Efficient R programming

Colin Gillespie
Robin Lovelace

2016-07-13

Contents

Welcome to Efficient R Programming **7**

Authors 8

Preface **9**

1 Introduction **11**

1.1 Who this book is for 11

1.2 What is efficiency? 11

1.3 Why efficiency? 12

1.4 What is efficient R programming? 12

1.5 Touch typing 12

1.6 Benchmarking 13

1.7 Profiling 14

2 Efficient set-up **17**

2.1 Top 5 tips for an efficient R set-up 17

2.2 Operating system 17

2.3 R version 20

2.4 R startup 22

2.5 RStudio 30

2.6 BLAS and alternative R interpreters 37

3 Efficient programming **41**

3.1 Top 5 tips for efficient programming 41

3.2 General advice 41

3.3 Communicating with the user 46

3.4 Factors 48

3.5 S3 objects 50

3.6 The apply family 52

3.7 Caching variables 55

3.8 The byte compiler 59

4 Efficient workflow **63**
4.1 Top 5 tips for efficient workflow . . . 63
4.2 A project planning typology . . . 63
4.3 Project planning and management . . . 64
4.4 Package selection . . . 68
4.5 Publication . . . 70

5 Efficient input/output **75**
5.1 Top 5 tips for efficient data I/O . . . 75
5.2 Versatile data import with rio . . . 76
5.3 Plain text formats . . . 77
5.4 Binary file formats . . . 81
5.5 The feather file format . . . 82
5.6 Getting data from the internet . . . 83
5.7 Accessing data stored in packages . . . 84

6 Efficient data carpentry **87**
6.1 Top 5 tips for efficient data carpentry . . . 87
6.2 Efficient data frames with tibble . . . 88
6.3 Tidying data with tidyr . . . 89
6.4 Efficient data processing with dplyr . . . 91
6.5 Combining datasets . . . 99
6.6 Working with databases . . . 101
6.7 Data processing with data.table . . . 104

7 Efficient performance **107**
7.1 Top 5 tips for efficient performance . . . 107
7.2 Efficient base R . . . 107
7.3 Code profiling . . . 112
7.4 Parallel computing . . . 115
7.5 Rcpp . . . 117

8 Efficient hardware **125**
8.1 Top 5 tips for efficient hardware . . . 125
8.2 Background: what is a byte? . . . 125
8.3 Random access memory: RAM . . . 126
8.4 Hard drives: HDD vs SSD . . . 129
8.5 Operating systems: 32-bit or 64-bit . . . 130
8.6 Central processing unit (CPU) . . . 130
8.7 Cloud computing . . . 131

9 Efficient collaboration **133**

9.1 Top 5 tips for efficient collaboration 133

9.2 Coding style 133

9.3 Version control 137

9.4 Refactoring code 140

10 Efficient learning **143**

10.1 Top 5 tips for efficient learning 143

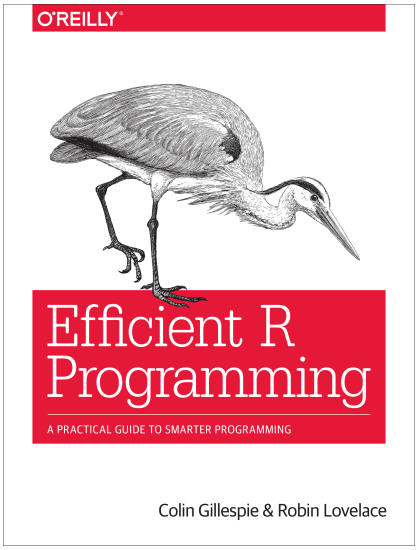
10.2 Using R help 143

10.3 Online resources 145

A Package Dependencies **147**

References **149**

Welcome to Efficient R Programming



This is the online version of the O'Reilly book: Efficient R programming. Pull requests and general comments are welcome.

To build the book:

1. Install the latest version of R
    - If you are using RStudio, make sure that's up-to-date as well
2. Install the book dependencies.

```
# Make sure you are using the latest version of `devtools`
# Older versions do not work.
devtools::install_github("csgillespie/efficientR")
```

3. Clone the efficientR repo
4. If you are using `RStudio`, open `index.Rmd` and click `Knit`.
    - Alternatively, use the bundled `Makefile`

Authors

Colin Gillespie is Senior lecturer (Associate professor) at Newcastle University, UK. His research interests are high performance statistical computing and Bayesian statistics. He has been running R courses for over six years at a variety of levels, ranging from beginners to advanced programming. Colin is regularly employed as an R consultant.

Robin Lovelace is a Research Fellow in the Leeds Institute for Data Analytics, which specialises in the handling of large datasets. Robin has 5 years using R for academic research and 3 years teaching R at all levels. Robin developed the popular tutorial Introduction to Visualising Spatial Data in R and authored Spatial Microsimulation with R Lovelace and Dumont (2016). Robin has used R on mission-critical contracts, including the creation of a nationally scalable interactive online mapping tool for the UK's Department for Transport (DfT) (see www.pct.bike).

Preface

Efficient R Programming is about increasing the amount of work you can do with R in a given amount of time. It's about both *computational* and *programmer* efficiency. There are many excellent R resources about topic areas such as visualisation (e.g. Chang 2012), data science (e.g. Grolemund and Wickham, n.d.) and package development (e.g. Wickham 2015). There are even more resources on how to use R in particular domains, including Bayesian Statistics, Machine Learning and Geographic Information Systems. However, there are very few unified resources on how to simply make R work effectively. Hints, tips and decades of community knowledge on the subject are scattered across hundreds of internet pages, email threads and discussion forums, making it challenging for R users to understand how to write efficient code.

In our teaching we have found that this issue applies to beginners and experienced users alike. Whether it's a question of understanding how to use R's vector objects to avoid for loops, knowing how to set-up your `.Rprofile` and `.Renviron` files or the ability to harness R's excellent C++ interface to do the 'heavy lifting', the concept of efficiency is key. The book aims to distill tips, warnings and 'tricks of the trade' down into a single, cohesive whole that will provide a useful resource to R programmers of all stripes for years to come.

The content of the book reflects the questions that our students, from a range of disciplines, skill levels and industries, have asked over the years to make their R work faster. How to set-up my system optimally for R programming work? How can one apply general principles from Computer Science (such as do not repeat yourself, DRY) to the specifics of an R script? How can R code be incorporated into an efficient workflow, including project inception, collaboration and write-up? And how can one learn quickly how to use new packages and functions?

The book answers each of these questions, and more, in 10 self-contained chapters. Each chapter starts simple and gets progressively more advanced, so there is something for everyone in each. While the more advanced topics such as parallel programming and C++ may not be immediately relevant to R beginners, the book helps to navigate R's famously steep learning curve with a commitment to starting slow and building on strong foundations. Thus even experienced R users are likely to find previously hidden gems of advice in the early parts of the chapters. "Why did no one tell me that before?" is a common exclamation we have heard while teaching this material.

Efficient programming should not be seen as an optional extra and the importance of efficiency grows with the size of projects and datasets. In fact, this book was devised while we were teaching a course on 'R for Big Data': it quickly became apparent that if you want to work with large datasets, your code must work efficiently. Even if you work with small datasets, efficient code, that is both fast to write *and* run is a vital component of successful R projects. We found that the concept of efficient programming is important to all branches of the R community. Whether you are a sporadic user of R (e.g. for its unbeatable range of statistical packages), looking to develop a package, or working on a large collaborative project in which efficiency is mission-critical, code efficiency will have a major impact on your productivity.

Ultimately efficiency is about getting more output for less work input. To take the analogy of a car, would you rather drive 1000 km on a single tank (or a single charge of your batteries) or refuel a heavy, clunky and ugly car every 50 km? In the same way, efficient R code is better than inefficient R code in almost every way: it is easier to read, write, run, share and maintain. This book cannot provide all the answers about how to produce such code but it certainly can provide ideas, example code and tips to make a start in the right direction of travel.

Chapter 1

Introduction

1.1 Who this book is for

This book is for anyone who wants to make their R code faster to type, faster to run and more scalable. These considerations generally come *after* learning the very basics of R for data analysis: we assume you are either accustomed to R or proficient at programming in other languages, although this book could still be of use for beginners. Thus the book should be of use to three groups, albeit in different ways:

- For **programmers with little R knowledge** this book will help you navigate the quirks of R to make it work efficiently: it is easy to write slow R code if you treat as if it were another language.
- For **R users who have little experience of programming** this book will show you many concepts and 'tricks of the trade', some of which are borrowed from Computer Science, that will make your work more time effective.
- An R beginner, you should probably read this book in parallel with other R resources such as the numerous, vignettes, tutorials and online articles that the R community has produced. At a bare minimum you should be familiar with data frames, looping and simple plots.

1.2 What is efficiency?

In everyday life efficiency roughly means 'working well'. An efficient vehicle goes far without guzzling gas. An efficient worker gets the job done fast without stress. And an efficient light shines bright with a minimum of energy consumption. In this final sense, efficiency (η) has a formal definition as the ratio of work done (W e.g. light output) over effort (Q energy consumption):

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

In the context of computer programming efficiency can be defined narrowly or broadly. The narrow sense, *algorithmic efficiency* refers to the way a particular task is undertaken. This concept dates back to the very origins of computing, as illustrated by the following quote by Lovelace (1842) in her notes on the work of Charles Babbage, one of the pioneers of early computing:

> In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

The issue of having a 'great variety' of ways to solve a problem has not gone away with the invention of advanced computer languages: R is notorious for allowing users to solve problems in many ways, and this notoriety has only grown with the proliferation of community contributed package. In this book we want to focus on the *best* way of solving problems, from an efficiency perspective.

The second, broader definition of efficient computing is productivity. This is the amount of *useful* work a *person* (not a computer) can do per unit time. It may be possible to rewrite your code base in C to make it 100 times faster. But if this takes 100 human hours it may not be worth it. Computers can chug away day and night. People cannot. Human productivity is the subject of Chapter 4.

By the end of this book you should know how to write R code that is efficient from both *algorithmic* and *productivity* perspectives. Efficient code is also concise, elegant and easy to maintain, vital when working on large projects.

1.3 Why efficiency?

Computers are always getting more powerful. Does this not reduce the need for efficient computing? The answer is simple: in an age of Big Data and stagnating computer clock speeds (see Chapter 8), computational bottlenecks are more likely than ever before to hamper your work. An efficient programmer can "solve more complex tasks, ask more ambitious questions, and include more sophisticated analyses in their research" (Visser et al. 2015).

A concrete example illustrates the importance of efficiency in mission critical situations. Robin was working on a tight contract for the UK's Department for Transport, to build the Propensity to Cycle Tool, an online application which had to be ready for national deployment in less than 4 months. To help his workflow he developed a function, `line2route()` in the **stplanr** to batch process calls to the (cyclestreets.net) API. But after a few thousand routes the code slowed to a standstill. Yet hundreds of thousands were needed. This endangered the contract. After eliminating internet connection issues, it was found that the slowdown was due to a bug in `line2route()`: it suffered from the 'vector growing problem', discussed in Section 3.2.1.

The solution was simple. A single commit made `line2route()` more than *ten times faster* and substantially shorter. This potentially saved the project from failure. The moral of this story is that efficient programming is not merely a desirable skill: it can be *essential*.

1.4 What is efficient R programming?

Efficient R programming is the implementation of efficient programming practices in R. All languages are different, so efficient R code does not look like efficient code in another language. Many packages have been optimised for performance so, for some operations, achieving maximum computational efficiency may simply be a case of selecting the appropriate package and using it correctly. There are many ways to get the same result in R, and some are very slow. Therefore *not* writing slow code should be prioritized over writing fast code.

Returning to the analogy of the two cars sketched in the preface, efficient R programming for some use cases can simply mean trading in your heavy and gas guzzling hummer for a normal hatchback. The search for optimal performance often has diminishing returns so it is important to find bottlenecks in your code to prioritise work for maximum increases in computational efficiency.

1.5 Touch typing

The other side of the efficiency coin is programmer efficiency. There are many things that will help increase the productivity of yourself and your collaborators, not least following the advice of Janert (2010) to 'think

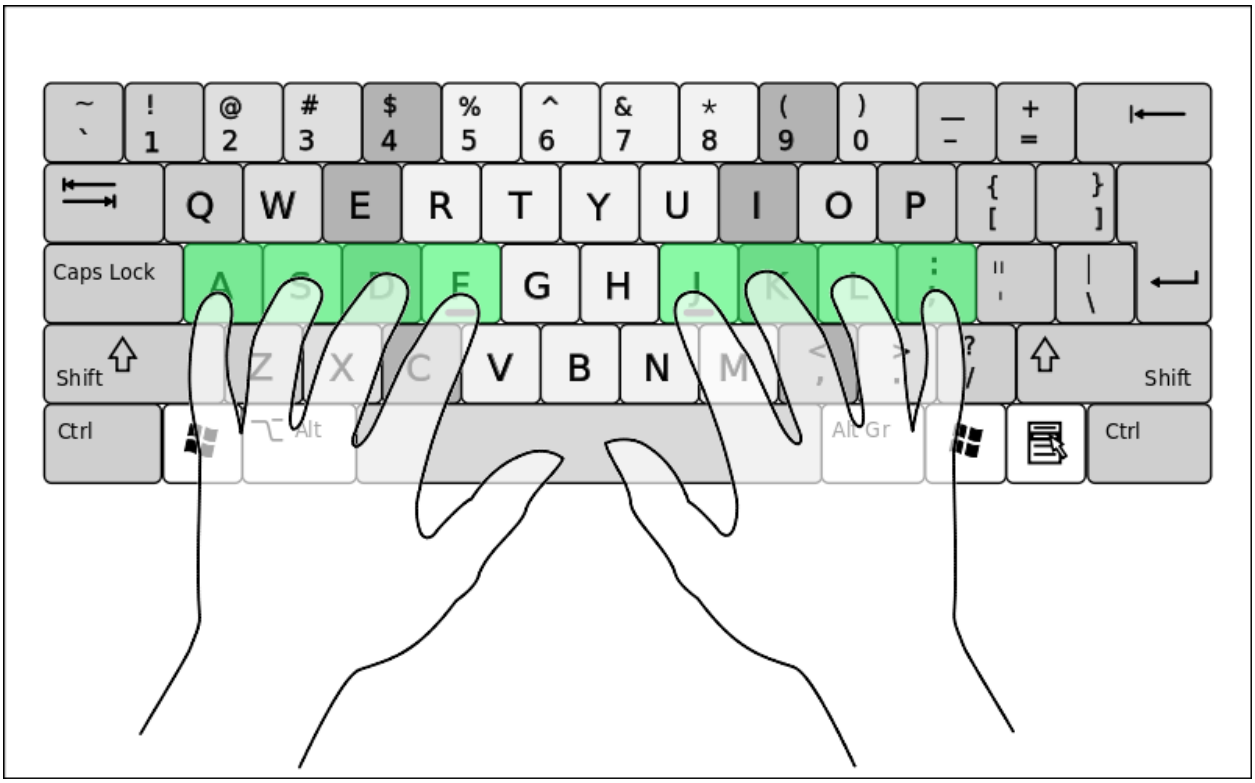


Figure 1.1: The starting position for touch typing, with the fingers over the 'home keys'. Source: [Wikipedia](https://commons.wikimedia.org/wiki/File:QWERTY-home-keys-position.svg) under the Creative Commons license.

more work less'. The evidence suggests that good diet, physical activity, plenty of sleep and a healthy work-life balance can all boost your speed and effectiveness at work (Jensen 2011; Pereira et al. 2015; Grant, Wallace, and Spurgeon 2013).

While we recommend the reader to reflect on this evidence and their own well-being, this is not a self help book. It is about programming. However, there is one non-programming skill that *can* have a huge impact on productivity: touch typing. This skill can be relatively painless to learn, and can have a huge impact on your ability to write, modify and test R code quickly. Learning to touch type properly will pay off in small increments throughout the rest of your programming life (of course, the benefits are not constrained to R programming).

The key difference between a touch typist and someone who constantly looks down at the keyboard, or who uses only two or three fingers for typing, is hand placement. Touch typing involves positioning your hands on the keyboard with each finger of both hands touching or hovering over a specific letter (figure 1.1). This takes time and some discipline to learn. Fortunately there are many resources that will help you get in the habit of touch typing early, including open source software projects Klavaro and TypeFaster.

1.6 Benchmarking

Benchmarking is the process of testing the performance of specific operations repeatedly. Modifying things from one benchmark to the next and recording the results after changing things allows experimentation to see which bits of code are fastest. Benchmarking is important in the efficient programmer's toolkit: you may *think* that your code is faster than mine but benchmarking allows you to *prove* it. The easiest way to

benchmark a function is to use `system.time()`. The **microbenchmark** package is more flexible and runs a test many times (by default 1000), enabling the user to detect microsecond difference in code performance.

1.6.1 Benchmarking example

A good example is testing different methods to look-up a single value in a data frame.

```
library("microbenchmark")
df = data.frame(v = 1:4, name = letters[1:4])
microbenchmark(df[3, 2], df[3, "name"], df$name[3])
#> Unit: microseconds
#>           expr  min   lq mean median   uq   max neval
#>       df[3, 2] 22.7 24.2 28.9   25.0 25.9 195.4   100
#>  df[3, "name"] 23.1 24.1 26.3   24.9 25.8  67.1   100
#>     df$name[3] 16.2 17.9 19.4   18.6 19.5  31.7   100
```

The results show that seemingly arbitrary changes to how R code is written can affect the efficiency of computation. Without benchmarking, these differences would be very hard to detect.

1.7 Profiling

Benchmarking generally tests execution time of one function against another. Profiling, on the other hand, is about testing large chunks of code.

It is difficult to over-emphasise the importance of profiling for efficient R programming. Without a profile of what took longest, you will have only a vague idea of why your code is taking so long to run. The example below (which generates figure 1.3 an image of ice-sheet retreat from 1985 to 2015) shows how profiling can be used to identify bottlenecks in your R scripts:

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

  # Stage 1: load packages
  library("rnoaa")
  library("ggplot2")

  # Stage 2: load and process data
  out = readRDS("data/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")
```

The result of this profiling exercise are displayed in figure 1.2.

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	

Figure 1.2: Profiling results of loading and plotting NASA data on icesheet retreat.

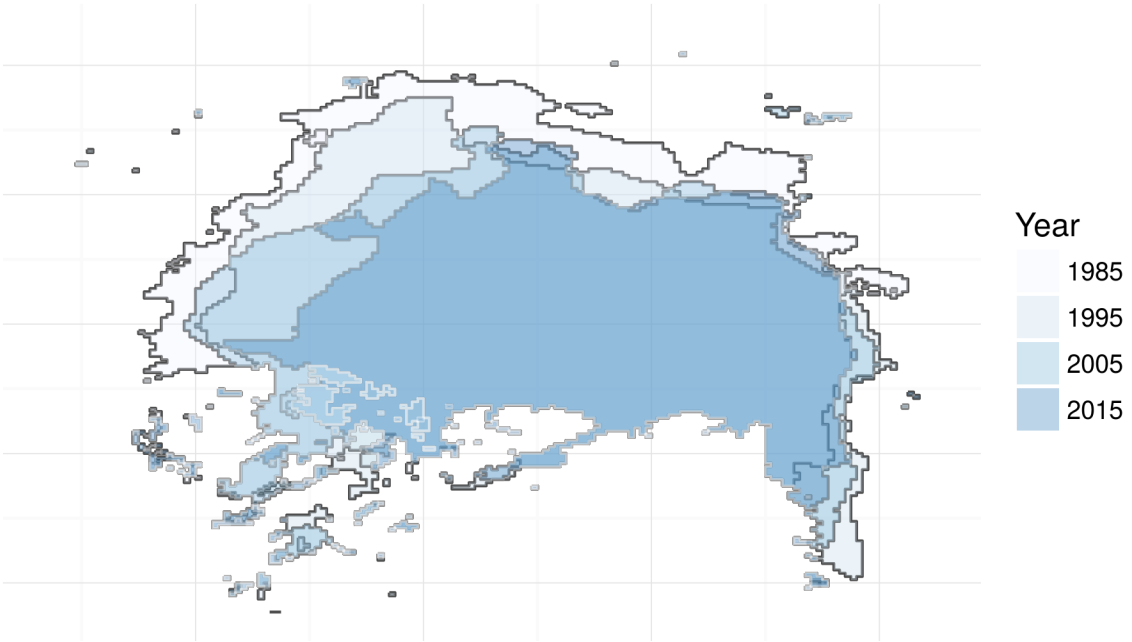


Figure 1.3: Visualisation of North Pole icesheet decline, generated using the code profiled using the profvis package.

Chapter 2

Efficient set-up

An efficient computer set-up is analogous to a well-tuned vehicle: its components work in harmony, it is well-serviced, and it is fast. This chapter describes the software decisions that will enable a productive workflow. Starting with the basics and moving to progressively more advanced topics, we explore how the operating system, R version, startup files and IDE can make your R work faster (though IDE could be seen as basic need for efficient programming). Ensuring correct configuration of these elements will have knock-on benefits in many aspects of your R workflow. That's why we cover them at this early stage (hardware, the other fundamental consideration, is covered in the next chapter. By the end of this chapter you should understand how to set-up your computer and R installation (skip to section 2.3 if R is not already installed on your computer) for optimal computational and programmer efficiency. It covers the following topics:

- R and the operating systems: system monitoring on Linux, Mac and Windows
- R version: how to keep your base R installation and packages up-to-date
- R start-up: how and why to adjust your `.Rprofile` and `.Renviron` files
- RStudio: an integrated development environment (IDE) to boost your programming productivity
- BLAS and alternative R interpreters: looks at ways to make R faster

For lazy readers, and to provide a taster of what's to come, we begin with our 'top 5' tips for an efficient R set-up. It is important to understand that efficient programming is not simply the result of following a recipe of tips: understanding is vital for knowing when to use a memorised solution to a problem and when to go back to first principles. Thinking about and *understanding* R in depth, e.g. by reading this chapter carefully, will make efficiency second nature in your R workflow.

2.1 Top 5 tips for an efficient R set-up

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

2.2 Operating system

R works on all three consumer operating systems (OS) (Linux, Mac and Windows) as well as the server-orientated Solaris OS. R is predominantly platform-independent, meaning that it should behave in the same

way on each of these platforms. This is partly facilitated by CRAN tests which ensure that R packages work on all OSs mentioned above. There are some operating system-specific quirks that may influence the choice of OS and how it is set-up for R programming in the long-term. Basic system information can be queried from within R using `Sys.info()`, as illustrated below for a selection its output:

```
Sys.info()
#R> sysname     release                 machine      user
#R> "Linux"     "4.2.0-35-generic"      "x86_64"     "robin"
```

Translated into English, this means that R is running on a 64 bit (`x86_64`) Linux distribution (kernel version `4.2.0-35-generic`) and that the current user is `robin`. Four other pieces of information (not shown) are also produced by the command, the meaning of which is well documented in `?Sys.info`.

The **assertive.reflection** package can be used to report additional information about your computer's operating system and R set-up with functions for asserting operating system and other system characteristics. The `assert_*` functions work by testing the truth of the statement and erroring if the statement is untrue. On a Linux system `assert_is_linux()` will run silently, whereas `assert_is_windows` will cause an error. The package can also test for IDE you are using (e.g. `assert_is_rstudio()`), the capabilities of R (`assert_r_has_libcurl_capability` etc.), and what OS tools are available (e.g. `assert_r_can_compile_code`). These functions can be useful for running code that designed only to run on one type of set-up.

2.2.1 Operating system and resource monitoring

Minor differences aside[1], R's computational efficiency is broadly the same across different operating systems. This is important as it means the techniques will, in general, work equally well on different OSs. Beyond the 32 vs 64 bit issue (covered in the next chapter) and *process forking* (covered in Chapter 7) the main issue for many will be user friendliness and compatibility other programs used alongside R for work. Changing operating system can be a time consuming process so our advice is usually to stick to whatever OS you are most comfortable with.

Some packages (e.g. those that must be compiled and that depend on external libraries) are best installed at the operating system level (i.e. not using `install.packages`) on Linux systems. On Debian-based operating systems such as Ubuntu, these are named with the prefix `r-cran-` (see Section 2.3.4).

Regardless of your operating system, it is good practice to track how system resources (primarily CPU and RAM use) respond when running time-consuming or RAM-intensive tasks. If you only process small datasets, system monitoring may not be necessary but when handling datasets at the limits of your computer's resources, it can be a useful tool for identifying bottlenecks, such as when you are running low on RAM. Alongside R profiling functions such as **profvis** (see Section 7.3), system monitoring can help identify performance bottlenecks and opportunities for making tasks run faster.

A common use case for system monitoring of R processes is to identify how much RAM is being used and whether more is needed (covered in Chapter 3). System monitors also report the percentage of CPU resource allocated over time. On modern multi-threaded CPUs, many tasks will use only a fraction of the available CPU resource because R is by default a single-threaded program (see Chapter 7.4 on parallel programming).

[1] Benchmarking conducted for a presentation "R on Different Platforms" at useR 2006 found that R was marginally faster on Windows than Linux set-ups. Similar results were reported in an academic paper, with R completing statistical analyses faster on a Linux than Mac OS's (Sekhon 2006). In 2015 Revolution R supported these results with slightly faster run times for certain benchmarks on Ubuntu than Mac systems. The data from the `benchmarkme` package also suggests that running code under the Linux OS is faster.

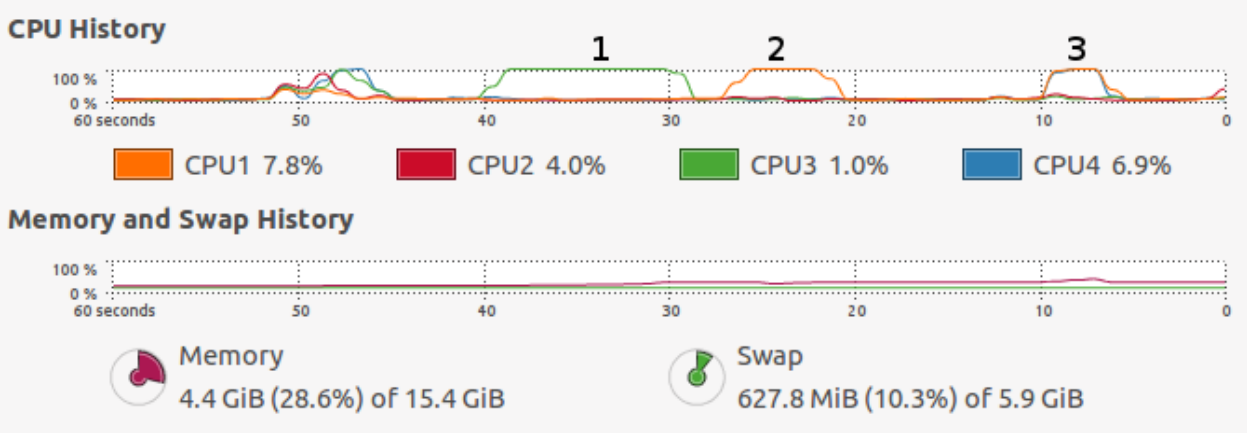


Figure 2.1: Output from a system monitor ('gnome-system-monitor' running on Ubuntu) showing the resources consumed by running the code presented in the second of the Exercises at the end of this section. The first increases RAM use, the second is single-threaded and the third is multi-threaded.

Monitoring CPU load in this context can be useful for identifying whether R is running in parallel (see Figure 2.1).

System monitoring is a complex topic that spills over into system administration and server management. Fortunately there are many tools designed to ease monitoring all major operating systems.

- On Linux, the shell command `top` displays key resource use figures for most distributions. `htop` and Gnome's **System Monitor** (`gnome-system-monitor`, see figure 2.1) are more refined alternatives which use command-line and graphical user interfaces respectively. A number of options such as `nethogs` monitor internet usage.
- On Windows the **Task Manager** provides key information on RAM and CPU use by process. This can be started in modern Windows versions by typing `Ctrl-Alt-Del` or by clicking the task bar and 'Start Task Manager'.
- On Mac the **Activity Monitor** provides similar functionality. This can be initiated from the Utilities folder in Launchpad.

Exercises

1. What is the exact version of your computer's operating system?
2. Start an activity monitor then type and execute the following code. How do the results on your system compare to those presented in Figure 2.1?

```
# 1: Create large dataset
X = matrix(rnorm(1e8), nrow = 1e7)
# 2: Find the median of each column using a single core
r1 = lapply(X, median)
# 3: Find the median of each column using many cores
r2 = parallel::mclapply(X, median) # runs in serial on Windows
```

3. What do you notice regarding CPU usage, RAM and system time, during and after each of the three operations?
4. Bonus question: how would the results change depending on operating system?

2.3 R version

It is important to be aware that R is an evolving software project, whose behaviour changes over time. In general base R is very conservative about making changes that breaks backwards compatitability. However, packages occasionally change substantially from one release to the next; typically it depends on the age of the package. For most use cases it we recommend always using the most up-to-date version of R and packages, so you have the latest code. In some circumstances (e.g. on a production server or working in a team) you may alternatively want to use specific versions which have been tested, to ensure stability. Keeping packages up-to-date is desirable because new code tends to be more efficient, intuitive, robust and feature rich. This section explains how.

Previous R versions can be installed from CRAN's archive or previous R releases. The binary versions for all OSs can be found at cran.r-project.org/bin/. To download binary versions for Ubuntu 'Wily', for example, see cran.r-project.org/bin/linux/ubuntu/wily/. To 'pin' specific versions of R packages you can use the **packrat** package. For more on pinning R versions and R packages see articles on RStudio's website Using-Different-Versions-of-R and rstudio.github.io/packrat/.

2.3.1 Installing R

The method of installing R varies for Windows, Linux and Mac.

On Windows, a single `.exe` file (hosted at cran.r-project.org/bin/windows/base/) will install the base R package.

On a Mac, the latest version should be installed by downloading the `.pkg` files hosted at cran.r-project.org/bin/macosx/.

On Debian-based systems adding the CRAN repository in the format. The following bash command will add the repository to `/etc/apt/sources.list` and keep your operating system updated with the latest version of R:

```
sudo apt-add-repository https://cran.rstudio.com/bin/linux/ubuntu
```

In the above code `cran.rstudio.com` is the 'mirror' from which `r-base` and other `r-` packages can be installed using the `apt` system. The following two commands, for example, would install the base R package (a 'bare-bones' install) and the package **rcurl**, which has an external dependency:

```
sudo apt-get install r-cran-base # install base R
sudo apt-get install r-cran-rcurl # install the rcurl package
```

Typical output from the second command is illustrate below:

```
The following extra packages will be installed:
  libcurl3-nss
The following NEW packages will be installed
  libcurl3-nss r-cran-rcurl
0 to upgrade, 2 to newly install, 0 to remove and 16 not to upgrade.
Need to get 699 kB of archives.
After this operation, 2,132 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

R also works on FreeBSD and other Unix-based systems.[2]

Once R is installed it should be kept up-to-date.

2.3.2 Updating R

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.

Release notes with details on each of these issues are hosted at cran.r-project.org/src/base/NEWS. R release versions have 3 components corresponding to major.minor.patch changes. Generally 2 or 3 patches are released before the next minor increment - each 'patch' is released roughly every 3 months. R 3.2, for example, has consisted of 3 versions: 3.2.0, 3.2.1 and 3.2.2.

- On Ubuntu-based systems, new versions of R should be automatically detected through the software management system, and can be installed with `apt-get upgrade`.
- On Mac, the latest version should be installed by the user from the `.pkg` files mentioned above.
- On Windows **installr** package makes updating easy:

```
# check and install the latest R version
installr::updateR()
```

For information about changes to expect in the next version, you can subscribe to the R's NEWS RSS feed: developer.r-project.org/blosxom.cgi/R-devel/NEWS/index.rss. It's a good way of keeping up-to-date.

2.3.3 Installing R packages

Large projects may need several packages to be installed. In this case, the required packages can be installed at once. Using the example of packages for handling spatial data, this can be done quickly and concisely with the following code:

```
pkgs = c("raster", "leaflet", "rgeos") # package names
install.packages(pkgs)
```

In the above code all the required packages are installed with two not three lines, reducing typing. Note that we can now re-use the `pkgs` object to load them all:

```
inst = lapply(pkgs, library, character.only = TRUE) # load them
```

In the above code `library(pkg[i])` is executed for every package stored in the text string vector. We use `library` here instead of `require` because the former produces an error if the package is not available.

Loading all packages at the beginning of a script is good practice as it ensures all dependencies have been installed *before* time is spent executing code. Storing package names in a character vector object such as `pkgs` is also useful because it allows us to refer back to them again and again.

[2] See jason-french.com/blog/2013/03/11/installing-r-in-linux/ for more information on installing R on a variety of Linux distributions.

2.3.4 Installing R packages with dependencies

Some packages have external dependencies (i.e. they call libraries outside R). On Unix-like systems, these are best installed onto the operating system, bypassing `install.packages`. This will ensure the necessary dependencies are installed and setup correctly alongside the R package. On Debian-based distributions such as Ubuntu, for example, packages with names starting with `r-cran-` can be search for and installed as follows (see cran.r-project.org/bin/linux/ubuntu/ for a list of these):

```
apt-cache search r-cran- # search for available cran Debian packages
sudo apt-get-install r-cran-rgdal # install the rgdal package (with dependencies)
```

On Windows the **installr** package helps manage and update R packages with system-level dependencies. For example the **Rtools** package for compiling C/C++ code on Window can be installed with the following command:

```
installr::install.rtools()
```

2.3.5 Updating R packages

An efficient R set-up will contain up-to-date packages. This can be done *for all packages* with:

```
update.packages() # update installed CRAN packages
```

The default for this function is for the `ask` argument to be set to `TRUE`, giving control over what is downloaded onto your system. This is generally desirable as updating dozens of large packages can consume a large proportion of available system resources.

To update packages automatically, you can add the line `update.packages(ask = FALSE)` to your `.Rprofile` startup file (see the next section for more on `.Rprofile`). Thanks to Richard Cotton for this tip.

An even more interactive method for updating packages in R is provided by RStudio via Tools > Check for Package Updates. Many such time saving tricks are enabled by RStudio, as described in a subsequent section. Next (after the exercises) we take a look at how to configure R using start-up files.

Exercises

1. What version of R are you using? Is it the most up-to-date?
2. Do any of your packages need updating?

2.4 R startup

Every time R starts a number of things happen. It can be useful to understand this startup process, so you can make R work the way you want it, fast. This section explains how.

2.4.1 R startup arguments

The arguments passed to the R startup command (typically simply `R` from a shell environment) determine what happens. The following arguments are particularly important from an efficiency perspective:

- `--no-environ` tells R to only look for startup files in the current working directory. (Do not worry if you don't understand what this means at present: it will become clear as the later in the section.)
- `--no-restore` tells R not to load any `.RData` files knocking around in the current working directory.
- `--no-save` tells R not to ask the user if they want to save objects saved in RAM when the session is ended with `q()`.

Adding each of these will make R load slightly faster, and mean that slightly less user input is needed when you quit. R's default setting of loading data from the last session automatically is potentially problematic in this context. See An Introduction to R, Appendix B, for more startup arguments.

Some of R's startup arguments can be controlled interactively in RStudio. See the online help file Customizing RStudio for more on this.

2.4.2 An overview of R's startup files

There are two special files, `.Renviron` and `.Rprofile`, which determine how R performs for the duration of the session. R searches and *acts on* both files *by default* every time it starts. To turn off this default behaviour, R can be launched with the command line options `--no environ` and `--no-init-file` respectively. Thus to launch R without your usual `.Rprofile` settings (e.g. to ensure that your personal settings are not responsible for an error) from a shell, one would enter the following:

```
R --no-init-file
```

But what do these mysterious files do? Their purpose is summarised in the bullet points below and the rest of this chapter is dedicated to demystifying them.

- The primary purpose of `.Renviron` is to set *environment variables*. These are settings that relate to the operating system for telling where to find external programs and the contents of user-specific variables that other users should not have access to such as *API key*, small text strings used to verify the user when interacting web services.
- `.Rprofile` is a plain text file (which is always called `.Rprofile`, hence its name) that simply runs lines of R code every time R starts. If you want R to check for package updates each time it starts (as explained in the previous section), you simply add the relevant line somewhere in this file.

When R starts (unless it was launched with `--no-environ`) it first searches for `.Renviron` and then `.Rprofile`, in that order. Although `.Renviron` is searched for first, we will look at `.Rprofile` first as it is simpler and for many set-up tasks more frequently useful. Both files can exist in three directories on your computer.

Modification of R's startup files should not be taken lightly. This is an advanced topic. If you modify your startup files in the wrong way, it can cause problems. We recommend proceeding with caution and take a conservative approach to modifying your startup files so that your R installation behaves similar to other 'vanilla' R installations, aiding reproducibility.

2.4.3 The location of startup files

Confusingly, multiple versions of these files can exist on the same computer, only one of which will be used per session. Note also that these files should only be changed with caution and if you know what you are doing. This is because they can make your R version behave differently to other R installations, potentially reducing the reproducibility of your code.

Files in three folders are important in this process:

- `R_HOME`, the directory in which R is installed. The `etc` sub-directory can contain start-up files read early on in the start-up process. Find out where your `R_HOME` is with the `R.home()` command.
- `HOME`, the user's home directory. Typically this is `/home/username` on Unix machines or `C:\Users\username` on Windows (since Windows 7). Ask R where your home directory with, `Sys.getenv("HOME")`.
- R's current working directory. This is reported by `getwd()`.

It is important to know the location of the `.Rprofile` and `.Renviron` set-up files that are being used out of these three options. R only uses one `.Rprofile` and one `.Renviron` in any session: if you have a `.Rprofile` file in your current project, R will ignore `.Rprofile` in `R_HOME` and `HOME`. Likewise, `.Rprofile` in `HOME` overrides `.Rprofile` in `R_HOME`. The same applies to `.Renviron`: you should remember that adding project specific environment variables with `.Renviron` will de-activate other `.Renviron` files.

To create a project-specific start-up script, simply create a `.Rprofile` file in the project's root directory and start adding R code, e.g. via `file.edit(".Rprofile")`. Remember that this will make `.Rprofile` in the home directory be ignored. The following commands will open your `.Rprofile` from within an R editor:

```
file.edit("~/.Rprofile") # edit .Rprofile in HOME
file.edit(".Rprofile") # edit project specific .Rprofile
```

File paths provided by Windows operating systems will not always work in R. Specifically, if you use a path that contains single backslashes, such as `C:\\DATA\\data.csv`, as provided by Windows, this will generate the error: `Error: unexpected input in "C:\\"`. To overcome this issue R provides two functions, `file.path` and `normalizePath`. The former can be used to specify file locations without having to use symbols to represent relative file paths, as follows: `file.path("C:", "DATA", "data.csv")`. The latter takes any input string for a file name and outputs a text string that is standard (cannonical) for the operating system. `normalizePath("C:/DATA/data.csv")`, for example, outputs `C:\\DATA\\data.csv` on a Windows machine but `C:/DATA/data.csv` on Unix-based platforms. Note that only the latter would work on both platforms so standard Unix file path notation is safe for all operating systems.

Editing the `.Renviron` file in the same locations will have the same effect. The following code will create a user specific `.Renviron` file (where API keys and other cross-project environment variables can be stored), without overwriting any existing file.

```
user_renviron = path.expand(file.path("~", ".Renviron"))
if(!file.exists(user_renviron)) # check to see if the file already exists
  file.create(user_renviron)
file.edit(user_renviron) # open with another text editor if this fails
```

The **pathological** package can help find where `.Rprofile` and `.Renviron` files are located on your system, thanks to the `os_path()` function. The output of `example(startup)` is also instructive.

The location, contents and uses of each is outlined in more detail below.

2.4.4 The `.Rprofile` file

By default, R looks for and runs `.Rprofile` files in the three locations described above, in a specific order. `.Rprofile` files are simply R scripts that run each time R runs and they can be found within `R_HOME`, `HOME` and the project's home directory, found with `getwd()`. To check if you have a site-wide `.Rprofile`, which will run for all users on start-up, run:

```
site_path = R.home(component = "home")
fname = file.path(site_path, "etc", "Rprofile.site")
file.exists(fname)
```

The above code checks for the presence of `Rprofile.site` in that directory. As outlined above, the `.Rprofile` located in your home directory is user-specific. Again, we can test whether this file exists using

```
file.exists("~/.Rprofile")
```

We can use R to create and edit `.Rprofile` (warning: do not overwrite your previous `.Rprofile` - we suggest you try project-specific `.Rprofile` first):

```
if(!file.exists("~/.Rprofile")) # only create if not already there
  file.create("~/.Rprofile")    # (don't overwrite it)
file.edit("~/.Rprofile")
```

2.4.5 An example `.Rprofile` file

The example below provides a taster of what goes into `.Rprofile`. Note that this is simply a usual R script, but with an unusual name. The best way to understand what is going on is to create this same script, save it as `.Rprofile` in your current working directory and then restart your R session to observer what changes. To restart your R session from within RStudio you can click `Session > Restart R` or use the keyboard shortcut `Ctrl+Shift+F10`.

```
# A fun welcome message
message("Hi Robin, welcome to R")
# Customise the R prompt that prefixes every command
# (use " " for a blank prompt)
options(prompt = "R4geo> ")
# Don't convert text strings to factors with base read functions
options(stringsAsFactors = FALSE)
```

To quickly explain each line of code: the first simply prints a message in the console each time a new R session is started. The latter two modify *options* used to change R's behavior, first to change the prompt in the console (set to `R>` by default) and second to ensure that unwanted `factor` variables are not created when `read.csv` and other functions derived from `read.table` are used to load external data into R. Note that simply adding more lines the `.Rprofile` will set more features. An important aspect of `.Rprofile` (and `.Renviron`) is that *each line is run once and only once for each R session*. That means that the options set within `.Rprofile` can easily be changed during the session. The following command run mid-session, for example, will return the default prompt:

```
options(prompt = "> ")
```

More details on these, and other potentially useful `.Rprofile` options are described subsequently. For more suggestions of useful startup settings, see Examples in `help("Startup")` and online resources such as those at statmethods.net. The help pages for R options (accessible with `?options`) are also worth a read before writing you own `.Rprofile`.

Ever been frustrated by unwanted `+` symbols that prevent copyied and pasted multi-line functions from working? These potentially annoying `+`s can be erradicated by adding `options(continue = "  ")` to your `.Rprofile`.

2.4.5.1 Setting options

The function `options`, used above, contains a number of default settings. Typing `options()` provides a good indication of what be configured. Because `options()` are often related to personal preference (with few implications for reproducibility), that you will want for many your R sessions, `.Rprofile` in your home directory or in your project's folder are sensible places to set them. Other illustrative options are shown below:

```
options(prompt = "R> ", digits = 4, show.signif.stars = FALSE)
```

This changes three default options in a single line.

- The R prompt, from the boring `>` to the exciting `R>`.
- The number of digits displayed.
- Removing the stars after significant p-values.

Try to avoid adding options to the start-up file that make your code non-portable. The `stringsAsFactors = FALSE` argument used above, for example, to your start-up script has knock-on effects for `read.table` and related functions including `read.csv`, making them convert text strings into characters rather than into factors as is default. This may be useful for you, but can make your code less portable, so be warned.

2.4.5.2 Setting the CRAN mirror

To avoid setting the CRAN mirror each time you run `install.packages` you can permanently set the mirror in your `.Rprofile`.

```
# `local` creates a new, empty environment
# This avoids polluting .GlobalEnv with the object r
local({
  r = getOption("repos")
  r["CRAN"] = "https://cran.rstudio.com/"
  options(repos = r)
})
```

The RStudio mirror is a virtual machine run by Amazon's EC2 service, and it syncs with the main CRAN mirror in Austria once per day. Since RStudio is using Amazon's CloudFront, the repository is automatically distributed around the world, so no matter where you are in the world, the data doesn't need to travel very far, and is therefore fast to download.

2.4.5.3 The fortunes package

This section illustrate what `.Rprofile` does with reference to a package that was developed for fun. The code below could easily be altered to automatically connect to a database, or ensure that the latest packages have been downloaded.

The **fortunes** package contains a number of memorable quotes that the community has collected over many years, called R fortunes. Each fortune has a number. To get fortune number 50, for example, enter

```
fortunes::fortune(50)
#>
#> To paraphrase provocatively, 'machine learning is statistics minus any
#> checking of models and assumptions'.
#>     -- Brian D. Ripley (about the difference between machine learning and
#>        statistics)
#>        useR! 2004, Vienna (May 2004)
```

It is easy to make R print out one of these nuggets of truth each time you start a session, by adding the following to `~/.Rprofile`:

```
if(interactive())
  try(fortunes::fortune(), silent = TRUE)
```

The `interactive` function tests whether R is being used interactively in a terminal. The `fortune` function is called within `try`. If the **fortunes** package is not available, we avoid raising an error and move on. By using `::` we avoid adding the **fortunes** package to our list of attached packages.

Typing `search()`, gives the list of attached packages. By using `fortunes::fortune()` we avoid adding the **fortunes** package to that list.

The function `.Last`, if it exists in the `.Rprofile`, is always run at the end of the session. We can use it to install the **fortunes** package if needed. To load the package, we use `require`, since if the package isn't installed, the `require` function returns `FALSE` and raises a warning.

```
.Last = function() {
  cond = suppressWarnings(!require(fortunes, quietly = TRUE))
  if(cond)
    try(install.packages("fortunes"), silent = TRUE)
  message("Goodbye at ", date(), "\n")
}
```

2.4.5.4 Useful functions

You can use `.Rprofile` define new 'helper' functions or redefine existing ones so they're faster to type. For example, we could load the following two functions for examining data frames:

```
# ht == headtail
ht = function(d, n=6) rbind(head(d, n), tail(d, n))
# Show the first 5 rows & first 5 columns of a data frame
hh = function(d) d[1:5, 1:5]
```

and a function for setting a nice plotting window:

```
nice_par = function(mar = c(3, 3, 2, 1), mgp = c(2, 0.4, 0), tck = -0.01,
                    cex.axis = 0.9, las = 1, mfrow = c(1, 1), ...) {
    par(mar = mar, mgp = mgp, tck = tck, cex.axis = cex.axis, las = las,
        mfrow = mfrow, ...)
}
```

Note that these functions are for personal use and are unlikely to interfere with code from other people. For this reason even if you use a certain package every day, we don't recommend loading it in your `.Rprofile`. Shortening long function names for interactive (but not reproducible code writing). If you frequently use `View()`, for example, you may be able to save time by referring to it in abbreviated form. This is illustrated below to make it faster to view datasets (although with IDE-driven autocompletion, outlined in the next section, the time savings is less.)

```
v = utils::View
```

Also beware the dangers of loading many functions by default: it may make your code less portable. Another potentially useful setting to change in `.Rprofile` is R's current working directory. If you want R to automatically set the working directory to the R folder of your project, for example, one would add the following line of code to the project-specific `.Rprofile`:

```
setwd("R")
```

2.4.5.5 Creating hidden environments with .Rprofile

Beyond making your code less portable, another downside of putting functions in your `.Rprofile` is that it can clutter-up your work space: when you run the `ls()` command, your `.Rprofile` functions will appear. Also if you run `rm(list=ls())`, your functions will be deleted. One neat trick to overcome this issue is to use hidden objects and environments. When an object name starts with `.`, by default it doesn't appear in the output of the `ls()` function

```
.obj = 1
".obj" %in% ls()
#> [1] FALSE
```

This concept also works with environments. In the `.Rprofile` file we can create a *hidden* environment

```
.env = new.env()
```

and then add functions to this environment

```
.env$ht = function(d, n = 6) rbind(head(d, n), tail(d, n))
```

At the end of the `.Rprofile` file, we use `attach`, which makes it possible to refer to objects in the environment by their names alone.

```
attach(.env)
```

2.4.6 The `.Renviron` file

The `.Renviron` file is used to store system variables. It follows a similar start-up routine to the `.Rprofile` file: R first looks for a global `.Renviron` file, then for local versions. A typical use of the `.Renviron` file is to specify the `R_LIBS` path, which determines where new packages are installed:

```
# Linux
R_LIBS=~/R/library
# Windows
R_LIBS=C:/R/library
```

After setting this, `install.packages` saves packages in the directory specified by `R_LIBS`. The location of this directory can be referred back to subsequently as follows:

```
Sys.getenv("R_LIBS_USER")
#> [1] "/home/travis/R/Library"
```

All currently stored environment variables can be seen by calling `Sys.getenv()` with no arguments. Note that many environment variables are already pre-set and do not need to be specified in `.Renviron`. `HOME`, for example, which can be seen with `Sys.getenv('HOME')`, is taken from the operating system's list of environment variables. A list of the most important environment variables that can affect R's behaviour is documented in the little known help page `help("environment variables")`.

To set or unset environment variable for the duration of a session, use the following commands:

```
Sys.setenv("TEST" = "test-string") # set an environment variable for the session
Sys.unsetenv("TEST") # unset it
```

Another common use of `.Renviron` is to store API keys and authentication tokens that will be available from one session to another.[3] A common use case is setting the 'envvar' `GITHUB_PAT`, which will be detected by the **devtools** package via the function `github_pat()`. To take another example, the following line in `.Renviron` sets the `ZEIT_KEY` environment variable which is used in the **diezeit** package:

[3] See `vignette("api-packages")` from the `httr` package for more on this.

```
ZEIT_KEY=PUT_YOUR_KEY_HERE
```

You will need to sign-in and start a new R session for the environment variable (accessed by `Sys.getenv`) to be visible. To test if the example API key has been successfully added as an environment variable, run the following:

```
Sys.getenv("ZEIT_KEY")
```

Use of the `.Renviron` file for storing settings such as library paths and API keys is efficient because it reduces the need to update your settings for every R session. Furthermore, the same `.Renviron` file will work across different platforms so keep it stored safely.

2.4.6.1 Example `.Renviron` file

My `.Renviron` file has grown over the years. I often switch between my desktop and laptop computers, so to maintain a consistent working environment, I have the same `.Renviron` file on all of my machines. As well as containing an `R_LIBS` entry and some API keys, my `.Renviron` has a few other lines:

- `TMPDIR=/data/R_tmp/`. When R is running, it creates temporary copies. On my work machine, the default directory is a network drive.
- `R_COMPILE_PKGS=3`. Byte compile all packages (covered in Chapter 3).
- `R_LIBS_SITE=/usr/lib/R/site-library:/usr/lib/R/library` I explicitly state where to look for packages. My University has a site-wide directory that contains out of date packages. I want to avoiding using this directory.
- `R_DEFAULT_PACKAGES=utils,grDevices,graphics,stats,methods`. Explicitly state the packages to load. Note I don't load the `datasets` package, but I ensure that `methods` is always loaded. Due to historical reasons, the `methods` package isn't loaded by default in certain applications, e.g. `Rscript`.

Exercises

1. What are the three locations where the startup files are stored? Where are these locations on your computer?
2. For each location, does a `.Rprofile` or `.Renviron` file exist?
3. Create a `.Rprofile` file in your current working directory that prints the message `Happy efficient R programming` each time you start R at this location.
4. What happens to the startup files in `R_HOME` if you create them in `HOME` or local project directories?

2.5 RStudio

RStudio is an Integrated Development Environment (IDE) for R. It makes life easy for R users and developers with its intuitive and flexible interface. RStudio encourages good programming practice. Through its wide range of features RStudio can help make you a more efficient and productive R programmer. RStudio can, for example, greatly reduce the amount of time spent remembering and typing function names thanks to intelligent autocompletion. Some of the most important features of RStudio include:

- Flexible window pane layouts to optimise use of screen space and enable fast interactive visual feed-back.

- Intelligent autocompletion of function names, packages and R objects.
- A wide range of keyboard shortcuts.
- Visual display of objects, including a searchable data display table.
- Real-time code checking and error detection.
- Menus to install and update packages.
- Project management and integration with version control.

The above list of features should make it clear that a well set-up IDE can be as important as a well set-up R installation for becoming an efficient R programmer.[4] As with R itself, the best way to learn about RStudio is by using it. It is therefore worth reading through this section in parallel with using RStudio to boost your productivity.

2.5.1 Installing and updating RStudio

RStudio can be installed from the RStudio website rstudio.com and is available for all major operating systems. Updating RStudio is simple: click on `Help > Check for Updates` in the menu. For fast and efficient work keyboard shortcuts should be used wherever possible, reducing the reliance on the mouse. RStudio has many keyboard shortcuts that will help with this. To get into good habits early, try accessing the RStudio Update interface without touching the mouse. On Linux and Windows dropdown menus are activated with the `Alt` button, so the menu item can be found with:

```
Alt+H U
```

On Mac it works differently. `Cmd+?` should activate a search across menu items, allowing the same operation can be achieved with:

```
Cmd+? update
```

Note: in RStudio the keyboard shortcuts differ between Linux and Windows versions on one hand and Mac on the other. In this section we generally only use the Windows/Linux shortcut keys for brevity. The Mac equivalent is usually found by simply replacing `Ctrl` and `Alt` with the Mac-specific `Cmd` button.

2.5.2 Window pane layout

RStudio has four main window 'panes' (see Figure 2.2), each of which serves a range of purposes:

- The **Source pane**, for editing, saving, and dispatching R code to the console (top left). Note that this pane does not exist by default when you start RStudio: it appears when you open an R script, e.g. via File -> New File -> R Script. A common task in this pane is to send code on the current line to the console, via `Ctrl-Enter` (or `Cmd-Enter` on Mac).
- The **Console pane**. Any code entered here is processed by R, line by line. This pane is ideal for interactively testing ideas before saving the final results in the Source pane above.
- The **Environment pane** (top right) contains information about the current objects loaded in the workspace including their class, dimension (if they are a data frame) and name. This pane also contains tabbed sub-panes with a searchable history that was dispatched to the console and (if applicable to the project) Build and Git options.
- The **Files pane** (bottom right) contains a simple file browser, a Plots tab, Help and Package tabs and a Viewer for visualising interactive R output such as those produced by the leaflet package and HTML 'widgets'.

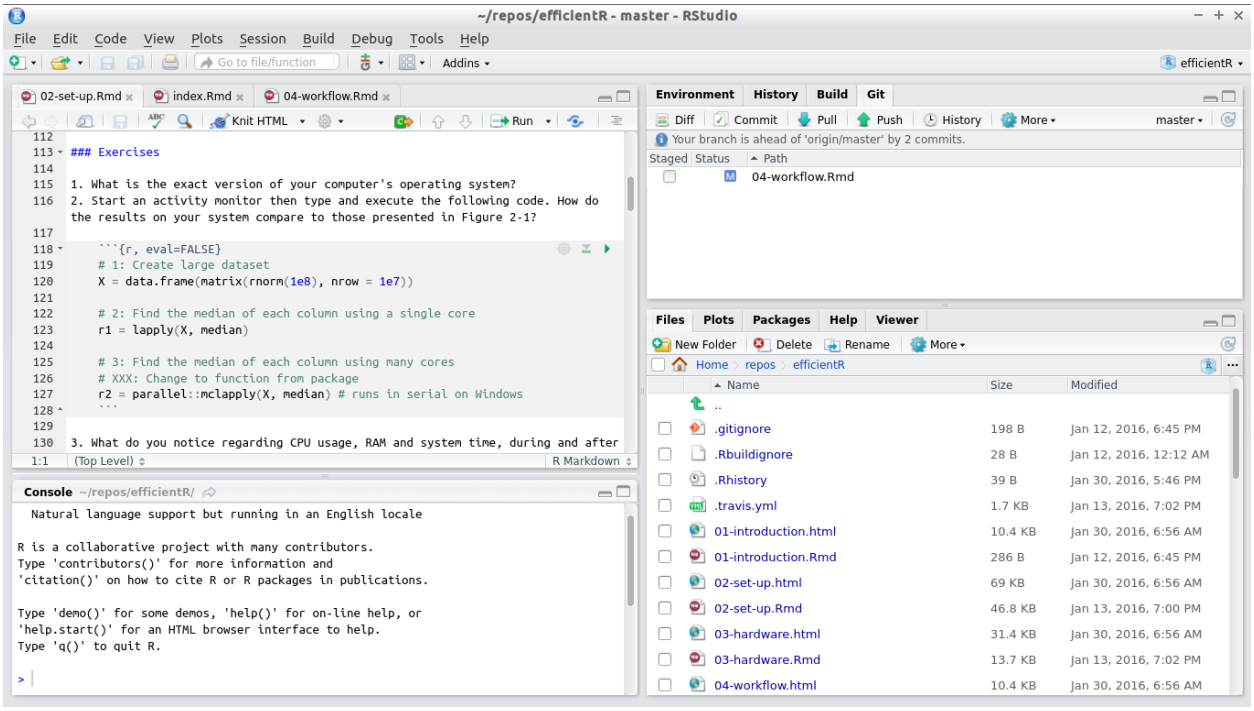


Figure 2.2: RStudio Panels

Using each of the panels effectively and navigating between them quickly is a skill that will develop over time, and will only improve with practice.

Exercises

You are developing a project to visualise data. Test out the multi-panel RStudio workflow by following the steps below:

1. Create a new folder for the input data using the **Files pane**.
2. Type in `downl` in the **Source pane** and hit `Enter` to make the function `download.file()` autocomplete. Then type `"`, which will autocomplete to `""`, paste the URL of a file to download (e.g. `https://www.census.gov/2010census/csv/pop_change.csv`) and a file name (e.g. `pop_change.csv`).
3. Execute the full command with `Ctrl-Enter`:

```
download.file("https://www.census.gov/2010census/csv/pop_change.csv",
              "data/pop_change.csv")
```

4. Write and execute a command to read-in the data, such as

```
pop_change = read.csv("data/pop_change.csv", skip = 2)
```

5. Use the **Environment pane** to click on the data object `pop_change`. Note that this runs the command `View(pop_change)`, which launches an interactive data explore pane in the top left panel (see Figure 2.3).

[4]Other open source R IDEs exist, including RKWard, Tinn-R and JGR. emacs is another popular software environment. However, it has a very steep learning curve.

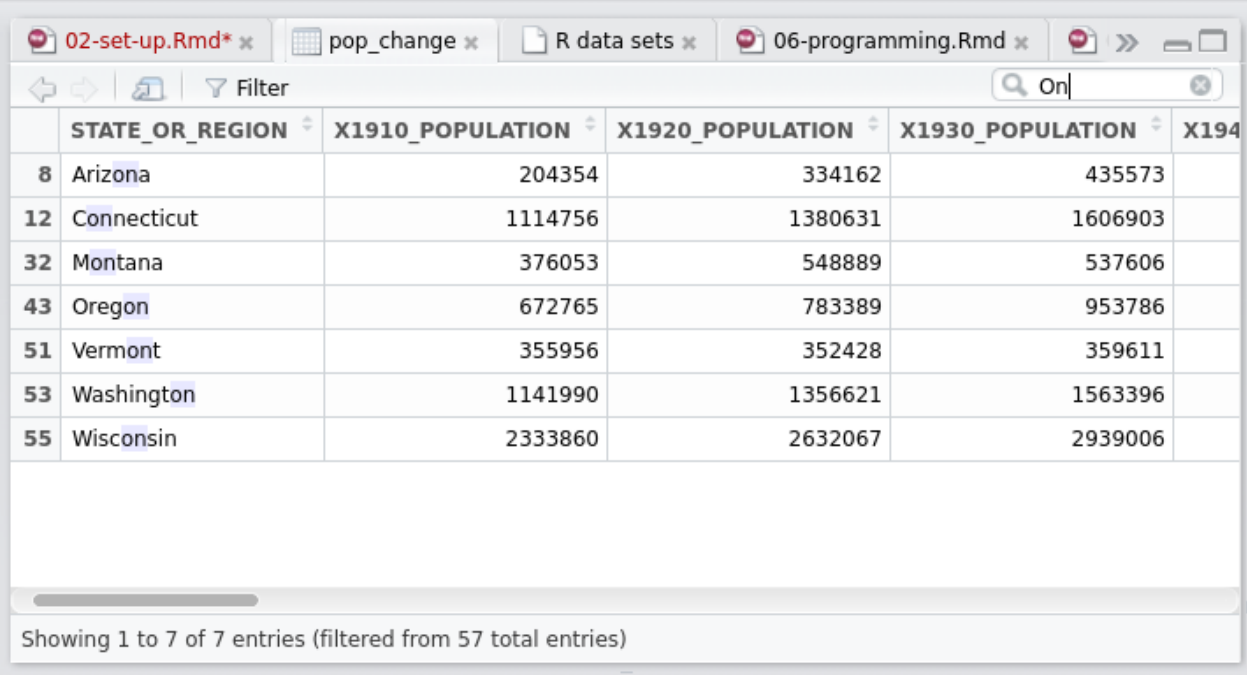


Figure 2.3: The data viewing tab in RStudio.

6. Use the **Console** to test different plot commands to visualise the data, saving the code you want to keep back into the **Source pane**, as `pop_change.R`.

7. Use the **Plots tab** in the Files pane to scroll through past plots. Save the best using the Export dropdown button.

The above example shows understanding of these panes and how to use them interactively can help with the speed and productivity of you R programming. Further, there are a number of RStudio settings that can help ensure that it works for your needs.

2.5.3 RStudio options

A range of `Project Options` and `Global Options` are available in RStudio from the `Tools` menu (accessible in Linux and Windows from the keyboard via `Alt+T`). Most of these are self-explanatory but it is worth mentioning a few that can boost your programming efficiency:

- GIT/SVN project settings allow RStudio to provide a graphical interface to your version control system, described in Chapter XX.
- R version settings allow RStudio to 'point' to different R versions/interpreters, which may be faster for some projects.
- `Restore .RData`: Unticking this default preventing loading previously creating R objects. This will make starting R quicker and also reduce the change of getting bugs due to previously created objects.
- Code editing options can make RStudio adapt to your coding style, for example, by preventing the autocompletion of braces, which some experienced programmers may find annoying. Enabling `Vim mode` makes RStudio act as a (partial) Vim emulator.

- Diagnostic settings can make RStudio more efficient by adding additional diagnostics or by removing diagnostics if they are slowing down your work. This may be an issue for people using RStudio to analyse large datasets on older low-spec computers.
- Appearance: if you are struggling to see the source code, changing the default font size may make you a more efficient programmer by reducing the time overheads associated with squinting at the screen. Other options in this area relate more to aesthetics, which are also important because feeling comfortable in your programming environment can boost productivity.

2.5.4 Autocompletion

R provides some basic autocompletion functionality. Typing the beginning of a function name, for example `rn` (short for `rnorm()`), and hitting `Tab` twice will result in the full function names associated with this text string being printed. In this case two options would be displayed: `rnbinom` and `rnorm`, providing a useful reminder to the user about what is available. The same applies to file names enclosed in quote marks: typing `te` in the console in a project which contains a file called `test.R` should result in the full name `"test.R"` being autocompleted. RStudio builds on this functionality and takes it to a new level.

The default settings for autocompletion in RStudio work well. They are intuitive and are likely to work well for many users, especially beginners. However, RStudio's autocompletion options can be modfified, but navigating to **Tools > Global Options > Code > Completion** in RStudio's top level menu.

Instead of only auto completing options when `Tab` is pressed, RStudio auto completes them at any point. Building on the previous example, RStudio's autocompletion triggers when the first three characters are typed: `rno`. The same functionality works when only the first characters are typed, followed by `Tab`: automatic autocompletion does not replace `Tab` autocompletion but supplements it. Note that in RStudio two more options are provided to the user after entering `rn Tab` compared with entering the same text into base R's console described in the previous paragraph: `RNGkind` and `RNGversion`. This illustrates that RStudio's autocompletion functionality is not case sensitive in the same way that R is. This is a good thing because R has no consistent function name style!

RStudio also has more intelligent autocompletion of objects and file names than R's built-in command line. To test this functionality, try typing `US`, followed by the Tab key. After pressing down until `USArrests` is selected, press Enter so it autocompletes. Finally, typing `$` should leave the following text on the screen and the four columns should be shown in a drop-down box, ready for you to select the variable of interest with the down arrow.

```
USArrests$ # a dropdown menu of columns should appear in RStudio
```

To take a more complex example, variable names stored in the `data` slot of the class `SpatialPolygonsDataFrame` (a class defined by the foundational spatial package sp) are referred to in the long form `spdf@data$varname`.[5] In this case `spdf` is the object name, `data` is the slot and `varname` is the variable name. RStudio makes such `S4` objects easier to use by enabling autocompletion of the short form `spdf$varname`. Another example is RStudio's ability to find files hidden away in sub-folders. Typing `"te` will find `test.R` even if it is located in a sub-folder such as `R/test.R`. There are a number of other clever autocompletion tricks that can boost R's productivity when using RStudio which are best found by experimenting and hitting `Tab` frequently during your R programming work.

[5] 'Slots' are elements of an object (specifically, S4 objects) analogous to a column in a `data.frame` but referred to with `@` not `$`.

2.5.5 Keyboard shortcuts

RStudio has many useful shortcuts that can help make your programming more efficient by reducing the need to reach for the mouse and point and click your way around code and RStudio. These can be viewed by using a little known but extremely useful keyboard shortcut (this can also be accessed via the **Tools** menu).

```
Alt+Shift+K
```

This will display the default shortcuts in RStudio. It is worth spending time identifying which of these could be useful in your work and practising interacting with RStudio rapidly with minimal reliance on the mouse. The power of these autocompletion capabilities can be further enhanced by setting your own keyboard shortcuts. However, as with setting `.Rprofile` and `.Renviron` settings, this risks reducing the portability of your workflow. To set your own RStudio keyboard shortcuts, navigate to **Tools > Modify Keyboard Shortcuts**.

Some more useful shortcuts are listed below. There are many more gems to find that could boost your R writing productivity:

- `Ctrl+Z/Shift+Z`: Undo/Redo.
- `Ctrl+Enter`: Execute the current line or code selection in the Source pane.
- `Ctrl+Alt+R`: Execute all the R code in the currently open file in the Source pane.
- `Ctrl+Left/Right`: Navigate code quickly, word by word.
- `Home/End`: Navigate to the beginning/end of the current line.
- `Alt+Shift+Up/Down`: Duplicate the current line up or down.
- `Ctrl+D`: Delete the current line.

2.5.6 Object display and output table

It is useful to know what is in your current R environment. This information can be revealed with `ls()`, but this function only provides object names. RStudio provides an efficient mechanism to show currently loaded objects, and their details, in real-time: the Environment tab in the top right corner. It makes sense to keep an eye on which objects are loaded and to delete objects that are no longer useful. Doing so will minimise the probability of confusion in your workflow (e.g. by using the wrong version of an object) and reduce the amount of RAM R needs. The details provided in the Environment tab include the object's dimension and some additional details depending on the object's class (e.g. size in MB for large datasets).

A very useful feature of RStudio is its advanced viewing functionality. This is triggered either by executing `View(object)` or by double clicking on the object name in the Environment tab. Although you cannot edit data in the Viewer (this should be considered a good thing from a data integrity perspective), recent versions of RStudio provide an efficient search mechanism to rapidly filter and view the records that are of most interest (see Figure 2.3).

2.5.7 Project management

In the far top-right of RStudio there is a diminutive drop-down menu illustrated with R inside a transparent box. This menu may be small and simple, but it is hugely efficient in terms of organising large, complex and long-term projects.

The idea of RStudio projects is that the bulk of R programming work is part of a wider task, which will likely consist of input data, R code, graphical and numerical outputs and documents describing the work. It is possible to scatter each of these elements at random across your hard-discs but this is not recommended. Instead, the concept of projects encourages reproducible working, such that anyone who opens the particular project folder that you are working from should be able to repeat your analyses and replicate your results.

It is therefore *highly recommended* that you use projects to organise your work. It could save hours in the long-run. Organizing data, code and outputs also makes sense from a portability perspective: if you copy the folder (e.g. via GitHub) your can work on it from any computer without worrying about having the right files on your current machine. These tasks are implemented using RStudio's simple project system, in which the following things happen each time you open an existing project:

- The working directory automatically switches to the project's folder. This enables data and script files to be referred to using relative file paths, which are much shorter than absolute file paths. This means that switching directory using `setwd()`, a common source of error for R users, is rarely if ever needed.
- 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. This assists with continuity between one session and the next.
- The `File` tab displays the associated files and folders in the project, allowing you to quickly find your previous work.
- Any settings associated with the project, such as Git settings, are loaded. This assists with collaboration and project-specific set-up.

Each project is different but most contain input data, R code and outputs. To keep things tidy, we recommend a sub-directory structure resembling the following:

```
project/
  - README.Rmd # Project description
  - set-up.R  # Required packages
  - R/ # For R code
  - input # Data files
  - graphics/
  - output/ # Results
```

Proper use of projects ensures that all R source files are neatly stashed in one folder with a meaningful structure. This way data and documentation can be found where one would expect them. Under this system figures and project outputs are 'first class citizens' within the project's design, each with their own folder.

Another approach to project management is to treat projects as R packages. This is not recommended for most use cases, as it places restrictions on where you can put files. However, if the aim is *code development and sharing*, creating a small R package may be the way forward, even if you never intend to submit it on CRAN. Creating R packages is easier than ever before, as documented in (Cotton 2013) and, more recently (Wickham 2015). The **devtools** package help manage R's quirks, making the process much less painful. If you use GitHub, the advantage of this approach is that anyone should be able to reproduce your working using `devtools::install_github("username/projectname")`, although the administrative overheads of creating an entire package for each small project will outweigh the benefits for many.

Note that a `set-up.R` or even a `.Rprofile` file in the project's root directory enable project-specific settings to be loaded each time people work on the project. As described in the previous section, `.Rprofile` can be used to tweak how R works at start-up. It is also a portable way to manage R's configuration on a project-by-project basis.

Exercises

1. Try modifying the look and appearance of your RStudio setup.
2. What is the keyboard shortcut to show the other shortcut? (Hint: it begins with `Alt+Shift` on Linux and Windows.)

3. Try as many of the shortcuts revealed by the previous step as you like. Write down the ones that you think will save you time, perhaps on a post-it note to go on your computer.

2.6 BLAS and alternative R interpreters

In this section we cover a few system-level options available to speed-up R's performance. 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 and b) you are confident tweaking system-level settings. This should therefore be seen as an advanced section: if you are not interested in speeding-up base R, feel free to skip to the next section of hardware.

Many statistical algorithms manipulate matrices. R uses the Basic Linear Algebra System (BLAS) framework for linear algebra operations. Whenever we carry out a matrix operation, such as transpose or finding the inverse, we use the underlying BLAS library. By switching to a different BLAS library, it may be possible to speed-up your R code. Changing your BLAS library is straightforward if you are using Linux, but can be tricky for Windows users.

The two open source alternative BLAS libraries are ATLAS and OpenBLAS. The Intel MKL is another implementation, designed for Intel processors by Intel and used in Revolution R (described in the next section) but it requires licensing fees. The MKL library is provided with the Revolution analytics system. Depending on your application, by switching you BLAS library, linear algebra operations can run several times faster than with the base BLAS routines.

If you use Linux, you can find whether you have a BLAS library setting with the following function, from **benchmarkme**:

```
library("benchmarkme")
get_linear_algebra()
```

2.6.1 Testing performance gains from BLAS

As an illustrative test of the performance gains offered by BLAS, the following test was run on a new laptop running Ubuntu 15.10 on a 6th generation Core i7 processor, before and after OpenBLAS was installed.[6]

```
res = benchmark_std() # run a suit of tests to test R's performance
```

It was found that the installation of OpenBLAS led to a 2-fold speed-up (from around 150 to 70 seconds). The majority of the speed gain was from the matrix algebra tests, as can be seen in figure 2.4. Note that the results of such tests are highly dependent on the particularities of each computer. However, it clearly shows that 'programming' benchmarks (e.g. the calculation of 3,500,000 Fibonacci numbers) are now much faster, whereas matrix calculations and functions receive a substantial speed boost. This demonstrates that the speed-up you can expect from BLAS depends heavily on the type of computations you are undertaking.

2.6.2 Other interpreters

The R *language* can be separated from the R *interpreter*. The former refers to the meaning of R commands, the latter refers to how the computer executes the commands. Alternative interpreters have been developed to try to make R faster and, while promising, none of the following options has fully taken off.

[6] OpenBLAS was installed on the computer via `sudo apt-get install libopenblas-base`, which automatically detected and used by R.

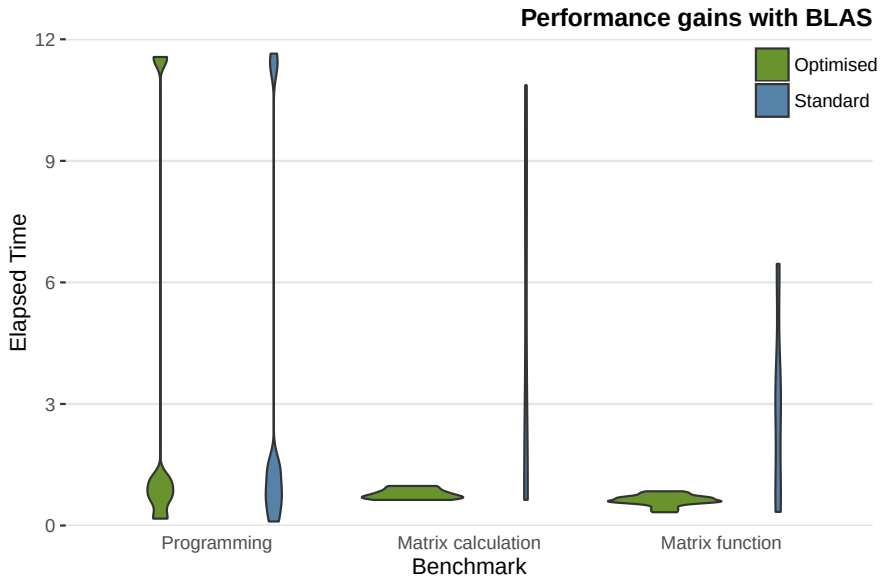


Figure 2.4: Performance gains obtained changing the underlying BLAS library (tests from 'benchmark_std()').

- Microsoft R Open, formerly known as Revolution R Open (RRO), is the enhanced distribution of R from Microsoft. The key enhancement is that is uses multithreaded mathematics libraries, which can improve performance.
- Rho (previously called CXXR, short for C++), a re-implementation of the R interpreter for speed and efficiency. Of the new interpreters, this is the one that has the most recent development activity (as of April 2016).
- pqrR (pretty quick R) is a new version of the R interpreter. One major downside, is that it is based on R-2.15.0. The developer (Radford Neal) has made many improvements, some of which have now been incorporated into base R. **pqR** is an open-source project licensed under the GPL. One notable improvement in pqR is that it is able to do some numeric computations in parallel with each other, and with other operations of the interpreter, on systems with multiple processors or processor cores.
- 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.
- Tibco created a C++ based interpreter called TERR.
- Oracle also offer an R-interpreter that uses Intel's mathematics library and therefore achieves a higher performance without changing R's core.

At the time of writing, switching interpreters is something to consider carefully. But in the future, it may become more routine.

2.6.3 Useful BLAS/benchmarking resources

- The gcbd package benchmarks performance of a few standard linear algebra operations across a number of different BLAS libraries as well as a GPU implementation. It has an excellent vignette summarising the results.
- Brett Klamer provides a nice comparison of ATLAS, OpenBLAS and Intel MKL BLAS libraries. He also gives a description of how to install the different libraries.
- The official R manual section on BLAS.

Exercises

1. What BLAS system is your version of R using?

Chapter 3

Efficient programming

Many people who use R would not describe themselves as "programmers". Instead they tend to have advanced domain level knowledge, understand standard R data structures, such as vectors and data frames, but have little formal training in computing. Sound familiar? In that case this chapter is for you.

In this chapter we will discuss "big picture" programming techniques. We cover general concepts and R programming techniques about code optimisation, before describing idiomatic programming structures. We conclude the chapter by examining relatively easy ways of speeding up code using the **compiler** package and parallel processing, using multiple CPUs.

3.1 Top 5 tips for efficient programming

- Be careful never to grow vectors.
- Use `invisible` to return potentially useful information.
- Use factors when appropriate.
- Clever use of S3 objects can make code easier to understand.
- Byte compile packages for an easy performance boost.

3.2 General advice

Low level languages like C and Fortran demand more from the programmer. They force you to declare the type of every variable used and give you the burdensome responsibility of 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 type.

The wikipedia page on compiler opimisations gives a nice overview of standard optimisation techniques (https://en.wikipedia.org/wiki/Optimizing_compiler).

R users don't tend to worry about data types. This is advantageous in terms of creating concise code, but can result in R programs that are slow. While optimisations such as going parallel can double speed, poor code can easily run 100's of times slower, so it's important to understand the causes of slow code. These are covered in Burns (2011), which should be considered essential reading for any aspiring R programmers.

Ultimately calling an R function always ends up calling some underlying C/Fortran code. For example the base R function `runif` only contains a single line that consists of a call to `C_runif`.

```
function (n, min = 0, max = 1)
  .Call(C_runif, n, min, max)
```

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. For example, suppose `x` is a standard vector of length `n`. Then

```
x = x + 1
```

involves a single function call to the `+` function. Whereas the `for` loop

```
for(i in 1:n)
  x[i] = x[i] + 1
```

has

- `n` function calls to `+`;
- `n` function calls to the `[` function;
- `n` function calls to the `[<-` function (used in the assignment operation);
- A function call to `for` and to the `:` operator.

It isn't that the `for` loop is slow, rather it is because we have many more function calls. Each individual function call is quick, but the total combination is slow.

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

Exercise

Use the **microbenchmark** package to compare the vectorised construct `x = x + 1`, to the `for` loop version. Try varying the size of the input vector.

3.2.1 Memory allocation

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

You should also consider pre-allocating memory for data frames and lists. Never grow an object. A good rule of thumb is to compare your objects before and after a `for` loop; have they increased in length?

Let's consider three methods of creating a sequence of numbers. **Method 1** creates an empty vector and grows the object

```
method1 = function(n) {
  vec = NULL # Or vec = c()
  for(i in 1:n)
    vec = c(vec, i)
  vec
}
```

Method 2 creates an object of the final length and then changes the values in the object by subscripting:

```
method2 = function(n) {
  vec = numeric(n)
  for(i in 1:n)
    vec[i] = i
  vec
}
```

Method 3 directly creates the final object

```
method3 = function(n) 1:n
```

To compare the three methods we use the `microbenchmark` function from the previous chapter

```
microbenchmark(times = 100, unit="s", method1(n), method2(n), method3(n))
```

The table below shows the timing in seconds on my machine for these three methods for a selection of values of `n`. The relationships for varying `n` are all roughly linear on a log-log scale, but the timings between methods are drastically different. Notice that the timings are no longer trivial. When $n = 10^7$, method 1 takes around an hour whilst method 2 takes 2 seconds and method 3 is almost instantaneous. Remember the golden rule; access the underlying C/Fortran code as quickly as possible.

Table 3.1: Time in seconds to create sequences. When $n = 10^7$, method 1 takes around an hour while the other methods take less than 3 seconds.

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

3.2.2 Vectorised code

The vector is one of the key data types in R, with many functions offering a vectorised version. For example, the code

```
x = runif(n) + 1
```

performs two vectorised operations. First `runif` returns `n` random numbers. Second we add `1` to each element of the vector. In general it is a good idea to exploit vectorised functions. Consider this piece of R code that calculates the sum of $\log(x)$

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

Using `1:length(x)` can lead to hard-to-find bugs when `x` has length zero. Instead use `seq_along(x)` or `seq_len(length(x))`.

This code could easily be vectorised via

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

Writing code this way has a number of benefits.

- It's faster. When $n = 10^7$ the "R way" is about forty times faster.
- It's neater.
- It doesn't contain a bug when `x` is of length 0.

Exercises

Time the two methods for calculating the log sum. Try different values of n.

Example: Monte-Carlo integration

It's also important to make full use of R functions that use vectors. For example, suppose we wish to estimate the integral

$$\int_0^1 x^2 dx$$

using a Monte-Carlo method. Essentially, we throw darts at the curve and count the number of darts that fall below the curve (as in 3.1).

Monte Carlo Integration

1. Initialise: `hits = 0`
2. **for i in 1:N**
3. Generate two random numbers, U_1, U_2, between 0 and 1
4. If $U_2 < U_1^2$, then `hits = hits + 1`
5. **end for**
6. Area estimate = `hits/N`

Implementing this Monte-Carlo algorithm in R would typically lead to something like:

```
monte_carlo = function(N){
  hits = 0
  for(i in 1:N)  {
    u1 = runif(1); u2 = runif(1)
    if(u1^2 > u2)
      hits = hits + 1
```

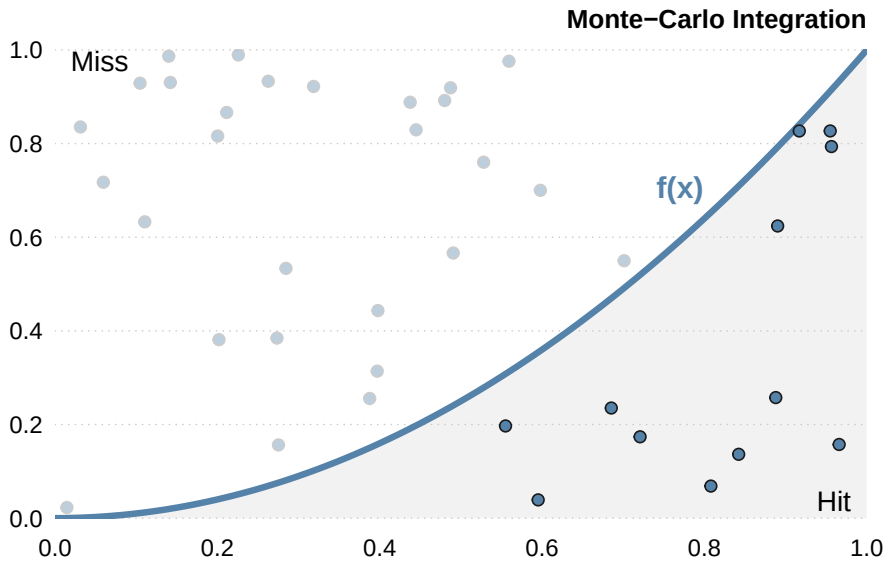


Figure 3.1: Example of Monte-Carlo integration. To estimate the area under the curve throw random points at the graph and count the number of points that lie under the curve.

```
  }
  return(hits/N)
}
```

In R this takes a few seconds

```
N = 500000
system.time(monte_carlo(N))
#>    user  system elapsed
#>   2.724   0.013   2.736
```

In contrast a more R-centric approach would be

```
monte_carlo_vec = function(N) sum(runif(N)^2 > runif(N))/N
```

The `monte_carlo_vec` function contains (at least) four aspects of vectorisation

- The `runif` function call is now fully vectorised;
- We raise entire vectors to a power via `^`;
- Comparisons using `>` are vectorised;
- Using `sum` is quicker than an equivalent for loop.

The function `monte_carlo_vec` is around 30 times faster than `monte_carlo`.

Exercise

Verify that `monte_carlo_vec` is faster than `monte_carlo`. How does this relate to the number of darts, i.e. the size of `N`, that is used

3.3 Communicating with the user

When we create a function we often want the function to give feedback on the current state. For example, are there missing arguments or has a numerical calculation failed. There are three main techniques of communicating with the user.

3.3.1 Fatal errors: `stop`

Fatal errors are raised by calling the `stop`, i.e. execution is terminated. When `stop` is called, there is no way for a function to continue. For instance, when we generate random numbers using `rnorm` the first argument is the sample size,`n`. If the number of observations to return less than 1, an error is raised.

Errors can be caught using `try` and `tryCatch`. For example,

```
# Suppress the error message
good = try(1 + 1, silent = TRUE)
bad = try(1 + "1", silent = TRUE)
```

When we inspect the objects, the variable `good` just contains the number `2`

```
good
#> [1] 2
```

However, the `bad` object is a character string with class `try-error` and a `condition` attribute that contains the error message

```
bad
#> [1] "Error in 1 + \"1\" : non-numeric argument to binary operator\n"
#> attr(,"class")
#> [1] "try-error"
#> attr(,"condition")
#> <simpleError in 1 + "1": non-numeric argument to binary operator>
```

We can use this information in a standard conditional statement

```
if(class(bad) == "try-error")
  # Do something
```

Further details on error handling, as well as some excellent advice on general debugging techniques, are given in (H. Wickham 2014a).

3.3.2 Warnings: `warning`

Warnings are generated using the `warning` function. When a warning is raised, it indicates potential problems. For example, `mean(NULL)` returns `NA` and also raises a warning. Warnings can be hidden using `suppressWarnings`:

```
suppressWarnings(mean(NULL))
#> [1] NA
```

In general, it is good practice to solve the underlying cause of the warning message, instead of evading the issue via `suppressWarning`.

3.3.3 Informative output: `message` and `cat`

To give informative output, use the `message` function. In my package for fitting powerlaws, I use messages to give the user an estimate of expected run time. Similar to warnings, messages can be suppressed with `suppressMessages`.

Another function used for printing messages is `cat`. In general `cat` should only be used in `print`/`show` methods, e.g. look at the function definition of the S3 print method for `difftime` objects, `getS3method("print", "difftime")`.

3.3.4 Example: Retrieving a web resource

In some cases it isn't clear what should be returned, a message, a `NA` value or an error. For example, suppose we wish to download data from a web site.

The function `GET` from the **httr** package can be used to download a webpage. When we attempt to download data, the function also obtains an HTML status code. If the page exists, the function will return text (with the correct status code), e.g.

```
GET("http://google.com/") # Status: 200
```

If the url is incorrect, e.g. a broken link or is a redirect, then we can interrogate the output and use the Status to determine what to do next.

```
GET("http://google.com/made_up") # Status: 404
```

By examining the error code, the user has complete flexibility. However if the web-page doesn't exist

```
GET("http://google1.com/")
```

or if we don't have an internet connection, then `GET` raises an error via `stop`. Instead of raising an error the author could have returned `NA` (the web-page isn't available). However for the typically use case, it's not clear how returning `NA` would be helpful.

The `install_github` function in the **devtools** package either installs the package or raises an error. When an error is raised, the function uses a combination of `message` and `stop` to indicate the source of an error, e.g.

```
devtools::install_github("bad/package")
#> Downloading GitHub repo bad/package@master
#> from URL https://api.github.com/repos/bad/package/zipball/master
#>  Error in curl::curl_fetch_memory(url, handle = handle) :
#>   Couldn't resolve host name
```

With the key idea being that if a package isn't able to be installed, future calculations will break; so it's optimal to raise an error as soon as possible.

Exercises

The `stop` function has an argument `call.` that indicates if the function call should be part of the error message. Create a function and experiment with this option.

3.3.5 Invisible returns

The `invisible` function allows you to return a temporarily invisible copy of an object. This is particularly useful for functions that return values which can be assigned, but are not printed when they are not assigned. For example suppose we have a function that plots the data and fits a straight line

```
regression_plot = function(x, y, ...) {
  # Plot and pass additional arguments to default plot method
  plot(x, y, ...)

  # Fit regression model
  model = lm(y ~ x)
  # Add line of best fit to the plot
  abline(model)
  invisible(model)
}
```

When the function is called, a scatter graph is plotted with the line of best fit, but the output is invisible. However when we assign the function to an object, i.e. `out = regression_plot(x, y)` the variable `out` contains the output of the `lm` call.

Another example is `hist`. Typically we don't want anything displayed in the console when we call the function

```
hist(x)
```

However if we assign the output to an object, `out = hist(x)`, the object `out` is actually a list containing, *inter alia*, information on the mid-points, breaks and counts.

3.4 Factors

Factors are much maligned objects. While at times they are awkward, they do have their uses. A factor is used to store categorical variables. This data type is unique to R (or at least not common among programming languages). Often categorical variables get stored as 1, 2, 3, 4, and 5, with associated documentation elsewhere that explains what each number means. This is clearly a pain. Alternatively we store the data as a character vector. While this is fine, the semantics are wrong because it doesn't convey that this is a categorical variable. It's not sensible to say that you should **always** or **never** use factors, since factors have both positive and negative features. Instead we need to examine each case individually. As a guide of when it's appropriate to use factors, consider the following examples.

3.4.1 Example: Months of the year

Suppose our data set relates to months of the year

```
m = c("January", "December", "March")
```

If we sort `m` in the usual way, `sort(m)`, we perform standard alpha-numeric ordering; placing `December` first. This is technically correct, but not that helpful. We can use factors to remedy this problem by specifying the admissible levels

```
# month.name contains the 12 months
fac_m = factor(m, levels=month.name)
sort(fac_m)
#> [1] January  March    December
#> 12 Levels: January February March April May June July August ... December
```

3.4.2 Example: Graphics

Factors can be used for ordering in graphics. For instance, suppose we have a data set where the variable `type`, takes one of three values, `small`, `medium` and `large`. Clearly there is an ordering. Using a standard `boxplot` call,

```
boxplot(y ~ type)
```

would create a boxplot where the x-axis was alphabetically ordered. By converting `type` into factor, we can easily specify the correct ordering.

```
boxplot(y ~ factor(type, levels = c("Small", "Medium", "Large")))
```

3.4.3 Example: Analysis of variance

Analysis of variance (ANOVA) is a type of statistical model that is used to determine differences among group means, while taken into account other factors. The function `aov` is used to fit standard analysis of variance models. Potential catastrophic bugs arise when a variable is numeric, but in reality is a categorical variable.

Consider the `npk` dataset on the growth of peas that comes with R. The column `block`, indicates the block (typically a nuisance parameter) effect. This column takes values 1 to 5, but has been carefully coded as a factor. Using the `aov` function to estimate the `block` effect, we get

```
aov(yield ~ block, npk)
#> Call:
#>    aov(formula = yield ~ block, data = npk)
#>
#> Terms:
#>                   block Residuals
#> Sum of Squares    343       533
#> Deg. of Freedom     5        18
#>
#> Residual standard error: 5.44
#> Estimated effects may be unbalanced
```

If we repeat the analysis, but change `block` to a numeric data type we get different (and incorrect) results

```
aov(yield ~ as.numeric(block), npk)
#> Call:
#>    aov(formula = yield ~ as.numeric(block), data = npk)
#>
#> Terms:
#>                   as.numeric(block) Residuals
#> Sum of Squares                   22       854
#> Deg. of Freedom                   1        22
```

```
#>
#> Residual standard error: 6.23
#> Estimated effects may be unbalanced
```

When we pass a numeric variable, the `aov` function is interpreting this variable as continuous, and fits a regression line.

3.4.4 Example: data input

Most users interact with factors via the `read.csv` function where character columns are automatically converted to factors. This feature can be irritating if our data is messy and we want to clean and recode variables. Typically when reading in data via `read.csv`, we use the `stringsAsFactors=FALSE` argument.

Although this argument can add in the global `options()` list and placed in the `.Rprofile`, this leads to non-portable code, so should be avoided.

3.4.5 Example: Factors are not character vectors

Although factors look similar to character vectors, they are actually integers. This leads to initially surprising behaviour

```
x = 4:6
c(x)
#> [1] 4 5 6
c(factor(x))
#> [1] 1 2 3
```

In this case the `c` function is using the underlying integer representation of the factor.

Overall factors are useful, but can lead to unwanted side-effects if we are not careful. Used at the right time and place, factors can lead to simpler code.

Exercise

Factors are slightly more space efficient than characters. Create a character vector and corresponding factor and use `pryr::object_size` to calculate the space needed for each object.

3.5 S3 objects

R has three built-in object oriented (OO) systems. These systems differ in how classes and methods are defined. The easiest and oldest system is the S3 system. S3 refers to the third version of S. The syntax of R is largely based on this version of S. In R there has never been S1 and S2 classes. The other two OO frameworks are S4 classes (used mainly in bioconductor packages) and reference classes.

There are also packages that also provide additional OO frameworks, such as **proto**, **R6** and **R.oo**. If you are new to OO in R, then S3 is the place to start.

In this section we will just discuss the S3 system since that is the most popular. The S3 system implements a generic-function object oriented (OO) system. This type of OO is different to the message-passing style of Java and C++. In a message-passing framework, messages/methods are sent to objects and the object determines which function to call, e.g. `normal.rand(1)`. The S3 class system is different. In S3, the *generic* function decides which method to call - it would have the form `rand(normal, 1)`. By using an OO framework, we avoid an explosion of exposed functions, such as, `rand_normal`, `rand_uniform`, `rand_poisson` and instead have a single function call `rand` that passes the object to the correct function.

The S3 system is based on the class of an object. In S3, a class is just an attribute which can be determined with the `class` function.

```
class(USArrests)
#> [1] "data.frame"
```

The S3 system can be used to great effect. When we pass an object to a *generic* function, the function first examines the class of the object, and then dispatches the object to another method. For example, the `summary` function is a S3 generic function

```
functionBody("summary")
#> UseMethod("summary")
```

Note that the only operational line is `UseMethod("summary")`. This handles the method dispatch based on the object's class. So when `summary(USArrests)` is executed, the generic `summary` function passes `USArrests` to the function `summary.data.frame`. If the function `summary.data.frame` does not exist, then `summary.default` is called (if it exists). If neither function exist, an error is raised.

This simple message passage mechanism enables us to quickly create our own functions. Consider the distance object:

```
dist_usa = dist(USArrests)
```

The `dist_usa` object has class `dist`. To visualise the distances, we create an `image` method. First we'll check if the existing `image` function is generic, via

```
# In R3.3, a new function isS3stdGeneric is going to be introduced.
functionBody("image")
#> UseMethod("image")
```

Since `image` is already a generic method, we just have to create a specific `dist` method

```
image.dist = function(x, ...) {
  x_mat = as.matrix(x)
  image(x_mat, main=attr(x, "method"), ...)
}
```

The `...` argument allows us to pass arguments to the main `image` method, such as `axes` (see figure 3.2.

Many S3 methods work in the same way as the simple `image.dist` function created above: the object is manipulated into a standard format, then passed to the standard method. Creating S3 methods for standard functions such as `summary`, `mean`, and `plot` provides a nice uniform interface to a wide variety of data types.

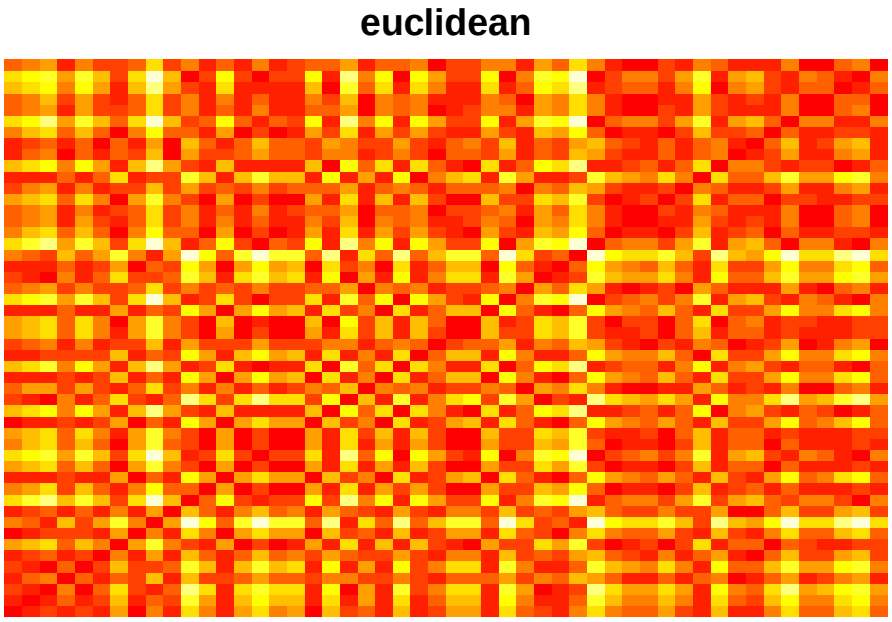


Figure 3.2: S3 image method for data of class 'dist'.

Exercises

A data frame is just an R list, with class `data.frame`.

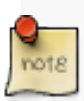
1. Use a combination of `unclass` and `str` on a data frame to confirm that it is indeed a list.
2. Use the function `length` on a data frame. What is return? Why?

3.6 The apply family

The apply functions can be an alternative to writing for loops. The general idea is to apply (or map) a function to each element of an object. For example, you can apply a function to each row or column of a matrix. A list of available functions is given in 3.2, with a short description. In general, the all apply functions have similar properties:

- Each function takes at least two arguments: an object and another function. The function is passed as an argument.
- Every apply function has the dots, `...`, argument that is used to pass on arguments to the function that is given as an argument.

Using apply functions when possible, can lead to shorter, more succinct idiomatic R code. In this section, we will cover the three main functions, `apply`, `lapply`, and `sapply`. Since the apply functions are covered in most R textbooks, we just give a brief introduction to the topic and provide pointers to other resources at the end of this section.



Most people rarely use the other apply functions. For example, I have only used `eapply` once. Students in my class uploaded R scipts. Using `source`, I was able to read in the scripts to a separate

Table 3.2: The apply family of functions from base R

Function	Description
‘apply‘	Apply functions over array margins
‘by‘	Apply a function to a data frame split by factors
‘eapply‘	Apply a function over values in an environment
‘lapply‘	Apply a function over a list or vector
‘mapply‘	Apply a function to multiple list or vector arguments
‘rapply‘	Recursively apply a function to a list
‘tapply‘	Apply a function over a ragged array

environment. I then applied a marking scheme to each environment using `eapply`. Using separate environments, avoided object name clashes.

The `apply` function is used to apply a function to the each row or column of a matrix. In many data science problems, this is a common task. For example, to calculate the standard deviation of the row we have

```
data("ex_mat", package="efficient")
# MARGIN=1: corresponds to rows
row_sd = apply(ex_mat, 1, sd)
```

The first argument of `apply` is the object of interest. The second argument is the `MARGIN`. This is a vector giving the subscripts which the function (the third argument) will be applied over. When the object is a matrix, a margin of `1` indicates rows and `2` indicates columns. So to calculate the column standard deviations, the second argument is changed to `2`

```
col_med = apply(ex_mat, 2, sd)
```

Additional arguments can be passed to the function that is to be applied to the data. For example, to pass the `na.rm` argument to the `sd` function, we have

```
row_sd = apply(ex_mat, 1, sd, na.rm=TRUE)
```

The `apply` function also works on higher dimensional arrays; a one dimensional array is a vector, a two dimensional array is a matrix.

The `lapply` function is similar to `apply`; the main differences are the input types are vectors or lists and the return type is a list. Essentially, we apply a function to each element of a list or vector. The functions `sapply` and `vapply` are similar to `lapply`, but the return type is not necessary a list.

3.6.1 Example: the movies data set

The internet movie database is a website that collects movie data supplied by studios and fans. It is one of the largest movies databases on the web and is maintain by Amazon. The `ggplot2movies` package contains about sixty thousand movies stored as a data frame

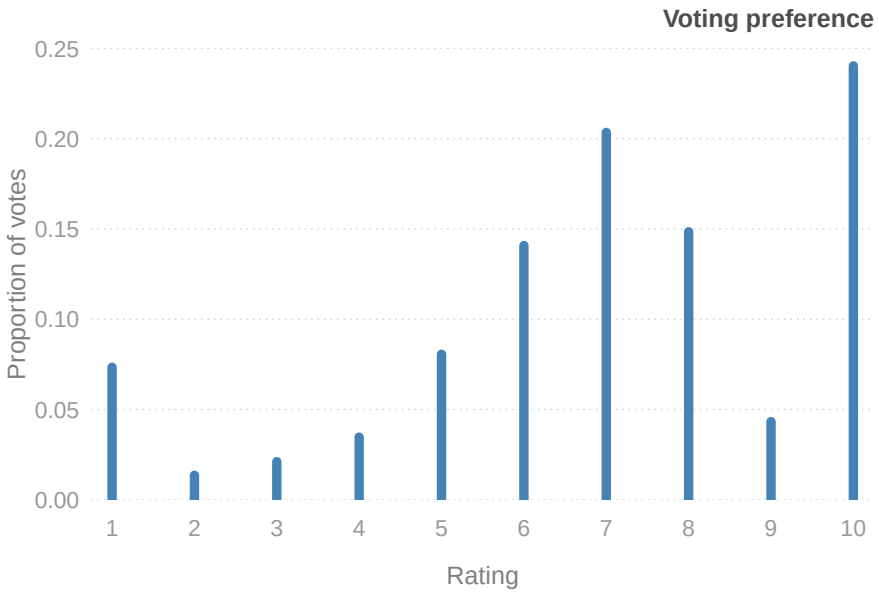


Figure 3.3: Movie voting preferences.

```
data(movies, package="ggplot2movies")
```

Movies are rated between 1 and 10 by fans. Columns 7 to 16 of the `movies` data set gives the percentage of voters for a particular rating.

```
ratings = movies[, 7:16]
```

For example, 4.5% of voters, rated the first movie a rating of 1

```
ratings[1, ]
#> # A tibble: 1 x 10
#>      r1    r2    r3    r4    r5    r6    r7    r8    r9   r10
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1   4.5   4.5   4.5   4.5  14.5  24.5  24.5  14.5   4.5   4.5
```

We can use the `apply` function to investigate voting patterns. The function `nnet::which.is.max` finds the maximum position in a vector, but breaks ties at random; `which.max` just returns the first value. Using `apply`, we can easily determine the most popular rating for each movie and plot the results

```
popular = apply(ratings, 1, nnet::which.is.max)
plot(table(popular))
```

Figure @(fig:3-3) highlights that voting patterns are clearly not uniform between 1 and 10. The most popular vote is the highest rating, 10. Clearly if you went to the trouble of voting for a movie, it was either very good, or very bad (there is also a peak at 1). Rating a movie 7 is also a popular choice (search the web for "most popular number" and 7 dominates the rankings.)

3.6.2 Type consistency

When programming it is helpful if the return value from a function always takes the same form. Unfortunately, not all of base R functions follow this idiom. For example the functions `sapply` and `[.data.frame` aren't

type consistent

```
two_cols = data.frame(x = 1:5, y = letters[1:5])
zero_cols = data.frame()
sapply(two_cols, class)  # a character vector
sapply(zero_cols, class) # a list
two_cols[, 1:2]          # a data.frame
two_cols[, 1]            # an integer vector
```

This can cause unexpected problems. The functions `lapply` and `vapply` are type consistent. Likewise `dplyr::select` and `dplyr:filter`. The **purrr** package has some type consistent alternatives to base R functions. For example, `map_dbl` etc. to replace `Map` and `flatten_df` to replace `unlist`.

Other resources

Almost every R book has a section on the apply function. Below, we've given the resources we feel are most helpful.

- Each function has a number of examples in the associated help page. You can directly access the examples using the `example` function, e.g. to run the `apply` examples, use `example("apply")`.
- There is a very detailed Stackoverflow answer description when, where and how to use each of the functions.
- In a similar vein, Neil Saunders has a nice blog post giving an overview of the functions.
- The apply functions are an example of functional programming. Chapter 16 of *R for data Science* describes the interplay between loops and functional programming in more detail (Grolemund and Wickham, n.d.), while H. Wickham (2014a) gives a more in-depth description of the topic.

Exercises

1. Rewrite the `sapply` function calls above using `vapply` to ensure type consistency.
2. How would you make subsetting data frames with `[` type consistent? Hint: look at the `drop` argument.

3.7 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 with replacing `log(x)` in multiple function calls with the object `log_x` that is defined once and reused. This small saving in time quickly multiplies when the cached variable is used inside a `for` loop.

A more advanced form of caching is to use the **memoise** package. If a function is called multiple times with the same input, it may be possible to speed things up by keeping a cache of known answers that it can retrieve. The **memoise** package allows us 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. For example, suppose we are developing a shiny (an interactive graphic) application where the user can fit regression line to data. The user can remove points and refit the line. An example function would be

```
# Argument indicates row to remove
plot_mpg = function(row_to_remove) {
  data(mpg, package="ggplot2")
  mpg = mpg[-row_to_remove,]
  plot(mpg$cty, mpg$hwy)
  lines(lowess(mpg$cty, mpg$hwy), col=2)
}
```

We can use **memoise** speed up by caching results. A quick benchmark

```
library("memoise")
m_plot_mpg = memoise(plot_mpg)
microbenchmark(times=10, unit="ms", m_plot_mpg(10), plot_mpg(10))
#> Unit: milliseconds
#>            expr   min    lq  mean median    uq   max neval cld
#>  m_plot_mpg(10)  0.04 4e-02  0.07  8e-02 8e-02   0.1    10  a
#>    plot_mpg(10) 40.20 1e+02 95.52  1e+02 1e+02 107.1    10   b
```

suggests that we can obtain a 100-fold speed-up.

Exercise

Construct a box plot of timings for the standard plotting function and the memoised version.

3.7.1 Function closures

The following section is meant to provide an introduction to function closures with example use cases. See [@Wickham2014] for a detailed introduction.

More advanced caching is available using *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 we use the term function closure 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 was the function created. We can determine the enclosing environment using `environment`:

```
environment(plot_mpg)
#> <environment: R_GlobalEnv>
```

The `plot_mpg` function's enclosing environment is the `.GlobalEnv`. This is important for variable scope, i.e. where should be look for a particular object. Consider the function `f`

```
f = function() {
  x = 5
  function() x
}
```

When we call the function `f`, the object returned is a function. While the enclosing environment of `f` is `.GlobalEnv`, the enclosing environment of the *returned* function is something different

```
g = f()
environment(g)
#> <environment: 0x37133d0>
```

When we call this new function `g`,

```
x = 10
g()
#> [1] 5
```

The value returned is obtained from `environment(g)` not from the `.GlobalEnv`. This persistent environment allows us to cache variables between function calls.

The operator `<<-` makes R search through the parent environments for an existing defintion. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment.

The `simple_counter` function exploits this feature to enable variable caching

```
simple_counter = function() {
  no = 0
  function() {
    no <<- no + 1
    no
  }
}
```

When we call the `simple_counter` function, we retain object values between function calls

```
sc = simple_counter()
sc()
#> [1] 1
sc()
#> [1] 2
```

The key points of the `simple_counter` function are

- The `simple_counter` function returns a function;
- The enclosing environment of `sc` is not `.GlobalEnv`, instead it's the binding environment of `sc`;
- The function `sc` has an environment that can be used to store/cache values;
- The operator `<<-` is used to alter the object `no` in the `sc` environment.

Example

We can exploit function closures to simplify our code. Suppose we wished to simulate a games of Snakes and Ladders. We have function that handles the logic of landing on a snake

```
check_snake = function(square) {
   switch(as.character(square),
       '16'=6,  '49'=12, '47'=26, '56'=48, '62'=19,
       '64'=60, '87'=24, '93'=73, '96'=76, '98'=78,
       square)
}
```

If we then wanted to determine how often we landed on a Snake, we could use a function closure to easily keep track of the counter.

```
check_snake = function() {
  no_of_snakes = 0
  function(square) {
    new_square = switch(as.character(square),
       '16'=6,  '49'=12, '47'=26, '56'=48, '62'=19,
       '64'=60, '87'=24, '93'=73, '96'=76, '98'=78,
       square)
    no_of_snakes <<- no_of_snakes + (new_square != square)
    new_square
  }
}
```

Keeping the variable `no_of_snakes` attached to the `check_snake` function, enables us to have cleaner code.

Exercise

The following function implements a stop-watch function

```
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()
```

It contains two functions. One function for starting the timer

```
watch$start()
```

the other for stopping the timer

```
watch$stop()
```

Many stop-watches have the ability to measure not only your overall time but also you individual laps. Add a `lap` function to the `stop_watch` function that will record individual times, while still keeping track of the overall time.

A related idea to function closures, is non-standard evaluation (NSE), or programming on the language. NSE crops up all the time in R. For example, when we execute, `plot(height, weight)` R automatically labels the x- and y-axis of the plot with `height` and `weight`. This is powerful concept that enables us to simplify code. More detail is given in the "Non-standard evaluation" of [@Wickham2014].

3.8 The byte compiler

The **compiler** package, written by R Core member Luke Tierney has been part of R since version 2.13.0. The **compiler** package allows R functions to be compiled, resulting in a byte code version that may run faster[1]. The compilation process eliminates a number of costly operations the interpreter has to perform, such as variable lookup.

Since R 2.14.0, all of the standard functions and packages in base R are pre-compiled into byte-code. This is illustrated by the base function `mean`:

```
getFunction("mean")
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x351ab80>
#> <environment: namespace:base>
```

The third line contains the `bytecode` of the function. This means that the **compiler** package has translated the R function into another language that can be interpreted by a very fast interpreter. Amazingly the **compiler** package is almost entirely pure R, with just a few C support routines.

3.8.1 Example: the mean function

The **compiler** package comes with R, so we just need to load the package in the usual way

```
library("compiler")
```

Next we create an inefficient function for calculating the mean. This function takes in a vector, calculates the length and then updates the `m` variable.

```
mean_r = function(x) {
  m = 0
  n = length(x)
  for(i in seq_len(n))
    m = m + x[i]/n
  m
}
```

This is clearly a bad function and we should just `mean` function, but it's a useful comparison. Compiling the function is straightforward

[1] The authors have yet to find a situation where byte compiled code runs significantly slower.

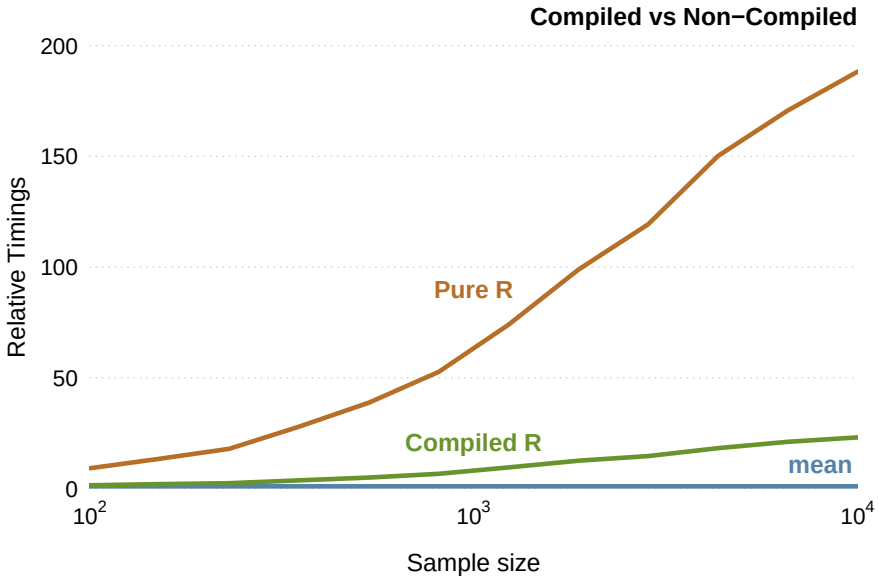


Figure 3.4: Comparsion of mean functions.

```
cmp_mean_r = cmpfun(mean_r)
```

Then we use the `microbenchmark` function to compare the three variants

```
# Generate some data
x = rnorm(1000)
microbenchmark(times=10, unit="ms", # milliseconds
          mean_r(x), cmp_mean_r(x), mean(x))
#> Unit: milliseconds
#>           expr   min    lq  mean median    uq  max neval cld
#>      mean_r(x) 0.358 0.361 0.370  0.363 0.367 0.43    10   c
#>  cmp_mean_r(x) 0.050 0.051 0.052  0.051 0.051 0.07    10  b
#>       mean(x) 0.005 0.005 0.008  0.007 0.008 0.03    10 a
```

The compiled function is around seven times faster than the uncompiled function. Of course the native `mean` function is faster, but compiling does make a significant difference (figure 3.4).

3.8.2 Compiling code

There are a number of ways to compile code. The easiest is to compile individual functions using `cmpfun`, but this obviously doesn't scale. If you create a package, you can automatically compile the package on installation by adding

```
ByteCompile: true
```

to the `DESCRIPTION` file. Most R packages installed using `install.packages` are not compiled. We can enable (or force) packages to be compiled by starting R with the environment variable `R_COMPILE_PKGS` set to a positive integer value and specify that we install the package from `source`, i.e.

```
## Windows users will need Rtools
install.packages("ggplot2", type="source")
```

A final option to use just-in-time (JIT) compilation. The `enableJIT` function disables JIT compilation if the argument is `0`. Arguments `1`, `2`, or `3` implement different levels of optimisation. JIT can also be enabled by setting the environment variable `R_ENABLE_JIT`, to one of these values.

I always set the compile level to the maximum value of 3.

Chapter 4

Efficient workflow

Efficient programming is an important and skill for generating the correct result, on time. Yet coding is only one part of a wider skillset needed for successful outcomes for projects involving R programming. Unless your project is to write generic R code (i.e. unless you are on the R Core Team), the project will probably transcend the confines of the R world: it must engage with a whole range of other factors. In this context we define 'workflow' as the sum of practices, habits and systems that enable productivity.[1] To some extent workflow is about personal preferences. Everyone's mind works differently so the most appropriate workflow varies from person to person and from one project to the next. Project management practices will also vary depending the scale and type of the project: it's a big topic but can usefully be condensed in 5 top tips.

4.1 Top 5 tips for efficient workflow

- 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.
- Make a plan. The size and nature will depend on the project but time-lines, resources and 'chunking' the work will make you more effective when you start.
- 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.
- 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.
- Make your entire workflow as reproducible as possible. **knitr** can help with this in the phase of documentation.

4.2 A project planning typology

Appropriate project management structures and workflow depend on the *type* of project you are undertaking. The typology below demonstrate the links between project type and project management requirements.[2]

- *Data analysis.* Here you are trying to explore datasets to discover something interesting/answer some questions. The emphasis is on speed of manipulating your data to generate interest results. Formality

[1] The Oxford Dictionary's definition of workflow is similar, with a more industrial feel: "The sequence of industrial, administrative, or other processes through which a piece of work passes from initiation to completion."

[2] Thanks to Richard Cotton for suggesting this typology.

is less important in this type of project. Sometimes this analysis project may only be part of a larger project (the data may have to be created in a lab, for example). How the data analysts interact with the rest of the team may be as important for the project's success as how they interact with each other.
- *Package creation.* Here you want to create code that can be reused across projects, possibly by people whose use case you don't know (if you make it publicly available). The emphasis in this case will be on clarity of user interface and documentation, meaning style and code review are important. Robustness and testing are important in this type of project too.
- *Reporting and publishing.* Here you are writing a report or journal paper or book. The level of formality varies depending upon the audience, but you have additional worries like how much code it takes to arrive at the conclusions, and how much output does the code create.
- *Software applications.* This could range from a simple Shiny app to R being embedded in the server of a much larger piece of software. Either way, since there is limited opportunity for human interaction, the emphasis is on robust code and gracefully dealing with failure.

Based on these observations we recommend thinking about the type of workflow, file structure and project management system suits your projects best. Sometimes it's best not to be prescriptive so we recommend trying different working practices to discover which works best, time permitting.[3]

There are, however, concrete steps that can be taken to improve workflow in most projects that involve R programming. Learning them will, in the long-run, improve productivity and reproducibility. With these motivations in mind, the purpose of this chapter is simple: to highlight some key ingredients of an efficient R workflow. It builds on the concept of an R/RStudio *project*, introduced in Chapter 2, and is ordered chronologically throughout the stages involved in a typical project's lifespan, from its inception to publication:

- Project planning. This should happen before any code has been written, to avoid time wasted using a mistaken analysis strategy. Project management is the art of making project plans happen.
- Package selection. After planning your project you should identify which packages are most suitable to get the work done quickly and effectively. With rapid increases in the number and performance of packages. `*_join` from **dplyr**, for example, is often more appropriate than `merge`, as we'll see in 6. It is therefore more important than ever to consider the range of options at the outset.
- Publication. This final stage is relevant if you want your R code to be useful for others in the long term. To this end Section 4.5 touches on documentation using knitr and the much stricter approach to code publication of package development.

4.3 Project planning and management

Good programmers working on a complex project will rarely just start typing code. Instead, they will plan the steps needed to complete the task as efficiently as possible: "smart preparation minimizes work" (Berkun 2005). Although search engines are useful for identifying the appropriate strategy, trail-and-error approaches (for example typing code at random and Googling the inevitable error messages) are usually highly *inefficient*.

Strategic thinking 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. So detrimental and ubiquitous is this phenomenon in software development that a term has been coined to describe it: *technical debt*. This has been defined as "not quite right code which we postpone making it right" (Kruchten, Nord, and Ozkaya 2012). Dozens of academic papers have been written on the subject but, from the perspective of *beginning* a project (i.e. in the planning stage, where we are now), all you need to know is that it is absolutely vital to make sensible decisions at the outset. If you do not, your project may be doomed to failure of incessant rounds of refactoring (we return to the topic of refactoring in Chapter 9).

[3]The importance of workflow has not gone unnoticed by the R community and there are a number of different suggestions to boost R productivity. Rob Hyndman, for example, advocates the strategy of using four self-contained scripts to break up R work into manageable chunks: `load.R`, `clean.R`, `func.R` and `do.R`.

To minimise technical debt at the outset, the best place to start may be with a pen and paper and an open mind. Sketching out your ideas and deciding precisely what you want to do, free from the constraints of a particular piece of technology, can be rewarding exercise before you begin. Project planning and 'visioning' can be a creative process not always well-suited to the linear logic of computing, despite recent advances in project management software, some of which are outlined in the bullet points below.

Scale can loosely be defined as the number of people working on a project. It should be considered at the outset because the importance of project management increases exponentially with the number of people involved. Project management may be trivial for a small project but if you expect it to grow, implementing a structured workflow early could avoid problems later. On small projects consisting of a 'one off' script, project management may be a distracting waste of time. Large projects involving dozens of people, on the other hand, require much effort dedicated to project management: regular meetings, division of labour and a scalable project management system to track progress, issues and priorities will inevitably consume a large proportion of the project's time. Fortunately a multitude of dedicated project management systems have been developed to cater for projects across a range of scales. These include, in rough ascending order of scale and complexity:

- the interactive code sharing site GitHub, which is described in more detail in Chapter 9)
- ZenHub, a browser plugin that is "the first and only project management suite that works natively within GitHub"
- web-based and easy-to-use tools such as Trello
- Dedicated desktop project management software such as ProjectLibre and GanttProject
- fully featured, enterprise scale open source project management systems such as OpenProject and redmine.

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?

Minutes spent thinking through such issues before writing a single line can save hours in the future. This is emphasised in books such as Berkun (2005) and PMBoK (2000) and useful online resources such those by teamgantt.com and lasa.org.uk, which focus exclusively on project planning. This section condenses some of the most important lessons from this literature in the context of typical R projects (i.e. which involve data analysis, modelling and visualisation).

4.3.1 'Chunking' your work

Once a project overview has been devised and stored, in mind (for small projects, if you trust that as storage medium!) or written, a plan with a time-line can be drawn-up. The up-to-date visualisation 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. Setting start dates and deadlines for each task will help prioritise the work and ensure you are on track. Breaking a large project into smaller chunks is highly recommended, making huge, complex tasks more achievable and modular PMBoK (2000). 'Chunking' the work will also make collaboration easier, as we shall see in Chapter 5.

The tasks that a project should be split into will depend the nature of the work and the phases illustrated in Figure 4.1 represent a rough starting point, not a template and the 'programming' phase will usually need to be split into at least 'data tidying', 'processing', and 'visualisation'.

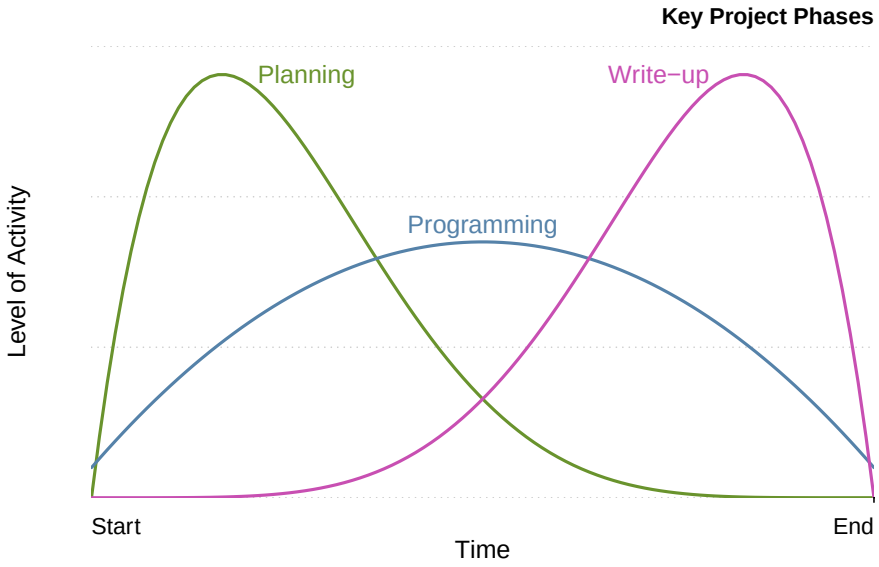


Figure 4.1: Schematic illustrations of key project phases and levels of activity over time, based on @PM-BoK2000.

4.3.2 Making your workflow SMART

A more rigorous (but potentially onerous) way to project plan is to divide the work into a series of objectives and tracking their progress throughout the project's duration. One way to check if an objective is appropriate for action and review is by using the SMART criteria.

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

If the answer to each of these questions is 'yes', the task is likely to be suitable to include in the project's plan. Note that this does not mean all project plans need to be uniform. A project plan can take many forms, including a short document, a Gantt chart (see Figure 4.2 or simply a clear vision of the project's steps in mind.

4.3.3 Visualising plans with R

Various R packages can help visualise the project plan. While these are useful, they cannot compete with the dedicated project management software outlined at the outset of this section. However, if you are working on relatively simple project, it is useful to know that R can help represent and keep track of your work. Packages for plotting project progress include:[4]

- the **plan** package, which provides basic tools to create burndown charts (which concisely show whether a project is on-time or not) and Gantt charts.
- **plotrix**, a general purpose plotting package, provides basic Gantt chart plotting functionality. Enter `example(gantt.chart)` for details.

[4]For a more comprehensive discussion of Gantt charts in R, please refer to stackoverflow.com/questions/3550341.

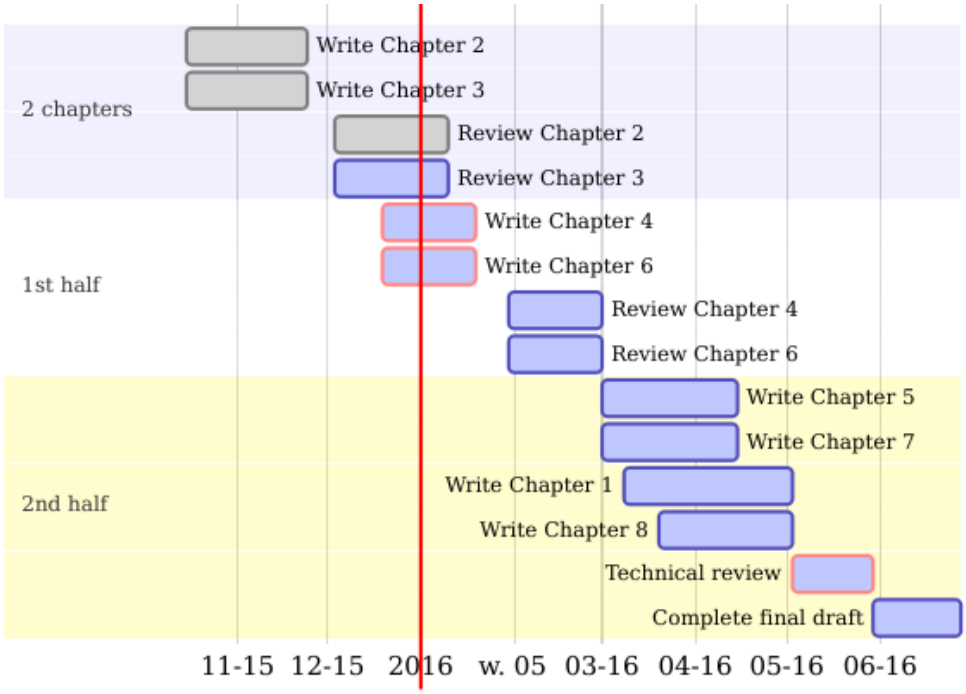


Figure 4.2: A Gantt chart created using **DiagrammeR** illustrating the steps needed to complete this book at an early stage of its development.

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

The small example below (which provides the basis for creating charts like Figure 4.2 illustrates how **DiagrammeR** can take simple text inputs to create informative up-to-date Gantt charts. Such charts can greatly help with the planning and task management of long and complex R projects, as long as they do not take away valuable programming time from core project objectives. {#DiagrammeR}

```
library("DiagrammeR")
# Define the Gantt chart and plot the result (not shown)
mermaid("gantt
        Section Initiation
        Planning           :a1, 2016-01-01, 10d
        Data processing    :after a1  , 30d")
```

In the above code `gantt` defines the subsequent data layout. `Section` refers to the project's section (useful for large projects, with milestones) and each new line refers to a discrete task. `Planning`, for example, has the code `a`, begins on the first day of 2016 and lasts for 10 days. See knsv.github.io/mermaid/gantt.html for more detailed documentation.

Exercises

1. What are the three most important work 'chunks' of your current R project?
2. What is the meaning of 'SMART' objectives (see Making your workflow SMART).
3. Run the code chunk at the end of this section to see the output.

4. Bonus exercise: modify this code to create a basic Gantt chart of an R project you are working on.

4.4 Package selection

A good example of the importance of prior planning to minimise effort and reduce technical debt is package selection. An inefficient, poorly supported or simply outdated package can waste hours. When a more appropriate alternative is available this waste can be prevented by prior planning. 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. Furthermore, because R packages are generally (but not always) written by competent programmers and subject to user feedback, they may work faster and more effectively that the hastily prepared code you may have written. All R code is open source and potentially subject to peer review. A prerequisite of publishing an R package is that developer contact details must be provided, and many packages provide a site for issue tracking. Furthermore, R packages can increase programmer productivity by dramatically reducing the amount of code they need to write because all the code is *packaged* in functions behind the scenes.

Let's take an example. Imagine for a project you would like to find the distance between sets of points (origins, `o` and destinations, `d`) on the Earth's surface. Background reading shows that a good approximation of 'great circle' distance, which accounts for the curvature of the Earth, can be made by using the Haversine formula, which you duly implement, involving much trial and error:

```
# Function to convert degrees to radians
deg2rad = function(deg) return(deg*pi/180)

# Create origins and destinations
o = c(lon = -1.55, lat = 53.80)
d = c(lon = -1.61, lat = 54.98)

# Convert to radians
o_rad = deg2rad(o)
d_rad = deg2rad(d)

# Find difference in degrees
delta_lon = (o_rad[1] - d_rad[1])
delta_lat = (o_rad[2] - d_rad[2])

# Calculate distance with Haversine formula
a = sin(delta_lat / 2)^2 + cos(o_rad[2]) * cos(d_rad[2]) * sin(delta_lon / 2)^2
c = 2 * asin(min(1, sqrt(a)))
(d_hav1 = 6371 * c) # multiply by Earth's diameter
#> [1] 131
```

This method works but it takes time to write, test and debug. Much better to package it up into a function. Or even better, use a function that someone else has written and put in a package:

```
# Find great circle distance with geosphere
(d_hav2 = geosphere::distHaversine(o, d))
#> [1] 131415
```

The difference between the hard-coded method and the package method is striking. One is 7 lines of tricky R code involving many subetting stages and small, similar functions (e.g. `sin` and `asin`) which are easy to confuse. The other is one line of simple code. The package method using **geosphere** took perhaps 100^{th} of the time *and* gave a more accurate result (because it uses a more accurate estimate of the diameter of the Earth). This means that a couple of minutes searching for a package to estimate great circle distances would have been time well spent at the outset of this project. But how do you search for packages?

4.4.1 Searching for R packages

Building on the example above, how can on find out if there is a package to solve your particular problem? The first stage is to guess: if it is a common problem, someone has probably tried to solve it. The second stage is to search. A simple Google query, `haversine formula R`, returned a link to the **geosphere** package in the second result (a hardcoded implementation was first).

Beyond Google, there are also several sites for searching for packages and functions. rdocumentation.org provides a multi-field search environmnet to pinpoint the function or package you need. Amazingly, the search for `haversine` in the Description field yielded 10 results from 8 packages: R has at least *8* implementations of the Haversine formula! This shows the importance of careful package selection as there are often many packages that do the same job, as we see in the next section. There is also a way to find the function from within R, with `RSiteSearch()`, which opens a url in your browser linking to a number of functions (40) and vignettes (2) that mention the text string:

```
# Search CRAN for mentions of haversine
RSiteSearch("haversine")
```

4.4.2 How to select a package

Due to the conservative nature of base R development, which rightly prioritises stability over innovation, much of the innovation and performance gains in the 'R ecosystem' has occurred in recent years in the packages. The increased ease of package development (Wickham 2015) and interfacing with other languages (e.g. Eddelbuettel et al. 2011) has accelerated their number, quality and efficiency. An additional factor has been the growth in collaboration and peer review in package development, driven by code-sharing websites such as GitHub and online communities such as ROpenSci for peer reviewing code.

Performance, stability and ease of use should be high on the priority list when choosing which package to use. Another more subtle factor is that some packages work better together than others. The 'R package ecosystem' is composed of interrelated package. Knowing something of these inter-dependencies can help select a 'package suite' when the project demands a number of diverse yet interrelated programming tasks. The 'tidyverse', for example, contains many interrelated packages that work well together, such as **readr**, **tidyr**, and **dplyr**.[5] These may be used together to read-in, tidy and then process the data, as outlined in the subsequent sections.

There is no 'hard and fast' rule about which package you should use and new packages are emerging all the time. The ultimate test will be empirical evidence: does it get the job done on your data? However, the following criteria should provide a good indication of whether a package is worth an investment of your precious time, or even installing on your computer:

- **Is it mature?** The more time a package is available, the more time it will have for obvious bugs to be ironed out. The age of a package on CRAN can be seen from its Archive page on CRAN. We can see from cran.r-project.org/src/contrib/Archive/ggplot2/, for example, that **ggplot2** was first released on the 10^{th} June 2007 and that it has had 28 releases. The most recent of these at the time of writing was

[5] An excellent overview of the 'tidyverse', formerly now as the 'hadleyverse' and its benefits is available from barryrowlingson.github.io/hadleyverse.

ggplot2 2.0.0: reaching 1 or 2 in the first digit of package versions is usually an indication from the package author that the package has reached a high level of stability.

- **Is it actively developed?** It is a good sign if packages are frequently updated. A frequently updated package will have its latest version 'published' recently on CRAN. The CRAN package page for **ggplot2**, for example, said `Published: 2015-12-18`, less than a month old at the time of writing.
- **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. Using a poorly documented package can be inefficient due to the hours spent trying to work out how to use it! To check if the package is well documented, look at the help pages associated with its key functions (e.g. `?ggplot`), try the examples (e.g. `example(ggplot)`) and search for package vignettes (e.g. `vignette(package = "ggplot2")`).
- **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 when typed into a generic search engine such as Google's. More specific (and potentially useful) indications of use will narrow down the search to particular users. A package widely used by the programming community will likely visible GitHub. At the time of writing a search for **ggplot2** on GitHub yielded over 400 repositories and almost 200,000 matches in committed code! Likewise, a package that has been adopted for use in academia will tend to be mentioned in Google Scholar (again, **ggplot2** scores extremely well in this measure, with over 5000 hits).

An article in simplystats discusses this issue with reference to the proliferation of GitHub packages (those that are not available on CRAN). In this context well-regarded and experienced package creators and 'indirect data' such as amount of GitHub activity are also highlighted as reasons to trust a package.

The websites of MRAN and METACRAN can help the package selection process by providing further information on each package uploaded to CRAN. METACRAN, for example, provides metadata about R packages via a simple API and the provision of 'badges' to show how many downloads a particular package has per month. Returning to the Haversine example above, we could find out how many times two packages that implement the formula are downloaded each month with the following urls:

- `http://cranlogs.r-pkg.org/badges/last-month/geosphere`, downloads of **geosphere**:

- `http://cranlogs.r-pkg.org/badges/last-month/geoPlot`, downloads of **geoPlot**:

It is clear from the results reported above that **geosphere** is by far the more popular package, so is a sensible and mature choice for dealing with distances on the Earth's surface.

4.5 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) discussed in Chapter 3 will make your publications faster to write, easier to maintain and more useful to others.

Instead of attempting a comprehensive treatment of the topic we will touch briefly on a couple of ways of documenting your work in R: dynamic reports and R packages. There is a wealth of material on each of these online. A wealth of online resources exists on each of these; to avoid duplication of effort the focus is on documentation from a workflow efficiency perspective.

4.5.1 Dynamic documents with knitr

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 such as LibreOffice to import and communicate the results in prose. This is inefficient: it makes updating and maintaining the outputs difficult (when the data changes, steps 1 to 3 will have to be done again) and there is an overhead involved in jumping between incompatible computing environments.

To overcome this inefficiency in the documentation of R outputs the **knitr** package was developed. Used in conjunction with RStudio and building on a version of Markdown that accepts R code (RMarkdown, saved as .Rmd files) **knitr** allows for documents to be generated automatically. Furthermore, *nothing* is efficient unless you can quickly redo it. Documenting your code inside dynamic documents in this ways ensures that this is the case.

This not briefly explains RMarkdown for the un-initiated. RMarkdown is a form of Markdown. Markdown is a pure text document format that has become a standard for documentation for software. It is the default format for displaying text on GitHub. RMarkdown allows the user to embed R code in a Markdown document. This is a powerful addition to Markdown, as it allows custom images, tables and even interactive visualisations, to be included in your R documents. RMarkdown is an efficient file format to write in because it is light-weight, human and computer readable, and is much less verbose than HTML, LaTeX. This book was written in RMarkdown.

Results are generated *on the fly* by including 'code chunks' such as that illustrated below:

````
```r
(1:5)^2
#> [1]  1  4  9 16 25
```
````

This code, will result in the following output when the RMarkdown document is compiled:

```r
(1:5)^2
#> [1]  1  4  9 16 25
```

The resulting output is evaluated each time the document is compiled. To tell **knitr** that `(1:5)^2` is R code that needs to be evaluated, it must by preceded by `"‘{r}` on the line before the R code, and `"‘` at the end of the chunk. When you adapt to this workflow it is highly efficient, especially as RStudio provides a number of shortcuts that make it easy to create and modify code chunks. To create a chunk while editing a .Rmd file, for example, simply enter `Ctrl+Alt+I` on Windows or Linux or select the option from the Code dropdown menu.

Once your document has compiled it should appear on your screen into the file format requested. If a html file has been generated (as is the default), RStudio provides a feature that allows you to put it up online rapidly. This is done using the rpubs website, a store of a huge number of dynamic documents (which could be a good source of inspiration for your publications). Assuming you have an RStudio account, clicking the 'Publish' button at the top of the html output window will instantly publish your work online, with a minimum of effort, enabling fast and efficient communication with many collaborators and the public.

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. This can save hours of fiddly copying and pasting of R output between different programs. Also, if your client wants pages and pages of documented output, **knitr** can provide them with a minimum of typing, e.g. by creating slightly different versions of the same plot over and over again. From a delivery of content perspective, that is certainly an efficiency gain compared with hours of copying and pasting figures!

If your RMarkdown documents include time-consuming processing stages, a speed boost can be attained after the first build by setting `opts_chunk$set(cache = TRUE)` in the first chunk of the document. This setting was used to reduce the build times of this book, as can be seen on GitHub.

Furthermore dynamic documents written in RMarkdown can compile into a range of output formats including html, pdf and Microsoft's docx. There is a wealth of information on the details of dynamic report writing that is not worth replicating here. Key references are RStudio's excellent website on RMarkdown hosted at rmarkdown.rstudio.com/ and, for a more detailed account of dynamic documents with R, (Xie 2015).

4.5.2 R packages

A strict approach to project management and workflow is treating your projects as R packages. This approach has advantages and limitations. The major risk with treating a project as a package is that the package is quite a strict way of organising work. Packages are suited for code intensive projects where code documentation is important. An intermediate approach is to use a 'dummy package' that includes a DESCRIPTION file in the root directory telling users of the project which packages must be installed for the code to work. This book is based on a dummy package so that we can easily keep the dependencies up-to-date (see the book's DESCRIPTION file online for an insight into how this works).

Creating packages is good practice in terms of learning to correctly document your code, store example data, and even (via vignettes) ensure reproducibility. But it can take a lot of extra time so should not be taken lightly. This approach to R workflow is appropriate for managing complex projects which repeatedly use the same routines which can be converted into functions. Creating project packages can provide foundation for generalising your code for use by others, e.g. via publication on GitHub or CRAN. And R package development has been made much easier in recent years by the development of the **devools** package, which is highly recommended for anyone attempting to write an R package.

The number of essential elements of R packages differentiate them from other R projects. Three of these are outlined below from an efficiency perspective.

- The `DESCRIPTION` file contains key information about the package, including which packages are required for the code contained in your package to work, e.g. using `Imports:`. This is efficient because it means that anyone who installs your package will automatically install the other packages that it depends on.
- The `R/` folder contains all the R code that defines your package's functions. Placing your code in a single place and encouraging you to make your code modular in this way can greatly reduce duplication of code on large projects. Furthermore the documentation of R packages through Roxygen tags such as `#' This function does this...` makes it easy for others to use your work. This form of efficient documentation is facilitated by the **roxygen2** package.
- The `data/` folder contains example code for demonstrating to others how the functions work and transporting datasets that will be frequently used in your workflow. Data can be added automatically

to your package project using the **devtools** package, with `devtools::use_data()`. This can increase efficiency by providing a way of distributing small to medium sized datasets and making them available when the package is loaded with the function `data('data_set_name')`.

The package **testthat** makes it easier than ever to test your R code as you go, ensuring that nothing breaks. This, combined with 'continuous integration' services, such as that provided by Travis, make updating your code base as efficient and robust as possible. This, and more, is described in (Cotton 2016).

As with dynamic documents, package development is a large topic. For small 'one-off' projects the time taken in learning how to set-up a package may not be worth the savings. However packages provide a rigorous way of storing code, data and documentation that can greatly boost productivity in the long-run. For more on R packages see (Wickham 2015): the online version provides all you need to know about writing R packages for free (see r-pkgs.had.co.nz/).

Chapter 5

Efficient input/output

Input/output (I/O) is 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, the majority of R resources and documentation start with the optimistic assumption that your data has already been loaded, ignoring the fact that importing datasets into R, and exporting them to the world outside the R ecosystem, can be a time-consuming and frustrating process. Tricky, slow or ultimately unsuccessful data I/O can cripple efficiency right at the outset of a project. Conversely, reading and writing your data efficiently will make it your R projects more likely to succeed in the outside world. With these motivations in mind, this chapter explains how to efficiently read and write data in R.

The first section introduce **rio**, a 'meta package' for efficiently reading and writing data in a range of file formats. **rio** requires only two intuitive functions for data I/O, making it efficient to learn and use. Next we explore in more detail efficient functions for reading in files stored in common *plain text* file formats from the **readr** and **data.table** packages. Binary formats, which can dramatically reduce file sizes and read/write times, are covered next.

With the accelerating digital revolution and growth in open data, an increasing proportion of the world's data can be downloaded from the internet. This trend is set to continue, making section 5.6, on downloading and importing data from the web, important for 'future-proofing' you I/O skills. The benchmarks in this chapter demonstrate that choice of file format and packages for data I/O can have a huge impact on computational efficiency.

Before reading in a single line of data, it is worth considering a general principle for reproducible data management: never modify raw data files. Raw data should be seen as read-only, and contain information about its provenance. Keeping the original file name and commenting on its origin are a couple of ways to improve reproducibility, even when the data are not publicly available.

5.1 Top 5 tips for efficient data I/O

- 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.
- R's native file format is `.Rds`. These files can imported and exported using `readRDS` and `saveRDS` for fast and space efficient data storage.
- Use `import()` from the **rio** package to efficiently import data from a wide range of formats, avoiding the hassle of loading format-specific libraries.
- Use **readr** or **data.table** equivalents of `read.table()` to efficiently import large text files.

- 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.

5.2 Versatile data import with rio

rio is is a 'A Swiss-Army Knife for Data I/O'. **rio** provides easy-to-use and computationally efficient functions for importing and exporting tabular data in a range of file formats. As stated in the package's vignette, **rio** aims to "simplify the process of importing data into R and exporting data from R." The vignette goes on to to explain how many of the functions for data I/O described in R's Data Import/Export manual are out of date (for example referring to **WriteXLS** but not the more recent **readxl** package) and difficult to learn.

This is why **rio** is covered at the outset of this chapter: if you just want to get data into R, with a minimum of time learning new functions, there is a fair chance that **rio** can help, for many common file formats. At the time of writing, these include `.csv`, `.feather`, `.json`, `.dta`, `.xls`, `.xlsx` and Google Sheets (see the package's github page for up-to-date information). Below we illustrate the key **rio** functions of `import()` and `export()`:

```
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")
```

There was no need to specify the optional `format` argument for data import and export functions because this is inferred by the *suffix*, in the above example `.tsv` and `.xlsx` respectively. You can override the inferred file format for both functions with the `format` argument. You could, for example, create a comma-delimited file called `voc_voyages.xlsx` with `export(voyages, "voc_voyages.xlsx", format = "csv")`. However, this would **not** be a good idea: it is important to ensure that a file's suffix matches it's format.

To provide another example, the code chunk below downloads and imports as a data frame information about the countries of the world stored in `.json` (downloading data from the internet is covered in more detail in Section 5.6):

```
capitals = import("https://github.com/mledoze/countries/raw/master/countries.json")
```

The ability to import and use `.json` data is becoming increasingly common as it a standard output format for many APIs. The **jsonlite** and **geojsonio** packages have been developed to make this as easy as possible.

Exercises

1. The final line in the code chunk above shows a neat feature of **rio** and some other packages: the output format is determined by the suffix of the file-name, which make for concise code. Try opening the `voc_voyages.xlsx` file with an editor such as LibreOffice Calc or Microsoft Excel to ensure that the export worked, before removing this rather inefficient file format from your system:

```
file.remove("voc_voyages.xlsx")
```

2. Try saving the the `voyages` data frames into 3 other file formats of your choosing (see `vignette("rio")` for supported formats). Try opening these in external programs. Which file formats are more portable?

3. As a bonus exercise, create a simple benchmark to compare the write times for the different file formats used to complete the previous exercise. Which is fastest? Which is the most space efficient?

5.3 Plain text formats

'Plain text' data files are encoded in a format (typically UTF-8) that can be read by humans and computers alike. The great thing about plain text is their simplicity and their ease of use: any programming language can read a plain text file. The most common plain text format is `.csv`, comma-separated values, in which columns are separated by commas and rows are separated by line breaks. This is illustrated in the simple example below:

Person, Nationality, Country of Birth Robin, British, England Colin, British, Scotland

There is often more than one way to read data into R and `.csv` files are no exception. The method you choose has implications for computational efficiency. This section investigates methods for getting plain text files into R, with a focus on three approaches: base R's plain text reading functions such as `read.csv`; the **data.table** approach, which uses the function `fread`; and the newer **readr** package which provides `read_csv` and other `read_` functions such as `read_tsv`. Although these functions perform differently, they are largely cross-compatible, as illustrated in the below chunk, which loads data on the concentration of CO^2 in the atmosphere over time:

In general, you should never "hand-write" a CSV file. Instead, you should use `write.csv` or an equivalent function. The Internet Engineering Task Force has the CSV definition that facilites sharing CSV files between tools and operating systems.

```
df_co2 = read.csv("data/co2.csv")
df_co2_dt = readr::read_csv("data/co2.csv")
df_co2_readr = data.table::fread("data/co2.csv")
```

Note that a function 'derived from' another in this context means that it calls another function. The functions such as `read.csv` and `read.delim` in fact are *wrappers* around `read.table`. This can be seen in the source code of `read.csv`, for example, which shows that the function is roughly the equivalent of `read.table(file, header = TRUE, sep = ",")`.

Although this section is focussed on reading text files, it demonstrate the wider principle that the speed and flexibility advantages of additional read functions can be offset by the disadvantages of addition package dependency (in terms of complexity and maintaining the code) for small datasets. The real benefits kick in on large datasets. Of course, there are some data types that *require* a certain package to load in R: the **readstata13** package, for example, was developed solely to read in `.dta` files generated by versions of Stata 13 and above.

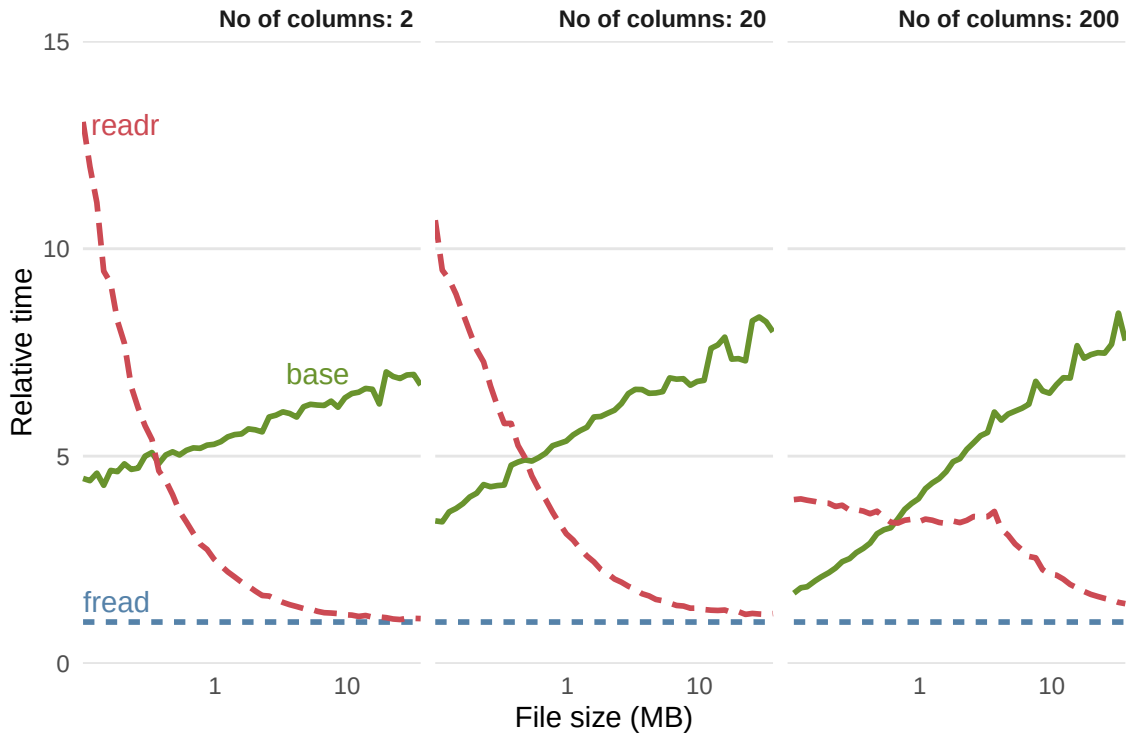


Figure 5.1: Benchmarks of **base**, **data.table** and **readr** functions for reading csv files. The facets ranging from 2 to 200 represent the number of columns in the csv file.

Figure 5.1 demonstrates that the relative performance gains of the **data.table** and **readr** approaches increase with data size, especially so for data with many rows. Below around 1 MB `read.delim` is actually *faster* than `read_csv` while `fread` is much faster than both, although these savings are likely to be inconsequential for such small datasets.

For files beyond 100 MB in size `fread` and `read_csv` can be expected to be around *5 times faster* than `read.delim`. This efficiency gain may be inconsequential for a one-off file of 100 MB running on a fast computer (which still takes less than a minute with `read.csv`), but could represent an important speed-up if you frequently load large text files.

When tested on a large (4GB) `.csv` file it was found that `fread` and `read_csv` were almost identical in load times and that `read.csv` took around 5 times longer. This consumed more than 10GB of RAM, making it unsuitable to run on many computers (see Section 8.3 for more on memory). Note that both **readr** and base methods can be made significantly faster by pre-specifying the column types at the outset (see below). Further details are provided by the help in `?read.table`.

```
read.csv(file_name, colClasses = c("numeric", "numeric"))
```

In some cases with R programming there is a trade-off between speed and robustness. This is illustrated below with reference to differences in how **readr**, **data.table** and base R approaches handle unexpected values. Table 5.1 shows that `read_tsv` is around 3 times faster, re-enforcing the point that the benefits of efficient functions increase with dataset size (made with Figure 5.1). This is a small (1 MB) dataset: the relative difference between `fread` and `read_` functions will tend to decrease as dataset size increases.

```
library("microbenchmark")
library("readr")
library("data.table")
```

Table 5.1: Execution time of base, **readr** and **data.table** functions for reading in a 1 MB dataset relative to the mean execution time of 'fread', around 0.02 seconds on a modern computer.

Function	min	mean	max
base_read	10.7	11.1	11.4
readr_read	3.4	4.7	13.5
dt_fread	1.0	1.0	1.0

```
fname = system.file("extdata/voc_voyages.tsv", package = "efficient")
res_v = microbenchmark(times = 10,
  base_read = voyages_base <- read.delim(fname),
  readr_read = voyages_readr <- read_tsv(fname),
  dt_fread = voyages_dt <- fread(fname))
```

The benchmark above produces warning messages (not shown) for the `read_tsv` and `fread` functions but not the slowest base function `read.delim`. An exploration of these functions can shed light on the speed/robustness trade-off.

- The **readr** function `read_csv` generates a warning for row 2841 in the `built` variable. This is because `read_*()` decides what class each variable is based on the first 1000 rows, rather than all rows, as base `read.*` functions do.

As illustrated by printing the result for the row which generated a warning, the `read_tsv` output is more sensible than the `read.delim` output: `read.delim` coerced the date field into a factor based on a single entry which is a text. `read_tsv` coerced the variable into a numeric vector, as illustrated below.

```
class(voyages_base$built) # coerced to a factor
#> [1] "factor"
class(voyages_readr$built) # numeric based on first 1000 rows
#> [1] "numeric"
voyages_base$built[2841] # contains the text responsible for coercion
#> [1] 1721-01-01
#> 182 Levels:  1 784 1,86 1135 1594 1600 1612 1613 1614 1615 1619 ... taken 1672
voyages_readr$built[2841] # an NA: text cannot be converted to numeric
#> [1] NA
```

- The **data.table** function `fread` generates 5 warning messages stating that columns 2, 4, 9, 10 and 11 were `Bumped to type character on data row ...`, with the offending rows printed in place of `....` Instead of changing the offending values to `NA`, as **readr** does for the `built` column (9), `fread` automatically converts any columns it thought of as numeric into characters. An additional feature of `fread` is that it can read-in a selection of the columns, either by their index or name, using the `select` argument. This is illustrated below by reading in only half (the first 11) columns from the voyages dataset and comparing the result with `fread`'ing all the columns in.

```
microbenchmark(times = 5,
  with_select = fread(fname, select = 1:11),
  without_select = fread(fname)
  )
#> Unit: milliseconds
#>            expr  min   lq mean median   uq  max neval
#>     with_select 10.4 10.6 11.3   11.5 11.7 12.2     5
#>  without_select 19.4 20.8 21.0   20.8 21.6 22.3     5
```

Table 5.2: Execution time of base, **readr** and **data.table** functions for reading in a 1 MB dataset

Function	number	boatname	built	departure_date
base_read	integer	factor	factor	factor
readr_read	integer	character	numeric	Date
dt_fread	integer	character	character	character

To summarise, the differences between base, **readr** and **data.table** functions for reading in data go beyond code execution times. The functions `read_csv` and `fread` boost speed partially at the expense of robustness because they decide column classes based on a small sample of available data. The similarities and differences between the approaches are summarised for the Dutch shipping data in Table 5.2.

The file `voc_voyages` was taken from a dataset on Dutch naval expeditions used with permission from the CWI Database Architectures Group. The data is dscribed more fully at monetdb.org. From this dataset we primarily use the 'voyages' table with lists Dutch shipping expeditions by their date of departure.

Table 5.2 shows 4 main similarities and differences between the three read types of read function:

- For uniform data such as the 'number' variable in Table 5.2, all reading methods yield the same result (integer in this case).
- For columns that are obviously characters such as 'boatname', the base method results in factors (unless `stringsAsFactors` is set to `TRUE`) whereas `fread` and `read_csv` functions return characters.
- For columns in which the first 1000 rows are of one type but which contain anomalies, such as 'built' and 'departure_data' in the shipping example, `fread` coerces the result to characters. `read_csv` and siblings, by contrast, keep the class that is correct for the first 1000 rows and sets the anomalous records to `NA`. This is illustrated in 5.2, where `read_tsv` produces a `numeric` class for the 'built' variable, ignoring the non numeric text in row 2841.
- `read_*` functions generate objects of class `tbl_df`, an extension of the `data.frame`, as discussed in Section 6.4. `fread` generates objects of class `data.table`. These can be used as standard data frames but differ subtly in their behaviour.

An additional difference is that `read_csv()` creates data frames of class `tbl_df`, *and* the `data.frame`. This makes no practical difference, unless the **tibble** package is loaded, as described in section 6.2 in the next chapter.

The wider point associated with these tests is that functions that save time can also lead to additional considerations or complexities your workflow. Taking a look at what is going on 'under the hood' of fast functions to increase speed, as we have done in this section, can help understand the knock-on consequences of choosing fast functions over slower functions from base R.

5.3.1 Preprocessing text outside R

There are circumstances when datasets become too large to read directly into R. Reading in a 4 GB text file using the functions tested above, for example, consumes all available RAM on an 16 GB machine. To overcome this limitation, external *stream processing* tools can be used to preprocess large text files. The

following command, using the Linux command line 'shell' (or Windows based Linux shell emulator Cygwin) command `split`, for example, will break a large multi GB file many one GB chunks, each of which is more manageable for R:

```
split -b100m bigfile.csv
```

The result is a series of files, set to 100 MB each with the `-b100m` argument in the above code. By default these will be called `xaa`, `xab` and which could be read in *one chunk at a time* (e.g. using `read.csv`, `fread` or `read_csv`, described in the previous section) without crashing most modern computers.

Splitting a large file into individual chunks may allow it to be read into R. This is not an efficient way to import large datasets, however, because it results in a non-random sample of the data this way. A more efficient, robust and scalable way to work with large datasets is via databases, covered in Section 6.6 in the next chapter.

5.4 Binary file formats

Once you have read-in (e.g. from a plain text file) and tidied your data (covered in the next chapter), it is common to want to save it for future use. Saving it after tidying is recommended, to reduce the chance of having to run all the data cleaning code again. We recommend saving tidied versions of large datasets in one of the binary formats covered in this section: this will decrease read/write times and file sizes, making your data more portable.[1]

Unlike plain text files data stored in binary formats cannot be read by humans. This allows space-efficient data compression but means that the files will be less language agnostic. R's native file format, `.Rds`, for example may be difficult to read and write using external programs such as Python or LibreOffice Calc. This section provides an overview binary file formats in R, with benchmarks to show how they compare with the plain text format `.csv` covered in the previous section.

5.4.1 Native binary formats: Rdata or Rds?

`.Rds` and `.RData` are R's native binary file formats. These formats are optimised for speed and compression ratios. But what is the difference between them? The follow code chunk demonstrates the key difference between them:

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

The first method is the most widely used. It uses the `save` function which takes any number of R objects and writes them to a file, which must be specified by the `file =` argument. `save` is like `save.image`, which saves *all* the objects currently loaded in R.

The second method is slightly less used but we recommend it. Apart from being slightly more concise for saving single R objects, the `readRDS` function is more flexible: as shown in the subsequent line, the resulting object can be assigned to any name. In this case we called it `df_co2_rds` (which we show to be identical to

[1] Geographical data, for example, can be slow to read in external formats. A large `.shp` or `.geojson` file can take more than 100 times longer to load than an equivalent `.Rds` or `.Rdata` file.

Table 5.3: Absolute and relative (compared with the smallest size for each column) read times for the 3 column, 468 row 'co2' dataset, saved with different functions. Columns headed 10x, 100x and 1000x show the results for disk usage after increasing the number of rows by 10, 100 and 1000 fold respectively.

Function	Time	Rel	Time (10x)	Rel (10x)	Time (100x)	Rel (100x)	Time (1000x)	Rel (1000x)
read.csv	1.8	15.1	13.1	110.8	141.5	250.7	1639.3	450.0
read_feather	0.1	1.0	0.1	1.0	0.6	1.0	3.6	1.0
load	0.2	1.4	0.4	3.3	2.6	4.6	28.1	7.7
readRDS	0.1	1.1	0.4	3.0	2.5	4.5	27.7	7.6

`df_co2`, loaded with the `load` command) but we could have called it anything or simply printed it to the console.

Using `saveRDS` is good practice because it forces you to specify object names. If you use `save` without care, you could forget the names of the objects you saved and accidentally overwrite objects that already existed.

5.5 The feather file format

Feather was developed as collaboration between R and Python developers to create a fast, light and language agnostic format for storing data frames. The code chunk below shows how it can be used to save and then re-load the `df_co2` dataset loaded previously in both R and Python:

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

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

5.5.1 Benchmarking binary file formats

We know that binary formats are advantageous from space and read/write time perspectives. The benchmarks in this section, based on the replicating the `co2` dataset used in the previous section, are designed to help understand *how much* more efficient they are. First we look at file size savings. Figure 5.2 shows that the *relative* space saving of native R formats increase with dataset size. It is striking to note that binary `.Rds` files can be more than 100 times more space efficient than plain text `.csv` files: a 30 MB `.csv` containing replicated data from the `.co2` dataset contains the same information as 0.1 MB `.Rds` file!

The results of this simple disk usage benchmark show that saving data in a compressed binary format can improve efficiency from hard-disk and, if your data will be shared on-line, data download time and bandwidth usage perspectives. But how does each method compare from a computational efficiency perceptive?

The read and write times for the functions showcased above are presented in Table 5.3 and Table 5.4 respectively.

The results show that the relative *size* of different formats is not a reliable predictor of data read and write times. This is due to the computational overheads of compression. Although the binary `.feather` format did not perform well in terms of read and write times, the function `read_feather` is *faster* than R's native functions for saving `.Rds` and `.RData` formats, for the datasets used in the benchmark. `write_feather` is also faster than `save` and `saveRDS` for all but the largest dataset. In all cases, `read.csv` and `write.csv` is

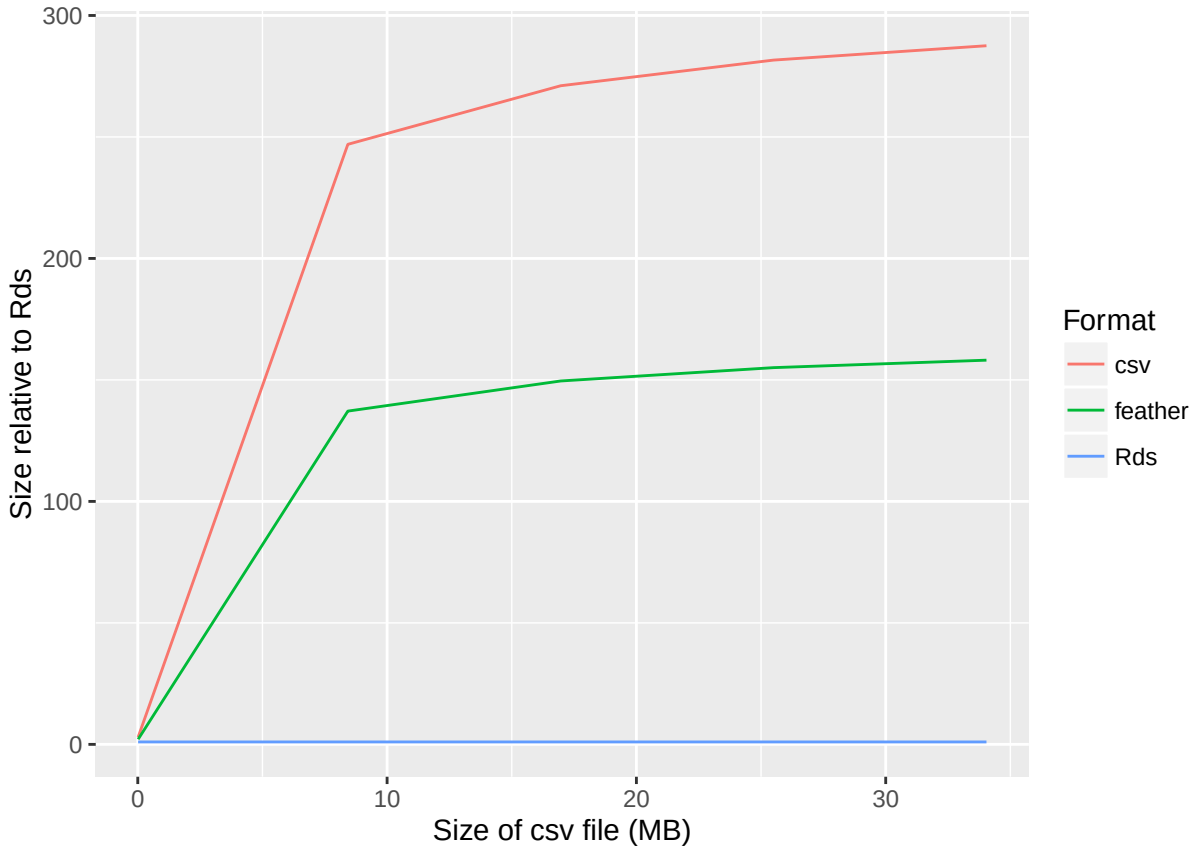


Figure 5.2: Absolute (MB) and relative (compared with the Rds file format) file sizes of the replicated 'co2' dataset saved in various file formats. Different compression ratios can be expected for more complex datasets.

several times slower than the binary formats and this relative slowness worsens with increasing dataset size. In the next section we explore the performance of alternatives to these base R functions for reading and writing plain text data files.

5.6 Getting data from the internet

The code chunk below shows how the functions `download.file`[2] and `unzip` can be used to download and unzip a dataset from the internet. R can automate processes that are often performed manually, e.g. through the graphical user interface of a web browser, with potential advantages for reproducibility and programmer efficiency. The result is data stored neatly in the `data` directory ready to be imported. Note we deliberately kept the file name intact help with documentation, enhancing understanding of the data's *provenance*, so future users can quickly find out where the data came from. Note also that part of the dataset is stored in the **efficient** package. Using R for basic file management can help create a reproducible workflow, as illustrated below.

```
url = "https://www.monetdb.org/sites/default/files/voc_tsvs.zip"
download.file(url, "voc_tsvs.zip") # download file
unzip("voc_tsvs.zip", exdir = "data") # unzip files
file.remove("voc_tsvs.zip") # tidy up by removing the zip file
```

[2]Since R 3.2.3 the base function `download.file()` can be used to download from secure (`https://`) connections on any operating system.

Table 5.4: Absolute and relative (compared with the smallest size for each column) write times for the 3 column, 468 row 'co2' dataset, saved with different functions. Columns headed 10x, 100x and 1000x show the results for disk usage after increasing the number of rows by 10, 100 and 1000 fold respectively.

Function	Time	Rel	Time (10x)	Rel (10x)	Time (100x)	Rel (100x)	Time (1000x)	Rel (1000x)
write.csv	2.8	6.9	21.1	33.0	197.3	27.5	2312.4	27.6
save_feather	0.4	1.0	0.6	1.0	7.2	1.0	142.8	1.7
save	1.7	4.2	1.6	2.5	10.5	1.5	95.3	1.1
saveRDS	1.7	4.2	1.5	2.3	9.5	1.3	83.9	1.0

This workflow equally applies to downloading and loading single files. Note that one could make the code more concise by entering replacing the second line with `df = read.csv(url)`. However, we recommend downloading the file to disk so that if for some reason it fails (e.g. if you would like to skip the first few lines), you don't have to keep downloading the file over and over again. The code below downloads and loads data on atmospheric concentrations of CO^2. Note that this dataset is also available from the **datasets** package.

```
url = "https://vincentarelbundock.github.io/Rdatasets/csv/datasets/co2.csv"
download.file(url, "data/co2.csv")
df_co2 = read_csv("data/co2.csv")
```

There are now many R packages to assist with the download and import of data. The organisation ROpenSci supports a number of these. The example below illustrates this using the WDI package (not supported by ROpenSci) to accesses World Bank data on CO2 emissions in the transport sector:

```
library("WDI") # load the WDI library (must be installed)
WDIsearch("CO2") # search for data on a topic
co2_transport = WDI(indicator = "EN.CO2.TRAN.ZS") # import data
```

There will be situations where you cannot download the data directly or when the data cannot be made available. In this case, simply providing a comment relating to the data's origin (e.g. `# Downloaded from http://example.com`) before referring to the dataset can greatly improve the utility of the code to yourself and others.

5.7 Accessing data stored in packages

Most well documented packages provide some example data for you to play with. This can help demonstrate use cases in specific domains, that uses a particular data format. The command `data(package = "package_name")` will show the datasets in a package. Datasets provided by **dplyr**, for example, can be viewed with `data(package = "dplyr")`.

Raw data (i.e. data which has not been converted into R's native `.Rds` format) is usually located with the sub-folder `extdata` in R (which corresponds to `inst/extdata` when developing packages. The function `system.file` outputs file paths associated with specific packages. To see all the external files within the **readr** package, for example, one could use the following command:

```
list.files(system.file("extdata", package = "readr"))
#> [1] "compound.log"      "epa78.txt"         "example.log"
#> [4] "fwf-sample.txt"    "massey-rating.txt" "mtcars.csv"
#> [7] "mtcars.csv.bz2"    "mtcars.csv.zip"
```

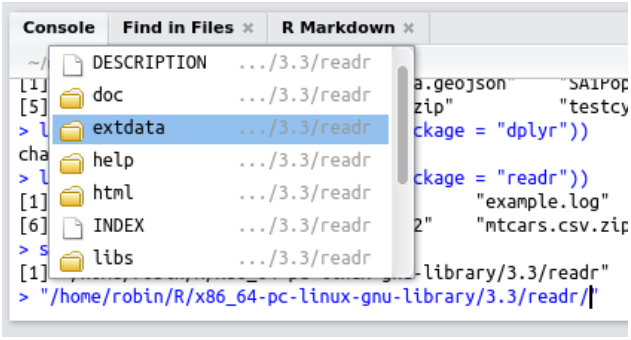


Figure 5.3: Discovering files in R packages using RStudio's 'intellisense'.

Further, to 'look around' to see what files are stored in a particular package, one could type the following, taking advantage of RStudio's intellisense file completion capabilities (using copy and paste to enter the file path):

```
system.file(package = "readr")
#> [1] "/home/robin/R/x86_64-pc-linux-gnu-library/3.3/readr"
"/home/robin/R/x86_64-pc-linux-gnu-library/3.3/readr/"
```

Hitting `Tab` after the second command should trigger RStudio to create a miniature pop-up box listing the files within the folder, as illustrated in figure 5.3.

Chapter 6

Efficient data carpentry

There are many words for data processing. You can **clean**, **hack**, **manipulate**, **munge**, **refine** and **tidy** your dataset, ready for the next stage, typically modelling and visualisation. Each word says something about perceptions towards the process: data processing is often seen as *dirty work*, an unpleasant necessity that must be endured before the *real*, *fun* and *important* work begins. This perception is wrong. Getting your data 'ship shape' is a respectable and in some cases vital skill. For this reason we use the more admirable term *data carpentry*.

The metaphor is not accidental. Carpentry is the process of taking rough pieces of wood and working with care, diligence and precision to create a finished product. A carpenter does not hack at the wood at random. He or she will inspect the raw material and select the right tool for the job. In the same way *data carpentry* is the process of taking rough, raw and to some extent randomly arranged input data and creating neatly organised and *tidy* data. Learning the skill of data carpentry early will yield benefits for years to come. "Give me six hours to chop down a tree and I will spend the first four sharpening the axe" as Abraham Lincoln put it, continuing the metaphor.

Data processing is a critical stage in any project involving any datasets from external sources, i.e. most real world applications. In the same way that *technical debt*, discussed in the previous Chapter, can cripple your workflow, working with messy data can lead to project management hell.

Fortunately, done efficiently, at the outset of your project (rather than half way through, when it may be too late) and using appropriate tools this data processing stage can be highly rewarding. More importantly from an efficiency perspective, working with clean data will be beneficial for every subsequent stage of your R project. So, for data intensive applications, this could be the most important chapter of book. In it we cover the following topics:

- Tidying data with **tidyr**
- Processing data with **dplyr**
- Working with databases
- Data processing with **data.table**

6.1 Top 5 tips for efficient data carpentry

- Give data processing the proper time and attention it needs as it could save hours of frustration in the long run.
- Tidy your data carefully at the earliest stage of the analysis process, perhaps using functions provided by **tidyr**.

- Use the `tbl` class defined by the **tibble** package and the default object type of **dplyr**, to make data frames efficient to print and easier to work with.
- Don't rely on base R for data processing. We recommend **dplyr** for most applications, although **data.table** may be optimal in contexts where speed is critical.
- Using the `%>%` 'pipe' operator in combination with **dplyr**'s verbs can help clarify complex data processing workflows, easing the writing of analysis code and communication with others.

6.2 Efficient data frames with tibble

tibble is a package that defines a new data frame class for R, the `tbl_df`. These 'tibble diffs' (as their inventor suggests they should be pronounced) are like the base class `data.frame`, but with more user friendly printing, subsetting, and factor handling.

A tibble data frame is an S3 object with three classes, `tbl_df`, `tbl`, and `data.frame`. Since the object has the `data.frame` tag, this means that if a `tbl_df` or `tbl` method isn't available, the object will be passed on to the appropriate `data.frame` function.

To create a tibble data frame, we use `tibble` function

```
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
```

The example above illustrates the main differences between the **tibble** and base R approach to data frames:

- When printed, the tibble diff reports the class of each variable. `data.frame` objects do not.
- Character vectors are not coerced into factors when they are incorporated into a `tbl_df`, as can be seen by the `<chr>` heading between the variable name and the second column. By contrast, `data.frame()` coerces characters into factors which can cause problems further down the line.
- When printing a tibble diff to screen, only the first ten rows are displayed. The number columns printed depends on the window size.

Other differences can be found in the associated help page - `help("tibble")`.

You can create a tibble data frame row-by-row using the `frame_data` function.

Exericse

Create the following data frame

```
df_base = data.frame(colA = "A")
```

Try and guess the output of the following commands

```
print(df_base)
df_base$colA
df_base$col
df_base$colB
```

Now create a tibble data frame and repeat the above commands.

6.3 Tidying data with tidyr

A key skill in data analysis is understanding the structure of datasets and being able to 'reshape' them. This is important from a workflow efficiency perspective: more than half of a data analyst's time can be spent re-formatting datasets (H. Wickham 2014b), so getting it into a suitable form early could save hours in the future. Converting data into a 'tidy' form is also advantageous from a computational efficiency perspective: it is usually faster to run analysis and plotting commands on tidy data.

Data tidying includes data cleaning and data reshaping. Data cleaning is the process of re-formatting and labelling messy data. Packages including **stringi** and **stringr** can help update messy character strings using regular expressions; **assertive** and **assertr** packages can perform diagnostic checks for data integrity at the outset of a data analysis project. A common data cleaning task is the conversion of non-standard text strings into date formats as described in the **lubridate** vignette (see `vignette("lubridate")`). Tidying is a broader concept, however, and also includes re-shaping data so that it is in a form more conducive to data analysis and modelling. The process of reshaping is illustrated by Tables 6.1 and 6.2, provided by H. Wickham (2014b).

These tables may look different, but they contain precisely the same information. Column names in the 'wide' form in Table 6.1 became a new variable in the 'long' form in Table 6.2. According to the concept of 'tidy data', the long form is correct. Note that 'correct' here is used in the context of data analysis and graphical visualisation. For tabular presentation the wide form may be better. Wide data can also be less memory consuming because it requires fewer cells.

Tidy data has the following characteristics (H. Wickham 2014b):

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

Because there is only one observational unit in the example (religions), it can be described in a single table. Large and complex datasets are usually represented by multiple tables, with unique identifiers or 'keys' to join them together (Codd 1979).

Two common operations facilitated by **tidyr** are *gathering* and *splitting* columns.

- Gathering: this means making 'wide' tables 'long' by converting column names to a new variable. This is done is done with the function `gather` (the inverse of which is `spread`) , as illustrated in Table 6.1 and Table 6.2 and in the code block below:

Table 6.1: First 6 rows of the aggregated 'pew' dataset from Wickham (2014a) in an 'untidy' form.

religion	<$10k	$10–20k	$20–30k
Agnostic	27	34	60
Atheist	12	27	37
Buddhist	27	21	30

Table 6.2: Long form of the Pew dataset represented above.

religion	Income	Count
Agnostic	<$10k	27
Atheist	<$10k	12
Buddhist	<$10k	27
Agnostic	$10–20k	34
Atheist	$10–20k	27
Buddhist	$10–20k	21
Agnostic	$20–30k	60
Atheist	$20–30k	37
Buddhist	$20–30k	30

```
library("tidyr")
library("readr")
raw = read_csv("data/pew.csv") # read in the 'wide' dataset
dim(raw)
#> [1] 18 10
rawt = gather(raw, Income, Count, -religion)
dim(rawt)
#> [1] 162   3
rawt[1:3,]
#> # A tibble: 3 x 3
#>   religion Income Count
#>      <chr>  <chr> <int>
#> 1 Agnostic  <$10k    27
#> 2  Atheist  <$10k    12
#> 3 Buddhist  <$10k    27
```

Note that the dimensions of the data change from having 10 observations across 18 columns to 162 rows in only 3 columns. Note that when we print the object `rawt[1:3,]`, the class of each variable is given (`chr`, `fctr`, `int` refer to character, factor and integer classes, respectively). This is because **readr** creates `tbl` objects from the **tibble** package.

- Splitting: this means taking a variable that is really two variables combined and creating two separate columns from it. A classic example is age-sex variables (e.g. `m0-10` and `f0-15` to represent males and females in the 0 to 10 age band). Splitting such variables can be done with `separate`, as illustrated in Table 6.3 and 6.4.

Table 6.3: Joined age and sex variables in one column

agesex	n
m0-10	3
f0-10	5

Table 6.4: Age and sex variables separated by the funtion 'separate'.

sex	age	n
m	0-10	3
f	0-10	5

```
agesex = c("m0-10", "f0-10") # create compound variable
n = c(3, 5) # create a value for each observation
agesex_df = data.frame(agesex, n) # create a data frame
separate(agesex_df, agesex, c("sex", "age"), 1)
#>   sex  age n
#> 1   m 0-10 3
#> 2   f 0-10 5
```

There are other tidying operations that **tidyr** can perform, as described in the package's vignette (`vignette("tidy-data")`). The wider issue of manipulation is a large topic with major potential implications for efficiency (Spector 2008) and this section only covers some of the key operations. More important is understanding the principles behind converting messy data into standard output forms. These same principles can also be applied to the representation of model results: the **broom** package provides a standard output format for model results, easing interpretation (see the broom vignette).

Note that the above code used non standard evaluation, which means not surrounding variable names in quote marks for ease of typing and autocompletion. This is fine when using R interactively. But when you'd like to use R non-interactively, code is generally more robust using standard evaluation.

Usually it is more efficient to use the non-standard evaluation version of variable names, as these can be auto completed by RStudio. In some cases you may want to use standard evaluation and refer to variable names using quote marks. To do this, affix `_` can be added to **dplyr** and **tidyr** function names to allow the use of standard evaluation. Thus the standard evaluation version of `separate(agesex_df, agesex, c("sex", "age"), 1)` is `separate_(agesex_df, "agesex", c("sex", "age"), 1)`.

6.4 Efficient data processing with dplyr

After tidying your data, the next stage is generally data processing. This includes the creation of new data, for example a new column that is some function of existing columns, or data analysis, the process of asking directed questions of the data and exporting the results in a user-readable form.

Following the advice in Section 4.4, we have carefully selected an appropriate package for these tasks: **dplyr**, which roughly means 'data pliers'. **dplyr** has a number of advantages over base R and **data.table** approaches to data processing:

Table 6.5: dplyr verb functions.

dplyr function(s)	Description	Base R functions
filter(), slice()	Subset rows by attribute (filter) or position (slice)	subset(), []
arrange()	Return data ordered by variable(s)	sort()
select()	Subset columns	subset()
rename()	Rename columns	names()
distinct()	Return unique rows	unique()
mutate()	Create new variables (transmute drops existing variables)	[[]]
summarise()	Collapse data into a single row	aggregate()
sample_n()	Return a sample of the data	sample()

Table 6.6: The top 3 countries in terms of average CO2 emissions from transport since 1971, and growth in transport emissions over that period (MTCO2e/yr).

Country	Mean	Growth
United States	1462	709
China	214	656
India	85	170

- **dplyr** is fast to run (due to its C++ backend) and intuitive to type
- **dplyr** works well with tidy data, as described above
- **dplyr** works well with databases, providing efficiency gains on large datasets

Furthermore, **dplyr** is efficient to *learn* (see Chapter 10. It has small number of intuitively named functions, or 'verbs'. These can be used in combination with each other, and the grouping function `group_by()`, to solve a wide range of data processing challenges (see Table 6.5).

Unlike the base R analogues, **dplyr**'s data processing functions work in a consistent way. Each function takes a data frame object as its first argument and results in another data frame. Variables can be called directly without using the `$` operator. **dplyr** was designed to be used with the 'pipe' operator `%>%` provided by the **magittr** package, allowing each data processing stage to be represented as a new line. This is illustrated in the code chunk below, which loads a tidy country level dataset of greehouse gas emissions from the **efficient** package, and then identifies the countries with the greatest absolute growth in emissions from 1971 to 2012:

```
library("dplyr")
data("ghg_ems", package = "efficient")
top_table =
  ghg_ems %>%
  filter(!grepl("World|Europe", Country)) %>%
  group_by(Country) %>%
  summarise(Mean = mean(Transportation),
            Growth = diff(range(Transportation))) %>%
  top_n(3, Growth) %>%
  arrange(desc(Growth))
```

The results, illustrated in table 6.6, show that the USA has the highest growth and average emissions from the transport sector, followed closely by China. The aim of this code chunk, which you are not expected to completely understand, is to provide a taster of **dplyr**'s unique syntax.

Building on the 'learning by doing' ethic, the remainder of this section works through these functions to process and begin to analyse a dataset on economic equality provided by the World Bank. The input dataset can be loaded as follows:

```
fname = system.file("extdata/world-bank-ineq.csv", package = "efficient")
idata = read_csv(fname)
idata # print the dataset
#> # A tibble: 6,925 x 9
#>   Country Name Country Code  Year Year Code
#>          <chr>        <chr> <int>     <chr>
#> 1  Afghanistan          AFG  1974    YR1974
#> 2  Afghanistan          AFG  1975    YR1975
#> 3  Afghanistan          AFG  1976    YR1976
#> 4  Afghanistan          AFG  1977    YR1977
#> # ... with 6,921 more rows, and 5 more variables: Income share held by
#> #   highest 10% [SI.DST.10TH.10] <chr>, Income share held by lowest 10%
#> #   [SI.DST.FRST.10] <chr>, GINI index [SI.POV.GINI] <chr>, Survey mean
#> #   consumption or income per capita, bottom 40% (2005 PPP $ per day)
#> #   [SI.SPR.PC40] <chr>, Survey mean consumption or income per capita,
#> #   total population (2005 PPP $ per day) [SI.SPR.PCAP] <chr>
```

You should not be expecting to master **dplyr**'s functionality in one sitting: the package is large and can be seen as a language in its own right. Following the 'walk before you run' principle, we'll start simple, by filtering and aggregating rows.

6.4.1 Renaming columns

Renaming data columns is a common task that can make writing code faster by using short, intuitive names. The **dplyr** function `rename()` makes this easy.

Note in this code block the variable name is surrounded by back-quotes (`). This allows R to refer to column names that are non-standard. Note also the syntax: `rename` takes the data frame as the first object and then creates new variables by specifying `new_variable_name = original_name`.

```
library("dplyr")
idata = rename(idata, Country = `Country Name`)
```

To rename multiple columns the variable names are simply separated by commas. The base R and **dplyr** way of doing this is illustrated for clarity.

```
# The dplyr way (rename two variables)
idata = rename(idata,
 top10 = `Income share held by highest 10% [SI.DST.10TH.10]`,
 bot10 = `Income share held by lowest 10% [SI.DST.FRST.10]`)
# The base R way (rename five variables)
names(idata)[5:9] = c("top10", "bot10", "gini", "b40_cons", "gdp_percap")
```

6.4.2 Changing column classes

The *class* of R objects is critical to performance. If a class is incorrectly specified (e.g. if numbers are treated as factors or characters) this will lead to incorrect results. The class of all columns in a `data.frame` can be queried using the function `str()` (short for display the **str**ucture of an object), as illustrated below, with the inequality data loaded previously.[1]

[1] `vapply(idata, class, character(1))` is an alternative way to query the class of columns in a dataset.

```
str(idata)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    6925 obs. of  9 variables:
#>  $ Country     : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
#>  $ Country Code: chr  "AFG" "AFG" "AFG" "AFG" ...
#>  $ Year        : int  1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 ...
#>  $ Year Code   : chr  "YR1974" "YR1975" "YR1976" "YR1977" ...
#>  $ top10       : chr  ".." ".." ".." ".." ...
#>  $ bot10       : chr  ".." ".." ".." ".." ...
#>  $ gini        : chr  ".." ".." ".." ".." ...
#>  $ b40_cons    : chr  ".." ".." ".." ".." ...
#>  $ gdp_percap  : chr  ".." ".." ".." ".." ...
```

This shows that although we loaded the data correctly all columns are seen by R as characters. This means we cannot perform numerical calculations on the dataset: `mean(idata$gini)` fails.

Visual inspection of the data (e.g. via `View(idata)`) clearly shows that all columns except for 1 to 4 (`Country`, `Country Code`, `Year` and `Year Code`) should be numeric. We can re-assign the classes of the numeric variables one-by one:

```
idata$gini = as.numeric(idata$gini)
#> Warning: NAs introduced by coercion
mean(idata$gini, na.rm = TRUE) # now the mean is calculated
#> [1] 40.5
```

However the purpose of programming languages is to *automate* tasks and reduce typing. The following code chunk re-classifies all of the numeric variables using `data.matrix()`, which converts the data frame to a numeric `matrix`:

```
id = 5:9 # column ids to change
idata[id] = data.matrix(idata[id])
vapply(idata, class, character(1))
#>      Country Country Code         Year    Year Code        top10
#>  "character"  "character"    "integer"  "character"    "numeric"
#>        bot10         gini     b40_cons   gdp_percap
#>    "numeric"    "numeric"    "numeric"    "numeric"
```

As is so often the case with R, there are many ways to solve the problem. Below is a one-liner using `unlist()` which converts list objects into vectors:

```
idata[id] = as.numeric(unlist(idata[id]))
```

Another one-liner to achieve the same result uses **dplyr**'s `mutate_each` function:

```
idata_mutate = mutate_each(idata, "as.numeric", id)
```

As with other operations there are other ways of achieving the same result in R, including the use of loops via `apply()` and `for()`. These are shown in the chapter's source code.

6.4.3 Filtering rows

dplyr offers an alternative and more flexible way of filtering data, using `filter()`.

```
# Base R: idata[idata$Country == "Australia",]
aus2 = filter(idata, Country == "Australia")
```

In addition to being more flexible (see `?filter`), `filter` is slightly faster than base R's notation.[2] Note that **dplyr** does not use the `$` symbol: it knows that that `Country` is a variable of `idata`: the first argument of **dplyr** functions usually a `data.frame`, and subsequent in this context variable names can be treated as vector objects.[3]

There are **dplyr** equivalents of many base R functions but these usually work slightly differently. The **dplyr** equivalent of `aggregate`, for example is to use the grouping function `group_by` in combination with the general purpose function `summarise` (not to be confused with `summary` in base R), as we shall see in Section 6.4.5. For consistency, however, we next look at filtering columns.

6.4.4 Filtering columns

Large datasets often contain much worthless or blank information. This consumes RAM and reduces computational efficiency. Being able to focus quickly only on the variables of interest becomes especially important when handling large datasets.

Imagine that we have a text file called `miniaa` which is large enough to consume most of your computer's RAM. We can load it with the following command:

```
data(miniaa, package="efficient") # load imaginary large data
dim(miniaa)
#> [1]   9 329
```

Note that the data frame has 329 columns, and imagine it has millions of rows, instead of $9. That's a lot of variables. Do we need them all? It's worth taking a glimpse at this dataset to find out:

```
glimpse(miniaa)
#> Observations: 9
#> Variables: 329
#> $ NPI
#>     <int> ...
#> ...
```

The majority of variables in this dataset only contain `NA`. To clean the giant dataset, removing the empty columns, we need to identify which variables these are.

```
# Identify the variable which are all NA
all_na = vapply(miniaa, function(x) anyNA(x), logical(1))
summary(all_na) # summary of the results
#>    Mode   FALSE    TRUE    NA's
#> logical     321       8       0
miniaa1 = miniaa[!all_na] # subset the dataframe
```

The new `miniaa` object has fewer than a third of the original columns.

[2]Note that `filter` is also the name of a function used in the base **stats** library. Usually packages avoid using names already taken in base R but this is an exception.

[3]Note that this syntax is a defining feature of **dplyr** and many of its functions work in the same way. Later we'll learn how this syntax can be used alongside the `%>%` 'pipe' command to write clear data manipulation commands.

6.4.5 Data aggregation

Data aggregation is the process of creating summaries of data based on a grouping variable. The end result usually has the same number of rows as there are groups. Because aggregation is a way of condensing datasets it can be a very useful technique for making sense of large datasets. The following code finds the number of unique countries (country being the grouping variable) from the `ghg_ems` dataset stored in the **efficient** package.

```
# data available online, from github.com/csgillespie/efficient_pkg
data(ghg_ems, package = "efficient")
names(ghg_ems)
#> [1] "Country"        "Year"           "Electricity"    "Manufacturing"
#> [5] "Transportation" "Other"          "Fugitive"
nrow(ghg_ems)
#> [1] 7896
length(unique(ghg_ems$Country))
#> [1] 188
```

Note that while there are almost 8000 rows, there are fewer than 200 countries.

Note that the `ghg_ems` column names were cleaned using the command `word(names(ghg_ems), sep = " |/")` from the **stringr** package before it was saved as a dataset within the **efficient** package. `" |/"` in this context means "any space or forward slash is a word break". See the Pattern matching section in `vignette("stringr")` for more on this. For details about the `ghg_ems` dataset, see the documentation in `?efficient::ghg_ems`.

To aggregate the dataset using **dplyr** package, you divide the task in two: to *group* the dataset first and then to summarise, as illustrated below.[4]

```
library("dplyr")
group_by(ghg_ems, Country) %>%
  summarise(mean_eco2 = mean(Electricity, na.rm  = TRUE))
#> # A tibble: 188 x 2
#>       Country mean_eco2
#>         <chr>     <dbl>
#> 1 Afghanistan       NaN
#> 2     Albania     0.641
#> 3     Algeria    23.015
#> 4      Angola     0.791
#> # ... with 184 more rows
```

The example above relates to a philosophical question in programming: how much work should one function do? The Unix philosophy states that programs should "do one thing well". And shorter functions are easier to understand and debug. But having too many functions can also make your call stack confusing, and the code hard to maintain. In general, being modular and specific is

[4]The equivalent code in base R is: `e_ems = aggregate(ghg_ems$Electricity, list(ghg_ems$Country), mean, na.rm  = TRUE, data = ghg_ems) nrow(e_ems)`.

advantageous for clarity, and this modular approach is illustrated in the above example with the dual `group_by` and `summarise` stages.

To reinforce the point, this operation is also performed below on the `idata` dataset:

```
data(idata_mutate, package="efficient")
countries = group_by(idata_mutate, Country)
summarise(countries, gini = mean(gini, na.rm  = TRUE))
#> # A tibble: 176 x 2
#>       Country  gini
#>         <chr> <dbl>
#> 1 Afghanistan   NaN
#> 2     Albania  30.4
#> 3     Algeria  37.8
#> 4      Angola  50.6
#> # ... with 172 more rows
```

Note that `summarise` is highly versatile, and can be used to return a customised range of summary statistics:

```
summarise(countries,
  # number of rows per country
  obs = n(),
  med_t10 = median(top10, na.rm  = TRUE),
  # standard deviation
  sdev = sd(gini, na.rm  = TRUE),
  # number with gini > 30
  n30 = sum(gini > 30, na.rm  = TRUE),
  sdn30 = sd(gini[ gini > 30 ], na.rm  = TRUE),
  # range
  dif = max(gini, na.rm  = TRUE) - min(gini, na.rm  = TRUE)
  )
#> # A tibble: 176 x 7
#>       Country   obs med_t10  sdev   n30  sdn30   dif
#>         <chr> <int>   <dbl> <dbl> <int>  <dbl> <dbl>
#> 1 Afghanistan    40      NA   NaN     0     NA    NA
#> 2     Albania    40    24.4  1.25     3  0.364  2.78
#> 3     Algeria    40    29.8  3.44     2  3.437  4.86
#> 4      Angola    40    38.6 11.30     2 11.300 15.98
#> # ... with 172 more rows
```

To showcase the power of `summarise` used on a `grouped_df`, the above code reports a wide range of customised summary statistics *per country*:

- the number of rows in each country group
- standard deviation of gini indices
- median proportion of income earned by the top 10%
- the number of years in which the gini index was greater than 30
- the standard deviation of gini index values over 30
- the range of gini index values reported for each country.

Exercises

1. Refer back to the greenhouse gas emissions example at the outset of section 6.4, in which we found the top 3 countries in terms of emissions growth in the transport sector. a) Explain in words what is going on in each line. b) Try to find the top 3 countries in terms of emissions in 2012 - how is the list different?
2. Referring back to Section 6.4.1, rename the variables 4 to 8 using the **dplyr** function `rename`. Follow the pattern `ECO2`, `MCO2` etc.
3. Explore **dplyr**'s documentation, starting with the introductory vignette, accessed by entering `vignette("introduction")`.
4. Test additional **dplyr** 'verbs' on the `idata` dataset. (More vignette names can be discovered by typing `vignette(package = "dplyr")`.)

6.4.6 Chaining operations

Another interesting feature of **dplyr** is its ability to chain operations together. This overcomes one of the aesthetic issues with R code: you can end end-up with very long commands with many functions nested inside each other to answer relatively simple questions.

> What were, on average, the 5 most unequal years for countries containing the letter g?

Here's how chains work to organise the analysis in a logical step-by-step manner:

```
idata_mutate %>%
  filter(grepl("g", Country)) %>%
  group_by(Year) %>%
  summarise(gini = mean(gini, na.rm  = TRUE)) %>%
  arrange(desc(gini)) %>%
  top_n(n = 5)
#> Selecting by gini
#> # A tibble: 5 x 2
#>    Year  gini
#>   <int> <dbl>
#> 1  1980  46.9
#> 2  1993  46.0
#> 3  2013  44.5
#> 4  1981  43.6
#> # ... with 1 more rows
```

The above function consists of 6 stages, each of which corresponds to a new line and **dplyr** function:

1. Filter-out the countries we're interested in (any selection criteria could be used in place of `grepl("g", Country)`).
2. Group the output by year.
3. Summarise, for each year, the mean gini index.
4. Arrange the results by average gini index
5. Select only the top 5 most unequal years.

To see why this method is preferable to the nested function approach, take a look at the latter. Even after indenting properly it looks terrible and is almost impossible to understand!

```
top_n(
  arrange(
    summarise(
      group_by(
        filter(idata, grepl("g", Country)),
        Year),
      gini = mean(gini, na.rm  = TRUE)),
    desc(gini)),
  n = 5)
```

This section has provided only a taster of what is possible **dplyr** and why it makes sense from code writing and computational efficiency perspectives. For a more detailed account of data processing with R using this approach we recommend *R for Data Science* (Grolemund and Wickham, n.d.).

6.5 Combining datasets

The usefulness of a dataset can sometimes be greatly enhanced by combining it with other data. If we could merge the global `ghg_ems` dataset with geographic data, for example, we could visualise the spatial distribution of climate pollution. For the purposes of this section we join `ghg_ems` to the `world` data provided by **ggmap** to illustrate the concepts and methods of data *joining* (also referred to as merging).

```
library("ggmap")
world = map_data("world")
names(world)
#> [1] "long"      "lat"       "group"     "order"     "region"    "subregion"
```

Visually compare this new dataset of the `world` with `ghg_ems` (e.g. via `View(world); View(ghg_ems)`). It is clear that the column `region` in the former contains the same information as `Country` in the latter. This will be the *joining variable*; renaming it in `world` will make the join more efficient.

```
world = rename(world, Country = region)
ghg_ems$All = rowSums(ghg_ems[3:7])
```

Ensure that both joining variables have the same class (combining `character` and `factor` columns can cause havoc).

How large is the overlap between `ghg_ems$Country` and `world$Country`? We can find out using the `%in%` operator, which finds out how many elements in one vector match those in another vector. Specifically, we will find out how many *unique* country names from `ghg_ems` are present in the `world` dataset:

```
# save the result as 'm', for match
c_u = unique(ghg_ems$Country)
w_u = unique(world$Country)
summary({m = c_u %in% w_u})
#>    Mode   FALSE    TRUE    NA's
#> logical      20     168       0
```

This comparison exercise has been fruitful: most of the countries in the `co2` dataset exist in the `world` dataset. But what about the 20 country names that do not match? We can identify these as follows:

```
(n = c_u[!m]) # non-matching country names in world data
#>  [1] "Antigua & Barbuda"           "Bahamas, The"
#>  [3] "Bosnia & Herzegovina"        "Congo, Dem. Rep."
#>  [5] "Congo, Rep."                 "Cote d'Ivoire"
#>  [7] "European Union (15)"         "European Union (28)"
#>  [9] "Gambia, The"                 "Korea, Dem. Rep. (North)"
#> [11] "Korea, Rep. (South)"         "Macedonia, FYR"
#> [13] "Russian Federation"          "Saint Kitts & Nevis"
#> [15] "Saint Vincent & Grenadines" "Sao Tome & Principe"
#> [17] "Trinidad & Tobago"           "United Kingdom"
#> [19] "United States"               "World"
```

It is clear from the output that some of the non-matches (e.g. the European Union) are not countries at all. However, others, such as 'Gambia, The' and the United States clearly should have matches. *Fuzzy matching* can help to rectify this, as illustrated the first non-matching country below:

```
grep(n[1], w_u) # no match
#> integer(0)
(fm = agrep(n[1], w_u, max.distance = 10))
#> [1] 15 16
(w_u1 = w_u[fm])
#> [1] "Antigua" "Barbuda"
```

What just happened? We verified that first unmatching country in the `ghg_ems` dataset was not in the `world` country names with a `grep`. So we used the more powerful `agrep` to search for fuzzy matches (with the `max.distance` argument set to `10` after trial and error). The results show that the country `Antigua & Barbuda` from the `ghg_ems` data is matched *two* countries in the `world` dataset. We can update the names in the dataset we are joining to accordingly:

```
world$Country[world$Country %in% w_u1] = n[1]
```

Fuzzy matching is still a laborious process that must be complemented by human judgement. It takes a human to know for sure that `United States` is represented as `USA` in the `world` dataset, without risking false matches via `agrep`. This can be fixed manually:

```
world$Country[world$Country == "USA"] = "United States"
```

To fix the remaining issues, we simply continued with the same method, using a `for` loop and verifying the results instead of doing all by hand. The code used to match the remaining unmatched countries can be seen on the book's GitHub page.

There is one more stage that is needed before global CO^2 emissions can be mapped for any year: the data must be *joined*. The base function `merge` can do this but we strongly recommend using one of the `join` functions from **dplyr**, such as `left_join` (which keeps all rows in the original dataset) and `inner_join` (which keeps only rows with matches in both datasets), as illustrated below:

```
nrow({world_co2 = left_join(world, ghg_ems)})
#> Joining, by = "Country"
#> [1] 3330794
nrow(inner_join(world, ghg_ems))
#> Joining, by = "Country"
#> [1] 3310272
```

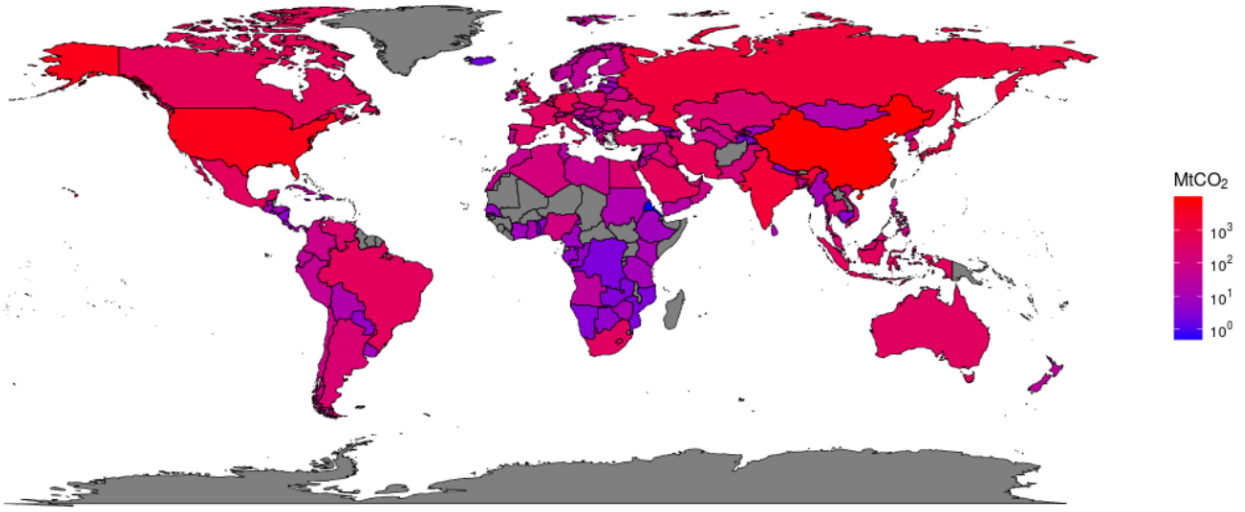


Figure 6.1: The geographical distribution of carbon dioxide emissions in 2012.

Note that `inner_join` removes rows from the `world` dataset which have no match in `ghg_ems`: if we were to plot the resulting dataset, the continent of Antarctica and a number of countries not represented in the `ghg_ems` dataset would be absent. Figure 6.1 shows the results of this data carpentry, produced using a modified version of the **ggplot2** code below, were worth the effort.

```
world_co2_2012 = filter(world_co2, Year == 2012 | is.na(Year))
ggplot(world_co2_2012, aes(long, lat)) +
  geom_polygon(aes(fill = All, group = group))
```

6.6 Working with databases

Instead of loading all the data into RAM, as R does, databases query data from the hard-disk. This can allow a subset of a very large dataset to be defined and read into R quickly, without having to load it first. R can connect to databases in a number of ways, which are briefly touched on below. Databases is a large subject area undergoing rapid evolution. Rather than aiming at comprehensive coverage, we will provide pointers to developments that enable efficient access to a wide range of database types. An up-to-date history of R's interfaces to databases can be found in README of the **DBI** package, which provides a common interface and set of classes for driver packages (such as **RSQLite**).

RODBC is a veteran package for querying external databases from within R, using the Open Database Connectivity (ODBC) API. The functionality of **RODBC** is described in the package's vignette (see `vignette("RODBC")`). **RODBC** connects to 'traditional' databases such as MySQL, PostgreSQL, Oracle and SQLite. Since then, the **DBI** package has created a unified structure for accessing databases allowing for other drivers to be added as modular packages. Thus new packages that build on **DBI** can be seen as a replacement of **RODBC** (**RMySQL**, **RPostgreSQL**, and **RSQLite**) (see `vignette("backend")` for more on how **DBI** drivers work). Because the **DBI** syntax applies to a wide range of database types we use it here with a worked example.

Imagine you have access to a database that contains the `ghg_ems` data set.

```
# Connect to a database driver
library("RSQLite")
```

```
con = dbConnect(SQLite(), dbname = ghg_db) # Also username & password arguments
dbListTables(con)
rs = dbSendQuery(con, "SELECT * FROM `ghg_ems` WHERE (`Country` != 'World')")
df_head = dbFetch(rs, n = 6) # extract first 6 row
```

The above code chunk shows how the function `dbConnect` connects to an external database, in this case a MySQL database. The `username` and `password` arguments are used to establish the connection. Next we query which tables are available with `dbListTables`, query the database (without yet extracting the results to R) with `dbSendQuery` and, finally, load the results into R with `dbFetch`.

Be sure never to release your password by entering it directly into the command. Instead, we recommend saving sensitive information such as database passwords and API keys in `.Renviron`, described in Chapter 2. Assuming you had saved your password as the environment variable `PSWRD`, you could enter `pwd = Sys.getenv("PSWRD")` to minimise the risk of exposing your password through accidentally releasing the code or your session history.

Recently there has been a shift to the 'noSQL' approach for storing large datasets. This is illustrated by the emergence and uptake of software such as MongoDB and Apache Cassandra, which have R interfaces via packages mongolite and RJDBC, which can connect to Apache Cassandra data stores and any source compliant with the Java Database Connectivity (JDBC) API.

MonetDB is a recent alternative to traditional and noSQL approaches which offers substantial efficiency advantages for handling large datasets (Kersten et al. 2011). A tutorial on the MonetDB website provides an excellent introduction to handling databases from within R.

There are many wider considerations in relation to databases that we will not cover here: who will manage and maintain the database? How will it be backed up locally (local copies should be stored to reduce reliance on the network)? What is the appropriate database for your project. These issues can have major efficiency, especially on large, data intensive projects. However, we will not cover them here because it strays too far out of the R ecosystem and into the universe of databases for the purposes of this book. Instead, we direct the interested reader towards further resources on the subject, including:

- db-engines.com/en/: a website comparing the relative merits of different databases.
- The `databases` vignette from the **dplyr** package.
- Getting started with MongoDB in R, an introductory vignette on non-relational databases and map reduce from the **mongolite** package.

6.6.1 Databases and dplyr

To access a database in R via **dplyr**, one must use one of the `src_` functions to create a source. Continuing with the SQLite example above, one would create a `tbl` object, that can be queried by **dplyr** as follows:

```
library("dplyr")
ghg_db = src_sqlite(ghg_db)
ghg_tbl = tbl(ghg_db, "ghg_ems")
```

The `ghg_tbl` object can then be queried in a similar way as a standard data frame. For example, suppose we wished to filter by `Country`. Then we use the `filter` function as before

```
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** use lazy evaluation. The SQL command associated with `rm_world` hasn't yet been executed, this is why `tail(rm_world)` doesn't work. By using lazy evaluation, **dplyr** is more efficient at handling large data structures since it avoids unnecessary copying. When you want your SQL command to be executed, use `collect(rm_world)`.

Exercises

Follow the worked example below to create and query a database on land prices in the UK using **dplyr** as a front end to an SQLite database. The first stage is to read-in the data:

```
# See help("land_df", package="efficient") for detials
data(land_df, package="efficient")
```

The next stage is to create an SQLite database to hold the data:

```
# install.packages("RSQLite") # Requires RSQLite package
my_db = src_sqlite("land.sqlite3", create = TRUE)
land_sqlite = copy_to(my_db, land_df, indexes = list("trnsctnuni", "dtoftrnsfr"))
class(land_sqlite)
#> [1] "tbl_sqlite" "tbl_sql"    "tbl_lazy"   "tbl"
```

From the above code we can see that we have created a `tbl`. This can be accessed using **dplyr** in the same way as any data frame can. Now we can query the data. You can use SQL code to query the database directly or use standard **dplyr** verbs on the table.

```
# Method 1: using sql
tbl(my_db, sql('SELECT "price", "dtoftrnsfr", "postcode"  FROM land_df'))
#> Source:   query [?? x 3]
#> Database: sqlite 3.8.6 [land.sqlite3]
#>
#>    price dtoftrnsfr postcode
#>    <int>      <dbl>    <chr>
#> 1  84000   1.15e+09  CW9 5EU
#> 2 123500   1.14e+09 TR13 8JH
#> 3 217950   1.17e+09 PL33 9DL
#> 4 147000   1.16e+09 EX39 5XT
#> # ... with more rows

# Method 2: using dplyr
select(land_sqlite, price, dtoftrnsfr, postcode, proprtytyp)
#> Source:   query [?? x 4]
#> Database: sqlite 3.8.6 [land.sqlite3]
#>
#>    price dtoftrnsfr postcode proprtytyp
#>    <int>      <dbl>    <chr>      <chr>
#> 1  84000   1.15e+09  CW9 5EU          F
#> 2 123500   1.14e+09 TR13 8JH          T
```

```
#> 3 217950   1.17e+09 PL33 9DL          D
#> 4 147000   1.16e+09 EX39 5XT          T
#> # ... with more rows
```

6.7 Data processing with data.table

data.table is a mature package for fast data processing that presents an alternative to **dplyr**. There is some controversy about which is more appropriate for different tasks[5] so it should be stated at the outset that which to use can be a matter of personal preference. Both are powerful and efficient packages that take time to learn, so it is best to learn one and stick with it, rather than have the duality of using two for similar purposes. There are situations in which one works better than another: **dply** provides a more consistent and flexible interface (e.g. with its interface to databases) so for most purposes we recommend **dplyr**. However, **data.table** has a few features that make it very fast for some operations that it is worth at least being aware of from an efficiency perspective.

This section provides a few examples to illustrate how **data.table** differs and (at the risk of inflaming the debate further) some benchmarks to explore which is more efficient. As emphasised throughout the book, efficient code writing is often more important than efficient execution on many everyday tasks so to some extent it's a matter of preference.

The foundational object class of **data.table** is the `data.table`. Like **dplyr**'s `tbl_df`, **data.table**'s `data.table` objects behave in the same was as the base `data.frame` class. However the **data.table** paradigm has some unique features that make it highly computationally efficient for many common tasks in data analysis. Building on subsetting methods using `[` and `filter()` presented in Section 6.4.4, we'll see **data.tables**'s unique approach to subsetting. Like base R **data.table** uses square brackets but you do not need to refer to the object name inside the brackets:

```
library("data.table")
idata = readRDS("data/idata-renamed.Rds")
idata_dt = data.table(idata) # convert to data.table class
aus3a = idata_dt[Country == "Australia"]
```

To boost performance, one can set 'keys'. These are 'supercharged rownames' which order the table based on one or more variables. This allows a *binary search* algorithm to subset the rows of interest, which is much, much faster than the *vector scan* approach used in base R (see `vignette("datatable-keys-fast-subset")`). **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 below).

```
setkey(idata_dt, Country)
aus3b = idata_dt[.("Australia")]
```

The result is the same, so why add the extra stage of setting the key? The reason is that this one-off sorting operation can lead to substantial performance gains in situations where repeatedly subsetting rows on large datasets consumes a large proportion of computational time in your workflow. This is illustrated in Figure 6.2, which compares 4 methods of subsetting incrementally larger versions of the `idata` dataset.

Figure 6.2 demonstrates that **data.table** is *much faster* than base R and **dplyr** at subsetting. As with using external packages to read in data (see Section 5.3), the relative benefits of **data.table** improve with dataset size, approaching a ~70 fold improvement on base R and a ~50 fold improvement on **dplyr** as the dataset size reaches half a Gigabyte. Interestingly, even the 'non key' implementation of **data.table** subset

[5] One question on the stackoverflow website titled 'data.table vs dplyr' illustrates this controversy and delves into the philosophy underlying each approach.

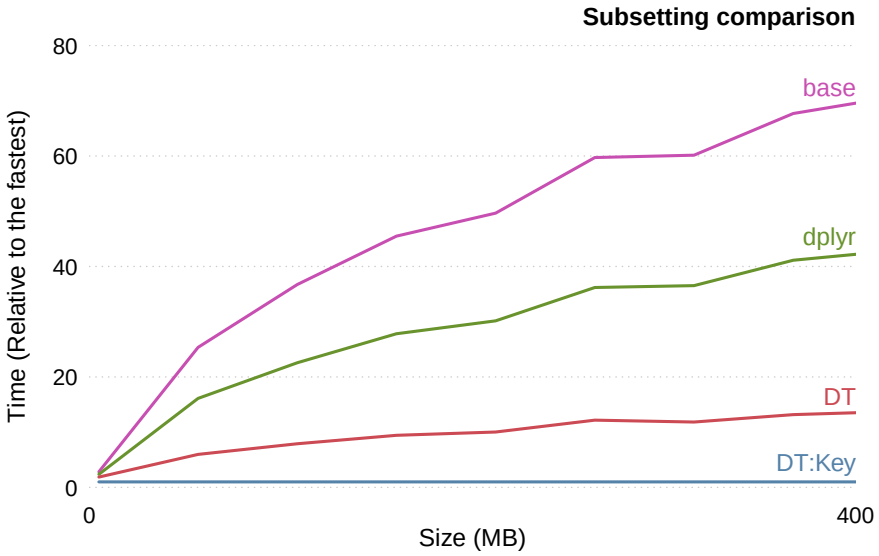


Figure 6.2: Benchmark illustrating the performance gains to be expected for different dataset sizes.

method is faster than the alternatives: this is because **data.table** creates a key internally by default before subsetting. The process of creating the key accounts for the ~10 fold speed-up in cases where the key has been pre-generated.

In summary, this section has introduced **data.table** as a complimentary approach to base and **dplyr** methods for data processing. It offers performance gains due to its implementation in C and use of *keys* for subsetting tables. **data.table** offers much more, however, including: highly efficient data reshaping; dataset merging (also known as joining, as with `left_join` in **dplyr**); and grouping. For further information on **data.table**, we recommend reading the package's `datatable-intro`, `datatable-reshape` and `datatable-reference-semantics` vignettes.

Chapter 7

Efficient performance

Donald Knuth is a famous American computer scientist who developed a number of the key alogorithms that we use today. On the subject of optimisation he give this advice.

> 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.

That's why in the previous chapters we've focused on tools and your working environment to increase efficiency. These are (relatively) easy ways of saving time, that once implemented, work for future projects. In this chapter we are going to consider code optimisation. Simply stated, making your program run faster.

In many scernios, code is running slowly due to a couple of key bottlenecks. By specifically targetting these key areas, we may be able to speed up the entire program.

In this chapter we will consider a variety of code tweaks that can eke out an extra speed boost. We'll begin with general hints and tips about using base R. Then look at code profiling to determine where our efforts should be focused. We conclude by looking at `Rcpp`, an efficient way of incorparting C++ code into an R analysis.

7.1 Top 5 tips for efficient performance

- Before you start to optimise you code, ensure you know where the bottleneck lies; use a code profiler.
- The `ifelse` function is optimal for vectorised comparisons.
- If the data in your data frame is all of the same type, consider converting it to a matrix for a speed boost.
- The **parallel** package is ideal for Monte-Carlo simulations.
- For optimal performance, consider re-writing key parts of your code in C++.

7.2 Efficient base R

In R there is often more than one way to solve a problem. In this section we highlight standard tricks or alternative methods that may improve performance.

The `if` vs `ifelse` functions

The `ifelse` function

```
ifelse(test, yes, no)
```

is a vectorised version of the standard control-flow `if` function. The return value of `ifelse` is filled with elements from the `yes` and `no` arguments that are determined by whether the element of `test` is `TRUE` or `FALSE`.

If the length of `test` is equal to 1, i.e. `length(test) == 1`, then the standard `if(test) yes else no` is more efficient.

Sorting and ordering

Sorting a vector a single vector is relatively quick; sorting a vector of length 10^7 takes around 0.01 seconds. If you only sort a vector once at the top of a script, then don't worry too much about this. However if you are sorting inside a loop, or in a shiny application, then it can be worthwhile thinking about how to optimise this operation.

There are currently three sorting algorithms, `c("shell", "quick", "radix")` that can be specified in the `sort` function; with `radix` being a new addition to R 3.3. Typically the `radix` (the non-default option) is optimal for most situations.

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)`.

Reversing elements

The `rev` function provides a reversed version of its argument. If you wish to sort in increasing order, then use the more efficient `sort(x, decreasing=TRUE)` instead of `rev(sort(x))`.

Which indices are `TRUE`

To determine which index of a vector or array are `TRUE`, we would typically use the `which` function. If we want to find the index of just the minimum or maximum value, i.e. `which(x == min(x))` then use the more efficient `which.min`/`which.max` variants.

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]
```

where `f` is the factor.

String concatenation

To concatenate strings use the `paste` function

```
paste("A", "B")
```

The separation character is specified via the `sep` argument. The function `paste0(..., collapse)` is equivalent to `paste(..., sep = "", collapse)`, but is slightly more efficient.

Logical AND and OR

The logical AND (`&`) and OR (`|`) operators are vectorised functions and are typically used whenever we perform subsetting operations. For example

```
x < 0.4 | x > 0.6
#> [1]  TRUE FALSE  TRUE
```

When R executes the above comparison, it will **always** calculate `x > 0.6` regardless of the value of `x < 0.4`. In contrast, the non-vectorised version, `&&`, only executes the second component if needed. This is efficient and leads to more neater code, e.g.

```
# read.csv is only executed if the file exists
file.exists("data.csv") && read.csv("data.csv")
#> [1] FALSE
```

However care must be taken not to use `&&` or `||` on vectors since it will give the incorrect answer

```
x < 0.4 || x > 0.6
#> [1] TRUE
```

Row and column operations

In data analysis we often want to apply a function to each column or row of a data set. For example, we might want to calculation the column or row sums. The `apply` function makes this type of operation straightforward.

```
# Second argument: 1 -> rows. 2 -> columns
apply(data_set, 1, function_name)
```

There are optimised functions for calculating row and columns sums/means, `rowSums`, `colSums`, `rowMeans` and `colMeans` that should be used whenever possible.

`is.na` and `anyNA`

To test whether a vector (or other obeject) contains missing values we use the `is.na` function. Often we are interested in whether a vector contains *any* missing values. In this case, `anyNA(x)` is usually more efficient than `any(is.na(x))`.

Matrices

A matrix is similar to a data frame: it is a two dimensional object and sub-setting and other functions work in the expected way. However all matrix elements must have the same type. Matrices tend to be used during statistical calculations. Calculating the line of best fit using the `lm()` function, internally converts the data to a matrix before calculating the results; any characters are thus recoded as numeric dummy variables.

Matrices are generally faster than data frames. For example, the datasets `ex_mat` and `ex_df` from the **efficient** package each have 1000 rows and 100 columns and contain the same random numbers. However selecting rows from a data frame is around 150 times slower than 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, ] 0.00312 0.0037 0.055 0.00615 0.00676 4.95   100
#>   ex_df[1, ] 0.74312 0.8252 1.092 0.87096 1.15442 7.72   100
```

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

The integer data type

Numbers in R are usually stored in double-precision floating-point format - see Braun and Murdoch (2007) and Goldberg (1991). The term 'double' refers to the fact that on 32 bit systems (for which the format was developed) two memory locations are used to store a single number. Each double-precision number is accurate to around 17 decimal places.

When comparing floating point numbers we should be particularly careful, since `y = sqrt(2)*sqrt(2)` is not exactly 2, instead it's **almost** 2. Using `sprintf("%.17f", y)` will give you the true value of `y` (to 17 decimal places).

There is also another data type, called an integer. Integers primarily exist to be passed to C or Fortran code. Typically we don't worry about creating integers, however they are occasionally used to optimise sub-setting operations. When we subset a data frame or matrix, we are interacting with C code. For example, if we look at the arguments for the `head` function

```
args(head.matrix)
#> function (x, n = 6L, ...)
#> NULL
```

Using the `:` operator automatically creates a vector of integers.

The default argument for `n` is `6L` (the `L` is short for Literal and is used to create an integer). Since this function is being called by almost everyone that uses R, this low level optimisation is useful. To illustrate the speed increase, suppose we are selecting the first 100 rows from a data frame (`clock_speed`, from the **efficient** package). The speed increase is illustrated below, using the **microbenchmark** package:

```
s_int = 1:100; s_double = seq(1, 100, 1.0)
microbenchmark(clock_speed[s_int, 2L], clock_speed[s_double, 2.0], times=1000000)
#> Unit: microseconds
#> expr   min    lq  mean median    uq   max neval cld
#> clock_speed[s_int, 2L] 11.79 13.43 15.30  13.81 14.22 87979 1e+06  a
#> clock_speed[s_double, 2] 12.79 14.37 16.04  14.76 15.18 21964 1e+06   b
```

The above result shows that using integers is slightly faster, but probably not worth worrying about.

Integers are more space efficient. If we compare size of a integer vector to a standard numeric vector

```
pryr::object_size(1:10000)
#> 40 kB
pryr::object_size(y = seq(1, 10000, by=1.0))
#> 80 kB
```

The integer version is (approximately) half the size. However most mathematical operations will convert the integer vector into a standard numerical vector, e.g.

```
is.integer(1L + 1)
#> [1] FALSE
```

Sparse matrices

Another efficient data structure is a sparse matrix. This is simply 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. Large sparse matrices often crop up when performing numerical calculations. Typically, our data isn't sparse but the resulting data structures we create may be sparse. There are a number of techniques/methods used to store sparse matrices. Methods for creating sparse matrices can be found in the **Matrix** package[1].

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

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

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. This is much more memory efficient

```
pryr::object_size(sp)
#> 161 kB
pryr::object_size(m)
#> 800 MB
```

[1] Technically this isn't in base R, it's a recommend package.

Exercises

1. Create a vector `x`. Benchmark `any(is.na(x))` against `anyNA`. Do the results vary with the size of the vector.

2. Examine the following function definitions to give you an idea of how integers are used.

   - `tail.matrix`
   - `lm`.

3. Contruct a matrix of integers and a matrix of numerics. Using `pryr::object_size`, compare the objects.

4. How does the function `seq.int`, which was used in the `tail.matrix` function, differ to the standard `seq` function?

A related memory saving idea is to replace `logical` vectors with vectors from the **bit** package which take up just over 16th of the space (but you can't use `NA`s).

7.3 Code profiling

A circumstance that happens all too often, is that we simply want our code to run faster. In some cases it's obvious where the bottle neck lies. Other times we have to rely on our intuition. One major drawback of relying on our intuition is that we could be wrong and we end up pointlessly optimising the wrong piece of code. To make slow code run faster, we first need to determine where the slow code lives.

The `Rprof` function is built-in tool for profiling the execution of R expressions. At regular time intervals, the profiler stops the R interpreter, records the current function call stack, and saves the information to a file. The results from `Rprof` are stochastic. Each time we run a function R, the conditions have changed. Hence, each time you profile your code, the result will be slightly different.

Unfortunately `Rprof` is not user friendly. The **profvis** package provides an interactive graphical interface for visualising data data from `Rprof`.

7.3.1 Getting started with profvis

The **profvis** package can be installed in the usual way

```
install.packages("profvis")
```

As a simple example, we will use the `movies` data set, which contain information on around 60,000 movies from the IMDB. First, we'll select movies that are classed as comedies, then plot year vs movies rating, and draw a local polynomial regression line to pick out the trend. The main function from the **profvis** package, is `profvis` which profiles the code and creates an interactive HTML page. The first argument of `profvis` is the R expression of interest.

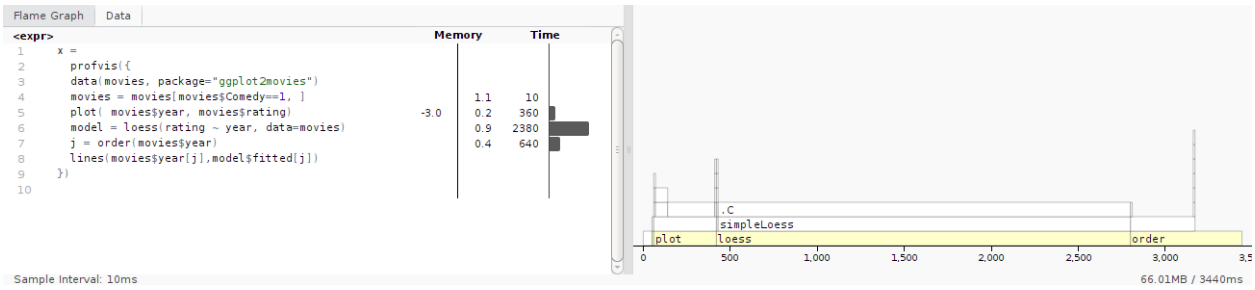


Figure 7.1: Output from profvis

```
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
})
```

The above code provides an interactive HTML page (figure 7.1). On the left side is the code and on the right is a flame graph (horizontal direction is time in milliseconds and the vertical direction is the call stack).

The left hand panel gives the amount of time spent on each line of code. We see that majority of time is spent calculating the `loess` smoothing line. The bottom line of the right panel also highlights that most of the execution time is spent on the `loess` function. Travelling up the function, we see that `loess` calls `simpleLoess` which in turn calls `.C` function.

The conclusion from this graph is that if optimisation were required, it would make sense to focus on the `loess` and possibly the `order` function calls.

7.3.2 Example: Monopoly Simulation

Monopoly is a board game that originated in the United States over 100 years ago. The object of the game is to go round the board and purchase squares (properties). If other players land on your properties they have to pay a tax. The player with the most money at the end of the game, wins. To make things more interesting, there are Chance and Community Chest squares. If you land on one of these squares, you draw card, which may send to you to other parts of the board. The other special square, is Jail. One way of entering Jail is to roll three successive doubles.

The **efficient** package contains a Monte-Carlo function for simulating a simplified game of monopoly. By keeping track of where a person lands when going round the board, we obtain an estimate of the probability of landing on a certain square. The entire code is around 100 lines long. In order for `profvis` to fully profile the code, the **efficient** package needs to be installed from source

```
# args is also a valid argument for install.packages
devtools::install_github("csgillespie/efficient_pkg", args="--with-keep.source")
```

The function can then be profiled via

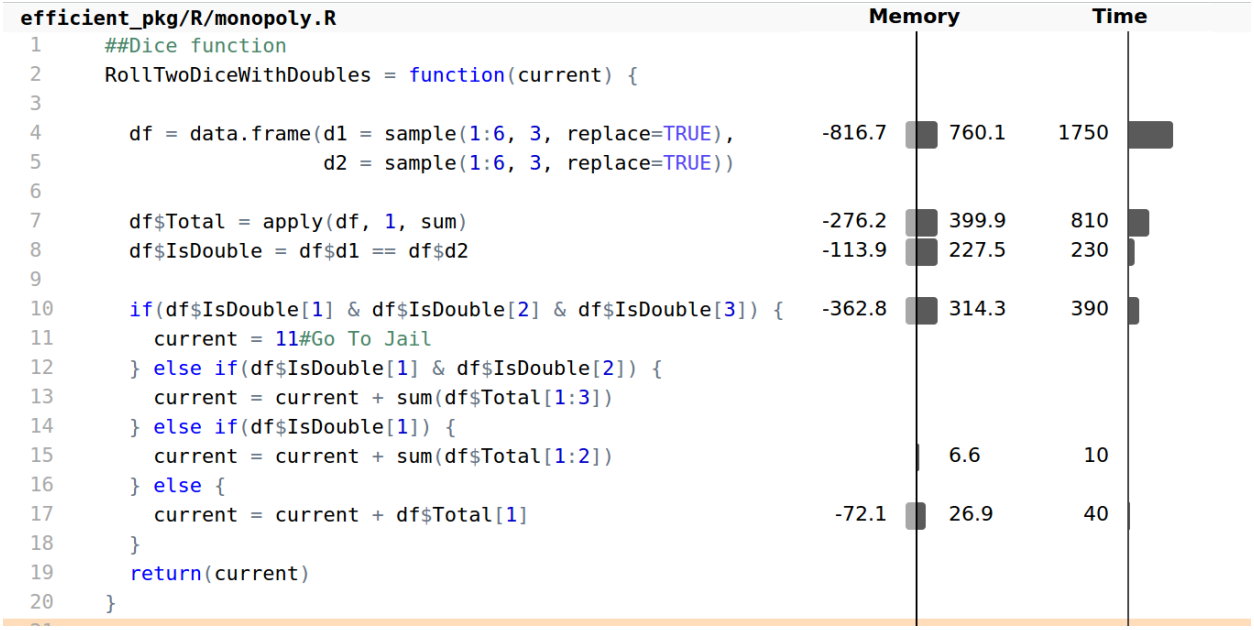


Figure 7.2: Code profiling for simulating the game of Monopoly.

```
library("efficient")
profvis(SimulateMonopoly(10000))
```

to get 7.2

The output from `profvis` shows that the vast majority of time is spent in the `move` function. In Monopoly rolling a double (a pair of 1's, 2's, ..., 6's) is special:

- Roll two dice (`total1`): `total_score = total1`;
- If you get a double, roll again (`total2`) and `total_score = total1 + total2`;
- If you get a double, roll again (`total3`) and `total_score = total1 + total2 + total3`;
- If roll three is a double, Go To Jail, otherwise move `total_score`.

The `move` function spends around 50% of the time creating a data frame, 25% time calculating row sums, and the remainder on a comparison operations. This piece of code can be optimised fairly easily (will still retaining the same overall structure) by incorporating the following improvements[2]:

- Use a matrix instead of a data frame;
- Sample 6 numbers at once, instead of two groups of three;
- Switch from `apply` to `rowSums`;
- Use `&&` in the `if` condition.

Implementing this features, results in the following code.

```
move = function(current) {
  # data.frame -> matrix
  rolls = matrix(sample(1:6, 6, replace=TRUE), ncol=2)
  Total = rowSums(rolls)    # apply -> rowSums
```

[2]Solutions are available in the **efficient** package vignette.

```
  IsDouble = rolls[,1] == rolls[,2]
  # & -> &&
  if(IsDouble[1] && IsDouble[2] && IsDouble[3]) {
    current = 11#Go To Jail
  } else if(IsDouble[1] && IsDouble[2]) {
    current = current + sum(Total[1:3])
  } else if(IsDouble[1]) {
    current = current + Total[1:2]
  } else {
    current = Total[1]
  }
  current
}
```

which gives a 25-fold speed improvement.

Exercise

The `move` function above uses a vectorised solution. Whenever we move, we always roll six dice, then examine the outcome and determine the number of doubles. However, this is potentially wasteful, since the probability of getting one double is 1/6 and two doubles is 1/36. Another method is too only roll additional dice if and when they are needed. Implement and time this solution.

7.4 Parallel computing

This chapter provides a brief foray into the word of parallel computing and only looks at shared memory systems. The idea is to give a flavour of what is possible, instead of covering all possible varities. For a fuller account, see McCallum and Weston (2011).

In recent R versions (since R 2.14.0) the **parallel** package comes pre-installed with base R. The **parallel** package must still be loaded before use however, and you must determine the number of available cores manually, as illustrated below.

```
library("parallel")
no_of_cores = detectCores()
```

7.4.1 Parallel versions of apply functions

The most commonly used parallel applications are parallelised replacements of `lapply`, `sapply` and `apply`. The parallel implementations and their arguments are shown below.

```
parLapply(cl, x, FUN, ...)
parApply(cl = NULL, X, MARGIN, FUN, ...)
parSapply(cl = NULL, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

The key point is that there is very little difference in arguments between `parLapply` and `apply`, so the barrier to using (this form) of parallel computing is low. Each function above has an argument `cl`, which is created by a `makeCluster` call. This function, amongst other things, specifies the number of processors to use.

7.4.2 Example: Snakes and Ladders

Parallel computing is ideal for Monte-Carlo simulations. Each core independently simulates a realisation from model. At the end, we gather up the results. In the **efficient** package, there is a function that simulates a single game of Snakes and Ladders - `snakes_ladders()`[3]

If we wanted to simulate `N` games we could use `sapply`

```
N = 10^4
sapply(1:N, snakes_ladders)
```

Rewriting this code to make use of the **parallel** package is straightforward. We begin by making a cluster object

```
library("parallel")
cl = makeCluster(4)
```

Then swap `sapply` for `parSapply`

```
parSapply(cl, 1:N, snakes_ladders)
```

before stopping the cluster

```
stopCluster(cl)
```

On my computer I get a four-fold speed-up.

7.4.3 Exit functions with care

We should always call `stopCluster` to free resources when we are finished with the cluster object. However if the parallel code is within function, it's possible that function ends as the results of an error and so `stopCluster` is ommitted.

The `on.exit` function handles this problem with the minimum of fuss; regardless how the function ends, `on.exit` is always called. In the context of parallel programming we will have something similar to

```
simulate = function(cores) {
  cl = makeCluster(cores)
  on.exit(stopCluster(cl))
  # Do something
}
```

Another common use of `on.exit` is with the `par` function. If you use `par` to change graphical parameters within a function, `on.exit` ensures these parameters are reset to their previous value when the function ends.

[3]The idea for this example came to one of the authors after a particularly long and dull game of Snakes and Ladders with his son.

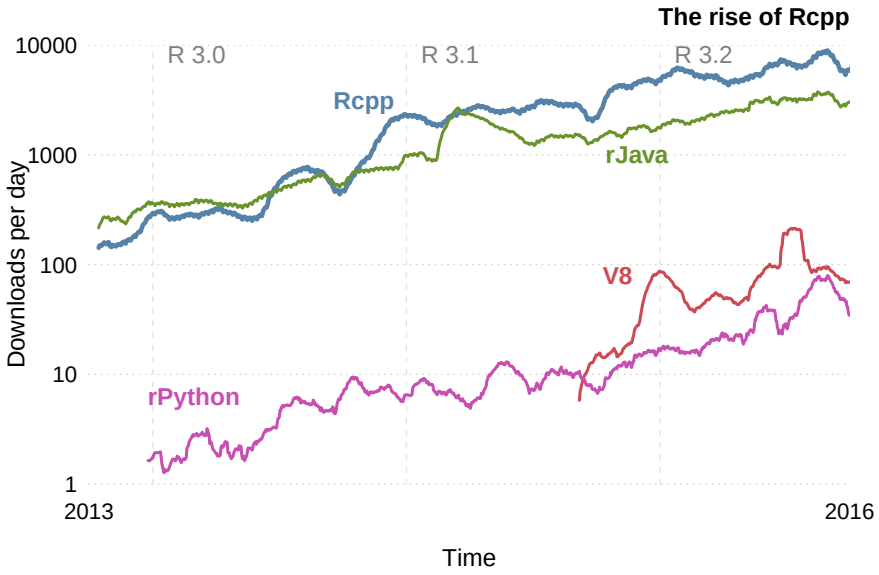


Figure 7.3: Downloads per day from the RStudio CRAN mirror of packages that provide R interfaces to other languages.

7.4.4 Process forking

Another way of running code in parallel is to use the `mclapply` and `mcmapply` functions, i.e.

```
# This will run on Windows, but will only use 1 core
mclapply(1:2, snakes_ladders)
```

These functions use forking, that is creating a new copy of a process running on the CPU. However Windows does not support this low-level functionality in the way that Linux does. If I'm writing code that I don't intend to share, I use `mclapply` since it is more efficient.

7.5 Rcpp

Sometimes 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** is the most popular of these (figure 7.3).

C++ is a modern, fast and very well-supported language with libraries for performing many kinds of computational task. **Rcpp** makes incorporating C++ code into your R workflow easily.

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. Typical bottlenecks that C++ addresses are loops and recursive functions.

C++ is a powerful programming language about which entire books have been written. This section therefore is focussed on getting started and provide a flavour of what is possible. It is structured as follows. After ensuring that your computer is set-up for **Rcpp** in Section 7.5.1, we proceed by creating a simple C++ function, to show how C++ compares with R (Section 7.5.2). This is converted into an R function using `cppFunction()` in Section 7.5.3. The remainder of the chapter explains C++ data types (Section 7.5.4),

illustrates how to source C++ code directly (Section 7.5.5) and explains vectors (Section 7.5.6) **Rcpp** sugar (Section 7.5.7) and finally provides guidance on further resources on the subject (Section 7.5.8).

7.5.1 Pre-requisites

To write and compile C++ functions, you need a working C++ compiler. The installation method depends on your operating system:

- 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.

The code in this chapter was generated using version 0.12.5 of **Rcpp**. The latest version can be installed from CRAN:

```
install.packages("Rcpp")
```

Rcpp is well documented, as illustrated by the number of vignettes on the package's CRAN page. In addition to its popularity, many other packages depend on **Rcpp**, which can be seen by looking at the `Reverse Imports` section.

To check that you have everything needed for this chapter, run the following piece of code from the course R package:

```
efficient::test_rcpp()
#> Everything seems fine
```

7.5.2 A simple C++ function

A C++ function is similar to an R function: you pass a set of inputs to function, some code is run, a single object is returned. However 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 `/*...*/`

Suppose we want to create a function that adds two numbers together. In R this would be a simple one line affair:

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

In C++ it is a bit more long winded

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

If we were writing a C++ programme we would also need another function called `main`. We would then compile the code to obtain an executable. The executable is platform dependent. The beauty of using **Rcpp** is that it makes it very easy to call C++ functions from R and the user doesn't have to worry about the platform, or compilers or the R/C++ interface.

7.5.3 The cppFunction command

Load the **Rcpp** package using the usual `library` function call:

```
library("Rcpp")
```

Pass the C++ function created in the previous section as a text string argument to `cppFunction`:

```
cppFunction('
  double add_c(double x, double y) {
    double value = x + y;
    return value;
  }
')
```

Rcpp will magically compile the C++ code and construct a function that bridges the gap between R and C++. After running the above code, we now have access to the `add_c` function

```
add_c
#> function (x, y)
#> .Primitive(".Call")(<pointer: 0x2b97613f68a0>, x, y)
```

and can call the `add_c` function in the usual way

```
add_c(1, 2)
#> [1] 3
```

We don't have to worry about compilers. Also, if you include this function in a package, users don't have to worry about any of the **Rcpp** magic. It just works.

7.5.4 C++ data types

The most basic type of variable is an integer, `int`. An `int` variable can store a value in the range -32768 to $+32767$. To store floating point numbers, there are single precision numbers, `float` and double precision numbers, `double`. A `double` takes twice as much memory as a `float`. For **single** characters, we use the `char` data type.

Table 7.1: Overview of key C++ object types.

Type	Description
char	A single character.
int	An integer.
float	A single precision floating point number.
double	A double-precision floating point number.
void	A valueless quantity.

There is also something called an unsigned int, which goes from 0 to 65,535 and a `long int` that ranges from 0 to $2^{31} - 1$.

A pointer object is a variable that points to an area of memory that has been given a name. Pointers are a very powerful, but primitive facility contained in the C++ language. They are very useful since rather than passing large objects around, we pass a pointer to the memory location; rather than pass the house, we just give the address. We won't use pointers in this chapter, but mention them for completeness. Table @ref{tab:cpptypes} gives an overview.

7.5.5 The `sourceCpp` function

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. However we need to include a few headers at the top of the file. The first line we add gives us access to the **Rcpp** functions. The file `Rcpp.h` contains a list of function and class definitions supplied by **Rcpp**. This file will be located where **Rcpp** is installed. The `include` line

```
#include <Rcpp.h>
```

causes the compiler to replace that lines with the contents of the named source file. This means that we can access the functions defined by **Rcpp**. To access the **Rcpp** functions we would have to type `Rcpp::function_1`. To avoid typing `Rcpp::`, we use the namespace facility

```
using namespace Rcpp;
```

Now we can just type `function_1`; this is the same concept that R uses for managing function name collisions when loading packages. Above each function we want to export/use in R, we add the tag

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

This would give the complete file

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

// [[Rcpp::export]]
double add_c(double x, double y) {
```

```
  double value = x + y;
  return value;
}
```

There are two main benefits with putting your C++ functions in separate files. First, we have the benefit of syntax highlighting (RStudio has great support for C++ editing). Second, it's easier to make syntax errors when the switching between R and C++ in the same file. To save space we we'll omit the headers for the remainder of the chapter.

7.5.6 Vectors and loops

Let's now consider a slightly more complicated example. Here we want to write our own function that calculates the mean. This is just an illustrative example: R's version is much better and more robust to scale differences in our data. For comparison, let's create a corresponding R function - this is the same function we used in chapter 3. The function takes a single vector `x` as input, and returns the mean value, `m`:

```
mean_r = function(x) {
  m = 0
  n = length(x)
  for(i in 1:n)
    m = m + x[i]/n
  m
}
```

This is a very bad R function; we should just use the base function `mean` for real world applications. However the purpose of `mean_r` is to provide a comparison for the C++ version, which we will write in a similar way.

In this example, we will let **Rcpp** smooth the interface between C++ and R by using the `NumericVector` data type. This **Rcpp** data type mirrors the R vector object type. Other common classes are: `IntegerVector`, `CharacterVector`, and `LogicalVector`.

In the C++ version of the mean function, we specify the arguments types: `x` (`NumericVector`) and the return value (`double`). The C++ version of the `mean` function is a few lines longer. Almost always, the corresponding C++ version will be, possibly much, longer.

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

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

To use the C++ function we need to source the file (remember to put the necessary headers in).

```
sourceCpp("src/mean_c.cpp")
```

Although the C++ version is similar, there are a few crucial differences.

1. We use the `.size()` method to find the length of `x`.

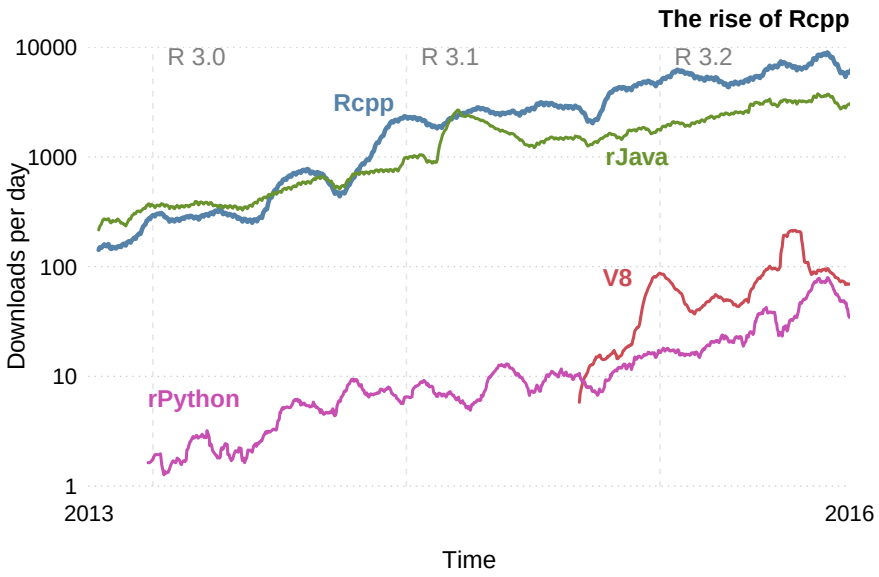


Figure 7.4: Comparison of mean functions.

2. The `for` loop has a more complicated syntax.

```
for (variable initialization; condition; variable update ) {
  // Code to execute
}
```

3. C++ provides operators to modify variables in place. So `i++` increases the value of `i` by `1`. Similarly, we could rewrite part of the loop as

```
mean += x[i]/n;
```

   The above code adds `x[i]/n` to the value of `mean`. Other similar operators are `-=`, `*=`, `/=` and `i--`.

4. A C++ vector starts at `0` **not** `1`

To compare the C++ and R functions, we'll generate some normal random numbers for the comparison

```
x = rnorm(1e4)
```

Then call the `microbenchmark` function (results plotted in figure 7.4).

```
# com_mean_r is the compiled version of mean_r
z = microbenchmark(
  mean(x), mean_r(x), com_mean_r(x), mean_c(x),
  times=1000
)
```

In this simple example, the Rcpp variant is around 100 times faster than the corresponding R version. This sort of speed-up is not uncommon when switching to an Rcpp solution.

Exercises

Consider the following piece of code

```
double test1() {
  double a = 1.0 / 81;
  double b = 0;
  for (int i = 0; i < 729; ++ i)
    b = b + a;
  return b;
}
```

1. Save the function `test1` in a separate file. Make sure it works.
2. Write a similar function in R and compare the speed of the C++ and R versions.
3. Create a function called `test2` where the `double` variables have been replaced by `float`. Do you still get the correct answer?
4. Change `b = b + a` to `b += a` to make you code more `C` like.
5. (Hard) What's the difference between `i++` and `++i`?

Matrices

Each vector type has a corresponding matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix` and `LogicalMatrix`. We use these types in a similar way to how we used `NumericVector`'s. The main differences are:

- When we initialise, we need specify the number of rows and columns

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

- We subset using `()`, i.e. `mat(5, 4)`.
- The first view in a matrix is `mat(0, 0)` - remember indexes start with `0`.
- To determine the number of rows and columns, we use the `.nrow()` and `.ncol()` methods.

7.5.7 C++ with sugar on top

Rcpp sugar brings a higher-level of abstraction to C++ code written using the **Rcpp** API. What this means in practice is that we can write C++ code in the style of R. For example, suppose we wanted to find the squared difference of two vectors; a squared residual in regression. In R we would use

```
sq_diff_r = function(x, y) (x - y)^2
```

Rewriting the function in standard C++ would give

```
NumericVector res_c(NumericVector x, NumericVector y) {
  int i;
  int n = x.size();
  NumericVector residuals(n);
  for(i=0; i<n; i++) {
    residuals[i] = pow(x[i] - y[i], 2);
  }
  return residuals;
}
```

With **Rcpp** sugar we can rewrite this code to be more succinct and have more of an R feel:

```
NumericVector res_sugar(NumericVector x, NumericVector y) {
  return pow(x-y, 2);
}
```

In the above C++ code, the `pow` function and `x-y` are valid due to **Rcpp** sugar. Other functions that are available include the d/q/p/r statistical functions, such as `rnorm` and `pnorm`. The sweetened versions aren't usually faster than the C++ version, but typically there's very little difference between the two. However with the sugared variety, the code is shorter and is constantly being improved.

Exercises

1. Construct an R version, `res_r` and compare the three function variants.
2. In the above example, `res_sugar` is faster than `res_c`. Do you know why?

7.5.8 Rcpp resources

The aim of this section was to provide an introduction to `Rcpp`. One of the selling features of `Rcpp` is that there is a great deal of documentation available.

- The Rcpp website;
- The original Journal of Statistical Software paper describing `Rcpp` and the follow-up book (Eddelbuettel and François 2011; Eddelbuettel 2013);
- H. Wickham (2014a) provides a very readable chapter on `Rcpp` that goes into a bit more detail than this section;
- The `Rcpp` section on the stackoverflow website. Questions are often answered by the `Rcpp` authors.

Chapter 8

Efficient hardware

This chapter is odd for a book on R programming. It contains very little code, and yet the chapter has the potential to speed up your algorithms by orders of magnitude. This chapter considers the impact that your computer has on your time.

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.

We'll begin this chapter with an background section on computer storage and memory and how it is measured. Then we consider individual computer components, before concluding with renting machines in the cloud.

8.1 Top 5 tips for efficient hardware

- Use the package **benchmarkme** to assess your CPUs number crunching ability; is it worth upgrading your hardware?
- If possible, add more RAM.
- Double check that you have installed a 64-bit version of R.
- Cloud computing is a cost effective way of obtaining more compute power.
- A solid state drives 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.

8.2 Background: what is a byte?

A computer cannot store "numbers" or "letters". The only thing a computer can store and work with is bits. A bit is binary, it is either a 0 or a 1. In fact from a physics perspective, a bit is just a blip of electricity that either is or isn't there.

In the past the ASCII character set dominated computing. This set defines 128 characters including 0 to 9, upper and lower case alpha-numeric and a few control characters such as a new line. To store these characters required 7 bits since $2^7 = 128$, but 8 bits were typically used for performance reasons. Table 8.1 gives the binary representation of the first few characters.

The limitation of only having 256 characters led to the development of Unicode, a standard framework aimed at creating a single character set for every reasonable writing system. Typically, Unicode characters require sixteen bits of storage.

Table 8.1: The bit representation of a few ASCII characters.

Bit representation	Character
01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01010010	R

Eight bits is one byte, or ASCII character. So two ASCII characters would use two bytes or 16 bits. A pure text document containing 100 characters would use 100 bytes (800 bits). Note that mark-up, such as font information or meta-data, can impose a substantial memory overhead: an empty `.docx` file requires about $3,700$ bytes of storage.

When computer scientists first started to think about computer memory, they noticed that $2^{10} = 1024 \simeq 10^3$ and $2^{20} = 1,048,576 \simeq 10^6$, so they adopted the short hand of kilo- and mega-bytes. Of course, *everyone* knew that it was just a short hand, and it was really a binary power. When computers became more wide spread, foolish people like you and me just assumed that kilo actually meant 10^3 bytes.

Fortunately the IEEE Standards Board intervened and created conventional, internationally adopted definitions of the International System of Units (SI) prefixes. So a kilobyte (kB) is $10^3 = 1000$ bytes and a megabyte (MB) is 10^6 bytes or 10^3 kilobytes (see table 8.2). A petabyte is approximately 100 million drawers filled with text. Astonishingly Google processes around 20 petabytes of data every day.

Factor	Name	Symbol	Origin	Derivation
2^{10}	kibi	Ki	Kilobinary:	$(2^{10})^1$
2^{20}	mebi	Mi	Megabinary:	$(2^{10})^2$
2^{30}	gibi	Gi	Gigabinary:	$(2^{10})^3$
2^{40}	tebi	Ti	Terabinary:	$(2^{10})^4$
2^{50}	pebi	Pi	Petabinary:	$(2^{10})^5$

Table 8.2: Data conversion table. Credit: http://physics.nist.gov/cuu/Units/binary.html

Even though there is now an agreed standard for discussing memory, that doesn't mean that everyone follows it. Microsoft Windows, for example, uses 1MB to mean 2^{20}B. Even more confusing the capacity of a 1.44MB floppy disk is a mixture, 1MB $= 10^3 \times 2^{10}$B. Typically RAM is specified in kibibytes, but hard drive manufactors follow the SI standard!

8.3 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. RAM is found in computers, phones, tablets and even printers. 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 analysis.

Even if the original data set is relatively small, your analysis can generate large objects. For example, suppose we want to perform standard cluster analysis. The built-in data set `USAarrests`, is a data frame with 50 rows and 4 columns. Each row corresponds to a state in the USA

```
head(USArrests, 3)
#>         Murder Assault UrbanPop Rape
#> Alabama   13.2     236       58 21.2
#> Alaska    10.0     263       48 44.5
```

```
d = dist(USArrests)
```

When we inspect the object size of the original data set and the distance object using the **pryr** package

```
pryr::object_size(USArrests)
#> 5.23 kB
pryr::object_size(d)
#> 14.3 kB
```

The distance object `d` is actually a vector that contains the distances in the upper triangular region.

we have managed to create an object that is three times larger than the original data set. In fact the object `d` is a symmetric $n \times n$ matrix, where n is the number of rows in `USAarrests`. Clearly, as `n` increases the size of `d` increases at rate $O(n^2)$. So if our original data set contained $10,000$ records, the associated distance matrix would contain almost 10^8 values. Of course since the matrix is symmetric, this corresponds to around 50 million unique values.

A rough rule of thumb is that your RAM should be three times the size of your data set.

Another benefit of having increasing the amount of onboard RAM is that the 'garbage collector', a process that runs periodically to free-up system memory occupied by R, is called less often.

It is straightforward to determine how much RAM you have. Under Windows,

1. Clicking the `Start` button picture of the Start button, then right-clicking Computer. Next click on `Properties`.
2. In the `System` section, you can see the amount of RAM your computer has next to the `Installed memory (RAM)` section. Windows reports how much RAM it can use, not the amount installed. This is only an issue if you are using a 32-bit version of Windows.

In Mac, click the Apple menu. Select `About This Mac` and a window appears with the relevant information.

On almost all Unix-based OSs, you can find out how much RAM you have using the code `vmstat`, whilst on all Linux distributions, you can use the command `free`. Using this in conjunction with the `-h` tag will provide the answer in human readable format, as illustrated below for a 16 GB machine:

```
$ free -h
             total       used       free
Mem:           15G       4.0G        11G
```

It is sometimes possible to increase your computer's RAM. On a computer motherboard there are typically 2 to 4 RAM or memory slots. If you have free slots, then you can add more memory. RAM comes in the form of dual in-line memory modules (DIMMs) that can be slotted into the mother board spaces (see figure 8.1 for example). However it is common that all slots are already taken. This means that to upgrade your computer's memory, some or all of the DIMMs will have to be removed. To go from 8GB to 16GB, for example, you may have to discard the two 4GB RAM cards and replace them with two 8GB cards. Increasing your laptop/desktop from 4GB to 16GB or 32GB is cheap and should definitely be considered. As R Core member Uwe Ligges states,

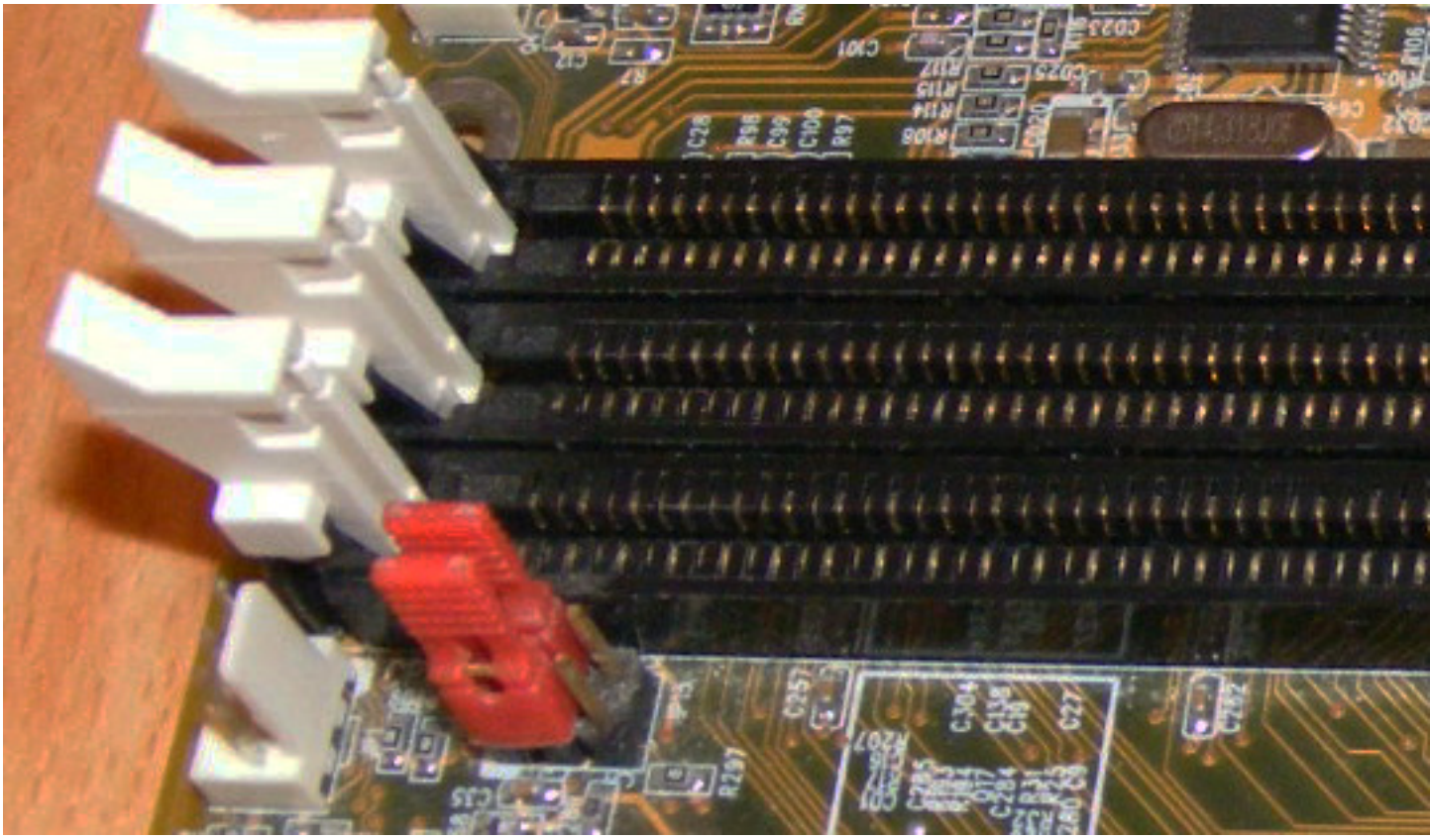


Figure 8.1: Three DIMM slots on a computer motherboard used for increasing the amount of available RAM. Credit: Wikimedia.org

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

It is a testament to the design of R that it is still relevant and its popularity is growing. Ross Ihaka, one of the originators of the R programming language, made a throw-away comment in 2003:

```
fortunes::fortune(21)
#>
#> I seem to recall that we were targetting 512k Macintoshes. In our dreams
#> we might have seen 16Mb Sun.
#>    -- Ross Ihaka (in reply to the question whether R&R thought when they
#>       started out that they would see R using 16G memory on a dual Opteron
#>       computer)
#>       R-help (November 2003)
```

Considering that a standard smart phone now contains 1GB of RAM, the fact that R was designed for "basic" computers, but can scale across clusters is impressive. R's origins on computers with limited resources helps explain its efficiency at dealing with large datasets.

Figure 8.2: A standard 2.5" hard drive, found in most laptops. Credit: https://en.wikipedia.org/wiki/Hard_disk_drive

Exercises

The following two exercises aim to help you determine if it is worthwhile upgrading your RAM.

1. R loads everything into memory, i.e. your computers RAM. How much RAM does your computer you have?
2. Using your preferred search engine, how much does it cost to double the amount of available RAM on your system?

8.4 Hard drives: HDD vs SSD

You are using R because you want to analyse data. The data is typically stored on your hard drive; but not all hard drives are equal. Unless you have a fairly expensive laptop your computer probably has a standard hard disk drive (HDD). HDDs were first introduced by IBM in 1956. Data is stored using magnetism on a rotating platter, as shown in Figure 8.2. The faster the platter spins, the faster the HDD can perform. Many laptop drives spin at either 5400RPM (Revolutions per Minute) or 7200RPM. The major advantage of HDDs is that they are cheap, making a 1TB laptop standard.

In the authors' experience, having an SSD drive doesn't make **much** difference to R. However, the reduction in boot time and general tasks makes an SSD drive a wonderful purchase.

Solid state drives (SSDs) can be thought of as large, but more 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 50MB). For SSDs, speeds are typically over 200MB/s. For top-of-the-range models this can approach 500MB/s. If you're wondering, read/write speeds for RAM is around $2 - 20$GB/s. So at best SSDs are at least one order of magnitude slower than RAM, but still faster than standard HDDs.

If you are unsure what type of hard drive you have, then time how long your computer takes to reach the log-in screen. If it is less then five seconds, you probably have a SSD. There are links on the book's website detailing more precise methods for each OS.

8.5 Operating systems: 32-bit or 64-bit

R comes in two versions: 32-bit and 64-bit. Your operating system also comes in two versions, 32-bit and 64-bit. Ideally you want 64-bit versions of both R and the operating system. Using a 32-bit version of either has severe limitations on the amount of RAM R can access. So when we suggest that you should just buy more RAM, this assumes that you are using a 64-bit operating system, with a 64-bit version of R.

If you are using an OS version from the last five years, it is unlikely to be 32-bit OS.

A 32-bit machine can access at most only 4GB of RAM. Although some CPUs offer solutions to this limitation, if you are running a 32-bit operating system, then R is limited to around 3GB RAM. If you are running a 64-bit operating system, but only a 32-bit version of R, then you have access to slightly more memory (but not much). Modern systems should run a 64-bit operating system, with a 64-bit version of R. Your memory limit is now measured as 8 terabytes for Windows machines and 128TB for Unix-based OSs. An easy method for determining if you are running a 64-bit version of R is to run

```
.Machine$sizeof.pointer
```

which will return 8 if you a running a 64-bit version of R.

To find precise details consult the R help pages `help("Memory-limits")` and `help("Memory")`.

Exercises

These exercises aim to condense the previous section into the key points.

1. Are you using 32-bit or 64-bit version of R?
2. If you are using Windows, what are the results of running the command `memory.limit()`?

8.6 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 frequency with which the CPU executes instructions. The faster the clock speed, the more instructions a CPU can execute in a section. CPU clock speed for a single CPU has been fairly static in the last couple of years, hovering around 3.4GHz (see figure 8.3).

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

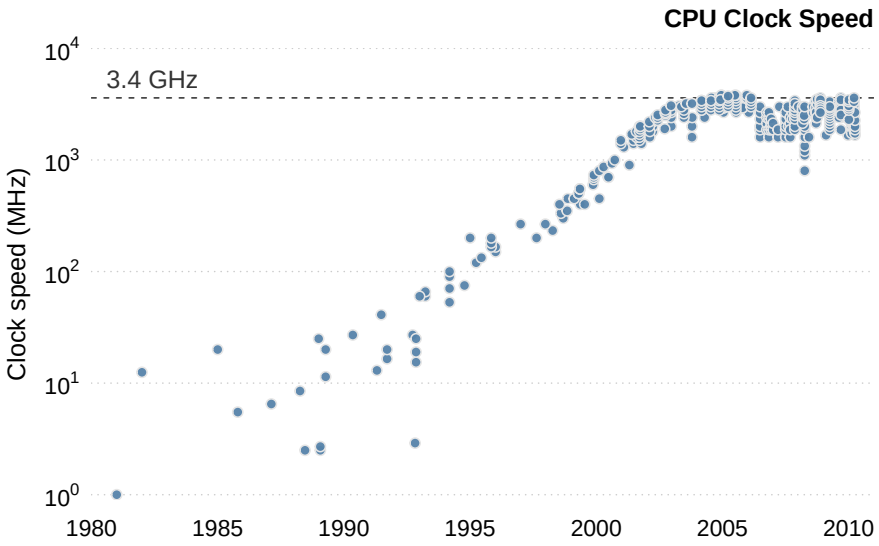


Figure 8.3: CPU clock speed. The data for this figure was collected from web-forum and wikipedia. It is intended to indicate general trends in CPU speed.

for benchmarking your system and contains data from previous benchmarks. Figure 8.4 shows the relative performance for over 150 CPUs.

Running the benchmarks and comparing your CPU to others is straightforward. First load the package

```
library("benchmarkme")
```

Then run the benchmarks and plot via

```
res = benchmark_std()
plot(res)
# Upload your benchmarks for future users
upload_results(res)
```

You get the model specifications of the top CPUs using `get_datatable(res)`.

8.7 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.

8.7.1 Amazon EC2

Amazon Elastic Compute Cloud (EC2) is one of a number of providers of this service. EC2 makes it (relatively) easy to run R instances in the cloud. Users can configure the operating system, CPU, hard drive type, the amount of RAM and where your project is physically located.

If you want to run a server in the Amazon EC2 cloud, you have to select the system you are going to boot up. There are a vast array of pre-packaged system images. Some of these images are just basic operating systems, such as Debian or Ubuntu, which require further configuration. There is also an Amazon machine image that specifically targets R and RStudio.

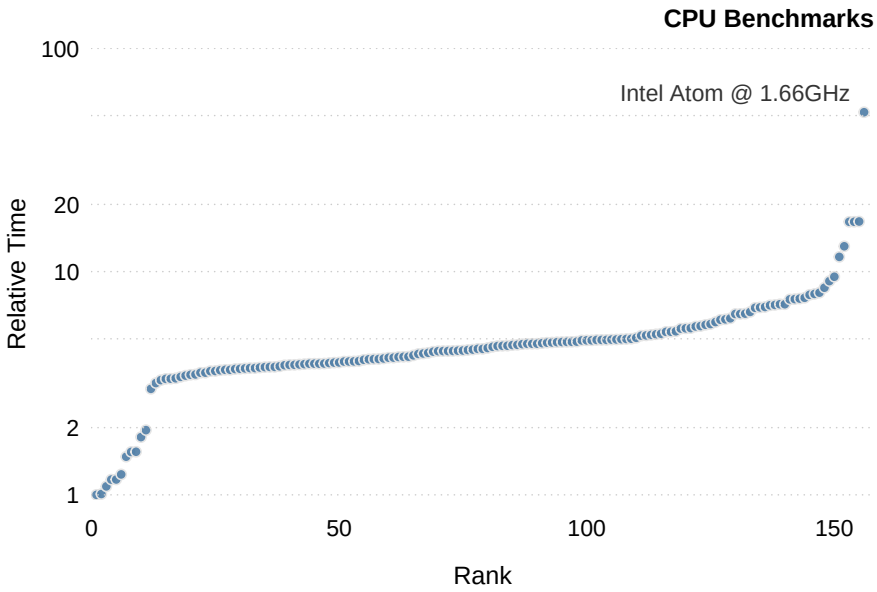


Figure 8.4: CPU benchmarks from the R package, **benchmarkme**. Each point represents an individual CPU result.

Exercise

To assess whether you should consider cloud computing, how much does it cost to rent a machine comparable to your laptop in the cloud?

Chapter 9

Efficient collaboration

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. This chapter provides guidance on how to minimise the risks and maximise the benefits of collaborative R programming.

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, as discussed in Chapter 4.

Collaborative programming provides an opportunity for people to review each other's code. This can be encouraged by using a uniform style with many comments as described in Section 9.2. Like using a clear style in human language, following a style guide has the additional advantage of making your code more understandable to others.

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. Fortunately it is now easier than ever before to integrate version control into your project, using RStudio's interface to the version control software `git` and online code sharing websites such as GitHub. This is the subject of Section 9.3.

The final section, 9.4, addresses the question of how to respond when you find inefficient code. Refactoring is the process of re-writing poorly written or scripts so they are faster, more comprehensible, more portable and easier to maintain.

9.1 Top 5 tips for efficient collaboration

- Have a consistent coding style.
- Think carefully about your comments and keep them up to date.
- Use version control whenever possible.
- Use informative commit messages.
- Don't be afraid to update (refactor) code when needed.

9.2 Coding style

To be a successful programmer you need to use a consistent programming style. There is no single 'correct' style. To some extent good style is subjective and down to personal taste. There are, however, general principles that most programmers agree on, such as:

- Use modular code;
- Comment your code;
- Don't Repeat Yourself (DRY);
- Be concise, clear and consistent.

Good coding style will make you more efficient even if you are the only person who reads it. When your code is read by multiple readers or you are developing code with co-workers, having a consistent style is even more important. There are a number of R style guides online that are broadly similar, including one by Google and one by Hadley Whickham. The style followed in this book is based on a combination of Hadley Wickham's guide and our own preferences (we follow Yihui Xie in preferring `=` to `<-` for assignment, for example).

In-line with the principle of automation (automate any task that can save time by automating), the easiest way to improve your code is to ask your computer to do it, using RStudio.

9.2.1 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. An example is provided below.

```
# Poorly indented/formatted code
if(!exists("x")){
x=c(3,5)
y=x[2]}
```

This code chunk works but is not pleasant to read. RStudio automatically indents the code after the `if` statement as follows.

```
# Automatically indented code (Ctrl+I in RStudio)
if(!exists("x")){
  x=c(3,5)
  y=x[2]}
```

This is a start, but it's still not easy to read. This can be fixed in RStudio as illustrated below (these options can be seen in the Code menu, accessed with `Alt+C` on Windows/Linux computers).

```
# Automatically reformat the code (Ctrl+Shift+A in RStudio)
if(!exists("x")) {
  x = c(3, 5)
  y = x[2]
  }
```

Note that some aspects of style are subjective: we would not leave a space after the `if` and `)`.

9.2.2 File names

File names should use the `.R` extension and should be lower case (e.g. `load.R`). Avoid spaces. Use a dash or underscore to separate words.

```
# Good names
normalise.R
load.R
# Bad names
Normalise.r
load data.R
```

9.2.3 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. Package names should be surrounded with speech marks.

```
# Good
library("ggplot2")
# Bad
library(ggplot2)
```

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

9.2.4 Commenting

Avoid using plain English to explain standard R code

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

Instead comments should explain at a higher level of abstraction the programmer's intention (McConnell 2004). Each comment line should begin with a single hash (`#`), followed by a space. Comments can be toggled (turned on and off) in this way with `Ctl+Shift+C` in RStudio. The double hash (`##`) can be reserved for R output. If you follow your comment with four dashes (`# ----`) RStudio will enable code folding until the next instance of this.

9.2.5 Object names

Function and variable names should be lower case, with an underscore (`_`) separating words. Unless you are creating an S3 object, avoid using a `.` in the name. Create names that are concise, but still have meaning.

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

It's tempting to use `T/F` as shortcuts. But it is easy to accidently redefine these variables, e.g. `F = 10`. R raises an error if you try to redefine `TRUE/FALSE`

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`. Avoid using names of existing functions. Do not, for example, use `data` as a variable name.

9.2.6 Example package

The `lubridate` package is a good example of a package that has a consistent naming system, to make it easy for users to guess its features and behaviour. Dates are encoded in a variety of ways, but the `lubridate` package has a neat set of functions consisting of the three letters, **y**ear, **m**onth and **d**ay. For example,

```
library("lubridate")
ymd("2012-01-02")
dmy("02-01-2012")
mdy("01-02-2012")
```

9.2.7 Assignment

The two most common ways of assigning objects to values in R is with `<-` and `=`. In most (but not all) contexts, they can be used interchangeably. Regardless of which operator you prefer, consistency is key, particularly when working in a group. In this book we use the `=` operator for assignment, as it's faster to type and more consistent with other languages.

The one place where a difference occurs is during function calls. Consider the following piece of code used for timing random number generation

```
system.time(expr1 <- rnorm(10e5))
system.time(expr2 = rnorm(10e5)) # error
```

The first lines will run correctly **and** create a variable called `expr1`. The second line will raise an error. When we use `=` in a function call, it changes from an *assignment* operator to an *argument passing* operator. For further information about assignment, see `?assignOps`.

9.2.8 Spacing

Consistent spacing is an easy way of making your code more readable. Even a simple command such as `x = x + 1` takes a bit more time to understand when the spacing is removed, i.e. `x=x+1`. 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. As with English, add a space after a comma, e.g.

```
z[z$colA > 1990, ]
```

9.2.9 Indentation

Use two spaces to indent code. Never mix tabs and spaces. RStudio can automatically convert the tab character to spaces (see `Tools -> Global options -> Code`).

9.2.10 Curly braces

Consider the following code:

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

Typing this straight into R will result in an error. An opening curly brace, `{` should not go on its own line and should always be followed by a line break. A closing curly brace should always go on its own line (unless it's followed by an `else`, in which case the `else` should go on its own line). The code inside a curly braces should be indented (and RStudio will enforce this rule), as shown below.

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

Exercises

Look at the difference between your style and RStudio's based on a representative R script that you have written (see Section 9.2). What are the similarities? What are the differences? Are you consistent? Write these down and think about how you can use the results to improve your coding style.

9.3 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 Dropbox 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 version control system we recommend is Git, a command-line application created by Linus Torvalds, who also invented Linux.[1] The easiest way to integrate your R projects with Git, if you're not accustomed to using a shell (e.g. the Unix command line), is with RStudio's Git tab, in the top right-hand window (see figure 9.1). This shows a number of files have been modified (as illustrated with the blue M symbol) and that some are new (as illustrated with the yellow ? symbol). Checking the tick-box will enable these files to be *committed*.

9.3.1 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'.

Committing code only updates the files on your 'local' branch. To update the files stored on a remote server (e.g. on GitHub), you mush 'push' the commit. This can be done using `git push` from a shell or using the

[1] We recommend '10 Years of Git: An Interview with Git Creator Linus Torvalds' from Linux.com for more information on this topic.

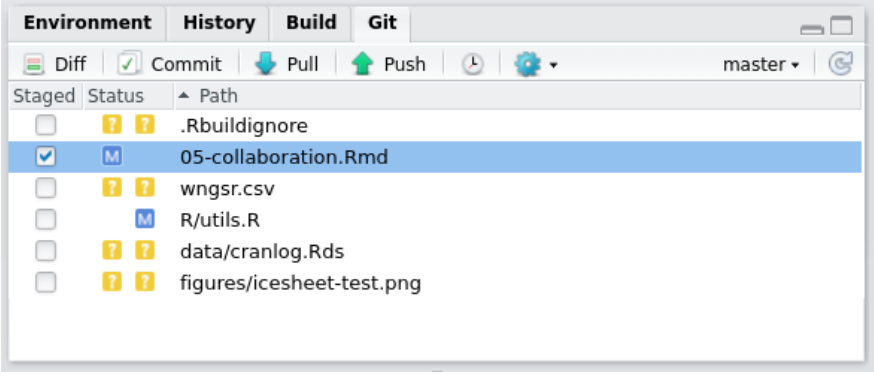


Figure 9.1: The Git tab in RStudio

green up arrow in RStudio, illustrated in figure 9.1. The blue down arrow will 'pull' the latest version of the repository from the remote.[2]

9.3.2 Git integration in RStudio

How to enable this functionality on your installation of RStudio? RStudio can be a GUI Git only if Git has been installed *and* RStudio can find it. You need a working installation of Git (e.g. installed through `apt-get install git` Ubuntu/Debian or via GitHub Desktop for Mac and Windows). RStudio can be linked to your Git installation via Tools > Global Options, in the Git/SVN tab. This tab also provides a link to a help page on RStudio/Git.

Once Git has been linked to your RStudio installation, it can be used to track changes in a new project by selecting `Create a git repository` when creating a new project. The tab illustrated in figure 9.1 will appear, allowing functionality for interacting with Git via RStudio.

RStudio provides a useful GUI for navigating past commits. This allows you to see the entire history of your project. To navigate and view the details of past commits click on the Diff button in the Git pane, as illustrated in figure 9.2.

9.3.3 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. And GitHub makes it easy to get your work 'out there' to the world for efficiently collaborating with others, without the restraints placed on CRAN packages.

To install the GitHub version of the **benchmarkme** package, for example one would enter

```
devtools::install_github("csgillespie/benchmarkme")
```

Note that `csgillespie` is the GitHub user and `benchmarkme` is the package name. Replacing `csgillespie` with `robinlovelace` in the above code would install Robin's version of the package. This is useful for fast collaboration with many people, but you must remember that GitHub packages will not update automatically with the command `update.packages` (see 2.3.5).

[2]For a more detailed account of this process, see GitHub's help pages.

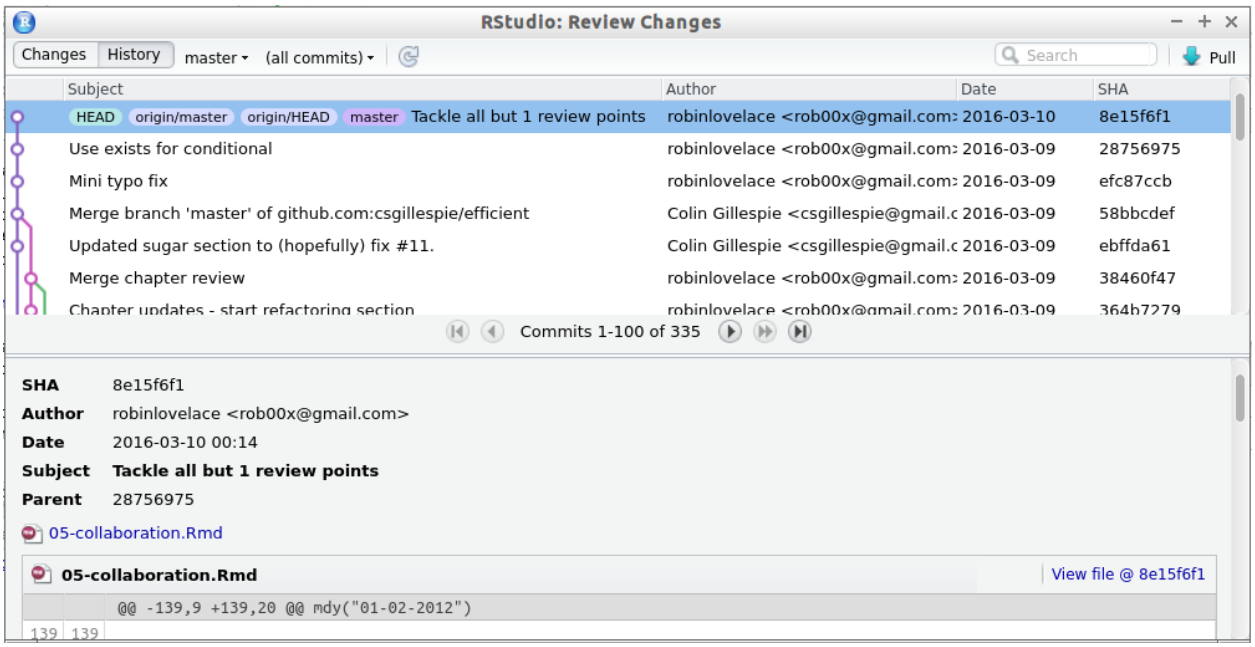


Figure 9.2: The Git history navigation interface

Warning: although GitHub is fantastic for collaboration, it can end up creating more problems than it solves if your collaborators are not git-literate. In one project, Robin eventually abandoned using GitHub to collaborate after his collaborator found it impossible to work with. More time was being spent debugging git/GitHub than actually working. Our advice therefore is to **never impose git** and always ensure that other lines of communication (e.g. phone calls, emails) are open as different people prefer different ways of communicating.

9.3.4 Branches, forks, pulls and clones

Git is a large program which takes a long time to learn in depth. However, getting to grips with the basics of some of its more advanced functions can make you a more efficient collaborator. Using and merging branches, for example, can make you work more scalable but testing new features in a self-contained environment before it is used in production (e.g. when shifting to an updated version of a package which is not backwards compatible). Instead of bogging you down with a comprehensive discussion of what is possible, this section cuts to the most important features for collaboration: branches, forks, fetches and clones. For a more detailed description of Git's powerful functionality, we recommend the book Pro Git, which is available online at git-scm.com/book/.

Branches are distinct versions of your repository. Git allows you jump seamlessly between different versions of your entire project. To create a new branch called test, you need to enter the shell and use the Git command line:

```
git checkout -b test
```

This is equivalent of entering two commands: `git branch test` to create the branch and then `git checkout test` to *checkout* that branch. Checkout means switch into that branch. Any changes will not affect your

previous branch. In RStudio you can jump quickly between branches using the drop down menu in the top right of the Git pane. This is illustrated in figure 9.1: see the `master` text followed by a down arrow. Clicking on this will allow you to select other branches.

Forks are like branches but they exist on other people's computers. You can fork a repository on GitHub easily, as described on the site's help pages. If you want an exact copy of this repository (including the commit history) you can *clone* this fork to your computer using the command `git clone` or by using a Git GUI such as GitHub Desktop. This is preferable from a collaboration perspective than cloning the repository directly, because any changes can be pushed back online easily if you are working from your own fork. You cannot push to forks that you have not created. If you want your work to be incorporated into the original fork you can use a *pull request*. Note: if you don't need the project's entire commit history, you can simply download a zip file containing the latest version of the repository from GitHub (see at the top right of any GitHub repository).

A pull request (PR) is a mechanism on GitHub by which your code can be added to an existing project. One of the most useful features of a PR from a collaboration perspective is that it provides an opportunity for others to comment on your code, line by line, before it gets merged. This is all done online on GitHub, as discussed in GitHub's online help. Following feedback, you may want to refactor code, written by you or others.

9.4 Refactoring code

It is likely that your skills as an R programmer will improve over time (e.g. after reading books such as this!). Furthermore, as R packages evolve, the best way to accomplish a task 6 months ago may not be the best way to accomplish it now. Thus it is likely that you will want to rewrite your code at some point. From a collaboration perspective, you may also need to adapt someone else's code to be more efficient. This is *refactoring*.

An example would be messy code that creates many objects and uses overlapping packages. The refactored code would be much clearer, efficient and more concise. There is no 'magic bullet' formula to refactor code, but a series of techniques can be used, including improving variable names, breaking the code into smaller, more modular chunks and making the code more abstract and generalisable. Refactoring it is a skill that you will learn over time. Good practice involves frequently updating your code - that will get you into the habit of reading code at a 'high level' so you identify quickly its key components and hopefully write it in a better form.

Let's use some code introduced in Chapter 5 as an example of something that could be written more efficiently:

```
# old code
fname = system.file("extdata/ghg-ems.csv", package = "efficient")
co2_ems = read.csv(fname)
names(co2_ems) = gsub("\\..*", "", names(co2_ems))
e_ems = aggregate(co2_ems$Electricity, list(co2_ems$Country),
                  mean, data = co2_ems)
```

```
# new code
library("dplyr")
fname = system.file("extdata/ghg-ems.csv", package = "efficient")
co2_ems = readr::read_csv(fname)
co2_elec_agg = group_by(co2_ems, Country) %>%
  summarise(Elec = mean(`Electricity/Heat (CO2) (MtCO2)`, na.rm = TRUE))
```

Although both approaches generate the same output data using the same number of lines of code (5), the new, refactored code is better in a number of ways. It is more efficient to type (containing 10 fewer characters)

and more efficient to run (it is around 20% faster). More importantly, the refactored code is more readable and easily extended, which is a vital consideration from the perspective of efficient collaboration. Adding the following line after the penultimate bracket in the second code chunk would create a new aggregate variable, for example:

```
, Trans = mean(`Transportation (CO2) (MtCO2)`, na.rm= TRUE)
```

Furthermore, by using the origin column names, the refactored code remains more faithful to the source data. The benefits of being explicit and verbose (i.e. including the units in the variable names) will likely outweigh the costs of additional characters in the analysis process when it comes to publishing the results.

Note that the number of lines of code is not a good measure of code efficiency. Clear and easily readable that is many lines of code is better than code that is compressed into a few lines that is tricky to read. Do not be afraid to make functions with multiple arguments more than one line. The `%>%` pipe operator encourages multi-line code that is easy to read. The number of *characters* in code is a better measure, but is still an imperfect measure of efficiency, let alone readability.

Exercise

Refactor the following code chunk to make it more efficient:

```
df = data.frame(
  letters = c("a", "a", "b", "b", "c", "c"),
  numbers = c(1, 2, 3, 4, 5, 6)
)
aggregate(df$numbers, list(df$letters), sum)
#>   Group.1  x
#> 1       a  3
#> 2       b  7
#> 3       c 11
```

Chapter 10

Efficient learning

As with any vibrant open source software community, R is fast moving. This can be disorientating because it means that you can never 'finish' learning R. On the other hand it can make R a fascinating subject: there is always more to learn. Even experienced R users keep finding new functionality that helps solve problems quicker and more elegantly. Therefore *learning how to learn* is one of the most important skills to have if you want to learn R *in depth*. We emphasise *depth* of learning because it is more efficient to learn something properly than to Google it repeatedly every time we forget how it works.

This chapter equips you with concepts and tips that will accelerate the transition from an R *hacker* to an R *programmer*. This inevitably involves effective use of R's help, reading R source code and use of online material.

10.1 Top 5 tips for efficient learning

- Monitor the R tag at stackoverflow for new questions.
- Read about the latest developments in the R Journal.
- Regularly browse R-bloggers to get an overview of the R eco-system.
- If asking a question, simplify your code and data as much as possible; the question should be reproducible.
- Subscribe (but not necessarily post) to the `R-devel` mailing list to gain a deeper insight into the R language.

10.2 Using R help

The R-project website contains seven detailed manuals about the R language, development, installation and data import. While the manuals are long, they do contain all necessary information. In particular if you are developing a package and want to submit that package to CRAN, you must confirm that you have read the extension documentation.

All functions have help files. For example to see the help file for `plot`, just type:

```
# Or help("plot")
?plot
```

The resulting help page is divided into a number of sections. The most *helpful* section (I find) is the examples (at the bottom of the help page) showing precisely how the function works. You can either copy and paste the code, or actually run the example code using the `example` command:.

```
example(plot)
```

When a package is added to CRAN, the example part of the documentation is run on all major platforms. This helps ensure that a package works on multiple systems.

Another useful section in the help file is `See Also:`. In the `plot` help file, it gives pointers to 3d plotting.

To look for help about a certain *topic* rather than a specific *function* use `??topic`, which is analogous to `?function`. To search for information about regression in all installed packages, for example, use the following command:

```
# Or help.search("regression")
??regression
```

To search more specifically for objects the `appropos` function can be useful. To search for all objects and functions in the current workspace containing the text string `lm`, for example, one would enter:

```
# Showing the first six results
#> [1] ".__C__anova.glm"      ".__C__anova.glm.null" ".__C__diagonalMatrix"
#> [4] ".__C__generalMatrix"  ".__C__glm"            ".__C__glm.null"
```

Sometimes a package contains vignettes. To browse any vignettes associated with a particular package, we can use the handy function

```
browseVignettes(package = "benchmarkme")
```

10.2.1 Reading R source code

R is open source. This means that we view the underlying source code and examine any function. Of course the code is complex, and diving straight into the source code won't help that much. However, watching to the github R source code mirror will allow you to monitor small changes that occur. This gives a nice entry point into a complex code base. Likewise examining the source of small functions, such as `NCOL` is informative, e.g. `getFunction("NCOL")`

Subscribing to the R NEWS blog is an easy way of keeping track of future changes.

Many R packages are developed in the open on github or r-forge. Select a few well known packages and examine their source. A good package to start with is **drat**. This is a relatively simple package developed by Dirk Eddelbuettel (author of Rcpp) that only contains a few functions. It gives you an excellent pointer into software development by one of the key R package writers.

10.3 Online resources

There is plenty of online help available. R-bloggers is a blog aggregator of content contributed by bloggers who write about R (in English). It is a great way to get exposed to new and different packages. Similarly monitoring the *#rstats* twitter tag keeps you up-to-date with the latest news.

The R-journal regularly publishes articles describing new R packages, as well as general programming hints. Similarly, the articles in the Journal of Statistical Software have a strong R bias.

There are also mailing lists, Google groups and the Stack Exchange Q & A sites. Before requesting help, read a few other questions to learn the format of the site. Make sure you search previous questions so you are not duplicating work. Perhaps the most important point is that people aren't under **any** obligation to answer your question. One of the fantastic things about the open-source community is that you can ask questions and one of core developers may answer your question free; but remember, everyone is busy!

10.3.1 Stackoverflow

The number one place on the internet for getting help on programming is Stackoverflow. This website provides a platform for asking and answering questions. Through site membership, questions and answers are voted up or down. Users of Stackoverflow earn reputation points when their question or answer is up-voted. Anyone (with enough reputation) can edit a question or answer. This helps the content remain relevant.

Questions are tagged. The R questions can be found under the R tag. The R page contains links to Official documentation, free resources, and various other links. Members of the Stackoverflow R community have tagged, using `r-faq`, a few question that often crop up.

10.3.2 Mailing lists and groups.

There are a large number of mailing lists and Google groups focused on R and particular packages. The main list for getting help is `R-help`. This is a high volume mailing list, with around a dozen messages per day. A more technical mailing list is `R-devel`. This list is intended for questions and discussion about code development in R. The discussion on this list is very technical. However it's a good place to be introduced to new ideas - but it's not the place to ask about these ideas! There are many other special interest mailing lists covering topics such as high performance computing to ecology. Many popular packages also have their own mailing list or Google group, e.g. **ggplot2** and **shiny**. The key piece of advice is before mailing a list, read the relevant mailing archive and check that your message is appropriate.

10.3.3 Asking a question

Asking questions on stackoverflow and R-help is hard. Your question should contain just enough information that you problem is clear and can be reproducible, while at the same time avoid unnecessary details. Fortunately there is a SO question - How to make a great R reproducible example? - that provides excellent guidance!

Minimal data set

What is the smallest data set you can construct that will reproduce your issue? Your actual data set may contain 10^5 rows and 10^4 columns, but to get your idea across you might only need 4 rows and 3 columns. Making small example data sets is easy. For example, to create a data frame with two numeric columns and a column of characters just use

```r
set.seed(1)
example_df = data.frame(x = rnorm(4), y = rnorm(4), z = sample(LETTERS, 4))
```

Note the call to `set.seed` ensures anyone who runs the code will get the same random number stream. Alternatively, you use one of the many data sets that come with R - `library(help="datasets")`.

If creating an example data set isn't possible, then use `dput` on your actual data set. This will create an ASCII text representation of the object that will enable anyone to recreate the object

```r
dput(example_df)
#> structure(list(
#>   x = c(-0.626453810742332, 0.183643324222082, -0.835628612410047, 1.59528080213779),
#>   y = c(0.329507771815361, -0.820468384118015, 0.487429052428485, 0.738324705129217),
#>   z = structure(c(3L, 4L, 1L, 2L), .Label = c("J", "R", "S", "Y"), class = "factor")),
#>   .Names = c("x", "y", "z"), row.names = c(NA, -4L), class = "data.frame")
```

Minimal example

What you should not do, is simply copy and paste your entire function into your question. It's unlikely that your entire function doesn't work, so just simplify it the bare minimum. The aim is to target your actual issue. Avoid copying and pasting large blocks of code; remove superfluous lines that are not part of the problem. Before asking your question, can you run you code in a clean R environment and reproduce your error?

Appendix A

Package Dependencies

The book depends on the following packages:

References

Berkun, Scott. 2005. *The Art of Project Management*. O'Reilly Media.

Braun, John, and Duncan J Murdoch. 2007. *A First Course in Statistical Programming with R*. Vol. 25. Cambridge University Press Cambridge.

Burns, Patrick. 2011. *The R Inferno*. Lulu.com.

Chang, Winston. 2012. *R Graphics Cookbook*. O'Reilly Media.

Codd, E. F. 1979. "Extending the database relational model to capture more meaning." *ACM Transactions on Database Systems* 4 (4): 397–434. doi:10.1145/320107.320109.

Cotton, Richard. 2013. *Learning R*. O'Reilly Media.

———. 2016. *Testing R Code*.

Eddelbuettel, Dirk. 2013. *Seamless R and C++ Integration with Rcpp*. Springer.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18.

Eddelbuettel, Dirk, Romain François, J. Allaire, John Chambers, Douglas Bates, and Kevin Ushey. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18.

Goldberg, David. 1991. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys (CSUR)* 23 (1). ACM: 5–48.

Grant, Christine A, Louise M Wallace, and Peter C Spurgeon. 2013. "An Exploration of the Psychological Factors Affecting Remote E-Worker's Job Effectiveness, Well-Being and Work-Life Balance." *Employee Relations* 35 (5). Emerald Group Publishing Limited: 527–46.

Grolemund, G., and H. Wickham. n.d. *R for Data Science*. O'Reilly Media.

Janert, Philipp K. 2010. *Data Analysis with Open Source Tools*. "O'Reilly Media".

Jensen, Jørgen Dejgård. 2011. "Can Worksite Nutritional Interventions Improve Productivity and Firm Profitability? A Literature Review." *Perspectives in Public Health* 131 (4). SAGE Publications: 184–92.

Kersten, Martin L, Stratos Idreos, Stefan Manegold, Erietta Liarou, and others. 2011. "The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds." *PVLDB Challenges and Visions* 3.

Kruchten, Philippe, Robert L Nord, and Ipek Ozkaya. 2012. "Technical Debt: From Metaphor to Theory and Practice." *IEEE Software*, no. 6. IEEE: 18–21.

Lovelace, Ada Countess. 1842. "Translator's Notes to an Article on Babbage's Analytical Engine." *Scientific Memoirs* 3: 691–731.

Lovelace, Robin, and Morgane Dumont. 2016. *Spatial Microsimulation with R*. CRC Press. https://www.

crcpress.com/Spatial-Microsimulation-with-R/Lovelace-Dumont/9781498711548.

McCallum, Ethan, and Stephen Weston. 2011. *Parallel R*. O'Reilly Media.

McConnell, Steve. 2004. *Code Complete*. Pearson Education.

Pereira, Michelle Jessica, Brooke Kaye Coombes, Tracy Anne Comans, and Venerina Johnston. 2015. "The Impact of Onsite Workplace Health-Enhancing Physical Activity Interventions on Worker Productivity: A Systematic Review." *Occupational and Environmental Medicine* 72 (6). BMJ Publishing Group Ltd: 401–12.

PMBoK, A. 2000. "Guide to the Project Management Body of Knowledge." *Project Management Institute, Pennsylvania USA*.

Sekhon, Jasjeet S. 2006. "The Art of Benchmarking: Evaluating the Performance of R on Linux and OS X." *The Political Methodologist* 14 (1): 15–19.

Spector, Phil. 2008. *Data Manipulation with R*. Springer Science & Business Media.

Visser, Marco D., Sean M. McMahon, Cory Merow, Philip M. Dixon, Sydne Record, and Eelke Jongejans. 2015. "Speeding Up Ecological and Evolutionary Computations in R; Essentials of High Performance Computing for Biologists." Edited by Francis Ouellette. *PLOS Computational Biology* 11 (3): e1004140. doi:10.1371/journal.pcbi.1004140.

Wickham, Hadley. 2014a. *Advanced R*. CRC Press.

———. 2014b. "Tidy Data." *The Journal of Statistical Software* 14 (5).

———. 2015. *R Packages*. O'Reilly Media.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. Vol. 29. CRC Press.