Working with R

Eric Marcon

05/22/2024

This document is made in a dynamic and reproducible way thanks to:

- LaTeX, in its Miktex distribution (http://miktex.org/) and the memoir class (http://www.ctan.org/pkg/memoir).
- R (http://www.r-project.org/) and RStudio (http://www.rstudio.com/)
- bookdown (http://bookdown.org/) and memoiR (https://ericmarcon.github.io/memoiR/)

Its source code is on GitHub: https://github.com/EricMarcon/WorkingWithR/.

The continuously updated text can be read at https://ericmarcon.github.io/WorkingWithR/.

Cover photograph: Hadrien Lalagüe

Contents

PRESENTATION

Objectives

This document is the support of the course *Working with R*.

It proposes an organization of the work around R and RStudio in order to, beyond statistics, write documents efficiently with R Markdown, in various formats (memos, scientific articles, student theses, books, slideshows), create a web site and online R applications (Shiny), produce packages and use R for teaching. It complements *Reproducible Research with R and R Studio* (Gandrud 2015) with a more hands-on approach, with ready-to-use solutions.

Optimizing the use of the many tools available is covered in detail: **rmarkdown**, **bookdown** and **blogdown** for writing, **roxygen2**, **testthat** and **pkgdown** for packages, source control with git and GitHub, continuous integration with GitHub Actions and Codecov. Examples are presented at each step, and the necessary code is provided.

Chapter 1 is dedicated to the installation of the necessary tools: R, git and LaTeX. Chapter 2 details some advanced aspects of using R: the different languages, the environments, the performance of the code. The basic use of R is not covered here: good courses are suggested. Chapter 3 presents source control with git and GitHub.

Chapter 4 shows how to write simple (articles) or complex (books) documents with R Markdown, integrating the data, the code to process them and the text to present them. Chapter 5 presents a step-by-step method to efficiently create a package. Chapter 6 introduces the use of continuous integration to automatically produce documents, verify package code and produce package vignettes. Chapter 7 introduces Shiny to develop R interactive applications. Finally, chapter 8 introduces the tools for teaching R.

Conventions

Package names are in bold in the text, for example: **ggplot2**.

The identifier used on GitHub is noted *GitHubID*. Project names are the same as their GitHub repository, noted *RepoID*.

The sign `|>` in the code of the examples indicates that the rest of the code should be on the same line, but is cut for the formatting of this document. Its use

is limited to YAML configuration files, mostly in chapter 6. In all other cases, the code can be copied directly.

CHAPTER 1

SOFTWARE

The central tool is obviously R, but its operation is today difficult to consider without its development environment RStudio. For source control, git and GitHub are the de facto standards. The set must be completed by a LaTeX distribution for the production of documents in PDF format. A bibliographic management tool is essential: Zotero and its extension Better BibTeX are perfectly adapted to the framework presented here. Finally, other software of more occasional use may be necessary, such as Go.

Their installation and coherent organization are presented in this chapter.

1.1 R

1.1.1 Installation

R is included in Linux distributions: the package is named `r-base`. It does not contain development tools that are often needed, so it is better to install the `r-base-dev` package as well. The version of R is often a bit old. To get the latest version, you have to use a CRAN mirror as a source for the packages: see

the full documentation for Ubuntu[1].

On Windows or Mac, install R after downloading it from CRAN[2].

1.1.2 Rtools

On Mac, the installation of R is sufficient from version 4.0.0.

On Windows, the installation must be completed by the "Rtools", which contain the development tools, including those necessary to compile packages containing C++ code.

The path of the Rtools (before version 4.2) must be declared to R, by executing the following command in the RStudio console (adapted to version 4.0 of the Rtools):

```
# Rtools : path declaration,
# requires restarting RStudio
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"',
           con = "~/.Renviron")
```

Since version 4.2, this action is unnecessary.

The Rtools must be completed by some missing utilities, to be installed when the need appears (usually a warning from R that the software is not installed).

Package checking returns a warning if *qpdf*[3] is not installed. Download the zip file and paste the entire contents of the `bin` folder into the `usr/bin` folder of `Rtools` (`C:\Rtools42\r\bin` for version 4.2).

Another warning is returned if *Ghostscript* is not available. Download and install it[4]. Then copy the contents of the `bin` folder to the `usr/bin` folder of `Rtools`.

1.1.3 Update

It is recommended to use the latest minor version of R: for example, 4.0.x until the release of version 4.1. It is mandatory to use the latest version to prepare a package submitted to CRAN.

Important changes occur between major versions (version 4 does not allow to use a package compiled for version 3) but also sometimes between minor versions (a binary data file `.rda` saved under version 3.3 cannot be read by version 3.6). It is therefore useful to update R regularly.

Installing a new version does not automatically uninstall older versions, which allows you to use more than one version if necessary (for example, if an old and essential package is no longer available). In common use, it is preferable to uninstall old versions manually after installing a new one.

[1] https://doc.ubuntu-fr.org/r
[2] https://cran.r-project.org/
[3] https://sourceforge.net/projects/qpdf/
[4] https://www.ghostscript.com/

1.1.4 Libraries

R packages are found in two folders:

- the *System Library* contains the packages that come with R: **base**, **utils**, **graphics** for example. It is located in a subdirectory of the installation program (`C:\Program Files\R-4.1.0\library` for R version 4.1.0 on Windows 10).
- The *User Library* contains those installed by the user. Until version 4.1, it is located in the user's home folder, in a subfolder `R\win-library\4.1\`). Since version 4.2, this folder is in the user's local settings, whose folder location is in the environment variable `%LOCALAPPDATA%`.

Until version 4.1, if the user's home folder is backed up (for example, if it is replicated in the cloud by OneDrive on Windows), it is not optimal to place the packages there: the traffic generated by backing them up would be heavy and unnecessary. In order for packages to be installed automatically in the system library, the user must have the right to write to it. On Windows, give the computer's user group the "Modify" permission to the library folder, in addition to the default read permissions. From version 4.2 onwards, there is no reason to change the default operation: local settings are not saved.

If the user library is selected, you must remember to empty the folder corresponding to the old version of R in case of minor version change.

The location of the libraries is given by the function `.libPaths()`:

```
.libPaths()
```

```
## [1] "/Users/runner/work/_temp/Library"
## [2] "/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/library"
```

1.2 RStudio

RStudio is a graphical interface for R and much more: it is designed to simplify project management, make writing and publishing documents easier and integrate source control for example.

1.2.1 Installation

Install the latest version of *RStudio Desktop* from the RStudio website[5].

A command is available in the "Help" menu of RStudio to check for a more recent version to install.

[5] https://rstudio.com/products/rstudio/download/

1.2.2 File encoding

The files manipulated in R are mostly text files. Special characters, especially accents, can be encoded in various ways, but the encoding declaration is not integrated in the files. The default encoding depends on the operating system, which regularly causes problems with the readability of shared files. The UTF8 encoding has become the standard because it is universally recognized and supports all alphabets without ambiguity.

The first time you use RStudio, create a new R file ("File > New File > R Script" menu), save it in UTF8 format ("File > Save with Encoding…"), choose UTF8 in the list of formats and check the box "Set as default encoding for source files". Delete the file after saving it.

New files will be encoded in UTF8 format. Files encoded in another format will not be displayed correctly: they can be reopened with their original encoding ("File > Reopen with Encoding…"), possibly trying several encodings until they are displayed correctly, and then saved in UTF8 format.

1.2.3 Working folder

The default working folder is the user's home folder, called ~ by RStudio:

```
Sys.getenv("R_USER")
```

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

- `My Documents` on Windows.
- `Home` on Mac or Linux.

You should always work in subfolders of `~`, for example: `~/Training`.

For *RTools* to work properly, the full name of the working directory must not contain spaces (use underscores `_`) or special characters. The current working directory is obtained by the command `getwd()`.

```
getwd()
```

Using source control (see chapter 3) creates many working files. Source-controlled projects should not be located in a folder that is already backed up by another means, such as a OneDrive on Windows, otherwise resources will be used excessively: each change validation generates a backup of the modified files, but also of the control files, which can be very large.

1.2.4 Solution chosen

The organization of the work environment is a personal matter, depending on the preferences of each individual. The organization proposed here is only a possibility, to be adapted to one's own choices, but respecting the constraints mentioned.

On Windows, an optimal organization is as follows:

- In one's personal folder (`My Documents`, `~` for R), an `R` folder is used for simple projects, without source control. The backup of this folder is managed elsewhere.
- A folder outside the home folder is used for source-controlled projects. The user must have the right to write to it. In the Windows organization, the folder corresponding to these criteria is `%LOCALAPPDATA%`, typically `C:\Users\Name\AppData`. The folder will therefore be `%LOCALAPPDATA%\RProjects`. To create it, run `md %LOCALAPPDATA%\RProjects` in a command prompt. Pin this folder to the quick access of the file explorer (figure 1.1): paste `%LOCALAPPDATA%\RProjects` in the address bar of the file explorer, validate, then right click on "Quick Access" and pin the folder.

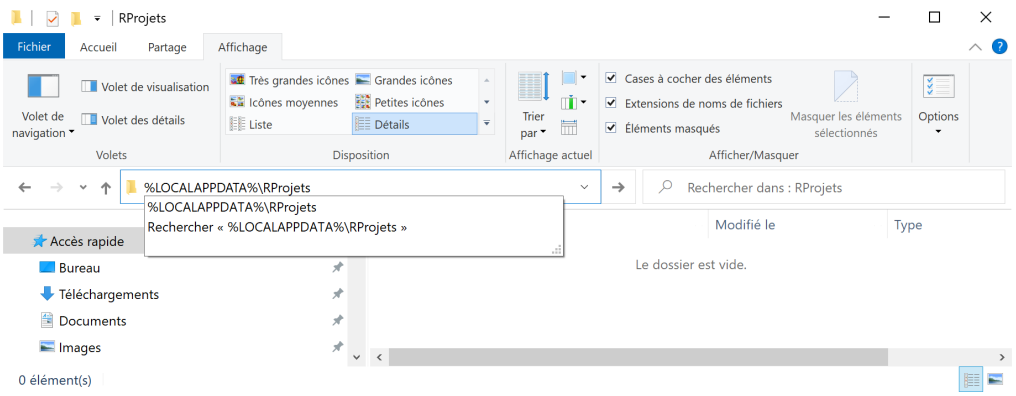


Figure 1.1: Folder for projects under source control, on Windows.

1.2.5 Character font

The Fira Code[6] font provides ligatures: the "<-" characters used for assignment in R, for example, are displayed as an arrow. To use it in the RStudio editor, simply install it according to the instructions for your operating system and declare it in the global options ("Tools > Global Options…" menu): select *Appearance* and the option *Editor Font*: Fira Code.

1.3 Packages

1.3.1 Installation from CRAN

The classic installation of packages uses CRAN. There is an "Install" button in the *Packages* window of RStudio.

[6] https://github.com/tonsky/FiraCode

Packages are uploaded to CRAN by their authors as source code, compressed in a `.tar.gz` file. They are available for download as soon as they are validated. They must then be put in binary format for Windows (in a '.zip' file), which takes some time.

When asked to install a package on Windows, CRAN proposes the source version rather than the binary version if it is more recent (figure 1.2).

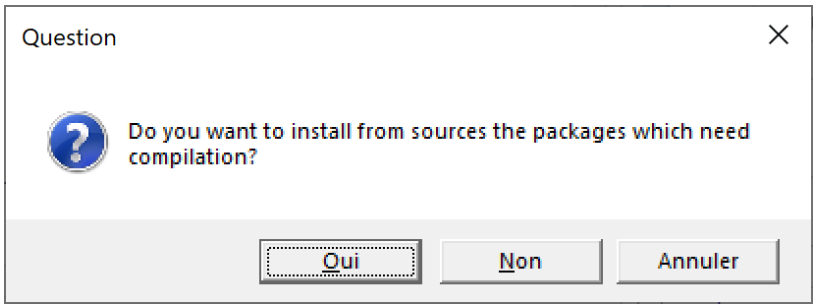


Figure 1.2: Choice of the version of the packages to install.

The list of packages concerned is displayed in the console, for example:

```
  There are binary versions available but the source
  versions are later:
             binary   source needs_compilation
boot          1.3-24   1.3-25             FALSE
class         7.3-16   7.3-17              TRUE
```

Some packages require compilation (column `needs_compilation`), usually because they contain C++ code. They can only be installed by *Rtools*.

The installation of packages in source version is much longer than in binary version. Unless a specific version of a package is needed, it is better to refuse the installation of source versions.

Packages can be updated a little later, after they have been compiled by CRAN.

The "Update" button in the RStudio *Packages* window allows you to update all installed packages.

1.3.2 Installation from GitHub

Some packages are not available on CRAN but only on GitHub because they are still under development or because they are not intended to be widely used by the R user community. It can also be useful to install a development version of a package published on CRAN for a specific use like testing new features.

The installation is handled by the **remotes** package. The `build_vignettes` argument is needed to create the vignettes of the package.

```
remotes::install_github("EricMarcon/memoiR", build_vignettes = TRUE)
```

The package name is entered as "GitHubID/PackageName". The installation is done from the source code and therefore requires the Rtools if a build is needed. `install_github()` checks that the version on GitHub is more recent than the version installed on the workstation and does nothing if they are identical.

1.3.3 Installation from Bioconductor

Bioconductor is a complementary platform to CRAN that hosts packages specialized in genomics. Installing packages from Bioconductor requires the **BiocManager** package for its `install()` function. The first argument of the function is a vector of characters containing the names of the packages to be installed, for example:

```
BiocManager::install(c("GenomicFeatures", "AnnotationDbi"))
```

The `install()` function called without arguments updates the packages.

1.3.4 Selected solution

At each minor update of R, all packages must be reinstalled. The most efficient way to do this is to create a `Packages.R` script to place in `~\R`. It contains a function that checks if each package is already installed so that it is not redone unnecessarily.

```
# Install R packages ####

# Install packages if necessary ####
InstallPackages <- function(Packages) {
  sapply(Packages, function(Package)
    if (!Package %in% installed.packages()[, 1])
      {install.packages(Package)})
}


# Development tools ####
InstallPackages(c(
  # Development tools. Import remotes, etc.
  "devtools",
  # Run Check by RStudio
  "rcmdcheck",
  # Formatting R code (used by knitr)
  "formatR",
  # Documentation of packages in /docs on GitHub
  "pkgdown",
  # Bibliography with roxygen
  "Rdpack",
  # Performance measurement
  "rbenchmark",
  # Automatic package documentation
  "roxygen2",
  # Package testing
  "testthat"
  ))

# Markdown ####
InstallPackages(c(
```

```
  # Knit
  "knitr",
  # Complex markdown documents
  "bookdown",
  # Websites
  "blogdown",
  # Document templates
  "memoiR"
  ))

# Tidyverse ####
InstallPackages("tidyverse")
```

The last part of the script is to be completed with the packages used regularly.

This script is to be executed each time R is updated, after having activated the right to write in the system library if needed (see section 1.1.4).

1.4 git and GitHub

1.4.1 git

git is the source control software used here. Its use is detailed in the chapter 3.

For Windows and Mac, the installation is done from the git website[7].

git is integrated in Linux distributions. For Ubuntu, the apt package is `git-all`.

git is installed without a graphical interface, provided by RStudio.

In RStudio, modify the global options (menu "Tools > Global Options…"). Select *Terminal* and the option *New Terminals open with*: GitBash.

Check that git is installed correctly by typing the command `git -h` in the RStudio terminal: help should be displayed.

After installing git, the RStudio terminal may not work properly and return an error message containing the following:

```
*** fatal error - cygheap base mismatch detected
This problem is probably due to using incompatible
versions of the cygwin DLL.
```

The error message is inaccurate: the library that should only exist in one copy is not `cygwin1.dll` but `msys-2.0.dll`. Look for this file in the git and Rtools installation folders. They are normally found in `usr/bin`. Replace the git one by the Rtools one: the version of the two files must be identical.

Enter your credentials by running the following commands in the terminal:

```
git config user.name
git config user.email
```

The user name is free, preferably "FirstName LastName".

[7] https://git-scm.com/

1.4.2 GitHub

GitHub is the platform accessible through a website that allows to share the content of *git* repositories. To use it, you just have to open an account with the same email address as the one registered in git.

The name of the GitHub account is noted here *GitHubID*. Each GitHub account allows to host repositories (a repository contains the files of a project) at the address https://github.com/GitHubID/RepoID[8]. Each repository can have a website at https://GitHubID.github.io/RepoID/[9]. Finally, a global web site is provided for each user at https://GitHubID.github.io/[10].

1.4.3 SSH authentication

Communication between git (installed on the local computer) and GitHub (online platform) requires authentication.

Two methods are available: HTTPS (also called SSL) and SSH. SSH is the most robust but requires the creation of a private key.

In the RStudio terminal, run:

```
ssh-keygen -t ed25519 -C "user.email"
```

The email address (which replaces "user.email") must be the one registered in the git configuration and the GitHub account. The key is saved in the `.ssh` folder of the user's home directory. It is possible to add a passphrase to the key, which will have to be typed the first time each work session is used. If the computer is properly secured (no physical access by third parties), leaving it empty allows to gain fluidity.

Warning: the private key is strictly confidential and must not be copied anywhere where it could be read by a third party (beware of automatic backups in particular). It does not need to be well backed up: in case of loss, it will be easily replaced.

Keys are normally stored in the `~/.ssh` folder, regardless of the operating system, but the location of the ~ home folder is ambiguous on Windows: for R, it is the `Documents` folder, but for other software, it is the user's root folder, parent of `Documents`.

In the RStudio terminal, check that the key is working correctly:

```
ssh -T git@github.com
```

If an error message indicates that no key is found, there are two possible solutions:

- Duplicate the `.ssh` folder (with File Explorer) in `Documents`.

[8] Example: https://github.com/EricMarcon/travailleR
[9] Example: https://EricMarcon.github.io/travailleR/
[10] Example: https://EricMarcon.github.io/

- Duplicate the `.ssh` folder in the RStudio program folder (usually `C:\Program Files\RStudio`), in `resourcesterminalbash`.

If successful, a message indicates that the authenticity of the GitHub server cannot be verified: a manual check is required for the first connection. Check with GitHub that the server's fingerprint is correct[11] and type `yes`. The server is automatically added to the list of known servers, in the `known_hosts` file.

In the `.ssh` folder, two files are created: one contains the private key, the other, with the `.pub` extension, the corresponding public key. Open the second one with a text editor and copy the public key to the clipboard. On GitHub, display the settings of your account ("Settings" menu), select "SSH and GPG Keys", click on "New SSH Key" and paste the key in the "Key" field. Give a name to the key in the "Title" field. The name can be the name of the computer on which the key was created. The key must not be copied on several computers: if necessary, create a new key on each workstation used.

If the key is compromised (lost or loaned from the computer that contains it), delete it on GitHub and create a new one.

1.4.4 Obtaining a personal access token

HTTPS authentication is the alternative to SSL authentication: choose a method and stick to it afterwards. To use HTTPS authentication, the creation of a personal access token is required.

Tokens are created on GitHub, in the settings of one's user account, in "Developer Settings > Personal Access Tokens"[12].

Generate a new token, describe it as "git-RStudio" and give it "repo" permission, i.e. modify *all* repositories (it is not possible to limit access to a particular repository). The token is a string that cannot be read later: it must be saved as a password.

1.5 LaTeX compiler

To produce documents in PDF format, a LaTeX distribution is needed. The light solution is to install the **tinytex** package which in turn installs a LaTeX distribution optimized for R Markdown.

A full distribution allows the use of LaTeX beyond RStudio but is useless if the use of LaTeX is limited to knitting R Markdown documents. MiKTeX is a very good solution for Windows and Mac.

[11] https://docs.github.com/en/github/authenticating-to-github/githubs-ssh-key-fingerprints

[12] https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line

1.5.1 tinytex

Install the package and run it:

```
install_tinytex()
```

Adding LaTeX packages not included in the minimal starting distribution is automatic but can be slow.

The distribution can be updated by the command:

```
tinytex::tlmgr_update()
```

1.5.2 MiKTeX

Installation

Download the installation file[13] and run it. There are several choices to make during the installation:

- Install the program for all users (with administrator rights).
- The default paper size: choose A4.
- The installation mode of the missing packages: choose "Always Install" so that they are downloaded automatically if needed.

For Linux, follow the instructions on the MiKTeX website.

Updates

MiKTeX is installed with the most used LaTeX packages. If a document needs a missing package, it is loaded automatically. Package updates should be done periodically with the MiKTeX console, accessible from the Start menu.

When launched without elevation of privileges, the console offers to switch to administrator mode. Click on "Switch to Administrator mode".

In the settings, check that the packages always install automatically and that the paper size is A4.

In the "Updates" menu, click on "Check for updates" then "Update now".

If the automatic installation fails, it is possible to manually install a package in the "Packages" menu.

1.6 Zotero

Zotero[14] is the most used bibliographic management software. Its extensions allow you to complete its functionalities according to your needs. Better BibTeX allows you to export and maintain a selection of bibliographic references

[13] https://miktex.org/download
[14] https://www.zotero.org/

(a Zotero collection) as a BibTeX file in an R project, where it can be used in writing documents or documenting packages.

Download the installation file and run it. Create a user account on the Zotero website. Link the local installation to the account: in the "Edit > Preferences" menu, select "Sync > Settings" and authenticate in the "Data Syncing" area. Then check the box "Sync automatically" but not "Sync full-text content" because the total size of full text synchronized in this way between the online Zotero account and the workstation is limited to 300 MB.

Download the Better BibTeX extension[15] and install it with the "Tools > Add-ons" menu: click on the settings button at the top right of the window, then "Install Add-on From File…" and select the file just downloaded.

Set up Better BibTeX from the menu "Edit > Preferences > Better BibTeX". The options to modify are the following:

- "Citation Keys > Citation Key Format": `auth:capitalize+year` so that citations have a unique identifier of the form "Name2021".
- "Citation Keys > Keep citation keys unique": "across all libraries" so that citation identifiers are not ambiguous.
- "Export > Fields > Fields to omit from export": "abstract, file" to avoid generating bibliographic files overweighted by useless information in R projects.

It is recommended to use the ZotFile[16] extension to better control the location of the full text (the PDF files linked to the bibliographic references). Install it and then set it up in "Tools > ZotFile Preferences…> General settings > Location of Files > Custom Location": choose the folder for storing full text. If the user's personal folder is backed up (for example, if it is replicated in the cloud by OneDrive on Windows), placing this storage folder there allows both to save the full texts and to access them from multiple workstations or directly online. This solution is much more efficient than Zotero's default synchronization, which is limited in volume.

1.7 Go

Go[17] is only used by the Hugo web site generator (see section 4.7).

Download the installation file and run it. At the end of the installation, run the command `go version` in a terminal to check that it works.

Upgrades are done by installing the new version over the previous one.

[15] https://retorque.re/zotero-better-bibtex/installation/
[16] http://zotfile.com/
[17] https://golang.org/

CHAPTER 2

USE R

The literature devoted to learning R is flourishing. The following books are an arbitrary but useful selection:

- R for Data Science (Wickham and Grolemund 2016) presents a complete working method, consistent with the tidyverse.
- Advanced R (Wickham 2014) is the reference for mastering the subtleties of the language and understanding how R works.
- Finally, Efficient R programming (Gillespie and Lovelace 2016) deals with code optimization.

Some advanced aspects of coding are seen here. Details on the different languages of R are useful for creating packages. The environments are presented next, for the proper understanding of the search for objects called by the code. Finally, the optimization of code performance is discussed in depth (loops, C++ code and parallelization) and illustrated by a case study.

2.1 The languages of R

R includes several programming languages. The most common is S3, but it is not the only one[1].

2.1.1 Base

The core of R is the primitive functions and basic data structures like the `sum` function and `matrix` data:

```
pryr::otype(sum)
```

```
## [1] "base"
```

```
typeof(sum)
```

```
## [1] "builtin"
```

```
pryr::otype(matrix(1))
```

```
## [1] "base"
```

```
typeof(matrix(1))
```

```
## [1] "double"
```

The **pryr** package allows to display the language in which objects are defined. The `typeof()` function displays the internal storage type of the objects:

- the `sum()` function belongs to the basic language of R and is a builtin function.
- the elements of the numerical matrix containing a single 1 are double precision reals, and the matrix itself is defined in the basic language.

Primitive functions are coded in C and are very fast. They are always available, whatever the packages loaded. Their use is therefore to be preferred.

2.1.2 S3

S3 is the most used language, often the only one known by R users.

It is an object-oriented language in which classes, i.e. the type of objects, are declarative.

```
MyFirstName <- "Eric"
class(MyFirstName) <- "FirstName"
```

[1] https://adv-r.had.co.nz/OO-essentials.html

The variable `MyFirstName` is here classed as `FirstName` by a simple declaration.

Unlike the way a classical object-oriented language works[2], S3 methods are related to functions, not objects.

```
# Default display
MyFirstName
```

```
## [1] "Eric"
## attr(,"class")
## [1] "FirstName"
```

```
print.Firstname <- function(x) cat("The first name is", x)
# Modified display
MyFirstName
```

```
## [1] "Eric"
## attr(,"class")
## [1] "FirstName"
```

In this example, the `print()` method applied to the "Firstname" class is modified. In a classical object-oriented language, the method would be defined in the class `Firstname`. In R, methods are defined from generic methods.

`print` is a generic method ("a generic") declared in **base**.

```
pryr::otype(print)
```

```
## [1] "base"
```

Its code is just a `UseMethod("print")` declaration:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x1058302e8>
## <environment: namespace:base>
```

There are many S3 methods for `print`:

```
head(methods("print"))
```

```
## [1] "print.acf"              "print.activeConcordance"
## [3] "print.AES"              "print.all_vars"
## [5] "print.anova"            "print.any_vars"
```

Each applies to a class. `print.default` is used as a last resort and relies on the type of the object, not its S3 class.

[2] https://www.troispointzero.fr/le-blog/introduction-a-la-programmation-orientee-objet-poo/

```
typeof(MyFirstName)
```

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

```
pryr::otype(MyFirstName)
```

```
## [1] "S3"
```

An object can belong to several classes, which allows a form of inheritance of methods. In a classical object oriented language, inheritance allows to define more precise classes ("FrenchFirstName") which inherit from more general classes ("FirstName") and thus benefit from their methods without having to redefine them. In R, inheritance is simply declaring a vector of increasingly broad classes for an object:

```
# Definition of classes by a vector
class(MyFirstName) <- c("FrenchFirstName", "FirstName")
# Alternative code, with inherits()
inherits(MyFirstName, what = "FrenchFirstName")
```

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

```
inherits(MyFirstName, what = "FirstName")
```

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

The generic looks for a method for each class, in the order of their declaration.

```
print.FrenchFirstName <- function(x) cat("French first name:",
    x)
MyFirstName
```

```
## French first name: Eric
```

In summary, S3 is the common language of R. Almost all packages are written in S3. Generics are everywhere but go unnoticed, for example in packages:

```
library("entropart")
.S3methods(class = "SpeciesDistribution")
```

```
## [1] autoplot plot
## see '?methods' for accessing help and source code
```

The `.S3methods()` function displays all available methods for a class, as opposed to `methods()` which displays all classes for which the method passed as an argument is defined.

Many primitive functions in R are generic methods. To find out about them, use the `help(InternalMethods)` helper.

2.1.3 S4

S4 is an evolution of S3 that structures classes to get closer to a classical object oriented language:

- Classes must be explicitly defined, not simply declared.
- 1ttributes (i.e. variables describing objects), called *slots*, are explicitly declared.
- The constructor, i.e. the method that creates a new instance of a class (i.e. a variable containing an object of the class), is explicit.

Using the previous example, the S4 syntax is as follows:

```
# Definition of the class Person, with its slots
setClass("Person",
         slots = list(LastName = "character", FirstName = "character"))
#  Construction of an instance
Me <- new("Person", LastName = "Marcon", FirstName = "Eric")
# Langage
pryr::otype(Me)
```

```
## [1] "S4"
```

Methods always belong to functions. They are declared by the `setMethod()` function:

```
setMethod("print", signature = "Person", function(x, ...) {
    cat("The person is:", x@FirstName, x@LastName)
})
print(Me)
```

```
## The person is: Eric Marcon
```

The attributes are called by the syntax `variable@slot`.

In summary, S4 is more rigorous than S3. Some packages on CRAN : **Matrix**, **sp**, **odbc**… and many on Bioconductor are written in S4 but the language is now clearly abandoned in favor of S3, notably because of the success of the **tidyverse**.

2.1.4 RC

RC was introduced in R 2.12 (2010) with the **methods** package.

Methods belong to classes, as in C++: they are declared in the class and called from the objects.

```
library("methods")
# Declaration of the class
PersonRC <- setRefClass("PersonRC",
    fields = list(LastName = "character", FirstName = "character"),
    methods = list(print = function() cat(FirstName, LastName)))
# Constructeur
```

```
MeRC <- new("PersonRC", LastName = "Marcon", FirstName ="Eric")
# Language
pryr::otype(MeRC)
```

```
## [1] "RC"
```

```
# Call the print method
MeRC$print()
```

```
## Eric Marcon
```

RC is a confidential language, although it is the first "true" object-oriented language of R.

2.1.5 S6

S6[3] enhances RC but is not included in R: it requires installing its package.

Attributes and methods can be public or private. An `initialize()` method is used as a constructor.

```
library("R6")
PersonR6 <- R6Class("PersonR6", public = list(LastName = "character",
    FirstName = "character", initialize = function(LastName = NA,
        FirstName = NA) {
        self$LastName <- LastName
        self$FirstName <- FirstName
    }, print = function() cat(self$FirstName, self$LastName)))
MeR6 <- PersonR6$new(LastName = "Marcon", FirstName = "Eric")
MeR6$print()
```

```
## Eric Marcon
```

S6 allows to program rigorously in object but is very little used. The performances of S6 are much better than those of RC but are inferior to those of S3[4].

The non-inclusion of R6 to R is shown by **pryr**:

```
pryr::otype(MeR6)
```

```
## [1] "S3"
```

2.1.6 Tidyverse

The tidyverse is a set of coherent packages that have evolved the way R is programmed. The set of essential packages can be loaded by the **tidyverse** package which has no other use:

[3] https://r6.r-lib.org/
[4] https://r6.r-lib.org/articles/Performance.html

```
library("tidyverse")
```

This is not a new language per se but rather an extension of S3, with deep technical modifications, notably the unconventional evaluation of expressions[5], which it is not essential to master in detail.

Its principles are written in a manifesto[6]. Its most visible contribution for the user is the sequence of commands in a flow (code pipeline).

In standard programming, the sequence of functions is written by successive nesting, which makes it difficult to read, especially when arguments are needed:

```
# Base-2 logarithm of the mean of 100 random numbers in a
# uniform distribution
log(mean(runif(100)), base = 2)
```

```
## [1] -1.127903
```

In the tidyverse, the functions are chained together, which often better matches the programmer's thinking about data processing:

```
# 100 random numbers in a uniform distribution
runif(100) %>%
  # Mean
  mean %>%
  # Base-2 logarithm
  log(base=2)
```

```
## [1] -0.9772102
```

The pipe %>% is an operator that calls the next function by passing it as first argument the result of the previous function. Additional arguments are passed normally: for the readability of the code, it is essential to name them. Most of the R functions can be used without difficulty in the tidyverse, even though they were not designed for this purpose: it is sufficient that their first argument is the data to be processed.

The pipeline allows only one value to be passed to the next function, which prohibits multidimensional functions, such as `f(x,y)`. The preferred data structure is the *tibble*, which is an improved dataframe: its `print()` method is more readable, and it corrects some unintuitive dataframe behavior, such as the automatic conversion of single-column dataframes to vectors. The columns of the dataframe or tibble allow to pass as much data as needed.

Finally, data visualization is supported by **ggplot2** which relies on a theoretically sound graph grammar (Wickham 2010). Schematically, a graph is constructed according to the following model:

[5] https://dplyr.tidyverse.org/articles/programming.html
[6] https://cran.r-project.org/web/packages/tidyverse/vignettes/manifesto.html

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

- The data is necessarily a dataframe.
- The geometry is the type of graph chosen (points, lines, histograms or other).
- The aesthetics (function `aes()`) designates what is represented: it is the correspondence between the columns of the dataframe and the elements necessary for the geometry.
- Statistics is the treatment applied to the data before passing it to the geometry (often "identity", i.e. no transformation but "boxplot" for a boxplot). The data can be transformed by a scale function, such as `scale_y_log10()`.
- The position is the location of the objects on the graph (often "identity"; "stack" for a stacked histogram, "jitter" to move the overlapping points slightly in a `geom_point`).
- The coordinates define the display of the graph (`coord_fixed()` to avoid distorting a map for example).
- Finally, facets offer the possibility to display several aspects of the same data by producing one graph per modality of a variable.

The set formed by the pipeline and **ggplot2** allows complex processing in a readable code. Figure 2.1 shows the result of the following code:

```
# Diamonds data provided by ggplot2
diamonds %>%
  # Keep only diamonds larger than half a carat
  filter(carat > 0.5) %>%
  # Graph: price vs. weight
  ggplot(aes(x = carat, y = price)) +
    # Scatter plot
    geom_point() +
    # Logarithmic scale
    scale_x_log10() +
    scale_y_log10() +
    # Linear regression
    geom_smooth(method = "lm")
```

In this figure, two geometries (scatterplot and linear regression) share the same aesthetics (price vs. weight in carats) which is therefore declared upstream, in the `ggplot()` function.

The tidyverse is documented in detail in Wickham and Grolemund (2016) and **ggplot2** in Wickham (2017).

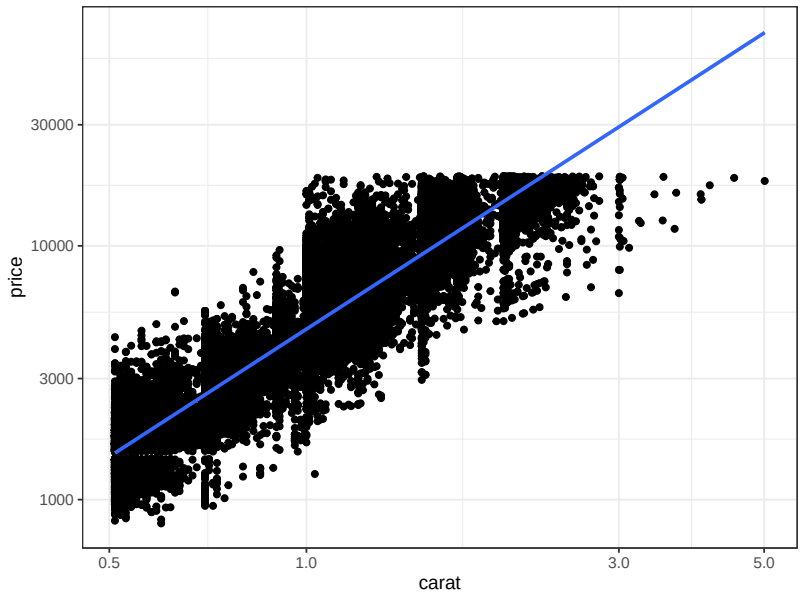


Figure 2.1: Price of diamonds according to their weight. Demonstration of the **ggplot2** code combined with tidyverse data processing.

2.2 Environments

R's objects, data and functions, are named. Since R is modular, with the ability to add any number of packages to it, it is very likely that name conflicts will arise. To deal with them, R has a rigorous system of name precedence: code runs in a defined environment, inheriting from parent environments.

2.2.1 Organization

R starts in an empty environment. Each loaded package creates a child environment to form a stack of environments, of which each new element is called a "child" of the previous one, which is its "parent".

The console is in the global environment, the child of the last loaded package.

```
search()
```

```
##  [1] ".GlobalEnv"        "package:R6"
##  [3] "package:entropart" "package:lubridate"
##  [5] "package:forcats"   "package:stringr"
##  [7] "package:dplyr"     "package:purrr"
##  [9] "package:readr"     "package:tidyr"
## [11] "package:tibble"    "package:ggplot2"
## [13] "package:tidyverse" "package:stats"
## [15] "package:graphics"  "package:grDevices"
## [17] "package:utils"     "package:datasets"
## [19] "package:methods"   "Autoloads"
## [21] "package:base"
```

The code of a function called from the console runs in a child environment of the global environment:

```r
# Current environment
environment()
```

```
## <environment: R_GlobalEnv>
```

```r
# The function f displays its environment
f <- function() environment()
# Display the environment of the function
f()
```

```
## <environment: 0x119e77c38>
```

```r
# Parent environment of the function's environment
parent.env(f())
```

```
## <environment: R_GlobalEnv>
```

2.2.2 Search

The search for ab object starts in the local environment. If it is not found, it is searched in the parent environment, then in the parent of the parent, until the environments are exhausted, which generates an error indicating that the object was not found.

Example:

```r
# Variable q defined in the global environment
q <- "GlobalEnv"
# Function defining q in its environment
qLocalFunction <- function() {
    q <- "Function"
    return(q)
}
# The local variable is returned
qLocalFunction()
```

```
## [1] "Function"
```

```r
# Poorly written function using a variable it does not
# define
qGlobalEnv <- function() {
    return(q)
}
# The global environment variable is returned
qGlobalEnv()
```

```
## [1] "GlobalEnv"
```

```r
# Delete this variable
rm(q)
# The function base::q is returned
qGlobalEnv()
```

```
## function (save = "default", status = 0, runLast = TRUE)
## .Internal(quit(save, status, runLast))
## <bytecode: 0x11b1b8e80>
## <environment: namespace:base>
```

The variable `q` is defined in the global environment. The function `qLocalFunction` defines its own variable `q`. Calling the function returns the its local value because it is in the function's environment.

The `qGlobalEnv` function returns the `q` variable that it does not define locally. So it looks for it in its parent environment and finds the variable defined in the global environment. By removing the variable from the global environment with `rm(q)`, the `qGlobalEnv()` function scans the stack of environments until it finds an object named `q` in the **base** package, which is the function to exit R. It could have found another object if a package containing a `q` object had been loaded.

To avoid this erratic behavior, a function should *never* call an object not defined in its own environment.

2.2.3 Package namespaces

It is time to define precisely what packages make visible. Packages contain objects (functions and data) which they *export* or not. They are usually called by the `library()` function, which does two things:

- It *loads* the package into memory, allowing access to all its objects with the syntax `package::object` for exported objects and `package:::object` for non-exported ones.
- It then *attaches* the package, which places its environment on top of the stack.

It is possible to detach a package with the `unloadNamespace()` function to remove it from the environment stack. Example:

```
# entropart loaded and attached
library("entropart")
# is it attached?
isNamespaceLoaded("entropart")
```

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

```
# stack of environments
search()
```

```
##  [1] ".GlobalEnv"        "package:R6"
##  [3] "package:entropart" "package:lubridate"
##  [5] "package:forcats"   "package:stringr"
##  [7] "package:dplyr"     "package:purrr"
##  [9] "package:readr"     "package:tidyr"
## [11] "package:tibble"    "package:ggplot2"
## [13] "package:tidyverse" "package:stats"
## [15] "package:graphics"  "package:grDevices"
## [17] "package:utils"     "package:datasets"
## [19] "package:methods"   "Autoloads"
## [21] "package:base"
```

```r
# Diversity(), a function exported by entropart is found
Diversity(1, CheckArguments = FALSE)
```

```
## None
##    1
```

```r
# Detach and unload entropart
unloadNamespace("entropart")
# Is it attached?
isNamespaceLoaded("entropart")
```

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

```r
# Stack of environments, without entropart
search()
```

```
##  [1] ".GlobalEnv"        "package:R6"
##  [3] "package:lubridate" "package:forcats"
##  [5] "package:stringr"   "package:dplyr"
##  [7] "package:purrr"     "package:readr"
##  [9] "package:tidyr"     "package:tibble"
## [11] "package:ggplot2"   "package:tidyverse"
## [13] "package:stats"     "package:graphics"
## [15] "package:grDevices" "package:utils"
## [17] "package:datasets"  "package:methods"
## [19] "Autoloads"         "package:base"
```

```r
# Diversity() cannot be found
tryCatch(Diversity(1), error = function(e) print(e))
```

```
## <simpleError in Diversity(1): could not find function "Diversity">
```

```r
# but can be called with its full name
entropart::Diversity(1, CheckArguments = FALSE)
```

```
## None
##    1
```

Calling `entropart::Diversity()` loads the package (i.e., implicitly executes `loadNamespace("entropart")`) but does not attach it.

In practice, one should limit the number of attached packages to limit the risk of calling an unwanted function, homonymous to the desired function. In critical cases, the full name of the function should be used: `package::function()`.

A common issue occurs with the `filter()` function of **dplyr**, which is the namesake of the **stats** function. The **stats** package is usually loaded before **dplyr**, a package in the tidyverse. Thus, `stats::filter()` must be called explicitly.

However, the **dplyr** or **tidyverse** package (which attaches all the tidyverse packages) can be loaded systematically by creating a `.RProfile` at the root of the project containing the command:

```
library("tidyverse")
```

In this case, **dplyr** is loaded *before* **stats** so its function is inaccessible.

2.3 Measuring execution time

The execution time of long code can be measured very simply by the `system.time` command. For very short execution times, it is necessary to repeat the measurement: this is the purpose of the **microbenchmark** package.

2.3.1 system.time

The function returns the execution time of the code.

```
# Mean absolute deviation of 1000 values in a uniform
# distribution, repeated 100 times
system.time(for (i in 1:100) mad(runif(1000)))
```

```
##    user  system elapsed
##   0.008   0.001   0.009
```

2.3.2 microbenchmark

The **microbenchmark** package is the most advanced.

The goal is to compare the speed of computing the square of a vector (or a number) by multiplying it by itself ($x \times x$) or by raising it to the power of 2 (x^2).

```
# Functions to test
f1 <- function(x) x * x
f2 <- function(x) x^2
f3 <- function(x) x^2.1
f4 <- function(x) x^3
# Initialization
X <- rnorm(10000)
# Test
library("microbenchmark")
(mb <- microbenchmark(f1(X), f2(X), f3(X), f4(X)))
```

```
## Unit: microseconds
##   expr     min       lq      mean   median       uq     max
##  f1(X)   1.804   5.8630  19.46721   7.2980  26.937 359.242
##  f2(X)   3.895   7.5440  21.25973  11.5825  28.987 359.857
##  f3(X)  93.644  97.9080 112.19486 101.4340 120.786 458.298
##  f4(X) 124.394 129.4165 144.70499 131.5485 151.577 642.306
##  neval
##    100
##    100
##    100
##    100
```

The returned table contains the minimum, median, mean, max and first and third quartile times, as well as the number of repetitions. The median value is

to be compared. The number of repetitions is by default 100, to be modulated (argument `times`) according to the complexity of the calculation.

The test result, a `microbenchmark` object, is a raw table of execution times. The statistical analysis is done by the `print` and `summary` methods. To choose the columns to display, use the following syntax:

```
summary(mb)[, c("expr", "median")]
```

```
##    expr   median
## 1 f1(X)   7.2980
## 2 f2(X)  11.5825
## 3 f3(X) 101.4340
## 4 f4(X) 131.5485
```

To make calculations on these results, we must store them in a variable. To prevent the results from being displayed in the console, the simplest solution is to use the `capture.output` function by assigning its result to a variable.

```
dummy <- capture.output(mbs <- summary(mb))
```

The previous test is displayed again.

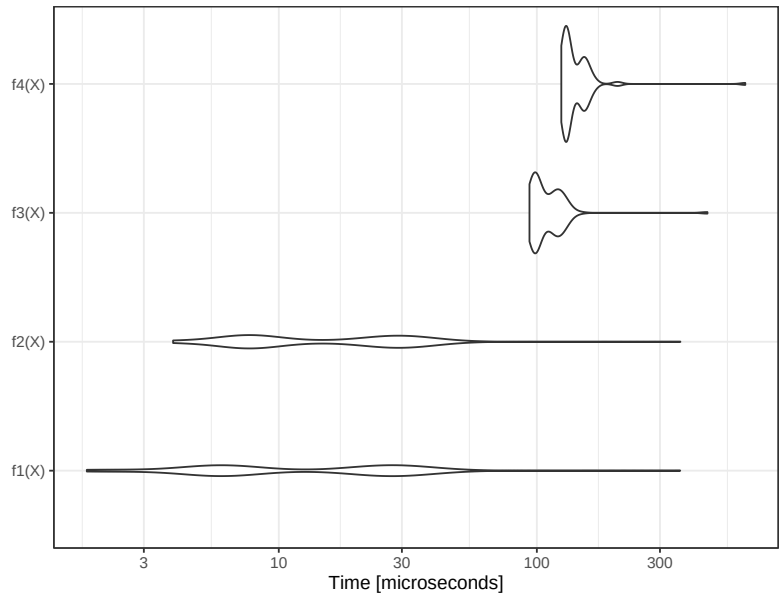
```
summary(mb)[, c("expr", "median")]
```

```
##    expr   median
## 1 f1(X)   7.2980
## 2 f2(X)  11.5825
## 3 f3(X) 101.4340
## 4 f4(X) 131.5485
```

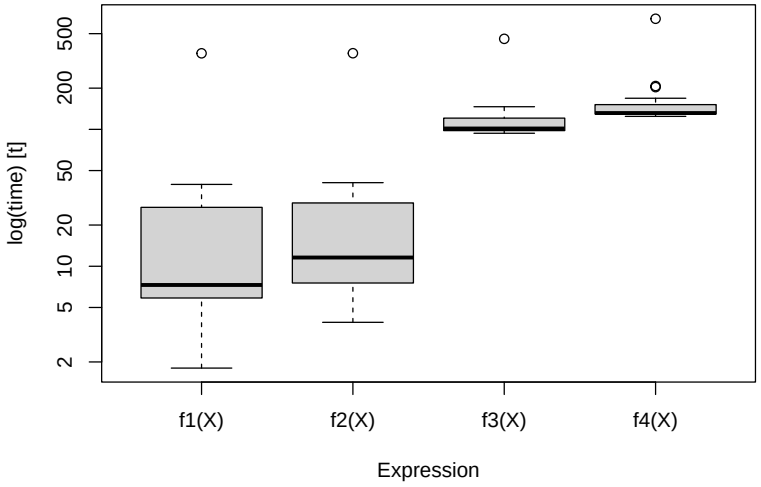
The computation time is about the same between $x \times x$ and x^2. The power calculation is much longer, especially if the power is not integer, because it requires a logarithm calculation. The computation of the power 2 is therefore optimized by R to avoid the use of log.

Two graphical representations are available: the violins represent the probability density of the execution time; the boxplots are classical.

```
library("ggplot2")
autoplot(mb)
```




```
boxplot(mb)
```



2.3.3 Profiling

profvis is RStudio's profiling tool.

It tracks the execution time of each line of code and the memory used. The goal is to detect slow code portions that need to be improved.

```
library(profvis)
p <- profvis({
    # Cosine calculations
    cos(runif(10^7))
    # 1/2 second pause
```

```
    pause(1/2)
})
htmlwidgets::saveWidget(p, "docs/profile.html")
```

The result is an HTML file containing the profiling report[7]. It can be observed that the time to draw the random numbers is similar to that of the cosine calculation.

Read the complete documentation[8] on the RStudio website.

2.4 Loops

The most frequent case of long code to execute is loops: the same code is repeated a large number of times.

2.4.1 Vector functions

Most of R's functions are vector functions: loops are processed internally, extremely fast. Therefore, you should think in terms of vectors rather than scalars.

```
# Draw two vectors of three random numbers between 0 and 1
x1 <- runif(3)
x2 <- runif(3)
# Square root of the three numbers in x1
sqrt(x1)
```

```
## [1] 0.9427738 0.8665204 0.4586981
```

```
# Respective sums of the three numbers of x1 and x2
x1 + x2
```

```
## [1] 1.6262539 1.6881583 0.9063973
```

We also have to write vector functions on their first argument. The function `lnq` of the package **entropart** returns the deformed logarithm of order q of a number x.

```
# Code of the function
entropart::lnq
```

```
## function (x, q)
## {
##     if (q == 1) {
##         return(log(x))
##     }
##     else {
##         Log <- (x^(1 - q) - 1)/(1 - q)
##         Log[x < 0] <- NA
##         return(Log)
```

[7] https://EricMarcon.github.io/WorkingWithR/profile.html
[8] https://rstudio.github.io/profvis/

```
##     }
## }
## <bytecode: 0x11bf9c940>
## <environment: namespace:entropart>
```

For a function to be vector, each line of its code must allow the first argument to be treated as a vector. Here: `log(x)` and `x^` are a vector function and operator and the condition `[x < 0]` also returns a vector.

2.4.2 lapply

Code that cannot be written as a vector function requires loops.

`lapply()` applies a function to each element of a list. There are several versions of this function:

- `lapply()` returns a list (and saves the time of rearranging them in an array).
- `sapply()` returns a dataframe by collapsing the lists (this is done by the `simplify2array()` function).
- `vapply()` is almost identical but requires that the data type of the result be provided.

```
# Draw 1000 values in a uniform distribution
x1 <- runif(1000)
# The square root can be calculated for the vector or each
# value
identical(sqrt(x1), sapply(x1, FUN = sqrt))
```

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

```
mb <- microbenchmark(sqrt(x1), lapply(x1, FUN = sqrt), sapply(x1,
    FUN = sqrt), vapply(x1, FUN = sqrt, FUN.VALUE = 0))
summary(mb)[, c("expr", "median")]
```

```
##                                     expr   median
## 1                               sqrt(x1)   1.5580
## 2               lapply(x1, FUN = sqrt) 145.8165
## 3               sapply(x1, FUN = sqrt) 174.7625
## 4 vapply(x1, FUN = sqrt, FUN.VALUE = 0) 142.8645
```

`lapply()` is much slower than a vector function. `sapply()` requires more time for `simplify2array()`, which must detect how to gather the results. Finally, `vapply()` saves the time of determining the data type of the result and allows for faster computation with little effort.

2.4.3 For loops

Loops are handled by the `for` function. They have the reputation of being slow in R because the code inside the loop must be interpreted at each execution. This is no longer the case since version 3.5 of R: loops are compiled systematically

before execution. The behavior of the just-in-time compiler is defined by the `enableJIT` function. The default level is 3: all functions are compiled, and loops in the code are compiled too.

To evaluate the performance gain, the following code removes all automatic compilation, and compares the same loop compiled or not.

```
library("compiler")
# No automatic compilation
enableJIT(level = 0)
```

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

```
# Loop to calculate the square root of a vector
Loop <- function(x) {
    # Initialization of the result vector, essential
    Root <- vector("numeric", length = length(x))
    # Loop
    for (i in 1:length(x)) Root[i] <- sqrt(x[i])
    return(Root)
}
# Compiled version
Loop2 <- cmpfun(Loop)
# Comparison
mb <- microbenchmark(Loop(x1), Loop2(x1))
(mbs <- summary(mb)[, c("expr", "median")])
```

```
##        expr   median
## 1  Loop(x1) 363.1780
## 2 Loop2(x1)  43.3575
```

```
# Automatic compilation by default since version 3.5
enableJIT(level = 3)
```

```
## [1] 0
```

The gain is considerable: from 1 to 8.

For loops are now much faster than `vapply`.

```
# Test
mb <- microbenchmark(vapply(x1, FUN = sqrt, 0), Loop(x1))
summary(mb)[, c("expr", "median")]
```

```
##                         expr   median
## 1 vapply(x1, FUN = sqrt, 0) 132.2045
## 2                  Loop(x1)  42.1890
```

Be careful, the performance test can be misleading:

```
# Preparing the result vector
Root <- vector("numeric", length = length(x1))
# Test
mb <- microbenchmark(vapply(x1, FUN = sqrt, 0),
                     for(i in 1:length(x1))
                       Root[i] <- sqrt(x1[i]))
summary(mb)[, c("expr", "median")]
```

```
##                                             expr   median
## 1                       vapply(x1, FUN = sqrt, 0)  143.500
## 2 for (i in 1:length(x1)) Root[i] <- sqrt(x1[i]) 1363.763
```

In this code, the for loop is not compiled so it is much slower than in its normal use (in a function or at the top level of the code).

The long loops allow tracking of their progress by a text bar, which is another advantage. The following function executes pauses of one tenth of a second for the time passed in parameter (in seconds).

```
MonitoredLoop <- function(duration = 1) {
    # Progress bar
    pgb <- txtProgressBar(min = 0, max = duration * 10)
    # Boucle
    for (i in 1:(duration * 10)) {
        # Pause for a tenth of a second
        Sys.sleep(1/10)
        # Track progress
        setTxtProgressBar(pgb, i)
    }
}
MonitoredLoop()
```

```
## ============================================================
```

2.4.4 replicate

`replicate()` repeats a statement.

```
replicate(3, runif(1))
```

```
## [1] 0.9453453 0.5262818 0.7233425
```

This code is equivalent to `runif(3)`, with performance similar to `vapply`: 50 to 100 times slower than a vector function.

```
mb <- microbenchmark(replicate(1000, runif(1)), runif(1000))
summary(mb)[, c("expr", "median")]
```

```
##                         expr   median
## 1 replicate(1000, runif(1)) 739.6810
## 2               runif(1000)   6.2525
```

2.4.5 Vectorize

`Vectorize()` makes a function that is not vectorized, by loops. Write the loops instead.

2.4.6 Marginal statistics

`apply` applies a function to the rows or columns of a two dimensional object.

`colSums` and similar functions (`rowSums`, `colMeans`, `rowMeans`) are optimized.

```
# Sum of the numeric columns of the diamonds dataset of ggplot2
# Loop identical to the action of apply(, 2, )
SumLoop <- function(Table) {
  Sum <- vector("numeric", length = ncol(Table))
  for (i in 1:ncol(Table)) Sum[i] <- sum(Table[, i])
  return(Sum)
}
mb <- microbenchmark(SumLoop(diamonds[-(2:4)]),
                     apply(diamonds[-(2:4)], 2, sum),
                     colSums(diamonds[-(2:4)]))
summary(mb)[, c("expr", "median")]
```

```
##                              expr   median
## 1       SumLoop(diamonds[-(2:4)]) 1.842581
## 2 apply(diamonds[-(2:4)], 2, sum) 4.281814
## 3       colSums(diamonds[-(2:4)]) 1.364357
```

`apply` clarifies the code but is slower than the loop, which is only slightly slower than `colSums`.

2.5 C++ code

Integrating C++ code into R is greatly simplified by the **Rcpp** package but is still difficult to debug and therefore should be reserved for very simple code (to avoid errors) repeated a large number of times (to be worth the effort). The preparation and verification of the data must be done in R, as well as the processing and presentation of the results.

The common practise is to include C++ code in a package, but running it outside a package is possible:

- C++ code can be included in a C++ document (file with extension `.cpp`): it is compiled by the `sourceCpp()` command, which creates the R functions to call the C++ code.
- In an RMarkdown document, Rcpp code snippets can be created to insert the C++ code: they are compiled and interfaced to R at the time of knitting.

The following example shows how to create a C++ function to calculate the double of a numerical vector.

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

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

An R function with the same name as the C++ function is now available.

```
timesTwo(1:5)
```

```
## [1]  2  4  6  8 10
```

The performance is two orders of magnitude faster than the R code (see the case study, section 2.7).

2.6 Parallelizing R

When long computations can be split into independent tasks, the simultaneous (*parallel*) execution of these tasks reduces the total computation time to that of the longest task, to which is added the cost of setting up the parallelization (creation of the tasks, recovery of the results…).

Read Josh Errickson's excellent introduction[9] which details the issues and constraints of parallelization.

Two mechanisms are available for parallel code execution:

- *fork*: the running process is duplicated on multiple cores of the computing computer's processor. This is the simplest method but it does not work on Windows (it is a limitation of the operating system).
- *Socket*: a cluster is constituted, either physically (a set of computers running R is necessary) or logically (an instance of R on each core of the computer used). The members of the cluster communicate through the network (the internal network of the computer is used in a logical cluster).

Different R packages allow to implement these mechanisms.

2.6.1 mclapply (fork)

The `mclapply` function of the **parallel** package has the same syntax as `lapply` but parallelizes the execution of loops. On Windows, it has no effect since the system does not allow *fork*: it simply calls `lapply`. However, a workaround exists to emulate `mclapply` on Windows by calling `parLapply`, which uses a cluster.

```
##
## mclapply.hack.R
##
## Nathan VanHoudnos
## nathanvan AT northwestern FULL STOP edu
## July 14, 2014
##
## A script to implement a hackish version of
## parallel:mclapply() on Windows machines.
```

[9] http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html

```
## On Linux or Mac, the script has no effect
## beyond loading the parallel library.

require(parallel)
```

```
## Loading required package: parallel
```

```
## Define the hack
# mc.cores argument added: Eric Marcon
mclapply.hack <- function(..., mc.cores=detectCores()) {
  ## Create a cluster
  size.of.list <- length(list(...)[[1]])
  cl <- makeCluster( min(size.of.list, mc.cores) )

  ## Find out the names of the loaded packages
  loaded.package.names <- c(
    ## Base packages
    sessionInfo()$basePkgs,
    ## Additional packages
    names( sessionInfo()$otherPkgs ))

  tryCatch( {

    ## Copy over all of the objects within scope to
    ## all clusters.
    this.env <- environment()
    while( identical( this.env, globalenv() ) == FALSE ) {
      clusterExport(cl,
                    ls(all.names=TRUE, env=this.env),
                    envir=this.env)
      this.env <- parent.env(environment())
    }
    clusterExport(cl,
                  ls(all.names=TRUE, env=globalenv()),
                  envir=globalenv())

    ## Load the libraries on all the clusters
    ## N.B. length(cl) returns the number of clusters
    parLapply( cl, 1:length(cl), function(xx){
      lapply(loaded.package.names, function(yy) {
        require(yy , character.only=TRUE)})
    })

    ## Run the lapply in parallel
    return( parLapply( cl, ...) )
  }, finally = {
    ## Stop the cluster
    stopCluster(cl)
  })
}

## Warn the user if they are using Windows
if( Sys.info()[['sysname']] == 'Windows' ){
  message(paste(
    "\n",
    "   *** Microsoft Windows detected ***\n",
    "   \n",
    "   For technical reasons, the MS Windows version of mclapply()\n",
    "   is implemented as a serial function instead of a parallel\n",
    "   function.",
    "   \n\n",
    "   As a quick hack, we replace this serial version of mclapply()\n",
    "   with a wrapper to parLapply() for this R session. Please see\n\n",
    "     http://www.stat.cmu.edu/~nmv/2014/07/14/
```

```
    implementing-mclapply-on-windows \n\n",
    "   for details.\n\n"))
}

## If the OS is Windows, set mclapply to the
## the hackish version. Otherwise, leave the
## definition alone.
mclapply <- switch( Sys.info()[['sysname']],
                    Windows = {mclapply.hack},
                    Linux   = {mclapply},
                    Darwin  = {mclapply})

## end mclapply.hack.R
```

The following code tests the parallelization of a function that returns its argument unchanged after a quarter-second pause. This is knitted with 3 cores, all of which are used except for one so as not to saturate the system.

```
f <- function(x, time = 0.25) {
    Sys.sleep(time)
    return(x)
}
# Leave one core out for the system
nbCores <- detectCores() - 1
# Serial : theoretical time = nbCores/4 seconds
(tserie <- system.time(lapply(1:nbCores, f)))
```

```
##    user  system elapsed
##   0.003   0.000   0.628
```

```
# Parallel : theoretical time = 1/4 second
(tparallele <- system.time(mclapply(1:nbCores, f, mc.cores = nbCores)))
```

```
##    user  system elapsed
##   0.002   0.023   0.422
```

Setting up parallelization has a cost of about `0.17` seconds here. The execution time is much longer in parallel on Windows because setting up the cluster takes much more time than parallelization saves. Parallelization is interesting for longer tasks, such as a one second break.

```
# Serial
system.time(lapply(1:nbCores, f, time = 1))
```

```
##    user  system elapsed
##   0.000   0.000   2.094
```

```
# Parallel
system.time(mclapply(1:nbCores, f, time = 1, mc.cores = nbCores))
```

```
##    user  system elapsed
##   0.002   0.009   1.070
```

The additional time required for parallel execution of the new code is relatively smaller: the costs become less than the savings when the time of each task increases.

If the number of parallel tasks exceeds the number of cores used, performance collapses because the additional task must be executed after the first ones.

```
system.time(mclapply(1:nbCores, f, time = 1, mc.cores = nbCores))
```

```
##    user  system elapsed
##   0.002   0.025   1.168
```

```
system.time(mclapply(1:(nbCores + 1), f, time = 1, mc.cores = nbCores))
```

```
##    user  system elapsed
##   0.001   0.020   2.106
```

The time then remains stable until the number of cores is doubled. Figure 2.2 shows the evolution of the computation time according to the number of tasks.

```
Tasks <- 1:(2 * nbCores+1)
Time <- sapply(Tasks, function(nbTasks) {
      system.time(mclapply(1:nbTasks, f, time=1, mc.cores=nbCores))
                })
library("tidyverse")
tibble(Tasks, Time=Time["elapsed", ]) %>%
  ggplot +
  geom_line(aes(x = Tasks, y = Time)) +
  geom_vline(xintercept = nbCores, col = "red", lty = 2) +
  geom_vline(xintercept = 2 * nbCores, col = "red", lty = 2)
```

The theoretical shape of this curve is as follows:

- For a task, the time is equal to one second plus the parallelization setup time.
- The time should remain stable until the number of cores used.
- When all the cores are used (red dotted line), the time should increase by one second and then remain stable until the next limit.

In practice, the computation time is determined by other factors that are difficult to predict. The best practice is to adapt the number of tasks to the number of cores, otherwise performance will be lost.

2.6.2 parLapply (socket)

`parLapply` requires to create a cluster, export the useful variables on each node, load the necessary packages on each node, execute the code and finally stop the cluster. The code for each step can be found in the `mclapply.hack` function above.

For everyday use, `mclapply` is faster, except on Windows, and simpler (including on Windows thanks to the above workaround).

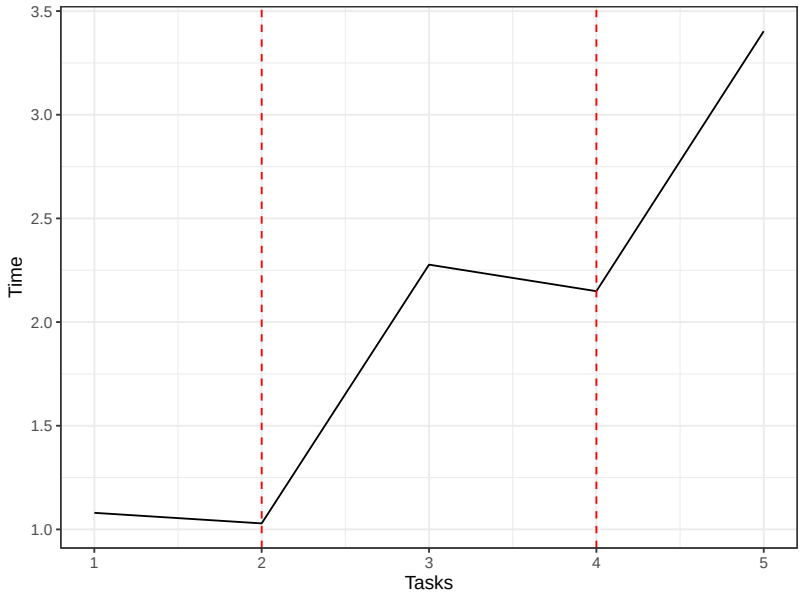


Figure 2.2: Parallel execution time of tasks requiring one second (each task is a one second pause). The number of tasks varies from 1 to twice the number of cores used (equal to 2) plus one.

2.6.3 foreach

How it works

The **foreach** package allows advanced use of parallelization. Read its vignettes.

```
# Manual
vignette("foreach", "foreach")
# Nested loops
vignette("nested", "foreach")
```

Regardless of parallelization, **foreach** redefines *for* loops.

```
for (i in 1:3) {
    f(i)
}
# becomes
library("foreach")
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
foreach(i = 1:3) %do% {
    f(i)
}
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

The `foreach` function returns a list containing the results of each loop. The elements of the list can be combined by any function, such as `c`.

```
foreach(i = 1:3, .combine = "c") %do% {
    f(i)
}
```

```
## [1] 1 2 3
```

The `foreach` function is capable of using iterators, that is, functions that pass to the loop only the data it needs without loading the rest into memory. Here, the `icount` iterator passes the values 1, 2 and 3 individually, without loading the 1:3 vector into memory.

```
library("iterators")
foreach(i = icount(3), .combine = "c") %do% {
    f(i)
}
```

```
## [1] 1 2 3
```

It is therefore very useful when each object of the loop uses a large amount of memory.

Parallelization

Replacing the %do% operator with %dopar% parallelizes loops, provided that an adapter, i.e. an intermediate package between `foreach` and a package implementing parallelization, is loaded. **doParallel** is an adapter for using the **parallel** package that comes with R.

```
library(doParallel)
registerDoParallel(cores = nbCores)
# Serial
system.time(foreach(i = icount(nbCores), .combine = "c") %do%
    {
        f(i)
    })
```

```
##    user  system elapsed
##   0.002   0.000   0.614
```

```
# Parallel
system.time(foreach(i = icount(nbCores), .combine = "c") %dopar%
    {
        f(i)
    })
```

```
##    user  system elapsed
##   0.005   0.021   0.409
```

The fixed cost of parallelization is low.

2.6.4 future

The **future** package is used to abstract the code of the parallelization implementation. It is at the centre of an ecosystem of packages that facilitate its use[10].

The parallelization strategy used is declared by the `plan()` function. The default strategy is `sequential`, i.e. single-task. The `multicore` and `multisession` strategies are based respectively on the *fork* and *socket* techniques seen above. Other strategies are available for using physical clusters (several computers prepared to run R together): the **future** documentation details how to do this.

Here we will use the `multisession` strategy, which works on the local computer, whatever its operating system.

```
library("future")
# Socket strategy on all available cores except 1
usedCores <- availableCores() - 1
plan(multisession, workers = usedCores)
```

The **future.apply** package allows all `*apply()` and `replicate()` loops to be effortlessly parallelized by prefixing their names with `future_`.

```
library("future.apply")
system.time(future_replicate(usedCores - 1, f(usedCores)))
```

```
##    user  system elapsed
##   0.023   0.001   0.344
```

foreach loops can be parallelized with the **doFuture** package by simply replacing %dopar% with %dofuture%.

```
library("doFuture")
system.time(foreach(i = icount(nbCores), .combine = "c") %dofuture%
    {
        f(i)
    })
```

```
##    user  system elapsed
##   0.037   0.001   0.458
```

The strategy is reset to `sequential` at the end.

```
plan(sequential)
```

[10] https://www.futureverse.org/

2.7 Case study

This case study tests the different techniques seen above to solve a concrete problem. The objective is to compute the average distance between two points of a random set of 1000 points in a square window of side 1.

Its expectation is computable[11]. It is equal to $\frac{2+\sqrt{2}+5\ln(1+\sqrt{2})}{15} \approx 0.5214$.

2.7.1 Creation of the data

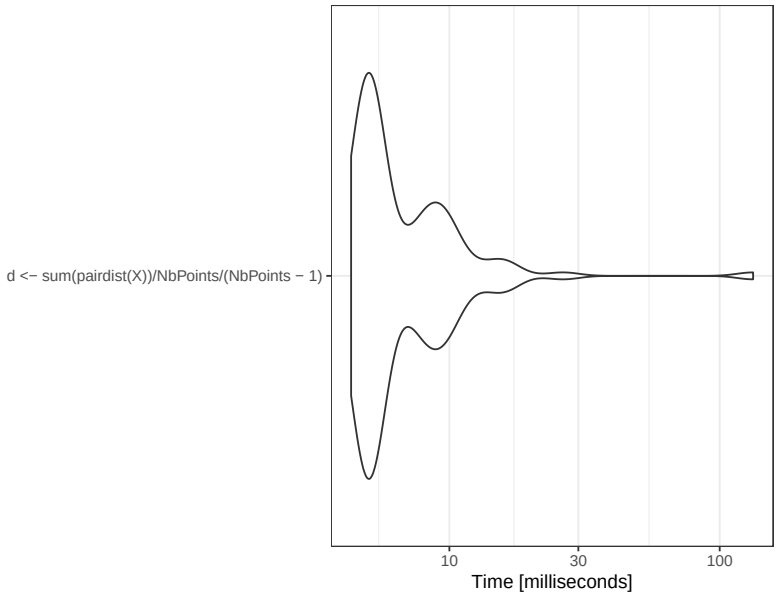
The point set is created with the **spatstat** package.

```
NbPoints <- 1000
library("spatstat")
X <- runifpoint(NbPoints)
```

2.7.2 Spatstat

The `pairdist()` function of **spatstat** returns the matrix of distances between points. The average distance is calculated by dividing the sum by the number of pairs of distinct points.

```
mb <- microbenchmark(d <- sum(pairdist(X))/NbPoints/(NbPoints -
    1))
# suppressMessages to eliminate useless messages
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5154879
```

[11] https://mindyourdecisions.com/blog/2016/07/03/distance-between-two-random-points-in-a-square-sunday-puzzle/

The function is fast because it is coded in C language in the **spatstat** package for the core of its calculations.

2.7.3 apply

The distance can be calculated by two nested `sapply()`.

```
fsapply1 <- function() {
    distances <- sapply(1:NbPoints, function(i) sapply(1:NbPoints,
        function(j) sqrt((X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2)))
    return(sum(distances)/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply1())
```

```
##    user  system elapsed
##   2.422   0.012   2.438
```

```
d
```

```
## [1] 0.5154879
```

Some time can be saved by replacing `sapply` with `vapply`: the format of the results does not have to be determined by the function. The gain is negligible on a long computation like this one but important for short computations.

```
fsapply2 <- function() {
    distances <- vapply(1:NbPoints, function(i) vapply(1:NbPoints,
        function(j) sqrt((X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2),
        0), 1:1000 + 0)
    return(sum(distances)/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply2())
```

```
##    user  system elapsed
##   2.370   0.008   2.381
```

```
d
```

```
## [1] 0.5154879
```

The output format is not always obvious to write:

- it must respect the size of the data: a vector of size 1000 for the outer loop, a scalar for the inner loop.
- it must respect the type: `0` for an integer, `0.0` for a real number. In the outer loop, adding `0.0` to the vector of integers turns it into a vector of real numbers.

A more significant improvement is to compute the square roots only at the end of the loop, to take advantage of the vectorization of the function.

```
fsapply3 <- function() {
    distances <- vapply(1:NbPoints, function(i) vapply(1:NbPoints,
        function(j) (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2,
        0), 1:1000 + 0)
    return(sum(sqrt(distances))/NbPoints/(NbPoints - 1))
}
system.time(d <- fsapply3())
```

```
##    user  system elapsed
##   2.462   0.008   2.475
```

```
d
```

```
## [1] 0.5154879
```

The computations are performed twice (distance between points i and j, but also between points j and i): a test on the indices allows to divide the time almost by 2 (not quite because the loops without computation, which return 0, take time).

```
fsapply4 <- function() {
    distances <- vapply(1:NbPoints, function(i) {
        vapply(1:NbPoints, function(j) {
            if (j > i) {
                (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2
            } else {
                0
            }
        }, 0)
    }, 1:1000 + 0)
    return(sum(sqrt(distances))/NbPoints/(NbPoints - 1) * 2)
}
system.time(d <- fsapply4())
```

```
##    user  system elapsed
##   1.399   0.005   1.407
```

```
d
```

```
## [1] 0.5154879
```

In parallel, the computation time is not improved on Windows because the individual tasks are too short. On MacOS or Linux, the computation is accelerated.

```
fsapply5 <- function() {
    distances <- mclapply(1:NbPoints, function(i) {
        vapply(1:NbPoints, function(j) {
            if (j > i) {
                (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2
            } else {
                0
            }
        }, 0)
    })
    return(sum(sqrt(simplify2array(distances)))/NbPoints/(NbPoints -
```

```
        1) * 2)
}
system.time(d <- fsapply5())
```

```
##    user  system elapsed
##   2.215   0.713   2.065
```

```
d
```

```
## [1] 0.5154879
```

2.7.4 future.apply

The `fsapply4()` function optimised above can be parallelled directly by prefixing the `vapply` function with `future_`. Only the main loop is parallelized: nesting `future_vapply()` would collapse performance.

```
library("future.apply")
# Socket strategy on all available cores except 1
plan(multisession, workers = availableCores() - 1)
future_fsapply4_ <- function() {
    distances <- future_vapply(1:NbPoints, function(i) {
        vapply(1:NbPoints, function(j) {
            if (j > i) {
                (X$x[i] - X$x[j])^2 + (X$y[i] - X$y[j])^2
            } else {
                0
            }
        }, 0)
    }, 1:1000 + 0)
    return(sum(sqrt(distances))/NbPoints/(NbPoints - 1) * 2)
}
system.time(d <- future_fsapply4_())
```

```
##    user  system elapsed
##   0.067   0.013   1.124
```

```
d
```

```
## [1] 0.5154879
```

```
plan(sequential)
```

2.7.5 for loop

A for loop is faster and consumes less memory because it does not store the distance matrix.

```
distance <- 0
ffor <- function() {
    for (i in 1:(NbPoints - 1)) {
        for (j in (i + 1):NbPoints) {
            distance <- distance + sqrt((X$x[i] - X$x[j])^2 +
                (X$y[i] - X$y[j])^2)
```

```
        }
    }
    return(distance/NbPoints/(NbPoints - 1) * 2)
}
# Calculation time, stored
(for_time <- system.time(d <- ffor()))
```

```
##    user  system elapsed
##   0.981   0.009   0.991
```

```
d
```

```
## [1] 0.5154879
```

This is the simplest and most efficient way to write this code with core R and no parallelization.

2.7.6 foreach loop

Parallelization executes for loops inside a foreach loop, which is quite efficient. However, distances are calculated twice.

```
registerDoParallel(cores = detectCores())
fforeach3 <- function(Y) {
  distances <- foreach(
    i = icount(Y$n),
    .combine = '+') %dopar% {
      distance <- 0
      for (j in 1:Y$n) {
        distance <- distance +
          sqrt((Y$x[i] - Y$x[j])^2 + (Y$y[i] - Y$y[j])^2)
      }
      distance
    }
  return(distances / Y$n / (Y$n - 1))
}
system.time(d <- fforeach3(X))
```

```
##    user  system elapsed
##   2.364   0.174   0.935
```

```
d
```

```
## [1] 0.5154879
```

It is possible to nest two foreach loops, but they are extremely slow compared with a simple loop. The test is run here with 10 times fewer points, so 100 times fewer distances to calculate.

```
NbPointsReduit <- 100
Y <- runifpoint(NbPointsReduit)
fforeach1 <- function(Y) {
    distances <- foreach(i = 1:NbPointsReduit, .combine = "cbind") %:%
        foreach(j = 1:NbPointsReduit, .combine = "c") %do% {
        if (j > i) {
```

```
            (Y$x[i] - Y$x[j])^2 + (Y$y[i] - Y$y[j])^2
        } else {
            0
        }
    }
    return(sum(sqrt(distances))/NbPointsReduit/(NbPointsReduit -
        1) * 2)
}
system.time(d <- fforeach1(Y))
```

```
##    user  system elapsed
##   0.862   0.008   0.871
```

```
d
```

```
## [1] 0.5304197
```

Nested foreach loops should be reserved for very long tasks (several seconds at least) to compensate the fixed costs of setting them up.

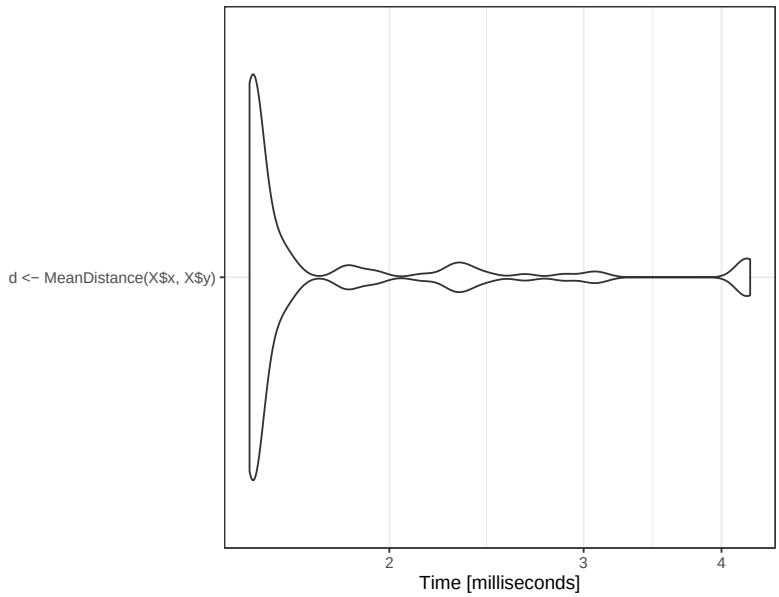
2.7.7 RCpp

The C++ function to calculate distances is the following.

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

// [[Rcpp::export]]
double MeanDistance(NumericVector x, NumericVector y) {
  double distance=0;
  double dx, dy;
  for (int i=0; i < (x.length()-1); i++) {
    for (int j=i+1; j < x.length(); j++) {
    // Calculate distance
        dx = x[i]-x[j];
        dy = y[i]-y[j];
        distance += sqrt(dx*dx + dy*dy);
    }
  }
  return distance/(double)(x.length()/2*(x.length()-1));
}
```

It is called in R very simply. The computation time is very short.

```
mb <- microbenchmark(d <- MeanDistance(X$x, X$y))
# suppressMessages to eliminate superfluous messages
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5154879
```

2.7.8 RcppParallel

RcppParallel allows to interface parallelized C++ code, at the cost of a more complex syntax than **RCpp**. Documentation is available[12].

The C++ function exported to R does not perform the computations but only organizes the parallel execution of another, non-exported, function of type `Worker`.

Two (C++) parallelization functions are available for two types of tasks:

- `parallelReduce` to accumulate a value, used here to sum distances.
- `parallelFor` to fill a result matrix.

The syntax of the `Worker` is a bit tricky but simple enough to adapt: the constructors initialize the C variables from the values passed by R and declare the parallelization.

```
// [[Rcpp::depends(RcppParallel)]]
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp;
using namespace RcppParallel;

// Working function, not exported
struct TotalDistanceWrkr : public Worker
{
  // source vectors
```

[12] http://rcppcore.github.io/RcppParallel/

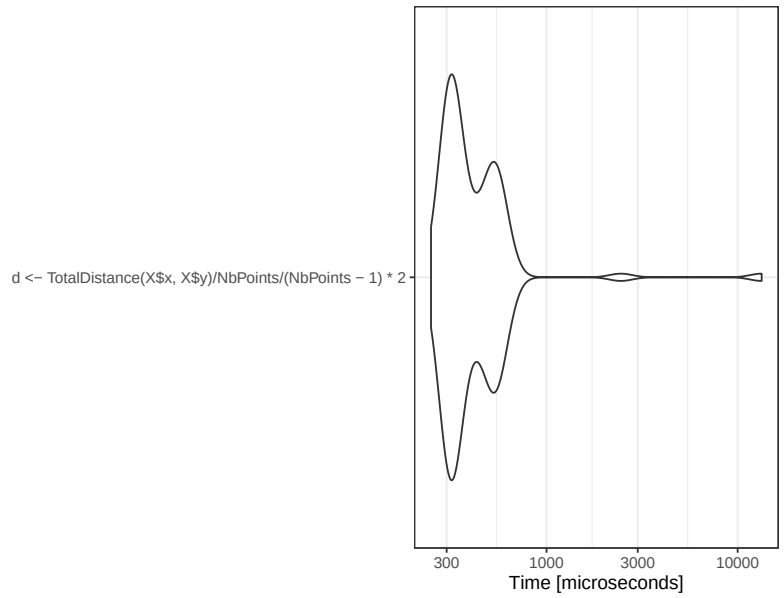
```
  const RVector<double> Rx;
  const RVector<double> Ry;

  // accumulated value
  double distance;

  // constructors
  TotalDistanceWrkr(const NumericVector x, const NumericVector y) :
    Rx(x), Ry(y), distance(0) {}
  TotalDistanceWrkr(const TotalDistanceWrkr& totalDistanceWrkr, Split) :
    Rx(totalDistanceWrkr.Rx), Ry(totalDistanceWrkr.Ry),  distance(0) {}

  // count neighbors
  void operator()(std::size_t begin, std::size_t end) {
    double dx, dy;
    unsigned int Npoints = Rx.length();

    for (unsigned int i = begin; i < end; i++) {
      for (unsigned int j=i+1; j < Npoints; j++) {
          // Calculate squared distance
          dx = Rx[i]-Rx[j];
          dy = Ry[i]-Ry[j];
          distance += sqrt(dx*dx + dy*dy);
      }
    }
  }

  // join my value with that of another Sum
  void join(const TotalDistanceWrkr& rhs) {
    distance += rhs.distance;
  }
};


// Exported function
// [[Rcpp::export]]
double TotalDistance(NumericVector x, NumericVector y) {

  // Declare TotalDistanceWrkr instance
  TotalDistanceWrkr totalDistanceWrkr(x, y);

  // call parallel_reduce to start the work
  parallelReduce(0, x.length(), totalDistanceWrkr);

  // return the result
  return totalDistanceWrkr.distance;
}
```

The usage in R is identical to the usage of C++ functions interfaced by **RCpp**.

```
(mb <- microbenchmark(d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints -
    1) * 2))
```

```
## Unit: microseconds
##                                                  expr
##  d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints - 1) * 2
##      min     lq    mean   median       uq     max neval
##  248.542 312.83 548.997 328.5945 525.4765 13358.5   100
```

```
# suppressMessages to eliminate superfluous messages
suppressMessages(autoplot(mb))
```




```
d
```

```
## [1] 0.5154879
```

The setup time for parallel tasks is much longer than the serial computation time.

Multiplying the number of points by 50, the serial computation time must be multiplied by about 2500.

```
NbPoints <- 50000
X <- runifpoint(NbPoints)
system.time(d <- MeanDistance(X$x, X$y))
```

```
##    user  system elapsed
##   4.024   0.004   4.034
```

In parallel, the time increases little: parallelization becomes really efficient. This time is to be compared to that of the reference for loop, multiplied by 2500, that is 2477 seconds.

```
system.time(d <- TotalDistance(X$x, X$y)/NbPoints/(NbPoints -
    1) * 2)
```

```
##    user  system elapsed
##   1.455   0.002   0.488
```

2.7.9 Conclusions on code speed optimization

From this case study, several lessons can be learned:

- A for loop is a good basis for repetitive calculations, faster than `vapply()`, simple to read and write.

- **foreach** loops are extremely effective for parallelizing for loops;
- Optimized functions may exist in R packages for common tasks (here, the `pairdist()` function of **spatstat** is two orders of magnitude faster than the for loop).
- the **future.apply** package makes it very easy to parallelize code that has already been written with `*apply()` functions, regardless of the hardware used;
- The use of C++ code allows to speed up the calculations significantly, by three orders of magnitude here.
- Parallelization of the C++ code further divides the computation time by about half the number of cores for long computations.

Beyond this example, optimizing computation time in R can be complicated if it involves parallelization and writing C++ code. The effort must therefore be concentrated on the really long computations while the readability of the code must remain the priority for the current code. C code is quite easy to integrate with **RCpp** and its parallelization is not very expensive with **RCppParallel**.

The use of for loops is no longer penalized since version 3.5 of R. Writing vector code, using `sapply()` is still justified for its readability.

The choice of parallelizing the code must be evaluated according to the execution time of each parallelizable task. If it exceeds a few seconds, parallelization is justified.

2.8 Workflow

The **targets** package allows you to manage a *workflow*, i.e. to break down the code into elementary tasks called *targets* that follow each other, the result of which is stored in a variable, itself saved on disk. In case of a change in the code or in the data used, only the targets concerned are reevaluated.

The operation of the flow is similar to that of a cache, but does not depend on the computer on which it runs. It is also possible to integrate the flow into a document project (see section 4.9), and even to use a computing cluster to process the tasks in parallel.

2.8.1 How it works

The documentation[13] of **targets** is detailed and provides a worked example to learn how to use the package[14]. It is not repeated here, but the principles of how the flow works are explained..

The workflow is unique for a given project. It is coded in the `_targets.R` file at the root of the project. It contains:

[13] https://books.ropensci.org/targets/
[14] https://books.ropensci.org/targets/walkthrough.html

- Global commands, such as loading packages.
- A list of targets, which describe the code to be executed and the variable that stores their result.

The workflow is run by the `tar_make()` function, which updates the targets that need it. Its content is placed in the `_targets` folder. Stored variables are read by `tar_read()`.

If the project requires long computations, **targets** can be used to run only those that are necessary. If the project is shared or placed under source control (chapter 3), the result of the computations is also integrated. Finally, if the project is a document (chapter 4), its formatting is completely independent of the calculation of its content, for possibly considerable time saving.

2.8.2 Minimal example

The following example is even simpler than the one in the **targets** manual, which will allow you to go further. It takes up the previous case study: a set of points is generated and the average distance between the points is calculated. A map of the points is also drawn. Each of these three operations is a target in the vocabulary of **targets**.

The workflow file is therefore the following:

```
# File _targets.R
library("targets")
tar_option_set(packages = c("spatstat", "dbmss"))
list(
  # Draw points
  tar_target(X,
    runifpoint(NbPoints)
  ),
  # Choose Parameters
  tar_target(NbPoints,
    1000
  ),
  # Average Distance
  tar_target(d,
    sum(pairdist(X)) / NbPoints / (NbPoints - 1)
  ),
  # Map
  tar_target(map,
    autoplot(as.wmppp(X))
  )
)
```

The global commands consist in loading the **targets** package itself and then listing the packages needed for the code. The execution of the workflow takes place in a new instance of R.

The targets are then listed. Each one is declared by the `tar_target()` function whose first argument is the name of the target, which will be the name of the variable that will receive the result. The second argument is the code that produces the result. Targets are very simple here and can be written in a single command. When this is not the case, each target can be written as a function,

stored in a separate code file loaded by the `source()` function at the beginning of the workflow file.

The `tar_visnetwork` command displays the sequence of targets and their possibly obsolete status.

```
library("targets")
tar_visnetwork()
```

The order of declaration of the targets in the list is not important: they are ordered automatically.

The workflow is run by `tar_make()`.

```
tar_make()
```

```
##   dispatched target NbPoints
##   completed target NbPoints [0.836 seconds]
##   dispatched target X
##   completed target X [0.003 seconds]
##   dispatched target d
##   completed target d [0.011 seconds]
##   dispatched target map
##   completed target map [0.022 seconds]
##   ended pipeline [0.999 seconds]
```

The workflow is now up to date and `tar_make()` does not recompute anything.

```
tar_visnetwork()
```

```
tar_make()
```

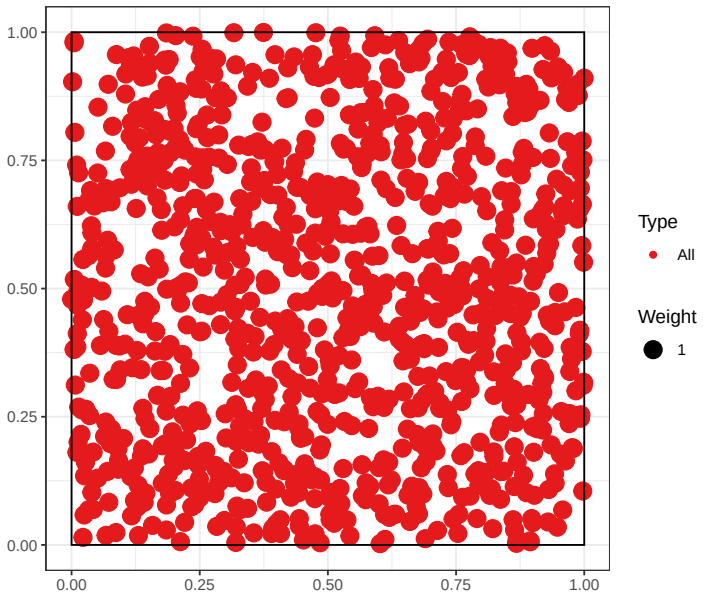
```
##   skipped target NbPoints
##   skipped target X
##   skipped target d
##   skipped target map
##   skipped pipeline [0.053 seconds]
```

The results are read by `tar_read()`.

```
tar_read(d)
```

```
## [1] 0.5165293
```

```
tar_read(map)
```



2.8.3 Practical interest

In this example, **targets** complicates writing the code and `tar_make()` is much slower than simply executing the code it processes because it has to check if the targets are up to date. In a real project that requires long computations, processing the status of the targets is negligible and the time saved by just evaluating the necessary targets is considerable. The definition of targets remains a constraint, but forces the user to structure their project rigorously.

CHAPTER 3

GIT AND GITHUB

Source control consists in recording all the modifications made on the tracked files. The advantages are numerous: traceability and security of the project, possibility to collaborate efficiently, to go back, to try new developments without jeopardizing the stable version…

3.1 Principles

3.1.1 Source control

The standard tool today is *git*.

The git commands can be executed in the RStudio terminal.

The `git status` command (figure 3.1) returns the status of the repository, that is, the set of data managed by git to track the current project.

RStudio integrates a graphical interface for git that is sufficient to do without the command line for standard use, presented here.

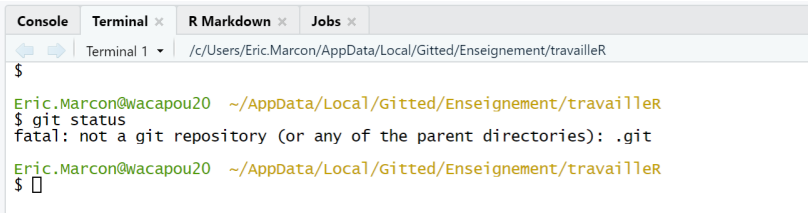


Figure 3.1: Screenshot of the RStudio terminal. The `git status` command, which is supposed to describe the state of the repository, returns an error if the R project is not under source control.

3.1.2 git and GitHub

git is the software installed on the workstation.

GitHub is a web platform[1], which allows to share the content of git repositories (to work with several people) and to share documentation in the form of a web site (*GitHub Pages*).

As GitHub allows at least the backup of git repositories, the two are always used together. GitHub is not the only platform that can be used but the main one. Alternatives are Bitbucket[2] and GitLab[3] for example.

3.2 Create a new repository

3.2.1 From an existing project

In an existing R project, enable source control in the project options (figure 3.2). The command executed is `git init`. Restart RStudio on demand.

A new *Git* window appears in the upper right panel. It contains the list of project files (figure 3.3).

At this point, the files are not taken into account by git: their status is a double yellow question mark. For git, the local working directory is a *sandbox* where all changes are possible without consequences.

The `.gitignore` file contains the list of files that are never intended to be taken into account, so there is no need to display them in the list: automatically produced intermediate files for example. The syntax of `.gitignore` files is detailed in the git[4] documentation. As a general rule, use an existing file: document templates in particular include their `.gitignore` file.

[1] https://github.com/
[2] https://bitbucket.org/
[3] https://about.gitlab.com/
[4] https://git-scm.com/docs/gitignore

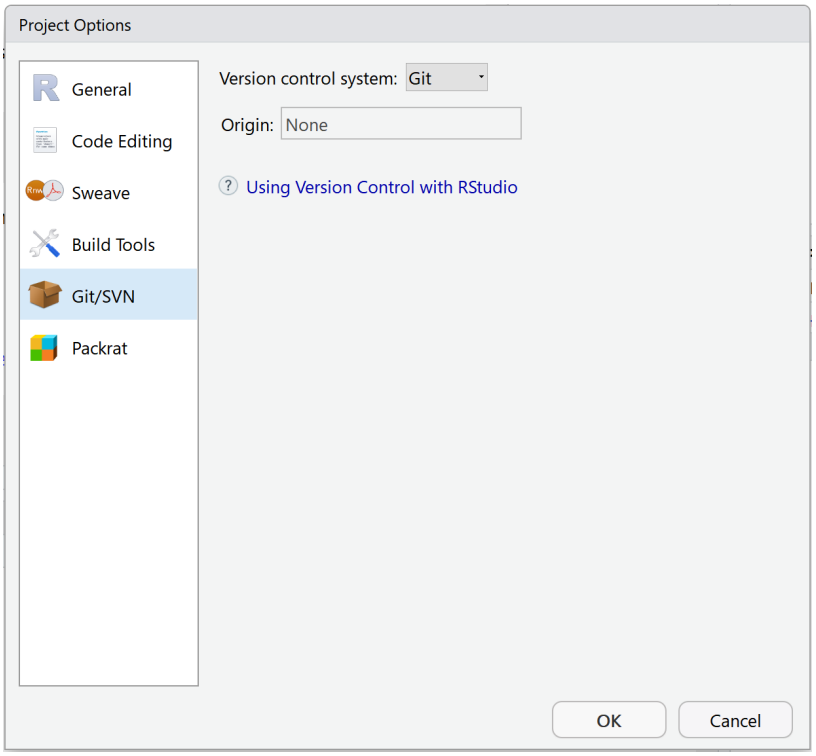


Figure 3.2: Activation of source control in the menu “Tools > Project Options…”.

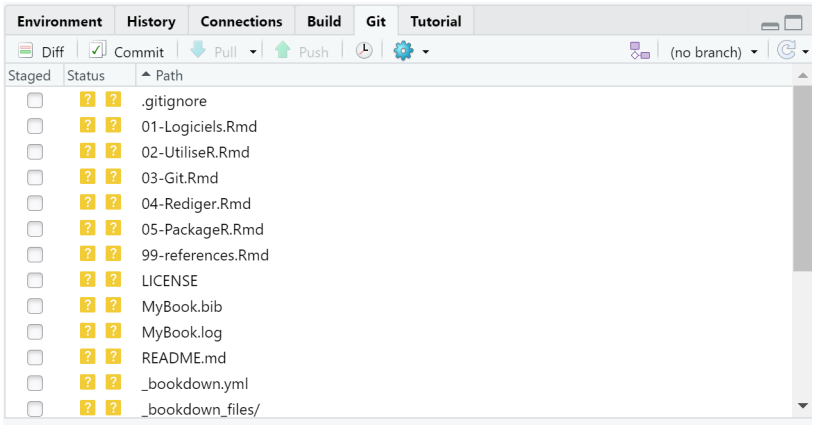


Figure 3.3: Project files, not yet taken into account by git.

3.2.2 Taking files into account

In the git window, checking the *Staged* checkbox allows you to stage each file. The command executed is `git add <FileName>`. Files taken into account the first time have the status "A" for "Added".

The files taken into account are part of the git *index*.

3.2.3 Committing changes

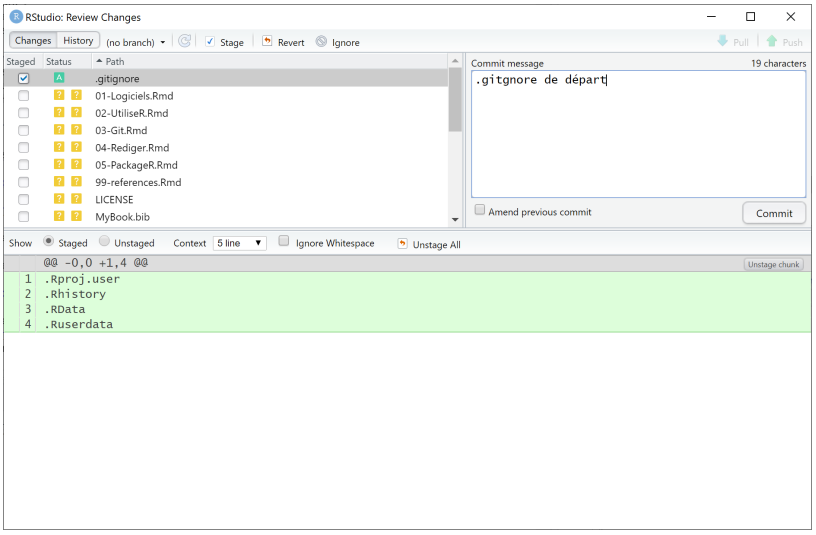


Figure 3.4: Commit window.

The staged files can be validated (*Committed*) by clicking on the "Commit" button in the *Git* window. A new window opens (figure 3.4), which allows to visualize all the modifications by file (additions in green, deletions in red). The modification grain treated by git is the text line, ended by a line break. Binary files such as images are processed as a whole.

Each commit comes with a description text. The first line is the short description. A detailed description can be added after a line break. For the readability of the project history, each commit corresponds to an action, corresponding to the short description: not all modified files are necessarily taken into account and validated at once. The command executed is `git commit -m "Commit message"`.

Commits are linked to their author, who must be identified by git. Generally, git uses the system information. If it does not succeed, a window asks the user to identify himself before making his first *commit* (figure 3.5). The commands shown are to be executed in the RStudio terminal. They can also be used to check the values known by git:

```
git config user.name
git config user.email
```

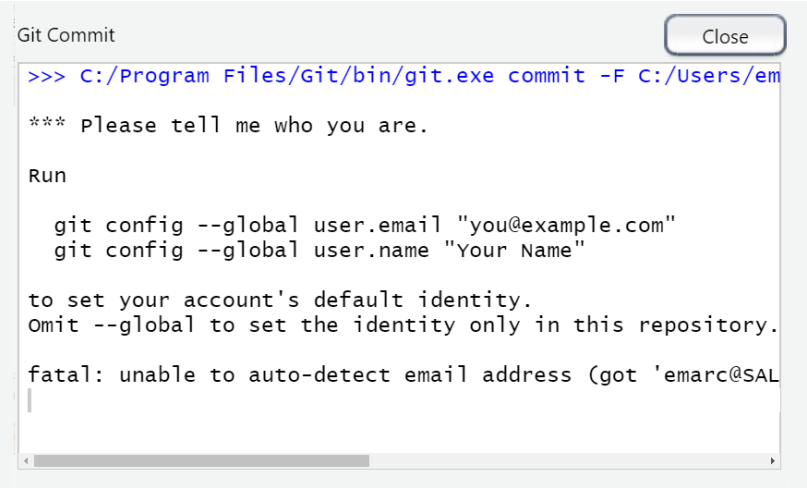


Figure 3.5: Identification window.

After the first commit, the main branch of the repository, called "master", is created. A branch is a version of the repository, with its own history and therefore its own files. Branches allow:

- To develop new features in a project, without disturbing the main branch which may contain a stable version. If the development is accepted, its branch can be merged with the *master* branch to create a new stable version.
- To host files totally different from those of the main branch, for other purposes. On GitHub, a project's web pages can be placed in a branch called "gh-pages" which will never be merged.

The git repository is fully constituted. In git vocabulary, it consists of three *trees* (figure 3.6):

- The working directory, or sandbox, which contains files that are not staged: unknown, modified, deleted or renamed ("Staged" box unchecked).
- The index, which contains the files taken into account ("Staged" box checked).
- The head, which contains the validated files.

The status of the files is represented by two icons in the RStudio *Git* window: two question marks when they are not in git's index. Then, the icon on the right describes the difference between the working directory and the index. The one on the left describes the difference between the index and the head. So a modified file will have the `M` icon displayed on the right before it is taken into account, then on the left after it is taken into account. It is possible, although better to avoid it, to modify again a file that has been taken into account before it is validated: then, both icons will be displayed.

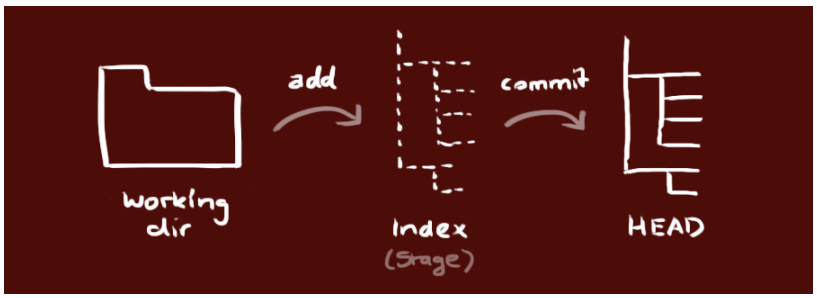


Figure 3.6: git's trees. Source: https://rogerdudler.github.io/git-guide/index.fr.html

3.2.4 Create an empty repository on GitHub

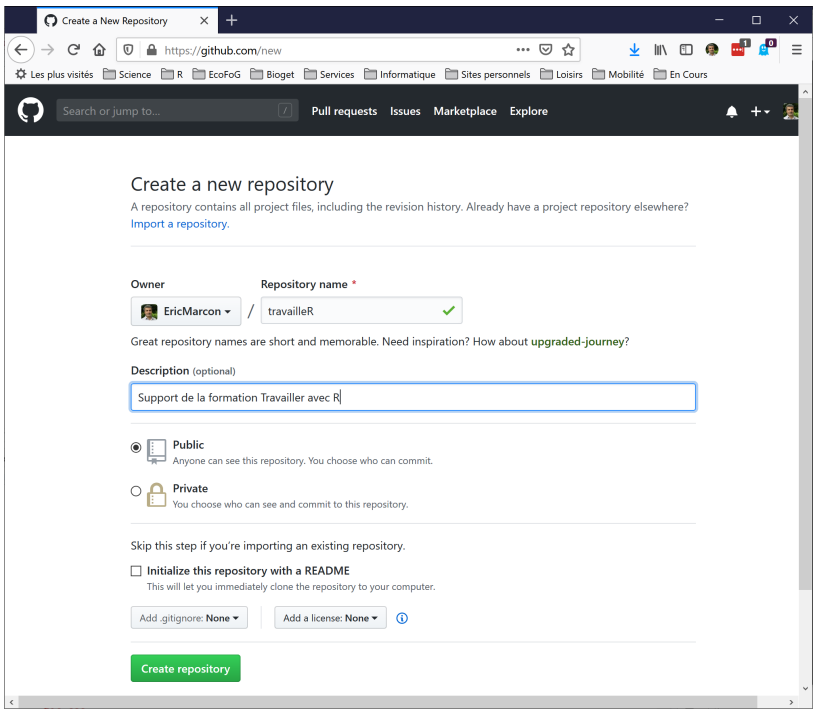


Figure 3.7: Create a repository on GitHub.

An empty repository on GitHub must be created (figure 3.7):

- On GitHub, click on the green "New repository" button.
- Enter the name of the repository, identical to the local R project.
- Add a description, which will appear only on the GitHub page of the repository.
- Choose the status of the repository:
  - Public: visible to everyone.
  - Private: visible only to project collaborators, which excludes adding web pages.

- Do not add any `README`, `.gitignore` or license: the project must be empty.
- Click on "create Repository".
- Copy the address of the repository (https://github.com/… or git@github.com:…).

The choice of the address is linked to the authentication method. SSH authentication (see section 1.4.3) is preferred.

3.2.5 Linking git and GitHub

In RStudio, a first *commit* must at least have taken place for the main branch of the project, named "master", to exist. At the top right of the *Git* window (figure 3.3), it shows "(no branch)" before that. Then it is displayed "master", the default name of the main branch of the project. The project can then be linked to the GitHub repository.

Graphical method

Click on the purple button next to "master": a window appears (usually used for creating a new branch, see section 3.4). Enter the name of the "master" branch, click on "Add Remotes" and complete:

- Remote Name: `origin`.
- Remote URL: paste the address of the GitHub repository.
- Click on "Add".

Check the "Sync with Remote" box.

At the message indicating that a *master* branch already exists, click on "Overwrite".

On the command line

Instead of the previous manipulation, the link between Git and GitHub can be set up by some git commands executed in the RStudio terminal. These are displayed on the home page of any newly created empty repository on GitHub and can therefore be copied and pasted directly to the terminal.

```
git remote add origin git@github.com:<GitHubID>/<RepoID>.git
git branch -M master
git push -u origin master
```

The first command declares the GitHub repository as a remote repository. The name *origin* is a git convention. It can be changed, but the organization of the project will be more readable if it follows the convention. The repository address is `https://github.com/<GitHubID>/<RepoID>.git` if HTTPS authentication is chosen.

The following commands activate the main branch of the project and push its content to GitHub.

Be careful with the name of the main branch (see section 3.4): by default, it is called "master" in a project created in RStudio but "main" on GitHub. The above command lines provided by GitHub therefore replace `master` with `main` and must be corrected to match the name of the branch created by RStudio.

Authentication

If HTTPS authentication is chosen, the first time RStudio connects to GitHub, a window allows you to enter your GitHub credentials (figure 3.8).

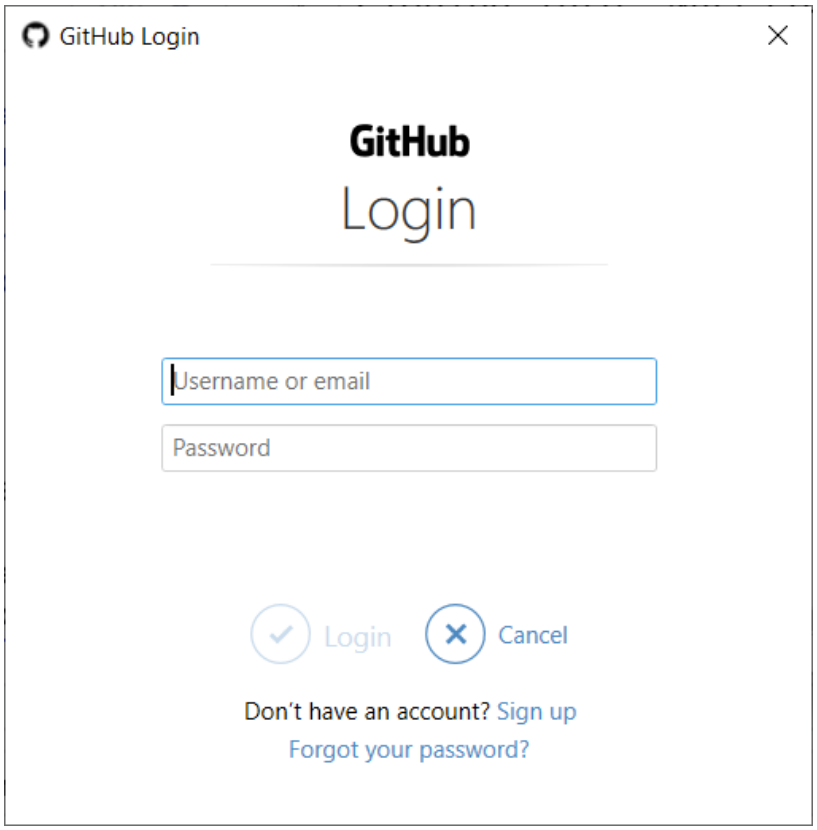


Figure 3.8: Identification HTTPS sur GitHub.

As of August 2021, GitHub no longer accepts the user's account password for this authentication: the personal token (PAT) created in section 1.4.4 must be entered instead.

If SSH authentication is chosen and has been configured at git installation (section 1.4.3), no action is necessary.

3.2.6 Push the first modifications

The previous manipulation has automatically pushed the validated modifications on GitHub. Afterwards, you will have to click on the "Push" button of the *Git*

window to do it.

On GitHub, the files resulting from the modifications recorded by git are now visible.

Each commit done locally is counted by git and a message "Your branch is ahead of 'origin/master' by *n* commits" displayed in the top of the *Git* window indicates that it is time to update GitHub by pushing all these commits. Click on the "Push" button to do so.

At this point, the project should have a `README.md` file that presents its contents on GitHub. Its minimal content is a title and a few lines of description:

```
# Project name

Description in a few lines.
```

It is advisable to use badges[5], to be placed just after the title, to declare the maturity status of the project, for example:

```
![stability-wip](https://img.shields.io/badge/|>
stability-work_in_progress-lightgrey.svg)
```

3.2.7 Clone a repository from GitHub

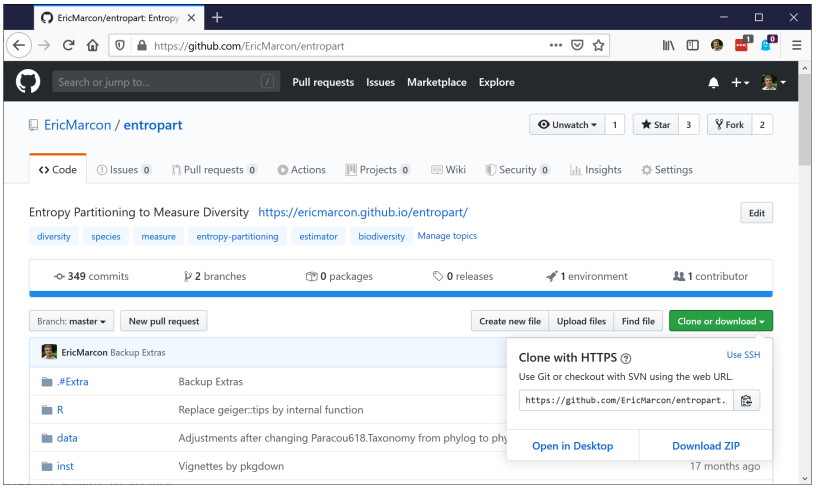


Figure 3.9: Cloning a repository from *GitHub*.

Any repository on GitHub can be installed (*cloned*) on the workstation by copying its address which appears by clicking on the green button (figure 3.9).

In RStudio, create a new project and, in the wizard, choose "Version Control", "Git" and paste the address in the "Repository URL" field. The name of the directory to create for the project is automatically deduced from the address. Choose the directory in which the project will be created and click on "Create Project". The created project is linked to the remote repository on GitHub.

[5] https://github.com/orangemug/stability-badges

To work with several people on the same project, the project owner must give access to the project to collaborators (figure 3.10), i.e. other GitHub users in the repository settings.

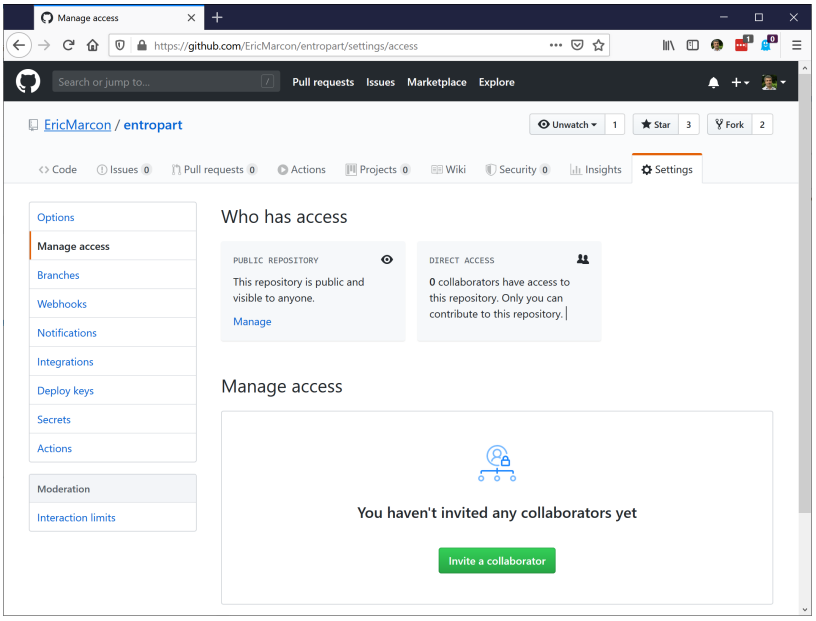


Figure 3.10: Assigning access rights on GitHub.

Collaborators are invited by a message sent by *GitHub*.

3.3 Common usage

3.3.1 Pull, modify, commit, push

Any work session on a project starts by pulling ("Pull" button) from the *Git* window to integrate to the local repository the updates made on GitHub by other collaborators.

The changes made to the project files are then taken into account (check the "Staged" boxes) and committed with an explanatory message. A good practice is to validate changes each time an elementary task which can be described in the explanatory message is completed rather than making *commits* that group many changes with a vague description.

As soon as possible, *Push* updates so that they are visible to collaborators.

3.3.2 Resolve conflicts

It is not possible to push validated changes if a collaborator has modified the remote repository on GitHub. In this case, you have to pull them to the local repository before pushing the merged changes.

A conflict occurs if a *Pull* imports a change into the local file that cannot be merged automatically because a conflicting change occurred locally. Git considers each line to be indivisible: changing the same line in the remote repository and the local repository therefore generates a conflict.

Git inserts both versions in the file containing a conflict with a particular presentation:

```
<<<<<<<<< HEAD # Imported version of the conflict
Lines in conflict, imported version
========= # boundary between the two versions
Lines in conflict, local version
>>>>>>>>> # End of conflict
```

The formatting lines containing the <<<<, the ==== and the >>>> must be deleted and only one version of the problematic lines kept, which can be different from the two original versions. The conflict resolution must be taken into account and validated.

To limit conflicts in a document containing text (typically, an R Markdown document), a good practice is to treat each sentence as a line, terminated by a line break that will not be visible in the formatted document: two line breaks are required to separate paragraphs.

3.3.3 See the differences

In the RStudio *Git* window, the context menu (displayed by right-clicking) "Diff" can be used to display the changes made to each file (figure 3.11).

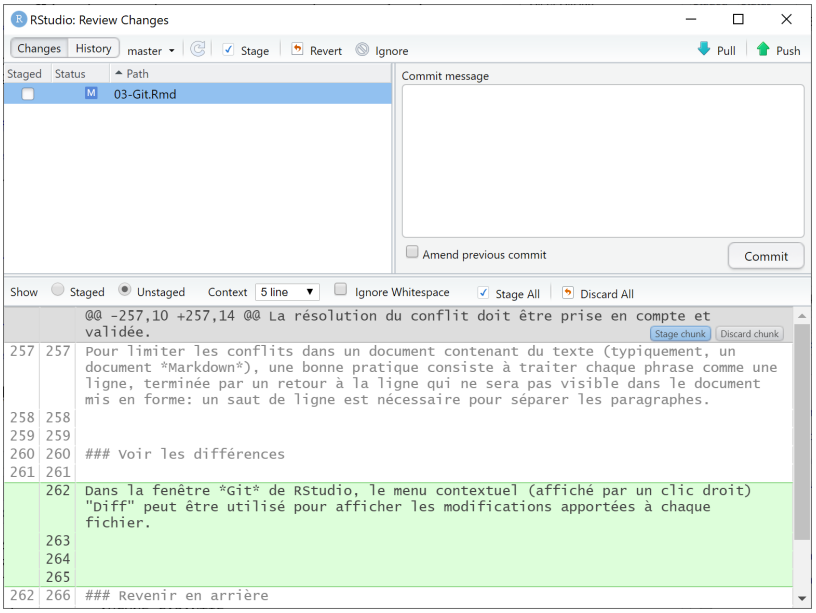


Figure 3.11: Differences between the working directory and the head.

3.3.4 Revert

The contextual menu “Revert” allows you to undo all the modifications made to a file (displayed by *Diff*) and to restore its content validated the last time (its state in the head).

It is not easy to go back beyond the last validation because the modifications may have been taken into account by collaborators: deleting them would make the project incoherent.

3.3.5 View history

The clock-shaped button in the RStudio *Git* window displays the history of the project (figure 3.12).

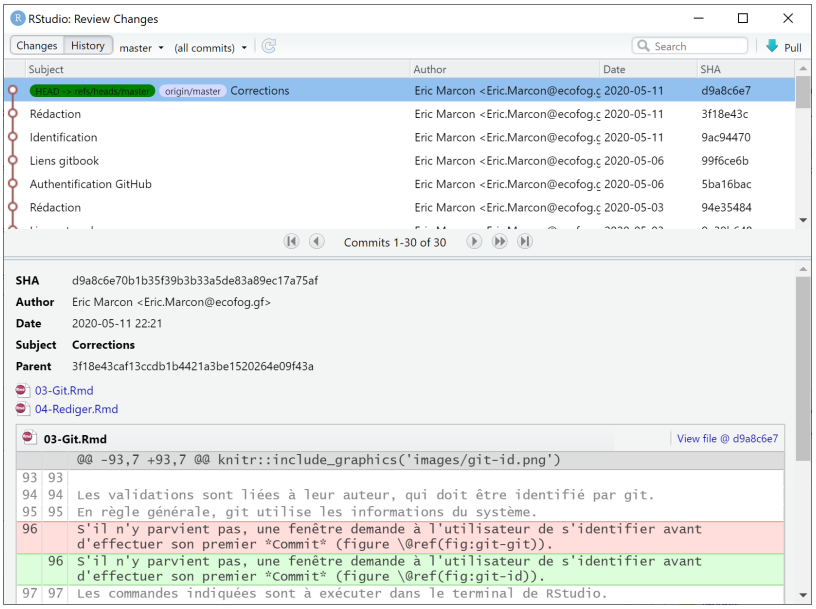


Figure 3.12: History of commits in the repository.

At the top is the head, and then all the commits that made it up. For each validation, the differences of each file can be displayed by clicking on the file name in the lower part of the window.

3.4 Branches

The branches of a project are different but simultaneous versions. A typical use is the development of a new feature. If it takes a long time to write, the project is disturbed by the current work in progress: the code may not work anymore. If the development turns out to be impossible or useless, it must be abandoned without damage. In order to isolate it during its development and to be able to validate or abandon it at the end, it must be placed in a branch.

The main branch of the project is called “master” or “main” from November 2020[6]. It must always be in a stable state: it is the one that is cloned from GitHub by other possible users.

The change of convention for the name of the “master” branch means that from November 2020, projects created on GitHub cloned in RStudio have the main branch “main” while projects created on RStudio and then linked to GitHub keep the “master” name.

3.4.1 Create a new branch

Click on the purple “New Branch” button in the RStudio *git* window. Enter its name and click on “Create”.

The new branch is now active.

The git commands can also be run in the terminal (to create the branch and activate it):

```
git branch new_branch
git checkout new_branch
```

3.4.2 Change branch

Select the branch to activate from the list of local branches in the *git* window.

The *commits* apply to the active branch. Each branch behaves as a different version of the project.

Warning: to avoid confusion, save the changes, stage them and commit them before changing branch.

3.4.3 Pushing the new branch

The first modifications of the new branch must be pushed with the command line because the “Push” and “Pull” buttons of the *Git* window do not work as long as the branch does not exist on the remote repository.

Run, in the terminal:

```
git push -u origin new_branch
```

3.4.4 Filesystem behavior

Each time a branch changes, git rewrites the project files to reflect the state of the branch. The changes can be seen outside of RStudio, in the file browser for example.

Files ignored by `.gitignore` are not changed. It is therefore essential that the `.gitignore` files in the different branches are identical, otherwise files ignored in one branch will appear as added in the displayed branch after a change.

[6] https://github.com/github/renaming

Development branches have a content close to that of the main branch. This is not the case with specialized branches seen later, such as `gh-pages` (see section 3.7) which contains the repository's presentation web site. It is best not to attempt to display these branches in RStudio: their content is produced automatically and should not be modified manually. If it is necessary, it will be necessary to copy the `.gitignore` file of the main branch and keep in mind that the ignored files actually belong to another branch than the one displayed.

3.4.5 Merge with `merge`

Merging a development branch with the main branch marks the achievement of its goal: its code will be integrated into the project. RStudio's GUI does not allow for merging, so you have to use the terminal: first, go to the target branch (possible with the GUI):

```
git checkout master
```

Then, merge:

```
git merge new_branch
```

In most situations, the merge will be automatic ("Fast Forward"). It is possible that conflicts appear: use the `git status` command to display the list of files concerned, open them, resolve the conflict and perform a *commit*.

The merged branch is not deleted: it can be used again for further development or deleted manually with the following command:

```
git branch -d new_branch
```

3.4.6 Merging with a pull request

The other way of merging is more formal but also more general: it allows you to merge a branch into another user's repository to contribute to it, or to have your branch validated by another team member in a collaborative project.

To contribute to another GitHub user's project[7], you have to start by creating a *fork* of it, i.e. a copy in the form of a repository linked to the original. It will be possible to pull changes from the original to stay up to date[8] (as opposed to a simple snapshot copy made by downloading a zip of the project) and, at the end of the development, to merge the *fork* to the original repository (as opposed to a clone that would not allow to contribute afterwards).

Next, create a development branch as before, modify it and finally ask the repository owner to merge it. This process is described in detail in the git documentation.

[7] https://git-scm.com/book/fr/v2/GitHub-Contribution-%C3%A0-un-projet
[8] https://ardalis.com/syncing-a-fork-of-a-github-repository-with-upstream/

In the simpler case of a branch of one's own project as in the case of a *fork*, the development branch is ready to be merged. It must have been pushed on GitHub. On the GitHub page of the project, a "Create Pull Request" button allows to request the merge. A message describing the proposed changes with their arguments must be added.

The owner of the project (the team members in the case of a collaborative project, or yourself if the team is reduced to one person) is notified of the pull request. On the original project page, it is possible to see the message, the list of modifications (chronology of *commits* or comparison of files), to start a discussion with the author of the request… If the request is not accepted, it can be closed. If it is validated, the "Merge Pull Request" button allows to merge the development branch with the "master" branch (or another one) of the source project.

Pull requests are the only way to contribute to a repository on which you don't have write rights. It is also the way to merge a development branch into your own project by keeping an explicit trace of it (in the *Pull requests* section of the project's GitHub page). In a collaborative project, the proposals of a member (author of the request) can be validated by another (who accepts the merge).

3.5 Advanced usage

3.5.1 Git commands

Beyond the common use allowed by the RStudio graphical interface, advanced manipulations of projects are allowed by using git in command line. Some useful examples are presented here.

A short guide of commands is proposed by Roger Dudler[9]. It summarizes the essential commands, thus integrated in the graphical interface of RStudio. Links to more complete references are given at the bottom of the page.

3.5.2 Size of a repository

To find out how much disk space a repository occupies, use the command `git count-objects -vH`[10].

The data for this document at the time of writing is presented as an example.

```
$ git count-objects -v
count: 200
size: 2.66 MiB
in-pack: 0
packs: 0
size-pack: 0
prune-packable: 0
garbage: 0
size-garbage: 0
```

[9] https://rogerdudler.github.io/git-guide/index.fr.html
[10] https://git-scm.com/docs/git-count-objects

The total size is on the *size* line. Packs are a method used by git to reduce the size of the repository: similar files are stored as a common part and differences. The *prune-packable* line gives the size of objects stored both individually and in packs. If their size is large, run `git prune-packed` to reduce it to zero.

The *size-garbage* line gives the size of objects that can be deleted. `git gc` removes them, but not only that: it optimizes storage.

```
git gc
Enumerating objects: 194, done.
Counting objects: 100% (194/194), done.
Delta compression using up to 8 threads
Compressing objects: 100% (188/188), done.
Writing objects: 100% (194/194), done.
Total 194 (delta 83), reused 0 (delta 0)

$ git count-objects -vH
count: 1
size: 5.72 KiB
in-pack: 194
packs: 1
size-pack: 4.00 MiB
prune-packable: 0
garbage: 0
size-garbage: 0 bytes
```

Here, the majority of the objects in the repository have been placed in a pack (but its size is larger than the individual objects).

There is usually no need to do garbage collection manually: git handles the organization of its repositories well.

GitHub limits the size of repositories. As of May 2020, the limit is 100 GB. The size of all repositories of an authenticated user can be displayed in his account settings ("Personal Settings", "Repositories")[11].

3.5.3 Delete a folder

All changes made to a repository are stored in its history. It can be useful to delete them in some particular cases

- if a file containing confidential information was inadvertently committed. Committing its deletion does not remove it from the history, and the confidential information remains visible when consulting the history.
- if large files are no longer needed, e.g. PDF files produced by R Markdown (chapter 4) which are binary (thus unsuitable for git) and reproducible from the code.

Typically, the `docs` folder is used to store documents produced from R Markdown code. The HTML and PDF files must be in this folder to constitute the GitHub pages of the project. Each modification of the repository generates a

[11] https://github.com/settings/repositories

new version of these files whose history volume quickly becomes huge. An efficient solution is to delegate the creation of these files to a continuous integration system (chapter 6) and to remove the `docs` folder from the main branch (*master*) of the repository. You then have to delete all its history to recover the space it occupies, which can be most of the size of the repository.

The commands to completely delete a folder from a repository are presented here[12]. The repository must be clean, i.e. without unvalidated changes, and the remote and local versions synchronized.

The following three commands completely remove the `docs` folder from the git repository history:

```
git filter-branch --tree-filter "rm -rf docs" |>
    --prune-empty HEAD
git for-each-ref --format="%(refname)" refs/original/ |>
    | xargs -n 1 git update-ref -d
```

The `docs` folder is not removed from the working directory. It must therefore be added to the `.gitignore` file so that it is no longer tracked. The modification of `.gitignore` must be validated. These operations can be done with the RStudio interface or on the command line:

```
echo docs/ >> .gitignore
git add .gitignore
git commit -m 'Removing docs folder from git history'
```

The repository cleanup is necessary to physically remove the removed data:

```
git gc
```

Finally, the repository must be pushed. The `--force` option involves replacing the contents of the remote repository with those of the local repository: all changes made by collaborators are erased, so this cleanup operation involves suspending the development of the project while it takes place.

```
git push origin master --force
```

This code can be used to completely remove any file or folder from a repository by simply replacing `docs` in the initial `git filter-branch` command. The reduction in repository size can be tracked using `git count-objects -vH` before the operation, before `git gc` (the repository size remains stable but has been moved to *garbage*) and at the end (the repository size is significantly reduced).

[12] https://stackoverflow.com/questions/10067848/remove-folder-and-its-contents-from-git-githubs-history

3.5.4 Revert

It is possible to restore a repository to a previous state by placing its head (figure 3.6) at the level of an old *commit*. All subsequent modifications are then destroyed. This operation should not be performed on a shared repository: other users would not be able to push their modifications anymore.

Display the repository history and look for the identifier (SHA) of the last *commit* to keep. In the RStudio terminal, run:

```
git reset --hard <SHA>
git push -f
```

All repository history after the chosen restore point is lost.

A less drastic method that can be used on a shared repository is to perform a *commit* that undoes another's changes but does not destroy any history data. This operation only undoes one *commit* at a time, so it must be repeated to undo several, starting with the most recent. In the RStudio terminal, run:

```
git revert <SHA>
```

To undo the last *commit*, execute:

```
git revert HEAD
```

Using `HEAD` simply avoids searching for the corresponding ID.

3.6 Confidential data in a public repository

A public repository on GitHub causes problems when the data used in the project is not.

An unsatisfactory solution is to not include the data in the project, which makes it non-reproducible. A better solution is to encrypt them, allowing some users to decrypt them. This is the purpose of the **secret** package.

A safe (`vault` folder) is created in the project. It contains a list of authorized users: each of them must have a pair of encryption keys, a public key included in the safe and a private key, kept secret. The data is encrypted with all the public keys available (and therefore duplicated). The users then each use their own private key for decryption.

To avoid duplicating data, the repository owner should create a generic user for the project, whose private key he will communicate outside GitHub. The vault will contain the keys of the project owner and the generic user only. If the generic user's private key is compromised, it will be sufficient to remove it from the vault and create a new one.

3.6.1 Generating a key pair for the project owner

The keys are generated by the *ssh* software, installed with *git* or by default on Linux.

The procedure is the same as in the section 1.4.3, but the key used must be in RSA format (supported by the **secret** package, as opposed to the more secure ed25519 format used for authentication on GitHub).

Run the following command in the RStudio terminal to create an RSA key:

```
ssh-keygen -t rsa -b 4096 -C "user.email"
```

Store the public key on GitHub in "Settings > SSH and GPG Keys". Note the position of the key: if an authentication key has already been stored for two workstations for example, the RSA key will be the third one.

3.6.2 Generating a key pair for the project

Generate a key in RSA format in the RStudio terminal:

```
ssh-keygen -t rsa -b 4096"
```

- Enter the name of the key: `<RepoID>.rsa`.
- Do not enter a passphrase to allow the key to be used without interaction.

The private key `<RepoID>.rsa` should only be distributed to the rightful owners of the project. You must therefore add the line `*.rsa` to the `.gitignore` file of the project to avoid pushing the key on GitHub.

To allow the continuous integration of the project (chapter 6), the private key must be stored as a secret of the GitHub repository containing the project. Apply the procedure in section 6.2.2 to create a secret named "RSA" and paste the content of the file `<RepoID>.rsa` in the "Value" field of the form.

The use of the secret is described in section 6.2.2.

3.6.3 Creating a safe

Execute:

```r
library("secret")
vault <- "vault"
create_vault(vault)
```

3.6.4 Adding users

The owner of the project is added from his public key stored on GitHub, which is the third one in our example.

```
# GitHub ID of the project owner
github_user <- "EricMarcon"
# Read and store the key, i is the key number
add_github_user(github_user, vault = vault, i = 3)
```

The generic project user's key is added by:

```
library("openssl")
```

```
## Linking to: OpenSSL 3.3.0 9 Apr 2024
```

```
project_id <- "ProjectName"
# Read the key
rsa_project <- read_pubkey(paste0(project_id, ".rsa.pub"))
# Add to the vault
add_user(project_id, public_key = rsa_project, vault = vault)
```

3.6.5 Storing the data

The data, stored in R variables, are stored one by one by the `add_secret()` function. In the following example, the variable is called `X` and equals 1.

```
X <- 1
add_secret(
  # Name of the data
  "X",
  # Value
  value = X,
  # Authorized users: owner and generic
  users = c(paste0("github-", github_user), project_id),
  # Vault
  vault = vault)
```

The contents of the vault can be checked:

```
# List of vault data
list_secrets(vault = vault)
```

```
##   secret        email
## 1      X github-E....
```

```
# List of owners of the data 'X
list_owners("X", vault = vault)
```

```
## [1] "github-EricMarcon" "ProjectName"
```

The data will be read into the project code by the `get_secret()` command. The private key of the generic project user, communicated by a secure means to the owners, must be in the project folder.

```
# Select the private key
Sys.setenv(USER_KEY = usethis::proj_path(paste0(project_id, ".rsa")))
```

```
## v Setting active project to
## '/Users/runner/work/WorkingWithR/WorkingWithR'
```

```
# Read the data 'X
get_secret("X", vault = vault)
```

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

The key can be verified:

```
local_key()
```

```
## [4096-bit rsa private key]
## md5: e81dcb0745a755286c2dc1fc4c6ad117
## sha256: cca11ef82e17c3b77b699e7f3c23e083e8f0f79cb70be8274799f076c44b0c2d
```

3.7 GitHub pages

Any project on GitHub must have contained a `README.md` file to present it. This file is written in Markdown format.

The file can be placed in the `docs` folder to provide both the repository's home page and its website. The **memoiR** package provides commands to automate these tasks in document projects. A repository containing an article written in R Markdown (see section 4.3.2) is used as an example[13].

Its `README.md` file exists in both locations: it is written by the developer at the root of the project and duplicated in `docs`.

3.7.1 Activation

To activate GitHub pages, you have to open the repository settings and modify the "GitHub Pages" item (in "Options"). Select the project branch and the folder containing the web pages, here: `master` and `/docs`. As an option, choosing a theme customizes the appearance of the pages.

The web site is accessible at an address[14] of the domain *github.io*.

The `README.md` file displayed on the home page has a very different look but the same content as the one displayed with the code on the repository page in GitHub.

The interest of the GitHub pages is to allow an easy access to the formatted documents when the repository contains a written production or to the documentation of R packages. These contents will be presented in the next chapter.

A main website is proposed with each GitHub account, at https://GitHubID.github.io[15]. It will be used to host a personal website produced by **blogdown**.

[13] https://github.com/EricMarcon/Krigeage
[14] https://EricMarcon.github.io/Krigeage/
[15] Example: https://EricMarcon.github.io/Krigeage/

3.7.2 Badges

Badges are small images, possibly dynamically updated, that provide quick information about the status of a project. They should be placed immediately after the title of the `README.md` file.

A good practice is to indicate the progress in the life cycle of the project. The corresponding badges are listed on the Tidyverse site[16].

Their Markdown code is as follows:

```
![stability-wip]
(https://img.shields.io/badge/lifecycle-maturing-blue.svg)
```

The **usethis** package simplifies their creation by placing the necessary code in the clipboard. Then just paste it into the file.

```
usethis::use_lifecycle_badge("maturing")
```

[16] https://www.tidyverse.org/lifecycle/

CHAPTER 4

WRITING

R and RStudio make it possible to efficiently write documents of all formats, from simple notepads to theses to slide shows. The tools to do this are the subject of this chapter, completed by the production of web sites (including a personal site).

Two document production processes are available:

- *R Markdown* with the **knitR** and **bookdown** packages. This is the classic method, presented here in detail.
- *Quarto*, designed to be used with languages beyond R and in working environments beyond RStudio. Quarto is under active development but does not yet allow documents to be produced with the same quality as *R Markdown*: for example, punctuation in French documents is not handled correctly in PDF[1], tables cannot include equations[2] and the width of fig-

[1] https://github.com/jgm/pandoc/issues/8283/
[2] https://github.com/quarto-dev/quarto-cli/issues/555

ures is inconsistent in PDF documents formatted with several columns[3]. The use of Quarto is well documented on its site[4] and is not presented here.

4.1 Markdown notebook (R Notebook)

In an `.R` file, the code should always be commented to make it easier to read. When the explanation of the code requires several lines of comment per line or block of code, it is time to reverse the logic and place the code in the text.

The concept of literate programming was developed by Knuth (1984). It consists in describing the objectives and methods by text, in which the code is integrated.

The simplest tool is the Markdown notebook (Menu "File > New File > R Notebook"). The document template contains its instructions for use.

The language for formatting the text is Markdown[5], an easy to use markup language:

- Paragraphs are separated by line breaks.
- The document is structured by headings: their line starts with a number of # corresponding to their level.
- Character formats are limited to the essentials: italic or bold (text surrounded by one or two *).
- Other simple codes allow all useful formatting.

This language is the core of the pandoc[6] software, dedicated to converting documents of different formats.

The **rmarkdown** package (Xie 2015) bridges the gap between R and Markdown, relying on the RStudio interface which is not essential but greatly simplifies its use. The Markdown dialect used by the package is called *R Markdown*. Its syntax is summarized in a cheat sheet[7]. Its complete documentation is online (Xie et al. 2018).

Equations are written in the LaTeX format[8].

The simplest organization of a *R Markdown* document can be seen in the notepad template. It starts with a header in YAML format[9]:

```
---
title: "R Notebook"
output: html_notebook
---
```

[3] https://github.com/quarto-dev/quarto-cli/issues/855
[4] Exemple: https://EricMarcon.github.io/Krigeage/
[5] https://fr.wikipedia.org/wiki/Markdown
[6] https://fr.wikipedia.org/wiki/Pandoc
[7] https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
[8] https://fr.wikibooks.org/wiki/LaTeX/%C3%89crire_des_math%C3%A9matiques
[9] https://fr.wikipedia.org/wiki/YAML

The first entry is the title, the second is the output format: more precisely the name of the function that will process the document.

The document contains Markdown formatted text and code chunks surrounded by three backquotes (the Markdown syntax of a code block) and a language description, here `r`. These code chunks are processed by **knitr** which transforms the result of the execution of the R code into Markdown and integrates it into the text of the document.

Processing an R Markdown document is called *knitting*. The production chain is as follows:

- **knitr** processes the code snippets: calculations, figure production.
- **rmarkdown** integrates the production of code and text snippets to produce a standard Markdown file.
- pandoc (installed with RStudio) converts this file to HTML, LaTeX or Word format.
- LaTeX produces a PDF file when that format is requested.

RStudio allows knitting to be started by buttons rather than commands: in the source window (the top left one), a "Knit" button accompanies R Markdown documents. For R Markdown notebooks, it is replaced by a "Preview" button with the same functions. It can be scrolled down to choose the output format: HTML, Word, PDF (via LaTeX) and, for notepads, a "Preview" command that displays the document in HTML without executing the code snippets to save time. As soon as the first knitting is done in Word or HTML format, you will notice that the "Preview" button disappears.

In the end, using R Markdown combines several advantages:

- Simplicity of writing: the raw text is easier to read and format than in LaTeX for example.
- Automation of the production: formatting and layout are fully automatic.
- Reproducibility: each document can be self-sufficient with its data. Re-knitting regenerates the whole document, including the necessary calculations and the production of figures.

It also has some disadvantages:

- Formatting depends on templates, and developing new templates is not easy.
- Knitting errors are sometimes difficult to correct, especially when they occur at the LaTeX compilation stage.
- Reproducibility consumes computing time. To limit this problem, a cache system allows not to re-evaluate all the R code bits at each modification of the text. The production of large documents can also be delegated to a continuous integration system (see chapter 6).

4.2 R Markdown templates

More elaborate document templates than the notepad are provided by packages, including **rmarkdown**. They are accessible via the menu "File > New File > R Markdown…" (figure 4.1).

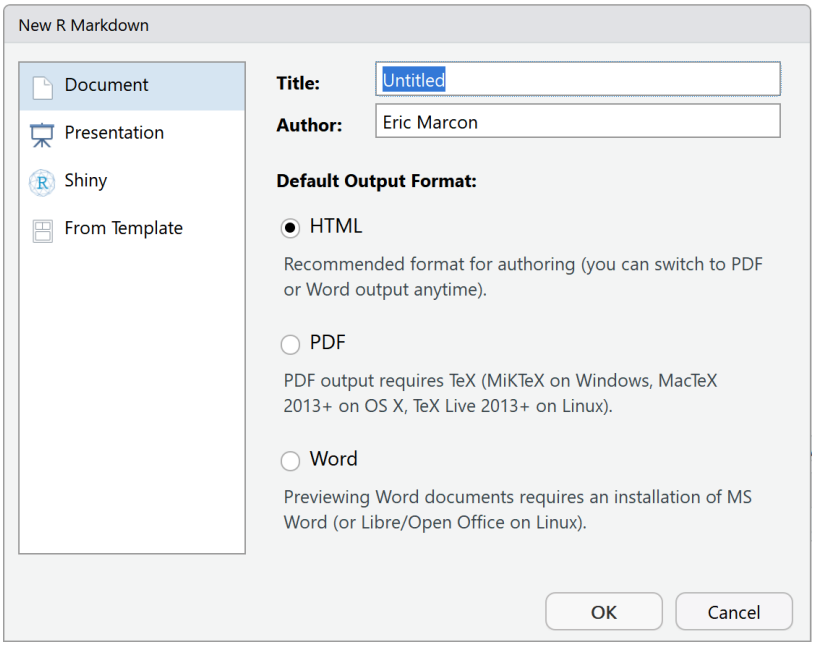


Figure 4.1: New Markdown document from a template.

The simplest templates are *Document* and *Presentation*. The information to be provided is the title and the name of the author, and the format of the expected document (which can be modified later). These templates create a single file which will only need to be saved when knitting.

The syntax is the same as for the notepad. In the header, an extra entry is used for the date, which can be calculated by R at each knitting:

```
date: "|r format(Sys.Date(), '%d/%m/%Y')|"
```

Replace the vertical bars | in the above example with backquotes: since this document is written with R Markdown, the date would be calculated and displayed instead of the code if the backquotes were used directly.

Inline R code (as opposed to code snippets) can be used anywhere in an R Markdown document, including in the header for the date display. It starts with a backquote followed by 'r' and ends with another backquote.

Documents can be knitted in HTML, PDF (via LaTeX) or Word format. The header of the R Markdown file is rewritten when the knitting is started by the RStudio button which places the current output format on top of the list.

Presentations can be knitted in two HTML formats, ioslide[10] or Slidy[11], in Beamer (PDF) format[12] or in Powerpoint[13].

The level 2 outline (##) marks the change of slide.

Additional code, presented in the HTML format documentations, allows for specific functionality.

These templates are simple but not very useful: the R notepad is easier to use than the document template for minimalist documents. More elaborate templates are available.

4.3 Articles with bookdown

R Markdown does not allow you to write a scientific article. Bibliography is not a problem because it is handled by pandoc for HTML or Word documents and outsourced to LaTeX for PDF documents. Equations, figures and tables are numbered by LaTeX but not in HTML. Cross-references (references to a figure number for example) are not supported. Finally, figure and table captions only support plain text, without any formatting.

bookdown fills these gaps. The package has been designed for writing books with several chapters but can be used for articles.

The **memoiR** package provides the templates shown here. It must be installed.

4.3.1 Writing

The main features of Markdown are summarized here. A quick and more complete training is offered by RStudio[14].

The text is written without any formatting other than line breaks. A simple line break has no effect on the document produced: it allows to separate sentences to simplify the tracking of the source code by git.

A line break marks a paragraph change.

The different levels of the plan are designated by the number of hashes at the beginning of the line: # for a level-1 title, ## for a level-2 title, etc. A space separates the hashes and the title text.

Bullet lists are marked by a dash (followed by a space) at the beginning of the line. A double line break is required before the beginning of the list, but the elements of the list are separated by a simple line break. Indented lists are created by inserting 4 spaces before the dash at the beginning of the line. Last, numbered lists are created in the same way by replacing the hyphens by numbers, whose value is not important.

[10] https://bookdown.org/yihui/rmarkdown/ioslides-presentation.html
[11] https://bookdown.org/yihui/rmarkdown/slidy-presentation.html
[12] https://bookdown.org/yihui/rmarkdown/beamer-presentation.html
[13] https://bookdown.org/yihui/rmarkdown/powerpoint-presentation.html
[14] https://rmarkdown.rstudio.com/lesson-1.html

In the text, the italicized parts are surrounded by a star or an underscore (`*italic*`), while two stars mark the bold.

R code

R code is included in code chunks that are easily created by clicking on the "Insert a new code chunk" button above the source code window in RStudio. They start and end with three backquotes on a new line. These code chunks can contain R code but also Python code for example: the type of code is indicated in the header on the first line, before the name of the code chunk, then a comma-separated list of options, for example:

````
```{r cars, echo=TRUE}
```
````

The name and options are optional: the minimum header is `{r}`.

The most useful options are :

- `echo` to show (`=TRUE`) or hide (`=FALSE`) the code.
- `message=FALSE` to hide the opening messages of some packages.
- `warning=FALSE` to hide warnings.

The default options are declared in the code snippet named "Options" at the beginning of the Markdown document, in the `opts_chunk$set()` function.

The `include=FALSE` option removes any display related to the code snippet. In a document such as a scientific article, which does not display its code, it should be used for all code snippets except those that produce figures.

Figures

```r
plot(pressure)
```

Figures can be created by the R code (figure 4.2). With Bookdown, a label is associated with each figure: its name is `fig:xxx` where `xxx` is the name of the R code snippet. References are made with the command `\@ref(fig:xxx)`.

The header of the code snippet of the figure 4.2 is:

````
```{r pressure, fig.cap="Caption of the figure"}
```
````

It contains at least the name of the figure and its caption. If the caption is long, the header is not very readable. Furthermore, the caption is limited to simple text. For more elaborate captions, it is possible to declare the caption in a separate paragraph that begins with the text `(ref:FigureName)`. Figure 4.3 benefits from an improved caption.

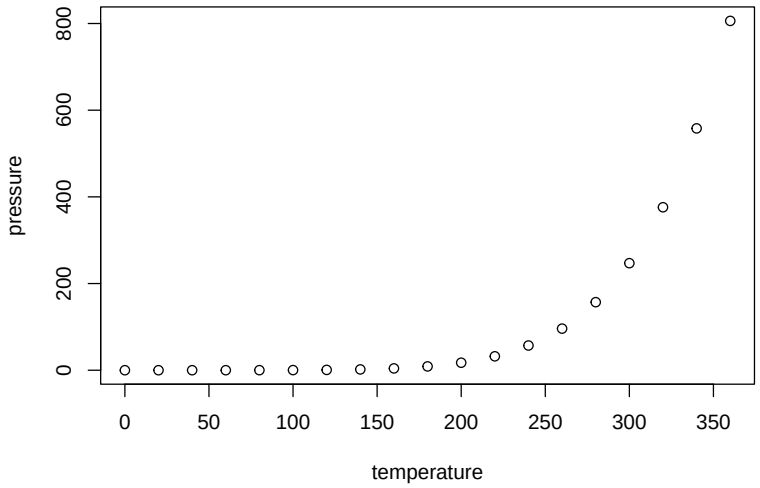


Figure 4.2: Figure Title

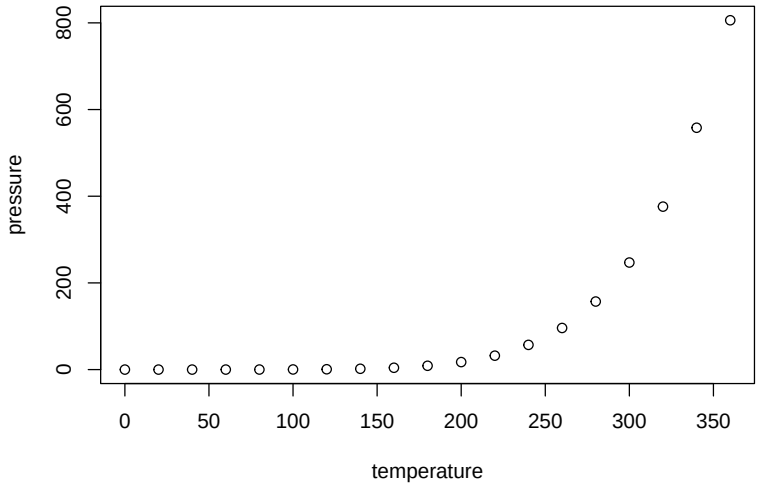


Figure 4.3: Title with *italic*, math ($\sqrt{\pi}$) and a reference to figure 4.2

Table 4.1: Table created by kable

Sepal length (l_s)	Width	Petal length	Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

The text in `fig.cap`, "Title of figure" previously, is replaced by `(ref:pressure)` *within the backquotes* which are retained and the caption is entered in a paragraph starting with `(ref:pressure)` followed by a space. Captions are limited to a single paragraph.

If a table of figures is used (option `lof: true` in the header), a short caption is required in addition to the full caption. It is declared in `fig.scap`.

Figures that are not created by R but come from files are integrated in a piece of code by the `include_graphics()` function whose argument is the file containing the image to be displayed. Always place these files in the `images` folder for good organization.

Tables

The horizontal - and vertical | separators allow to draw a table according to the Markdown syntax, but it is not the best method.

Tables can also be produced by R code. The content of the table is in a dataframe. The `kbl` function in the *kableExtra* package prepares the table for display and passes the result to the `kable_styling` function for final formatting.

```
library("tidyverse")
my_iris <- head(iris)
names(my_iris) <- c("Sepal length ($l_s$)", "Width", "Petal length",
    "Width", "Species")
kableExtra::kbl(my_iris, caption = "Table created by kable",
    booktabs = TRUE, escape = FALSE) %>%
    kableExtra::kable_styling(bootstrap_options = "striped",
        full_width = FALSE)
```

The caption is specified by the `caption` argument and referencing is possible because the table is given a label whose name is `tab:` followed by the name of the code snippet (table 4.1). As with the figures, an enhanced caption can be written in a separate paragraph. A short caption for a possible list of tables (option `lot: true` in the header) is declared in the `caption.short` argument of `kbl()`.

Always use the `booktabs = TRUE` argument so that the thickness of the separator lines is optimal in LaTeX. Since the table contains mathematics (in the name of the first column), the `escape = FALSE` option is necessary.

The `bootstrap_options = "striped"` style option provides more readable tables in HTML. Last, the `full_width = FALSE` option allows to adjust the width of the table to its content instead of occupying the whole available width.

The **flextable** package allows creating more elaborate tables, as in the following example which displays the long sepals in color.

```
library("flextable")
```

```
##
## Attaching package: 'flextable'
```

```
## The following objects are masked from 'package:spatstat.geom':
##
##     border, rotate
```

```
## The following object is masked from 'package:purrr':
##
##     compose
```

```
# iris dataset
iris %>%
    # First lines
head() %>%
    # Create a flextable object
flextable() %>%
    # Column titles
set_header_labels(Sepal.Length = "Sepal Length", Sepal.Width = "Width",
    Petal.Length = "Petal Length", Petal.Width = "Width", Species = "Species") %>%
    # Select long sepals (>5) and display them in red
color(~Sepal.Length > 5, ~Sepal.Length, color = "red")
```

Sepal Length	Width	Petal Length	Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

The package documentation[15] is available online, as well as a gallery[16].

flextable does not support caption numbering except in Word documents. This limitation is prohibitive.

[15] https://ardata-fr.github.io/flextable-book/
[16] https://ardata-fr.github.io/flextable-gallery/gallery/

Maths

Equations in LaTeX format can be inserted inline, like $A = \pi r^2$ (code: `$A=\pi r^2$`) or in a new line (the $ are doubled) like

$$e^{i\pi} = -1.$$

They can be numbered: see equation (4.1), using the `\equation` environment.

$$A = \pi r^2. \tag{4.1}$$

The numbered equation is created by the following code:

```
\begin{equation}
  A = \pi r^2.
  \label{eq:disc}
\end{equation}
```

Cross-references

Figures and tables have an automatically generated label, identical to the name of the code snippet prefixed with `fig:` and `tab:`.

For equations, the label is added manually by the code `(\#eq:xxx)` before the end of the equation.

Sections can be given a label by completing their title with `{#yyy}`. Sections receive by default an implicit label[17] corresponding to their text, in lower case, where special characters are replaced by dashes. Implicit labels are unstable (they change with the title of the section) and difficult to predict: this is why it is advisable to add an explicit label to each section being cross-referenced. This is the case for chapters, for which the name of the HMTL file produced is identical to the label. Chapter labels must follow file naming rules by not containing special characters.

Bookmarks can also be placed freely in the text with the command `(ref:zzz)`.

In all cases, the call to the reference is made by the command `\@ref(ref:zzz)`.

Bibliography

Bibliographic references in BibTeX format must be included in the `.bib` file declared in the header of the Markdown document.

```
bibliography: references.bib
```

[17] https://pandoc.org/MANUAL.html#extension-implicit_header_references

This file can be created and maintained by Zotero installed with the Better BibTeX extension (see section 1.6). To do this, you just have to create a Zotero collection corresponding to the project and drag the relevant references into it. Then use the contextual menu "Export collection…" and select:

- Format: "Better BibTeX" for articles and presentations or "Better BibLaTeX" for memoirs, depending on whether the bibliography is managed by BibTeX and natbib or biber and BibLaTeX for PDF production.
- Check the "Keep up to date" box so that any changes in Zotero are exported automatically.
- Click on "OK" then choose the name of the file (`references.bib`) and its location (the R project folder).

The references can be called in the text, between square brackets by the code `[@Reference]`, or in the text, by removing the brackets.

The bibliography is handled by pandoc when producing Word or HTML documents. The bibliographic style can be specified, by adding the line

```
csl:nom_du_fichier.csl
```

in the document header and copying the *.csl* style file to the project folder. Over a thousand styles are available[18].

For PDF documents, the bibliography is managed by LaTeX.

To prepare the submission of a manuscript to a journal, it will be necessary to open the intermediate *.tex* file produced by pandoc and copy the contents of the environment {document} into the template proposed by the journal, which will take care of the formatting.

Languages

The languages are to be declared in the header of the documents produced by the **memoiR** templates.

The main language of the document modifies the name of certain elements, such as the table of contents. The additional languages allow the creation of multilingual documents.

The header fields are:

```
lang: fr-FR
otherlangs: [en-US, it]
```

The change of language in the document is managed in LaTeX, but not in HTML, by inserting on a new line the following command:

```
\selectlanguage{english}
```

[18] https://github.com/citation-style-language/styles

The current language only has an effect in LaTeX output: a space is added before double punctuation in French, the size of spaces is larger at the beginning of sentences in English, etc. The `\selectlanguage` command is simply ignored in HTML.

The language names are different in the header (IETF codes) and in the text (language name). The correspondence and the complete list of languages can be found in table 3 of the package documentation **polyglossia**[19].

HTML formatting of punctuation in French documents is possible using a filter declared in pandoc [20]. The `fr-nbsp.lua` file must be copied into the project directory from its GitHub repository and declared into the header of the Markdown document.

```
output:
    pandoc_args:
      --lua-filter=en-nbsp.lua
```

The filter formats all the punctuation in the document, whatever the language: it should therefore only be used for documents written entirely in French.

4.3.2 Simple Article template

The *Simple Article* template of **memoiR** produces a simple HTML document with a floating table of contents (see example[21]). Other HTML formats are available: see the gallery[22] of the package. The PDF format is close to the *article* model of LaTeX (example[23]).

The template contains its own documentation.

Create

Use the menu "File > New File > R Markdown…" then select "From template" (figure 4.1). The list of available templates and the package that offers them is then displayed.

Select the *Simple Article* template from the **memoiR** package, choose the name of the project ("Name:", which will be the name of the folder in which it will be created, and its parent folder ("Location:"). In the organization proposed in section 1.2.4, the parent folder is `%LOCALAPPDATA%\ProjectsR`. The project name must not contain any special characters (accent, space…) to ensure its portability on all operating systems (Windows, Linux, MacOS).

Advanced templates create a folder with many files (bibliography, styles, LaTeX template…), unlike simple templates which create only one file.

[19] http://mirrors.ctan.org/macros/unicodetex/latex/polyglossia/polyglossia.pdf
[20] https://github.com/InseeFrLab/pandoc-filter-fr-nbsp
[21] https://EricMarcon.github.io/Krigeage/Krigeage.html
[22] https://ericmarcon.github.io/memoiR/
[23] https://EricMarcon.github.io/Krigeage/Krigeage.pdf

When a folder is created, for example by the *Simple Article* template, you have to make it an RStudio project: in the projects menu (top right of the RStudio window), use the "New Project…" menu, then "Existing Directory" and select the folder that has just been created.

Write

The instructions for using the template are contained in the text provided by default.

Knit

The document can be knitted in several formats:

- *html_document2* is the HTML format the template was designed for: a notepad with a floating table of contents.
- *gitbook* is an alternative HTML format, normally used for books.
- *downcute* is an HTML format provided by the **rmdformats** package.
- *pdf_book* produces a PDF document following the LaTeX *article* template, commonly used directly in LaTeX.
- *word_document2* creates a Word file.

Publish

The **memoiR** package simplifies the uploading of produced documents to a web server.

The `build_gitignore()` function creates a `.gitignore` file for source control which must be enabled (see section 3.1.1).

The `build_readme()` function creates a `README.md` file that is needed by GitHub. It contains the title of the project, its summary and links to the HTML and PDF versions of the documents produced.

The project must be linked to a GitHub repository (section 3.2).

Two publication strategies are possible. In the first one, the documents are knitted locally and placed in the `docs` folder, which will be the support of the GitHub pages. In the second one, the documents are knitted by GitHub Actions each time modifications are pushed on the repository: this is called continuous integration (section 6).

The local production strategy is covered here; continuous integration will be covered in section 6.3.1.

The `build_githubpages()` function places all the knitted documents (HTML and PDF) in the `docs` folder, along with a copy of the `README.md` file. This way, it is possible to activate the project's GitHub pages (on the `docs` folder of the `master` branch). The `README.md` file will be the home page of the produced web site.

In practice, we knit in HTML format during the whole writing phase, because the production is very fast. When the document is stabilized, it should be knitted in HTML and PDF format. Finally, the execution of `build_githubpages()` places all the files produced in `docs`. It remains to push the repository on GitHub and activate the GitHub pages.

4.3.3 Other templates

The *Stylish Article* template of **memoiR** is intended for the production of well-formatted PDF articles for self-archiving (typically, the HAL repository), in A4 format in double column[24].

The HTML format is the same as the *Simple Article* template.

The **rticles** package aims to provide templates for all scientific journals that accept article submission in LaTeX. It offers Markdown templates which produce PDF files conforming to the requirements of the journals and the possibility to recover the intermediate `.tex` file (pandoc produces a `.tex` file transmitted to the LaTeX compiler). The package does not allow HTML knitting because it uses LaTeX syntax in the R Markdown document instead of using **bookdown** to handle bibliographic and cross references. It is not possible to directly exchange standard R Markdown content with documents written for **rticles**, which limits the interest of the package.

4.4 Beamer Presentation

The *Beamer Presentation* template of **memoiR** allows to create HTML and PDF (beamer) presentations simultaneously, as shown in the example[25].

The approach is identical to that of articles in the same package. The title levels allow separating the parts of the presentation (`#`) and the slides (`##`). Two formats are available in HTML: ioslides[26] and Slidy[27]. Some specificities in the code allow to refine the presentation of the slides, for a two-column display for example: they are documented in the template.

4.5 memoir

The *Memoir* template of the **memoiR** package is intended for long documents, which have an important difference from the previous documents: a long document is composed of several chapters, each placed in its `.Rmd` file.

[24] Exemple: https://EricMarcon.github.io/Rochebrune2018/Entropie.pdf

[25] https://EricMarcon.github.io/Chao1/, Choose "Lecture" (Read in HTML) ou "Téléchargement" (Download PDF).

[26] https://bookdown.org/yihui/rmarkdown/ioslides-presentation.html

[27] https://bookdown.org/yihui/rmarkdown/slidy-presentation.html

The HTML format is gitbook[28], the standard for reading such documents online. The PDF format is derived from the LaTeX *memoir*[29] template, also optimized for long documents.

This document was written with this template.

4.5.1 Create

Creating a work project is identical to the one presented above: the template is: *Memoir*. The created folder must be turned into a project.

Run `build_git()` and `build_readme()`, enable source control and push the project to GitHub, in the same way as for an article (section 4.3.2).

Each chapter of the book is an Rmd file, whose name normally starts with its number (e.g.: `01-intro.Rmd`). All Rmd files in the project folder are actually treated as chapters, sorted by filename, including those provided by the template (startup and syntax) which should be deleted except for `99-references.Rmd` which contains the bibliography, placed at the end. The `index.Rmd` file is special: it contains the document header and the first chapter.

4.5.2 Write

The first chapter is placed in the front matter of the printed book: it should not be numbered (hence the `{-}` code next to the title) in the HTML version. It must end with the LaTeX command `\mainmatter` which marks the beginning of the body of the book.

The outline levels start with # for chapters (only one per file), ## for sections, etc.

4.5.3 Knit

Compiling to PDF is done by XeLaTeX, which must be installed.

While writing, it is strongly advised to create only the HTML file, which is much faster than a LaTeX compilation. Each chapter can be viewed very quickly by clicking on the "Knit" button above the source window. The entire book is created by clicking on the "Build Book" button in the RStudio *Build* window. The button's drop-down list allows you to create all documents or limit yourself to one format.

The files produced are placed directly in the `docs` folder, which will be used by the GitHub pages to allow online reading and downloading of the PDF. The home page of the website is created by bookdown from the `index.Rmd` file: the `README.md` file is not duplicated in `docs`.

[28] https://www.gitbook.com/
[29] https://www.ctan.org/pkg/memoir

4.5.4 Finishing

The layout is done fully automatically by pandoc (in HTML) and LaTeX (in PDF).

It is often useful to help LaTeX to solve some margin overruns due to too large layout constraints: for optimal readability, columns are narrow, but code (formatted text between backquotes) does not allow hyphenation.

If a line of text protrudes into the right margin in the PDF document, the solution is to manually add the `\break` code to the desired location for the line break in the R Markdown document. The command has no effect on the HTML document but forces the hyphenation in LaTeX. To break formatted text (between asterisks for italics or more frequently between backquotes for code), you must finish formatting before `\break` and start again afterwards. For example, to force a line break before `file.Rmd`:

```
The file `/path/`\break`file.Rmd`
```

In HTML, a space will be added between the two pieces of code.

R code snippets are automatically formatted by **knitr** when the `tidy=TRUE` option is applied to them. The default behavior is specified in the **knitr** options, in a code snippet at the beginning of the `index.Rmd` file:

```
# knitr options
knitr::opts_chunk$set(
  cache=TRUE, warning=FALSE, echo = TRUE,
  fig.env='SCfigure', fig.asp=.75,
  fig.align='center', out.width='80%',
  tidy=TRUE,
  tidy.opts=list(blank=FALSE, width.cutoff=55),
    size="scriptsize",
  knitr.graphics.auto_pdf = TRUE)
```

The maximum width of a line of formatted code here is 55 characters, optimal for the template. Sometimes automatic formatting does not work because **knitr** cannot find a line break that meets all the constraints, causing the code to overflow. In this case, manually format the code snippet by adding the `tidy=FALSE` option.

The literal code blocks, delimited by three backquotes, must be formatted manually, avoiding any line longer than 55 characters.

4.5.5 Gitbook site

The website containing the gitbook document must be set up in `_output.yml` so that :

- The title of the document appears at the top of the table of contents.
- An indication of the use of GitHub and bookdown is displayed at the bottom of the table of contents.

- A GitHub button in the title bar allows to open the project repository.
- Another button allows to download the PDF document.

The `_output.yml` file of this document is the following:

```
bookdown::gitbook:
  css: style.css
  config:
    sharing:
      github: yes
      facebook: false
      twitter: false
    toc:
      before: |
        <li><a href="./">Working with R</a></li>
      after: |
        <li>
          <a href="https://github.com/EricMarcon/WorkingWithR" target="blank">
            Hosted on GitHub, published by bookdown
          </a>
        </li>
    download: pdf
```

The `sharing:` section manages the buttons in the title bar. By default, the links to Facebook and Twitter are enabled but the one to GitHub is not. For it to work, the GitHub repository must be declared in the header of the `index.rmd` file:

```
github-repo: EricMarcon/WorkingWithR
```

The `toc:` section contains two portions of HTML code in which the title of the document and the link to its GitHub repository must be adapted to the project.

Finally, the `download:` section lists the downloadable document formats and displays a download button in the title bar.

4.5.6 Continuous integration

Building a book takes time, especially if it contains calculations. It must be launched in gitbook format and in PDF format. In production, it can be outsourced to GitHub (chapter 6.3.1).

4.5.7 Google Analytics

The tracking of the book's audience can be entrusted to Google Analytics. To do so, you have to create an account and add a Google Analytics *property*, i.e. a website, then a data feed, here a web feed[30].

Google Analytics provides a configuration script named `gtag.js` to be placed at the root of the project folder. Finally, declare the script in the header of the web pages by adding a statement in `_output.yml`, in its first section.

[30] https://support.google.com/analytics/answer/9304153?hl=fr&ref_topic=9303319

```
bookdown::gitbook:
  includes:
    in_header: gtag.js
```

4.6 R Markdown web site

A web site made of pages written with R Markdown (without the **bookdown** features) and a menu can be created very easily, with a good result[31].

4.6.1 Template

In RStudio, in the projects menu at the top right, click on "New Project…" then "New Directory" then "Simple R Markdown website". Enter the name of the project, select the folder in which the project will be created by clicking on "Browse" and finally click on "Create Project".

The default site contains two pages: `index`, the home page, and `about`, the "About" page. The `_site.yml` file contains the name of the site and the contents of its navigation bar: a title and the corresponding file. Other pages will be added by creating new `.Rmd` files and adding them to the `_site.yml` file.

4.6.2 Improvements

The site template can easily be enhanced by adding lines to `_site.yml`:

- To add a GitHub icon in the navigation bar to link to the site source code.
- To choose the method of knitting, to use **bookdown** instead of **rmarkdown**.
- To place the site files in the `docs` folder and thus separate code and production.

The completed `_site.yml` file is as follows:

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "About"
      href: about.html
  right:
    - icon: fa-github
      href: https://github.com/rstudio/rmarkdown
output_dir: "docs"
output:
  bookdown::html_document2:
    theme: sandstone
    highlight: tango
    toc: true
    toc_float: yes
```

[31] https://rstudio.github.io/learnr/ for example.

The GitHub icon is part of the Font Awesome collection of which all free icons[32] are usable with the same syntax: "fa-name".

The link corresponding to the icon must be the one in the website's GitHub repository.

The syntax of the `output` section is the same as that of the documents seen above. It applies to all pages (with the YAML header reduced to the minimum). The available themes are those of rmarkdown[33].

The `highlight` option specifies how any R code displayed will be formatted. Last, the table of contents is floating, which means that its position adjusts as the window scrolls.

4.6.3 Source control

The project must be put under source control and pushed to GitHub (chapter 3). The `.gitignore` file is the following:

```
# R
.Rbuildignore
.RData
.Rhistory
.Rprofile
.Rproj.user

# Web Site
/_site/
/*_cache/
/*_files/
```

Enable GitHub pages (section 3.7) on the `docs` folder to host the site. Add an empty file named `.nojekyll` in `docs` so that GitHub pages won't try to reformat the site. You can use the RStudio terminal to run:

```
touch docs/.nojekyll
```

4.7 Personal web site: blogdown

To create a personal web page, *Hugo* is a static site generator capable of producing HTML pages from Markdown code. Static sites have the advantage, compared to dynamic sites managed by a content management system (CMS, for example: Wordpress, Joomla, SPIP), to be portable on any web server without database support or code to run on the server side (such as PHP) and to be very fast since the pages are created once and not at each consultation. A Hugo site can be hosted for example on the personal page of any GitHub user. Its address is of the form "GitHubID.github.io".

Hugo offers many themes, which are templates for site structure, so the **Academic** theme, intended for researchers. In RStudio, the **blogdown** package is

[32] https://fontawesome.com/icons?d=gallery&m=free

[33] https://bookdown.org/yihui/rmarkdown/html-document.html#appearance-and-style

provided to easily produce web pages with Hugo. These pages can contain R code: they are very similar to an article, seen above, whose content can be easily copied and pasted. So we will use this solution, for a site like the one proposed in example[34].

The structure of the website is simple:

- A home page, containing various customizable components such as the author's biography, a selection of publications, blog posts or other elements and a contact form.
- Pages detailing the various elements (publications, posts, etc.) written in R Markdown.

4.7.1 Installing the tools

The first step is to install the **blogdown** package in R.

```
install.packages("blogdown")
```

blogdown is able to install Hugo on Windows, macOS or Linux.

```
blogdown::install_hugo()
```

The full documentation for **blogdown** is available[35].

Recent versions of Hugo use *Go* (the programming language) to install their modules on the fly: here the Academic theme is loaded from GitHub at the time of site creation. Go must therefore be installed[36].

4.7.2 Create

The easiest way is to create a repository on GitHub from the template. On the *starter-academic*[37] repository page, click on the "Use this template" button, optionally authenticate to GitHub, and then enter the name of the repository that will contain the project, for example "MySite".

The repository can be the one of the main site of your GitHub account (see section 3.7), at the address https://GitHubID.github.io[38]. The name to enter is simply "GitHubID.github.io" (*GitHubID* is the name of the GitHub account).

Create the repository. Copy the repository address by clicking on the "Code" button and then on the button to the right of the address (figure 4.4).

In RStudio, create a new project from GitHub: in the projects menu on the top right, click on "New Project…" then "Version Control" then "Git" then paste

[34] https://EricMarcon.github.io/
[35] https://bookdown.org/yihui/blogdown/
[36] https://golang.org/doc/install
[37] https://github.com/wowchemy/starter-academic
[38] Exemple: https://EricMarcon.github.io/Krigeage/

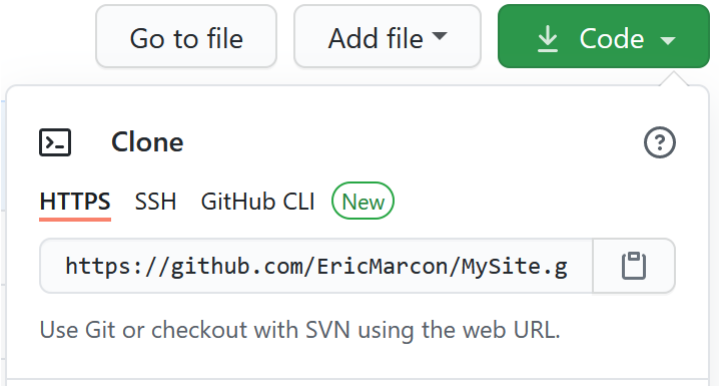


Figure 4.4: Copy of the address of a repository to clone on GitHub.

the address in the "Repository URL" field (figure 4.5). Select the folder in which the project will be created by clicking on "Browse" and finally click on "Create Project".

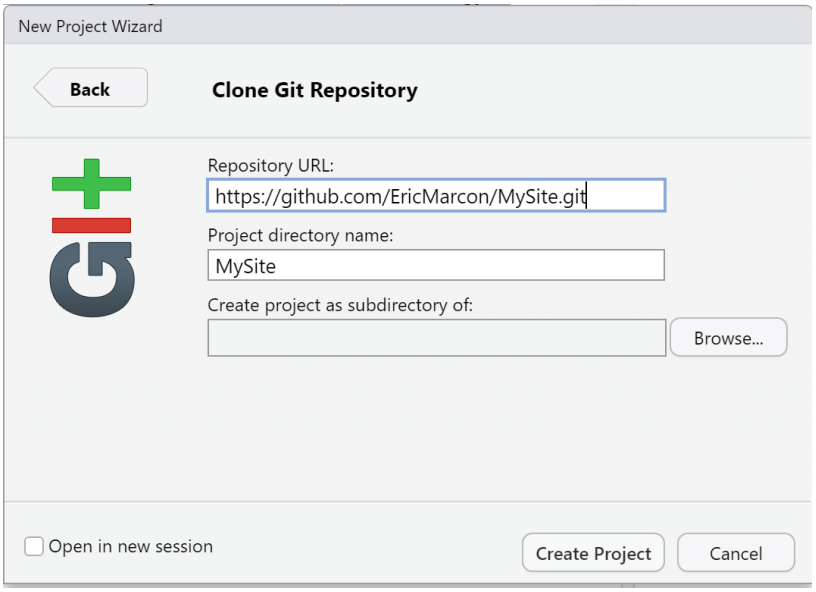


Figure 4.5: Copy the address of a repository to clone to GitHub.

The created project is an exact copy of the template, which must be customized.

RStudio automatically adds a line at the end of the `.gitignore` file to ignore its working files (`.Rproj.user` folder). Add a comment line to indicate this. The content of `.gitignore` should be as follows:

```
# R
.Rbuildignore
.RData
.Rhistory
.Rprofile
```

```
.Rproj.user

# Hugo
/resources/
/public/

# blogdown
/static/en/
/static/fr/
*.rmarkdown
_index.html
index.html
**/index_files/
```

A bug of **blogdown** requires to move the file `config.toml` from the folder `config/_default/` to the root of the project.

Take into account these modifications in git by making a commit.

4.7.3 Building the site

Run

```
blogdown::build_site(build_rmd = TRUE)
```

to build the website, including its future R Markdown pages.

To display the site, run :

```
blogdown:::serve_site()
```

It appears in the RStudio *Viewer* window, which can be viewed in the system's default web browser by clicking the enlarge button.

To modify the content of the site, it is best to stop the web server with the command:

```
blogdown:::stop_server()
```

The site produced by **blogdown** is located in the `public` folder which can be copied directly to a web server that will host it. A simple solution is to declare this folder as the root of the GitHub pages of the project (section 3.7). The optimal method is to use continuous integration (see section 6.3.2) to copy it to the root of the `gh-pages` branch which will be declared as the location of the site on GitHub.

4.7.4 Multilingual site

If the site is multilingual (say English and French), its `content` folder must be copied in a folder corresponding to each language. The files in `content/authors/admin/` must be duplicated into `content/en/authors/admin/` and `content/fr/authors/admin/`. In practice, create an `en` folder and an `fr` folder in `content`. Move all the original `content` folder `en` and then copy it into `fr`.

4.7.5 Set up

The site configuration files are well documented and offer many options. The main ones are reviewed here for a quick creation of a working site.

The `config.toml` file contains the general parameters of the site. The lines to be updated are the site title (the owner's name since it is a personal site) and its public address. For the example site:

```
title = "Eric Marcon"
baseurl = "https://EricMarcon.github.io/"
```

It also contains the default language selection line ("en" or "fr") and the line that allows to place the files produced by Hugo in each language folder ("true" mandatory for a multilingual site):

```
defaultContentLanguage = "en"
defaultContentLanguageInSubdir = true
```

The `config/_default/` folder contains the other configuration files.

The `languages.toml` folder contains the language settings and menu translations. For each language, the version used and the content folder are specified:

```
[en]
  languageCode = "en-us"
  contentDir = "content/en"
[fr]
  languageCode = "fr-fr"
  contentDir = "content/fr"
```

For additional languages, the site title, date display settings and menu translation are added. In the section `[fr]`:

```
[fr]
  languageCode = "fr-fr"
  contentDir = "content/fr"
  title = "Eric Marcon"
  description = "Page personnelle d'Eric Marcon"
  [fr.params]
    description = ""
    date_format = "02-Jan-2006"
    time_format = "15:04"
  [[fr.menu.main]]
    name = "Accueil"
    url = "#about"
    weight = 20
(...)
```

These lines are commented out in the template and must therefore be uncommented by removing the # at the head of the line.

The menus are described below.

`params.toml` describes the look of the site. The options are grouped by topic, for example "Theme" for the general appearance. In "Basic Info", the line

```
site_type = "Person"
```

selects a personal site. It is possible to use Academic for a scientific project site or a unit site, not documented in detail here. The main differences are, for a collective site:

- The management of authors: in the `/contents/<language>/outside` folder, only one `admin` folder is used for a personal site, whereas one folder per person is needed for a collective site.
- A component described below, which allows to present the persons, must be activated.

The description of the site in the default language is entered, for search engines:

```
description = "Eric Marcon's Homepage"
```

It must be translated in the file `languages.toml`, in each language.

In "Site Features", we select the coloring of the R code, the activation of the formatting of equations and the legal warning for the use of cookies.

```
highlight_languages = ["r"]
math = true
privacy_pack = true
```

The `edit_page` line needs to be updated: replace the default repository "https://github.com/gcushen/hugo-academic" with that of the site.

"Contact details" contains the contact information for the site owner. They must be entered.

"Regional Settings" contains the date display settings for the default language (those for other languages are in `languages.toml`). They usually do not need to be changed.

"Comments" allows you to enable visitor comments at the bottom of pages, with Disqus or Comment.io (an account is required with the provider). "Marketing" allows you to activate the tracking of site traffic by simply entering your Google Analytics ID (to be created with a Google account). "Content Management System" contains the line `netlify_cms` whose value must be `false` if the site is not hosted by Netlify. Finally "Icon Pack Extensions" allows you to activate Academicons icons if necessary.

4.7.6 Write

Use the online documentation[39] to complement the main information detailed here. The example used here is the author's personal site[40].

[39] https://wowchemy.com/docs/page-builder/
[40] https://EricMarcon.github.io/

The working method is to progress step by step by testing and then to validate each step:

- Make the changes.
- Build the site and check the result: `blogdown:::serve_site()`.
- Stop the site: `blogdown::stop_server()`.
- If the result is not satisfactory, try again.
- Commit the changes.

Home page

The home page of the site is made up of a series of elements (*widgets*) that are located in `/contents/<language>/home`. Each element is described by a markdown file. The first one is `index.md`. It is usually not modified. Its contents are as follows:

```
+++
# Homepage
type = "widget_page"
headless = true  # Homepage is headless, other widget pages are not.
+++
```

The file contains only a TOML header, surrounded by a line of `+++`. The component type indicates that this is a component page, in which the other components in the file will fit. `headless = true` means that the page has no header.

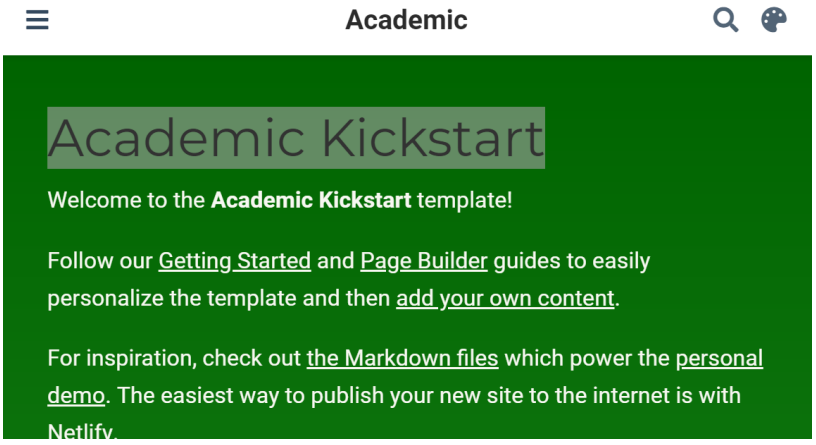


Figure 4.6: Component `demo` in Academic.

The `demo.md` component (figure 4.6) is a "blank" component, i.e. a free text page: it is used here to present the Academic Kickstart template and must therefore be disabled. The header contains its formatting information (title, number of columns, colors…) and the content of the page is written in markdown. The components appear in ascending order of weight in the header: 15 marks the first component in the Academic template. The component can be disabled by deleting its file or by changing its `active` property in the header:

```
active = false  # Activate this widget? true/false
```

Figure 4.7: The `about` component in Academic.

The next component is `about.md` (figure 4.7). It presents the owner of the site. Its title must be localized. In the '/content/en/home' folder, its value will be:

```
title = "Biography"
```

The author must correspond to a folder in `/contents/<language>/authors`. The `admin` is fine for a personal site. Academic allows for team sites: in this configuration, one folder per person would be needed. The image displayed by the component is the `avatar.jpg` file placed in this folder. Limit the size of the file for the performance of the site (less than a megabyte is a reasonable size), while ensuring a minimum size of a few hundred pixels per side for display quality.

The content of the component is read from the `_index.md` file in the same folder, which contains all information about the author. Its organization is quite clear: modify its content from the example provided. If `ai` type icons are used, enable the Academicons icon package in `config/_default/params.toml`.

The skills component (figure 4.8) presents the author's skills graphically. A collection of icons is available, and new icons can be added.

The `accomplishments` component presents the professional trainings and allows to access their certificates.

The `posts` component fetches its content from the folder which contains the blog posts, i.e. `/contents/<language>/post` (see below). The `posts.md` file contains layout options in its header.

The `projects` component works the same way. The difference between the two components is their formatting: `posts` is of the "pages" type, which displays the most recent items, while `projects` is of the "portfolio" type, which displays selected items that contain the description `featured: true` in their

Figure 4.8: The `skills` component in Academic.

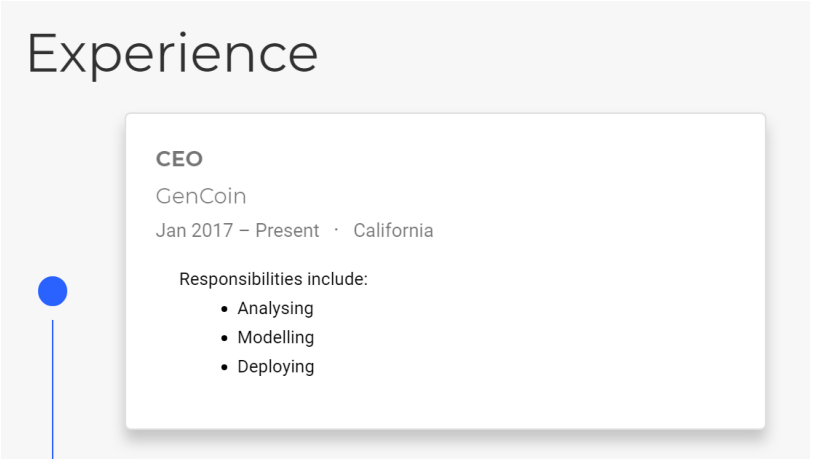


Figure 4.9: The `experience` component in Academic.

own header. It is possible to create components of these types freely, by specifying the folder containing the items in `page-type`. Example: create a component named `software.md` by renaming `projects.md`, change its `page_type = "software"` line and create a `/contents/<language>/software` folder to place content in.

The `publications` and `featured` components are of type `pages` and `portfolio` respectively and take their content from the `publication` folder.

The `tags` component presents a word cloud from the keywords declared in all content files (blog posts, publications…) in the following form in their header:

```
tags = ["Keyword 1", "Other Keyword"]
```

Last, the `contact` component allows to display a contact form. It uses the information from the `config/_default/params.toml` file in its part starting with:

```
############################
## Contact details
##
```

To display a map, enter the latitude and longitude of the address in the `coordinates` line. To display a mail form, choose the *formspree.io* service (`email_form = 2` in `contact.md`). To activate the mail service, you will have to build the web site, send yourself a first message using the form and follow the instructions of Formspree.

The `people` component is used in group sites to present the members. The `slider` component is used to display a carousel (scrolling elements) at the top of the page. To understand how it works, the easiest way is to activate it.

Home page menu

The home page has a menu that allows you to navigate quickly to its components or to other pages. It is set up in `config/_default/menus.toml`. Menu items have a displayed name, a link (starting with # to point to a component or a relative path in the site such as `publication/`), and a weight that defines their display order, similar to the order of the components on the home page.

A two-element menu for pointing to the site's home page and blog posts is thus as follows:

```
[[main]]
  name = "Home"
  url = "#about"
  weight = 10

[[main]]
  name = "Posts"
  url = "#posts"
  weight = 20
```

In the file `config/_default/languages.toml`, the menu must be translated into each language :

```
[fr]
  [[fr.menu.main]]
    name = "Accueil"
    url = "#about"
    weight = 10
  [[fr.menu.main]]
    name = "Articles"
    url = "#posts"
    weight = 20
```

Posts

The site is powered by blog posts placed in the `/contents/<language>/post` folder. They must be translated and placed in the `post` folder of each language to be available in the corresponding language. The example used here is a guide to correctly estimating the density of a bounded variable [41].

Its code is on GitHub[42].

[41] https://EricMarcon.github.io/post/densite/

[42] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/post/densite

A post is placed in a folder (`/content/en/post/densite`) that contains its Markdown R code and possibly images, data to feed the code and other elements called by the code. Hugo supports native markdown files. The contribution of **blogdown** relative to a native Hugo site is the support of R Markdown, thus the possibility of executing any R code as in a notepad (whose content can be reused without modification).

The main file of a post is `index.Rmd`. **blogdown** creates an `index.html` file during the construction of the site: it can be ignored (in `.gitgnore`) and deleted at any time. If a `featured.png` (optimal for a graphic) or `featured.jpg` (optimal for a photo) image is placed in the folder, it will be used as the thumbnail of the post.

The `index.Rmd` includes a header in yaml (surrounded by `---`) or toml (surrounded by `+++`) format that describes its display:

```
---
title: "Title of the post"
subtitle: "Subtitle"
summary: "Summary"
authors: []
tags: ["Keyword 1", "Other Keyword"]
categories: []
date: 2020-04-17
featured: false
draft: false

# Featured image
# To use, add an image named `featured.jpg/png` to
# your page's folder.
# Focal points: Smart, Center, TopLeft, Top, TopRight,
# Left, Right, BottomLeft, Bottom, BottomRight.
image:
  caption: ""
  focal_point: ""
  preview_only: false

bibliography: references.bib
---
```

Authors are used in collective sites. Tags are used to feed the word cloud component if it is activated in the home page. Categories are used to search for pages with similar content (keyword search on the site). The `featured: true` option makes the post appear in the `featured` components on the home page. The `draft: true` option hides the post.

The following elements specify the display of the thumbnail: caption and position. The `preview_only: true` option limits the display to thumbnails (on the home page), thus removing the image from the post itself.

The header elements needed for the Markdown body text, such as the name of the file containing the bibliographic references, placed in the same folder, are added.

The body text is that of a standard R Markdown document, with R code included. A piece of initial code allows to set the R options and load the necessary packages.

In practice, the most efficient way to create a new post is to copy the entire folder of a previous post, rename it and modify its contents. The `blogdown::new_post()` command can also be used, but it does not handle multiple languages (and so creates the post in the `/contents/post` folder unless you specify the `subdir` argument).

Rebuilding the site does not by default update pages based on a `.Rmd` file. To do this, you must force the `build_site()` command.

```
blogdown::build_site(build_rmd = TRUE)
blogdown::serve_site()
```

Publications

Publications are organized like posts, but placed in the `/contents/<language>/publications` folder.

The example used is a journal article[43] with its code[44].

A `cite.bib` file containing the reference in BibTeX format is placed in the folder. The name of the folder is preferably that of the publication identifier. The header of the `index.md` file (here in Markdown format, but `.Rmd` is possible if R code is needed) contains the same information as the BibTex file, but in the appropriate format (yaml), and the Academic-specific elements (`featured`):

```
---
title: "Evaluating the geographic concentration of |>
industries using distance-based methods"
authors: ["Eric Marcon", "Florence Puech"]
publication_types: ["2"]
abstract: "We propose (...)"
publication: "*Journal of Economic Geography*"
doi: "10.1093/jeg/lbg016"

date: 2003-10-01
featured: false
---
```

The publication types are:

- 0 = Uncategorized.
- 1 = Conference paper.
- 2 = Journal article.
- 3 = Preprint / Working Paper.
- 4 = Report.
- 5 = Book.
- 6 = Book section.
- 7 = Thesis.
- 8 = Patent.

[43] https://EricMarcon.github.io/publication/marcon-2003-a/

[44] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/publication/marcon-2003-a

Buttons are displayed at the top of the publication page depending on the information found:

- PDF: if the `url` line is present in the header.
- Citation: if the file `cite.bib` is present in the folder.
- DOI: if the line `doi` is present in the header.

The body of the publication contains a link (in HTML format) to the Dimension site which provides bibliometric information. This link can be reused very simply, by simply replacing the DOI of the document:

```
<span class="__dimensions_badge_embed__"
  data-doi="10.1093/jeg/lbg016"></span>
<script async src="https://badge.dimensions.ai/
  badge.js" charset="utf-8"></script>
```

Finally, a `/contents/<language>/publications/_index.Rmd` file is used to present the complete bibliography. It is accessible from the `publications` component of the home page, which displays a `More Publications` link.

The example file[45] with its code[46] allows to query Google Scholar to obtain the co-author network, the h-index and the number of annual citations of the author. It can be reused by simply changing the Google Scholar ID on line 30.

By having the code run regularly, for example through GitHub (see below), the displayed statistics are kept up to date without human intervention.

Communications

Communications are organized like publications, in the `/contents/<language>/talk` folder.

The example used is a communication in French, so in `/contents/en/talk`[47] with its code[48].

An image can be used more easily than for a publication.

The header contains special lines suitable for communications:

```
---
title: "Construction of the Chao1 biodiversity estimator"
event: "Mathematics Week 2020"
event_url: https://eduscol.education.fr/cid59178/|>
semaine-des-mathematiques.html

location: University of French Guiana

summary: []
```

[45] https://EricMarcon.github.io/publication/

[46] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/publication/marcon-2003-a

[47] https://EricMarcon.github.io/talk/chao1/

[48] https://github.com/EricMarcon/HomePage2020/tree/master/content/fr/talk/chao1

```
abstract: |
  To estimate the number of species (species richness) of a community
  of a community from a sample, the Chao1 estimator is the sample,
  the Chao1 estimator is the most commonly used tool.

  Its construction is explained and its efficiency is tested on
  is tested on simulated data.

# Talk start and end times.
#   End time can optionally be hidden by
# prefixing the line with `#`.
date: "2020-03-11T11:00:00Z"
date_end: "2020-03-11T12:00:00Z"
all_day: false

# Schedule page publish date (NOT talk date).
publishDate: "2020-04-14"

# Is this a featured talk? (true/false)
featured: false

image:
  caption: 'Produit scalaire des vecteurs $v_0$ |>
et $v_2$'
  focal_point: Smart

url_code: "https://github.com/EricMarcon/Chao1"
url_pdf: "https://EricMarcon.github.io/Chao1/|>
Chao1.pdf"
url_slides: "https://EricMarcon.github.io/Chao1/|>
Chao1.html"

# Enable math on this page?
math: true
---
```

The links (e.g. `url_code`) bring up buttons that display the source code of the presentation, a PDF file and the online slides respectively.

Other elements

It is possible to freely add additional elements to the site:

- In `/contents/<language>/`, create a folder whose name is the type of elements (example: `recipe`).
- Add items to this folder, each in its own folder.
- The mandatory file is `index.md` or `index.Rmd` with a header possibly containing all the fields found in `post`, `publication` and `talk` items.
- The thumbnail file, `featured.png` or `featured.jpg`, is optional.
- All files needed for knitting (images, data) can be added in the same folder.
- In `/contents/<language>/home`, add a home page component by copying and pasting an existing "pages" (like `publications`) or "portfolio" (like `featured`) element and set it to point to the right folder (in the example: `page-type=recipe`) and adjust its appearance (number of elements for example) and its position (weight).
- Optionally add a menu entry to point to the component, with the same weight as the component.

The index files can have the extension `.Rmd` or `.md`. In the first case, they will be processed by **blogdown**, which supports R code integration. In the other case, they will be processed by Hugo, which only supports the standard markdown format. The `.md` files require less resources and are therefore preferred when they are sufficient.

Polishing

The site icon, which appears in the address bar of web browsers, is located in `assets/images`. The `icon.png` file can be replaced.

4.7.7 Continuous integration

The construction of the website in production can be entrusted to GitHub (section 6.3.2), including its periodic update if pages of the site deal with data that evolve over time.

4.7.8 Updates

The Academic theme is updated regularly. The version used is indicated in the `go.mod` file. To use the latest official version, run the following command in the R console:

```
blogdown::hugo_cmd("mod get -u")
```

The `go.mod` and `go.sum` files, which contain the hash codes of the module files, are updated.

Each version change may require adaptations to the site content, referenced in the online documentation of the theme[49].

Update Hugo at the same time:

```
blogdown::update_hugo()
```

4.8 Exporting figures

When document production with R Markdown is not possible, figures from R must be exported as files to be integrated into another writing process. It is best to create scripts to create the figures in a reproducible way and in the optimal format.

4.8.1 Vector and Raster Formats

Figures should generally be produced in a vector format:

[49] https://wowchemy.com/updates/

- SVG for poster publication or posters.
- EMF (Extended Meta-File) for Word or the Microsoft Office suite that does not support other formats.
- EPS (Encapsulated PostScript) or PDF (Portable Document Format) for LaTeX.

Raster figures (composed of a set of points, like photographs) are rare in R. The `image()` function used to display maps uses polygons rather than points by default. Figure 4.10 shows the result of the following code:

```
x <- 10 * (1:nrow(volcano))
y <- 10 * (1:ncol(volcano))
image(x, y, volcano, col = hcl.colors(100, "terrain"), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5), add = TRUE,
    col = "brown")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
```

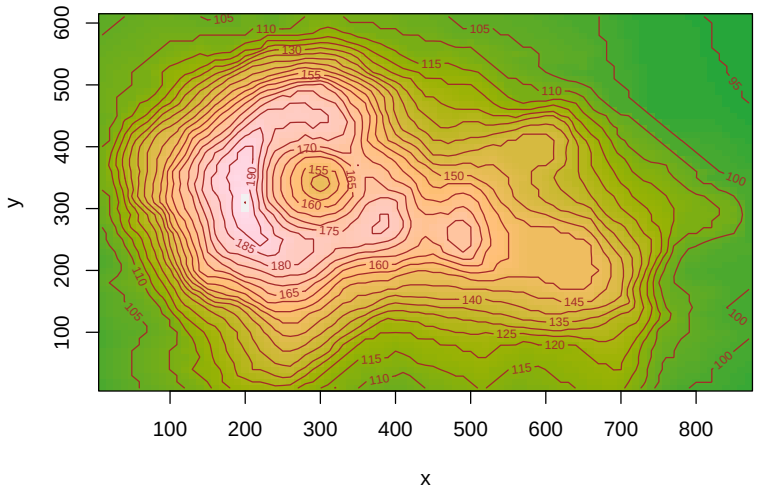


Figure 4.10: Maunga Whau volcano contours, code provided as an example of the `image()` function help.

It is composed of a set of colored rectangles: it is indeed a vector image.

If necessary, images can be produced in BMP (bitmap, without compression), JPEG (compressed with loss of quality), PNG (compressed without loss of quality, with possible transparency) or Tiff (compressed or not) formats.

4.8.2 Functions

ggplots can be saved to a file by the `ggsave()` function. The extension of the file name sets its format. See the help of the function for more details.

Other graphics require another method. The `postscript()` function produces an EPS file. The R code must call the function to create the file, produce the figure, and then close the file, for example:

```
# Open the file
postscript("Fig1.eps", width = 6, height = 4, horizontal = FALSE)
# Create the figure
plot(cars)
# Close the file
dev.off()
```

```
## pdf
##   2
```

The width and height (in inches) of a vector file are not important, but their ratio fixes the aspect of the figure. The size of the texts is fixed: increasing the size of the figure means decreasing the relative size of the texts: proceed by successive attempts, making sure that the legends remain readable at the final size of the figure.

The `horizontal` argument sets the orientation of the figure in a rather unpredictable way: proceed by trials.

The functions `eps()`, `pdf()`, `bmp()`, `jpeg()`, `png()` and `tiff()` work the same way. Refer to the function help for the choice of options (resolution, compression level, etc.). The `emf()` function is provided by the **devEMF** package.

Fonts are not included in EPS or PDF files. If necessary, the `embedFonts()` function can be used to remedy this, provided that GhostScript is installed.

4.8.3 ragg package

The **ragg**[50] package improves the quality of PNG, JPEG and TIFF files. The optimized functions are `agg_png()`, `agg_jpeg()` and `agg_tiff()`. Their usage is the same as that of the **grDevices** functions.

Markdown R documents produce PNG images for their HTML version. **ragg** improves their quality: the package must be installed and `dev = "ragg_png"` must be added to the **knitr** options. For this document, the options declared in `index.Rmd` are the following:

```
knitr::opts_chunk$set(
  cache=FALSE, # Cache chunk results
  echo = TRUE, # Show/Hide R chunks
  warning=FALSE, # Show/Hide warnings
  # Figure alignment and size
  fig.align='center', out.width='80%', fig.asp=.75,
  # Graphic devices (ragg_png is better than standard png)
  dev = c("ragg_png", "pdf"),
  # Code chunk format
  tidy=TRUE, tidy.opts=list(blank=FALSE, width.cutoff=60),
  size="scriptsize", knitr.graphics.auto_pdf = TRUE
  )
options(width=60)
```

[50] https://ragg.r-lib.org/

Finally, **ragg** can be used as the default graphics renderer in RStudio starting with version 1.4 (Menu "Tools > Global Options > General > Graphics > Backend").

4.9 Workflow

A workflow (see section 2.8) can be embedded in an R Markdown document starting with version 0.5 of the **targets** package.

```
library("targets")
```

4.9.1 Declaration of the workflow

The workflow is managed by code snippets of type `targets`. Their minimal header is `{targets}` instead of `{r}`, and they must be named. These code snippets are used to create the `_targets.R` file when they are run in non-interactive mode, namely while the document is being knitted. If they are run in interactive mode, for example in R Studio, their code is executed. The `tar_interactive = FALSE` option in their header allows them to be tested without knitting the whole document.

Any old workflow must be removed before writing the new one:

```
tar_unscript()
```

The first code chunk, with the `tar_globals=TRUE` option, writes the global options for the workflow To create the workflow shown in section 2.8, the code is simply:

````
```{targets targets_global, tar_globals=TRUE}
# Packages
tar_option_set(packages = c("spatstat", "dbmss"))
```
````

The functions used by the targets are declared in this type of code snippet: they are added to a file in the `_targets_r` working folder (different from the `_targets` folder which contains the target calculation files).

4.9.2 Declaration of targets

The targets themselves are declared in code snippets whose name is that of the destination variable.

````
```{targets X, tar_simple=TRUE}
runifpoint(NbPoints)
```
````

Each target requires a piece of code built in this way. The value of the target is the last value returned, just like a function that would not use `return()`.

During knitting, this simplified code (`tar_simple=TRUE`) is automatically transformed into a target code:

```
tar_target(X, {
  runifpoint(NbPoints)
})
```

```
## Define target X from chunk code.
## Establish _targets.R and _targets_r/targets/X.R.
```

The document readability is impaired by this particular syntax: **targets** is not useful for documents whose code, quick to execute, must be displayed in the text. On the other hand, if the code is long to execute and is not displayed, it is of considerable interest to limit the computation time.

The other bits of code needed to complete the flow are the following:

- `NbPoints`:

```
tar_target(NbPoints, {
    1000
})
```

```
## Define target NbPoints from chunk code.
## Establish _targets.R and _targets_r/targets/NbPoints.R.
```

- `d`:

```
tar_target(d, {
    sum(pairdist(X))/NbPoints/(NbPoints-1)
})
```

```
## Define target d from chunk code.
## Establish _targets.R and _targets_r/targets/d.R.
```

- `map`:

```
tar_target(map, {
    autoplot(as.wmppp(X))
})
```

```
## Define target map from chunk code.
## Establish _targets.R and _targets_r/targets/map.R.
```

4.9.3 Running the workflow

To start the target calculation, a standard piece of code (`{r}`) must call `tar_make()`:

```
tar_visnetwork()
```

```
tar_make()
```

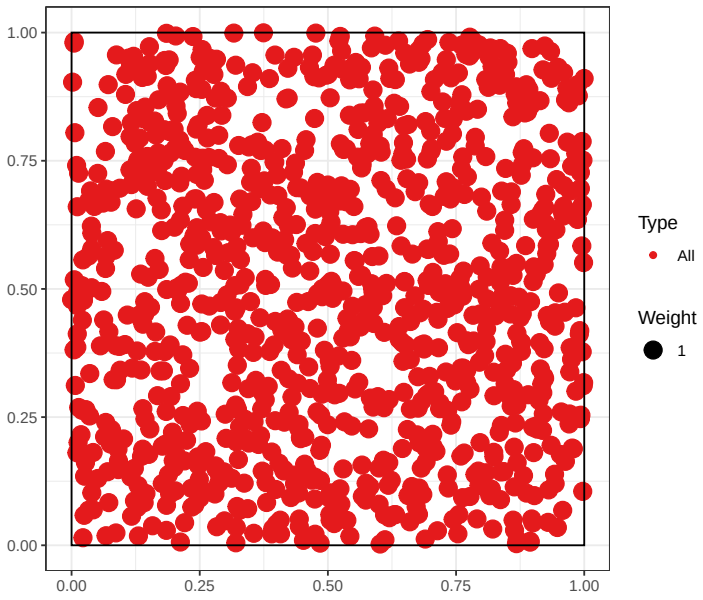
```
##   dispatched target NbPoints
##   completed target NbPoints [0.712 seconds]
##   dispatched target X
##   completed target X [0.001 seconds]
##   dispatched target d
##   completed target d [0.008 seconds]
##   dispatched target map
##   completed target map [0.015 seconds]
##   ended pipeline [0.852 seconds]
```

`tar_visnetwork()` is used to check that the workflow is correct before running it. When the document is finally produced, the `include=FALSE` option can be added to the header of this piece of code so that it does not produce any output.

4.9.4 Using the results

Code snippets that use target values must read them with `tar_read()`:

```
tar_read(map)
```



4.9.5 Source control

The **targets** files must be included in the source control. This way, calculations done locally will not be repeated by GitHub Actions (chapter 6) and building the document will be fast.

CHAPTER 5

PACKAGE

R packages extend its functionality with code provided by the developer community. They are the key to the success of R because they allow to quickly spread new methods resulting from research or to add new tools that can become standards, such as the **tidyverse**.

It is useful to produce a package when you have written new functions that form a coherent whole. A package for personal use or limited to a work team is simple to set up and the time saved by easily using the updated version of each function very quickly amortizes the time spent on making the package. This type of package is intended to be hosted on GitHub.

Packages with a wider use, which provide for example the code corresponding to a published method, are placed in the CRAN repository, from where they can be installed by the standard command `install.packages()`. CRAN per-

forms extensive code checks and only accepts packages that pass its test suite without any warning. They must respect the policies[1] of the repository.

The documentation for package creation is abundant. The reference book is Wickham (2015), which should be consulted as a reference.

The approach used here is to create a first package very quickly to understand that the process is quite simple. It will then be enriched with the elements necessary for a package distributed to other users than its designer: a complete documentation and tests of correct operation in particular.

5.1 First package

This introduction follows the recommendations of the blog *Creating a package in minutes*[2] from ThinkR.

5.1.1 Creation

Packages have a strict organization in a fixed file and directory structure. It is possible to create this structure manually but specialized packages can do it:

- **usethis** automates the creation of folders.
- **roxygen2** automates the mandatory documentation of packages.
- **devtools** is the developer's toolbox, allowing to build and test packages.

All three are to be installed first:

```
install.packages(c("usethis", "roxygen2", "devtools"))
```

The package to create will be an RStudio project. In the project menu, select "New Project > New Directory > R package using devtools…", choose the name of the project and its parent folder. The package will be called **multiple**, in the `%LOCALAPPDATA%\ProjectsR` folder, following the recommendations in the section 1.2.4.

The name of the package must respect the constraints of project names: no special characters, no spaces… It must also be evocative of the purpose of the package. If the package is to be distributed, all its documentation will be written in English, including its name.

The minimal structure is created:

- A `DESCRIPTION` file which indicates that the folder contains a package and specifies at least its name.
- A `NAMESPACE` file which declares how the package intervenes in the management of the names of R objects (its content will be updated by **roxygen2**).

[1] https://cran.r-project.org/web/packages/policies.html
[2] https://thinkr.fr/creer-package-r-quelques-minutes/

- An `R` file which contains the code of the functions offered by the package (empty at this stage).

The package can be tested right away: in the RStudio *Build* window, clicking on "Install and Restart" builds the package and loads it into R, after restarting the program to avoid any conflicts.

In the *Packages* window, **multiple** is now visible. It is loaded, but contains nothing.

5.1.2 First function

Files

Functions are placed in one or more `.R` files in the `R` folder. The organization of these files is free. For this example, a file with the name of each function will be created. Files grouping similar functions or a single file containing all the code are possible choices.

The choice made here is the following:

- A file that will contain the code common to the whole package: `package.R`.
- One file common to all functions: `functions.R`.

Creation

The first function, `double()`, is created and stored in the `functions.R` file:

```
double <- function(number) {
    return(2 * number)
}
```

At this point, the function is internal to the package and is not accessible from the working environment. To be sure, build the package (*Install and Restart*) and check that the function works:

```
double(2)
```

The result is a vector composed of two 0's because the called function is a homonym of the **base** package (see its documentation by typing `?double`):

```
base::double(2)
```

```
## [1] 0 0
```

In order for the function in our package to be visible, it must be *exported* by declaring it in the `NAMESPACE` file. This is the job of **roxygen2** which manages the documentation of each function at the same time. To activate it, place the cursor in the function and call the menu "Code > Insert Roxygen Skeleton". Comments are added before the function:

```r
#' Title
#'
#' @param number
#'
#' @return
#' @export
#'
#' @examples
double <- function(number) {
    return(2 * number)
}
```

Comments to **roxygen2** begin with `#'`:

- The first line contains the title of the function, i.e. a very short description: its name in general.
- The next line (separated by a line break) may contain its description (see *Description* in the help).
- The next line (after another line break) might contain more information (*Details* in the help).
- The arguments of the function are described by the `@param` lines.
- `@return` describes the result of the function.
- `@export` declares that the function is exported: it will be usable in the working environment.
- Examples can be added.

The documentation must be completed:

```r
#' double
#'
#' Double value of numbers.
#'
#' Calculate the double values of numbers.
#'
#' @param number a numeric vector.
#'
#' @return A vector of the same length as `number` containing the
#'   transformed values.
#' @export
#'
#' @examples
#' double(2)
#' double(1:4)
double <- function(number) {
    return(2 * number)
}
```

Don't hesitate to use the help of existing functions to respect R standards (here: `?log`):

- Keep in mind that functions are normally vector: `number` is by default a vector, not a scalar.
- Some elements start with a capital letter and end with a dot because they are paragraphs in the help file.

- The title does not have a period.
- The description of the parameters does not start with a capital letter.

Taking into account the changes in the documentation requires calling the `roxygenize()` function. In the *Build* window, the "More > Document" menu allows you to do this. Then build the package (*Install and Restart*) and check the result by running the function and displaying its help:

```
double(2)
?double
```

It is possible to automate the update of the documentation at each build of the package by the menu "Build > Configure Build Tools…": click on "Configure" and check the box "Automatically reoxygenize when running Install and Restart". This is an efficient choice for a small package but penalizing when the time to update the documentation increases with the complexity of the package. The package rebuild is most often used to test code changes: its speed is essential.

The documentation for **roxygen2** supports the Markdown[3] format.

At this stage, the package is functional: it contains a function and a beginning of documentation. It is time to run a check of its code: in the *Build* window, click on "Check" or use the `devtools::check()` command. The operation *reoxygenates* the package (updates its documentation), performs a large number of tests and returns a list of errors, warnings and notes detected. The goal is always to have no warnings: they must be handled immediately. For example, the following return is a warning about the non-conformity of the declared license:

```
> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    `use_gpl3_license()`
  Standardizable: FALSE

0 errors v | 1 warning x | 0 notes v
Erreur : R CMD check found WARNINGs
```

To correct it, update, run the update license command, starting with your name:

```
options(usethis.full_name = "Eric Marcon")
usethis::use_gpl3_license()
```

The list of valid licenses is provided by R[4].

After correction, run the tests again until the alerts disappear.

[3] https://roxygen2.r-lib.org/articles/markdown.html
[4] https://svn.r-project.org/R/trunk/share/licenses/license.db

5.1.3 Source control

It is time to put the code under source control.

Enable source control in the project options (figure 3.2). Restart RStudio on demand.

Create a repository on GitHub and push the local repository to it, as explained in the chapter 3.

Create the file `README.md`:

```
# multiple

An R package to compute mutiple of numbers.
```

The development of the package is punctuated by many commits at each modification and a push at each step, validated by a version number increment.

5.1.4 package.R

The `package.R` file is intended to receive the R code and especially the comments for **roxygen2** which concern the whole package. This file can also be named `multiple-package.R`, prefixed with the package name, for compatibility with **usethis**. It can be created under this name with the command:

```
usethis::use_package_doc()
```

The first comment block will generate the package help (`?multiple`).

```
#' @keywords internal
"_PACKAGE"
```

The "_PACKAGE" keyword indicates that package documentation must be produced. It could be written in the block, with a syntax identical to that of functions, but its default content is that of the `Description` field in the `DESCRIPTION` file. The `internal` keyword hides the package documentation in the help summary.

The documentation is updated by the `roxygen2::roxygenise()` command. After rebuilding the package, check that the help has appeared: `?multiple`.

5.2 Package organization

5.2.1 DESCRIPTION file

The file must be completed:

```
Package: multiple
Title: Calculate multiples of numbers
Version: 0.0.0.9000
Authors@R:
```

```
  person(given = "Eric",
           family = "Marcon",
           role = c("aut", "cre"),
           email = "e.marcon@free.fr",
           comment = c(ORCID = "0000-0002-5249-321X"))
Description: Simple computation of multiples of numbers,
  including fast algorithms for integers.
License: GPL-3
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

The package name is fixed and must not be changed.

Its title must describe in one line what it is used for. The title is displayed in the *Packages* window next to the package names.

The version must respect the conventions:

- The first number is the major version, 0 as long as the package is not stable, then 1. The major version only changes if the package is no longer compatible with its previous versions, which forces users to modify their code.
- The second is the minor version, incremented when new features are added.
- The third is the correction version: 0 at the origin, incremented at each code correction without new functionality.
- The fourth is reserved for development, and starts at 9000. It is incremented with each unstable version and disappears when a new stable version (*release*) is produced.

Example: a bug fix on version 1.3.0 produces version 1.3.1. The following development versions (unstable, not intended for production use) are 1.3.1.9000 then 1.3.1.9001, etc. The version number must be updated each time the package is pushed on GitHub. When the development is stabilized, the new version, intended to be used in production, is 1.3.2 if it does not bring any new functionality or 1.4.0 in the opposite case.

The description of the authors is rather heavy but simple to understand. The Orcid identifiers of academic authors can be used. If the package has several authors, they are placed in a `c()` function: `c(person(...), person(...))` for two authors. In this case, the role of each must be specified:

- “cre” for the creator of the package.
- “aut” for one of the other authors.
- “ctb” for a contributor, who may have reported a bug or provided some code.

The description of the package in one paragraph allows to give more information.

The license specifies how the package can be used and modified. GPL-3 is a good default, but other choices are possible[5].

The `LazyData` option means that the example data provided with the package can be used without calling it first by the `data()` function: this is the current standard.

Finally, the last two lines are handled by **roxygen2**.

5.2.2 NEWS.md file

The `NEWS.md` file contains the history of the package. New versions are added to the top of the file.

Create a first version of the file:

```
# multiple 0.0.0.9000

## New features

* Initial version of the package
```

The first level titles must contain the package name and version. Level 2 titles are free, but usually contain headings like "New features" and "Bug Fixes".

To avoid multiplying the versions described, it is advisable to change the current version and complete the documentation until the correction version changes (third number). Then, the entry corresponding to this version remains frozen and a new entry is added.

5.3 Vignette

A vignette is essential to document the package correctly:

```
usethis::use_vignette("multiple")
```

The file `multiple.Rmd` is created in the `vignette` folder. Add a subtitle in its header: the short description of the package:

```
title: "multiple"
subtitle: "Multiples of numbers"
```

The rest of the header allows R to build the vignette from R Markdown code.

The body of the vignette contains by default R code to declare the options for presenting the code snippets and loading the package. An introduction to the use of the package should be written in this document, in R Markdown.

During the development of the package, the vignette can be built manually by running:

[5] https://r-pkgs.org/description.html#description-license

```
devtools::build_vignettes("multiple")
```

The resulting files are placed in `doc/`: open the `.html` file to check the result.

RStudio does not create the package vignette when the "Install and Restart" command in the Build window is called. For a complete installation, two solutions are possible:

- Build the package source file ("Build > More > Build Source Package") and then install it ("Packages > Install > Install from > Package Archive file"). The source file is next to the project file.
- Push the package code on GitHub and then run:

```
remotes::install_github("multiple", build_vignettes = TRUE)
```

The vignette can then be displayed by the command:

```
vignette("multiple")
```

5.4 pkgdown

The **pkgdown** package creates a companion site to the package[6], which includes the `README.md` file as the home page, the vignette in a "Get Started" section, all of the help files with their executed examples (the "Reference" section), the `NEWS.md` file for a history of the package (the "Changelog" section), and information from the `DESCRIPTION` file.

Create the site with **usethis**:

```
usethis::use_pkgdown()
```

Then build the site. This command will be executed again at each version change of the package:

```
pkgdown::build_site()
```

The site is placed in the `docs` folder. Open the file `index.htm` with a web browser to view it. As soon as the project is pushed to GitHub, activate the repository pages so that the site is visible online (see section 3.7).

pkgdown places the site in the `docs` folder.

Add the address of the GitHub pages to a new line in the `DESCRIPTION` file:

```
URL: https://GitHubID.github.io/multiple
```

Also add it to the `_pkgdown.yml` file that was created empty, along with the following option:

[6]Example: https://EricMarcon.github.io/entropart/

```
url: https://GitHubID.github.io/multiple

development:
  mode: auto
```

pkgdown places the site in the `docs/dev` folder if the site for a stable (three-numbered) version of the package exists in `docs` and the current version is a development version (four-numbered). This way, users of a production version of the package have access to the site without it being disturbed by the development versions.

The site can be enriched in several ways:

- By adding articles in R Markdown format to the `vignettes/articles` folder. The vignette should not require significant computational resources to present examples because it is built at the same time as the package. The articles are generated by **pkgdown**, independently, and can therefore be more ambitious;
- By improving its presentation (grouping functions by themes, adding badges, a sticker[7]…): refer to the help of **pkgdown**.

To enrich the documentation of the package, it is possible to use a `README.Rmd` file in R Markdown format, to be knitted to create the standard `README.md` of GitHub, used as the home page of the **pkgdown** site, which can in this way present examples of use of the code. The approach is detailed in *R Packages*[8]. The added complexity is to be compared to the gain: a simple homepage (without code) with links to the vignette and articles is easier to implement.

5.5 Package specific code

5.5.1 Importing functions

Let's create a new function in `functions.R` that adds random noise to the double value:

```
fuzzydouble <- function(number, sd = 1) {
    return(2 * number + rnorm(length(number), 0, sd))
}
```

The noise is drawn in a centered normal distribution of standard deviation `sd` and added to the calculated value.

`rnorm()` is a function of the **stats** package. Even though the package is systematically loaded by R, the package to which the function belongs must be declared: the only exceptions are functions from the **base** package.

[7] The Shiny application **hexmake** allows easy creation of a sticker: https://connect.thinkr.fr/hexmake/

[8] https://r-pkgs.org/release.html?q=readme#readme-rmd

The **stats** package must first be declared in `DESCRIPTION` which contains an `Imports:` statement. All packages used by the **multiple** code will be listed, separated by commas.

```
Imports: stats
```

This "import" simply means that the **stats** package must be loaded, but not necessarily attached (see section 2.2), for **multiple** to work.

Then, the `rnorm()` function must be found in the **multiple** package environment. There are several ways to fulfill this requirement. First, the following comment could be provided for **roxygen2**:

```
#' @import stats
```

The entire namespace of the **stats** package would be attached to and accessible by the **multiple** package. This is not a good practice because it inreases the risk of name conflicts (see section 2.2). Note that the notion of import used here is different from that of `DESCRIPTION`, although they have the same name.

It is best to import only the `rnorm()` function by declaring it in the function documentation:

```
#' @importFrom stats rnorm
```

This is not an ideal practice either because the origin of the function would not be clear in the package code.

The best practice is to import nothing (in the sense of **roxygen2**) and to systematically qualify functions from other packages with the syntax `package::function()`. This is the solution chosen here because the `@importFrom` directive would import the function in the whole **multiple** package, not only in the `fuzzydouble()` function, at the risk of creating side effects (modifying the behavior of another function of the package which would not assume the import of `rnorm()`). Finally, the code of the function is as follows:

```
#' fuzzydouble
#'
#' Double value of numbers with an error
#'
#' Calculate the double values of numbers
#' and add a random error to the result.
#'
#' @param number a numeric vector.
#' @param sd the standard deviation of the Gaussian error added.
#'
#' @return A vector of the same length as `number`
#'  containing the transformed values.
#' @export
#'
#' @examples
#' fuzzydouble(2)
#' fuzzydouble(1:4)
```

```
fuzzydouble <- function(number, sd = 1) {
    return(2 * number + stats::rnorm(length(number), 0, sd))
}
```

5.5.2 S3 methods

S3 methods are presented in section 2.1.2.

Classes

Objects belong to classes:

```
# Class of a number
class(2)
```

```
## [1] "numeric"
```

```
# Class of a function
class(sum)
```

```
## [1] "function"
```

In addition to the basic classes, developers can create others.

Methods

The point of creating new classes is to adapt existing methods to them, the most common case being `plot()`. This is a generic method, i.e. a function template, without code, to be adapted to the class of object to be processed.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x109726518>
## <environment: namespace:base>
```

There are many variations of `plot` in R, which are functions with names of the form `plot.class()`. **stats** provides a function `plot.lm()` to create a figure from a linear model. Many packages create classes tailored to their objects and provide a `plot` method for each class. The functions can be listed:

```
# Some functions plot()
head(methods(plot))
```

```
## [1] "plot,ANY-method"   "plot,color-method"
## [3] "plot.AccumCurve"   "plot.acf"
## [5] "plot.ACF"          "plot.addvar"
```

```
# Total number
length(methods(plot))
```

```
## [1] 155
```

Conversely, the available methods for a class can be displayed:

```
methods(class = "lm")
```

```
##  [1] add1           alias          anova
##  [4] as_flextable   case.names     coerce
##  [7] confint        cooks.distance deviance
## [10] dfbeta         dfbetas        drop1
## [13] dummy.coef     effects        extractAIC
## [16] family         formula        fortify
## [19] hatvalues      influence      initialize
## [22] kappa          labels         logLik
## [25] model.frame    model.matrix   nobs
## [28] plot           predict        print
## [31] proj           qqnorm         qr
## [34] residuals      response       rstandard
## [37] rstudent       show           simulate
## [40] slotsFromS3    summary        variable.names
## [43] vcov
## see '?methods' for accessing help and source code
```

The `print` method is used to display any object (it is implicit when only the name of an object is entered):

```
my_lm <- lm(dist ~ speed, data = cars)
# Equivalent to '> my_lm'
print(my_lm)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

The `summary` method displays a readable summary of the object:

```
summary(my_lm)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
```

```
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

The other methods have been created specifically for the needs of the **stats** package.

Assigning an object to a class

In order for an object to belong to a class, it is sufficient to declare it:

```
x <- 1
class(x) <- "MyClass"
class(x)
```

```
## [1] "MyClass"
```

A more elegant way to do this is to add the new class to the set of classes to which the object already belongs:

```
y <- 1
class(y) <- c("MyClass", class(y))
class(y)
```

```
## [1] "MyClass" "numeric"
```

There is no consistency check between the real structure of the object and a structure of the class that would be declared elsewhere: the developer must make sure that the methods will find the right data in the objects that declare to belong to it. If not, errors will occur:

```
class(y) <- "lm"
tryCatch(print(y), error = function(e) print(e))
```

```
## <simpleError in x$call: $ operator is invalid for atomic vectors>
```

5.5.3 In practice

Creating a generic method

New generic methods can be created and declined according to the classes.

As an example, let's create a generic method `triple` which will calculate the triple of numbers in the package **multiple**, declined in two distinct functions: one for integers and one for reals. Calculations on integers are faster than those on reals, which justifies (at least in theory) the effort of writing two versions of the code.

```
# Generic Method
triple <- function(x, ...) {
    UseMethod("triple")
}
```

The generic method contains no code beyond its declaration. Its signature (i.e., the set of arguments) is important because functions derived from this method will necessarily have to have the same arguments in the same order and can only add additional arguments before ... (which is mandatory). As the nature of the first argument will depend on the class of each object, it is usual to call it x.

The method is declined in two functions:

```
triple.integer<- function (x, ...){
  return(x * 3L)
}
triple.numeric<- function (x, ...){
  return(x * 3.0)
}
```

In its integer version, x is multiplied by 3L, the suffix L meaning that 3 should be understood as an integer. In its real version, 3 can be written 3.0 to make it clear that it is a real. Under R, 3 without further specification is understood as a real.

The choice of function depends on the class of the object passed as argument.

```
# Integer argument
class(2L)
```

```
## [1] "integer"
```

```
# Integer result by the function triple.integer
class(triple(2L))
```

```
## [1] "integer"
```

```
# Real argument
class(2)
```

```
## [1] "numeric"
```

```
# Real result by the function triple.numeric
class(triple(2))
```

```
## [1] "numeric"
```

```
# Performance
microbenchmark::microbenchmark(triple.integer(2L), triple.numeric(2),
    triple(2L))
```

```
## Unit: nanoseconds
##                expr min  lq     mean median  uq    max
##  triple.integer(2L) 123 123  7345.56    164 164 719550
##   triple.numeric(2) 123 123 10093.79    164 164 994168
##          triple(2L) 533 574   613.77    574 574   4469
##  neval
##    100
##    100
##    100
```

The performance measurement by the **microbenchmark** package shows no difference between the functions `triple.integer()` and `triple.numeric` as expected because the time spent on the computation itself is negligible compared to the time spent calling the function. The generic method consumes much more time than the very simple calculations here. R indeed tests the existence of functions corresponding to the class of the object passed as argument to the generic methods. As an object can belong to several classes, it searches for a function adapted to the first class, then to the following classes successively. This search takes a lot of time and justifies the use of generic methods for the readability of the code rather than for performance: the interest of generic methods is to provide the user of the code with a single function for a given objective (`plot` to make a figure) whatever the data to be processed.

Creating a class

In a package, classes are created if the results of the functions justify it: a list structure and the identification of the class with an object ("lm" is the class of linear models). For each class created, the `print`, `summary` and `plot` methods (if a graphical representation is possible) should be written.

Let's write a function `multiple()` whose result will be an object of a new class, `multiple`, which will be a list storing the values to multiply, the multiplier and the result.

```
multiple <- function(number, times = 1) {
    # Calculate the multiples
    y <- number * times
    # Save in a list
    result <- list(x = number, y = y, times = times)
    # Set the class
    class(result) <- c("multiple", class(result))
    return(result)
}
# Class of the result
my_multiple <- multiple(1:3, 2)
class(my_multiple)
```

```
## [1] "multiple" "list"
```

The call to the `multiple()` function returns an object of class `multiple`, which is also of class `list`. In the absence of a `print.multiple()` function, R looks for the `print.list()` function, which does not exist, and falls back on the `print.default()` function:

```
my_multiple
```

```
## $x
## [1] 1 2 3
##
## $y
## [1] 2 4 6
##
## $times
## [1] 2
##
## attr(,"class")
## [1] "multiple" "list"
```

The `print.multiple` function must therefore be written for a readable display, limited to the result:

```
print.multiple <- function(x, ...) {
    print.default(x$y)
}
# New presentation
my_multiple
```

```
## [1] 2 4 6
```

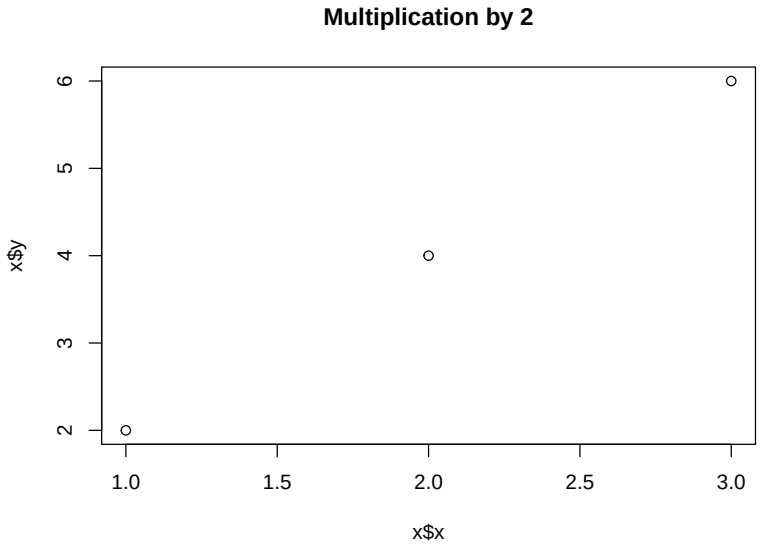
Details can be presented in the `summary` function:

```
summary.multiple <- function(object, ...) {
    print.default(object$x)
    cat("multiplied by", object$times, "is:\n")
    print.default(object$y)
}
# New display
summary(my_multiple)
```

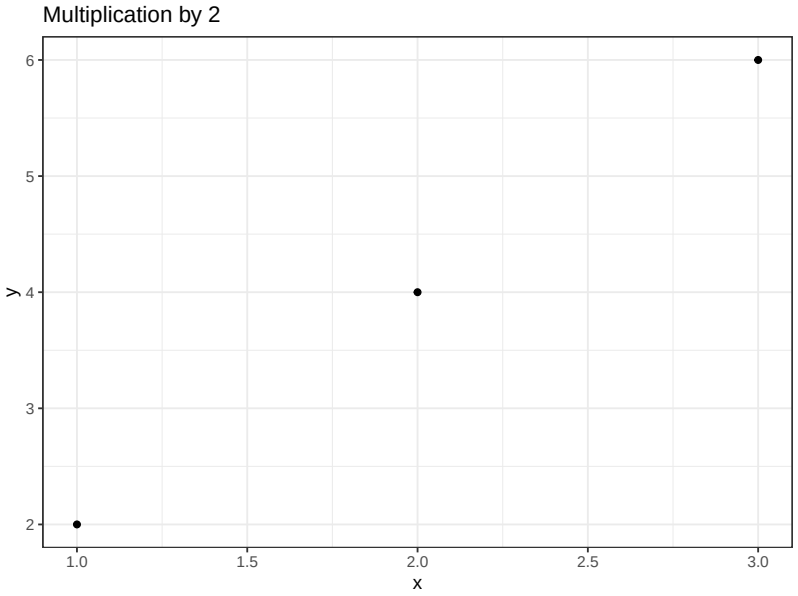
```
## [1] 1 2 3
## multiplied by 2 is:
## [1] 2 4 6
```

Finally, a `plot` function and an `autoplot` function complete the set:

```
plot.multiple <- function(x, y, ...) {
  plot.default(y=x$y, x=x$x, type = "p",
    main = paste("Multiplication by", x$times), ...)
}

autoplot.multiple <- function(object, ...) {
  data.frame(x = object$x, y = object$y) %>%
    ggplot2::ggplot() +
    ggplot2::geom_point(ggplot2::aes(x = .data$x, y = .data$y)) +
    ggplot2::labs(title = paste("Multiplication by",
                                object$times))
}

plot(my_multiple)
```




```
autoplot(my_multiple)
```



For technical reasons related to unconventional evaluation in the tidyverse, variable names used by `aes()` must be prefixed with `.data$` in packages and `rlang::.data` must be imported. Otherwise, the package check returns a note that the variables `x` and `y`, used by the arguments of `aes()` have not been declared and may not exist in the local environment (see section 2.2).

Documentation

Generic methods and functions that declare them must be documented like any other function.

Namespace management is a bit more complex:

- Generic methods must be exported:

```
#' @export
```

- Functions derived from generic methods should not be exported but declared as methods, with the name of the generic method and the processed class. **roxygen2** requires that an export directive be added but does not enforce it (as it should) in the `NAMESPACE` file that is used by R:

```
#' @method plot multiple
#' @export
```

- Since version 3 of **roxygen2**, the declaration `@method` is useless as long as the function name is unambiguously decomposable, like `plot.multiple`: `@export` is sufficient. If the derived function name has multiple dots, **roxygen2** may not automatically detect the generic and the object and `@method` must be maintained.

- Functions derived from generic methods from another package need to import the generic method, unless it is provided by **base** (`print` is provided by **base** and is therefore not affected):

```
#' @importFrom graphics plot
#' @importFrom ggplot2 autoplot
```

- The generics imported in this way must be re-exported by a directive to be placed for example just after the code of the derived function:

```
#' @export
graphics::plot

#' @export
ggplot2::autoplot
```

- **roxygen2** automatically creates a help file `reexports.Rd` in which there is a link to the original documentation of the re-exported generics.

In `DESCRIPTION`, the original package for each generic must be listed in the `Imports:` directive:

```
Imports: ggplot2, graphics
```

Last, importing functions from the tidyverse also requires some precautions:

- the **tidyverse** package is reserved for interactive use in R: there is no way to import it into `DESCRIPTION` because its dependencies may change and lead to unpredictable results. The **magrittr** package provides the pipes, mainly %>%. The **rlang** package provides the `.data` object shown below. They must be imported into `DESCRIPTION`.

```
Imports: magrittr, rlang, stats
```

- Since it is not possible to prefix the %>% with the package name, the function must be imported using the delimiters provided for functions whose names contain special characters:

```
#' @importFrom magrittr `%>%`
```

- Functions in the tidyverse that use column names from tibbles or dataframes generate warnings at package check time because these names are confused with undefined variable names. To avoid this confusion, the `.data` object of the **rlang** package is helpful (for example in `aes()` seen above). It must be imported:

```
#' @importFrom rlang .data
```

Finally, the complete code is as follows:

```
#' Multiplication of a numeric vector
#'
#' @param number a numeric vector
#' @param times a number to multiply
#'
#' @return an object of class `multiple`
#' @export
#'
#' @examples
#' multiple(1:2,3)
multiple <- function(number, times = 1) {
  # Calculate the multiples
  y <- number * times
  # Save in a list
  result <- list(x = number, y = y, times = times)
  # Set the class
  class(result) <- c("multiple", class(result))
  return(result)
}

#' Print objects of class multiple
#'
#' @param x an object of class `multiple`.
#' @param ... further arguments passed to the generic method.
#'
#' @export
#'
#' @examples
#' print(multiple(2,3))
print.multiple <- function(x, ...) {
  print.default(x$y)
}

#' Summarize objects of class multiple
#'
#' @param object an object of class `multiple`.
#' @param ... further arguments passed to the generic method.
#'
#' @export
#'
```

```
#' @examples
#' summary(multiple(2,3))
summary.multiple <- function(object, ...) {
  print.default(object$x)
  cat("multiplied by", object$times, "is:\n")
  print.default(object$y)
}

#' Plot objects of class multiple
#'
#' @param x a vector of numbers
#' @param y a vector of multiplied numbers
#' @param ... further arguments passed to the generic method.
#'
#' @importFrom graphics plot
#' @export
#'
#' @examples
#' plot(multiple(2,3))
plot.multiple <- function(x, y, ...) {
  plot.default(y=x$y, x=x$x, type = "p",
               main = paste("Multiplication by", x$times), ...)
}
#' @export
graphics::plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x109726518>
## <environment: namespace:base>
```

```
#' autoplot
#'
#' ggplot of the `multiple` objects.
#'
#' @param object an object of class `multiple`.
#' @param ... ignored.
#'
#' @return a `ggplot` object
#' @importFrom ggplot2 autoplot
#' @importFrom magrittr `%>%`
#' @importFrom rlang .data
#' @export
#'
#' @examples
#' autoplot(multiple(2,3))
autoplot.multiple <- function(object, ...) {
  data.frame(x = object$x, y = object$y) %>%
    ggplot2::ggplot() +
    ggplot2::geom_point(ggplot2::aes(x = .data$x, y = .data$y)) +
    ggplot2::labs(title = paste("Multiplication by",
                                object$times))
}
#' @export
ggplot2::autoplot
```

```
## function (object, ...)
## {
##     UseMethod("autoplot")
## }
## <bytecode: 0x1066c1a50>
## <environment: namespace:ggplot2>
```

5.5.4 C++ code

The use of C++ code has been seen in section 2.5. To integrate these functions in a package, the following rules must be respected:

- The `.cpp` files containing the code are placed in the `/src` folder of the project.
- The code is commented for **roxygen2** in the same way as for R functions, but with the C language comment marker:

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

//' timesTwo
//'
//' Calculates the double of a value.
//'
//' @param x A numeric vector.
//' @export
// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

- In `DESCRIPTION`, import the packages. **Rcpp**, and **RcppParallel** if parallelized code is used (delete its references otherwise), must be declared in `LinkingTo`:

```
Imports: Rcpp, RcppParallel
LinkingTo: Rcpp, RcppParallel
```

- Comments for **roxygen2** should be added to `package.R` ("multiple" is the package name):

```
#' @importFrom Rcpp sourceCpp
#' @importFrom RcppParallel RcppParallelLibs
#' @useDynLib multiple, .registration = TRUE
```

- C++ working files are excluded from source control in `.gitignore`:

```
# C binaries
src/*.o
src/*.so
src/*.dll
```

These changes are partly done automatically, for **Rcpp** only, by **usethis**, but manual insertion of the code is faster and more reliable: do not use this command.

```
# usethis::use_rcpp()
```

Building the package will lead to compiling the code: Rtools are therefore essential.

5.5.5 Tidy package

Any modern package should be tidyverse compatible, which requires little effort:

- To allow pipelines, the main argument of functions should be the first one.
- Functions that transform data should accept a dataframe or tibble as the first argument and return an object of the same format.
- Methods `plot()` should be doubled with methods `autoplot()` with the same arguments that produce the same graph with **ggplot2**.

5.6 Bibliography

The documentation of a package uses bibliographic references. They can be managed automatically with **Rdpack** and **roxygen2**. References used in R Markdown files (vignette, site produced by **pkgdown**) are not concerned.

5.6.1 Preparation

Bibliographic references must be placed in a BibTeX file `REFERENCES.bib` placed in the `inst` folder. This folder contains files that will be placed in the root of the package folder when it is installed.

Add the following line to `DESCRIPTION`:

```
RdMacros: Rdpack
```

Also add the package `Rdpack` to the list of imported packages:

```
Imports: magrittr, stats, Rcpp, Rdpack
```

Finally, import the `reprompt()` function from **Rdpack** by adding the following lines to the documentation for **roxygen2** in `package.R`:

```
#' @importFrom Rdpack reprompt
```

5.6.2 Citations

References are cited by the command `\insertCite{key}{package}` in the documentation for **roxygen2**. `package` is the name of the package in which the `REFERENCES.bib` file is to be searched: this will normally be the current package, but references to other packages are accessible, provided only that they use **Rdpack**.

`key` is the identifier of the reference in the file. The following example[9] is from the documentation of the **divent** package hosted on GitHub, in its `.R` file:

[9] **divent** package on GitHub: https://github.com/EricMarcon/divent/blob/master/R/package.R

```
#' divent
#'
#' Measures of Diversity and Entropy
#'
#' This package is a reboot of the **entropart** package \insertCite{Marcon2014c}{divent}.
#'
#' @importFrom Rdpack reprompt
#'
#' @references
#' \insertAllCited{}
"_PACKAGE"
```

```
## [1] "_PACKAGE"
```

The cited reference is in `inst/REFERENCES.bib`:

```
@Article{Marcon2014c,
  author  = {Marcon, Eric and Herault, Bruno},
  title   = {entropart, an R Package to Partition
             Diversity},
  journal = {Journal of Statistical Software},
  year    = {2015},
  volume  = {67},
  number  = {8},
  pages   = {1--26},
}
```

Citations are enclosed in parentheses. To place the author's name outside the parenthesis, add the statement `;textual`:

```
\insertCite{Marcon2014c;textual}{divent}
```

To cite several references (necessarily from the same package), separate them with commas.

At the end of the documentation of an object using citations, systematically add a list of references:

```
#' @references
#' \insertAllCited{}
```

5.7 Data

Data can be embedded in a package, especially for the clarity of the examples.

The simplest method is to use **usethis**. Create variables containing the data to be saved and then save them:

```
seq1_10 <- 1:10
seq1_100 <- 1:100
usethis::use_data(seq1_10, seq1_100)
```

An `.rda` file is created in the `data` folder for each variable created. With the `LazyData` option enabled in `DESCRIPTION`, variables will be available as soon

as the package is loaded, but will not actually be loaded into memory until after they are used for the first time.

Each variable must be documented in the `package.R` file:

```
#' seq1_10
#'
#' A sequence of numbers from 1 to 10
#'
#' @format A numeric vector.
#' @source Values computed by the R software,
#'   \url{https://www.r-project.org/}
"seq1_10"
```

The name of the variable is given in quotes after the comment block (instead of the R code of a function). `@format` describes the format of the data and `@source` is used to indicate its source.

5.8 Unit tests

Ideally, all code included in a package should be tested in multiple ways:

- Against syntax errors: R's checking procedures handle this quite well.
- To check the conformity of the computation results to the expected values.
- Against the occurrence of errors if users do not use the code as the developer intended (incorrect arguments passed to functions, inadequate data…).

Unit tests are used for the last two objectives. They are based on **testthat** to be integrated in the package:

```
usethis::use_testthat()
```

```
##
## Attaching package: 'testthat'

## The following object is masked from 'package:targets':
##
##     matches

## The following object is masked from 'package:dplyr':
##
##     matches

## The following object is masked from 'package:purrr':
##
##     is_null

## The following objects are masked from 'package:readr':
##
##     edition_get, local_edition
```

```
## The following object is masked from 'package:tidyr':
##
##     matches
```

The tests must be added as `.R` files whose names must begin with `test' in thetests/testthat‘ folder.

Each test (so the content of each file) starts with its context, i.e. a set of tests. For example, in a file `test_double.R`:

```
context("function double")
```

The tests are contained in files that group them by topic, for example `test_double.R`. The name of each test is passed as an argument to the function `test_that()`:

```
test_that("Double values are correct", {
    skip_on_cran()
    x <- 1:2
    # 2 x 2 should be 4
    expect_equal(double(x), c(2, 4))
    # The result should be a number (type = 'double')
    expect_type(double(x), "double")
    # Error management
    expect_error(double("a"))
})
```

```
## Test passed
```

All functions starting with `expect` allow to compare their first argument to a result: in the above example, the result of `double(1:2)` must be `2 4` and the type of this vector must be double precision real. The last test checks whether a string passed as an argument generates an error, which is not optimal: if the package handled the error, the returned message could be tested.

The `skip_on_cran()` command, to be used systematically, avoids running the tests on CRAN when the package is dropped there: CRAN has limited resources and strictly limits the time for checking packages on its platform. The tests will therefore have to be run on GitHub, thanks to continuous integration, see section 5.10.

The tests can be launched by the “More > Test package” menu of the *Build* window or by the `devtools::test()` command.

It is advisable to write the tests as soon as a function of the package is stabilized.

5.9 .gitignore file

The `.gitignore` file obtained at this stage is incomplete. It can be replaced by this one:

```
# History files
.Rhistory
.Rapp.history
# Session Data files
.RData
# Example code in package build process
*-Ex.R
# Output files from R CMD build
/*.tar.gz
# Output files from R CMD check
/*.Rcheck/
# RStudio files
.Rproj.user/
.Rprofile
# knitr and R markdown default cache directories
*_cache/
/cache/
# Temporary files created by R markdown
*.utf8.md
*.knit.md
# C binaries
src/*.o
src/*.so
src/*.dll
/src-i386/
/src-x64/
# uncomment if pkgdown is run by CI
# docs/
```

The last line is for the `docs/` folder, which receives the web site produced by **pkgdown**. It is commented out as long as the production of the site is done locally, but uncommented if it is entrusted to GitHub Actions (see next section).

5.10 Continuous integration

A package check must be done at each step of the development, which consumes a considerable amount of time. It can be automated very easily with the GitHub Actions service, triggered at each modification of the repository on GitHub. The analysis of the code coverage by tests (which parts of the code are tested or not) will be added.

GitHub is also able to rebuild the package documentation with **pkgdown**, another resource-consuming operation, after the tests have passed.

Section 6.3.3 details how to do this.

5.11 CRAN

Packages with an audience beyond the author's circle can be uploaded to CRAN. The rules to respect on CRAN are numerous[10]. They are checked by the `R CMD check` command with the `-- as.cran` option. The check must not return any errors, warnings, or notes before submitting the package.

[10] https://cran.r-project.org/web/packages/policies.html

5.11.1 Testing the package

Verification of the package by GitHub as part of continuous integration is not sufficient. The package must be tested on the development version of R. The *R-hub builder*[11] site allows to do it easily.

The package, which must not be a development version (limited to three numbers, see section 5.2.1), must be built in source format: in the *Build* window of RStudio, click on "More > Build Source Package". On the *R-hub builder* site, click on "Advanced", select the package source file and the test platform: *Debian Linux, R-devel, GCC*.

The **rhub** package allows you to use the same verification platform as the *R-hub builder* site from RStudio. The first step is to validate your email address with the `validate_email()` command. Then, just call the `check_for_cran()` function to run a full verification.

5.11.2 Submission

When the package is ready, submission to CRAN is done through the dedicated web site[12].

In case of rejection, process the requests and resubmit after incrementing the version number.

5.11.3 Maintenance

Requests for corrections are sent by CRAN from time to time, especially when the version of R changes. The email address of the package maintainer must remain valid and the requests must be processed quickly. Otherwise, the package is archived.

New versions of the package are submitted in the same way as the first one.

[11] https://builder.r-hub.io/
[12] https://xmpalantir.wu.ac.at/cransubmit/

CHAPTER 6

CONTINUOUS INTEGRATION

Continuous integration is the process of assigning an external service to verify a package, produce Markdown documents for web pages in a GitHub repository, or completely knit a website from code.

All of these tasks can be done locally on the desktop but are time consuming and may not be repeated with each update. In the context of continuous integration, they are systematically performed, in a transparent way for the user. In case of failure, an alert message is sent.

The implementation of continuous integration is justified for heavy projects, with regular updates. rather than for projects containing a simple Markdown document that is rarely modified.

6.1 Tools

6.1.1 GitHub Actions

The most frequently used tool for R projects filed on GitHub was *Travis CI*[1] but the service became fee-based in 2021.

GitHub Shares is a good replacement for Travis. This service is integrated with GitHub.

[1] https://travis-ci.org/

6.1.2 Codecov

To evaluate the code coverage rate of R packages, i.e. the proportion of code tested in some way (examples, unit tests, vignette), the *Codecov*[2] service integrates perfectly with GitHub.

You need to open an account, preferably by authenticating through GitHub.

6.1.3 GitHub Pages

GitHub web pages can be hosted in the `docs` directory of the master branch of the project: this is the solution chosen when they are produced on the workstation.

If they are produced by continuous integration, they will be hosted in a dedicated branch called `gh-pages`.

6.2 Principles

The production of a document is treated as an example. The objective is to have GitHub knit a Markdown project. This practice is appropriate for book projects, which require a lot of resources for their construction. In this type of project, the code is knit by **knitr** to produce several documents, typically in HTML and PDF formats, accessible on GitHub pages. When documents are produced locally, they are placed in the `docs` folder and pushed to GitHub.

In order for GitHub to do this, a few settings are required.

6.2.1 Getting a personal access token

To write to GitHub, the continuous integration service will have to authenticate with a Personal Access Token (PAT) whose creation is described in section 1.4.4.

Generate a new token, describe it as "GitHub Actions" and give it "repo" authorization, i.e. modify *all* repositories (it is not possible to limit access to a particular repository).

6.2.2 Project secrets

On GitHub, display the project settings and select "Secrets". The "New Repository Secret" button allows you to store variables used in GitHub Actions scripts (publicly visible) without exposing their value. The personal access token is essential for GitHub Actions to write their production in the project. Create a secret named "GH_PAT" and enter the previously saved token value. After clicking on "Add Secret", the token cannot be read anymore.

To allow sending success or failure messages without exposing your email address, create a secret named "EMAIL" which contains it.

[2] https://codecov.io/

6.2.3 Activation of the repository on CodeCov

The analysis of the code coverage of packages is useful to detect untested code portions. On the other hand, the analysis of the coverage of document projects is not useful.

To activate a repository, you need to authenticate on the CodeCov website with your GitHub account. The list of repositories is displayed and can be updated. If the repositories to be processed are hosted by an organization, for example the repositories of a GitHub classroom, you have to update the list of organizations by following the instructions (a link allows you to quickly modify the GitHub options to authorize Codecov to read the data of an organization) and update the list of repositories again. Finally, when the repository you are looking for is visible, you have to activate it. Ignore Codecov's token system.

6.2.4 Scripting GitHub actions

A GitHub workflow is a succession of jobs with steps. A workflow is triggered by an event, usually every *push* of the project, but also at regular intervals (*cron*).

Typically, the workflows created here contain two jobs: the first one installs R and the necessary components and executes R scripts (which constitute its successive steps); the second one publishes the obtained files in GitHub pages.

The workflows are configured in a YAML file placed in the `.github/workflows/` folder of the project. The different parts of the script are presented below. The full script is that of this document, accessible on GitHub[3].

Trigger

The action is triggered whenever updates are pushed to GitHub:

```
on:
  push:
     branches:
       - master
```

The branch taken into account is *master* (to be replaced by *main* if necessary).

To trigger the action periodically, use the syntax of cron (the Unix job scheduling system):

```
on:
  schedule:
    - cron: '0 22 * * 0' # every sunday at 22:00
```

The successive values are for minutes, hours, day (day of the month), month and day of the week (0 for Sunday to 6 for Saturday). The * allow to ignore a value.

The `push` and `schedule` entries can be used together:

[3] https://github.com/EricMarcon/WorkingWithR/blob/master/.github/workflows/bookdown.yml

```
on:
  push:
     branches:
       - master
  schedule:
    - cron: '0 22 * * 0'
```

Currently, scheduling is only taken into account in the *master* branch.

Workflow name

The name of the workflow is free. It will be displayed by the badge that will be added to the project's `README.md` file (see section 6.4).

```
name: bookdown
```

First job

The jobs are described in the `jobs` section. `renderbook` is the name of the first job: it is free. Here, the main action will be to produce a bookdown with the `render_book()` function, hence the name.

```
jobs:
  renderbook:
    runs-on: macOS-latest
```

The `runs-on` statement describes the operating system on which the job should run. The possible choices are Windows, Ubuntu or MacOS[4]. Continuous integration of R on GitHub usually uses MacOS which has the advantage of using compiled R packages so much simpler (some packages require libraries outside of R for their compilation) and quicker to install, while allowing the use of scripts.

First steps

The steps are described in the `steps` section.

```
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Setup R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        run: |
          brew install pandoc
```

Each step is described by its name (free) and what it does.

The strength of GitHub Actions is to allow the use of *actions* written by other people and stored in a public GitHub project. An action is a script with metadata describing its use. Its development is accompanied by successive version

[4] https://docs.github.com/en/free-pro-team@latest/actions/reference/workflow-syntax-for-github-actions#jobsjob_idruns-on

numbers. An action is called by the statement `uses:`, the GitHub project that contains it and its version.

In their respective GitHub project, actions exist in their development version (`@master`) and in step versions (*release*) accessible by their number (`@v1`). These stage versions are preferable because they are stable.

General actions are made available by GitHub in the GitHub Actions organization[5]. The "actions/checkout" action is used to get into the main branch of the project processed by the workflow: it is usually the first step of all workflows.

The next action is the installation of R, provided by the *R infrastructure* organization[6].

The installation of pandoc (software external to R but necessary for R Markdown) can be done by a command executed by MacOS. It is called by `run:` and can contain several lines (hence the `|`). This script depends on the operating system: `brew` is the package manager of MacOS. To avoid system specifics, it is better to use an action:

```
- name: Install pandoc
  uses: r-lib/actions/setup-pandoc@v2
```

It is often possible to choose between calling an action or writing the corresponding code. The choice made here is to favour actions for everything to do with the system, such as installing software, but to use scripts for R commands, such as checking packages. The aim is to control the R code precisely and limit dependencies on additional packages. The opposite strategy is developed in Wickham and Bryan (2023) which relies entirely on actions to perform R tasks.

Packages

This step uses `Rscript` as its command environment, which allows it to execute R commands directly.

```
- name: Install packages
  env:
    GITHUB_PAT: ${{ secrets.GH_PAT }}
  run: |
    options(pkgType = "binary")
    options(install.packages.check.source = "no")
    install.packages(c("remotes", "bookdown", "formatR", "tinytex"))
    tinytex::install_tinytex(bundle = "TinyTeX")
    remotes::install_deps(dependencies = TRUE)
  shell: Rscript {0}
```

The packages used to produce the document are listed:

- **remotes** for its `install_deps()` function.

[5] https://github.com/actions/
[6] https://github.com/r-lib/

- **bookdown** for knitting.
- **formatR** for formatting code snippets (`tidy=TRUE`).
- **tinytex** to have a LaTeX distribution.

The other packages, those used by the project, are read in the `DESCRIPTION` file by the `install_deps()` function.

On MacOS, packages are installed by default in binary version, but from their source code if it is more recent. The creation of binary packages takes a few days at CRAN: this situation is not rare. Packages containing only R code or C++ code without reference to external libraries can be installed without problems. On the other hand, if the package requires libraries external to R or a compilation of Fortran code, the installation fails. It would therefore be necessary to install the necessary libraries (and possibly a Fortran compiler) for all the packages on which the project depends: this solution is not realistic because it implies an inventory of all the dependencies, which may change, and a large number of time-consuming and useless installations most of the time, when the binary packages are up to date. A better solution is to force the installation of binary packages even if the source code is newer: this is the purpose of the two R options defined before calling `install.packages()`.

Finally, the secret `GH_PAT` is passed to R in an environment variable, `GITHUB_PAT`, necessary to install packages from their source code on GitHub. GitHub limits the rate of anonymous access for all GitHub actions (all accounts) and may reject the request: in practice, using `install_github()` in a GitHub action is only possible with this environment variable.

Knitting

The production of the document is started by an R command.

```
- name: Render pdf book
  run: |
    bookdown::render_book("index.Rmd", "bookdown::pdf_book")
  shell: Rscript {0}
- name: Render gitbook
  run: |
    bookdown::render_book("index.Rmd", "bookdown::gitbook")
  shell: Rscript {0}
```

The parameters declared in `_output.yml` are used.

The PDF file must be produced before the GitBook format in order for its download link to be added to the menu bar of the GitBook site. On the other hand, R must be closed and reopened between the two renderings otherwise the tables are not created correctly in GitBook[7]. The two steps should not be combined into one.

[7] https://stackoverflow.com/questions/46080853/why-does-rendering-a-pdf-from-rmarkdown-require-closing-rstudio-between-renders/46083308#46083308

Backup

The result of the knitting, placed in the `docs` folder of the virtual machine in charge of continuous integration, must be preserved so that the next job can use it.

The last step of the production job uses the `upload-artifact` action for this.

```
    - name: Upload artifact
      uses: actions/upload-artifact@v3
      with:
        name: _book
        path: docs/
```

The content of `docs` is saved as an *artifact* named "_book". Artifacts are publicly visible on the GitHub Project Actions page.

After its last step, the virtual machine is destroyed.

Publication

Publishing the artifact to the `gh-pages` branch of the project requires another job.

```
deploy:
  runs-on: ubuntu-latest
  needs: renderbook
  steps:
    - name: Download artifact
      uses: actions/download-artifact@v3
      with:
        # Artifact name
        name: _book
        # Destination path
        path: docs
    - name: Deploy to GitHub Pages
      uses: Cecilapp/GitHub-Pages-deploy@v3
      env:
        GITHUB_TOKEN: ${{ secrets.GH_PAT }}
      with:
        email: ${{ secrets.EMAIL }}
        build_dir: docs
        jekyll: no
```

The job is named "deploy" (the name is free). It runs on a virtual machine on Ubuntu. It can only be launched if the "renderbook" job has succeeded. Its steps are the following:

- *Download artifact*: Restore the `docs` folder;
- *Deploy to GitHub Pages*: copy the `docs` folder to the `gh-pages` branch.

This last step uses the `GitHub-Pages-deploy` action provided by the *Cecilapp* organization. It uses an environment variable, `GITHUB_TOKEN`, to authenticate and parameters:

- *email*: the email address to which the run report is sent. To avoid exposing the address publicly, it has been stored in a project secret.
- *buid_dir*: the directory to publish.
- *jekyll:no* to create an empty file named `.nojekyll` that tells GitHub pages not to try to treat their content as a Jekyll website.

6.2.5 Confidential data in a public repository

If the project contains confidential data (section 3.6), GitHub Actions must use the project's private key to extract it from their vault.

The private key must be stored in a project secret, named "RSA". The next step, to be inserted before the knitting step, writes the key to a file for the project code to access.

```
- name: Private key
  run: |
    cat("${{ secrets.rsa }}", file="<RepoID>.rsa"
  shell: Rscript {0}
```

6.3 Script templates

Script templates for all types of projects are presented here.

The `gh-pages` branch is created automatically by the scripts. Check after the first execution that GitHub pages are enabled on this branch (section 3.7). Then delete the `docs` folder if it existed, push the modification on GitHub and finally add the following line to the `.gitignore` file to be able to knit the projects locally without disturbing GitHub:

```
docs/
```

6.3.1 memoiR

The `build_ghworkflow()` function of the **memoiR** package automatically creates the scripts needed to produce the package's templates. The script is always named `memoir.yml`.

These scripts do not need a `DESCRIPTION` file for the installation of the dependencies but each document must contain in its parameter code (`Options`) the list of all packages needed for its knitting (stored in the `Packages` variable).

They all require the same preparation: the `GH_PAT` and `EMAIL` secrets must be registered in the GitHub project (section 6.2.2).

Book or thesis

The workflow is called `rmarkdown`; its production job is `render`.

```
on:
  push:
```

```
   branches:
     - master

name: rmarkdown

jobs:
  render:
    runs-on: macOS-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Setup R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install dependencies
        run: |
          options(pkgType = "binary")
          options(install.packages.check.source = "no")
          install.packages(c("distill", "downlit", "memoiR", "rmdformats", "tinytex"))
          tinytex::install_tinytex(bundle = "TinyTeX")
        shell: Rscript {0}
      - name: Render pdf book
        run: |
          bookdown::render_book("index.Rmd", "bookdown::pdf_book")
        shell: Rscript {0}
      - name: Render gitbook
        run: |
          bookdown::render_book("index.Rmd", "bookdown::gitbook")
        shell: Rscript {0}
      - name: Upload artifact
        uses: actions/upload-artifact@v3
        with:
          name: ghpages
          path: docs
  deploy:
    runs-on: ubuntu-latest
    needs: render
    steps:
      - name: Download artifact
        uses: actions/download-artifact@v3
        with:
          name: ghpages
          path: docs
      - name: Deploy to GitHub Pages
        uses: Cecilapp/GitHub-Pages-deploy@v3
        env:
          GITHUB_TOKEN: ${{ secrets.GH_PAT }}
        with:
          email: ${{ secrets.EMAIL }}
          build_dir: docs
          jekyll: no
```

Articles or slideshows

The workflow is called `rmarkdown`; its production job is `render`.

```
on:
  push:
   branches:
     - master

name: rmarkdown
```

```
jobs:
  render:
    runs-on: macOS-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Setup R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install dependencies
        run: |
          options(pkgType = "binary")
          options(install.packages.check.source = "no")
          install.packages(c("memoiR", "rmdformats", "tinytex"))
          tinytex::install_tinytex()
        shell: Rscript {0}
      - name: Render Rmarkdown files
        run: |
          RMD_PATH=($(ls | grep "[.]Rmd$"))
          Rscript -e 'for (file in commandArgs(TRUE)) |>
              rmarkdown::render(file, "all")' ${RMD_PATH[*]}
          Rscript -e 'memoiR::build_githubpages(bundle = "TinyTeX")'
      - name: Upload artifact
        uses: actions/upload-artifact@v3
        with:
          name: ghpages
          path: docs
  deploy:
    runs-on: ubuntu-latest
    needs: render
    steps:
      - name: Download artifact
        uses: actions/download-artifact@v3
        with:
          name: ghpages
          path: docs
      - name: Deploy to GitHub Pages
        uses: Cecilapp/GitHub-Pages-deploy@v3
        env:
          GITHUB_TOKEN: ${{ secrets.GH_PAT }}
        with:
          email: ${{ secrets.EMAIL }}
          build_dir: docs
          jekyll: yes
```

The knitting stage uses a script to list all the `.Rmd` files, process them (all the output formats listed in their yaml header are produced). The `build_githubpages()` function (see section 4.3.2) places the results in `docs`.

The deployment job tells GitHub pages to use Jekyll, i.e. to use the `README.md` file as the home page.

Localization

If the knitting step needs to change the language used by R, for example to correctly display the production date of documents, it can be changed like this:

```
      - name: Render Rmarkdown files
        run: |
          Sys.setlocale("LC_TIME", "fr_FR")
          lapply(list.files(pattern="*.Rmd"),
```

```
                      function(file) rmarkdown::render(file, "all"))
  memoiR::build_githubpages()
shell: Rscript {0}
```

The selection of files is done by an R script, which includes a localization command, here in French.

This step can be completed by selecting a GitHub Pages theme so that the home page contains a link to the code:

```
run: |
  echo 'theme: jekyll-theme-slate' > docs/_config.yml
```

The theme here is “Slate”, one of the choices offered by the GitHub pages.

Debugging

It can happen that the compilation of the `.tex` file to produce the PDF file fails, although knitting in HTML does not generate an error. The LaTeX compiler is indeed more demanding than pandoc (which produces the HTML file).

The first check consists in knitting the problematic document into PDF locally, on your workstation, with **tinytex**. The LaTeX packages must be updated to be the same as those used by the GitHub actions: to do this, run:

```
tinytex::tlmgr_update()
```

If the compilation works locally but not on Github, you have to inspect the `.log` file which records all the events generated by the compiler, but this file is not kept after the failure of GitHub Actions. So you have to modify the script to copy the file in `docs` and then save the result despite the error.

The “Upload artifact” step is modified to run despite the failure of the previous step by adding the line `if:`:

```
- name: Upload artifact
  if: always()
  uses: actions/upload-artifact@v3
  with:
    name: _book
    path: docs/
```

A step is added before “Upload artifact” to copy the result of the knitting and the `.log` file:

```
- name: Move files to docs
  if: always()
  run: |
    Rscript -e 'memoiR::build_githubpages()'
    cp *.log docs
```

After the action fails, the saved artifact can be downloaded from GitHub: it is on the action summary page. It is a compressed file that contains the whole `docs` folder.

6.3.2 Blogdown website

The file called `blogdown.yml` is very similar. The name of the workflow is `blogdown` and the name of the production job is `buildsite`.

```
on:
  push:
     branches:
       - master
  schedule:
    - cron: '0 22 * * 0'

name: blogdown

jobs:
  buildsite:
    runs-on: macOS-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Setup R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install packages
        run: |
          options(pkgType = "binary")
          options(install.packages.check.source = "no")
          install.packages(c("remotes", "blogdown", "formatR"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}
      - name: Build website
        run: |
          blogdown::install_hugo(force = TRUE)
          blogdown::build_site(local = TRUE, build_rmd = TRUE)
        shell: Rscript {0}
      - name: Upload artifact
        uses: actions/upload-artifact@v2
        with:
          name: _website
          path: public/
  deploy:
    runs-on: ubuntu-latest
    needs: buildsite
    steps:
      - name: Download artifact
        uses: actions/download-artifact@v3
        with:
          # Artifact name
          name: _website
          # Destination path
          path: public
      - name: Deploy to GitHub Pages
        uses: Cecilapp/GitHub-Pages-deploy@v3
        env:
          GITHUB_TOKEN: ${{ secrets.GH_PAT }}
        with:
          build_dir: public
          email: ${{ secrets.EMAIL }}
          jekyll: no
```

The `Build website` step uses the **blogdown** package to install Hugo (the website generator) and then build the website.

If the website uses online data that warrants periodic updating, GitHub Actions can be run daily, weekly or monthly in addition to the rebuilds triggered by a repository change (see section 6.2.4). Here, the site is rebuilt every Sunday at 10pm.

Example: the page that displays the bibliometrics of the author's website[8] queries Google Scholar to display the citations of the publications. The site is updated weekly to keep the statistics current.

6.3.3 R Packages

An optimal script for checking a package is as follows:

```
on:
  push:
    branches:
      - master

name: R-CMD-check

jobs:
  R-CMD-check:
    runs-on: macOS-latest
    steps:
      - name: Pull the repository
        uses: actions/checkout@v4
      - name: Install R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install R packages
        run: |
          options(pkgType = "binary")
          options(install.packages.check.source = "no")
          install.packages(c("remotes", "roxygen2", "rcmdcheck", "covr", "pkgdown"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}
      - name: Update the documentation
        run: roxygen2::roxygenize()
        shell: Rscript {0}
      - name: Commit and push the repository
        uses: EndBug/add-and-commit@v9
      - name: Check the package
        run: rcmdcheck::rcmdcheck(args = "--no-manual", error_on = "warning")
        shell: Rscript {0}
      - name: Test coverage
        run: covr::codecov(type="all")
        shell: Rscript {0}
      - name: Install the package
        run: R CMD INSTALL .
      - name: Pkgdown
        run: |
          git config --local user.email "actions@github.com"
          git config --local user.name "GitHub Actions"
          Rscript -e 'pkgdown::deploy_to_branch(new_process = FALSE)'
```

The file is named `check.yml`. It contains only one job, named `R-CMD-check`, as the workflow.

[8] https://EricMarcon.github.io/fr/publication/

The script does not use **Renv** to handle packages because package checking must work with the current versions on CRAN. **Remotes** installs the necessary packages from the `DESCRIPTION` file.

The `Roxygenize` step updates the package documentation. The updated files are pushed into the main project branch by the `add-and-commit` action. These two steps ensure that the package is in a consistent state, even if the author failed to execute the `roxygenize()` function before pushing his code to GitHub. To avoid triggering a loop to check code pushed in this way, the access token used must be that of the current script, created by GitHub each time it is run. By default, this token does not have the right to modify the repository. You therefore need to give it this right: on GitHub, display the project parameters and select 'Actions', 'General'. In the "Workflow permissions" section, select "Read and write permissions".

The `Check` step checks the package. Warnings are treated as errors.

The `Test coverage` step uses the **covr** package to measure the coverage rate and uploads the results to the Codecov site.

Finally, the last two steps install the package and then use **pkgdown** to create the documentation site for the package and push it into the `gh-pages` branch of the project.

This script contains only one job: the deployment of the documentation site is directly executed by **pkgdown**. Its success is displayed by a badge in the `README.md` file (see section 6.4)

More complex scripts are proposed by R-lib[9], in particular to run the tests on several operating systems and several versions of R. These advanced tests are to be performed before submitting to CRAN (section 5.11) but consume too much resource for systematic use.

6.3.4 Pull requests

Pull requests can be tested by very similar scripts to check that they do not generate errors before merging them.

One effective method is to create a new script in the `.github/workflows/` folder, starting from a copy of the existing script. The new script will be named `pr.yml`. The trigger must be changed: `pull_request` replaces `push`:

```
on:
  pull_request:
     branches:
       - master
```

memoiR

The scripts for checking documents created by **memoiR** must be cut after the `Render gitbook` step: the artefact must not be saved and the deployment task must be deleted. The script is as follows:

[9] https://github.com/r-lib/actions/tree/master/examples#standard-ci-workflow

```
on:
  pull_request:
    branches:
      - master

name: rmarkdown

jobs:
  render:
    runs-on: macOS-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Setup R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install dependencies
        run: |
          options(pkgType = "binary")
          options(install.packages.check.source = "no")
          install.packages(c("distill", "downlit", "memoiR", "rmdformats", "tinytex"))
          tinytex::install_tinytex(bundle = "TinyTeX")
        shell: Rscript {0}
      - name: Render pdf book
        env:
          GITHUB_PAT: ${{ secrets.GH_PAT }}
        run: |
          bookdown::render_book("index.Rmd", "bookdown::pdf_book")
        shell: Rscript {0}
      - name: Render gitbook
        env:
          GITHUB_PAT: ${{ secrets.GH_PAT }}
        run: |
          bookdown::render_book("index.Rmd", "bookdown::bs4_book")
        shell: Rscript {0}
      # Don't upload the artifact and don't deploy
```

R Packages

Scripts dedicated to checking packages should not push updates to their documentation through **Roxygenize2**, nor should they deploy their **pkgdown** updates to GitHub pages. The coverage rate does not need to be measured. The script is as follows:

```
on:
  pull_request:
    branches:
      - master

name: R-CMD-check

jobs:
  R-CMD-check:
    runs-on: macOS-latest
    steps:
      - name: Pull the repository
        uses: actions/checkout@v4
      - name: Install R
        uses: r-lib/actions/setup-r@v2
      - name: Install pandoc
        uses: r-lib/actions/setup-pandoc@v2
      - name: Install R packages
```

```
      run: |
        options(pkgType = "binary")
        options(install.packages.check.source = "no")
        install.packages(c("remotes", "roxygen2", "rcmdcheck", "covr", "pkgdown"))
        remotes::install_deps(dependencies = TRUE)
      shell: Rscript {0}
    - name: Update the documentation
      run: roxygen2::roxygenize()
      shell: Rscript {0}
      # Don't push
    - name: Check the package
      run: rcmdcheck::rcmdcheck(args = "--no-manual", error_on = "warning")
      shell: Rscript {0}
    # Don't test coverage
    - name: Install the package
      run: R CMD INSTALL .
    - name: Pkgdown
      # Build the package site locally
      run: Rscript -e 'pkgdown::build_site()'
```

When pull requests are submitted, the corresponding test is run and its results included in the discussion.

6.4 Add badges

The success of GitHub Actions can be seen by adding a badge to the `README.md` file, right after the file title. On the project page, choose "Actions" then select the action (in "Workflows"). Click on the "…" button and then on "Create Status Badge". Paste the Markdown code:

```
# Project name
![bookdown](https://github.com/<GitHubID>/<RepoID>/workflows/<Workflow>/badge.svg)
```

The name of the workflow was declared in the `name:` entry in the GitHub actions configuration file.

The coverage rate measured by Codecov can also be displayed by a badge:

```
[![codecov](https://codecov.io/github/<GitHubID>/
  <RepoID>/branch/master/graphs/badge.svg)]
  (https://codecov.io/github/<GitHubID>/<RepoID>)
```

CHAPTER 7

SHINY

Shiny allows to publish an interactive application using R code as a web site. The site can run locally, on a user's workstation that launches it from RStudio, or online, on a dedicated server running Shiny Server[1].

Basically, a form allows to enter the arguments of a function and a visualization window to display the results of the calculation.

A Shiny application makes the execution of the code very simple, even for users not familiar with R, but obviously limits the possibilities.

7.1 First application

In RStudio, create an application with the menu "File > New File > Shiny Web App…", enter the name of the application "MyShinyApp" and select the folder where to put it.

The name of the application has been used to create a folder that we now need to transform into a project (project menu in the top right of RStudio, "New Project > Existing Directory", select the application folder).

The application file named `app.R` contains two functions: `ui()` which defines the GUI and `server()` which contains the R code to be executed. The application can be launched by clicking on `Run App` in the code window.

[1] https://rstudio.com/products/shiny/download-server/

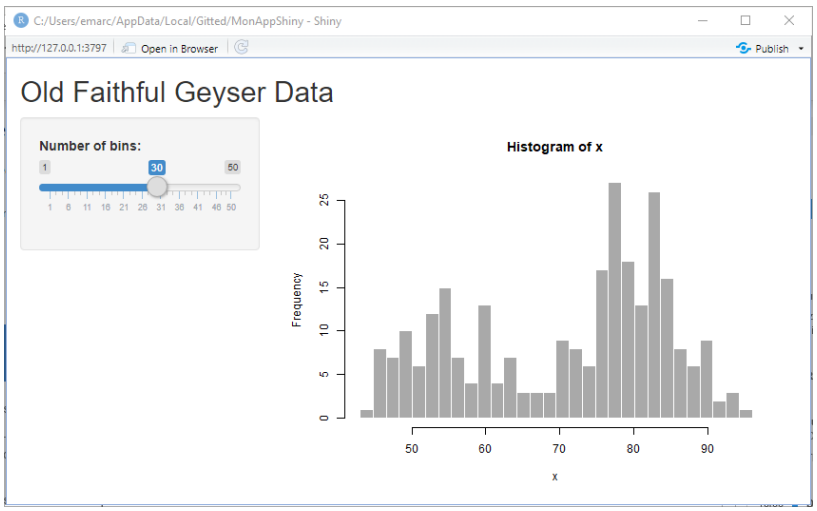


Figure 7.1: Shiny Application *Old Faithful Geyser Data*.

The correspondence between the displayed window (figure 7.1) and the `ui()` function code is easy to see:

- The title of the application is displayed by the `titlePanel()` function.
- The slider that sets the number of bars in the histogram is created by `sliderInput()`.
- The `sidebarLayout()` function sets the layout of the page elements, `sidebarPanel` for the input controls and `mainPanel()` for the result display.

The result is displayed by the `plotOutput()` function whose argument is the name of an element of `output`, the variable filled by the `server()` function.

Any modification of an element of the interface, precisely of an element displayed by a function whose name ends with `Input()` (there are some for all types of inputs, for example `textInput()`) of **Shiny** causes `server()` to be executed and the elements of `output` to be updated.

7.2 More elaborate application

7.2.1 Working method

An application is created by choosing:

- A window layout.
- The controls for entering parameters (*intput*).
- The controls for displaying the results (*output*).

The code to process the inputs and produce the outputs is then written to `server()`.

The RStudio tutorial[2] is very detailed and should be used to go further.

7.2.2 Example

This simple application uses the **scholar** package to query Google Scholar and get the bibliometric data of an author from his or her identifier.

The `app.R` file contains all the code and is built incrementally here. The full application, with graphical output in addition to its simplified version presented here is available on GitHub[3].

The beginning of the code consists of preparing the application to run by loading the necessary packages:

```
# Prepare the application ####
# Load packages
library("shiny")
library("tidyverse")
```

The code of the complete application includes a function to install the missing packages, to be executed only when the application is executed on a workstation (on a server, the management of packages is not the responsibility of the application).

The user interface is as follows:

```
# UI ####
ui <- fluidPage(
  # Application title
  titlePanel("Bibliometrics"),

  sidebarLayout(
    sidebarPanel(
      helpText("Enter the Google Scholar ID of an author."),
      textInput("AuthorID", "Google Scholar ID", "4iLBmbUAAAAJ"),
      # End of input
      br(),
      # Display author's name and h
      uiOutput("name"),
      uiOutput("h")
    ),
    # Show plots in the main panel
    mainPanel(
      plotOutput("network"),
      plotOutput("citations")
    )
  )
)
```

The application window is fluid, i.e. it reorganizes itself when its size changes, and is composed of a side panel (for text input and display) and a main panel, for displaying graphics.

The elements of the side panel are:

[2] https://shiny.rstudio.com/tutorial/

[3] https://github.com/EricMarcon/bibliometrics

- A help text: `helpText()`.
- A text input field, `textInput()`, whose arguments are the name, the displayed text, and the default value (an author ID).
- A line break: `br()`.
- HTML output controls: `uiOutput()`, whose single argument is the name.

The main panel contains two graphical output controls, `plotOutput()` whose argument is also the name.

The code to execute to process the inputs and produce the outputs is in the `server()` function.

```
# Server logic ####
server <- function(input, output) {
  # Run the get_profile function only once ####
  # Store the author profile
  AuthorProfile <- reactiveVal()
  # Update it when input$AuthorID is changed
  observeEvent(input$AuthorID,
               AuthorProfile(get_profile(input$AuthorID)))

  # Output ####
  output$name <- renderUI({
    h2(AuthorProfile()$name)
  })

  output$h <-
    renderUI({
      a(href = paste0(
        "https://scholar.google.com/citations?user=",
        input$AuthorID),
        paste("h index:", AuthorProfile()$h_index),
        target = "_blank"
      )
    })

  output$citations <- renderPlot({
    get_citation_history(input$AuthorID)  %>%
      ggplot(aes(year, cites)) +
      geom_segment(aes(xend = year, yend = 0),
                   size = 1,
                   color =
                     'darkgrey') +
      geom_point(size = 3, color = 'firebrick') +
      labs(title = "Citations per year",
           caption = "Source: Google Scholar")
  })

  output$network <- renderPlot({
    ggplot() + geom_blank()
  })
}
```

The information needed for the output fields `$name` and `$h` (author's name and h-index) is obtained by the `get_profile()` function of the **scholar** package. This function queries the author's Google Scholar web page and extracts the values from the result: this is a heavy processing, which is better executed only once rather than twice, in the `renderUI()` functions in charge of computing the values of `output$h` and `output$name`.

The simplest code to do this would be as follows:

```
  # Run the get_profile function only once ####
  # Store the author profile
  AuthorProfile <- get_profile(input$AuthorID)
```

The difficulty with programming a Shiny application is that any computation referring to an input interface element must be *reactive*. If the latter code were executed, the following error message would appear: "Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)"

In practice, the execution of the code is started by modifying an input control (here: `intput$AuthorID`). The code referring to one of these controls must be permanently waiting for a modification: it must therefore be placed in particular functions like `renderPlot()` in the *Old Faithful Geyser Data* application or `renderUI()` here. The following code would run without error:

```
# Output ####
output$name <- renderUI({
    AuthorProfile <- get_profile(input$AuthorID)
    h2(AuthorProfile$name)
})
```

The call to the value of the `input$AuthorID` control does occur in a reactive function (but `get_profile()` would have to be used a second time in the calculation of `output$h`, which we want to avoid). The function `h2(AuthorProfile$name)` produces HTML code, a level 2 title paragraph whose value is passed as an argument.

All functions whose names begin with `render` in the **shiny** package are reactive, and each is intended to produce a different type of output, for example text (`renderText()`) or HTML code (`renderUI()`).

If code is needed to compute variables common to several output controls (`output$name` and `output$h`), it must itself be responsive. Two functions are very useful:

- `observeEvent()` watches for changes in an input control and executes code when they occur.
- `reactiveVal()` allows you to define a reactive variable, which will be modified by the `observeEvent()` code and will in turn cause other reactive functions that use its value to execute.

So the optimal code creates a reactive variable to store the result of the Google Scholar query in:

```
# Store the author profile
AuthorProfile <- reactiveVal()
```

The reactive variable is empty at this point. Its use is then that of a function: `AuthorProfile(x)` assigns it the value `x` and `AuthorProfile()`, without arguments, returns its value. The `observeEvent()` function is triggered when `input$AuthorID` is modified and executes the code passed as the second argument, in this case the update of `AuthorProfile`.

```
# Update it when input$AuthorID is changed
observeEvent(input$AuthorID, AuthorProfile(get_profile(input$AuthorID)))
```

Finally, the `renderUI()` functions that provide output control values use the value of `AuthorProfile`:

```
# Output ####
output$name <- renderUI({
    h2(AuthorProfile()$name)
})
```

Note the parentheses in `AuthorProfile()`, a reactive variable, as opposed to the `AuthorProfile$name` syntax for a classic variable.

The value of `output$h` is an internet link, `<a href=...` in HTML, written by the `a()` function of the **htmltools** package used by `renderUI()`.

```
output$h <- renderUI({
    a(href = paste0("https://scholar.google.com/citations?user=",
        input$AuthorID), paste("h index:", AuthorProfile()$h_index),
        target = "_blank")
})
```

The link is to the author's Google Scholar page. The value displayed is its h index. The argument `target = "_blank"` indicates that the link should be opened in a new browser window.

The `output$citations` graph is created by the `renderPlot()` reactive function. The data provided by the `get_citation_history()` function of **scholar** (which queries the Google Scholar API) is processed by `ggplot()`.

Finally, the `output$network` graph is an empty graph in this simplified version of the application.

The full application takes this code and adds error handling in case the author code does not exist on Google Scholar and the co-author network graph.

7.3 Hosting

A Shiny application is not necessarily hosted by a web server: it can be run on users' workstations if they have R.

For a wider use, a dedicated server is necessary. Shinyapps.io[4] is a service from RStudio that allows to host 5 Shiny applications for free with a maximum uptime of 5 hours per month.

[4] https://www.shinyapps.io/

First of all, you have to open an account on the site, preferably with your GitHub identifiers. To allow the management of online applications directly from RStudio, you must then install the **rsconnect** package and set it up:

```
rsconnect::setAccountInfo(name = "prenom.nom", token = "xxx",
    secret = "<SECRET>")
```

The exact code, along with the username and token to use, are displayed on the Shinyapps.io homepage: click on "Show Secret", copy the code and paste it into the RStudio console to run it. A "Publish" button is available just to the right of the "Run App" button. Click on it and validate the publication (figure 7.2).

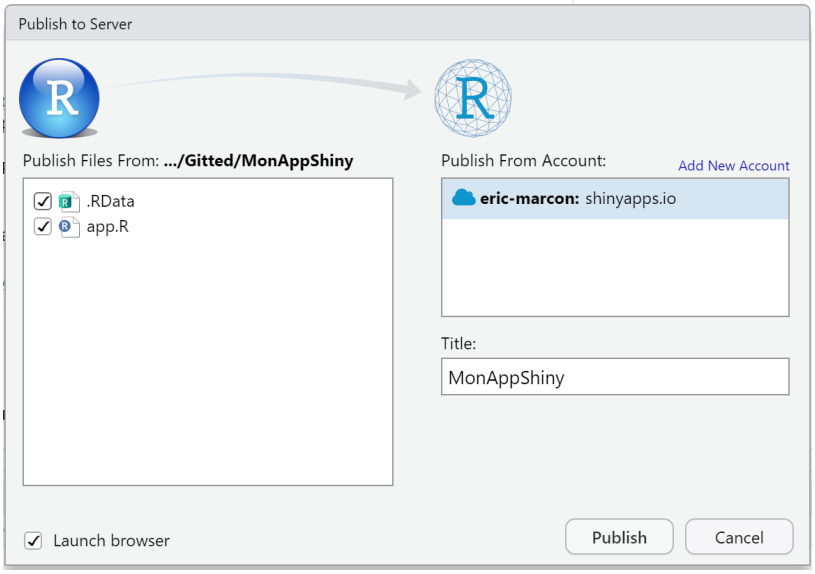


Figure 7.2: Publication of the Shiny application on Shinyapps.io.

The application is now available at https://firstname-lastname.shinyapps.io/MyShinyApp/

The "Bibliometrics" application does not work on Shinyapps.io because the way the **Scholar** package queries Google Scholar is not supported. Most Shiny applications work without difficulty, as long as they don't require complex networking features.

CHAPTER 8

TEACHING WITH R

R, RStudio and GitHub provide tools for teaching.

The **learnr** package allows you to make interactive tutorials.

We will also see how to use *GitHub Classrooms* which allow to distribute to a class (a list of students with a GitHub account) a repository model (a draft of an R project) that each student will have to develop and publish. The classroom tools allow to evaluate the work done quite easily.

8.1 learnr

learnr allows you to make code snippets of any document produced by R Markdown in HTML interactive, by transforming them into Shiny applications. The documentation on the RStudio website[1] is very clear and will not be repeated here: we will only see how to start and how to distribute the tutorials.

8.1.1 First tutorial

With the menu "File > New File > RMarkdown…" create a new document from an "Interactive Tutorial" template. The wizard creates a folder with the chosen name, to be transformed into an R project and put under source control, as for all documents seen previously (see section 4.3.2).

[1] https://rstudio.github.io/learnr/

To run the tutorial, click on the "Run Document" button, which is in the usual place of the "Knit" button.

Tutorials can include exercises, which are code snippets with the `exercise=TRUE` option. These exercises are displayed as a window of code that can be edited and executed by the user. Hints can be given[2], a button added to display the solution, a time limit can be set[3], and both the code and its result can be compared to an expected value[4].

Quizzes[5] can be added, in the form of multiple or single choice quizzes.

The user's progress in the tutorial (code entered, answers to questions…) is saved by **learnr** on the workstation. A tutorial can be stopped and resumed without loss of data. On the other hand, there is no easy way to recover this data for an evaluation by the trainer for example.

8.1.2 Sharing

Tutorials can be distributed by copying the files or by telling users to clone the GitHub projects that contain them.

They can also be hosted on Shinyapps.io (see section 7.3).

Last, they can be included in a package[6].

8.2 GitHub Classrooms

GitHub Classrooms allows to distribute to a student audience GitHub repositories to modify and control the result. The applications are as well the learning of R as the production of documents, for a personal work or an exam for example.

8.2.1 Registration

To start using the tool, you need to open an account. On the GitHub Classrooms website[7], click on "Sign in" and use your GitHub account to authenticate.

8.2.2 Organizations

The next step is to create a GitHub organization. A GitHub organization basically contains members (GitHub account holders) and repositories that can be accessed at https://github.com//.

[2] https://rstudio.github.io/learnr/exercises.html#Hints_and_Solutions
[3] https://rstudio.github.io/learnr/exercises.html#Time_Limits
[4] https://rstudio.github.io/learnr/exercises.html#Exercise_Checking
[5] https://rstudio.github.io/learnr/questions.html
[6] https://rstudio.github.io/learnr/publishing.html#R_Package
[7] https://classroom.github.com/

The simplest way to work is to create an organization per course, but other approaches are possible in structures that use the tool extensively. The organization created for the example here is "Cours-R"[8].

An email address is required (use the same as that of your GitHub account) and the organization must be declared as belonging to your personal account.

If the organization is not visible on the GitHub Classrooms page, click on "Grant us access".

8.2.3 New Classroom

A classroom is populated with students who will receive tasks to perform.

Click on *New Classroom*. Select the organization in charge of administering the classroom.

Enter the name of the classroom: a good practice is to prefix it with the name of the course and add the name of the session, for example "Cours-R-2020".

Do not add collaborators (this will be possible later), and eventually enter the list of students (one name per line, also possible later). The class is created.

All classrooms are visible from the GitHub Classrooms homepage[9]. Click on a name to open one. The "Settings" button allows to change its name or to delete it. The "TAs and Admins" button allows you to add collaborators, i.e. other GitHub users who will be able to administer the classroom.

The "Students" button allows to add students. The list of names is free, with no mandatory format. Click on "Create Roster" to activate it. The names must then be linked to GitHub accounts: this work can be done by the administrator or by the students themselves when they receive the first task to be done. Each student must have an account on GitHub.

8.2.4 Prepare a repository template

A task is a GitHub repository to modify. For example[10], create a repository containing an R project with a Markdown file describing the work to be done and possibly some of the code needed to do it, the other files in the R Markdown template used and a data file.

Open the repository properties on GitHub and check the *Template Repository* box to make it a template.

Assign a task

Open a classroom and click on "New Assignment".

Enter an explicit title for the students, an optional deadline and choose "Individual Assignment".

[8] https://github.com/Cours-R

[9] https://classroom.github.com/classrooms

[10] https://github.com/EricMarcon/Cours-R-Memo/settings

By default, the assignment name is used as a prefix for the students' submission names, but it can be replaced by a prefix of your choice. When students turn in their work, all repositories for all tasks will be stored in the organization.

The repository created on each student's account can be private or public, depending on whether you want students to be able to see each other's work or not. Give the administration right and make the site public if the students should be able to activate GitHub pages to present the results of their work. Click on "Continue".

Select the model repository (*starter code*) then click on "Continue" then "Create Assignment".

The new task is created. It is associated with an invitation link that must be copied and sent to the students. When they click on the link, they will reach a GitHub page that will allow them to associate their account with a name in the list (no control is possible: the first one connected can associate with any name). They will then be able to create a new RStudio project from the GitHub repository automatically created by GitHub Classrooms, modify that project according to the work instructions and push it to GitHub. The repository is on the account of the organization the class is connected to, and is suffixed with the student's GitHub ID.

Controlling student work

It is possible to view each repository created by students from the assignment page on GitHub Classrooms. If the assignment to be produced is a written document, have students place it in the repository's GitHub pages to read it directly online.

The GitHub Classrooms wizard[11] allows you to download all student repositories at once to correct them on your workstation.

[11] https://classroom.github.com/assistant

CHAPTER 9

CONCLUSION

The R and RStudio work environment allows for the production of all types of documents with a single language.

The objective of reproducibility of results is achieved by integrating statistical processing and writing. Collaborative work is allowed by the systematic use of source control and GitHub. The presentation of results is ensured by GitHub pages and document templates covering most needs.

For breaks, R even provides some games in the **fun** package, including the famous Minesweeper:

```
# Install the package
install.packages("fun")
# Open an X window and run
if (interactive()) {
    if (.Platform$OS.type == "windows")
        x11() else x11(type = "Xlib")
    fun::mine_sweeper()
}
```

This document does not aim to be exhaustive on the possibilities of R but rather to present a working method and simple ways to apply it quickly. You can refer to the more detailed books and documentations cited in the text to go deeper into a particular point.

It is updated regularly as the available tools evolve.

Bibliography

Gandrud, C. (2015). *Reproducible Research with R and RStudio*. 2nd ed. Chapman and Hall/CRC (cit. on p. ix).

Gillespie, C. and R. Lovelace (2016). *Efficient R Programming*. O'Reilly Media. URL: https://csgillespie.github.io/efficientR/ (cit. on p. 13).

Knuth, D. E. (1984). "Literate Programming." In: *The Computer Journal* 27.2, pp. 97–111. DOI: 10.1093/comjnl/27.2.97 (cit. on p. 76).

Wickham, H. (2010). "A Layered Grammar of Graphics." In: *Journal of Computational and Graphical Statistics* 19.1, pp. 3–28. DOI: 10.1198/jcgs.2009.07098. URL: http://vita.had.co.nz/papers/layered-grammar.pdf (cit. on p. 19).

— (2014). *Advanced R*. Chapman and Hall/CRC. URL: http://adv-r.had.co.nz/ (cit. on p. 13).

— (2015). *R Packages*. 1st ed. O'Reilly Media, Inc. (cit. on p. 114).

— (2017). *Ggplot2: Elegant Graphics for Data Analysis*. 2nd ed. Springer. DOI: 10.1007/978-3-319-24277-4. URL: http://had.co.nz/ggplot2/book (cit. on p. 20).

Wickham, H. and J. Bryan (2023). *R Packages*. 2nd ed. O'Reilly Media, Inc. URL: https://r-pkgs.org/ (cit. on p. 145).

Wickham, H. and G. Grolemund (2016). *R for Data Science*. O'Reilly Media. URL: http://r4ds.had.co.nz/ (cit. on pp. 13, 20).

Xie, Y. (2015). *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman and Hall/CRC. DOI: 10.1201/b15166. URL: https://yihui.name/knitr/ (cit. on p. 76).

Xie, Y., J. Allaire, and G. Grolemund (2018). *R Markdown: The Definitive Guide*. Boca Raton, Florida: Chapman and Hall/CRC. URL: https://bookdown.org/yihui/rmarkdown (cit. on p. 76).

List of Figures

Abstract This book proposes an organization of work around R and RStudio to, beyond statistics, write documents efficiently with R Markdown, in various formats (memos, scientific articles, student theses, books, slideshows), create websites and online R applications (Shiny), produce packages and use R for teaching.

.