Efficient R programming

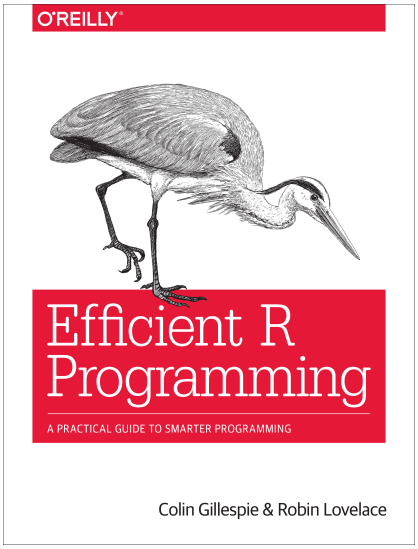
Colin Gillespie and Robin Lovelace

2016-03-07

Contents

Chapter 1

Introduction



This website contain text and code for the forthcoming O'Reilly book: Efficient R programming. Pull requests and general comments are welcome.

1.1 Package Dependencies

The book depends on the following packages:

Name	Title
benchmarkme	Crowd Sourced System Benchmarks
bookdown	Authoring Books with R Markdown
data.table	Extension of Data.frame
devtools	Tools to Make Developing R Packages Easier
DiagrammeR	Create Graph Diagrams and Flowcharts Using R
dplyr	A Grammar of Data Manipulation
drat	Drat R Archive Template
efficient	Becoming an Efficient R Programmer
formatR	Format R Code Automatically
fortunes	R Fortunes
ggplot2	An Implementation of the Grammar of Graphics
knitr	A General-Purpose Package for Dynamic Report Generation in R
microbenchmark	Accurate Timing Functions
pryr	Tools for Computing on the Language
rbenchmark	Benchmarking routine for R
readr	Read Tabular Data
tidyr	Easily Tidy Data with ‘spread()‘ and ‘gather()‘ Functions

These packages can be installed via

```
devtools::install_github("csgillespie/efficientR")
```

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. We explore how the operating system, R versions and settings, configuration of your R editor, and other settings can make your R work faster. The next chapter deals with hardware. By the end of this chapter you should understand how to optimize your computer for efficient R programming. That includes consideration of 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

2.1 Operating system

R works on all three major operating systems (OSs): Linux, Mac and Windows. 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 in all major operating systems. 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.

2.1.1 Operating system and resource monitoring

Minor differences aside,[1] R's computational efficiency is 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 6) 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.

[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.

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 2.4).

Regardless of your operating system, it is good practice to track how system resources (primarily CPU and RAM use) respond when running time-consuming tasks. Alongside R profiling functions such as `profvis` (see Section **XXX**), 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 6 on parallel programming). Monitoring CPU load in this context can be useful for identifying whether R is running in parallel (see Figure 2.1).

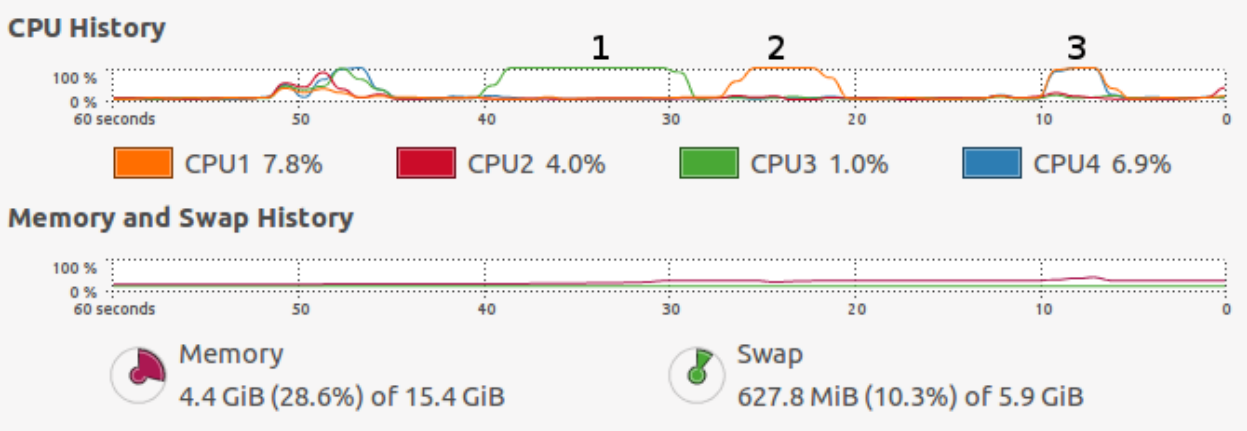


Figure 2.1: Output from a system monitor (`gnome-system-monitor` running on Ubuntu) showing the resources consumed by running the code presented in the second of the Exercises at the end of this section. The first increases RAM use, the second is single-threaded and the third is multi-threaded.

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 `Ctl-Alt-Del` or by clicking the task bar and 'Start Task Manager'.
- On Mac the **Activity Monitor** provides similar functionality. This can be initiated form the Utilities folder in Launchpad.

2.1.2 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 = data.frame(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
# XXX: Change to function from package
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?

2.2 R version

It is important to keep your R installation and packages up-to-date. This section explains how.

2.2.1 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. To add the RStudio mirror for Ubuntu 14.04 (codenamed Trusty), for example, add the following line to `/etc/apt/sources.list`:

```
echo 'deb https://cran.rstudio.com/bin/linux/ubuntu trusty`
```

In the above code `cran.rstudio.com` is the mirror and `trusty` is the Ubuntu version. Then `r-base` and other `r-` packages can be installed. R also works on FreeBSD and other Unix-based systems.[2]

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

2.2.2 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.

[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.

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. 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.

2.2.3 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. To provide another example, we can update only the packages named in `pkgs` with the following command:

```
update.packages(oldPkgs = pkgs)
```

2.2.4 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()
```

2.2.5 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.[3] This is a good thing: updating dozens of large packages can consume a large proportion of available system resources and much time!

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.

2.2.6 Exercises

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

2.3 R startup

Every time R starts, a number of files are read, in a particular order. The contents of these files determine how R performs for the duration of the session. Note that these files should only be changed with caution, as they may make your R version behave differently to other R installations. This could reduce 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, `path.expand("~")` (note the use of the Unix-like tilde to represent the home directory).
- 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`

[3] In the previous section we specified only a few packages to update.

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

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

2.3.1 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")
```

2.3.2 Example `.Rprofile` settings

An `.Rprofile` file is just an R script that is run at start-up. The examples at the bottom of the `.Rprofile` help file

```
help("Rprofile")
```

give clues as to the types of things we could place in our profile.

2.3.2.1 Setting options

The function `options` is a list that contains a number of default options. See `help("options")` or simply type `options()` to get an idea of what we can configure. In my `.Rprofile` file, we have the line

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

This changes three features.

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

Typically we want to avoid adding options to the start-up file that make our code non-portable. For example, adding

```
options(stringsAsFactors=FALSE)
```

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 it is dangerous as it may make your code less portable.

2.3.2.2 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 the global environment 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.

2.3.2.3 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)
```

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..

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")
}
```

2.3.2.4 Useful functions

You can also load useful functions in `.Rprofile`. 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:

```
setnicepar = function(mar = c(3, 3, 2, 1), mgp = c(2, 0.4, 0),
                      tck = -.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`. Also beware the dangers of loading many functions by default: it may make 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)
```

2.3.3 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

```
## Linux
R_LIBS=~/R/library

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

This variable points to a directory where R packages will be installed. When `install.packages` is called, new packages will be stored in `R_LIBS`.

Another common use of `.Renviron` is to store API keys that will be available from one session to another.[4] The following line in `.Renviron`, for example, sets the `ZEIT_KEY` environment variable which is used in the package diezeit package:

```
ZEIT_KEY=PUT_YOUR_KEY_HERE
```

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

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.

2.3.4 Exercises

1. What are the three locations where they 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.

2.4 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, for example by reducing 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 auto-completion 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.[5] 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.

2.4.1 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
```

[5] 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.

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 `Ctl` and `Alt` with the Mac-specific `Cmd` button.

2.4.2 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 `Ctl-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'.

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.

2.4.3 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 `Ctl-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

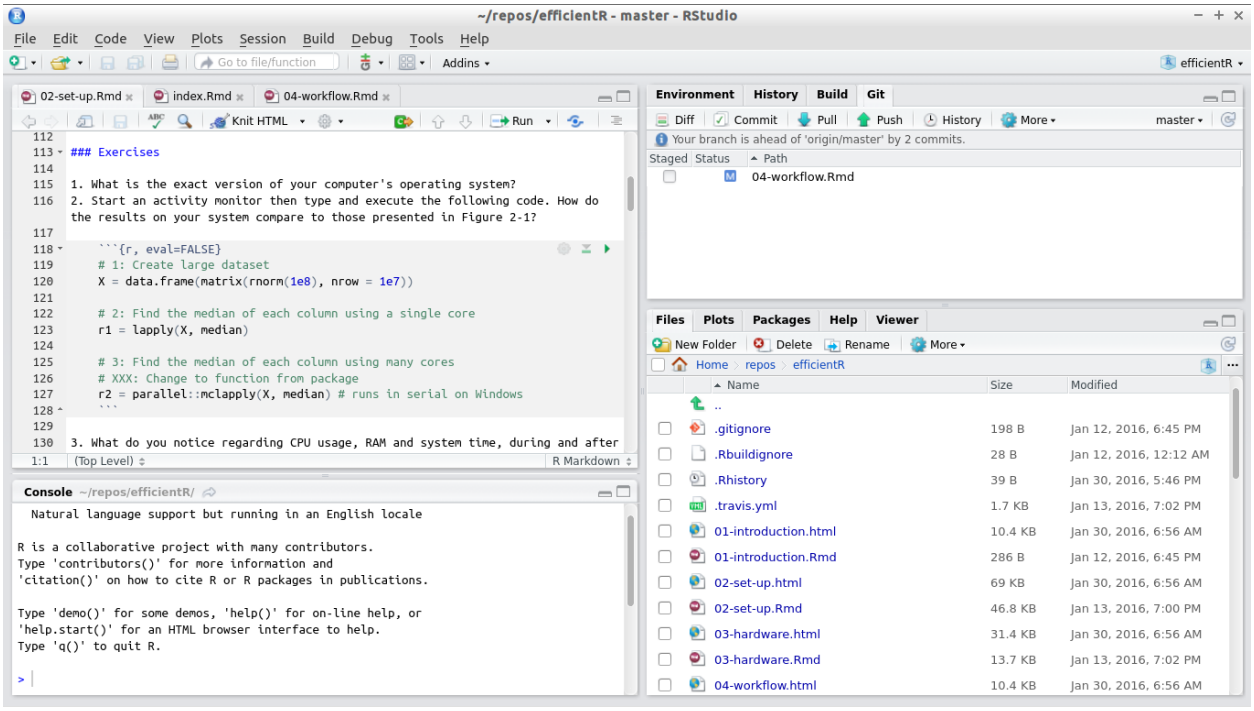


Figure 2.2: RStudio Panels

```
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).

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.

2.4.4 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.

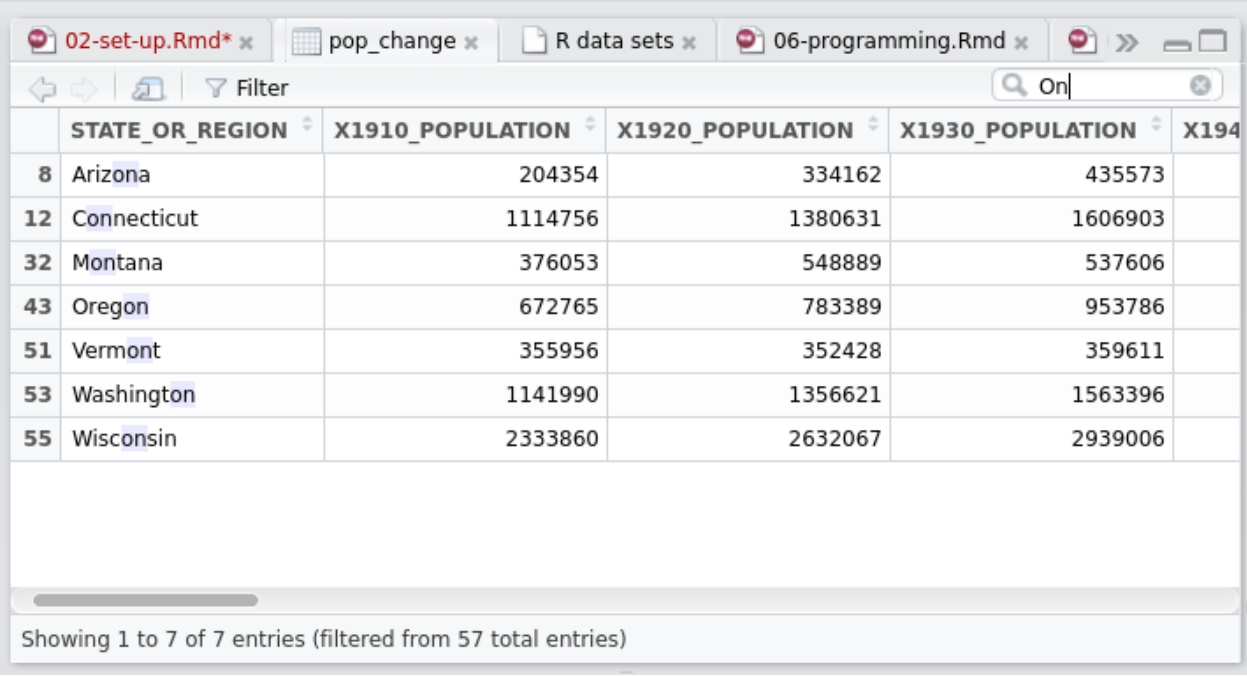


Figure 2.3: The data viewing tab in RStudio.

- `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.

2.4.5 Auto-completion

R provides some basic autocompletion functionality. Typing the beginning of a function name, for example `rn` (short for `rnorm()`), and hitting `Tab` 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 auto-completed. RStudio builds on this functionality and takes it to a new level.

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 auto-completion 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 auto-completion 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$varname`.[6] In this case `spdf` is the object name, `data` is the slot and `varname` is the variable name. RStudio makes such `S4` objects easier to use by enabling autocompletion of the short form `spdf$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 auto-completion 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.

2.4.6 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:

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

- `Ctl+Z/Shift+Z`: Undo/Redo.
- `Ctl+Enter`: Execute the current line or code selection in the Source pane.
- `Ctl+Alt+R`: Execute all the R code in the currently open file in the Source pane.
- `Ctl+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.
- `Ctl+D`: Delete the current line.

2.4.7 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

[6] 'Slots' are sub-elements of an object analogous to a column in a `data.frame` but referred to with `@` not `$`.

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 above).

2.4.8 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. Creating R packages is easier than ever before, with tools such as **devtools** for managing R's quirks and making the process user friendly. 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.

2.4.9 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`.)
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.

2.5 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.

2.5.1 Revolution R

Revolution R is the main software product offered by Revolution Analytics. It is "100%" compatible with R", supporting all available packages through the MRAN package repository. Revolution R provides faster performance on for certain functions than base R, through its use of MKL, an implementation of BLAS (as described above). Revolution R is available as a free and open source product, 'Revolution R Open' (RRO), and is reported to be faster than base R installations.[7]

Additional benchmarks reported by Eddelbuettel (2010) show the MKL implementations of R used in RRO and the commercial edition to be substantially faster than the reference case.

[7] See brodrigues.co/2014/11/11/benchmarks-r-blas-atlas-rro/, which finds Revolution R to be marginally faster than R using OpenBLAS and ATLAS BLAS implementations and Faster BLAS in R, which does not.

2.5.2 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.

- 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.

In this context it is also worth mentioning Julia. This is a fast language and interpreter which aims to provide "a high-level, high-performance dynamic programming language for technical computing" that will be familiar to R and Python users.

2.5.3 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.

2.5.4 Exercises

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

Chapter 3

Efficient hardware

In the previous chapter we saw how relatively simple tweaks to our system set-up can result in substantial time savings. In this chapter we explore the relationship between computer hardware and the speed of code execution. The goal is to help you decide whether the benefits of upgrading your hardware are worth the cost.

We'll begin with an introductory section on computer storage and memory and how it is measured, before moving on to individual computer components.

3.1 Introduction: 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 3.1 gives the binary representation of the first few characters.

Bit representation	Character
01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01010010	R

Table 3.1: The bit representation of a few ASCII characters.

The limitation of only gives 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.

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 3.2). 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 3.2: 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.

3.2 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 directly related to the size of data sets that it can process. As the amount of RAM your machine has increases, the size of datasets you can analyse also increases, so it is important to have sufficient RAM for your work.

Figure 3.1: Two 8GB RAM cards. Credit: https://commons.wikimedia.org/wiki/File:Two_8_GB_DDR4-2133_ECC_1.2_V_RDIMMs.jpg

Even if the original data set is relatively small, the 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
## Arizona    8.1     294       80 31.0
```

If we want to group states that have similar crime statistics, a standard first step is to calculate the distance or similarity matrix

```
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
```

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 (we will cover this in more detail in chapter XXX).

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 the computer motherboard, there are typically 2 or 4 RAM or memory slots. If you have free slots, then you can add more RAM. However, it is common that all slots are already taken. This means that to upgrade your computer's memory, some or all of the RAM 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,

```
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.

3.2.1 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 do you have?
2. Using Google, how much does it cost (in pounds) to double the amount of available RAM on your system?

3.3 Hard drives: HDD vs SSD

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 3.2. 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.

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).

Figure 3.2: A standard 2.5" hard drive, found in most laptops. Credit: https://en.wikipedia.org/wiki/Hard_disk_drive

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.

3.4 Operating systems: 32-bit or 64-bit

When we suggest that you should just buy more RAM, this assumes that you are using a 64-bit operating system. 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.

To find precise details consult the R help pages `help("Memory-limits")` and `help("Memory")`.

3.4.1 Exercises

These exercises aim to condense the previous section into the key points.

1. Are you using a 32-bit or 64-bit operating system?
2. Are you using 32-bit or 64-bit version of R?
3. What are the results of running the command `memory.limit()`?

3.5 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 our code 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 3.3).

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

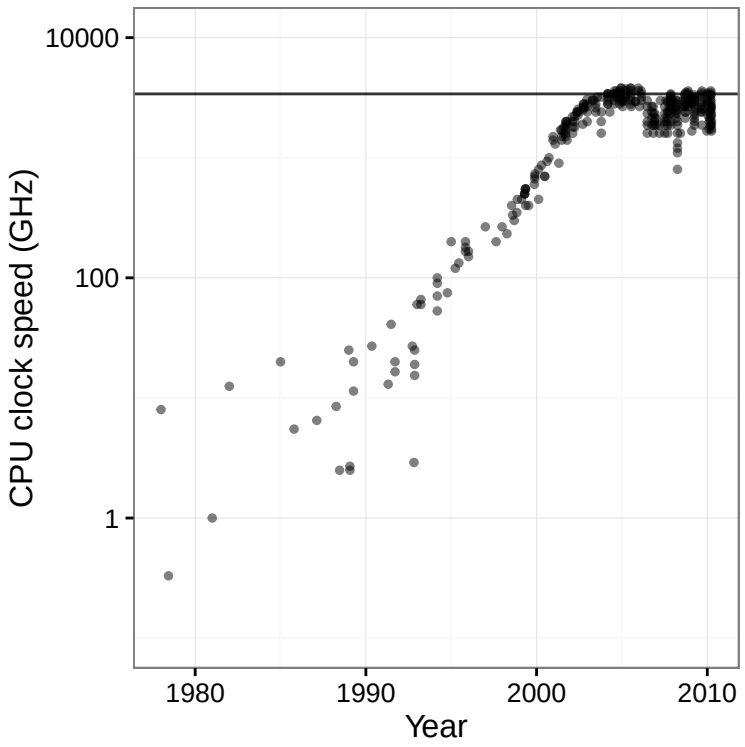


Figure 3.3: CPU clock speed. The data for this figure was collected from web-forum and wikipedia. It is intended to indicate general trends in CPU speed.

for benchmarking your system and contains data from previous benchmarks. Figure 3.4 shows the relative performance for eight different CPUs. All eight processors are Intel. The fastest processor in this benchmark study was the i7 CPU 870, which was released in 2009. In comparison, the Xeon Processor X3220, released in 2007, is over three times slower.

3.6 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.

3.6.1 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.

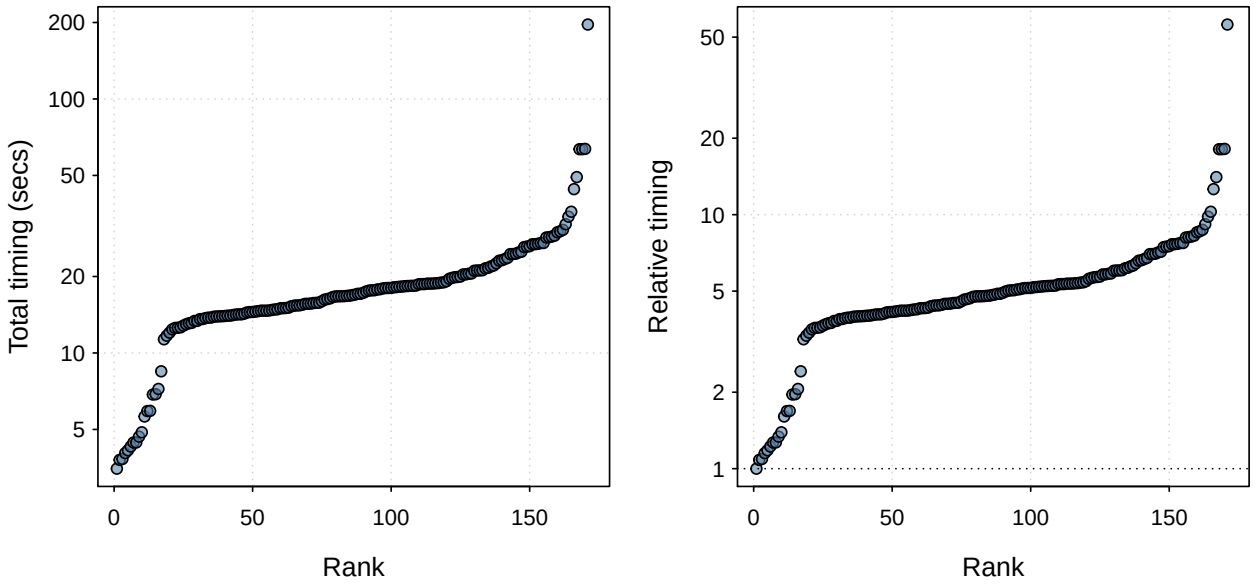


Figure 3.4: CPU benchmarks from the R package, **benchmarkme**. Each point represents an individual CPU result.

3.6.2 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 4

Efficient workflow

- **Note to reviewers:** We are planning to split this chapter in two and create a separate one on "Efficient data carpentry" to discuss data manipulation and possibly loading as a self-standing and very important part of a typical R project workflow. Feedback on this suggestion would be very welcome.

Efficient programming is an important and sometimes vital skill for generating the correct result, on time. Yet coding is only one part of a wider skillset needed for successful project outcomes which involve R programming. 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. We recommend trying different working practices to discover which works best for you.[2]

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 poor packages or a mistaken analysis strategy.
- Package selection. After planning your project you should identify which packages are most suitable to get the work done quickly and effectively. With the burgeoning number of packages available, and the phenomenon that some R packages now perform better than base R for certain functions (`*_join`, for example, is better than `merge`).
- Importing data. This can depend on external packages and represent a time-consuming and computational bottle-neck that prevents progress.
- Tidying the data. This critical stage results in datasets that are convenient for analysis and processing, with implications for the efficiency of all subsequent stages (Wickham 2014b).
- Data processing. This stage involves manipulating data to assist in the answering of hypotheses. The focus is on the **dplyr** and **data.table** packages. These are designed to make this stage both fast to type process.

[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] 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`.

- 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 4.7 touches on documentation using knitr and the much stricter approach to code publication of package development.

4.1 Project planning

Good programmers embarking 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, the trail-and-error approach — typing code at random and Googling the inevitable error messages — is usually highly *inefficient*. Strategic thinking is necessary.

Often the best place to start is often with a pen and paper. Project planning is a non-linear, open-ended and creative process not always well-suited to the linear logic of computing.[3] Planning simply involves thinking about the project's aims in the context of available resources (e.g. computational vs programming skill), the project's scope, timelines and suitable software (i.e. R packages, covered in the next section). Minutes spent before a single line of code is written have the potential to save hours later on. There are many excellent guides available that will help you develop a project plan.

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.

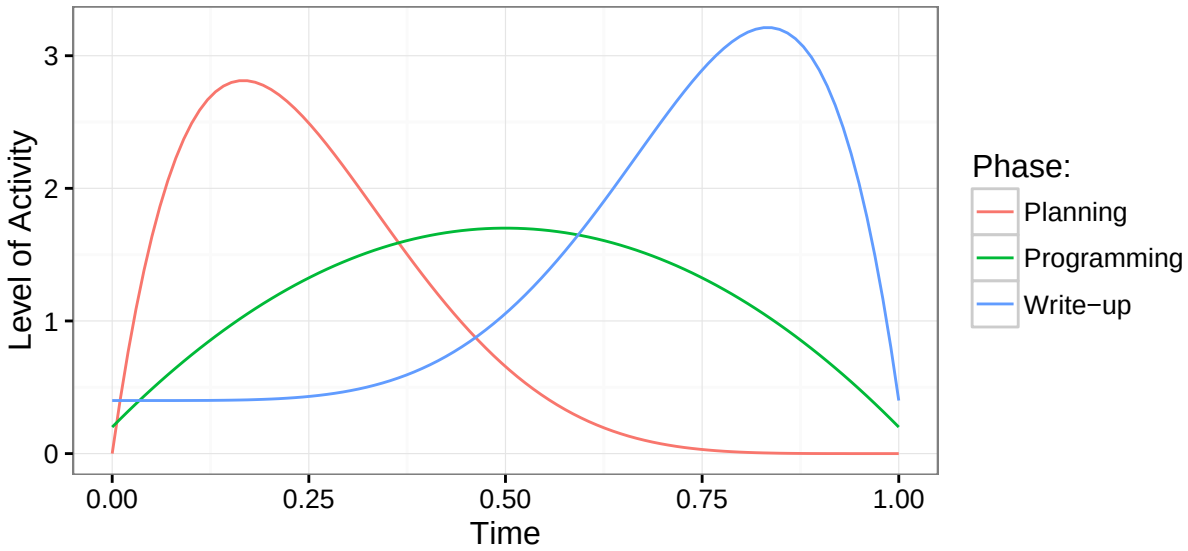


Figure 4.1: Schematic illustrations of key project phases and levels of activity over time, based on PMBoK (2000).

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'.

[3] A number of programs have been developed to assist project management and planning, however. These include ProjectLibre and GanttProject.

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.

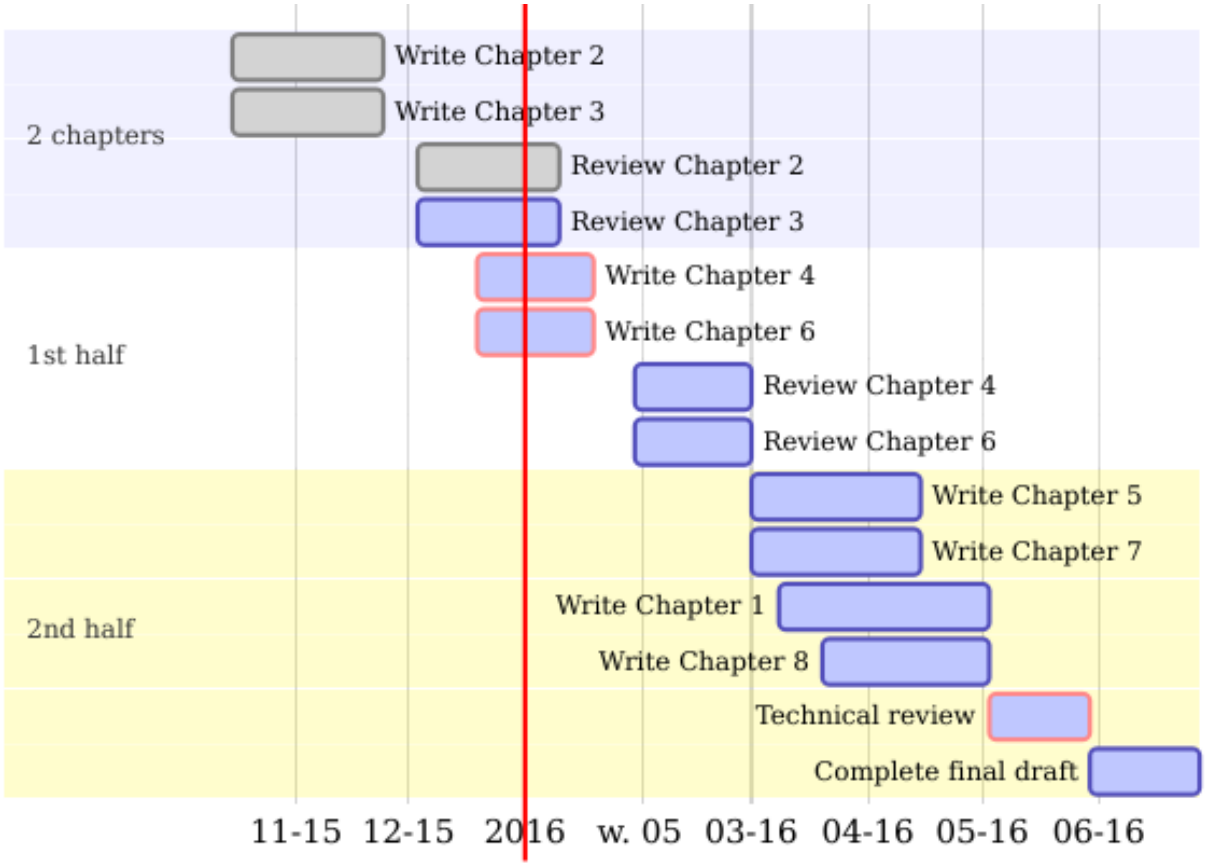


Figure 4.2: A Gantt chart created using **DiagrammeR** illustrating the steps needed to complete this book at an early stage of its development.

A number of R packages can assist with this process of formalising and visualising the project plan, including:[4]

- plan provides basic tools to create burndown charts (which concisely show whether a project is on-time or not) and Gantt charts.

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

- plotrix, a general purpose plotting package, provides basic Gantt chart plotting functionality. See `example(gantt.chart)` for details.
- **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.

```
library("DiagrammeR") # load the necessary package

# 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.

4.1.1 Exercises

1. What are the three most important work 'chunks' of your current R project?
2. What is the meaning of 'SMART' objectives?
3. Run the code chunk at the end of this section to see the output.
4. Bonus exercise: modify the code to create a very basic Gantt chart of an R project you are working on.

4.2 Package selection

A good example of the importance of prior planning to minimise effort 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*.

However, used well, packages can greatly improve productivity. Due to the conservative nature base R development, which prioritises stability, 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 'hadleyverse', 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 10th June 2007 and that it has had 28 releases. The most recent of these at the time of writing was **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.

4.3 Importing data

Before reading in 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 including a comment about its origin are a couple of ways to improve reproducibility, even when the data are not publicly available. This is illustrated below with functions `download.file`[6] and `unzip` to download and unzip the dataset,[7] illustrating how R can be used to automate processes that are conventionally done by hand. The result is data stored in the `data` directory ready to be read-in (note part of the dataset is also stored in the **efficient** package).

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

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

[7] This is a multi-table dataset on Dutch naval expeditions used with permission from the CWI Database Architectures Group and described more fully at monetdb.org.

```
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
```

To avoid the file download stage, many functions for reading in data can accept urls and read directly from the internet. This is illustrated below for `read.csv()`:

```
url = "https://www.osha.gov/dep/fatcat/FatalitiesFY10.csv"
df = read.csv(url)
```

There are now many R packages designed specifically 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) which accesses the World Bank's World Development Indicators:

```
library("WDI") # load the WDI library (must be installed)
WDIsearch("CO2") # search for data on a topic
df = 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.

4.3.1 Fast data reading

There is often more than one way to read data into R. Even a simple `.csv` file can be imported using a range of methods, with implications for computational efficiency. This section looks at three approaches: base R's reading functions such as `read.csv`, which are derived from `read.table`; the **data.table** approach, which uses the function `fread`; and the new **readr** package which provides `read_csv` and other `read_` functions such as `read_tsv`.

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 4.3 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.csv` 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.csv`. This efficiency gain may be inconsequential for a one-off file of 100 MB running on a fast computer (which still take 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 (4 GB) .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 10 GB of RAM, making it unsuitable to run on many computers (see Section 3.2 for more on memory). Note that both **readr** and base methods can be made faster by pre-specifying the column types at the outset, as illustrated in 4.3 and described in the help files.

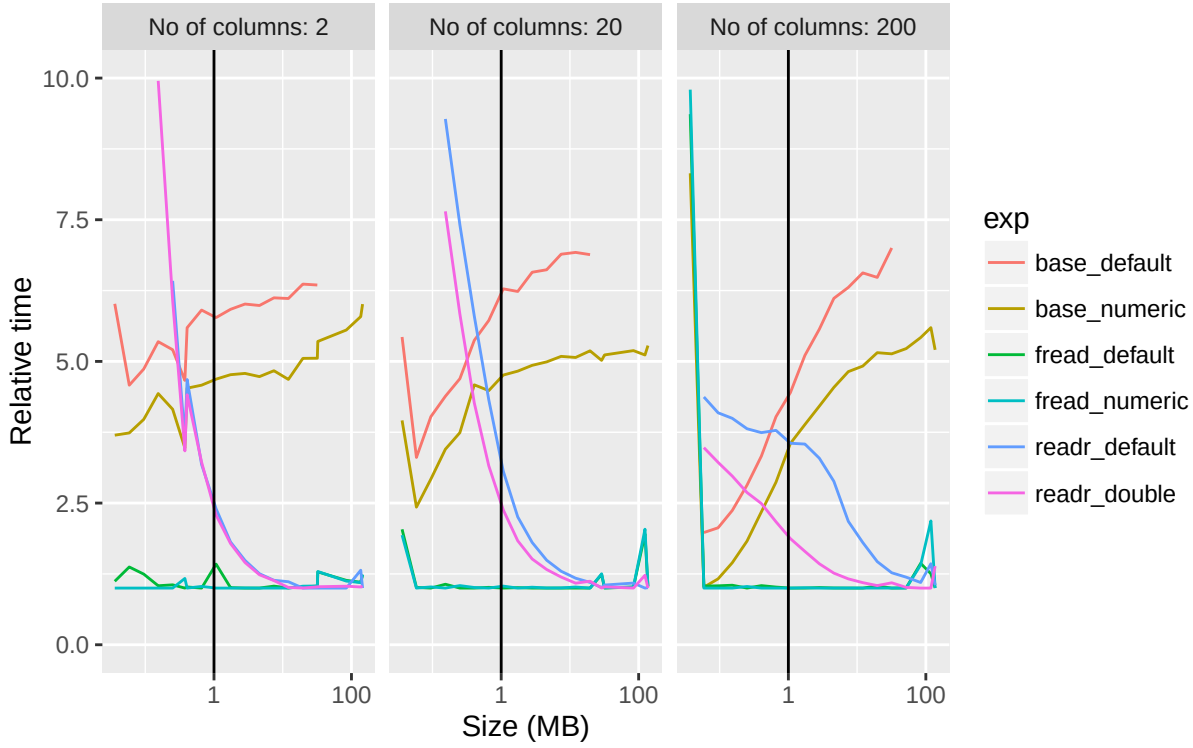


Figure 4.3: Benchmarks of base vs **data.table** vs **readr** functions for reading csv files. The facets ranging from 2 to 200 represent the number of columns.

Table 4.1: 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

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 4.1 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 4.3). 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")
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.csv`. An exploration of these can shed light on the speed/robustness trade-off.

- The **readr** function 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 probably more sensible than the `read.csv` output: the latter coerced the date field into a factor based on a single entry which is a text whereas the latter coerced the variable into a numeric data, 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.

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 4.2, which shows 4 main similarities and differences:

- For uniform data such as the 'number' variable in Table 4.2, 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`.
- `read_*` functions generate objects of class `tbl_df`, an extension of the `data.frame`, as discussed in Section 4.5. `fread` generates objects of class `data.table`. These can be used as standard data frames but differ subtly in their behaviour.

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.

Table 4.2: 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

4.3.2 Preprocessing outside R

There are circumstances when datasets become too large to read directly into R. Reading in 4 GB text file using the functions tested above, for example, consumed all available RAM on an 16 GB machine! To overcome the limitation that R reads all data directly into RAM, external *stream processing* tools can be used to preprocess large text files. The following command, using the shell command `split`, for example, would 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 Section **??**) without crashing most modern computers.

Splitting a large file into individual chunks may allow it to be read into R . But is not an efficient way to import large datasets because you will only ever have access to a non-random sample of the data this way. A more efficient way to work with very large datasets is via databases.

4.3.3 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. The most mature of these is via the RODBC package, which sets up links to external databases using the Open Database Connectivity (ODBC) API, as described in the packages vignette (which can be accessed with `vignette("RODBC")`, once the package has been installed). RODBC connects to 'traditional' databases such as MySQL, PostgreSQL, Oracle and SQLite.

The function used to set-up a connection to an external database with RODBC is `odbcConnect`, which takes Data Source Name (`dsn =`), User ID (`uid =`) and password (`pwd`) as required arguments. 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 2.3.3. 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 'noSQL' approach to data storage for handling 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. A new development showcased in this tutorial is the ability to interact with databases using exactly the same syntax used to interact with R objects stored in RAM. This innovation was made possible by **dplyr**, an R library for data processing that aims to provide a unified 'front end' to perform a wide range of analysis task on datasets using a variety of 'back ends' which do the number crunching. This is one of the main advantages of **dplyr** (see Section 4.5).

Table 4.3: 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 4.4: First 3 and last rows of the 'tidied' Pew dataset.

religion	Income	Count
Agnostic	<$10k	27
Atheist	<$10k	12
Buddhist	<$10k	27
...	...	...
Unaffiliated	>150k	258

4.4 Tidying 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 (Wickham 2014b). Converting data into a 'tidy' form (described below) is also advantageous from a computational efficiency perspective: it usually faster to run analysis and plotting commands on a few large vectors than many short vectors.

This is illustrated by Tables 4.3 and 4.4, provided by Wickham (2014b).

These tables may look different, but they contain precisely the same information Column names in the 'flat' form in Table 4.3 became a new variable in the 'long' form in Table 4.4. 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 (i.e. tables) the 'wide' or 'untidy' form may be better.

Tidy data has the following characteristics (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, as illustrated below.

- 'Wide' tables can become 'long', so that column names become a new variable. This is illustrated in Table 4.3 and Table 4.4 and can be achieved with the function `gather`:[8]

```
raw = read_csv("data/pew.csv") # read in the 'wide' dataset
dim(raw)
```

```
## [1] 18 10
```

[8]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 `read_csv` uses the `tbl` class from the **dplyr** package (described below).

Table 4.5: Joined age and sex variables in one column

agesex	n
m0-10	3
f0-10	5

Table 4.6: Age and sex variables separated by the funtion 'separate'.

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

```
rawt = gather(raw, Income, Count, -religion)
dim(rawt)
```

```
## [1] 162   3
```

```
rawt[1:3,]
```

```
## Source: local data frame [3 x 3]
##
##   religion Income Count
##      (chr)  (chr) (int)
## 1 Agnostic  <$10k    27
## 2  Atheist  <$10k    12
## 3 Buddhist  <$10k    27
```

- Splitting compound variables in two. 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 the difference between Table 4.5 and 4.6:

```
agesex = c("m0-10", "f0-10") # create compound variable
n = c(3, 5) # create a value for each observation
df = data.frame(agesex, n) # create a data frame
separate(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")`). Data manipulation is a large topic with major potential implications for efficiency (Spector 2008).

4.5 Data processing

Tidy data is easier to process than messy data. As with many aspects of R programming there are many ways to process a dataset, some more efficient than others. Following our own advice to decide appropriate packages for the work early on (see Section 4.2) uses **dplyr**, which has a number of advantages compared with base R and **data.table**:

- **dplyr** is fast to run and intuitive to type
- **dplyr** works well with tidy data, as described above
- **dplyr** works well with databases, providing efficiency gains on large datasets

```
library("readr")
fname = system.file("extdata/world-bank-ineq.csv", package = "efficient")
idata = read_csv(fname)
idata # print the dataset
```

dplyr is much faster than base implementations of various operations, but it has the potential to be even faster, as *parallelisation* is planned and the multidplyr package, a parallel backend for **dplyr**, is under development.

You should not be expecting to learn the **dplyr** package 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, building on the previous section on tidying data.

4.5.1 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.[9]

```
library("dplyr")
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:data.table':
##
##     between, last
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
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.

[9] 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'.

```
# 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")
```

Now we have usefully renamed the object we save the result for future reference:

```
saveRDS(idata, "data/idata-renamed.Rds")
```

4.5.2 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 `sapply()`, as illustrated below, with the inequality data loaded previously.

```
idata = readRDS("data/idata-renamed.Rds")
sapply(idata, class)
```

```
##      Country Country Code         Year    Year Code        top10
##  "character"  "character"    "integer"  "character"  "character"
##        bot10         gini     b40_cons   gdp_percap
##  "character"  "character"  "character"  "character"
```

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] 40.50363
```

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 a `data.frame` to a numeric `matrix`:

```
id = 5:9 # column ids to change
idata[id] = data.matrix(idata[id])
sapply(idata, class)
```

```
##      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 acheive the same result uses **dplyr**'s `mutate_each` function:

```
idata = mutate_each(idata, funs(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.

4.5.3 Filtering rows

The standard way to subset data by rows in R is with square brackets, for example:

```
aus1 = idata[idata$Country == "Australia",]
```

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

```
aus2 = filter(idata, Country == "Australia")
```

In addition to being more flexible (see the documentation provided by `?filter`), `filter` is slightly faster than base R's notation.[10] 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.[11]

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 4.5.5. For consistency, however, we next look at filtering columns.

4.5.4 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:

[10] 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.

[11] 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.

```
fname = system.file("extdata/miniaa", package = "efficient")
df = read.csv(fname) # load imaginary large data
dim(df)
```

```
## [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(df)
```

```
# $ NPI                   (int) 1679576722, ...
# $ Entity Type Code      (int) 1, 1, 2,    ...
# $ Replacement NPI       (lgl) NA, NA, NA, ...
# ...
```

Looking at the output, it becomes clear that the majority of the variables only contain `NA`. To clean the giant dataset, removing the empty columns, we need to identify which these variables are.

```
# Identify the variable which are all NA
all_na = sapply(df, function(x) all(is.na(x)))
summary(all_na) # summary of the results
```

```
##    Mode   FALSE    TRUE    NA's
## logical      96     233       0
```

```
df1 = df[!all_na] # subset the dataframe
```

The new `df` object has fewer than a third of the original columns. Another way to save storage space, beyond removing the superfluous columns, is to save the dataset in R's binary data format:

```
saveRDS(df1, "data/miniaa.Rds")
```

4.5.4.1 Exercises

1. How much space was saved by reducing the number of columns? (Hint: use `object.size()`.)
2. How many times smaller is the .Rds file saved above compared with the .csv file? (Hint: use `file.size()`.)

4.5.5 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' dataset stored in the **efficient** package.

```
fname = system.file("extdata/ghg-ems.csv", package = "efficient")
df = read.csv(fname)
names(df)
```

```
## [1] "X"
## [2] "Country"
## [3] "Year"
## [4] "Electricity.Heat..CO2...MtCO2."
## [5] "Manufacturing.Construction..CO2...MtCO2."
## [6] "Transportation..CO2...MtCO2."
## [7] "Other.Fuel.Combustion..CO2...MtCO2."
## [8] "Fugitive.Emissions..CO2...MtCO2."
```

```
nrow(df)
```

```
## [1] 7896
```

```
length(unique(df$Country))
```

```
## [1] 188
```

Note that while there are almost 8000 rows, there are less than 200 countries. Referring back to Section 4.5.1, the next stage should be to rename the columns so they are more convenient to work with. Having checked the verbose column names, this can be done in base R using the following command:

```
names(df)[4:8] = c("ECO2", "MCO2", "TCO2", "OCO2", "FCO2")
```

After the variable names have been updated, we can aggregate.[12]

```
e_ems = aggregate(df$ECO2, list(df$Country), mean, na.rm  = TRUE)
nrow(e_ems)
```

```
## [1] 188
```

Note that the resulting data frame has the same number of rows as there are countries: the aggregation has successfully reduced the number of rows we need to deal with. Now it is easier to find out per-country statistics, such as the three lowest emitters from electricity production:

```
head(e_ems[order(e_ems$x),], 3)
```

```
##     Group.1          x
## 77  Iceland 0.01785714
## 121   Nepal 0.02333333
## 18    Benin 0.04642857
```

Another way to specify the `by` argument is with the tilde (`~`). The following command creates the same object as `e_ems`, but with less typing.

[12]Note the first argument in the function is the vector we're aiming to aggregate and the second is the grouping variable (in this case Countries). A quirk of R is that the grouping variable must be supplied as a list. Next we'll see a way of writing this that is neater.

```
e_ems = aggregate(ECO2 ~ Country, df, mean, na.rm  = TRUE)
```

To aggregate the dataset using **dplyr** package one would divide the task in two: to *group* the dataset first and then to summarise, as illustrated below:

```
library("dplyr")
group_by(df, Country) %>%
  summarise(mean_eco2 = mean(ECO2, na.rm  = TRUE))
```

```
## Source: local data frame [188 x 2]
##
##              Country   mean_eco2
##               (fctr)       (dbl)
## 1        Afghanistan         NaN
## 2            Albania   0.6411905
## 3            Algeria  23.0147619
## 4             Angola   0.7914286
## 5  Antigua & Barbuda         NaN
## 6          Argentina  39.1054762
## 7            Armenia   1.8000000
## 8          Australia 150.5961905
## 9            Austria  17.3202381
## 10        Azerbaijan  16.0430435
## ..               ...         ...
```

```
countries = group_by(idata, Country)
summarise(countries, gini = mean(gini, na.rm  = TRUE))
```

```
## Source: local data frame [176 x 2]
##
##         Country     gini
##           (chr)    (dbl)
## 1   Afghanistan      NaN
## 2       Albania 30.43167
## 3       Algeria 37.76000
## 4        Angola 50.65000
## 5     Argentina 48.06739
## 6       Armenia 33.72929
## 7     Australia 33.14167
## 8       Austria 29.15167
## 9    Azerbaijan 24.79000
## 10 Bahamas, The      NaN
## ..          ...      ...
```

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)
  )
```

```
## Source: local data frame [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  24.435  1.252524     3  0.3642801  2.78
## 3       Algeria    40  29.780  3.436539     2  3.4365390  4.86
## 4        Angola    40  38.555 11.299566     2 11.2995664 15.98
## 5     Argentina    40  36.320  3.182462    23  3.1824622 11.00
## 6       Armenia    40  27.835  4.019532    12  3.9567778 14.84
## 7     Australia    40  24.785  1.075089     6  1.0750891  2.81
## 8       Austria    40  23.120  3.120849     4  0.6859300  8.48
## 9    Azerbaijan    40  17.960  9.479029     3  1.7386489 20.27
## 10 Bahamas, The    40      NA       NaN     0         NA    NA
## ..          ...   ...     ...       ...   ...        ...   ...
```

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.

4.5.5.1 Exercises

1. Referring back to Section 4.5.1, rename the variables 4 to 8 using the **dplyr** function `rename`. Follow the pattern `ECO2`, `MCO2` etc.
2. Explore **dplyr**'s documentation, starting with the introductory vignette, accessed by entering `vignette("introduction")`.
3. Test additional **dplyr** 'verbs' on the `idata` dataset. (More vignette names can be discovered by typing `vignette(package = "dplyr")`.)

4.5.6 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 %>%
  filter(grepl("g", Country)) %>%
  group_by(Year) %>%
  summarise(gini = mean(gini, na.rm  = TRUE)) %>%
  arrange(desc(gini)) %>%
  top_n(n = 5)
```

```
## Selecting by gini

## Source: local data frame [5 x 2]
##
##    Year   gini
##   (int)  (dbl)
## 1  1980 46.850
## 2  1993 45.996
## 3  2013 44.550
## 4  1981 43.650
## 5  2012 43.560
```

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 2016).

4.6 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[13] so it should be stated at the outset

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

that **dplyr** and **data.table** are not mutually exclusive competitors or that must be 'better' than another. These are both excellent packages and the important thing from an efficiency perspective is that they can help speed up data processing tasks.

The foundational object class of **data.table** is the `data.table`. Like **dplyr**'s `tbl_df` `data.table` objects will 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 4.5.4, 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 4.4, which compares 4 methods of subsetting incrementally larger versions of the `idata` dataset.

Figure 4.4 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 4.3.1), the relative benefits of **data.table** improve with dataset size, approaching a ~70 fold improvement speed gain 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 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.

This section has introduced **data.table** as a complimentary approach to base and **dplyr** methods for data processing and illustrated the performance gains of using *keys* for subsetting tables. **data.table** is a mature and powerful package which uses clever computational principles implemented in C to provide efficient methods for a number of other operations for data analysis. These include highly efficient data reshaping, dataset merging (also known as joining, as with `left_join` in **dplyr**) and grouping. These are explained in the vignettes `datatable-intro` and `datatable-reshape`. The `datatable-reference-semantics` vignette explains **data.table**'s unique syntax.

4.7 Publication

The final stage in a typical project workflow is publication. This could be 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. In every case the programming principles of reproducibility, modularity and DRY discussed in 6 will make your publications faster to write, easier to maintain and more useful to others.

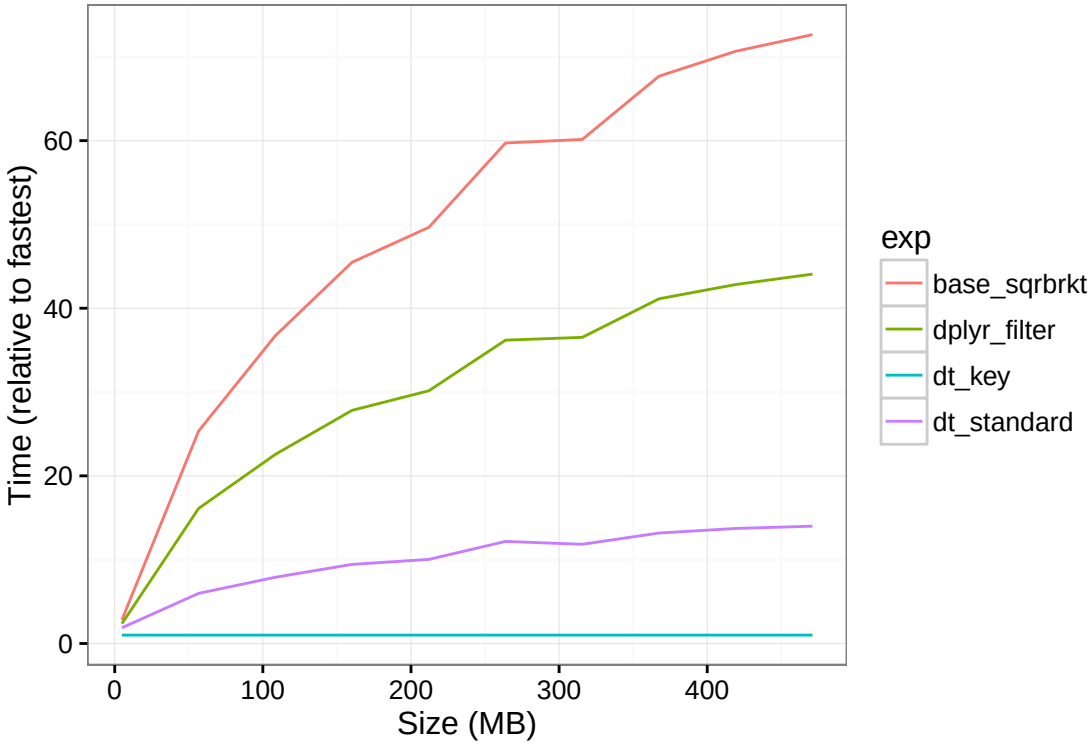


Figure 4.4: Benchmark illustrating the performance gains to be expected for different dataset sizes.

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 exits on each of these; to avoid duplication of effort (following the DRY principle) the focus is on documentation from a workflow efficiency perspective.

4.7.1 Dynamic documents

When writing a report using R outputs a typical workflow has historically been to a) do the analysis b) save the resulting graphics and record the main results outside the R project and c) 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 a to c will have to be done again) and there is an overhead involved in jumping between incompatible computing environments.

To overcome this source of 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. Results are generated *on the fly* by including 'code chunks' such as that illustrated below:

```
(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 `"‘{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 make it easy to create and modify code chunks. 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.

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).

4.7.2 R packages

A strict approach to project management and workflow is treating your projects as R packages. This is good practice in terms of learning to correctly document your code, store example data, and even (via vignettes) ensure reproducibility. 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.

The a number of key elements of an 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.
- 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')`.

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 rigourous way of storing code, data and documentation that can greatly boost productivity in the long-run. For more on R packages see (Wickham 2015).

Chapter 5

Efficient collaboration

Chapter 6

Efficient programming

In this chapter we will discuss key R data types and idiomatic programming style. Many people that use R would not describe themselves as "programmers". Instead, they have advanced domain level knowledge, but little formal training in programming. This chapter comes from their point of view; someone who has use standard R data structures, such as vectors and data frames, but has never looked as the inner workings of these objects.

We begin this chapter by discussing key data types, how they are used and potential computational gains available. Once we understand these objects, we will look at key R programming idioms, before covering techniques for speeding up code.

6.1 Data types

A data type is an object that has a set of predefined characteristics, such as a number or a character. When programming in C or FORTRAN, the data type of every object must be specified by the user. he advantage is that it allows the compiler to perform type-specific optimisation. The downside is verbose and fragile code, which is inefficient to type. In R data types are less critical, but understanding them will help you debug and optimize for computational efficiency. Essentially, we have a trade-off between CPU run time and developer thinking time. However an understanding of data types can help when debugging and optimizing for computational efficiency. In this chapter, we will pick out the key point data types from an efficiency perspective. Chapter 2 of Advanced R Programming (Wickham 2014a) provides a more comprehensive treatment.

6.1.1 Vectors

The vector is a fundamental data structure in R. Confusingly there are two varieties:

- Atomic vectors are where all elements have the same type and are usually created using the `c()` function;
- Lists are where elements can have different types.

To test if an object is a vector, we must use `is.atomic(x) || is.list(x)`. The more obvious choice for determining if an object is a vector, `is.vector(x)`, only returns `TRUE` is an object is a vector with no attributes other than names. For example, when we use the `table` function

```
x = table(rpois(100, 5))
```

the object `x` has additional attributes (such as `dim`), so `is.vector(x)` return `FALSE`. But the contents `x` is clearly a vector, so `is.atomic(x)` returns `TRUE`.

The core vector data types are logicals, integers, doubles and characters. When an atomic vector is created with a mixture of types, the output type is coerced to highest type in the following hierarchy:

```
logical < integer < double < character
```

This means that any vector containing a character string will be coerced to class, as illustrated below.

6.1.1.1 Numerics: doubles and integers

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 occupies 8 bytes and is accurate to around 17 decimal places (R does not print all of these, as you will see by typing `pi`). Somewhat surprisingly, when we run the command

```
x = 1
```

we have created an atomic vector, contain a single double-precision floating point number. When comparing floating point numbers, we should be particularly careful, since

```
y = sqrt(2)*sqrt(2)
y == 2
```

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

This is because the value of `y` is not exactly 2, instead it's **almost** 2

```
sprintf("%.16f", y)
```

```
## [1] "2.0000000000000004"
```

To compare numbers in R it is advisable to use `all.equal` and set an appropriate tolerance, e.g.

```
all.equal(y, 2, tolerance = 1e-9)
```

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

Although using double precision objects is the most common type, R does have other ways of storing numbers:

- `single`: R doesn't have a single precision data type. Instead, all real numbers are stored in double precision format. The functions `as.single` and `single` are identical to `as.double` and `double` except they set the attribute `Csingle` that is used in the `.C` and `.Fortran` interface.
- `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
```

The default argument 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 = seq(1, 100, 1.0)
microbenchmark(clock_speed[s_int, 2L], clock_speed[s, 2.0], times=1000000L)
```

```
## 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, 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.

- `numeric`: The function `numeric()` is identical to `double()`; it creates is a double-precision number. However, `is.numeric()` isn't the same as `as.double()`, instead `is.numeric()` returns `TRUE` for both numeric and double types.

To find out the type of data stored in an R vector use the command `typeof()`:

```
typeof(c("a", "b"))
```

```
## [1] "character"
```

6.1.1.2 Exercises

A good way of determining how to use more advanced programming concepts, is to examine the source code of R.

1. What are the data types of `c(1, 2, 3)` and `1:3`?
2. Have a look at the following function definitions:
   - `tail.matrix`
   - `lm`
3. How does the function `seq.int`, which was used in the `tail.matrix` function, differ to the standard `seq` function?

6.1.2 Factors

A factor is useful when you know all of the possible values a variable may take. For example, suppose our data set related to months of the year

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

If we sort `m` in the usual way `sort(m)`, we use standard alpha-numeric ordering, placing December first. While this is completely correct, it is also 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
```

Most users interact with factors via the `read.csv` function where character columns are automatically converted to factors. It is generally recommended to avoid this feature using the `stringsAsFactors=FALSE` argument. Although this argument can be also placed in the global `options()` list, this leads to non-portable code, so should be avoided.

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

```
c(m)
```

```
## [1] "January"  "December" "March"
```

```
c(fac_m)
```

```
## [1]  1 12  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.

In early versions of R, storing character data as a factor was more space efficient. However since identical character strings now share storage, the space gain in factors is now space.

6.1.3 Data frames

A data frame is a tabular (two dimensional or 'rectangular') object in which the columns may be composed of differing vector types such as `numeric`, `logical`, `character` and so on. Matrices can only accept a single data type for all cells as explained in the next section. Data frames are the workhorses of R. Many R functions, such as `boxplot`, `lm` and `ggplot`, expect your data set to be in a data frame. As a general rule, columns in your data should be variables and rows should be the thing of interest. This is illustrated in the `USAarrests` data set:

```
head(USArrests, 2)
```

```
##         Murder Assault UrbanPop Rape
## Alabama   13.2     236       58 21.2
## Alaska    10.0     263       48 44.5
```

Note that each row corresponds to a particular state and each column to a variable. One particular trap to be wary of is when using `read.csv` and `read.table` characters are automatically converted to factors. One can avoid this pitfall by using the argument `stringsAsFactors = FALSE`.

Since working with R frequently involves interacting with data frames, it's useful to be fluent a few key functions:

Table 6.1: Useful data frame functions.

Name	Description
`dim`	Data frame dimensions
`ncol/nrow`	No. of columns/rows
`NCOL/NROW`	As above, but also works with vectors
`cbind/rbind`	Column/row bind
`head/tail`	Select the first/last few rows
`colnames/rownames`	Column and row

When loading a dataset called `df` into R, a typical workflow would be:

- Check dimensions using `dim(df)`;
- Look at the first/last few rows using `head(df)` and `tail(df)`;
- Rename columns using `colnames(df) =`.

6.1.4 Matrix

A matrix is similar to a data frame: it is a two dimensional object and sub-setting and other functions work in the same way. However all matrix columns must have the same type. Matrices tend to be used during statistical calculations. Linear regression using `lm()`, for example, 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. The datasets `ex_mat` and `ex_df` from the **efficient** package each have 1000 rows and 100 columns. They contain the same random numbers. However, selecting rows from a data frame is around 150 times slower than a matrix. This illustrates the reason for using matrices instead of data frames for efficient modelling in R:

```
data(ex_mat, ex_df, package="efficient")
benchmark(replications=10000,
          ex_mat[1,], ex_df[1,],
          columns=c("test", "elapsed", "relative"))
```

```
##          test elapsed relative
## 2  ex_df[1, ]  12.506  223.321
## 1 ex_mat[1, ]   0.056    1.000
```

6.1.5 S3 objects

R has three built-in object oriented 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 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)`.

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

```
## [1] "data.frame"
```

The S3 system can be used to great effect. For example, a `data.frame` is simply a standard R list, with class `data.frame`. When we pass an object to a *generic* function, the function first examines the class of the object, and then decides what to do: it dispatches to another method. The generic `summary` function, for example, contains the following:

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x6383928>
## <environment: namespace:base>
```

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`.

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

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

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

```
image
```

```
## function (x, ...)
## UseMethod("image")
## <bytecode: 0x77afe88>
## <environment: namespace:graphics>
```

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 6.1.

Many S3 methods work in the same way as the simple `image.dist` function created above: the object is converted 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.

6.1.5.1 Exercises

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

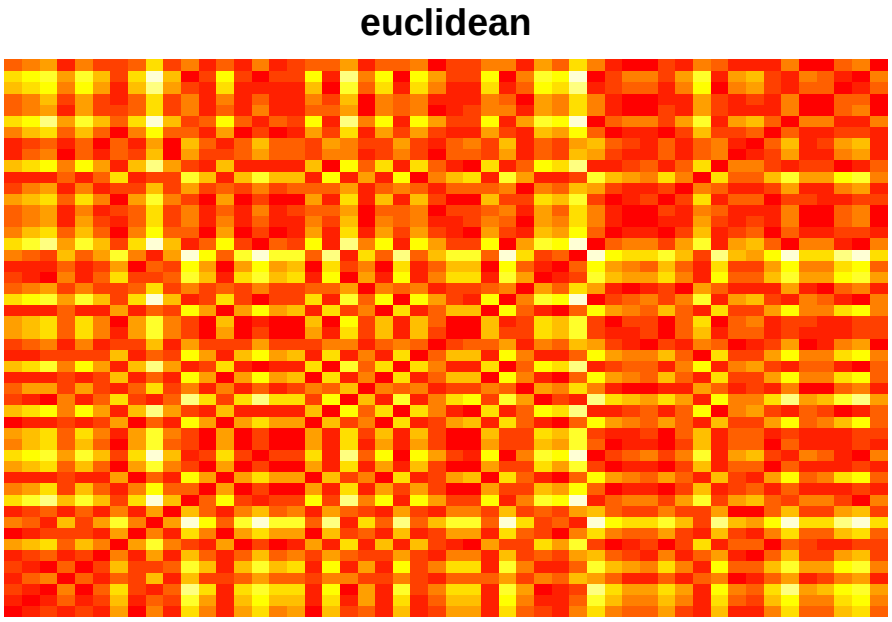


Figure 6.1: S3 image method for data of class ‘dist‘.

6.1.6 Efficient data structures

Even when our data set is small, the analysis can generate large objects. For example suppose we want to perform standard cluster analysis. Using the built-in data set `USAarrests`, we calculate a distance matrix:

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

The resulting object `dist_usa` measures the similarity between two states with respect to the input data. Since there are 50 states in the `USAarrests` data set, this results in a matrix with 50 columns and 50 rows. Intuitively, since the matrix `dist_usa` is symmetric around the diagonal, it makes sense to exploit this characteristic for efficiency, allowing storage to be halved. If we examine the object `dist_usa`, with `str(dist_usa)`, it becomes apparent that the data is efficiently stored as a vector with some attributes.

Another efficient data structure is a sparse matrix. This is simply a matrix in 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. For this `dist` object, since the structure is regular.

6.2 Good programming techniques

A major benefit of using R (as opposed to C or Fortran, say), is that coding time is greatly reduced. However if we are not careful, it's very easy to write programs that are incredibly slow. While optimisations such as going parallel can easily double speed, poor code can easily run 100s of times slower. For this reason a priority of an efficient programmer should be to avoid the following common mistakes. If you spend any time programming in R, then reading (Burns 2011) should be considered essential reading.

6.2.1 General tips

The key to making R code run fast is to access the underlying C/Fortran routines as quickly as possible. For example, suppose that `x` is a standard R vector of length `n`. Then

```
x = x + 1
```

involves a single function call to the `+` function. Whereas,

```
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 the `:` operator.

It isn't that the `for` loop is slow, rather it is because we calling many more functions. This point is indirectly tackled again in the section on vectorised code.

Another general technique is to be careful with memory allocation. In fact this could be considered the number 1 rule when programming in R. If possible always pre-allocate your vector or data frame then fill in the values. Let's consider three methods of creating a sequence of numbers.

Method 1 creates an empty vector, and grows the object

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

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

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

Method 3 directly creates the final object

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

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

```
n = 1e4
benchmark(replications=10,
          method1(n), method2(n), method3(n),
          columns=c("test", "elapsed"))
```

```
##         test elapsed
## 1 method1(n)   2.398
## 2 method2(n)   0.137
## 3 method3(n)   0.000
```

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.

Table 6.2: Time in seconds to create sequences. When $n = 10^7$, method 1 takes around an hour while methods 2 takes 2 seconds and method 3 almost instantaneous.

n	Method 1	Method 2	Method 3
10^5	0.208	0.024	0.000
10^6	25.500	0.220	0.000
10^7	3827.0000	2.212	0.000

6.2.2 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 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

```
m_plot = memoise(plot_mpg)
benchmark(m_plot(10), plot_mpg(10), columns = c("test", "relative", "elapsed"))
#         test relative elapsed
#1   m_plot(10)    1.000   0.007
#2 plot_mpg(10)  481.857   3.373
```

suggests that we can obtain a 500-fold speed-up.

6.2.3 Function closures

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 `.GlobalEnv`. One of the environments associated with function is known as the enclosing environment, that is, where was the function created. We can determine the enclosing environment using the `environment` function

```
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: 0x7e75508>
```

When we call this new function `g`,

```
x = 10
g()
```

```
## [1] 5
```

The value returned is obtained from `environment(g)` is 5, not `.GlobalEnv`. This environment allows to cache variables between function calls. The `counter` function is basic example of this feature

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

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

```
sc = counter()
sc()
```

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

```
sc()
```

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

The key points of the `counter` function are

- The counter function returns a function

  ```
  sc = counter()
  sc()
  ```

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

- 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 `no`.

We can exploit function closures to simplify our code. Suppose we wished to simulate a games of Snakes and Ladders. We could have function that checked if we landed on a Snake, and if so move

```
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 keep track

```
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
  }
}
```

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

6.2.4 Vectorised code

When writing code in R, you need to remember that you are using R and not C (or even Fortran 77!). For example,

```
# Change 1000 uniform random numbers
x = runif(1000) + 1
logsum = 0
for(i in 1:length(x))
  logsum = logsum + log(x[i])
```

is a piece R code that has a strong, unhealthy influence from C. Instead we should write

```
logsum = 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.

Another common example is sub-setting a vector. When writing in C, we would have something like:

```
ans = NULL
for(i in 1:length(x)) {
  if(x[i] < 0)
    ans = c(ans, x[i])
}
```

This of course can be done simply with

```
ans = x[x < 0]
```

6.2.4.1 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

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

using a basic Monte-Carlo method. Essentially, we throw darts at the curve and count the number of darts that fall below the curve (as in 6.2).

Monte Carlo Integration

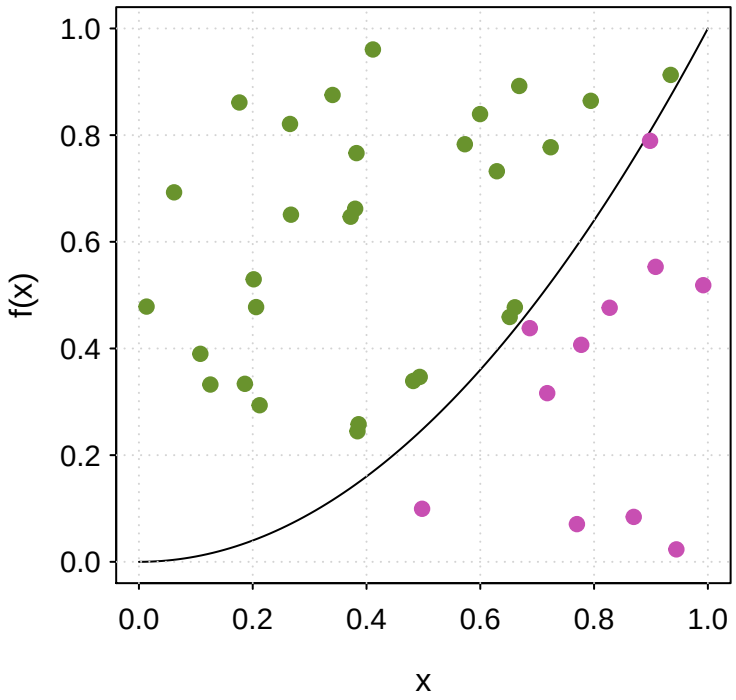


Figure 6.2: Example of Monte-Carlo integration. To estimate the area under the curve throw random points at the graph and count the number of points that lie under the curve.

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`

A standard C approach to implementing this Monte-Carlo algorithm would be something like:

```
N = 500000
f = function(N){
  hits = 0
  for(i in 1:N)  {
    u1 = runif(1); u2 = runif(1)
    if(u1^2 > u2)
      hits = hits + 1
  }
  return(hits/N)
}
```

In R this takes a few seconds:

```
system.time(f(N))
```

```
##    user  system elapsed
##   4.827   0.040   4.868
```

In contrast, a more R-centric approach would be the following:

```
f1 = function(N){
  hits = sum(runif(N)^2 > runif(N))
  return(hits/N)
}
```

`f1` is around 30 times faster than `f`, illustrating the efficiency gains that can be made by vectorising your code:

```
system.time(f1(N))
```

```
##    user  system elapsed
##   0.044   0.000   0.045
```

6.3 Parallel computing

In recent R versions (since R 2.14.0) ** 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()
```

The computer used to compile the published version of this book chapter has 32 CPUs/Cores.

6.3.1 Parallel versions of apply functions

The most commonly used parallel applications are parallelized 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)
```

Note that each function has an argument `cl` which must be created by `makeCluster`. This function, amongst other things, specifies the number of processors to use.

6.3.2 Example: parallel bootstraping

In 1965, Gordon Moore co-founder of Intel, observed that the number of transistors in a dense integrated circuit doubles approximately every two years. This observation is known as Moore's law. A scatter plot (figure 6.3) of processors over the last thirty years shows that that this law seems to hold.

We can estimate the trend using simple linear regression. A standard algorithm for obtaining uncertainty estimates on regression coefficients is bootstrapping. This is a simple algorithm; at each iteration we sample with replacement from the original data set and estimate the parameters of the new data set. The distribution of the parameters gives us our uncertainty estimate. We begin by loading the data set and creating a function for performing a single bootstrap

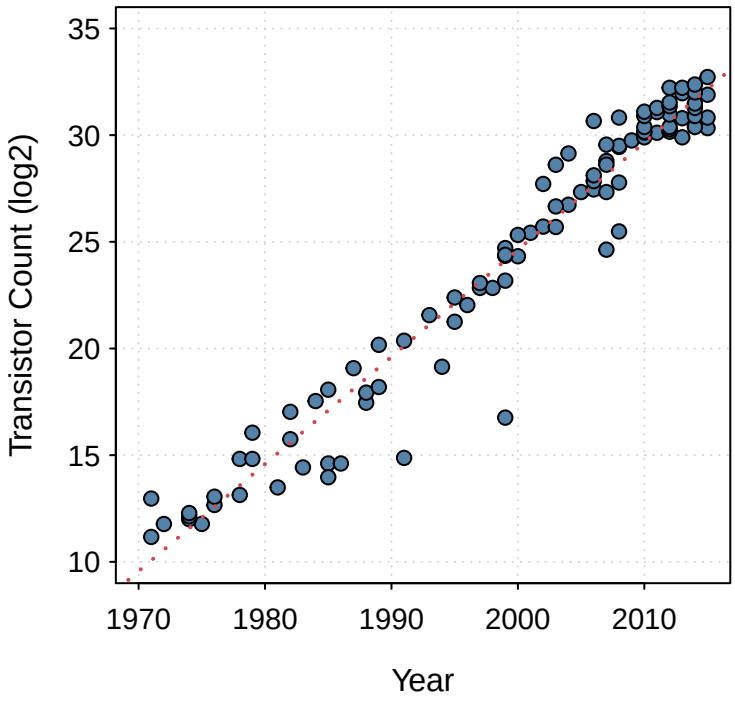


Figure 6.3: Transistor counts against introduction date. Credit: https://en.wikipedia.org/wiki/Transistor_count

```
data("transistors", package="efficient")
bs = function(i) {
  s = sample(1:NROW(transistors), replace=TRUE)
  trans_samp = transistors[s,]
  coef(lm(log2(Count) ~ Year, data=trans_samp))
}
```

We can then perform $N = 10^4$ bootstraps using `sapply`

```
N = 10000
sapply(1:N, bs)
```

Rewriting this code to make use of the ** parallel** package is straightforward. We begin by making a cluster and exporting the data set

```
library("parallel")
cl = makeCluster(6)
clusterExport(cl, "transistors")
```

Then use `parSapply` and stop the cluster

```
parSapply(cl, 1:N, bs)
stopCluster(cl)
```

On this computer, we get a four-fold speed-up.

```
stopCluster(cl)
```

6.3.3 Process forking

Another way of running code in parallel is to use the `mclapply` and `mcmapply` functions. These functions use forking 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.

6.4 The byte compiler

The ** compiler** package, written by R Core member Luke Tierney has been part of R since version 2.13.0. 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`:

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x9079848>
## <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.

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. Amazingly the compiler package is almost entirely pure R, with just a few C support routines.

6.4.1 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 `total` variable.

```
my_mean = function(x) {
  total = 0
  n = length(x)
  for(i in 1:n)
    total = total + x[i]/n
  total
}
```

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.

```
cmp_mean = cmpfun(my_mean)
```

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

```
# Generate some data
x = rnorm(100)
benchmark(my_mean(x), cmp_mean(x), mean(x),
          columns=c("test", "elapsed", "relative"),
          order="relative", replications=5000)
```

The compiled function is around seven times faster than the uncompiled function. Of course, the native `mean` function is faster, but the compiling does make a significant difference (figure 6.4).

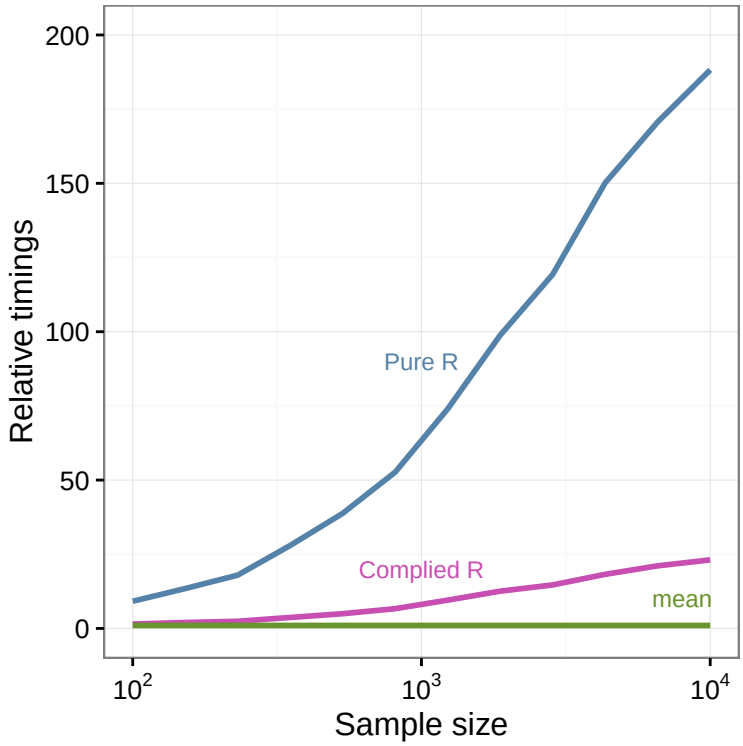


Figure 6.4: Comparsion of mean functions.

6.4.2 Compiling code

There are a number of ways to compile code. The easiest is to compile individual function using `cmpfun`, but this obviously doesn't scale. If you create a package, then you 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.

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.

Chapter 7

Efficient Rcpp

Chapter 8

Efficient Memory

Chapter 9

Efficient Learning

Berkun, Scott. 2005. *The Art of Project Management*. O'Reilly.

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.

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.

Eddelbuettel, Dirk. 2010. "Benchmarking Single-and Multi-Core BLAS Implementations and GPUs for Use with R." Mathematica.

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.

Grolemund, Garrett, and Hadley Wickham. 2016. *R for Data Science*. 1 edition. O'Reilly Media.

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.

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.

Wickham, Hadley. 2014a. *Advanced R*. CRC Press.

———. 2014b. "Tidy Data." *The Journal of Statistical Software* 14 (5).

———. 2015. *R Packages*. O'Reilly Media, Inc.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. Vol. 29. CRC Press.