Reproducible code

Draft

April 2019

There are many many different posts on reproducible workflows. This document collects the current resources avaliable in R and RMarkdown.

# Report writing reproducible reports

Since the development of R and RStudio (and a magnitude of other IT changes happening at the same time) there are now tools for working with issues to do with [reproducibility](https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970). Leading ecological journals have also identified this as an key issue in many conservation and enviromental issues (*cite*)

# Key References

# Report options

## What’s a reproducible report?

For the purposes of this guide, a report is a scientific document that contains not only the text that makes up the manuscript, but also the code that generates the figures and the statistics that are reported in your manuscript. Ideally, the report is part of a self-contained project that may contain your data, your initial exploratory analyses, the final product, and the code needed to generate them.

This manuscript can be a scientific article, a conference presentation, a technical report, or a document to share your progress with your collaborators. The end product may not show any code and therefore it may not look like it was generated differently from other documents.

Typically a report contains code for data manipulation, data analysis, and figure generation alongside the text that constitutes the heart of the report. Because of this hybrid nature, if left unchecked, this mix can lead to a big mess that can be difficult to maintain and debug. In this guide, we will provide you with some advice on how to keep your report manageable.

Box xx: What is the difference between repeatability and reproducibility?

**Repeatability** describes how close are the results of an experiment conducted under the same conditions (same instruments, same operators, etc.). **Reproducibility** describes how close are the results of an experiment conducted under similar but different conditions. Repeatability ensures that you would obtain similar results when running your code on your own laptop at different times; while reproducibility ensures that giving your code to someone else would allow them to obtain the same results as yours.

## Why a reproducible report?

Did you ever have to redo an analysis 6 months later, and it was difficult. You forgot which one of the 15 files with “final” in their names was really the one you should have used? Have you ever spent several hours assembling an intricate figure with your favorite drawing program, just to realize that your collaborators had forgotten to send you the latest batch of data? Writing a reproducible report alleviates some of these hurdles. By automating how the figures and the statistics in your report are generated, you are leaving a code trail that you, your collaborators, or your readers can take, and that leads to your original data. This path to the raw data increases the transparency of your science. However, in order for the six-month-in-the-future you, your collaborators, and your readers, to be able to take this path, it is important that you organize your code and your data files consistently.

Not only does writing a reproducible report increase the transparency of your science, it reduces the mistakes that result from copying and pasting across software. Keeping the content of your manuscript in sync with the output of your statistical program is challenging. By specifying directly the output of your model in your text, it is easier to make sure you are referring to the correct model with the correct parameters. To be the devil’s advocate, one could argue that the additional code that will need to be written to integrate the results within the text could lead to additional errors. However, these bugs are possible to detect (contrary to mistakes done by copying and pasting the correct numbers), and their consequences can be assessed by re-running the code generating the manuscript after fixing them.

Writing a reproducible report allows you to tell a much richer story than the narrative in the report by itself does. The text in your report does not usually show the different approaches and analyses you have tried before coming up with the final results. With a reproducible report, you can provide readers who want to know more about how you obtained the results in your paper, the approaches you tried and the their results. These can be included as supplementary material or tagged in the history of your version control system.

To make your report reproducible, your code will need to be self-contained. As a consequence, you will be able to re-use the code you wrote for one project in another one. Therefore, if initially it might slow you down to make your code reproducible, it is an investment in the future as you will be able to re-use and build upon it in the future. Additionally, others might be able to also re-use your code, and apply it for their own data. Your efforts may speed up the overall scientific process (you or your colleagues won’t need to re-invent the wheel for each project), and you could get more citations on your papers.

It can feel daunting to get started with writing a reproducible report because of the technical skills and knowledge required. However, a partially reproducible report is better than a non-reproducible one. So each step you take towards reproducibility is worth taking, and sets you up to take the next one for the next project.

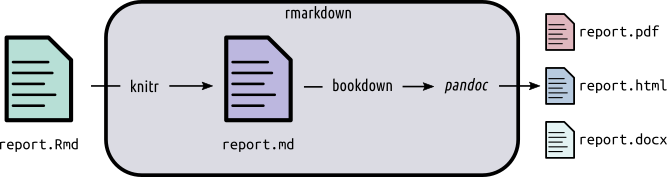
## How to do a report using RMarkdown?

Programming languages typically used by scientistics for data analysis have libraries or packages that can be used to generate reproducible reports. The most popular ones are Jupyter Notebooks for scientists who primarily use python and RMarkdown for those who use R. While they both share many commonalities, their implementation and everyday applications differ. Here, we focus on RMarkdown.

RMarkdown is a file format (typically saved with the Rmd extension) that can contain: a YAML header (see next section), text, code chunks, and inline code. The rmarkdown package converts this file into a report most commonly into HTML or PDF.

The rmarkdown package automates a multi-step process (Fig. xx). Under the hood, it calls the knitr package that converts the Rmd file into a markdown file. In the process, knitr takes all the code chunks and the inline code, run them through R (or other programs), capture their output, and incorporates them in the report. Afterwards, rmarkdown calls the pandoc program (it is an external program that is not related to R) that can take the markdown file and converts to a variety of formats. For pandoc to generate PDF files, you will need a functional [installation of LaTeX](https://www.latex-project.org/get/) that you will need to install separately.

The bookdown package comes in to take care of numbering the figures and tables, as well as dealing with citations. As its name suggests, this package can be used to author books, but it is also well-suited to help generating reports.



Relations between the different R packages and tools used to generate reports from RMarkdown files

### the YAML header

The YAML header is at the top of your file, it is delineated by three dashes (---) at the top and at the bottom of it. It is optional, but can be used to specify:

* the characteristics of your document: the title, authors, date of creation.
* the arguments to pass to pandoc to control the format of the output as well as additional information such as the bibliography file and the formatting of the list of references.
* parameters for your report: for instance, you can specify a parameter such that your report will only use a subset of your data so the final product will be generated quickly when you are developing the code for your project. Once your code is working, you can switch to the full dataset.

### Code chunks

Code chunks are interspersed within the text of the report. They are delineated by three backticks (```) at the top and at the bottom of it. The top backticks are followed by a curly bracket that specify: (1) the language in which the code chunk is written, (2) the name of the chunk (optional but good practice), (3) knitr options that control whether and how the code, the output, or the figure are interpreted and displayed. Everything that comes after the name of the chunk has to be a valid R expression: the strings need be quoted, the arguments are separated by commas, and logical values (TRUE/FALSE) need to be capitalized.

### How to deal with figures?

The knitr package provides many options to finely control how your figures are going to be generated. Some of knitr’s options can be set individually for each chunk or be set globally. For a reproducible report, it is common practice to have chunk at the beginning of the report that sets default options for the figures. It is also usually a good place to load all the packages you will need for your analysis. For instance the following chunk will do the following:

* all the figures generated by the report will be placed in the figures/ sub-directory
* all the figures will be 6.5 x 4 inches and centered in the text.

Additionally, this chunk will be named figure-setup, and we use the echo=FALSE option so the code for the chunk will not be displayed in the report, and use the include=FALSE option so no output produced by this chunk will be included in the report.

For our figures, we can now do

![Relation between sepal width and length in three species of \_Iris\_.](figures/sepal-width-length-1.png)

When this file will be processed, it will create an image file (figures/sepal-width-length.png) with the default dimension and the caption specified by the value of the fig.cap argument. You can use markdown formatting within the captions of your figures. This figure will have the label fig:sepal-width-length that we will be able to use for cross referencing (see below).

If you wish to incorporate a figure that is not generated by code (a photo of your field site or study organism), using the function knitr::include\_graphics() takes care of many details for you, and generates labels and captions as if it was generated by code.

![](figures/iris.jpg)<!-- -->

### How to deal with tables?

To generate tables, knitr comes with the function kable that might be sufficient to make simple tables to represent data frames within your report. However, there are many packages that provide more sophisticated approaches to display and format tabular data within your reports. This [page](https://hughjonesd.github.io/huxtable/design-principles.html) provides an overview of the capabilities of the different packages.

Table: Mean sepal width and length for three species of \_Iris\_.  
  
Species sepal\_length sepal\_width  
----------- ------------- ------------  
setosa 5.006 3.428  
versicolor 5.936 2.770  
virginica 6.588 2.974

Similarly to figures, when this is processed by knitr, the table will have the tab:iris-table label that can be used for cross-referencing.

### How to deal with cross-references?

Use the \@ref(label) syntax. For instance

On average \_setosa\_ has wider and shorter sepals than the other species  
(Fig. \@ref(fig:sepal-width-length), Table \@ref(tab:iris-table)).

### How to deal with citations?

You need two things: a BibTeX file that contains all the citations you use in your manuscript and a CSL (Citation Style Language) file that specifies the format of your citation. Software citation managers such as Zotero or Mendeley provide options to generate BibTeX files for your citations. CSL files exist for most journals, and can be downloaded from: <https://www.zotero.org/styles>. This is a convenient search interface provided by Zotero but you do not need to use Zotero to download or use these files.

## Where can I find more information?

* The RStudio Markdown website: <http://rmarkdown.rstudio.com/>
* The bookdown website: <https://bookdown.org/yihui/bookdown/>

# Documenting and managing dependencies

Reproducibility is also about making sure someone else can re-use your code to obtain the same results as yours. Understanding why your analysis may not lead to the same results on a different computer can be useful to determine how careful you need to be in documenting your setup.

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 developer takes backward compatibility (that old code works with recent version of R) very seriously. However, default values in some functions have changed, and new functions get introduced regularly. If you wrote your code on a recent version of R and give it to someone who hasn’t upgraded recently, they may not be able to run your code. If R itself is stable, the packages are generally much less stable. New functionalities get introduced with each versions, some functions get deprecated, and defaults options change. 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 a complicated. However, even simple documentation that helps others understand the setup you used can have a big impact. Here we present three levels of complexity to document the dependencies for your projects.

## Show the packages you used

With R, the simplest (but 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 provides a way to download all the packages at a given date from CRAN. 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; you were not using packages that are not available from CRAN (e.g. the development version of a package directly from a git repository).

Another approach is to use the packrat package. This package creates a library (a collection of packages) directly within your analysis directory. It increases the size of your project as all the source code for the packages is included, but it ensures that someone can recreate more reliably the same environment as the one you used for your analysis. It also makes it easier because the installation of these packages is fully automated for the person wanting to have the same setup.

## Use containers to share your setup

A step further in complexity is to use Docker. With Docker you recreate an entire operating system with all the software, data, and packages needed for your analysis. It is more technical to set up but it allows you to distribute the exact same environment as the one you used. If you want others to be able reproduce your results, and your analysis depends on software that can be difficult to install, it is an option that might be worth exploring.

Resources:

* Best practices for Scientific Computing (<http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745>)
* Good enough practices for Scientific Computing (<https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/>)
* 10 simple rules for reproducible computational research: <http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003285>
* A quick guide to organizing computational biology projects: <http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000424>
* Ten Simple Rules for Digital Data Storage (<http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005097>)
* The Reproducible Research CRAN Task View: (<https://cran.r-project.org/web/views/ReproducibleResearch.html>)