Computational Reproducibility Cookbook

Daniel H. Baker, University of York

2023-02-15

1 Intro

This document contains notes describing a process for making scientific papers computationally reproducible. It is written in R markdown, and intended to serve as a handbook for the 'ReproduceMe' pilot project at the University of York (in 2023). Most of the examples here involve R, as we anticipate using R for most projects, however many of the same things can be achieved in other languages.

1.1 Background on reproducibility

The goal of computational reproducibility is that all of the analyses in a paper can be reconstructed. For modelling and simulation papers, this requires sharing of code. For empirical papers it requires sharing of both code and data. Although data sharing has become commonplace in recent years, researchers appear to be much less willing to share analysis code. This could be for any number of reasons, such as fear that they have made an error, or lack of confidence in their own coding skills. It could also be simply be that current norms in most fields do not require code sharing. However making one's work reproducible has numerous benefits, including increasing the transparency of the analysis, and the confidence of readers, reviewers and editors. Other researchers can then use parts of an analysis pipeline in their own work, speeding up scientific progress. Finally, it is potentially the case that a reproducible workflow potentially benefits the authors themselves if they wish to revisit their analysis in the future.

2 Five levels of computational reproducibility

Computational reproducibility can be as simple as posting a script and data online. However there are additional steps that can make things easier for the end user, integrate the analysis code with the manuscript and figure generation, and preserve the computational environment used (e.g. package versions). A useful framework is the *reproducibility pyramid* illustrated in Figure 1. The relative widths of each layer of the pyramid indicate how common each level is, though note the proportions are not to scale - the vast majority of current research is not reproducible at all. Some further comments on each level follow.

Level 0 - the study is not computationally reproducible, usually because code and/or data are unavailable. This is the case for essentially all published research from the 20th century, as well as the vast majority of work published today. Note that non-reproducible work is not necessarily of a lower quality than reproducible work, it is just that this is harder to evaluate: we must trust that the authors' account of their analysis is full and accurate. Many high profile cases of research fraud might not have been possible, or would have been caught sooner, had reproducibility been the norm, or a requirement for publication.

Level 1 - a single script conducts the analysis using local raw or preprocessed data that is manually downloaded. This is the most basic form of computational reproducibility. However it usually requires quite a lot of effort on the part of the end user. For example there may be many files that need to be downloaded and stored in specific places on the computer, or the code might need modifying to locate the local copies of the data files. In some cases it could also be necessary for the end user to separately download and/or install additional packages or code repositories in order for the code to work. Finally, the output is likely to be

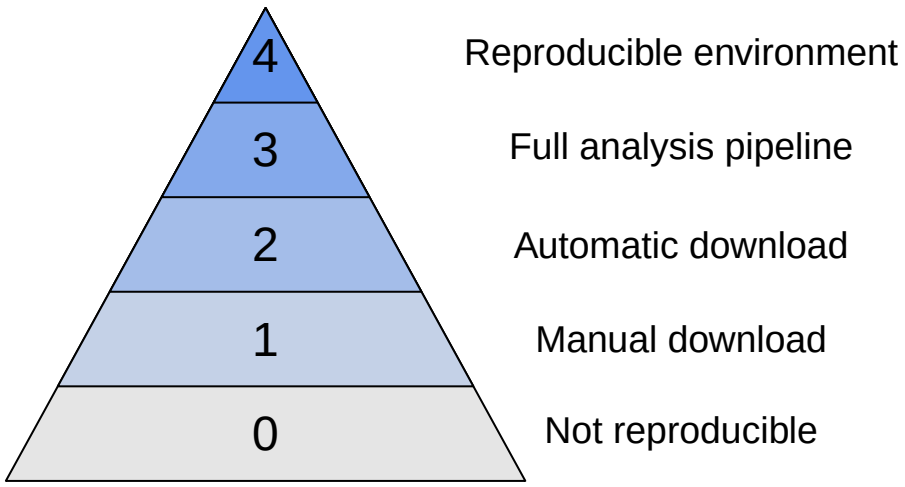

Figure 1: The reproducibility pyramid, indicating five levels of computational reproducibility.

the raw function output, e.g. for a statistical test, and it may require effort and expertise to find the values reported in the paper.

Level 2 - a single script automatically downloads and analyses raw or preprocessed data, and produces a formatted output containing values that can be incorporated into a paper. This is a more user-friendly solution, because the end user only needs to download a single script, and then all downloading of data is automated. There are several ways to do this, but the OSF provide an R package that makes it straightforward. At level 2, we anticipate that the output of any analysis is provided in a user-friendly format, such that the values included in a Results section are apparent from the output of the script.

Level 3 - a single script automatically downloads and analyses all raw data, generates all tables and figures, and produces a pdf of the entire manuscript (note that this script may execute other scripts, e.g. in different programming languages). This is a more impressive solution, as it is clear precisely where all of the values reported in the paper have come from, and how the figures were constructed. It can be substantially more work to get to this stage, but if the paper is already written then converting to an executable format is largely procedural. The journal eLife piloted something along these lines a few years ago, though it seems to have been forgotten about now.

Level 4 - all code is embedded in a Docker container (or similar) that includes the software required to run (e.g. specific package versions, and versions of the programming environment). This would be extremely technically challenging to do from scratch. However fortunately there are some useful tools already available. The Rocker project provides standard Docker containers for R studio, which can be downloaded and used as a wrapper for the entire computational environment. I have not tried doing this yet, but there is plenty of documentation, and also a useful paper by Peikert and Brandmaier (2021) that explains how to go about it. A less extreme version is to use the *renv* package to manage package versions in R.

Discussions about which level to aim for should be had with the study authors before work begins on making things reproducible. It is always easier to program something when the goal is clear, and there may be idiosyncracies specific to an individual study that means that Level 3 or 4 is not practical. Sometimes there are also restrictions on sharing of raw data (e.g. where this could potentially be used to identify a participant), and in such cases we might aim for reproducibility based on preprocessed or de-identified data. All of this is totally fine - for this pilot project our goal is simply to make things more reproducible than Level 0, so we need to be pragmatic rather than puritanical.

3 Implementation

3.1 *R Markdown* is really good

Markdown is a generic convention for document formatting. R Markdown takes this basic concept and integrates it into the R environment. This means you can produce a single script that contains both text and computer code. When the script is executed the code runs too, meaning that analyses can be conducted, and an output document is generated. It is possible to automate all parts of an analysis pipeline in this way, with different sections of the code importing and processing the data, generating figures, and formatting the output in the style of a paper. You can then 'knit' the markdown file to a variety of formats including pdf, Word documents, html and epub. Similar systems exist for other programming languages, including Jupyter notebooks for Python, and Matlab live scripts. We now have several examples of full papers written in R Markdown, available at the following repositories:

Baker (2021)

Baker et al., (2021)

Segala et al. (2023)

Baker et al. (2023)

In each case, the Rmd file contains the markdown script that will auto-generate everything in the paper. It's worth having a look through some of these to see how they are structured, though I expect (and hope!) that most of the papers we work on will be rather less complicated.

3.2 Markdown file structure

There is no set structure for Markdown files, which can interleave chunks of text, code, tables and images in any order. However it is sensible to follow some basic design principles to make documents easier to navigate. In principle we could include all of the R code at the start of the file, and all of the text for the manuscript below it. However this makes it quite difficult to find different sections of code. Instead it is better to break the analysis up into several distinct chunks of code, and scatter these throughout the manuscript in appropriate locations. The caveat here is that if we wish to embed values from some analysis, or figures that have been automatically generated, these things need to happen before we try to use the results in the text. In RStudio, section headings (created using either a single or double hash symbol) allow the user to navigate easily through the document via the menu in the lower left hand corner of the script window.

A chunk of R code is initiated with three reverse ticks (I can't get these to render properly in the markdown system, but it's ASCII code 96), followed by the letter r in curly brackets, and terminated with three more reverse ticks. Here is an example:

```
a <- 10.3
```

(see the raw markdown document for the actual syntax). By default the code will be reproduced in the output document. Often we wish to hide this, which we can do with the option `include=FALSE`, added inside the curly brackets. In between the opening and closing tick lines, we can include any R code we wish, loading in data and performing analyses. Any variables we create will be stored in a sandboxed Environment and are available to subsequent code chunks in the same markdown script. We can also save results to external files, such as RData files.

It is also possible to include pieces of 'inline' code as part of a markdown document. This allows us to report the outcomes of statistical tests automatically in a Results section, for example, which helps avoid typos from manual transcription. We include an R variable using a single reverse tick followed by an r. Then we type the variable we wish to display, and it appears in the text, for example we can insert the value we assigned to the variable 'a' earlier, which was: 10.3 (again see the markdown script for the syntax).

3.3 Flags to specify the level of analysis

Some analysis pipelines can become very complicated, requiring substantial storage space, and taking a long time to execute. In such cases I have found it helpful to set a flag (normally called *processdata*) at the start of the script to control the level of detail that the analysis is performed at. The flag is hierarchical, in that setting it to a value of 2 implies that the operations from levels 0 and 1 will also be executed. Whilst this will vary across studies, one possibility is as follows:

`processdata <- 0`: This means that no data are processed, and the manuscript is compiled using data that have already been analysed. The appropriate files are downloaded from the OSF (or other repository) if they are not available locally. This mode is particularly useful when writing a manuscript in markdown, as one can see the typeset text rapidly without waiting for analyses to execute.

`processdata <- 1`: This mode auto-generates any figures using pre-processed data. Again, any data that is required can be automatically downloaded. Once the figures have been created, the manuscript is knitted to the requested format.

`processdata <- 2`: Here we conduct 'second level' analyses that do not take a long time, for example statistical testing, using processed data and/or model outputs. These operations might require processed data files for individual participants, such as participant level averages of the dependent variable(s). An example for EEG might be the participant's average ERP waveforms for each trial.

`processdata <- 3`: The highest value for the flag specifies that all data should be downloaded and analysed from the lowest level available. Ideally this will be the raw data files recorded during an experiment. At this level of analysis we also perform any time-consuming analysis procedures, such as model fitting, bootstrapping, and so on. The outputs of all of these analyses are stored in an intermediate data file that allows the lower values of *processdata* to skip the resource-intensive analysis steps. This data file should also be stored in the project's online (i.e. OSF) repository so that it can be downloaded directly if the user does not have the resources available for the full analysis.

Different sections of code throughout the script begin with *if* statements that evaluate the *processdata* flag, and only execute their code segment if required. At the start of the script I include comments that specify what each level of the flag will do, and usually estimates of the time and storage space required, so that the user can make an informed decision about what they want to do.

3.4 Downloading data and other resources

R has a native function called *download.file* that can be used to copy files from the internet to the local computer. This is fine if you have a small number of files to download and they all have static URLs that are unlikely to change. However most of the time we are more likely to store files in a repository such as the Open Science Framework site. The *osfr package provides some useful tools for uploading, downloading, and also indexing OSF repositories. All you need to know is the five character identifier for the repository you are interested in. For example, for the repository https://osf.io/kthg3/ the five character code is the last part of the URL: kthg3

The first thing we need to do is index the root repository and see what files it contains:

```
library(osfr)

nodeID <- osf_retrieve_node('kthg3')
filelist <- osf_ls_files(nodeID, n_max=Inf)

filelist
```

```
## # A tibble: 33 x 3
##    name                     id                       meta
##    <chr>                    <chr>                    <list>
##  1 MovieS5.mp4              5f47c358746a81034b1a415e <named list [3]>
```

```
##  2 modelfiguresSubtractive.R 6069e5f9f2ad33013da74390 <named list [3]>
##  3 AdaptO.m                  5f34d389d42ad4001acdd66e <named list [3]>
##  4 toymodelfig.R             63728df264d67e2f80a07ac7 <named list [3]>
##  5 FigureS1.pdf              60fac935a13c6001fbb09d97 <named list [3]>
##  6 supplementary5_6.R        6133311ed1b14400eb6b5123 <named list [3]>
##  7 Figure5.pdf               63728dd964d67e2f80a07a96 <named list [3]>
##  8 MovieS4.mp4               5ec96f53aeeb6d00ec095330 <named list [3]>
##  9 MovieS3.mp4               5ec96f53aeeb6d00eb08adc0 <named list [3]>
## 10 MovieS1.mp4               5ec96f53aeeb6d00e408ec72 <named list [3]>
## # ... with 23 more rows
```

We can then download any of the files we might need, for example:

```
osf_download(filelist[1,])
```

You can specify the local directory where you want to store the file (more on that in the next section). I usually use text matching on the file names to identify the files I need, e.g.:

```
id <- pmatch('2011datalong.csv',filelist$name)
# only download the file if it doesn't already exist
if (!file.exists('local/2011datalong.csv')){
osf_download(filelist[id,],'local/')
}
```

There are also functions in the *osfr* package to enable automated uploading of files, which I find more stable than the drag and drop web interface. In order to upload files (or download them from private repositories) you need to provide an authentication key. However it is possible to download files from any public repository without requiring any log in details or authentication (as above), so our markdown files should work on any machine with internet access.

3.5 Local file management

When we download files, we need to store them somewhere sensible. I like having a directory in the project folder called something like */local* or */temp*, and then organising this into sensible subdirectories, such as */rawdata* and */processeddata*. Below we will introduce Github, which has quite strict storage limitations. Rather than syncing these directories of local files to Github, we can add the path to a file called .gitignore, which means that *git* will not try to sync it. Other subdirectories might also be useful, including */Figures* and */scripts*, depending on the files that are required to run the analysis. Note that it is generally a bad idea to use absolute referencing for file locations, for example including a path such as `C:\user\daniel\Documents` will not work on all devices. We will check that scripts work on multiple operating systems (Mac, Windows, Linux) to avoid any obvious problems.

3.6 Automating figure generation

One of my favourite things about R is that it can produce really excellent publication quality figures. I always found Matlab a bit lacking in this respect, and before switching to R I used to use a truly painful package called Grace (it can get good results, it's just very laborious). It is possible to generate figures directly in code chunks, and have them appear in your markdown file. This is the approach I have used in Figure 1 of this document (see the source Markdown code for details). However it is usually the case that journals want figures uploaded in separate files, and saving figures to an external file gives us more control over the exact sizing and other parameters of the plot.

With this in mind, my advice would be to use the *pdf* function when plotting figures, so that the output is saved to a pdf file (related functions such as *ps* and *tiff* can be used in the same way if other file formats are required). We open a new pdf file like this:

```
pdf('Figures/myfigure.pdf', bg="transparent", height = 8, width = 8)
```

Then we call whatever plotting functions we want to to generate our figure. I tend to do this using base R plotting functions, though many people use the *ggplot2* library. There are methods for splitting a figure up into multiple sub-panels, such as by using the *par* function. For example:

```
par(mfcol=c(1,3))
```

specifies a 1 row by 3 column plot layout. Each new plot appears in a different location within this grid. Once we are finished plotting, we close the file with:

```
dev.off()
```

Finally, at the appropriate place in the text we can include our figure:

```
knitr::include_graphics('Figures/myfigure.pdf')
```

Notice how this interacts with the *processdata* flag we talked about above. At the lowest level (*processdata* = 0) the figures don't get generated at all, but since they exist as independent files they can still be loaded in when creating the manuscript.

For really complicated plots with irregular layouts, it's sometimes necessary to do things in a slightly different way, e.g. by combining multiple postscript files and raster images at arbitrary locations. I explain how to do this in the plotting chapter (Ch 18) of my R book, which is available at: https://eprints.whiterose.ac.uk/181926/

3.7 Package management

The package management system in R is excellent, in that most of what we might need is available from a single repository (CRAN), and is stored consistently in a common library on the host computer. However package installation and activation is a little cumbersome. The *install.packages()* function will force install packages even if they are already present on the computer, which wastes a lot of time if it happens each time the script is run. I use the following code to check which required packages are present, install those that are missing, and activate everything:

```
packagelist <- c('knitr','remotes','tictoc','R.matlab') # list of CRAN packages

# find which packages are missing and install them
missingpackages <- packagelist[!packagelist %in% installed.packages()[,1]]
if (length(missingpackages)>0){install.packages(missingpackages)}

# then activate all the packages with the library function
toinstall <- packagelist[which(!packagelist %in% (.packages()))]
invisible(lapply(toinstall,library,character.only=TRUE))
```

However the above code will install the most recent package versions. Over time, packages get updated (along with the R language itself), and there is a risk that code will develop bugs as a consequence. An alternative is the *renv* package, which takes a snapshot of the precise package versions that are being used. This requires some minimal lines of code.

1. Initialise an *renv* 'lock' file to store package information in:

```
renv::init()
```

2. Once you have activated all the packages required for the project, take a snapshot and save it to the lock file:

```
renv::snapshot()
```

3. The above lines of code only need to be called when we first set up the project, after which they could

be commented out from the script. If the lock file is included as part of the project, the environment can then be restored with:

```
renv::restore()
```

This solution isn't perfect, for example it doesn't control the R version being used. However it does solve part of the problem of ensuring reproducibility. Ultimately we might attempt to wrap the entire R environment in a Docker container, but I haven't worked out how to do this quite yet.

3.8 Python chunks

It is possible to include sections of Python code in an R markdown document. This might be useful if one needs to access Python libraries and toolboxes. It requires the *reticulate* package to be available, through which it is possible to specify which Python version you wish to use on your machine (it must already have been installed). A python code chunk resembles an R code chunk, and is initiated by three reverse ticks followed by `{python}` (and terminated with three more reverse ticks). Within a python code chunk it is possible to import packages from the local python instance in the usual way, and conduct any Python analyses.

The Python instance has a sandboxed area of memory that cannot directly access the variables in the R Environment. However a special R variable is created called *py* that stores any Python variables as subfields. So a variable called *pyvar* in the Python code can be accessed in R as `py$pyvar`. Similarly, creating a new field of the *py* variable, such as `py$Rdata` will cause a variable called *RData* to appear in the Python memory. In this way it is possible to pass information between R and Python. It is a little more clunky than having a shared memory, but presumably necessary for technical reasons (i.e. to do with different data types). It is possible to do something similar for other languages, including Julia, C++, JavaScript, bash and SQL. I haven't tried this, but apparently it's also possible to use Octave in this way, which is an open source Matlab alternative. If we end up with some Matlab code that we can't translate to R, this might be a useful option.

3.9 Calling Matlab and other functions from the command line

Frustratingly, it is not possible to directly access the Matlab kernel through Markdown. There is a workaround, which is to create a Matlab script and send this directly to Matlab via the terminal. Annoyingly there is no way to directly store the outputs in R's memory (though they do get echoed to the console if you omit the semicolons at the end of each line), so the script will need to save any outputs to an external file, which would then be read back in by R. However this does permit access to Matlab-only toolboxes that might be critical for some analyses. It is also necessary to tell the terminal exactly where Matlab exists on the machine, which will probably need to be altered by the user. Here is some example code that generates a simple Matlab script, and then executes it via the terminal:

```
# build the matlab script
line1 <- "a = 1:3:30;"
line2 <- "output = a.^2;"
line3 <- "save('matlaboutput.mat','output')"
matlab_lines <- c(line1, line2, line3)

# output the script to an external file
writeLines(matlab_lines, con='~/myscript.m')

# execute the script in a Matlab instance with no GUI, called through the terminal
system("/Applications/MATLAB_R2022b.app/bin/matlab -nodisplay -r \"run('~/myscript.m'); exit\"")

# use the R.matlab library to read the data file into R
library(R.matlab)
m <- readMat('~/matlaboutput.mat')
```

```
# store the output variable in a native R vector
output <- m$output

# clean up by deleting the script and data file
file.remove('~/myscript.m')
file.remove('~/matlaboutput.mat')
```

It should be possible to use a similar approach to call other programs that are accessible directly from the terminal, for example FSL, Freesurfer, and other neuroimaging software.

3.10 Output formats

There are a wide array of possible output formats available when using Markdown. However the most useful are pdf, HTML and Microsoft Word. I include the following code in the header information of every Markdown file. RStudio then makes all three file formats available in the dropdown menu next to the 'Knit' button at the top of the script.

```
bookdown::pdf_document2:
  fig_caption: yes
  toc: no
  keep_tex: yes
word_document: default
html_document: default
```

Note that *pdf_document2* from the *bookdown* package seems to work much better than the default *pdf_document*, which can't cope with figure numbering for some reason. Actually Word and HTML documents also don't seem to handle figure referencing properly either, but they can sometimes be useful.

3.11 Typesetting equations using LaTeX

When Markdown creates pdf files as output, it actually uses LaTeX (pronounced 'laytech', or sometimes 'laatech'). This is a document typesetting system very popular in the mathematical sciences because it is good at typesetting equations, and it is possible to incorporate pieces of LaTeX code into a Markdown file (such as the fancy text in this paragraph for the word Latex). Some journals also accept LaTeX files for submission. A full overview of how to use LaTeX is beyond the scope of this document. However it is straightfoward to incorporate a simple equation between pairs of dollar symbols. For example, the code `$y = \sqrt{x^2}$` generates the equation: $y = \sqrt{x^2}$.

3.12 Citations and automatic figure and equation referencing

We can include citations using a BibTex file containing the references. This can be exported from most reference management software (e.g. Zotero), or created in a native BibTex package such as BibDesk. We specify the reference file in the header information of the markdown file:

```
bibliography: references.bib
```

Then, we can cite a paper either inline as `@Peikert2021`, which renders as Peikert and Brandmaier (2021), or in square brackets as `[@Peikert2021]`, which renders as (Peikert and Brandmaier, 2021). The bibliography appears at the end of the pdf file.

Similarly we can reference figures by giving them a tag. For example the pyramid figure has the tag *repropyramid*, so we can reference it with `\@ref(fig:repropyramid)`, which renders as Figure 1. Equations work similarly but with an independent numbering system from the figures, indexed by the *eqn:* tag.

3.13 Github syncing

Github is a useful tool for version management and online code storage. It integrates well with OSF and RStudio. If we set up a new public repository on the Github website, we can pull it down to our local computer in RStudio by choosing File: New Project. Select Version Control and then Git. It turns out that Github changed the way to interface with repositories, so much of the information online is out of date. The correct format for the repository URL is: `git@github.com:bakerdh/ReproduceMe`

The files will download, and we can then make modifications and then synchronise back to the repository by Committing and then Pushing in the Git tab (upper right pane in RStudio). There's a lot more that Github can do, but frankly I don't really understand it, and it's beyond what we'll need for this project.

4 An example:

Uploading data Downloading data Analysis and embedding code in text Autogenerating figures Github syncing Renv

5 Modality-specific issues

MRI, manual analyses

References

Peikert A, Brandmaier AM. 2021. A reproducible data analysis workflow with R markdown, git, make, and docker. *Quantitative and Computational Methods in Behavioral Sciences* **1**:e3763. doi:10.5964/qcmb.3763