Good practices guidelines in R projects

1. Rational & scope of the guideline 2
2. Note on text editors / IDEs 2
3. One folder, one project mentality 3
4. File system structure 4
5. Programming 7
6. Documenting and managing dependencies 9
7. Version control 10
8. Bibliographie / ressources 14

Authors: Grégoire Falq and Mathilde Mousset

1. Rational & scope of the guideline

The handover of a programming project from one programmer to another is often a challenge, and R projects at Epicentre are no exception to the rule. The challenge arise from differences in knowledge, in working habits, lack of time…

This guideline presents a set of good computing practices that **every R user in Epicentre should adopt regardless of their current level of computational skills**. The idea being to have a 'common denominator' in the R working habits. These practices, which encompass organizing the file system structure (directory structure), organizing programs and programming, are drawn from discussions with R users in Epicentre and from key literature (such as *Good enough practices in scientific computing*, Wilson).

Most of the good practices presented here are easy to implement, just necessitating a change of habits. Other tools, such as version control take a bit more time to learn. **We propose minimal good practice**, but also describe some more advanced tools that you need to know exist, so that you may choose to invest some time learning them in the future.

The aim of this document is to make cleaning, analysing and reporting or publishing data more **reproducible**, **robust**, **transparent**, **reusable** and **shareable**.

What's in there for me?

Following reproducibility / programming good practices helps with:

- Reducing the risk of performing an analysis on the wrong version of data (such as a not up-to-date file)
- Reducing the hassle when taking over someone else project or analyses, especially if the person is not around anymore
- Not cursing the 'you from two years ago', who did not document their analysis well enough, and now you don't find the same results anymore with the same data. Or are they the same data?
- Making sure your code is doing the things it should be doing and let other to check it does
- Reusing bits of your code for future projects and saving time
- Reducing the risk of making mistakes in data cleaning and analysis
- ...

We recommend R users in Epicentre to get familiar with section 3 (One folder, one project mentality), 4 (File system structure) and 6 (Documenting and managing dependencies). Section 5 (Programming) is more a tool-box that could help you to improve your code readability. Section 7 is a short introduction to the concept of version control.

2. Note on text editors / IDEs

It is generally advised to use a good text editor, or even an integrated development environment (IDE) to work with R, as these tools can bring useful options, such as syntax highlighting, advanced searching functions, keyboard shortcuts, debugging features etc.

Many good options exist for text editors or IDE able to handle R (potentially through an extension or two). People routinely use Rstudio, Tinn-R, VSCode, Eclipse, Vim, Emacs, SublimeText, Notepad++, Atom etc. (for more examples, see here).

If you do not use such a tool, we strongly advise you to invest some time learning one, as it will make your life easier. Do not hesitate to try a couple, or to see if your already favourite editor can accommodate R. Anything is better than the basic R GUY or notepad and there are many good options depending on your needs.

If you are unsure what to pick, it is safe to give a try to Rstudio, as it is widely used, well documented, easy to install, and beginner friendly.

3. One folder, one project mentality

In answer to: a script called a data file in another directory, which was not transmitted to me...

It is advised to keep all data and scripts related to a given project together in a dedicated directory with a descriptive name and a READ ME file when possible. This **one folder = one project** mentality makes the project **self-contained** and **portable**, i.e. it can be moved from computer to computer without losing parts of the scripts or data.

In answer to: I changed computer / someone gave me their analysis folder and I spent hours changing all the paths scattered all around in several scripts.

Several easy-to-enforce steps help with this:

1) **Using relative/internal paths** instead of absolute paths within the project directory is a **must-do**, so that paths to data and other scripts remain valid even when the project is moved to another computer (or somewhere else in the same computer). This enables easy communication with other R users. Note that the base R function *file.path()* allow creation of paths that will work on any operating system (i.e. *use file.path("home", "my_project", "data")* rather than *"home/my_project/data"* or *"home\\my_project\\data"*).
2) **Defining all paths that are needed in a script at the beginning of said script**, in a dedicated section, so that someone needing to change the paths has only one place to look for in each script. Even better, **having a "set-path.R" script** sourced by other scripts allows to have all paths in one place, which makes it fast to find and adapt.

With these easy and fast to implement steps, one has only to modify the path to the working directory and set it to the root of the project directory once, in the set_path.R script for example, and all the relative paths will work relative to this one absolute path.

Some tools can help with these steps. The here package allows to easily find the root of a given project, as well as constructing operating system independent paths. **If you are using Rstudio, Projects in Rstudio** have by default their working directory at the root of the project. If one uses relative paths in scripts, one just need to give the directory containing the Rstudio project to someone and they should be able to use the scripts directly. If you want to know more, here is a very short introduction on relative vs absolute paths, and Rstudio projects. Using here, especially in conjunction with Rstudio projects helps creating truly portable, self-contained projects.

In answer to: I need to import from a special directory shared with everybody working on these data.

Sometimes, despite our best intentions, it is not possible to have a fully self-contained directory for all the analysis. An example that many people in Epicentre will encounter is having to do an analysis (one project!) on data generated in the department and stored in a common Sharepoint.

For example, performing weekly analyses for Ebola monitoring and generating a sitrep: the data are usually updated, cleaned and shared every week in a shared folder which is distinct from the sitrep folder.

Even in this case, using the previous principles (gathering all path definitions in an easy-to-find script or place, and using relative paths in the rest of the project) will still greatly reduce the time needed for a new person to get going: they will need to modify the path to the Sharepoint library, and the path to their working directory (if *here* is not used) but that's all. It is advised to leave information on the Sharepoint and how to access it/who to ask for access in the comments or the README.

4. File system structure

4.1. General project structure

There is no single best way to organize a file system. The key is to make sure that the **structure of directories and location of files is consistent and informative**, and that input data is clearly different from outputs.

We *propose* here a template of file system structure in order to have a common basis for every R user in Epicentre.

- **0-R**: put scripts to work on your data. You may have specific subfolders for markdown or officer scripts (for instance).
    - **R-markdown**: put rmarkdown scripts to work on your data
    - **R-officer**: put officer scripts to create reports form your data
- **1-data-raw**: put raw data and metadata (except geographical data)
    - ⇨ Save data as originally generated (i.e. from a survey). It is tempting to overwrite raw data files with cleaned-up versions, but **faithful retention is essential for rerunning analyses from start to finish**, for recovery from analytical mishaps, and for experimenting without fear. Consider changing file permissions to read-only or using spreadsheet protection features so that it is harder to damage raw data by accident or to hand edit it in a moment of weakness.
- **2-data-geo**
    - raw: put raw shape files data. Same comments as 1-data-raw folder
    - gen: put generated shape files data
- **3-data-clean**: put files generated after data cleaning or intermediate data processing, such as cleaned data sets or simulated data
    - csv
    - excel
    - RData

   - RDS
   - …
- **4-output**: put files generated after data processing, such as final results (figures, tables, maps…).
   - csv
   - excel
   - img
   - …
- **5-report**: save reports in the appropriate format folder and 'elements' used to generate reports such as images (MSF/Epicentre banner), template (word template for officer…) in the asset folder.
   - assets
      - css
      - img
      - template
   - html
   - pdf
   - word
- **library-user**: put all the packages required for the project. You may also use renv package to manage dependencies (see section 6).

A template of directory structure is available on Epilink in "Toolbox 'R' Bonnes pratiques".

A **README file** should be at the root of the project, explaining at minima who work on this project, descript the projects (where data comes from, what, broadly is in there and where), who to contact and how to modify the project.

Note: for project stored as a package, or Shiny projects, the organization may differ markedly from the above, which is more "data analysis" centered.

4.2. The R subdirectory: organizing one's scripts

Split your scripts into logical thematic units. For example, you might separate your code into scripts that load data, clean data, analyse data, and produce outputs like figures, tables or report.

Most R directories have these or a subset of these scripts:

- **0.1_master.R** or **0.1_the_lord_of_the_scripts.r**: contains codes to source all other programs
- **0.2_helpers.R/0.2-utils.R/0.2-functions.R**: contains user functions for this project. You may create thematic functions files for ggplot, shiny, report generation...
   - **0.2.1_utils.R**: put general functions
   - **0.2.2_utils_cleaning.R**: put functions to clean raw data
   - **0.2.3_utils_report.R**: put functions to design report (r-markdown, officer…)
   - **0.2.4_utils_ggplot.R**: put functions that are called to create ggplot objects
- **0.3_library.R**: load all libraries required for the project.
- **0_setup.R**: there's also the possibility to have a unique program that loads the necessary libraries, users functions, and lists the paths of data folders.

- **1_import_data.R**: contains codes to import raw data and possibly to clean data if data are not too dirty (i.e. national surveillance data) otherwise cleaning data can be done in a separate file.
- **2_clean_data.R**: contains codes to clean data and finally to export cleaned dataset in a Rdata/rda file (folder 3-data-clean).
- **3_data_prepa.R/3_summarize_data.R/3_data_munging.R** contains codes to create indicators useful for data analysis (usually data.frame/flat table).
- **4_output.R / 4_analysis.R**: contains codes to create tables, figures and maps, basically all you need for your report. This may split this program:
    - 4.1_output_table.R
    - 4.2_output_figure.R
    - 4.3_output_map.R
- **5_report_name.Rmd**: markdown program to generate a specific report.
- **5_report_name.R**: officer files to generate report.

4.3. Saving code, not workspace

An important habit to take is to **not save the environment** (history, loaded packages, objects in the environment). The idea of a reproductible project is that everything can be made again by the scripts in a new session, on another computer. The corrolary of that is that it is important to restart R regularly, to make sure that the analyses are fully reproductible.

Important objects or figures should be explicitly saved into the project folders as opposed to implicitly kept in the environment or saved by the mouse.

Objects that are computationally intensive to generate can be produced in their own scripts and saved using *saveRDS(my_precious, here("results", "my_precious.rds"))* and imported in downstream scripts via *my_precious <- readRDS(here("results", "my_precious.rds")).*

And do not attach objects. **Never**.

From https://github.com/jennybc/debugging#readme:

"Renouncing .Rdata and restarting R often are not intrinsically important or morally superior behaviours. They are important because they provide constant pressure for you to do the right thing: save the source code needed to create all important artefacts of your analysis."

A good way to enforce this is by setting the default Rstudio global options to not save the .Rdata when you close R.

5. Programming

For data management we recommend to use dplyr, tydir, magrittr packages (piping operator %>%) that allow readable and powerful programming.

5.1. Comment your code

Challenge: How often have you revisited an old script six months down the line and not been able to figure out what you had been doing? Or have taken on a script from a collaborator and not been able to understand what their code is doing and why?

An easy win for making code more readable and reproducible is the liberal, and effective, use of comments.

One good principle to adhere to is to **comment the 'why' rather than the 'what'**. The code itself tells the reader what is being done, it is far more important to document the reasoning behind a particular section of code or, if it is doing something non-standard or complicated, to take some time to describe that section of code.

Good practice :-)

- Comment your code
- Comment your code *intelligently* 😊
- Break down your scripts in titled sections and navigate or fold code more easily.
- Provide a plain text description of the code, using numbered steps, and then refer to these steps in the code using comments
- Document the inputs and outputs of each function/method, using comments or, even better, the roxygen2 package (even if you are not creating a package).
- Use ## for comments, rather than a single # which are indented differently on different code editors

A different way to document code, which might be very pertinent in many cases is to use literate programming which allows to mix code and text. Options for literate programming include Sweave (R+LateX), and more recently the wonderful Rmarkdown.

5.2. Naming

Consistent and predictable naming helps streamline writing and reviewing code, and also increases readability: *"Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread*" (from the tidyverse style guide).

Most programming languages have official naming conventions, official in the sense that they are issued by the organization behind the language and accepted by its users. This is not the case with R. There exist many different naming conventions and below is a list of some of the most common. All are in use in the R community and the example names given are all from functions that are part of the base package.

5.2.1. Most common naming convention

We recommend using the following conventions, already used to a large extent in package development:

underscore_separated

All letters are lower case and multiple words are separated by an underscore as in seq_along or package_version. This naming convention is used for function and variable names in tidyverse packages, as well as in many languages including C++, Perl and Ruby.

period.separated

All letters are lower case and multiple words are separated by a period. This naming convention is unique to R and used in many core functions such as as.numeric or read.table.

lowerCamelCase

Single word names consist of lower case letters and in names consisting of more than one word all, except the first word, are capitalized as in colMeans or suppressPackageStartupMessage. This naming convention is used, for example, for method names in Java and JavaScript.

UpperCamelCase

All words are capitalized both when the name consists of a single word, as in Vectorize, or multiple words, as in NextMethod. This naming convention is used for class names in many languages including Java, Python and JavaScript.

While several conventions currently exists for R, the tidyverse style guide seem to be used by many people, and we advise that you read it once: https://style.tidyverse.org/

5.2.2. Good practice, independent of convention

- Name your variables explicitly
- The same name should never have two uses
- Look for a compromise: the variable names must be of a reasonable size... while remaining explicit (ex: l < lethality < number_of_death_over_total_cases)
- Never use special characters in file names, variable names or values such as éÈçôï\/#%?!&:;,@*^ and blank spaces
- If you really need to, only use special characters when defining labels for graphics or tables
- For dates: use the format yyyy-mm-dd, e.g. 2001-10-13 for the 13th October
- For encoding: **use UTF-8 encoding whenever possible**, as it guarantees that special characters will display correctly on different computers

5.3. Writing simple code

Writing simple code is essential for enabling others (and in fact, yourself) to read and understand your code, identify possible issues, or propose improvements. Good code should be simple to read and understand. Here are some guidelines for producing simple, readable code:

Good practice :-)

- use indentation to make the code readable
- write short lines (ideally less than 80 characters); many short lines are much easier to read than a single, long line of code
- start a new line after each piping operator %>%

Rtudio tip: select a code portion and type Ctrl+ I to automatically reindent code.

5.4. Architecture of the R code

Good practice :-)

- Two separate instructions should be on two separate lines
- Lines should be indented to highlight the blocks making up the code
- Each closing bracket should be vertically aligned with the instruction defining the corresponding opening bracket
- Instruction blocks or functions should be separated by lines
- Do not omit optional brackets
- All conditions should have an else, even if it is empty

5.5. Architecture of the Rmd

An Rmd file (R Markdown file) is file that mixes text and R code. It contains YAML metadata, markdown-formatted plain text, and “chunks” of R code. When rendered (or knitted), the Rmd file produces sophisticated data analysis document or reports in different format (html, doc, pdf).

It is a good practice to let the first chunk to setup the script (for example set the input/output paths and set the default options for the rendering), and the second chunk to upload and prepare the datasets required for the analysis. The analysis starts only at the third chunk. Follow the rule one chunk = one output (a table of a graph or a map). Equally important is to ensure that all chunks from the third to the last one, are independent from each other. This means that a chunk should produce the output even if you did not run the previous chunks (apar from the first two). This approach facilitates the collaboration with your colleagues on the same Rmd file. A colleague may work in a chunk while you are working in another one. If you see that you repeat the same lines in different chunks, consider to put the lines in the second chunk, or (even better) to create a function.

5.6. Miscellaneous

A few more good practice :-)

- Do not use abbreviations
- Never use a global variable. **Never**.
- Test your code regularly, don't write a long code to test it only at the end, it makes debugging very difficult
- Don't try to write optimal code. Write clear and simple code. Later, when your code is up and running, much later (maybe never), it will be time to think about optimization.
- Loops in R can often be replaced with functions from the *apply* family (base R) or the *map* family (*purr package*).

6. Documenting and managing dependencies

This *paragraph* is taken from ‘A Guide to Reproducible Code in Ecology and Evolution’ (page 18), British Ecological Society and authors, 2017 (Guide_BEC).

Reproducibility is also about making sure someone else can re-use your code to obtain the same results as you.

For someone else to be able to reproduce the results included in your report, you need to provide more than the code and the data. You also need to document the exact versions of all the packages, libraries, and software you used, and potentially your operating system as well as your hardware.

R itself is very stable, and the core team of developers takes backward compatibility (old code works with recent versions of R) very seriously. However, default values in some functions change, and new functions get introduced regularly. If you wrote your code on a recent version of R and give it to someone who has not upgraded recently, they may not be able to run your code. Code written for one version of a package may produce very different results with a more recent version.

Documenting and managing the dependencies of your project correctly can be complicated. However, even simple documentation that helps others understand the setup you used can have a big impact. The following are three levels of complexity to document the dependencies for your projects.

Show the packages you used With R, the simplest (but a useful and important) approach to document your dependencies is to report the output of sessionInfo() (or devtools::session_info()). Among other information, this will show all the packages and their versions that are loaded in the session you used to run your analysis. If someone wants to recreate your analysis, they will know which packages they will need to install.

Use packages that help recreate your setup. The checkpoint package in R provides a way to download all the packages at a given date from CRAN (The Comprehensive R Archive Network, cran.r-project.org). Thus, from the output provided by sessionInfo(), they could recreate your setup. It, however, makes two important assumptions: all your packages were up-to-date with CRAN at the time of your analysis; and you were not using packages that are not available from CRAN (e.g. the development version of a package directly from a git repository).

The current way to manage packages seems to be the *renv* package. The renv package is a new effort to bring project-local R dependency management to projects. Underlying the philosophy of renv is that any of your existing workflows should just work as they did before – renv helps manage library paths (and other project-specific state) to help isolate your project's R dependencies, and the existing tools you've used for managing R packages (e.g. install.packages(), remove.packages()) should work as they did before.

See here for a very short tutorial.

7. Version control

The paragraphs in italic are taken from 'A Guide to Reproducible Code in Ecology and Evolution' (page 28), British Ecological Society and authors, 2017 (Guide_BEC).

Scripts for reproducible code evolve over time, so keeping track of the changes that you make is important. You might need to access a specific version of your code to verify or reproduce the analysis used for a published paper, or to identify where you introduced a change that has caused problems in your code at a later stage.

Anyone who has wrestled with multiple versions of a document or script named by appending the word "final" will know how quickly such naming conventions can escalate into absurdity.

Version control provides a structured and transparent means of tracking changes to code and other files. It was designed for use in software development and it is equally applicable to scientific programming. By ***recording snapshots of a project at successive points in time, you can create a record of your project's development while keeping your workspace clean****. Version control also facilitates collaboration when used within project teams or when contributing to open source software projects.*

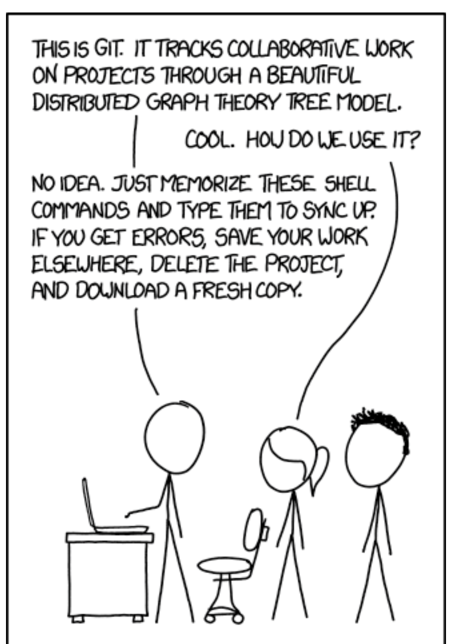
Git *is a good choice because it offers flexibility in how you can use it for version control. You can use it as a* ***stand-alone tool to manage files on your own computer*** *and, optionally,* ***you can use it to connect to an external service to archive your files****. Git can be used directly or through other software such as RStudio, so it is easily integrated into your normal workflow.*

Git is a distributed version control system, which means that each user interacts with a stand-alone copy of the versioned files. This stand-alone copy is called a repository and it is usually a folder on your computer that contains all the work for a particular paper or project. Individual repositories are synchronised with each other by exchanging information about what changes have been made.

Most of the good practice in this document are relatively easy to implement. They either require learning moderately simple new tools (project mind frame, sourcing a document, or

the two functions to use the *here* package), or require building new but simple habits of using better conventions, which pay off at relatively short time scales.

Version control is slightly different in the way that most people without a programmatic background (i.e. most of us here at Epicentre and in quantitative sciences in general) find it scary at first. Git currently is considered one of the best tools to do version control, but because it helps solving complex challenges (keeping tracks of versions, changes, and even work collaboratively on documents), it has a steeper learning curve than the rest of this document.



However, like the rest of the good practices (or like learning R), learning and using git is a gradual process and one can learn it a couple of functions at a time, with new pay-offs associated with each steps:

1. **Learn how to use a local directory and track changes on your own computer**, without collaborating with other people, without connecting to a hosting service (such as Github). This involve installing Git and learning about five commands. => learn tracking you own changes in a stressless situation (and benefit from being able to go back in your history).
2. **Learn how to set up an online directory linked to your local directory** (on Github/Gitlab/Bitbucket…), and how to synchronize it with your local folder (three more commands). => Learn the two commands to synchronise stuff online and get an extra backup independent from Sharepoint or your computer. If you change computer, clone the repository and get the exact same scripts as the original.
3. Learn how to clone someone else's directory and synchronize some changes with branches. => learn how to collaborate with others.

Practicing each of these steps will bring something to your project. If you are not working on a project where collaborating with people through Git is compulsory, **it is actually a good time to get confident using it** in solo before diving in collaboration.

What softwares and website do you need:

- **Git** is the version control system. You can use it locally on your computer or use it to synchronize a folder with a host website. By default, one uses a terminal to give Git instructions in command-line.
- You can use a **Git client** to avoid the command-line and perform the same actions ((at least for the simple, super common ones). Many options exist: Github Desktop, Tortoise Git, Source Tree, GitKraken the Git add-in in Rstudio. Quick explanation on Git clients.
- If you want to store your folder in a host website to collaborate with others, you may create an account at Github, Gitlab, Bitbucket (and others).

Mathilde's PepTalk

You know what else has concepts to learn, obscure native help pages, thousands of functions, thousands of additional modules, a dedicated search engine, frustrating and esoteric error messages, and the tolerance of a hard rock towards your typos?

R.

If you've learn enough R to have to read this guide, then you definitely can learn a couple of Git functions.

As for R (an most of the things, really), you do not have to become an expert in Git to begin using it and rip some benefits out of it!

Ressources to learn Git

Basic concepts:

- https://www.freecodecamp.org/news/an-introduction-to-git-for-absolute-beginners-86fa1d32ff71/ : good for learning the absolute basics to track changes in one folder on you own computer.
- https://speakerdeck.com/alicebartlett/git-for-humans

Tutorials covering both basic and more advanced subjects:

- https://tutorialzine.com/2016/06/learn-git-in-30-minutes
- https://dzone.com/articles/git-tutorial-commands-and-operations-in-git
- https://swcarpentry.github.io/git-novice/ (short course)
- https://rsjakob.gitbooks.io/git/content/chapter1.html

Cheat sheet : https://github.github.com/training-kit/downloads/github-git-cheat-sheet/

Manual versioning

If today is not the day you have the time to learn Git (or Subversion or Mercurial, other well-known versioning tools), the minimal good practice should be to do manual versioning of your projects as described by

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510#sec014 :

1) **Add a file called CHANGELOG.txt to the project's docs subfolder (5f)**, and make dated notes about changes to the project in this file in reverse chronological order (i.e., most recent first). This file is the equivalent of a lab notebook and should contain entries like those shown below.
2) **Copy the entire project whenever a significant change has been made (5g)** (i.e., one that materially affects the results), and store that copy in a subfolder whose name reflects the date in the area that's being synchronized. This approach results in projects being organized as shown below.

8. Bibliographie / ressources

https://martinctc.github.io/blog/rstudio-projects-and-working-directories-a-beginner's-guide/: good intro on structuring R projects (reading < 10min)

https://r4ds.had.co.nz/workflow-projects.html#paths-and-directories: chapter of R for data science on R projects and workflow (reading <15min)

https://www.britishecologicalsociety.org/wp-content/uploads/2017/12/guide-to-reproducible-code.pdf: introduction to reproductible analyses. Do not be fooled by the title, the good practices described in this guide are as applicable in social sciences or epidemiology as they are in ecology and evolution.

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510#sec014: Good Enough Practice for Scientific Computing: “This paper focuses on these first accessible skills and perspectives—the "good enough" practices—for scientific computing: a minimum set of tools and techniques that we believe every researcher can and should consider adopting.”