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:

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

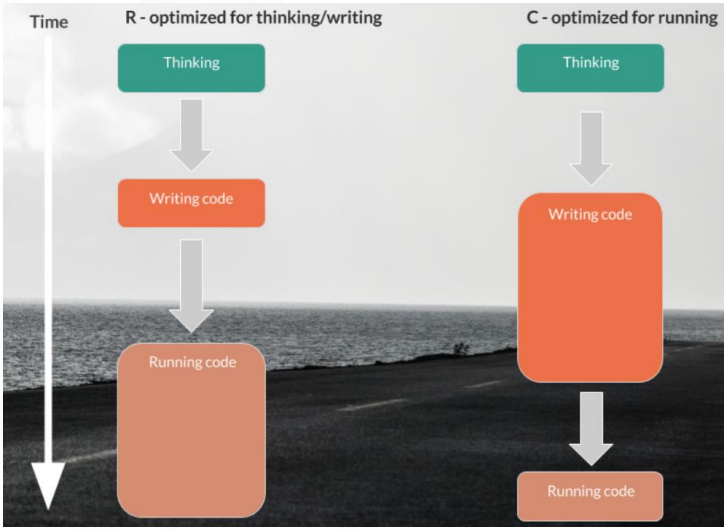
- The narrower definition of efficient programming is **algorithmic efficiency**: how quickly the computer can undertake a piece of work given a particular piece of code.
- The broader definition for efficient programming is **programmer productivity**: amount of ***useful*** work a person can do per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

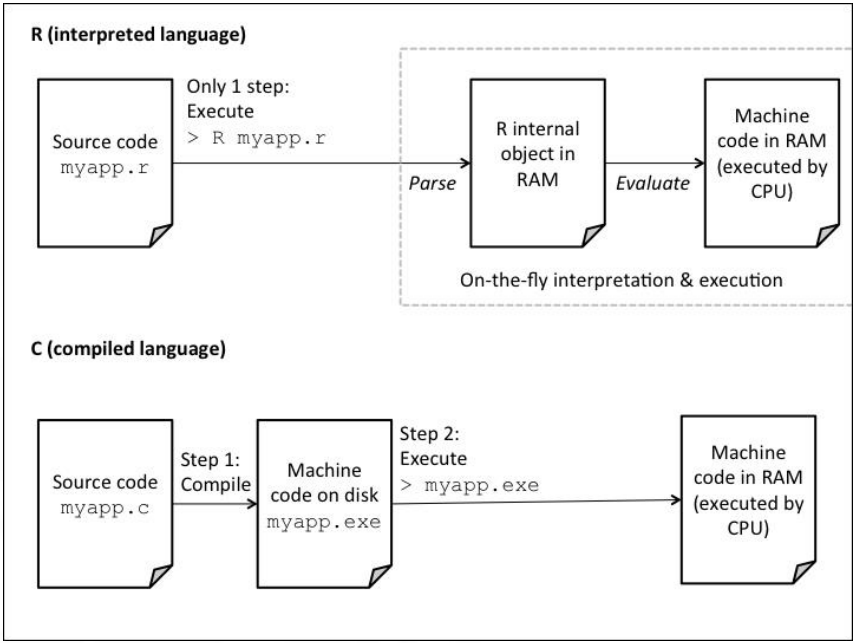
Unfair comparison

- 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**.



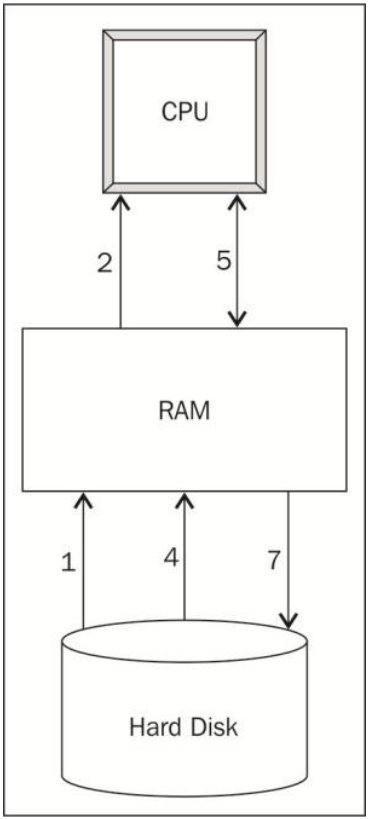
Unfair comparison

- Both types of languages run in a different manner.



3 constrains on code 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.

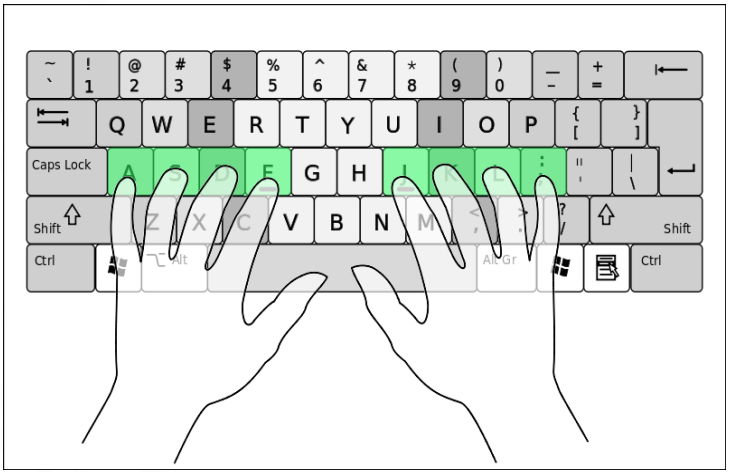
However, this presentation is about programming and these topics are out of the scope.

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")
```

Total time: 1540ms Sample interval: 10ms Settings

<expr>	Total (ms)	%	Proportion
1 profvis(expr = {	0	0	
2 # Stage 1: load packages	0	0	
3 library("rnoaa")	800	52	
4 library("ggplot2")	30	2	
5	0	0	
6 # Stage 2: load and process data	0	0	
7 out = readRDS("data/out-ice.Rds")	0	0	
8 df = dplyr::rbind_all(out, id = "Year")	0	0	
9	0	0	
10 # Stage 3: visualise output	0	0	
11 ggplot(df, aes(long, lat, group = paste(group, Year))) +	10	1	
12 geom_path(aes(colour = Year))	0	0	
13 ggsave("figures/icesheet-test.png")	700	45	
14 }, interval = 0.01, prof_output = "ice-prof")	0	0	
15	0	0	

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

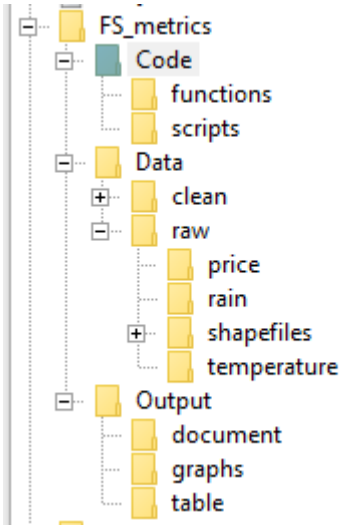
1. Use system monitoring to identify bottlenecks in your hardware/code.
2. Keep your R installation and packages up-to-date.
3. Make use of RStudio's powerful autocompletion capabilities and shortcuts.
4. Store API keys in the .Renviron file.
5. Use BLAS if your R number crunching is too slow.

Efficient 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!

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

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

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

```r
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

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 40 time faster.
- It's neater.
- It doesn't contain a bug when x is of length 0

Caching variables

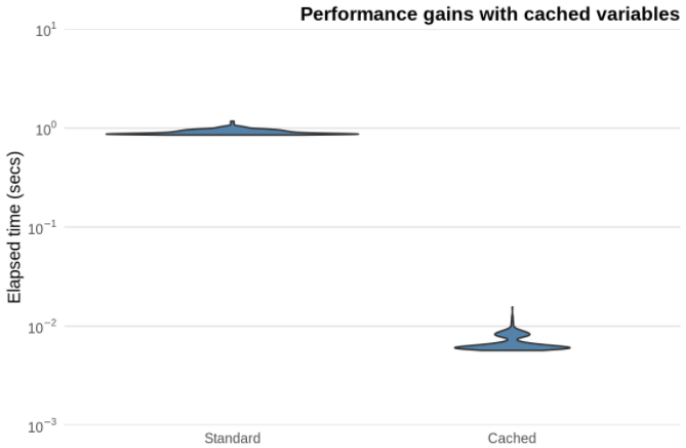
A straightforward method for speeding up code is to calculate objects once and reuse the value when necessary. This could be as simple as replacing `sd(x)` in multiple function calls with the object `sd_x` that is defined once and reused:

```
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:



Caching variables

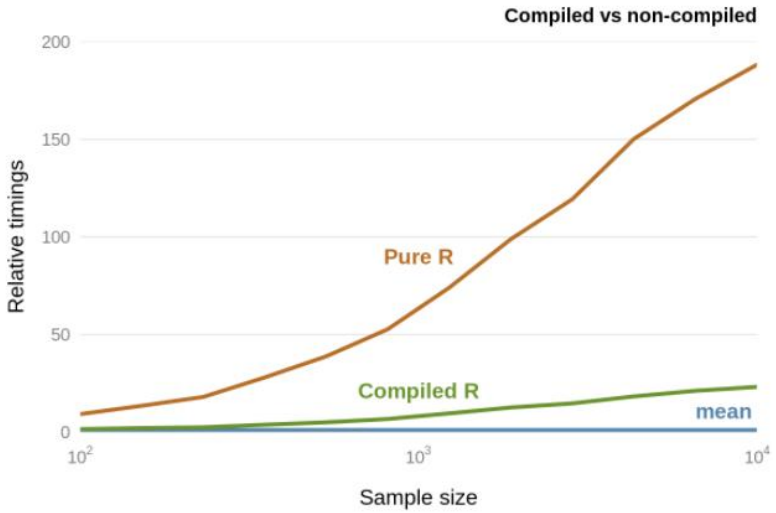
An advanced form of caching is to use the `memoise` package. It allows to easily store the value of function call and returns the cached result when the function is called again with the same arguments.

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.

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:

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

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".

Importance of project management increases exponentially with the number of people involved.

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:

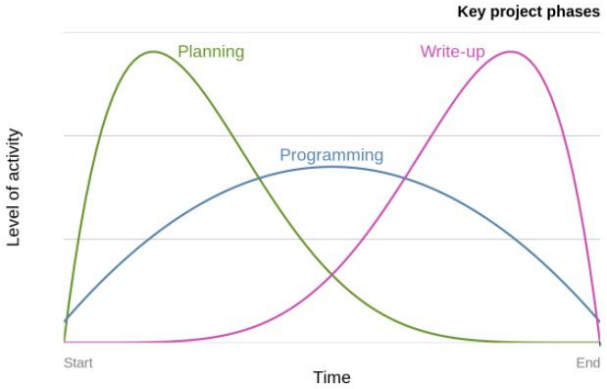
- Interactive code sharing site Github.
- Zenhub, project management browser plugin that works natively within Github.
- web-based and easy-to-use kanban tools such as Trello and Jira
- dedicated desktop project management software such as ProjectLibre and GanttProject
- fully featured, enterprise scale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

Project planning and management

- 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?

Package selection

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?
- Is it mature?
- Is it actively developed?
- Is it well documented?
- Is it well used?

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`

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.

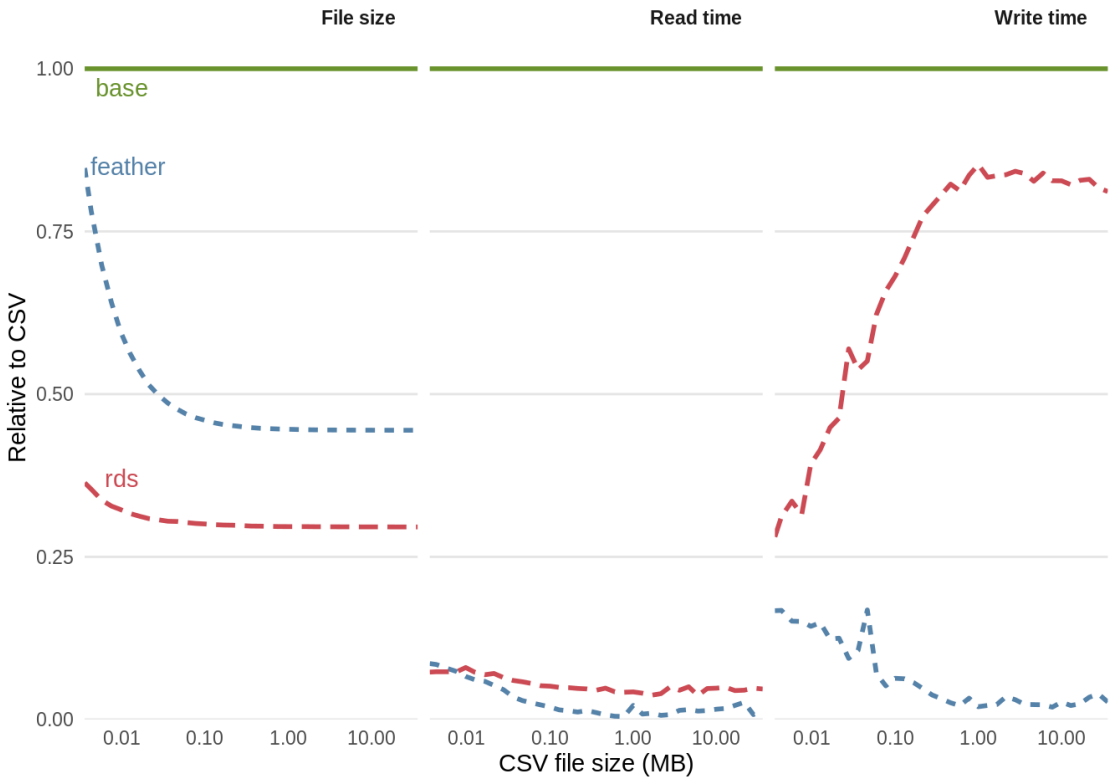
```r
library("feather")
write_feather(df_co2, "extdata/co2.feather")
df_co2_feather = read_feather("extdata/co2.feather")
```

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

So, which is better for data input/output?

Efficient workflow

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.

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.

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).

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.

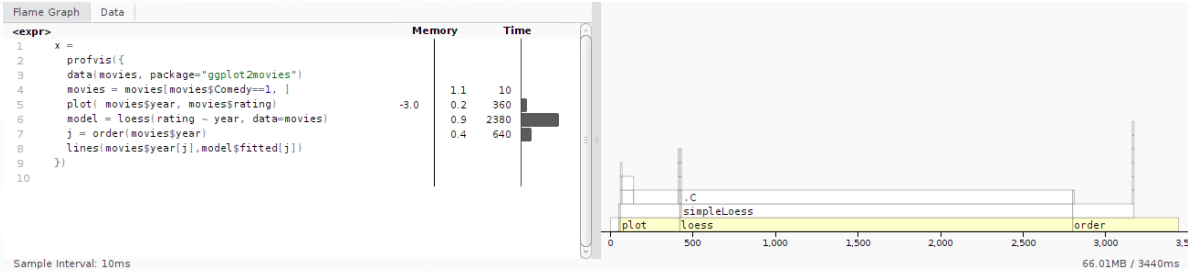
5 tips for efficient performance

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 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 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

- 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.

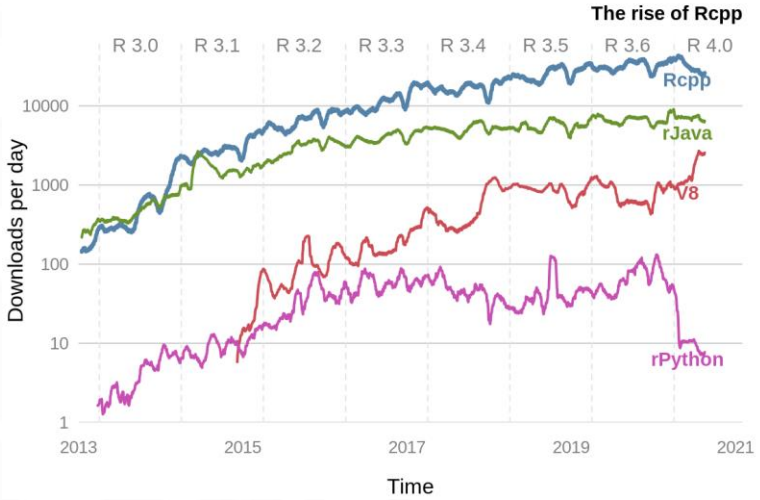
Parallel computing

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!

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.



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()
```

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;
  }
')
```

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;
}
```

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.

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.

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 analyze 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.

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.

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.

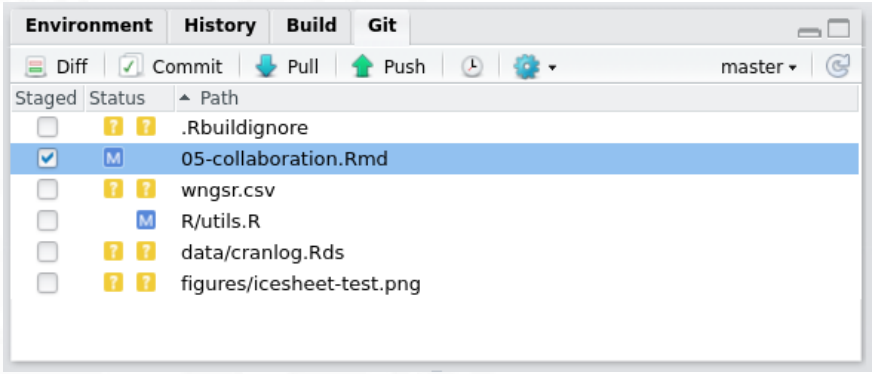
Some R style guides can be found here, here and here.

```
// 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
//
```

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.

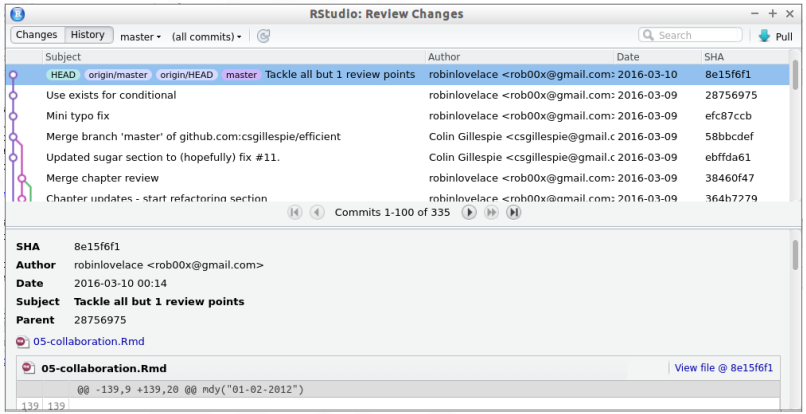


Version control

- 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?

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