "Shake and Throw"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

	Name	Description	Version
	System Library		
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	ash	David Scott's ASH Routines	1.0-15
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.2.1
✓	base	The R Base Package	4.0.5
☐	base64enc	Tools for base64 encoding	0.1-3
☐	bayestestR	Understand and Describe Bayesian Models and Posterior Distributions	0.10.5
☐	BH	Boost C++ Header Files	1.75.0-0
☐	bitops	Bitwise Operations	1.0-7
☐	blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.2
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-27
☐	brew	Templating Framework for Report Generation	1.0-6
☐	brio	Basic R Input Output	1.1.2
☐	broom	Convert Statistical Objects into Tidy Tibbles	0.7.9

Figure 1: Selecting the installation menu for installing packages



```
R version 4.0.5 (2021-03-31) -- "Shake and Throw"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

  Natural language support but running in an English locale

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

Type 'demo()' for some demos, 'help()' for
'help.start()' for an HTML browser interfac
Type 'q()' to quit R.

>
```

Install from: Configuring Repositories
Repository (CRAN)

Packages (separate multiple with space or comma):
ggplot2 1

Install to Library:
/Library/Frameworks/R.framework/Versions/4.0/Resources/librar

✓ Install dependencies

		scription	Version
		mbine Multidimensional Arrays	1.4-5
		vid Scott's ASH Routines	1.0-15
		fe Password Entry for R, Git, and H	1.1
		sy Pre and Post Assertions	0.2.1
		implementations of Functions roduced Since R-3.0.0	1.2.1
✓	base	The R Base Package	4.0.5
☐	base64enc	Tools for base64 encoding	0.1-3
☐	bayestestR	Understand and Describe Bayesian Models and Posterior Distributions	0.10.5
☐	BH	Boost C++ Header Files	1.75.0-0
☐	bitops	Bitwise Operations	1.0-7
☐	blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.2
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-27
☐	brew	Templating Framework for Report Generation	1.0-6
☐	brio	Basic R Input Output	1.1.2
☐	broom	Convert Statistical Objects into Tidy Tibbles	0.7.9

Figure 2: Installation menu for packages

About R

History

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 sell a value-added version called S-Plus (now marketed as Spotfire S+), which contains a graphical user interface. S-Plus used to dominate the high-end market (academic and industrial research).

R is an implementation of the S programming language, which is in many respects superior to the original S system. 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 free software, you can obtain it for free ("free as in free beer"), 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 ("free as in free speech"). R is extensible and a large selection of extensions packages can be obtained from CRAN.

Why use a command-line programme?

R is essentially a command-line programme, i.e. you control R by typing commands. Whilst command-line programs were very common until the mid-1990s, most software nowadays uses menu-driven graphical user interfaces (GUI), like Word, Excel or SPSS. Why should we then use an "old-fashioned' ' command-line programme 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 it in the way the authors have designed it to be used.

Command-line based programmes 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. SPSS) offer the option of using macros. However these macros are often very clumsy and less elegant than R code.

Integrated development environment like RStudio have many features that make coding in R a lot less daunting.

Why use R?

These days, R and Python are *the* platforms for Data Science. No other language is coming close to the market share of R and Python in Data Science.

Reasons for R's dominant position are a very good graphics engine and the large number of extension packages (more than 19,000), so there are only few statistical methods not implemented in R. Though R as a language is not the most elegant (especially compared to more recent languages such as Julia), there is a suite of R packages ('tidyverse') which we will look at in this course and which provide very elegant tools for data manipulation and visualisation. Other languages lack such elegant solutions, though some aspects of the "philosophy" of tidyverse have now been ported to other languages, notably Python.

When not to use R

R is however not always the most suitable tool: there are many situations in which one should think about alternatives which might be better suited for the task at hand.

- In terms of performance R is certainly not one of the fastest languages around. For this reason R functions can make use of compiled FORTRAN, C or C++ code, but writing code in these "low-level" languages can be time-consuming and complex.
- R needs to store the entire workspace in memory, so unless you have large amount of memory, R will struggle with truly big data sets. However, R can interface big data systems such as Spark, in which case R only references the data, which in turn is stored in Spark. You will learn more about Spark in the Large Scale Computing course.
- If close integration with production systems is desired, R might not be the ideal tool, though packages like plumber have made it a lot easier to create external interfaces for R code.
- Over the years, Python has become the preferred choice in the Machine Learning community. Python is the native interface for many key machine learning libraries such as scikit-learn, PyTorch or Tensorflow. So if you task requires high-performance Machine Learning libraries, Python is likely to be your better choice.
- Though Shiny provides an easy-to-use way of creating interative web apps, creating highly-interactive visualisations is difficult just using R. You might want to consider libraries such as d3 in this case. We will cover Shiny in Week 10 of the course.

Comparing R to other programming languages

R is an interpreted language In an interpreted language like R, programme code is executed step-by-step, without (explicitly invoked) prior translation to machine code ("compilation"), as would be required for languages such as C, C++ or Java.

Interpreted languages are typically easier to debug as code can easily be run interactively on a line-by-line basis. However there typically is a performance penalty involved. Compiled languages are typically a lot faster. This is why many operations in R such as matrix multiplication are under the hood implemented in Fortran or C.

Like many interpeted languages, R nowadays has a just-in-time compiler (JIT), which can translate the commands to be executed into an intermediate binary format, which is quicker to execute. However, this process is almost invisible to the user.

R is a dynamically-typed language Programming languages like C, C++ or Java require all variables to have a declared type before they can be used. If we want to assign the value 12 to a variable in a C programme we have to use

```
int var ;      // Declare the variable var to be an integer.
var = 1;
// We won't be allowed to set var to a character var="a"
```

This is not necessary in R. In this sense, R is like for example Python or Javascript. It will decide at run time what type to use for a variable, so we can simply set

```
var <- 1
var <- "a"
```

R is however a typed language, once defined, variables have types, so trying to run

```
1+"a"
```

produces an error message, while it would give `"1a"` in some other programming languages (for example JavaScript).

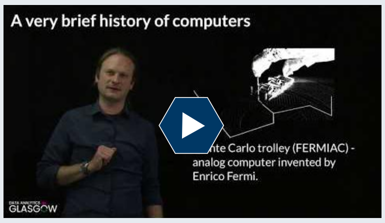
R is object-oriented In R and other object-oriented languages, data constructs can behave differently depending on their type. For example, the R method `summary` provides a summary for an object and the method `plot` plots an object. What these methods will actually do depends on what type of object they are invoked for. R's object orientation is a lot less visible than that of other programming languages, so many users don't even notice it. We will discuss object oriented programming in R in Week 9.

R is garbage-collected Just like in Java or Python, R looks after the memory management for you. Objects that are no longer referred to will be automatially removed from the memory ("garbage-collected").

About computers and computing

Before we take a look at R coding in more detail, let's take at computing in general. Data science wouldn't exist if computers weren't that powerful.

A brief history of computing


A very brief history of computers
Monte Carlo trolley (FERMIAC) - analog computer invented by Enrico Fermi.
GLASGOW

A brief history of computing

https://youtu.be/mjDbSsKkVdc

Duration: 3m52s

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 $\times$ 1,000 entries, i.e. we must compute 1,000,000 numbers, each of which is a sum of 1,000 products, meaning we need to carry out $2 \cdot 10^9$ additions and multiplications, which 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 device with a mid-range M1 processor purchased in 2021).

```
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
##   0.374   0.002   0.376
```

```
                                        # The third figure is the elapsed time in seconds.
```

So, this has only taken less than 1/100th of a second (looking at the system time here).

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
##   0.042   0.001   0.044
```

```
                                        # 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 and as a reminder to be careful when using a computer to perform even basic arithmetic.

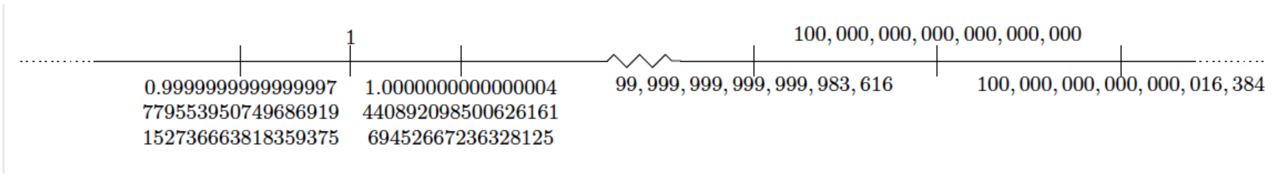


Figure 3: Illustration of the gaps between floating-point numbers and the nearest next smallest or largest number that can be represented

Isn't addition commutative (i.e. the order of terms usually doesn't matter)?

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, as this is the same sum, just with the terms re-arranged a little. 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.

The reason behind this is that 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 this way 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, a so-called 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 reason causing the computer to get the wrong result in the above example 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 Figure 1).

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.

This is in no way a problem that is unique to R. In the above example, Excel obtains exactly the same (wrong) results as R.

More supposedly 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. (Again, Excel doesn't get this "right" either, just enter `=0.1+0.1+0.1-0.3>0` into a cell)

Almost all modern computers (as opposed to calculators) internally use a binary system instead of the decimal system we are used to. In binary representation, the digits of 0.1 are periodic

$$0.1 = 0 \times 1 + 0 \times \frac{1}{2} + 0 \times \frac{1}{4} + 0 \times \frac{1}{8} + 1 \times \frac{1}{16} + 1 \times \frac{1}{32} + 0 \times \frac{1}{64} + 0 \times \frac{1}{128} + 1 \times \frac{1}{256} + 1 \times \frac{1}{512} \ldots$$

i.e. on a computer 0.1 (decmial) is actually $0.000110011\ldots$ (binary). So on a computer 0.1 is (just like $\frac{5}{6}$) not a “nice” number.

Task 1 (Binary numbers).

Explain why 0.3 is $0.01001100110\ldots$ in binary representation.

Thus our calculation is in binary numbers

$$0.000110011\ldots + 0.000110011\ldots + 0.000110011\ldots - 0.010011001\ldots$$

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.

For this reason, pocket calculators perform all their arithmetic in base 10 (even though this is a lot less efficient than base-2 arithmetic) so that school children are not tripped up by these issues.

The take-home message of these examples is that we should always expect small numerical errors and never test whether two numbers are exactly equal, but rather test whether their difference is quite small (say less than 10^{-8}).

Ariane V

Numerical overflow (and underflow) can have devastating effects. For example, the first Ariane V rocket was lost on its maiden flight in 1996 only 37 seconds after take-off, when it got out of control and was destroyed by its own safety mechanism.

It turned out that the crash was triggered by the conversion of a 64-bit floating point number to a 16-bit signed integer. 64-bit floating point numbers can store much bigger numbers than 16-bit signed integers. When the on-board computer attempted to convert one such number, it was too large for a 16-bit signed integer and caused a hardware exception, which in turn caused parts of the on-board computer to shut down. The software engineers had disabled the software handling of this exception to make the code run faster. The resulting economic loss was more than £200 million.

Medical studies

Data analysts and statisticians can also run into similar numerical problems. From the New York Times of June 5th, 2002:

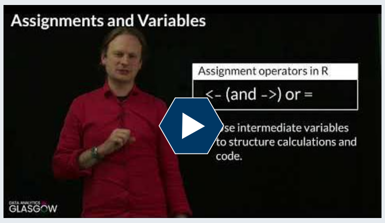
Revisiting their own data with new methods, scientists who conducted influential studies that linked sooty air pollution with higher death rates have lowered their estimate of the risk posed by bad-air days. … The work has been published for several years in a variety of leading journals like *The New England Journal of Medicine* and *The American Journal of Epidemiology*. The project, the National Morbidity, Mortality and Air Pollution Study, was given extra weight by policy makers because of the reputation of the Health Effects Institute and the Johns Hopkins group, led by Dr. Jonathan M. Samet, chairman of epidemiology at the public health school there. … As part of a continuing effort to check for flaws, those scientists in recent weeks used a new method to look at their figures and obtained different results. They re-examined the original figures and found that the problem lay with how they used off-the-shelf statistical software to identify telltale patterns that are somewhat akin to ripples from a particular rock tossed into a wavy sea. Instead of adjusting the program to the circumstances that they were studying, they used standard default settings for some calculations. That move apparently introduced a bias in the results, the team says in the papers on the Web.

The authors had used too lenient convergence criteria incorrectly suggesting that the model had already converged.

Don’t worry too much about these issues for now, but beware that if you are not careful one day you will get tripped up by one of these issues. You will come across one or the other oddity which is due to numerical problems during the course or later on in your life as an R user.

R as a calculator

Basic arithmetic operators


Assignments and Variables
Assignment operators in R
<- (and ->) or =
Use intermediate variables to structure calculations and code.
GLASGOW

R as a calculator

https://youtu.be/Gib3Wk2FFi8

Duration: 6m58s

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

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)`.

Task 2.

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

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
```

if you really want to perform arithmetic using complex numbers.

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

Example 2.

`log(0)` returns `-Inf`, as $\lim_{x\to 0} \log(x) = -\infty$.

`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`. Assignments can be also made using the function `assign`. It offers a few additional options, type `?assign` to read the corresponding help page.}

Supplementary material:
Differences between the assignment operators

<− and = do not always behave the same way. Inside a function call, = is used to provide named arguments, whereas <− is always an assignment. So, for instance, `a = c(b = 3, 4)` is not the same as `a <- c(b <- 3, 4)`. The former creates a vector `a` with the entries 3 and 4 and the first entry having the name `b`. The latter creates a variable `b` containing the number 3 and a vector containing the numbers 3 and 4 (without any labels).

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, so has so-called `camelCase` (e.g. `twoWords`).

Supplementary material:
Variable names with reserved characters

If you want to use a variable name that contains a reserved chracter like a space or a mathematical operator, you need to enclose it in "backticks".

```
`cat + mouse` <- 1
`cat + mouse` + 1
```

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

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 <- 40
b
```

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

`b` will stay `2`, even though `a` will be `40`. In this sense R is different to Microsoft Excel, where cells will update their values automatically based on the formula entered.

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"
##   [4] "B"                  "buildjob"           "C"
##   [7] "cat + mouse"        "conf"               "current_checksum"
##  [10] "f"                  "fn"                 "i"
##  [13] "master"             "n"                  "previous_checksum"
##  [16] "rebuild"            "self"               "subfn"
##  [19] "targetfn"           "tex_macros"         "write.value"
##  [22] "x"
```

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

Task 3 (Loan example revisited).

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

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

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. In more general, 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 $(P \cdot (1 - \alpha_{10}))$.

Functions

We will look at functions later on in this course in more detail (in Week 8), but we will take a quick look at them now. If we want to repeatedly perform the same calculation, we shouldn't be copying and pasting code. In this case we are better off defining a function. The example below illustrates how to define a function.

Example 3 (Loan example as a function).

Let's look again at the loan example from the video and the previous task. Suppose we want to calculate the repayments for different interest rates. Rather than setting `interest.rate` to each of the values and then repatedly submitting the code for the calculation, we are better off defining a function. We do this by wrapping the code into a function definition. Rather than setting the information we need to know to perform the calculation, we take it in as function arguments.

```
repayment <- function(n, loan, interest.rate) {
    v <- 1 / (1+interest.rate)                # effective annual discount factor
    payment <- loan * (1-v) / (v*(1-v^n))     # yearly payments
    return(payment)                           # Return the answer
}
```

We can then calculate the repayment for different interest rates just using

```
repayment(n=20, loan=9000, interest.rate=0.05)
```

```
## [1] 722.1833
```

```
repayment(n=20, loan=9000, interest.rate=0.10)
```

```
## [1] 1057.137
```

```
repayment(n=20, loan=9000, interest.rate=0.15)
```

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

Answers to tasks

Answer to Task 1 (Binary numbers).

$$0.3 = 0 \times 1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{4} + 0 \times \frac{1}{8} + 0 \times \frac{1}{16} + 1 \times \frac{1}{32} + 1 \times \frac{1}{64} + 0 \times \frac{1}{128} + 0 \times \frac{1}{256} + 0 \times \frac{1}{512} \dots$$

Answer to Task 2. 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
```

Answer to Task 3 (Loan example revisited). 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
```

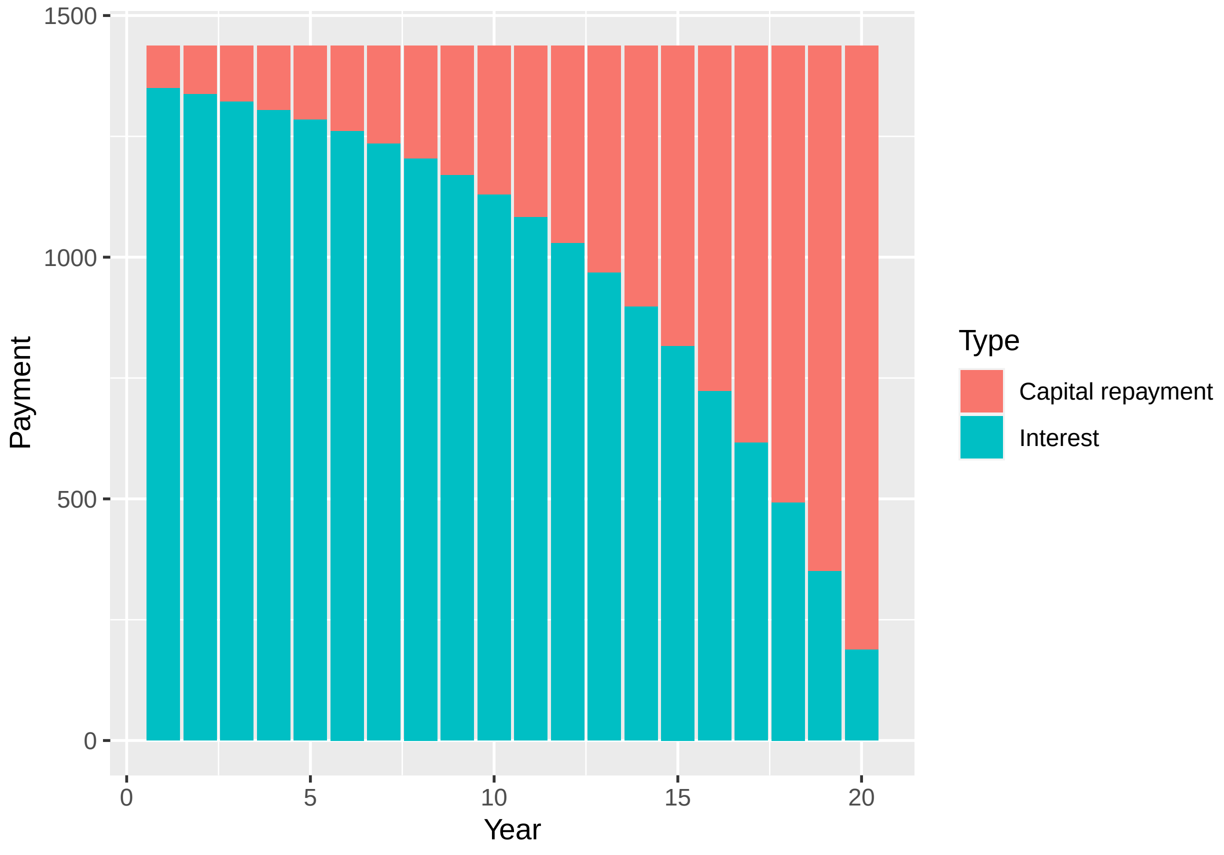
```
interest10 <- payment * (1-alpha10)     # interest part
interest10
```

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

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

We can also create a graph illustrating how the payments are split and how fast the loan is being repaid (you will learn later on in this course how to make sense of the code below).

```
k <- 1:n                                # create vector with all possible k
alpha <- v^(n+1-k)                      # split factors
capital <- alpha * payment              # capital repayments
interest <- (1-alpha) * payment         # interest part
data <- data.frame(Year=1:20, rbind(data.frame(Type="Capital repayment",
                                               Payment=capital),
                                    data.frame(Type="Interest",
                                               Payment=interest)))
library(ggplot2)
ggplot(data=data) + geom_col(aes(Year, Payment, fill=Type))
```


1500
1000
500
0
Payment
0
5
10
15
20
Year
Type
Capital repayment
Interest