comillas.edu

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

Efficient Programming

Contents

Prerequisites

- To begin analysing the performance of your code in R, the `microbenchmark` and `profvis` libraries are needed.
- `microbenchmark` library measures the computing time needed to execute a function.
- `profvis` library to create interactive visualizations of code profile analysis.
- `profmem` library to profile the memory usage of an R expression

What is efficiency?

- Efficiency η has a formal definition as the ratio of work W done per unit effort Q:

$$\eta = \frac{W}{Q}$$

- The narrower definition of efficient programming is **algorithmic efficiency**: how quickly the computer can undertake a piece of work given a particular piece of code.
- The broader definition for efficient programming is **programmer productivity**: amount of ***useful*** work a person can do per unit time.

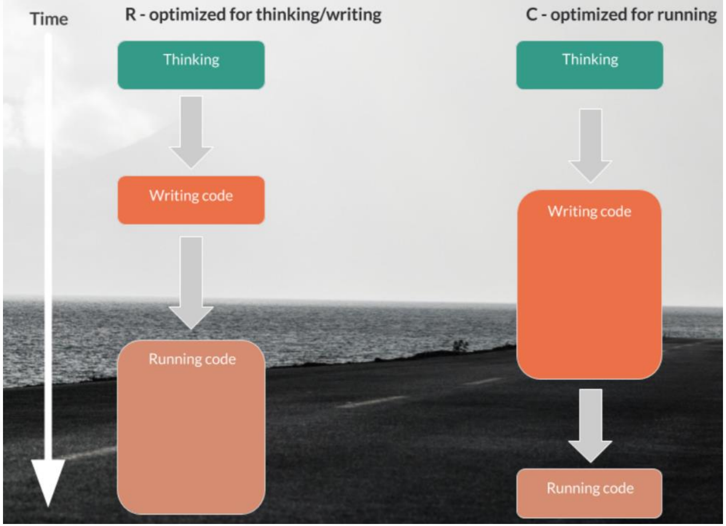
What is efficiency?

- By the end of this presentation, you should be able to write efficient code from both algorithmic and productivity perspectives.
- Efficient code is **concise, elegant** and **easy to maintain** (vital when working in a large project).
- Be careful when applying the lessons learnt here because efficient programming in R might not be efficient programming in other languages!
- *Efficient R programming can be interpreted as finding a solution that is **fast enough** in terms of **computational efficiency** but **as fast as possible** in terms of **programmer efficiency** without compromising hardware resources by **space complexity.***

Introduction

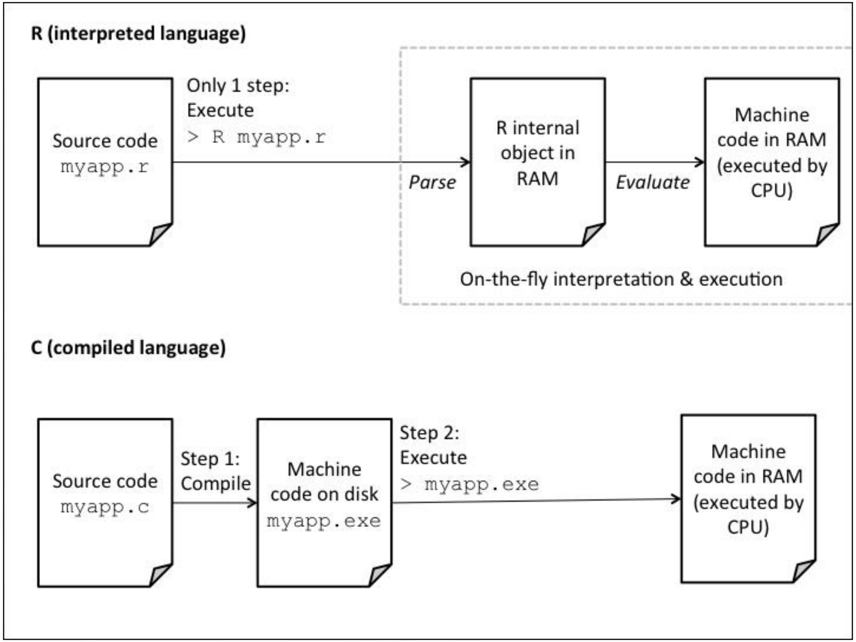
Unfair comparison

- Low-level programming languages like C o C++ are said to be way faster than “interpreted” languages like R or Python.
- However, this comparison is usually referred to **computational efficiency**.



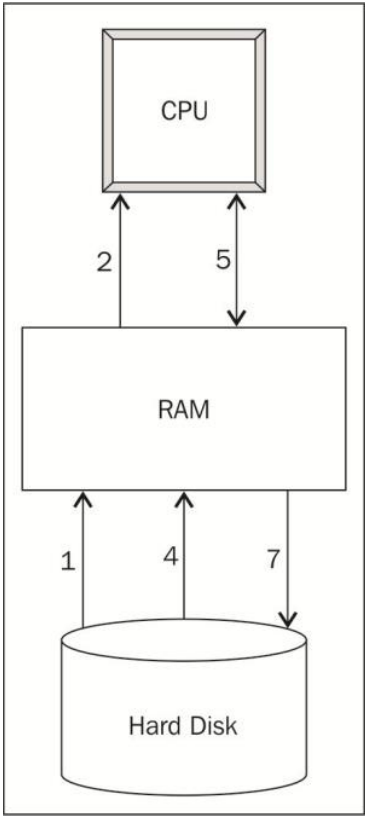
Unfair comparison

- Both types of languages run in a different manner.



3 constrains on R performance

- The speed and performance of the **CPU** determines how quickly computing instructions. This includes the interpretation of the R code into machine code and the actual execution of the machine code to process the data.
- The size of **RAM** available on the computer limits the amount of data that can be processed at any given time.
- The speed at which the data can be read from or written to the hard disk, that is, the speed of the **disk input/output (I/O)** affects how quickly the data can be loaded into the memory and stored back onto the hard disk.



Cross-transferable skills

There are many things that will help increase efficiency as the advice to 'think more work less'.

Evidence also suggests that good diet, physical activity, plenty of sleep and a healthy work-life balance can boost speed and effectiveness.

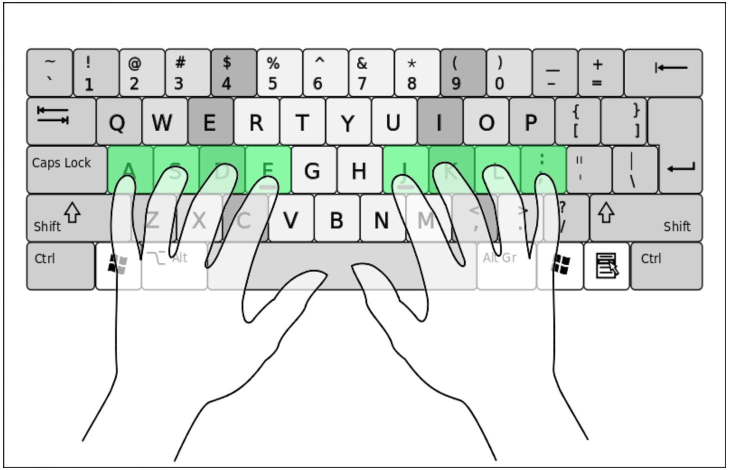
However, this presentation is about programming and these topics are out of the scope.

Cross-transferable skills

- **Touch typing**

This skill can be relatively painless to learn, and can have a huge impact on your ability to write, modify and test code quickly. Learning to touch type properly will pay off in small increments throughout the rest of your programming life.

Check Klavaro and Typefaster to get in the habit of touch typing. You may even want to change to English keyboard!



Cross-transferable skills

- **Consistent style and code convention**

Getting into the habit of clear and consistent style when writing anything will have benefits in many other projects, programming or non-programming.

You can check the books from Robert C. Martin (Uncle Bob) the *Clean code* and *The clean coder* to know more about good habits when writing code.

Cross-transferable skills

- **Benchmarking and profiling**

Benchmarking is the process of testing the performance of specific operations repeatedly. Profiling involves running many lines of code to find out where bottlenecks lie.

In some ways benchmarks can be seen as the building blocks of profiles. Profiling can be understood as automatically running many benchmarks, for every line in a script, and comparing the results line-by-line.

Cross-transferable skills

- **Benchmarking in R**

Using the `microbenchmark` library we can measure a function computing time executing it by default a 100 times. This is important to get a more reliable measure.

```
library("microbenchmark")
df = data.frame(v = 1:4, name = letters[1:4])
microbenchmark(df[3, 2], df[3, "name"], df$name[3])
# Unit: microseconds
#           expr   min    lq  mean median    uq   max neval cld
#       df[3, 2] 17.99 18.96 20.16  19.38 19.77 35.14   100   b
#  df[3, "name"] 17.97 19.13 21.45  19.64 20.15 74.00   100   b
#     df$name[3] 12.48 13.81 15.81  14.48 15.14 67.24   100   a
```

Cross-transferable skills

- **Profiling in R**

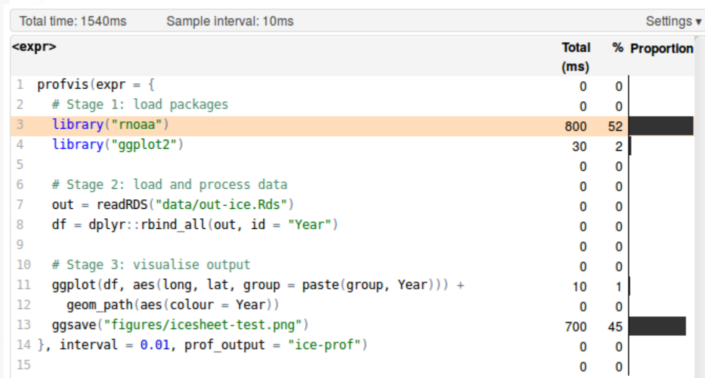
Using the `profvis` library you can check which part of your code take more time to execute:

```r
library("profvis")
profvis(expr = {

  # Stage 1: load packages
  # library("rnoaa") # not necessary as data pre-saved
  library("ggplot2")

  # Stage 2: load and process data
  out = readRDS("extdata/out-ice.Rds")
  df = dplyr::rbind_all(out, id = "Year")

  # Stage 3: visualise output
  ggplot(df, aes(long, lat, group = paste(group, Year))) +
    geom_path(aes(colour = Year))
  ggsave("figures/icesheet-test.png")
}, interval = 0.01, prof_output = "ice-prof")
```



Cross-transferable skills

- **Memory consumption in R**

Using the **`profmem`** library you can check which part of your code take more time to execute:

```
> library("profmem")
> options(profmem.threshold = 2000)
> p <- profmem({
+     x <- integer(1000)
+     Y <- matrix(rnorm(n = 10000), nrow = 100)
+ })
> p
Rprofmem memory profiling of:
{
    x <- integer(1000)
    Y <- matrix(rnorm(n = 10000), nrow = 100)
}
Memory allocations (>= 2000 bytes):
       what  bytes                  calls
1     alloc   4048              integer()
2     alloc  80048 matrix() -> rnorm()
3     alloc   2552 matrix() -> rnorm()
4     alloc  80048               matrix()
total       166696
```

5 tips for an efficient R set-up

1. Use system monitoring to identify bottlenecks in your hardware/code.
2. Keep your R installation and packages up-to-date.
3. Make use of RStudio's powerful autocompletion capabilities and shortcuts.
4. Store API keys in the .Renviron file.
5. Use BLAS if your R number crunching is too slow.

Efficient OS

In R, the `sys.info()` function gives information about the OS where it is running.

The OS resource monitor can be used to check key OS variables when running computationally intensive programs.

System monitoring provides a useful tool for understanding how R is performing in relation to variables reporting the OS state, such as how much RAM is in use.

CPU resource allocated over time is another common OS variable that is worth monitoring to check if there is spare CPU capacity to harness by parallel code.

Efficient R version

R is a mature and stable language so well-written code in base R should work on most versions. However, it is important to keep your R version relatively up-to-date, because:

- Bug fixes are introduced in each version, making errors less likely;
- Performance enhancements are made from one version to the next, meaning your code may run faster in later versions;
- Many R packages only work on recent versions on R.

Efficient R startup

When an R session is launched, R loads several files. Among them, there are 2 important to automatize tasks:

- `.Rprofile`, runs lines of R code every time R starts. If you want R to check for package updates each time or define your own function, you simply add the relevant lines somewhere in this file.
- `.Renviron`, the primary purpose of which is to set environment variables. These tell R where to find external programs and can hold user-specific information than needs to be kept secret, typically API keys.

Efficient R startup

R search for these files in:

- `R_HOME`, the directory in which R is installed. The etc sub-directory can contain start-up files.
- `HOME`, the user's home directory.
- R's current working directory.

R only uses one `.Rprofile` and one `.Renviron` in any session, choosing R's current working directory over HOME, and `HOME` over `R_HOME`. Check where `HOME` and `R_HOME` directories are using the `Sys.get_env()` function. Check here to know more about these files.

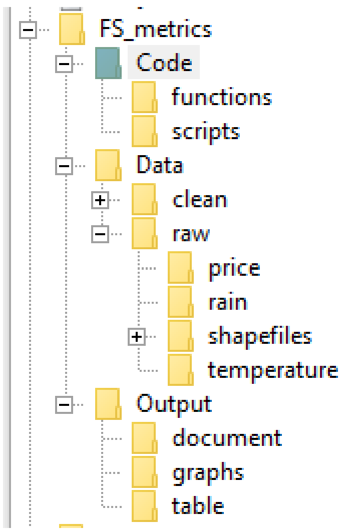
Warning: Customizing these files might produce non-portable code, for example, by creating a custom function in `.Rprofile` and using it in a script.

Efficient RStudio

RStudio has a lot of tools to improve your R development time, such as shortcuts, configuration, etc…

One of the most important is project management:

- The working directory automatically switches to the project's folder. Data and script files may be referred to using relative file paths.
- The last previously open file is loaded into the Source pane. The history of R commands executed in previous sessions is also loaded into the History tab.
- Any settings associated with the project, such as Git settings, are loaded.



Efficient R interpreter

Note that for many applications stability rather than speed is a priority, so these should only be considered if

a) you have exhausted options for writing your R code more efficiently.

b) you are confident tweaking system-level settings.

So be careful with this!

Efficient R interpreter

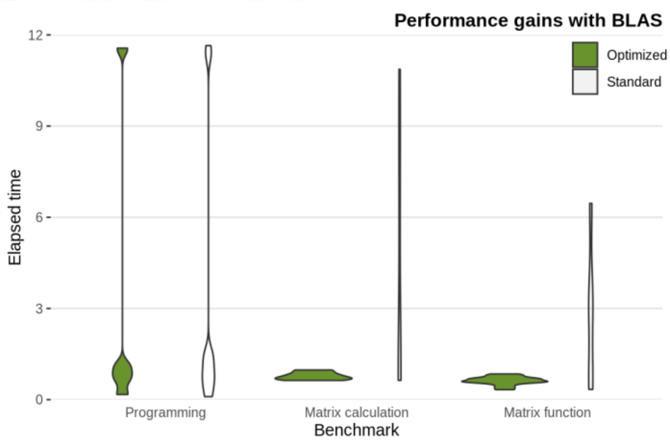
There are some R alternative interpreter which changes how the computer executes R commands:

- Microsoft R open, formerly known as Revolution R Open (RRO). The key enhancement is that it uses multi-threaded mathematics libraries.
- Rho, formerly known as CXXR, a re-implementation of the R interpreter for speed and efficiency.
- Renjin reimplements the R interpreter in Java, so it can run on the Java Virtual Machine (JVM). Since R will be pure Java, it can run anywhere.
- Oracle also offer an R-interpreter that uses Intel's mathematics library

Efficient R interpreter

For linear algebra operations, R uses the Basic Linear Algebra System (BLAS) framework. By switching to a different BLAS library, it may be possible to speed-up your R code.

The two open source alternative BLAS libraries are ATLAS and OpenBLAS. The Intel MKL is another implementation for Intel microprocessors, but it requires licensing fees.



5 tips for efficient programming

1. Be careful never to grow vectors.
2. Vectorise code whenever possible.
3. Use factors when appropriate.
4. Avoid unnecessary computation by caching variables.
5. Byte compile packages for an easy performance boost.

General advice

Low level languages like C and Fortran demand more from the programmer (static typing, memory management, …).

- The advantage of such languages, compared with R, is that they are faster to run.
- The disadvantage is that they take longer to learn and can not be run interactively.

Although code in R tends to be more concise and elegant, the poor code under-the-hood tasks might result in slow R programs. These are covered in Burns (2011).

Ultimately calling an R function always ends up calling some underlying C/Fortran code. A **golden rule** in R programming is to access the underlying C/Fortran routines as quickly as possible; the fewer functions calls required to achieve this, the better.

Note: Everything in R is a function call. When we execute `1 + 1`, we are actually executing `'+'(1, 1)`.

Memory allocation

Another general technique is to be careful with memory allocation. If possible, pre-allocate your object then fill in the values.

Comparing three method to create a sequence of numbers

```
method1 = function(n) {
  vec = NULL # Or vec = c()
  for (i in seq_len(n))
    vec = c(vec, i)
  vec
}
```

```
method2 = function(n) {
  vec = numeric(n)
  for (i in seq_len(n))
    vec[i] = i
  vec
}
```

```
method3 = function(n) seq_len(n)
```

n	Method 1	Method 2	Method 3
10^5	0.21	0.02	0.00
10^6	25.50	0.22	0.00
10^7	3827.00	2.21	0.00

Remember to quickly access C/Fortran

Efficient programming

Vectorized code

Recall the golden rule in R programming, access the underlying C/Fortran routines as quickly as possible; the fewer functions calls required to achieve this, the better.

With this mind, many R functions are vectorized, that is the function's inputs and/or outputs naturally work with vectors, reducing the number of function calls required.

Compare these two functions:

```
log_sum = 0
for (i in 1:length(x))
  log_sum = log_sum + log(x[i])
```

```
log_sum = sum(log(x))
```

Both codes do the same thing, but the vectorized code:

- It's faster. When $n = 10^7$ is about forty time faster.
- It's neater.
- It doesn't contain a bug when x is of length 0

Efficient programming

Factors

A factor is used to store categorical variables and, although they are similar to character vectors, they are actually integers:

- Factors store categorical variables as 1, 2, 3, ..., with associated documentation elsewhere that explains what each number means, i.e., memory consuming.
- Alternatively categorical variables can be stored as characters, losing the sense that the variable is categorical.

It cannot be said that you should **always** or **never** use factors to store categorical variables, since it has advantages and disadvantages.

As a **general rule**, if the variable has an **inherent order** or a **fixed set of categories**, then storing it as a factor should be considered.

Efficient programming
Caching variables

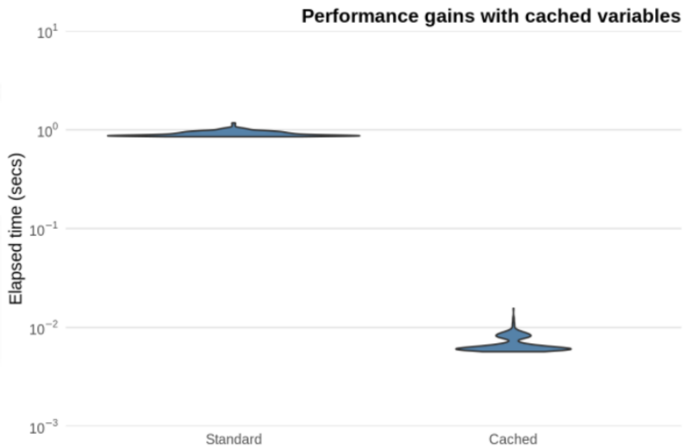
A straightforward method for speeding up code is to calculate objects once and reuse the value when necessary. This could be as simple as replacing `sd(x)` in multiple function calls with the object `sd_x` that is defined once and reused:

```
apply(x, 2, function(i) mean(i) / sd(x))
```

vs

```
sd_x = sd(x)
apply(x, 2, function(i) mean(i) / sd_x)
```

For a 100 row by 1000 columns matrix, the cached version is a 100 times faster:



Efficient programming

Caching variables

An advanced form of caching is to use the `memoise` package. It allows to easily store the value of function call and returns the cached result when the function is called again with the same arguments.

This package trades off memory versus speed, since the memoised function stores all previous inputs and outputs. To cache a function, we simply pass the function to the `memoise` function.

The classic `memoise` example is the factorial function. Another example is to limit use to a web resource where the user can vary some parameters.

Caching variables

An even more advanced form of caching is to use **function closures**. A closure in R is an object that contains functions bound to the environment the closure was created in.

Technically all functions in R have this property, but the term function closure is used to denote functions where the environment is not in `.GlobalEnv`.

One of the environments associated with a function is known as the **enclosing environment**, that is, where the function was created. This allows us to store values between function calls.

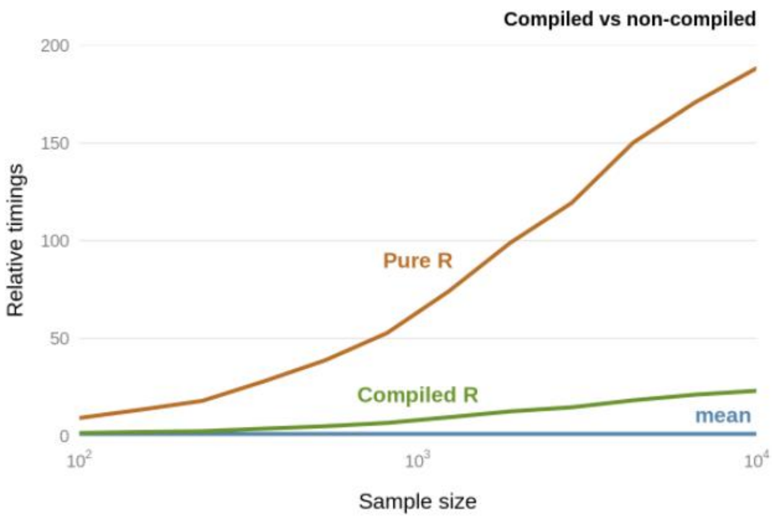
Caching variables

A good example of function closure is a stop-watch:

```r
# <<- assigns values to the parent environment
stop_watch = function() {
  start_time = stop_time = NULL
  start = function() start_time <<- Sys.time()
  stop = function() {
    stop_time <<- Sys.time()
    difftime(stop_time, start_time)
  }
  list(start = start, stop = stop)
}
watch = stop_watch()
watch$start()
watch$stop()
```

Byte compile

The `compiler` package allows R functions to be compiled, resulting in a byte code version that may run faster. The compilation process eliminates a number of costly operations the interpreter has to perform, such as variable lookup. To compile a function, simply pass the function to the `cmpfun` function of `compiler`.



Byte compile

However, sometimes you want to compile a whole package.

- If you have created your own package and want it to be compile on installation just add `Bytecompile: true` to the `DESCRIPTION` file.

- If you want to force compilation when installing a package set the `type` argument of the `install.packages()` function to "`source`" and the `INSTALL_opts` to "`--byte-compile`". For example:

```r
install.packages("ggplot2", type = "source", INSTALL_opts = "--byte-compile")
```

A final option is to use just-in-time (JIT) compilation, where R tries to compile the code just when it is executed. The enableJIT() function

Byte compile

A final option is to use just-in-time (JIT) compilation, where R tries to compile the code just when it is executed.

The `enableJIT()` function accept 4 values: 0 (JIT disabled), 1 (compile larger closures before their first use), 2 (compile also some small closures) and 3 (top level loops are also compiled). Calling `enableJIT()` with a negative argument returns the current JIT level.

JIT can also be enabled by setting the environment variable `R_ENABLE_JIT` to one of these values (3 is recommended).

5 tips for efficient programming

1. **Start without writing code** but with a clear mind and perhaps a **pen and paper**. This will ensure you keep your objectives at the forefront of your mind, without getting lost in the technology.
2. **Make a plan**. The size and nature will depend on the project but timelines, resources and ‘chunking’ the work will make you more effective when you start.
3. **Select the packages** you will use for implementing the plan early. Minutes spent researching and selecting from the available options could save hours in the future.
4. **Document your work at every stage**; work can only be effective if it’s communicated clearly and code can only be efficiently understood if it’s commented.
5. Make your entire workflow **as reproducible as possible**. knitr can help with this in the phase of documentation.

Project types

Appropriate project management depends on the types of project:

- *Data analysis*.
  - Explore datasets to discover something.
  - Emphasis on the speed of manipulating data to generate results.
  - Formality is less important.
  - May only be a part of a larger project.
  - Interaction between data analyst and the rest of the team is vital.
- *Package creation*
  - Create code to be reused across projects, possibly with unknown use cases.
  - Emphasis on user interface and documentation.
  - Code style, review, robustness and testing are important.

Project types

- *Reporting and publishing*
  - Writing a report or journal paper or book.
  - Level of formality varies depending upon the audience.
  - Amount of code that takes to arrive at the conclusion is important.
  - Code testing is also important.
- *Software applications*
  - Could range from a simple Shiny app to R being embedded in the server.
  - Since there is a limited human interaction, the emphasis is on code robustness and dealing with failure.

Sometimes it is best not to be prescriptive and try different working practices to discover which works best, time permitting.

Project types

There are some concrete steps that can be taken in most projects that involve programming (in R). In the long run, they improve productivity and reproducibility.

- Project planning: before any code has been written. *Project management is the art of making project plans happen.*
- Package selection: identify which packages are most suitable to get the work done quickly and effectively.
- Publication: especially relevant if you want your code to be useful for others in the long term.

Project planning and management

Good programmers working on a complex project will rarely just start typing code.

Strategic thinking is especially important during a project's inception; if you make a bad decision early on, it will have cascading negative impacts throughout the project's entire lifespan. This is called *technical debt*: "not quite right code which we postpone making it right". Sketching out your ideas using pen and deciding precisely what you want to do (and what you will do if something goes wrong) can minimize your technical debt.

Importance of project management increases exponentially with the number of people involved. On small projects of one script, project management may be a distracting waste of time. On large projects involving dozens of people require much effort on project management: regular meetings, division of labour and a scalable project management system to track progress, issues and priorities.

Project planning and management

There are multiple project management systems to cater for projects across a range of scales, in rough ascending order of scale and complexity:

- Interactive code sharing site Github.
- Zenhub, project management browser plugin that works natively within Github.
- web-based and easy-to-use kanban tools such as Trello and Jira
- dedicated desktop project management software such as ProjectLibre and GanttProject
- fully featured, enterprise scale open source project management systems such as OpenProject and Redmine.

Project planning and management

Regardless of the software (or lack thereof) used for project management, it involves considering the project's aims in the context of available resources (e.g., computational and programmer resources), project scope, time-scales and suitable software. And these things should be considered together.

To take one example, is it worth the investment of time needed to learn a particular R package which is not essential to completing the project but which will make the code run faster? Does it make more sense to hire another programmer or invest in more computational resources to complete an urgent deadline?

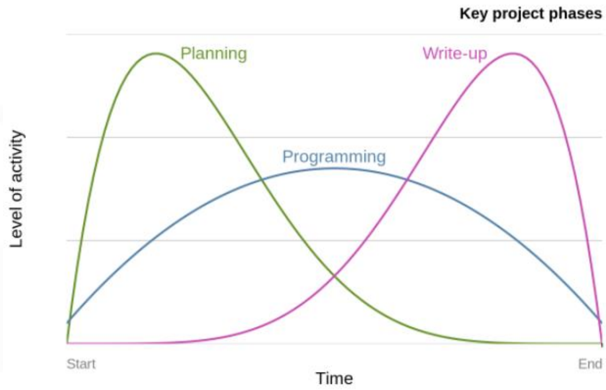
Efficient workflow

Project planning and management

- **Chunking your work**

Or divide your work in self-contained tasks. Once a project overview has devised and stored, a plan with a timeline can be drawn-up.

The up-to-date visualization of this plan can be a powerful reminder to yourself and collaborators of progress on the project so far. More importantly, the timeline provides an overview of what needs to be done next, i.e., **prioritizing tasks**.



Project planning and management

- **Making your workflow SMART**

Instead of dividing the work in tasks, it can be divided into a series of objectives. One way to check if an objective is appropriate for action and review is making it SMART:

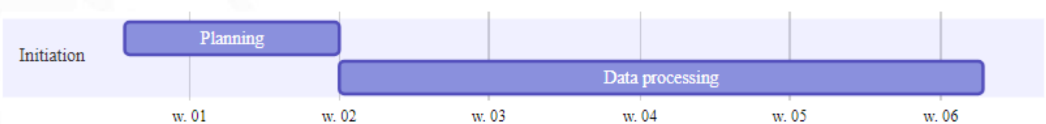
- Specific: is the objective clearly defined and self-contained?
- Measurable: is there a clear indication of its completion
- Attainable: can the target be achieved and assigned (to a person)?
- Realistic: have sufficient resources been allocated to the task?
- Time-bound: is there an associated completion date or milestone?

Project planning and management

- **Visualising plans with R**

DiagrammeR, a new package for creating network graphs and other schematic diagrams in R, provides an R interface to simple flow-chart file formats such as mermaid and GraphViz.

```r
library("DiagrammeR")
# Define the Gantt chart and plot the result
mermaid("gantt
        Section Initiation
        Planning           :a1, 2016-01-01, 10d
        Data processing    :after a1  , 30d")
```



Package selection

A good example of the importance of prior planning to minimize effort and reduce technical debt is package selection. An inefficient, poorly supported or simply outdated package can waste hours.

There are many poor packages on CRAN and much duplication so it’s easy to go wrong. Just because a certain package *can* solve a particular problem, doesn’t mean that it *should*.

Used well, however, packages can greatly improve productivity: not reinventing the wheel is part of the ethos of **open source software**. If someone has already solved a particular technical problem, you don’t have to re-write their code, allowing you to focus on solving the applied problem.

The [r-pkg.org](r-pkg.org) website provides a simple yet effective package online search system if simply Googling is not enough. To get more functionality, various packages dedicated to searching for R packages have been developed like `pkgsearch`.

Package selection

However, there might be a lot of packages that perform a certain task. To know if a package might be a good selection, ask these questions:

- **Can it be interconnected easily with your usual packages?** For example, `tidyverse` consists of several packages interconnected that work better together.
- **Is it mature?** The more time a package is available, the more time it will have for bugs to be solved. The age of a package on CRAN can be seen from its Archive page on CRAN.
- **Is it actively developed?** It is a good sign if packages are frequently updated.
- **Is it well documented?** This is not only an indication of how much thought, care and attention has gone into the package, it also has a direct impact on its ease of use.
- **Is it well used?** This can be seen by searching for the package name online. Most packages that have a strong user base will produce thousands of results.

Publication

The final stage in a typical project workflow is publication. Although it's the final stage to be worked on, that does not mean you should only document *after* the other stages are complete: making documentation integral to your overall workflow will make this stage much easier and more efficient.

Whether the final output is a report containing graphics produced by R, an online platform for exploring results or well-documented code that colleagues can use to improve their workflow, starting it early is a good plan. In every case, the programming principles of reproducibility, modularity and DRY (don't repeat yourself) will make your publications faster to write, easier to maintain and more useful to others.

Publication

- **Dynamic documents with R markdown**

When writing a report using R outputs, a typical workflow has historically been to 1) do the analysis 2) save the resulting graphics and record the main results outside the R project and 3) open a program unrelated to R to import and communicate the results in prose. This is inefficient; it makes updating and maintaining the outputs difficult and there is an overhead involved in jumping between incompatible computing environments.

To avoid this, the R Markdown framework in conjunction with the `knitr` package can:

- Process code chunks (via `knitr`)
- A notebook interface for R (via RStudio)
- The ability to render output to multiple formats (via `pandoc`)

R Markdown documents are plain text and have file extension `.Rmd`

Efficient workflow

Publication

In an R Markdown document, results are generated on the fly by including ‘code chunks’. Code chunks are R code that are preceded by `` ```{r, options} `` on the line before the R code, and `` ``` `` at the end of the chunk. For example:

````
```{r chunkName, eval = TRUE, echo = TRUE}
(1:5)^2
```
````

`chunkName` is the name provided to the code chunk, `eval = TRUE` indicates that this code chunk must be evaluated when compiling the file, and `echo = TRUE` controls whether the R code is displayed.

An important advantage of dynamically documenting work this way is that when the data or analysis code changes, the results will be updated in the document automatically.

Learn more from RMarkdown here.

Introduction

Input/output (I/O) is the technical term for reading and writing data: the process of getting information into a particular computer system (in this case R) and then exporting it to the 'outside world' again (in this case as a file format that other software can read).

Data I/O will be needed on projects where data comes from, or goes to, external sources. However, most of R resources and documentation start with the optimistic assumption that your data has already been loaded. Tricky, slow or ultimately unsuccessful data I/O can cripple efficiency right at the outset of a project.

5 tips for efficient data I/O

1. If possible, keep the names of local files downloaded from the internet or copied onto your computer unchanged. This will help you trace the provenance of the data in the future.
2. R's native file format is `.Rds`. These files can be imported and exported using `readRDS()` and `saveRDS()` for fast and space efficient data storage.
3. Use `import()` from the `rio` package to efficiently import data from a wide range of formats, avoiding the hassle of loading format-specific libraries.
4. Use the `readr` or `data.table` equivalents of `read.table()` to efficiently import large text files.
5. Use `file.size()` and `object.size()` to keep track of the size of files and R objects and take action if they get too big.

The `rio` package

`rio` is 'A Swiss-Army Knife for Data I/O'. It provides easy-to-use and computationally efficient functions for importing and exporting **tabular** data in a range of file formats.

An example of `import()`ing and `export()`ing:

```
library("rio")
# Specify a file
fname = system.file("extdata/voc_voyages.tsv", package = "efficient")
# Import the file (uses the fread function from data.table)
voyages = import(fname)
# Export the file as an Excel spreadsheet
export(voyages, "voc_voyages.xlsx")
```

However, there are more than tabular data.

Binary files

After loading the raw data and tidying it, it is common to save it to reduce the chance of having to run all the data tidying again. This data should be saved in binary format to include non-tabular data and reduce read/write times and file sizes.

In R, there are two native binary files: `.Rds` and `.Rdata`. The difference between them is shown in the next example:

```
save(df_co2, file = "extdata/co2.RData")
saveRDS(df_co2, "extdata/co2.Rds")
load("extdata/co2.RData")
df_co2_rds = readRDS("extdata/co2.Rds")
identical(df_co2, df_co2_rds)
#> [1] TRUE
```

`save` can store all the data in the workspace, but then all the data is loaded back again at once. `saveRDS` only allows one object, but it only loads one object, being more controllable.

Feather files

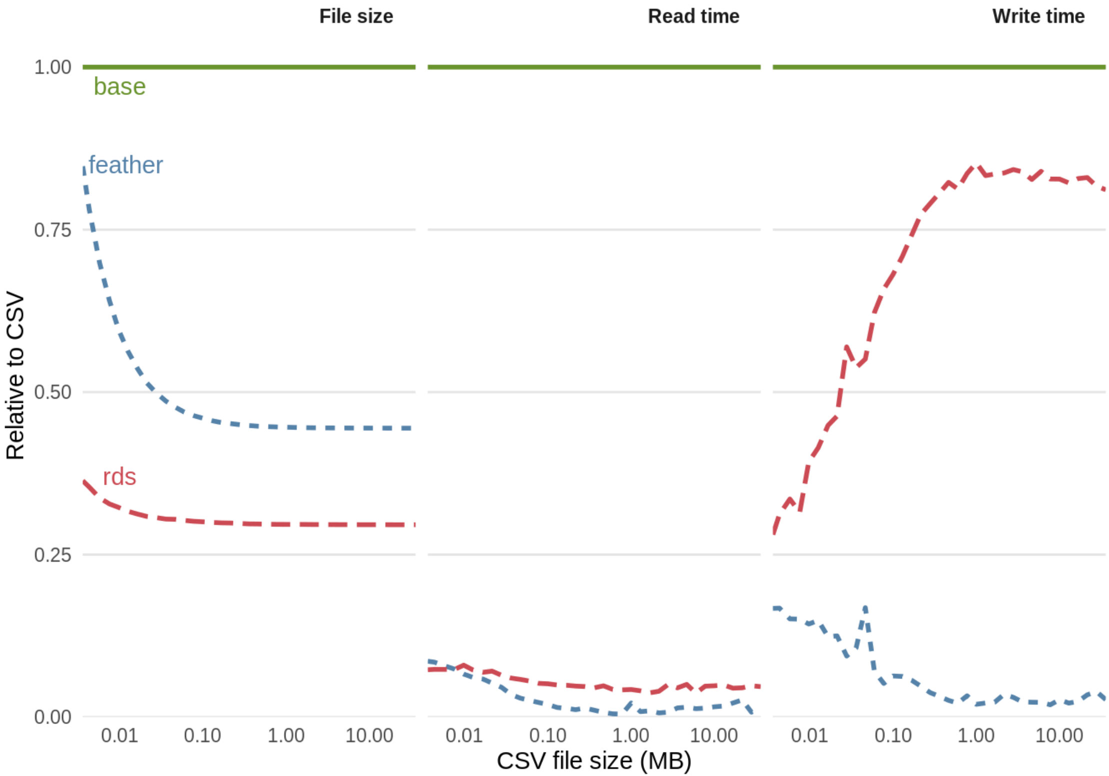
Although the previous files are very useful for files in R, the `.feather` files were conceived as a fast, light and language agnostic format for storing data frames in R and Python.

```r
library("feather")
write_feather(df_co2, "extdata/co2.feather")
df_co2_feather = read_feather("extdata/co2.feather")
```

```python
import feather
path = 'data/co2.feather'
df_co2_feather = feather.read_dataframe(path)
```

So, which is better for data input/output?

Benchmarking I/O methods



5 tips for efficient data carpentry

1. Time spent preparing your data at the beginning can save hours of frustration in the long run.
2. ‘Tidy data’ provides a concept for organising data and the package `tidyr` provides some functions for this work.
3. The `data_frame` class defined by the `tibble` package makes datasets efficient to print and easy to work with.
4. `dplyr` provides fast and intuitive data processing functions; `data.table` has unmatched speed for some data processing applications.
5. The `%>%` ‘pipe’ operator can help clarify complex data processing workflows.

Efficient data carpentry
tibble

- `tibble` is a package that defines the `tbl_df` class.
- Similar to the base class `data.frame`, with a more user friendly printing, subsetting and factor handling.

```
library("tibble")
tibble(x = 1:3, y = c("A", "B", "C"))
#> # A tibble: 3 x 2
#>       x y
#>   <int> <chr>
#> 1     1 A
#> 2     2 B
#> 3     3 C
```

Tidying data

Converting data into a ‘tidy’ form is computational efficient: it is usually faster to run analysis and plotting commands on tidy data. Data tidying includes data cleaning and data reshaping, and is an important part of the **Exploratory Data Analysis (EDA)**.

Tidy data follows the ‘long form’: **few long columns** instead than many short ones. However, wide data is useful for presentation like summary tables → Need to be able to transfer between wide and tidy format.

Tidy data has the following characteristics:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

Complex datasets are usually represented by multiple tables, with unique identifiers or ‘keys’ to join them together.

Regular expressions (regex)

`regex` is a language for describing and manipulating text strings. There are books and several tutorials on regex in R, here are only some basics on the topic.

The foundational regex operation is to detect whether or not a particular text string exists in an element or not which is done with `grepl()` and `str_detect()` in `base R` and `stringr` respectively:

```
library("stringr")
x = c("Hi I'm Robin.", "DoB 1985")

grepl(pattern = "9", x = x)
#> [1] FALSE  TRUE
str_detect(string = x, pattern = "9")
#> [1] FALSE  TRUE
```

Notice that `str_detect()` begins with `str_`. This is a common feature of `stringr` functions: they all do. Check here to see a deep tutorial of these functions.

Data processing

After tidying the data, the next stage is generally data processing. This includes creation of new data and data analysis.

`dplyr` is usually the go-to package for this, due to its advantages:

- Is fast to run (due to its C++ backend) and intuitive to use.
- Works well with tidy data.
- Works well with databases (several tables).

As this package has been explained before, let's just acknowledge this and go forward.

Efficient data carpentry

Working with databases

As databases has been explained in previous sessions (`DBI`, `RSQLite`, `mongolite`, …), here is only explained how to use `dplyr` with databases. To access a database in R via `dplyr` (once you have connected to a database via `DBI::dbConnect()`), the `src_*` functions shall be used to create a source. Imagine you have connected to a database named `ghg_db`, one can create a `tbl` object that can be queried as an standard `data.frame`:

```
library("dplyr")
ghg_db = src_sqlite(ghg_db)
ghg_tbl = tbl(ghg_db, "ghg_ems")
rm_world = ghg_tbl %>%
  filter(Country != "World")
```

In the above code, `dplyr` has actually generated the necessary SQL command, which can be examined using `explain(rm_world)`. When working with databases, `dplyr` uses lazy evaluation: the data is only fetched at the last moment when it's needed. When the SQL command must be executed, use `collect(rm_world)`.

Data processing – an alternative

An alternative to `dplyr` is the `data.table` package. In general, `dplyr` is more recommendable as it provides a more flexible and consistent interface. It also allows to work with the `data.table` class so it gets the best of both worlds.

However, `data.table` is faster than `dplyr` for some operations and offer some additional functionality:

- rolling joins: allow rows in one table to be selected based on proximity between shared variables (typically time)
- non-equi joins: join criteria can be inequalities.

Although `data.table` may be faster than `dplyr`, remember that efficient code writing is often more important than efficient execution.

Data processing – an alternative

Like base R `data.table` uses square brackets but (unlike base R but like `dplyr`) uses non-standard evaluation so you need not refer to the object name inside the brackets:

```
library("data.table")
# data(wb_ineq) # from the efficient package
wb_ineq_dt = data.table(wb_ineq) # convert to data.table class
aus3a = wb_ineq_dt[Country == "Australia"]
```

Note that the square brackets do not need a comma to refer to rows with `data.table` objects: in base R you would write `wb_ineq[wb_ineq$Country == “Australia”,]`.

Efficient data carpentry

Data processing – an alternative

To boost performance one can set 'keys' analogous to primary keys in SQL. These are supercharged rownames which order the table based on one or more variables. This allows a *binary search* algorithm instead of the *vector scan* used by base R.

`data.table` uses the key values for subsetting by default so the variable does not need to be mentioned again. Instead, using keys, the search criteria is provided as a list (invoked below with the concise `.()` syntax, which is synonymous with `list()`)

```
setkey(wb_ineq_dt, Country)

aus3b = wb_ineq_dt[.("Australia")]
```

Even the 'non key' implementation of data.table subset method is faster than the alternatives: this is because data.table creates a key internally by default before subsetting.

For further information on `data.table`, see here, here and here.

Efficient optimization

Introduction

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth

Code optimization is an advanced topic that should only be tackled once 'low hanging fruit' for efficiency gains have been taken. This means that **time spent optimizing code early** in the developmental stage **could be wasted**.

Pre-requisites for this section:

- Linux: A compiler should already be installed. Otherwise, install `r-base` and a compiler will be installed as a dependency.
- Macs: Install `xcode`.
- Windows: Install [`Rtools`]. Make sure you select the version that corresponds to your version of R.

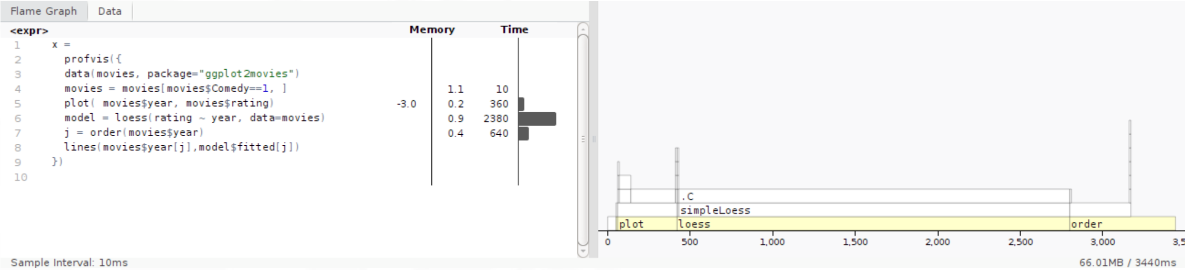
5 tips for efficient performance

1. Before you start to optimise your code, ensure you know where the bottleneck lies; use a code profiler (`profvis()` recommended).
2. If the data in your data frame is all of the same type, consider converting it to a matrix for a speed boost.
3. Use specialised row and column functions whenever possible.
4. The `parallel` package is ideal for Monte-Carlo simulations.
5. For optimal performance, consider re-writing key parts of your code in C++ (let's forget about C and Fortran).

Efficient optimization
profvis

The main function from the `profvis` package is `profvis()`, which profiles the code and creates an interactive HTML page of the results. The first argument of `profvis()` is the R expression of interest, which can be many lines long:

```r
library("profvis")
profvis({
  data(movies, package = "ggplot2movies") # Load data
  movies = movies[movies$Comedy == 1,]
  plot(movies$year, movies$rating)
  model = loess(rating ~ year, data = movies) # loess regression line
  j = order(movies$year)
  lines(movies$year[j], model$fitted[j]) # Add line to the plot
})
```



Efficient optimization

Efficient base R

- `if()` vs `ifelse()` function

`ifelse()` function can be used to perform a conditional operation based on a vectorized condition. `ifelse()` returns a vector of the same length as the condition provided, with values `if_yes` or `if_no`:

```
ifelse(test, if_yes, if_no)
```

An additional quirk of `ifelse()` is that although it is more *programmer efficient*, as it is more concise and understandable than multi-line alternatives, it is often **less** *computationally efficient* than a more verbose alternative (around 10-20 times slower).

A simple solution is to use the `if_else()` function from `dplyr` for most situations (although not for all and is still 3 times slower than the verbose alternative).

Efficient base R

- `sort()` and `order()` functions

Sorting a vector is relatively quick, but if sorting is done inside a loop or in a `shiny` application it can be worthwhile optimizing this operation.

There are currently three sorting algorithms `c("shell", "quick", "radix")` that can be specified in the `sort()` function, being `radix` the most computationally efficient option for most situation (almost 20% faster).

Another useful trick is to partially order the results. For example, if you only want to display the top ten results, then use the `partial` argument, i.e. `sort(x, partial = 1:10)`.

If you wish to sort in decreasing order, take into account that `sort(x, decreasing = TRUE)` is marginally (around 10%) faster than `rev(sort(x))`.

Efficient base R

- Converting factors to numerics

A factor is just a vector of integers with associated levels. Occasionally we want to convert a factor into its numerical equivalent. The most efficient way of doing this (especially for long factors) is:

```
as.numeric(levels(f))[f]
```

- Logical AND and OR

The logical AND (&) and OR (|) operators are vectorised functions and are typically used during multi-criteria subsetting operations. When R executes an expression with the AND (&) symbol, it calculate the first and second comparison and then calculate the logical operation.

In contrast, the non-vectorized version (&&) only executes the second component if the first is TRUE. Be careful not to use && or || on vectors since it only evaluates the first element of the vectors.

Efficient base R

- Row and column operations

In data analysis is common to apply a function to each column or row of a data set. The `apply()` function makes this type of operation straightforward.

For some operations, there are optimized functions like calculating row and columns sums/means (`rowSums()`, `colSums()`, `rowMeans()` and `colMeans()`) that should be used whenever possible. The package `matrixStats` contains additional optimized row/col functions.

- `is.na()` and `anyNA()`

To test whether a vector (or other object) contains missing values we use the `is.na()` function. A common operation is to check if whether a vector contains any missing values. In this case, `anyNA(x)` is more efficient than `any(is.na(x))`.

Efficient optimization

Efficient base R

- Matrices

A matrix is similar to a data frame: it is a two-dimensional object and sub-setting and other functions work in the same way. However, all matrix elements must have the same type. Matrices **are generally faster** than data frames.

Use the `data.matrix()` function to efficiently convert a `data.frame` into a `matrix`

```
data(ex_mat, ex_df, package="efficient")
microbenchmark(times=100, unit="ms", ex_mat[1,], ex_df[1,])
#> Unit: milliseconds
#>          expr    min     lq   mean  median      uq  max neval
#>  ex_mat[1, ] 0.0027 0.0034 0.0503 0.00424 0.00605 4.54   100
#>   ex_df[1, ] 0.4855 0.4974 0.5549 0.50535 0.51790 5.25   100
```

Efficient base R

- Integer data type

Numbers in R are usually stored in double-precision floating-point format, occupying 64 memory bits to store a number and being accurate to around 17 decimal places.

Integers are another numeric data type that primarily exist to be passed to C or Fortran code, and may occasionally be used to optimize sub-setting operations. To create an integer, you just have to add `L` to a number: `6L` (`integer`) instead of `6` (`double`).

This is actually faster to execute (around 0,1 microseconds!). Okay, maybe not exciting. The exciting part is that the integer counterpart of a number is roughly **half the size** of the double version. However, most mathematical operations will convert the integer vector into a standard numerical vector.

Efficient base R

- Sparse matrices

A sparse matrix is a matrix where most of the elements are zero. Conversely, if most elements are non-zero, the matrix is considered dense. The proportion of non-zero elements is called the sparsity.

As an example, suppose we have a large matrix where the diagonal elements are non-zero:

```
library("Matrix")
N = 10000
sp = sparseMatrix(1:N, 1:N, x = 1)
m = diag(1, N, N)
```

```
pryr::object_size(sp)
#> 162 kB
pryr::object_size(m)
#> 800 MB
```

Both objects contain the same information, but the data is stored differently. The matrix object stores each individual element, while the sparse matrix object only stores the location of the non-zero elements.

Parallel computing

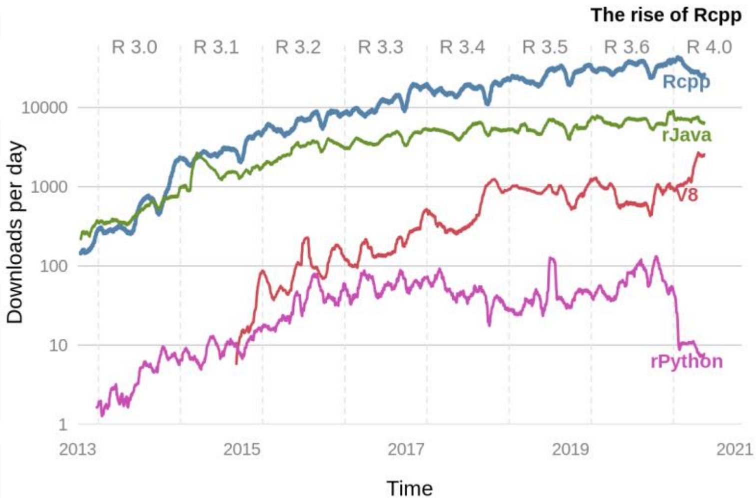
This tool can provide a vast speed-up when using all the resources available of a system (in a clever and organized way) to execute a function.

However, we will see it in a deep way in following lectures, so let's continue!

Efficient optimization

When R is just slow

You’ve tried every trick you know, and your code is still crawling along. At this point you could consider rewriting key parts of your code in another, faster language. `R` has interfaces to other languages via packages, such as `Rcpp`, `rJava`, `rPython` and recently `V8`. These provide `R` interfaces to `C++`, `Java`, `Python` and `JavaScript` respectively.



Efficient optimization

Rcpp

`C++` is a modern, fast and very well-supported language with libraries for performing many kinds of computational tasks. `Rcpp` makes incorporating `C++` code into your `R` workflow easy. Typical bottlenecks that `C++` addresses are loops and recursive functions.

Although `C/Fortran` routines can be used using the `.Call()` function this is not recommended: using `.Call()` can be a painful experience. `Rcpp` provides a friendly API (Application Program Interface) that lets you write high-performance code, bypassing `R`'s tricky `C API`.

To write and compile `C++` functions, you need a working `C++` compiler. To check that you have everything needed for this chapter, run the following piece of code from the course R package:

```
efficient::test_rcpp()
```

Efficient optimization

Rcpp functions

A **C++** function is similar to an **R** function: a set of inputs is passed to the function; some code is run, and a single object is returned. There are some key differences:

1. In the **C++** function each line must be terminated with **;** In **R**, we use **;** only when we have multiple statements on the same line.
2. We must declare object types in the **C++** version. In particular we need to declare the types of the function arguments, return value and any intermediate objects we create.
3. The function must have an explicit return statement. Similar to **R**, there can be multiple returns, but the function will terminate when it hits it's first return statement.
4. You do not use assignment when creating a function.
5. Object assignment must use = sign. The `<-` operator isn't valid.
6. One-line comments can be created using `//`. Multi-line comments are created using `/*...*/`

Efficient optimization
Rcpp functions

Let’s check the differences. Function that add two number:

```r
add_r = function(x, y) x + y
```

```cpp
/* Return type double
 * Two arguments, also doubles
 */
double add_cpp(double x, double y) {
  double value = x + y;
  return value;
}
```

To create the `add_cpp()` function in R:

```r
library("Rcpp")
cppFunction('
  double add_cpp(double x, double y) {
    double value = x + y;
    return value;
  }
')
```

Efficient optimization
Rcpp functions

The `cppFunction()` is great for getting small examples up and running. But it is better practice to put your C++ code in a separate file (with file extension .cpp) and use the function call `sourceCpp("path/to/file.cpp")` to compile them.

A header to include the Rcpp functions must be included at the top of the file:

```
#include <Rcpp.h>
using namespace Rcpp;
```

And above each function that shall be exported/used in R, the following task must be added:

```
// [[Rcpp::export]]
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double add_cpp(double x, double y) {
  double value = x + y;
  return value;
}
```

Efficient optimization

Rcpp types

Using vectors as arguments in C++ is far more difficult than R (Excuse me, do you have a moment to talk about pointers?). However, `Rcpp` smooth the interface by using some data types: `NumericVector`, `CharacterVector`, `LogicalVector`, … . A C++ function for calculating the mean is:

```
double mean_cpp(NumericVector x) {
  int i;
  int n = x.size();
  double mean = 0;

  for(i = 0; i < n; i++) {
    mean = mean + x[i] / n;
  }
  return mean;
}
```

*Remember that a vector in C++ begins at 0 not 1

Rcpp types

Each vector type has a corresponding matrix equivalent: `NumericMatrix`, `CharacterMatrix` and `LogicalMatrix`. The main differences with their vector counterpart are:

- When they are initialized, the number of rows and columns must be specified:

```
// 10 rows, 5 columns
NumericMatrix mat(10, 5);
```

- Subsetting is done using `()`, i.e., `mat(5,4)`
- The first element in a matrix is `mat(0,0)`
- To determine the number of rows and columns exists `.nrow()` and `.ncol()` methods.

Introduction

Your hardware is crucial. It will not only determine how fast you can solve your problem, but also whether you can even tackle the problem of interest. This is because everything is loaded in RAM. Of course, having a more powerful computer costs money. The goal is to help you decide whether the benefits of upgrading your hardware are worth that extra cost.

5 tips for efficient hardware

1. Use the package **`benchmarkme`** to assess your CPUs number crunching ability is it worth upgrading your hardware?
2. If possible, add more RAM.
3. Double check that you have installed a 64-bit version of R.
4. Cloud computing is a cost effective way of obtaining more compute power.
5. A solid state drive (SSD) typically won't have much impact on the speed of your R code, but will increase your overall productivity since I/0 is much faster.

Efficient hardware

Random Access Memory (RAM)

Random access memory (RAM) is a type of computer memory that can be accessed randomly: any byte of memory can be accessed without touching the preceding bytes.

The amount of RAM R has access to is incredibly important. Since R loads objects into RAM, the amount of RAM you have available can limit the size of data set you can analyze. A rough rule of thumb is that your RAM should be three times the size of the dataset you will be working with.

It is straightforward to determine how much RAM you have using the `benchmarkme` package:

```
benchmarkme::get_ram()
```

Random Access Memory (RAM)

It is sometimes possible to increase your computer's RAM. On a computer motherboard there are typically 2 to 4 RAM or memory slots. RAM comes in the form of dual in-line memory modules (DIMMs) that can be slotted into the motherboard spaces.

It is common that all these slots are already taken and you must discard some modules to a higher-capacity ones. Increasing your laptop/desktop is cheap and should be considered. Remember:

```
fortunes::fortune(192)
#>
#> RAM is cheap and thinking hurts.
#>    -- Uwe Ligges (about memory requirements in R)
#>       R-help (June 2007)
```

Hard drives: HDD

Data is typically stored on your hard drive, but not all hard drives are equal. Nowadays Hard Disk Drives (HDDs) are obsolete (IBM introduced them in 1956), where data is stored using magnetism on a rotating platter. The faster the platter spins, the faster the HDD can perform. Their major advantage is that they are cheap.

Hard drives: SSD

Solid State Drives (SSDs) can be thought of a large, sophisticated versions of USB sticks. They have no moving parts and information is stored in microchips. Since there are no moving parts, reading/writing is much quicker. SSDs have other benefits: they are quieter, allow faster boot time (no ‘spin up’ time) and require less power (more battery life).

The read/write speed for a standard HDD is usually in the region of 50 – 120 MB/s (usually closer to 50 MB). For SSDs, speeds are typically over 200 MB/s.

Really, really, really, upgrade your HDD to an SSD. For R it would not make a difference, but booting time of the OS goes from minutes to few seconds.

Operating systems: 32-bit or 64-bit

Ideally, both R and the operating system must be 64-bit, where R can access 8 TB of Windows machines and 128 TB for Unix-based OSs.

Using a 32-bit version of either has severe limitations on the amount of RAM R can access (4 GB at most). So don’t buy more RAM if you are using a 32-bit version (unlikely if you have bought your laptop/computer in the last 10 years).

An easy method for determining if you are running a 64-bit version of R is to run `.Machine$sizeof.pointer == 8`

Central Processing Unit (CPU)

The central processing unit (CPU), or the processor, is the brains of a computer. The CPU is responsible for performing numerical calculations. The faster the processor, the faster R will run.

The clock speed (or clock rate, measured in hertz) is the frequency with which the CPU executes instructions. The faster the clock speed, the more instructions a CPU can execute in a section.

Unfortunately, we can't simply use clock speeds to compare CPUs, since the internal architecture of a CPU plays a crucial role in determining the CPU performance. The R package `benchmarkme` provides functions for benchmarking your system and contains data from previous benchmarks.

```r
res = benchmark_std()

plot(res)

# Upload your benchmarks for future users

upload_results(res)
```

Cloud computing

Cloud computing uses networks of remote servers, instead of a local computer, to store and analyse data. It is now becoming increasingly popular to rent cloud computing resources.

Nowadays, Amazon Web Services, Microsoft Azure, Google Cloud, Alibaba Cloud, IBM Cloud, … might be a cheaper solution to develop a program than improving your local system.

Introduction

Large projects inevitably involve many people. This poses risks but also opportunities for improving computational efficiency and productivity, especially if project collaborators are reading and committing code.

Collaborative working has a number of benefits. A team with a **diverse skill set** is usually stronger than a team with a very narrow focus. It makes sense to **specialize**: clearly defining roles such as statistician, front-end developer, system administrator and project manager will make your team stronger. Even if you are working alone, dividing the work into discrete branches in this way can be useful.

When working on complex programming projects with multiple inter-dependencies **version control** is essential. Even on small projects tracking the progress of your project's code-base has many advantages and makes collaboration much easier.

5 tips for efficient collaboration

1. Have a consistent coding style.
2. Think carefully about your comments and keep them up to date.
3. Use version control whenever possible.
4. Use informative commit messages.
5. Don’t be afraid to elicit feedback from colleagues.

Efficient collaboration

Coding style

There is no single ‘correct’ style, but using multiple styles in the same project is wrong. There are, however, general principles that most programmers agree on:

- Use modular code – One function for one task.
- Comment your code.
- Don’t Repeat Yourself (Idiot) (DRY(i))
- Be concise, clear and consistent.

Some R style guides can be found here, here and here.

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//

```

Coding style

- Reformatting code with RStudio

RStudio can automatically clean up poorly indented and formatted code. To do this, select the lines that need to be formatted (e.g. via `Ctrl+A` to select the entire script) then automatically indent it with `Ctrl+I`. The shortcut `Ctrl+Shift+A` will reformat the code, adding spaces for maximum readability.

- File names

File names should use the .R extension and should be lower case (don't be like your teacher). Avoid spaces and use a dash or underscore to separate words. Avoid non-English alphabetic characters since they cannot be guaranteed to work across locales.

Coding style

- Loading packages

Library function calls should be at the top of your script. When loading an essential package, use library instead of require since a missing package will then raise an error. If a package isn’t essential, use require and appropriately capture the warning raised.

```
# Good
library("dplyr")
# Non-standard evaluation
library(dplyr)
```

Avoid listing every package you may need, instead just include the packages you actually use. If you find that you are loading many packages, consider putting all packages in a file called `packages.R` and using `source` appropriately.

Coding style

- Commenting

Comments can greatly improve the efficiency of collaborative projects by helping everyone to understand what each line of code is doing.

However, too many comments can be inefficient. Ensure that your comments are meaningful, i.e., avoid using verbose English to explain standard R code. Instead, comments should provide context.

BAD:

```
# Setting x equal to 1
x = 1
```

GOOD:

```
# Initialize counter
x = 1
```

Coding style

- Object names

It is important for objects and functions to be named consistently and sensibly. If an object is only used once, its name is less important, a case where `x` could be acceptable. Avoid using a dot (`.`) in the name. Names should be concise yet meaningful.

In functions the required arguments should always be first, followed by optional arguments. The special `...` argument should be last. If your argument has a boolean value, use `TRUE/FALSE` instead of `T/F` for clarity.

While it’s possible to write arguments that depend on other arguments, try to avoid using this idiom as it makes understanding the default behaviour harder to understand. Typically it’s easier to set an argument to have a default value of `NULL` and check its value using `is.null` than by using `missing`. Where possible, avoid using names of existing functions.

Coding style

- Assignment

The two most common ways of assigning objects to values in R is with <- and =. Regardless of which operator you prefer, consistency is key. The difference comes when defining a variable inside a function call:

```r
system.time(expr1 <- rnorm(10e5))

system.time(expr2 = rnorm(10e5)) # error
```

When = is used in a function call, it changes from an assignment operator to an argument passing operator.

Coding style

- Spacing

Consistent spacing is an easy way of making your code more readable. You should add a space around the operators +, -, \ and *. Include a space around the assignment operators, <- and =. Additionally, add a space around any comparison operators such as == and <. The latter rule helps avoid bugs.

```
# Bug. x now equals 1
x[x<-1]
# Correct. Selecting values less than -1
x[x < -1]
```

The exceptions to the space rule are `:`, `::` and `:::`, as well as $ and @ symbols for selecting sub-parts of objects.

Indentation shall also be converted to spaces to avoid different space length between different editors.

comillas.edu

Efficient collaboration

Coding style

- Curly braces

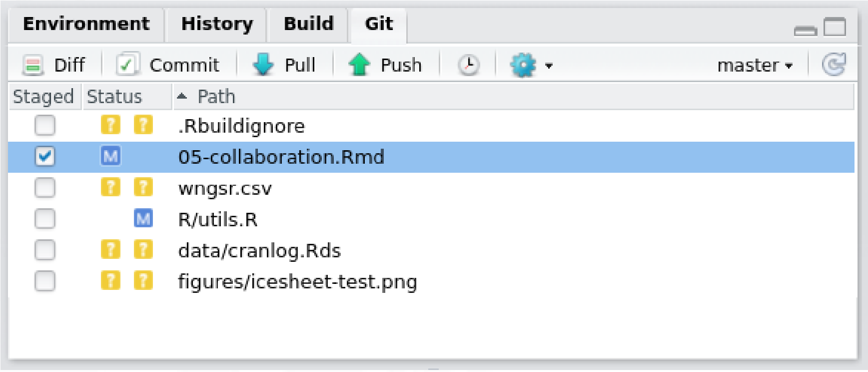
```r
# Bad style, fails
if(x < 5)
{
y}
else {
  x}
```

```r
# Good style
if(x < 5){
  x
} else {
  y
}
```

Version control

When a project gets large, complicated or mission-critical it is important to keep track of how it evolves. In the same way that OneDrive saves a 'backup' of your files, version control systems keep a backup of your code. The only difference is that version control systems back-up your code ***forever***.

The most common version control system is Git, a command-line application created by Linus Torvalds. Rstudio have a Git tab to integrate R in the project.

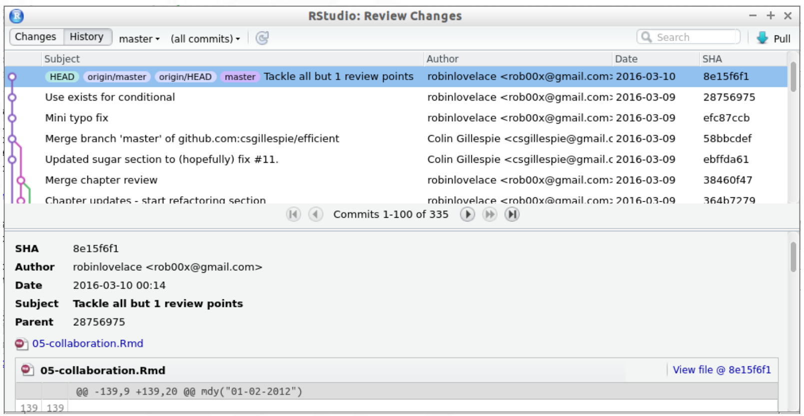


Efficient collaboration
Version control

- Commits

Commits are the basic units of version control. Keep your commits 'atomic': each one should only do one thing. Document your work with clear and concise commit messages, use the present tense, e.g.: 'Add analysis functions'. Remember to push your commits to the remote server when you are working in a team.

Rstudio provides a useful GUI for navigating past commits:



Version control

- GitHub

GitHub is an online platform that makes sharing your work and collaborative code easy. There are alternatives such as GitLab. The focus here is on GitHub as it's by far the most popular among R developers.

Also, through the command `devtools::install_github()`, preview versions of a package can be installed and updated in an instant. This makes 'GitHub packages' a great way to access the latest functionality.

- Learn more

For a more detailed description of Git's powerful functionality, check Jenny Byran's book, "Happy Git and GitHub for the user".

Code review

Simply when we have finished working on a piece of code, a colleague reviews our work and considers questions such as:

- Is the code correct and properly documented?
- Could the code be improved?
- Does the code conform to existing style guidelines?
- Are there any automated tests? If so, are they sufficient?

Regardless of the review method being employed, there a number of points to remember. First, as with all forms of feedback, be constructive. Rather than pointing out flaws, give suggested improvements. Second, if you are reviewing a piece of code set a time frame or the number of lines of code you will review. Third, a code review should be performed before the code is merged into a larger code base; fix mistakes as soon as possible.

Bibliography

- Gillespie, C. and Lovelace, R. (2016). *Efficient R Programming*. O'Reilly Media, Inc.
- Gillespie, C. and Lovelace, R. (2021). *Welcome to Efficient R Programming*. URL: https://csgillespie.github.io/efficientR/index.html
- Eddelbuettel, D. and François, R (2017). *Rcpp syntactic sugar*. URL: https://dirk.eddelbuettel.com/code/rcpp/Rcpp-sugar.pdf

Alberto Aguilera 23, E-28015 Madrid - Tel: +34 91 542 2800 - http://www.iit.comillas.edu