Reproducibility made simple

Automating reproducible research workflows

Aaron Peikert

2020-08-21

Contents

Abstract

This thesis discusses the practical and metascientific merits of reproducibility of scientific workflows and results. I present `repro`, an R package which simplifies the adherence to the best practices of reproducibility proposed by Peikert & Brandmaier (2019). The resulting workflow complies with Open Science principles, is transferable across machines and preserved over time. Research projects are virtually guaranteed to reproduce and consequently are more comfortable to replicate. I explained how the technical solutions, namely Git, RMarkdown, Make, and Docker, overcome common problems of reproducibility and how the interaction with these tools is simplified through automation. The high degree of automation allows the workflow to scale well across researchers, letting them collaborate more transparently and effectively, as well as across machines, including high performance and cloud computing technology. To adapt to the ever-shifting landscape of software requirements `repro` has a modular design allowing other packages implementing research workflows to build upon its infrastructure.

Acknowledgements

I dedicate this thesis to my mother, who endured what cannot be described to give what only can be felt.

I want to thank my advisers, Dr. Andreas M. Brandmaier and Prof. Manuel Völkle for their time and patience, and my friends for their resourceful advice:

- Jakob Eißner (commits `bc53e58`, `9841a36` & `4857dcb`)
- Elisabeth Riha (commit `82f32d4`)
- Caroline Gahrmann (commit `d9a730c`)
- Andreas Brandmaier (commit `9207348`)

1 Theoretical Considerations

Claerbout & Karrenbach (1992) define reproducibility as the ability to obtain the same results, from the same dataset. Conversely, they call a result replicable if one draws the same conclusion from a new dataset. This thesis is concerned with the former goal, providing researchers with an accessible yet modular workflow, that is virtually guaranteed to reproduce across time and computers.

The scientific community agrees that ideally, their work should be reproducible. Indeed it may be hard to find a researcher who distrusts a result because it is reproducible; to the contrary, many argue it is "good scientific practice" to ensure what they consider reproducible ("Reducing Our Irreproducibility," 2013; Deutsche Forschungsgemeinschaft, 2019; Epskamp, 2019). Several reasons, practical and meta-scientific, justify this consensus of reproducibility as a minimal standard of Science.

Reproducibility makes researchers life more productive in two ways: The act of reproduction provides, at the most basic level, an opportunity for the researcher to spot errors. At the same time, other researchers may also benefit from reusing materials from an analysis they reproduced.

Beyond these two purely pragmatic reasons, reproduction is crucial, depending on the philosophical view of Science one subscribes to, because it allows independent validation and enables replication. Philosophers of Science characterise Science mainly as a shared method of determining whether or not a statement about the world is "true" (Andersen & Hepburn, 2016) or more broadly evaluating the statements verisimilitude (Gilbert, 1991; Meehl, 1990; Popper, 1962; Tichỳ, 1976). If this method is for experts to agree on the assumptions and deduce "truth", reproducibility is hardly necessary. On the other hand, it does gain importance if one induces facts by carefully observing the world. The decisive difference between the above approaches is that the former gains credibility by the authority of the experts, while the latter is trustworthy because anyone may empirically verify it.

Accepting induction as a scientific method hence hinges on the verifiability by others. Some have even argued that such democratisation of Science is what fueled the so-called scientific revolution (Heilbron, 2004, Scientific Revolution). The scientific revolution had the experiment as an agreed-upon method to observe reality, and a much later revolution provided statistical modelling (Rodgers, 2010) as a means to induction. This consensus about how to observe and how to induce builds the foundation of modern empirical Science. Two reasons justify why we must assume reproducibility as a scientific standard if we accept induction as a scientific method: First, it allows independent verification of the process of induction, and second, it enables replication as a means to verify the induced truths.

However, neither verification of the induction nor the induces results are strictly enabled by the definition of reproducibility provided by Claerbout & Karrenbach (1992) given above. A simple thought experiment illustrates this shortcoming: Imagine a binary—therefore only machine-readable—program being perfectly reproducible; hence upon the input of the same dataset, it completes a scientific manuscript with the identical numbers at the right places. Furthermore, let us assume this hypothetical program may never hold if the dataset changes. Does the predicate "reproducible" in this situation reduce the number of mistakes or enables reuse? Unlikely. Or could one audit it and use it in replication? Hardly. This admittedly constructed case of a reproducible black box shows that we are not interested in reproducibility per se but rather in its secondary effects. Because it is a binary program, it does not enhance understanding and because it does not apply to other datasets, it does not facilitate productivity. In fact, such a program does not grant the researcher any practical or metascientific advantages over non-reproducible research products.

Spoiling its elegant simplicity, I, therefore, extend the definition by Claerbout & Karrenbach (1992) to address this issue, by further demanding that reproducibility must allow criticism and facilitate replication:

> A reproducible scientific product allows one to obtain the same results from the same dataset in a way that enables substantive criticism and therefore facilitates replication.

Thus, transparency should enable reproducibility: Ensuring a clear link between the data and its results, promotes both replicability and reproducibility. Comprehension is a necessary precondition to form substantive criticism, which motivates the iterative scientific process. Bowing to this general notion, scientific publications are required to provide enough detail about the research process so that others may contribute constructive criticism. I am convinced that we should accept the same standard for everything that is published — including code.

Consequently, something is no longer either reproducible or not, but there are shades because a research product can promote replication and comprehension to varying degrees. Also note that a scientific result can facilitate replication without anyone ever attempting to replicate it, e.g. by educating other researchers about the method of analyses, being openly accessible and providing reusable components.

Hence, reproducibility has a technical aspect, which is ensuring identical results, and a non-technical side, which is facilitating understanding and progress through cumulative Science. The former relates to the practical advantages while the latter serves the metascientific purposes of reproducibility. An important caveat of the technical aspect is that generating the same results from the same data should always be possible regardless of time and computer. As such, a reproducible analysis should be:

1. understandable by other researchers,
2. transferable across computers,
3. preserved over time.

This extended notion and demanding standard of reproducibility is justified by two recent developments in the social sciences in general and psychology in particular: the emergence of a "replication crises" (Ioannidis, 2005) and the rise of "machine learning" (Jordan & Mitchell, 2015) as a scientific tool. Both trends link to the use of statistical modelling on which the social sciences became reliant for testing and developing their theories (Gigerenzer et al., 2004; Meehl, 1978). It turns out that, if one fits the very same statistical model as published on newly gathered data, one fails more often to achieve results that are consistent with what was published, than one succeeds. (Open Science Collaboration, 2015).

Such failure to replicate findings that were believed to be robust has grown to a level that some social scientists call a crisis (Pashler & Wagenmakers, 2012). They put forth various causes and remedies to this crisis. Most remedies share a common motif: transparency. Some call for Bayesian statistics (Maxwell et al., 2015), as it makes assumptions more explicit, or demand preregistration (Nosek et al., 2018) as a means to clarify how to analyse the data, before data was collected. Others require the researchers to publish their data (Boulton et al., 2012). Similar calls for transparency, as a response to the replication crises, have formed the open science movement which stresses the necessity of six principles (Kraker et al., 2011):

- Open Access,
- Open Data,
- Open Source,
- Open Methodology,
- Open Peer Review and
- Open Educational Resources.

I argue that a research product resting on the first four pillars facilitates replication optimally and hence, it satisfies the highest standard of reproducibility. The last two pillars are then consequences of reproducibility. If everyone has access to a scientific product and its data along with the source code, everyone has the possibility of understanding the underlying methodology, which enables them to criticise the results and educate themselves. Having done so, they are in the best position for replication. Hence, any one's ability to reproduce such a result gives a tangible affirmation of its usefulness to the scientific community.

While establishing reproducibility is no hurdle if one can perform the calculations needed with a pocket calculator, the more and more frequent use of computer-intensive methods renders such expectation questionable. The use of machine learning techniques, which has been once enabled by the computer taking over strenuous works like estimating and comparing thousands of models, now impedes our quest for reproducibility. More massive amounts of more complicated computer code than ever before, create room for errors and misunderstandings, leading the machine learning community to believe that they face a reproducibility crisis themselves (Hutson, 2018). At the same time, machine learning becomes more and more popular ins psychological research (Brandmaier et al., 2013; Jacobucci et al., 2019; Yarkoni & Westfall, 2017). Therefore, I am far from calling for abstinence from machine learning, just because it complicates reproduction, but want to emphasise the need for solutions that allow anyone in any field to reproduce even the most sophisticated analysis. Such possibility enables commutative Science and allows the researcher to build a more complete and accurate understanding of the fields subject matter.

Peikert & Brandmaier (2019) put forth an analysis workflow which provides this convenience for everyone to reproduce any kind of analysis. However, they fail to provide the same level of convenience for the researcher who created an analysis in the first place. Setting up the workflow eats up a considerable amount of the researcher's time because it requires a level of technical sophistication that cannot be expected across all disciplines. This time should researchers rather spend on advancing research. This additional effort offsets the increase in productivity, promised by reproducibility, which I regard as most significant in the workflows adoption. Persuading researchers, who find the meta-scientific argumentation noble but impractical, do not care about it or even oppose it, requires concrete, practical benefits. Luckily, most of this setup process may become automated, letting the researcher enjoy the workflows advantages while decreasing the efforts necessary to achieve them. Providing a version of the analysis workflow by Peikert & Brandmaier (2019) that is easier to use and more accessible is the goal of this thesis and the herein presented `repro`-package for the R programming language (Peikert et al., 2020).

2 Technical Solutions

This section summarises the workflow proposed by Peikert & Brandmaier (2019; see also The Turing Way Community et al., 2019 for a similar approach). They argue that publicly sharing code and materials is not sufficient to ensure reproducibility: Instead, reproducibility has to rest on five pillars:

1. **file management** a folder containing all files, referring to each other using relative paths
2. **literate programming** a central dynamic document, that relates code to thought
3. **version control** a system in place that manages revisions of all files over time
4. **dependency management** a formal description of how files relate to each other
5. **containerisation** an exact specification of the computational environment

These pillars stipulate the relations between thought, code and data with their change over time and environment and hence, they reach all requirements of reproducibility through being:

1. understandable to other researchers,
2. transferable across machines,
3. preserved over time.

While comprehensibility to the scientific community is probably the most crucial objective, it is also the most difficult to achieve. That is because as a non-technical requirement, no set of rules can assure its fulfilment (though clear writing[1] and clean code[2] certainly help). Transfer and conservation, on the other hand, are problems with technical solutions.

Peikert & Brandmaier (2019) propose to use a combination of RMarkdown, Git, Make, and Docker, for users of the R programming language (R Core Team, 2020) and provide the high-level overview which can be seen in Figure 1.

However, they stress that any combination of tools is suitable as long as it facilitates the above pillars.

[1]Williams (2017) provides some excellent principles for writing clearly.

[2]Martin (2011) proposes a coding paradigm that found widespread use because of its focus on understandability.

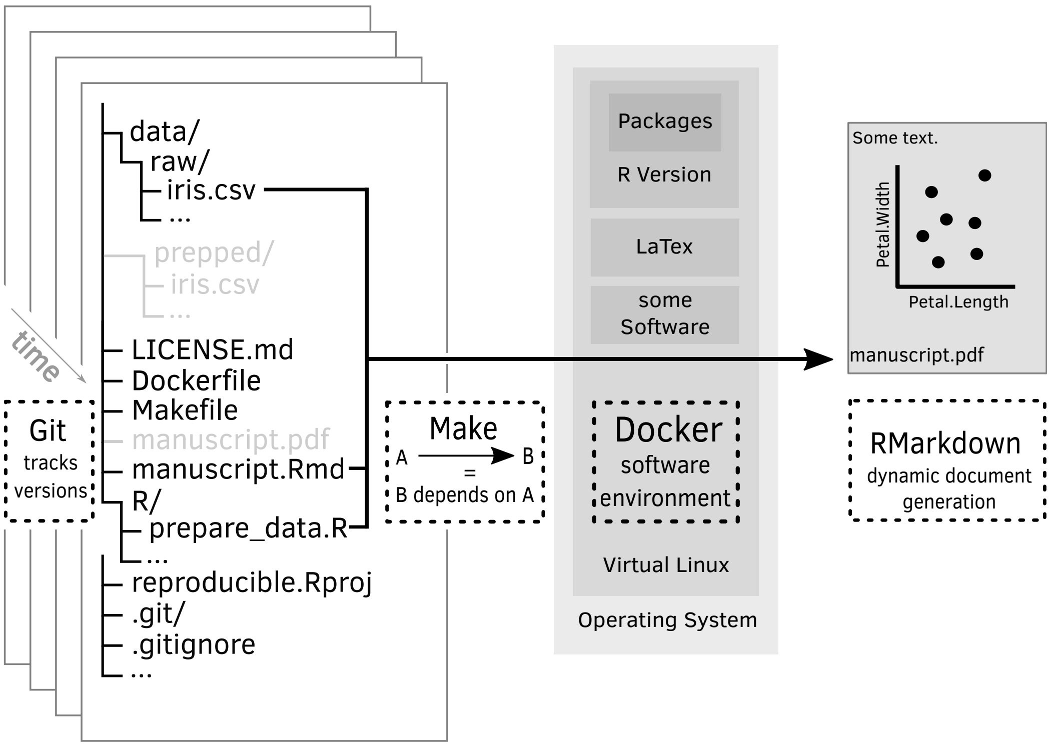


Figure 1: Overview of the interplay of RMarkdown, Git, Make, and Docker (Peikert & Brandmaier, 2019).

Each of the following sections first raises a challenge for reproducibility, then outlines a conceptual remedy along with a concrete tool and concludes how they relate to the package `repro`. The relation of these tools with `repro` is then expanded in the next chapter.

2.1 File Organisation

File organisation has to meet two challenges: First, the structure needs to be understandable for others, and second, it needs to be self-contained so that it can be moved to another machine.

Adhering to conventions can help other people understand how files are organised. For example, the filename `R/reshape.R` both follows standard naming conventions (i.e. all lowercase; ends with .R; placed within the R directory) and is meaningful. Contrarily, `myScripts/munge_Data.r` is probably a lot harder both to understand and to remember for most R-users.

Following two guidelines makes the file structure self-contained:

1. Everything is in one folder.
2. Every path is relative to that folder.

This simple concept of a self-contained folder is facilitated by two R specific tools: RStudio projects and the `here` package (Müller, 2017). The former exempt the user from changing the working directory manually; the latter infers absolute paths from relative ones. However, unlike the native R solution, this inference is consistent across operating systems, scripts and RMarkdowns.

The `repro` package offers a template for an RStudio Project, which sets up a file structure that follows best practices and conventions. This template provides the researcher with a minimal example of a reproducible analysis. It enables researchers to learn by easy to understand examples, e.g. how files should refer to each other. Such learning by example complements the abstract arguments for best practices with a more hands-on perspective. The researcher can thus adapt code and files to their need, either by merely changing them manually or by the modular structure of `repro`. Furthermore, does this template provide a reference for researchers that is more concise than any tutorial and connects the concept of file structure, with the ideas of the following sections.

2.2 Dynamic Document Generation

A clear file structure helps researchers to understand better how a scientific result relates to the code, but the otherwise strict segregation of code and document may obscure how both connect, e.g. which parts of code generate which results. Providing a direct link, dynamic document generation allows interspersing text with code and its results, in order to produce a human-readable document. The key feature is that every time such a document is rerun, the results are reproduced dynamically. This functionality eliminates errors due to copy and paste results from statistical software to a text processor. This mistake happens far too often; Nuijten et al. (2016) reports that 50% of papers from the psychological sciences contain at least one error that could have been prevented.

RMarkdown provides a convenient framework to write such dynamic documents and render them as a wide range of output formats[3]. In an RMarkdown, three parts can be distinguished:

- one specifying its output and metadata,
- one containing code, and
- one with descriptive text.

Each part uses its own language, all of them designed with ease of use and readability in mind. The one section containing the output format and other metadata

[3]The document you are viewing also results from a collection of RMarkdowns available as website, PDF and E-book

alongside is written in YAML (recursive acronym for “YAML Ain’t Markup Language”, see the example below). This specification is located on the top, separated by three dashes at the beginning and the end of the section. (R-)Code executing an analysis can be placed in a distinct chunk or inline within the text. The former has three backticks on their own line signifying beginning and end. The latter is quoted in a pair single backticks. Examples of both methods can be found below. Text which is not fenced by either three dashes or backticks is interpreted as literal text written in the Markup language “Markdown”. Markdown allows annotating text to signify formatting like bold, italic, links and the inclusion of images. This markup is designed to be well readable even as source files.

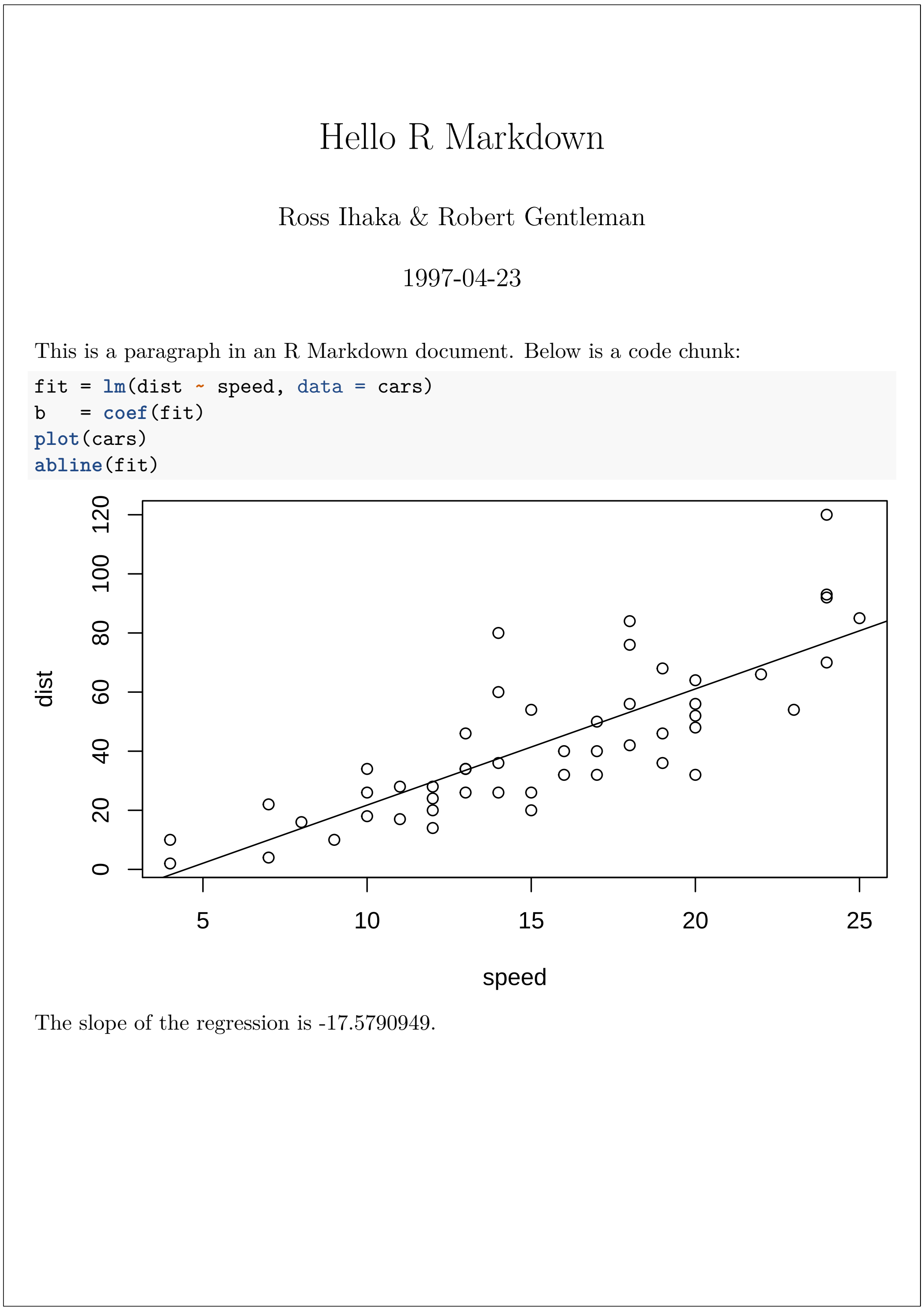
The following section shows examples of metadata, code and text, specified as above described, forming a minimal example of an RMarkdown (adapted source code from Xie et al. (2019)/CC BY-NC-SA 4.0):

```
---
title: ”Hello R Markdown”
author: ”Ross Ihaka & Robert Gentleman”
date: ”1997-04-23”
output: pdf_document
---
```

```
This is a paragraph in an R Markdown document.

Below is a code chunk:

‘‘‘{r}
fit = lm(dist ~ speed, data = cars)
b = coef(fit)
plot(cars)
abline(fit)
‘‘‘

The slope of the regression is ‘r b[1]‘.
```

Resulting in the rendered document seen in Figure 2:

Undeniably, RMarkdown facilitates reproducibility greatly, but it cannot ensure reproduction. To virtually guarantee reproduction, the `repro` package extends the YAML metadata to incorporate Dependency Management and Containerisation into the process of dynamic document creation. It, therefore, extends the existing metadata model of RMarkdown to capture all dependencies of the analysis and ensures reproduction. Because many researchers are already familiar with RMarkdown this solution provides minimal cognitive overhead. However, even though the metadata relates to a single document, `repro` compiles all needed in-


dist
0
20
40
60
80
100
120
5
10
15
20
25
speed

Hello R Markdown

Ross Ihaka & Robert Gentleman

1997-04-23

This is a paragraph in an R Markdown document. Below is a code chunk:

```r
fit = lm(dist ~ speed, data = cars)
b   = coef(fit)
plot(cars)
abline(fit)
```

The slope of the regression is -17.5790949.

Figure 2: Result of minimal RMarkdown example.

formation from all documents across the project into suitable Docker- and Makefiles.

2.3 Version Control

Text, code and results of a scientific document are typically modified in cycles of many revisions. As changes accumulate, different versions do so as well, posing a problem for reproducibility as it may be challenging to find out which version of code relates to the final product. One may argue that in the typical publication process, the final product is apparent: the published paper. However, reproducibility may be crucial even before publication as part of the collaboration and within the peer-review process. Also, recent trends in the publication process as preprints, open review, registered reports and post-publication review, blur the lines between published and unpublished.

To organise different versions as changes accumulate across the phases of a project across machines and users is a well-known challenge in software development. This challenge is met by a high degree of automation that keeps track of different versions and has advanced facilities to compare and merge them.

One such version control software is Git. Git tracks versions of a project folder by taking snapshots of a given state called commits. Each commit has a unique ID, called a hash, as well as a short description of the changes made, called commit message and a link to the previous commit. This linking procedure creates a "pedigree" of versions that makes it easy to see how things have evolved. Going back in time to a specific version only requires knowing the hash of the commit. To mark commits as special milestones, they can be tagged, e.g. as preregistration, preprint, submission or publication.

While mastering Git requires some experience, most of the time, only four commands are needed, all of which can be accessed through RStudio's Git interface:

git add take a snapshot of the given file
git commit create a commit of all added files
git push upload recent commits to a server
git pull download and integrate recent commits from the server

While a few other commands are necessary to set up Git in a given project directory, this work is done by the `repro`-package. In my experience, Git is overwhelming to learn because users have to interact with it through a terminal. While R users have some experience with interactive programs that can only be accessed through written commands (e.g. R itself), Git follows—from the perspective of R users—strange and archaic syntactical rules. Thanks to `usethis` and indirectly through `credentials` (Ooms, 2019a) and `gert` (Ooms, 2019b) users can use most of Git through R commands. Because `repro` has the same interface as `usethis` the cognitive overhead is kept as low as possible.

2.4 Dependency Management

In a scientific analysis of empirical data, the statistical or computational results depend on code which in turn depends on data. However, rarely the data is analysed as it is, but some code is dedicated to preparing it (e.g. removing outliers, reshaping, aggregating, artefact correction etc.). Most likely, each analysis needs a slightly different version of the data. An analysis of missingness requires the missings to be retained, but some statistical models do not allow that. Or the modelling software requires data to be differently shaped, then the plotting library. Often it is the case that one analysis is based on the output of another and so forth. As these relations can become quite complicated, it is necessary to make them explicit to avoid confusion. Dependency management provides a formalism that describes how files depend on other files. More specifically, it provides an automated way to create files from other files, e.g. it automatically generates a cleaned version of the data, by relying on a cleaning script and the raw data.

Such relations may be layered; hence, if a plot requires this cleaned dataset, first the cleaned dataset and then the plot is generated automatically. Such structure allows to save considerable computing time, as dependencies are not generated again if they already exist, but only if one of their dependencies has changed. In this example, upon recreation of the plot, the cleaned dataset is not generated as long as the cleaning script and the raw data remain unchanged. Such intelligent behaviour is most useful when the preprocessing requires a lot of computing time as is typical in neuroimaging or machine learning.

Make is a tool for dependency management. While originally designed for the compilation of programs, it is now increasingly recognised as a tool for ensuring reproducibility. It allows for all features mentioned above and even more as it a full-fledged programming language.

The repro package provides a simplified interface to the essential features of Make, eschewing the need to learn yet another language while leveraging its most important feature: "dependency management" for scientific data analyses. Similarly to Git Make uses an interface that is quite different from the way R is used. Because Make doesn't have to be used interactively, `repro` reuses the interface of RMarkdown to interact with Make. Users hence need only to learn RMarkdown and gain the ability to use Make (and as described in the next section Docker) if they use `repro`.

2.5 Containerisation

Most computer code is not self-contained but depends on libraries and other software to work (e.g. the R programming language or packages). These external dependencies pose a risk for reproducibility because it may not be clear what is necessary—besides the code and data—and how to install it. Even when all required software and their exact versions are recorded meticulously, it may be a challenge to install them on a different system. First, it is difficult to maintain different software versions on the same computer, and second, it may be unclear how to obtain an exact copy a specific software version. Setting up a computer exactly as someone else's is difficult enough, but replicating another computers state from several years ago is at best painstaking, if not impossible.

To overcome this challenge, the software environment of a project needs separation from the rest of the software environment. Technically such separation is called virtualisation because one software environment is hosted on another. Such virtual environment allows each project to have its own software environment without interfering with each other. Hence, such setup is ideal for conservation and can easily be recreated on another machine.

Docker allows virtualisation of the whole software stack down to the operating system, but in a much more lightweight way than traditional virtual machines. This lightweight but comprehensive virtualisation is called containerisation. Containers save storage by being based on each other, enabling reuse. Hence, one container is based on another, e.g. two containers with the same R version share the storage for everything but the different R packages. Containers are created from a plain specification called `Dockerfile`. This file defines on which container the result should be based upon and what software should be installed within it.

The `repro`-package automatically infers which packages are needed and creates an appropriate Dockerfiles and the container from it. This process relies on the same syntax as is employed for Make, so that users do not have to differentiate between file and software dependencies. E.g. for this project the `repro`-generated Dockerfile looks like this:

```
FROM rocker/verse:3.6.3
ARG BUILD_DATE=2020-07-16
```

```
WORKDIR /home/rstudio
RUN MRAN=https://mran.microsoft.com/snapshot/${BUILD_DATE} \
  && echo MRAN=$MRAN >> /etc/environment \
  && export MRAN=$MRAN \
 && echo ”options(repos = c(CRAN=’$MRAN’), download.file.method = ’libcurl’)”\
  >> /usr/local/lib/R/etc/Rprofile.site
RUN install2.r --error --skipinstalled \
  bookdown \
  devtools \
  gert \
  here \
  usethis
RUN installGithub.r \
  aaronpeikert/repro@adb5fa56
```

Furthermore, does `repro` provide the infrastructure that connects Make and Docker, e.g. it adds appropriate Make target for the container's image, and tunnels the other Make targets through the container. This results in a special Makefile for the Docker-related instruction that is incoporated in the main Makefile e.g. like this:

```
### Docker Options ###
# --user indicates which user to emulate
# -v which directory on the host should be accessable in the container
# the last argument is the name of the container which is the project name
DFLAGS = --rm $(DUSER) -v $(DDIR):$(DHOME) $(PROJECT)
DCMD = run
DHOME = /home/rstudio/

# docker for windows needs a pretty unusual path specification
# which needs to be specified *manually*
ifeq ($(WINDOWS),TRUE)
ifndef WINDIR
$(error WINDIR is not set)
endif
    DDIR := $(WINDIR)
    # is meant to be empty
    UID :=
    DUSER :=
else
    DDIR := $(CURDIR)
    UID = $(shell id -u)
    DUSER = --user $(UID)
endif

### Docker Command ###
```

```
ifeq ($(DOCKER),TRUE)
    DRUN := docker $(DCMD) $(DFLAGS)
    WORKDIR := $(DHOME)
endif

### Docker Image ###

docker: build
build: Dockerfile
    docker build -t $(PROJECT) .
rebuild:
    docker build --no-cache -t $(PROJECT) .
save-docker: $(PROJECT).tar.gz
$(PROJECT).tar.gz:
    docker save $(PROJECT):latest | gzip > $@
```

3 Working with repro

The `repro` package is designed to streamline the researchers' workflow. It helps researchers to set up, create, reproduce and change an analysis with little more than a simple mental model. To that end, it stands on the shoulders of giants and provides only a minimal layer of abstraction for the tools as mentioned earlier. While there is a considerable variety in how researchers can approach an empirical study and its analysis, Figure 3 offers an idealized workflow.

This workflow can be approached from two perspectives, that of an author of a reproducible analysis or as a contributor. I use the term contributor in its broadest possible meaning, because—in my view—even a reader who is thorough enough to reproduce an analysis contributes something of value. Consequently, I include everyone with interest in reproducing the analysis in the group of collaborators, be it a coauthor, a reviewer or the very person who created the analysis at an earlier point in time.

From the contributor's point of view `repro` acts as an assistance system, advising users on how to set up their computers and how to reproduce an analysis. It is a major difficulty for untrained users to detect their systems state accurately and act correspondingly (Parasuraman & Mouloua, 2018, Chapter 8: "Automation and Situation Awareness"). `repro` parses complex technical information and straightforwardly presents them. It, therefore, enables even relatively inexperienced users to make use of tools which otherwise would require extensive training. From the perspective of the author who creates an analysis, `repro` is a toolbox which provides the right tools for most of the user's requirements. In the creation phase, the benefit for inexperienced users is even more accentuated.

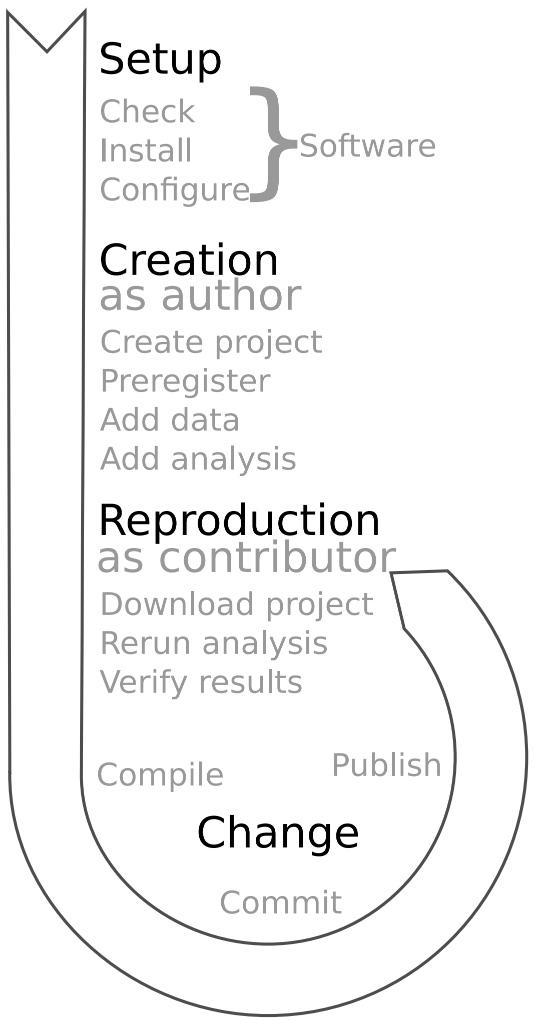


Figure 3: Schematic illustration of a reproducible workflow.

Following the standards of Peikert & Brandmaier (2019) usually requires experience in the use of Bash, Make, Docker, YAML, (R-)Markdown, R, Latex, knitr, and Git. Besides some practice, users need to be aware, remember, and correctly implement best practices in each tool. `repro` significantly lowers this threshold. `repro` still expects the user to have some training in the use of R, (R-)Markdown, and Git—which is increasingly recognized as a standard in the Open Science and R community—but forgoes training in Make and Docker. An essential guideline in designing `repro` was to make best practices more accessible to implement than their alternatives. Where possible `repro` allows the user to forget about the details of the often tricky implementation and abstracts them away.

The structure of this chapter mirrors the workflow in Figure 3, first from the perspective of a contributor, then from the perspective of a creator. It explains step by step how to:

1. Set up the required software;

2. reproduce an analysis that follows the here outlined standards;
3. apply and publish changes;
4. create a reproducible workflow from scratch.

`repro` supports several alternative software implementations for each step and integrates them into one coherent workflow. This modular structure is inspired by the usethis-package (Wickham & Bryan, 2020). The steps described here follow the recommendations of Peikert & Brandmaier (2019), and hence combine RMarkdown, Git & GitHub, Make and Docker.

3.1 Setup

Reproduction, change, and creation of an analysis require the user to have software installed that is specific to the workflow they choose, but independent of the analysis. A set of functions (following the pattern `check_*`, for a complete list see `help(check)`) assists users to ensure that everything they need is installed and correctly configured. If `repro` detects that something is not installed or configured, it guides users through a step by step procedure on how to resolve these issues in accordance with their specific software platform. Currently, it supports all major operating systems (Windows, OS X, Linux).

First users have to install `repro`. The following code snippet installs `repro` from GitHub:

```r
# check if remotes is installed, if not install it
if (!requireNamespace("remotes")){
  install.packages("remotes")
}
# install repro
# "package::function" means to use a function
# without loading the whole package
remotes::install_github("aaronpeikert/repro")
```

If repro is installed one may load it via:

```r
library("repro")
```

Subsequently, users can check if the required software is already installed. The workflow by Peikert & Brandmaier (2019) depends on Git (and GitHub), Make, and Docker. Consequently, the following commands check if the user has set up all the requirements:

```r
check_git()
```

```
> v Git is installed, don't worry.
```

```
check_make()
```

```
> v Make is installed, don't worry.
```

```
check_docker()
```

```
> v You are inside a Docker containter!
```

```
# check_github() renders this document irreproducible
# because it relies on user settings unavailable in the container
# hence it is not evaluated
check_github()
```

If everything is set up, users can proceed to reproduce an analysis that conforms to this workflow.

If not, e.g. because Docker is not installed, users get an informative message appropriate for their platform (the following code chunk shows the message windows users get when Git is missing).

```
check_git()
```

```
> x Git is not installed.

> i We recommend Chocolately for Windows users.

> x Chocolately is not installed.

> * To install it, follow directions on:
>   'https://chocolatey.org/docs/installation'

> i Use an administrator terminal to install chocolately.

> * Restart your computer.

> * Run `choco install -y git` in an admin terminal to install Git.
```

3.2 Reproduction

This thesis was written according to the proposed standards using `repro` and may serve as an example of reproduction. I also provide a minimal example that contains a data analysis in the Creation section below.

GitHub, Make, and Docker are sufficient to reproduce this very document. So if you followed the steps above, everything is set up to download the source files of this document, rerun the code within it, and verify its results.

The following command uses Git and GitHub to:

1. create a copy of the project, called "fork", in your GitHub account;

2. download this copy to your computer, and
3. verify that all files are intact and open them in a new RStudio instance.

```r
usethis::create_from_github("aaronpeikert/repro-thesis",
                            tempdir(),
                            fork = TRUE)
```

If executed, this code opens a new R session, and therefore, all code from here on out needs to run in the *new* session.

It is tempting to automate the reproduction part entirely and use a `rerun()` function that figures out what to do and does so for you. However, I decided the reproduction must be feasible without the `repro` package to avoid monopolization of reproducibility by a single software package. This decision is supposed to ensure that long term reproducibility does not depend on the availability of the package. I am confident that Git, Make, Docker will be available for years to come, whereas I cannot say the same about this package. To balance the needs of long term support and usability, `repro` offers advice about what to do, but stops right before doing it (you can see an example after the next code chunk).

Which steps one has to take, depends on the tools chosen to implement dependency management. This tool determines the "entry point" for an analysis. To detect the entry point, `repro` follows simple heuristics, which are informed by what most R users tend to use. These conventions are quite ambiguous, but the most explicit entry point is a `Makefile`. If no `Makefile` is available, the alternatives are either a central dynamic document (RMarkdown, Jupityer Notebook) or a primary script (R, Python, Oktave, Shell). In these cases, one can only guess from filenames like `manuscript.Rmd`, `analysis.Rmd`, `paper.Rmd`, `run.R` or `analysis.R`.

To recreate this document you have to follow the steps below:

```r
# because this is a new R project / session, reload repro
library("repro")
rerun(cache = FALSE)
```

```
> * To reproduce this project, run the following code in a terminal:

>   make docker &&
>   make -B DOCKER=TRUE
```

The argument `cache = FALSE` ensures that everything that can be recreated is recreated even when nothing was changed.

It is challenging to verify wether an analysis was reproduced. As a minimum standard one could demand, that the analysis is rerun free of errors or, as a maximal standard, that the resulting documents are exactly the same. Neither solution strikes the right balance because error-free does not imply the same results. At the same time, the comparison of binary files often leads to spurious differences, e.g. because of numerical instabilities.

Currently, researchers need to revert to manual checking and common sense to verify a successful reproduction. An automated verification procedure would require the researcher to state which results need to be identical explicitly. Then a software solution could track changes for only these digital objects and accordingly flag mismatches.

3.3 Change

For a researcher, reproducing an analysis and verifying its results, is often only a first step to make intentional changes. How researchers contribute to a project lays strictly outside the realm of reproducibility, but warrants discussion because easy collaboration is one of the most significant practical advantages of reproducibility. That the primary beneficiary of this advantage is the researcher collaborating with its past self is a pun in the open science community that bears some truth. However, the workflow of an external researcher contributing is more complicated and hence here described. It is quite a challenge to collaborate under the circumstances where people do not work side by side or even know each other. The core challenge is to allow the original creator full control over changes without burdening them too much. This problem confronted the open software community from its very beginning, and they came up with the following solution. A contributor first creates a public copy, makes and tracks changes to it and then asks the original owner to incorporate the changes. In the terminology of GitHub, the public copy is a "fork", the tracked changes are "commits", and the call for including the changes is a "pull request".

Working with pull requests is easy, thanks to the `usethis` package. If you reproduced this document, you could make changes to it—which could be something trivial, like correcting spelling—and ask me to incorporate them. You can initialize a pull request with:

```r
usethis::pr_init()
```

Subsequently, you may change the files of this thesis as you like and track them with Git. You should make sure that the analysis is still reproducible with:

```r
rerun(cache = TRUE)
```

```
> * To reproduce this project, run the following code in a terminal:
```

```
>   make docker &&
>   make DOCKER=TRUE
```

If you are satisfied with the changes you made, you can trigger the pull request with:

```
usethis::pr_push()
```

If I—as the author of this document—would like to accept, I can incorporate the changes on GitHub. If not, the changes can be discussed in the pull request, or I could make amends before merging them.

Such a distributed workflow allows for a much more controlled way of collaboration as opposed to mailing back and forth or using cloud storage systems. This higher level of control matches the high standards of scientific work. However, even more important is that this kind of collaboration scales well for many collaborators (Git was originally developed for the collaboration on the Linux kernel, where as of 2017 more than 15.000 developers contributed code (The Linux Foundation, 2017)). Empirical studies require a lot of work, which is usually distributed on many shoulders. As the authors carry the responsibility for the overall correctness, they ought to vet every single contribution.

Affirming the correctness of a contribution can be partly automated by confirming successful reproduction. Such automatic checks of changes are part of a software developing process, called continuous integration. Continuous integration runs code in cloud computing environments that asserts the correctness when changes are pushed to GitHub. In many ways, continuous integration is the logical next step for reproducible workflows. Because much effort was already invested in ensuring reproducibility across computers, it is easy to move the analysis to a continuous integration tool.

Hence, if you created a pull request, the continuous integration tool GitHub actions, will rebuild this document, affirming reproducibility and let me see the results of your changes.

3.4 Creation

Reproducing an analysis and creating a reproducible analysis are two very different issues. The `repro` packages main strength lies in simplifying the creation. First, the repro package comes along with a minimal, but comprehensive template including an example RMarkdown, R-script and data. This template can be accessed from within RStudio © via "File" ⟶ "New Project" ⟶ "New Directory" ⟶ "Example repro template", or from any R console via:

```
use_repro_template("path/to/new/project/")
```

`repro` infers the dependencies to data and external code as well as the required packages from the `yaml` metadata of the RMarkdowns. Because analytic data projects have a particular structure, this markup can be much simpler than writing `Dockerfiles` and `Makefiles` manually. While Docker allows installing arbitrary software, an analysis in R likely needs nothing but R-packages. Similarly, Make enables you to run any software, but an analysis in R only needs to execute R-Scripts and render RMarkdowns.

Hence, a minimal addition to the metadata, as can be seen in the following example, contains everything necessary to infer a complete `Dockerfile` and `Makefile`:

```
repro:
  packages:
    - usethis
    - fs
    - aaronpeikert/repro@d09def75df
  scripts:
    - R/clean.R
  data:
    mycars: data/mtcars.csv
```

From this specification, the function `automate()` creates a `Dockerfile` and a `Makefile`, which comply with all recommendations by Peikert & Brandmaier (2019). Strictly speaking, it creates four Dockerfiles and three Makefiles. Most of the files are created in the `.repro` directory and then assembled into the main `Dockerfile`/`Makefile` at top-level. One `Dockerfile` contains the base docker image, including the R version and the current date and another `Dockerfile` contains only the R packages. It also produces one file where the user can amend software installation or set up steps that are not covered by `repro`. The `Makefiles` are similarly separated, with one file dedicated to RMarkdowns and another one for the required logic that executes the make commands in the container.

The `automate()` function is designed to simplify the workflow proposed by Peikert & Brandmaier (2019) as much as possible. Such simplification means inevitably to restrict the user's freedom. While they can still do everything they want in the realm of Make and Docker, this approach does not allow other reproducibility software to be used. Users, which need more control and are more advanced, could instead rely on the modular nature of `repro`. Each component can be added to the project by the `use_*` functions. E.g. `use_make()` adds a basic `Makefile` or `use_make_singularity()` adds a Makefile that is compatible with Singularity (which is an alternative to Docker for High-Performance Computing). These functions extend the usethis-package (Wickham & Bryan, 2020), which was originally designed to facilitate package development with specific reproducibility tools.

3.5 Summary

To summarise, if you want to create a reproducible project, you can do so with the following code:

3.5.1 Install `repro` package

```
if (!requireNamespace(”remotes”)){
  install.packages(”remotes”)
}
remotes::install_github(”aaronpeikert/repro”)
library(”repro”)
```

3.5.2 Check required reproducibility software

```
check_git()
check_github()
check_make()
check_docker()
```

3.5.3 Configure Project

Either from template in new folder:

```
use_repro_template(”path/to/new/folder”)
automate()
```

Or semi automatic with more flexibility in already existing projects:

```
use_docker() # create Dockerfile
use_make_docker() # create docker compatible Makefile
usethis::use_git() # initialize git and add first commit
rmarkdown::draft(”pnas_article”, # use PNAS article template
                 package = ”rticles”) # requires rticles package
```

4 Discussion

This thesis started with a discussion about what functionality reproduction provides for research. I have argued there that the metascientific advantage is to easily validate and repeat the induction process, and the practical benefit is to increase productivity. I then showed how tools and principles from software engineering ensure reproducibility and presented `repro` a package for the R programming language that simplifies their use.

However, I have not focused on its function to archive scientific products, which is sometimes viewed as the traditional functionality of reproducibility. How likely is it that an analysis created with `repro` is still reproducible after decades? A valid point of criticism is that this workflow introduces a bulk of software solutions and services that may not be available for long. Superficially it relies on R, RStudio, Pandoc, Git, Make, Docker, Homebrew (OS X only) and Chocolaty (Windows only) as software and on GitHub, DockerHub, MRAN and Ubuntu Package Archive as services. However, this list of dependencies boils down to one service and one software. That is because all required software is bundled into the container, whose image can be stored. Hence, one needs a storage provider which saves the container image and the other files associated with the project. This problem of longterm storage for scientific data is widely recognised and because of the limited requirements—a typical project (except data) needs around one gigabyte of storage—there are several solutions available (Doorn & Tjalsma, 2007). When storage availability is guaranteed pretty much all long term reproducibility rests on the continued support of container software. To my knowledge, there are at least four different software solutions that can run the here proposed docker images natively (Docker, CoreOS rkt, Mesos Containerizer, Singularity). This diversity in possible solutions makes long term support more likely. Docker itself also prioritises backwards compatibility.[4] However, when long term support is of primary interest, one can convert the docker image to a raw disk image and also archive this. Such strategy opens two possibilities. First, such images are supported by all standard virtualisation software. Such reliance on platform virtualisation is standard practice in disaster recovery plans of technical infrastructure (Maitra et al., 2011) and should hence be available for some time to come. Second, such raw disk images can be directly installed on the hardware, without virtualisation layer. Hence, as long as there is hardware available, that is compatible with the current x64 architecture, this solution still allows reproducibility[5].

[4]The docker image of this project can be executed with the first stable release of Docker from October 16, 2014.

[5]The RAM limit drove the change from x32 to x64 architecture. x32 has an addressable memory of 4 GiB (1 Gigabyte = 2^{30} bit), while x64 has an addressable memory of 16 EiB (1 Exabyte = 2^{60}

The usage of containers to recreate the computational environment relies on the assumption that all required software can be indeed installed within the container. This assumption holds for most open-source software and is straight forward to implement for all software available in the Ubuntu Package Archive. However, even when there is no technical hurdle for commercial software, there might be a legal hurdle imposed by the license agreement. There is an ongoing debate about how compatible commercial closed software is with Open Science in general and reproducibility in particular (Ince et al., 2012), but, e.g. for Matlab, there are three different models for hybrid approaches: First, Mathworks provides a possibility for reusing the host's Matlab license within the container. Second, the MATLAB Coder program allows the cross-compilation of Matlab code to C/C++ and thus its inclusion in open source code. Third, the MATLAB Compiler can compile code to a binary that may be run within the container. Hopefully, similar solutions emerge for other research software with restrictive licenses.

The here presented workflow aspires to an ideal of full transparency, but may in practice require some compromises. A prime example of this are the ethical and legal considerations of publishing data that can only with great difficulty be de-identified. While there are several technical possibilities for this problem, they may require more time and skill than a researcher is willing to invest. The same holds for many other problems that are either field or even project-specific. While such problems are beyond the scope of this thesis, I hope that future research concentrates on field-specific reproducibility problems and their solution, e.g. for longitudinal or neuroimaging data.

These general limitations of the workflow are coupled with some `repro` specific restrictions. The most challenging problem with `repro` is the tradeoff between flexibility and usability. On the one hand, a wide range of possible user requirements call for much freedom and flexibility, but on the other hand, this might be overwhelming for some users. A significant difficulty in this context is the unintended flexibility users get because I decided to keep explicit specification to an absolute minimum. Hence, `repro` needs to infer the current state of the user's project and computer. But because users have unlimited freedom to design a project as they like, it is clear that `repro` can not cover all possibilities. To prevent errors, it is at the moment advisable to start with the provided template and customise it. As `repro` majors and is applied in more use-cases, it will become more flexible concerning the user's setup. Till then defensive programming techniques were employed to detect and friendly report such problems.

bit). Even when the time has come that computer architecture that is compatible with x64 is not available anymore, it is to be expected that this architecture can be emulated with platform virtualisation similarly to the current support of x32 on x64 only computers.

Although the `repro` package exports 35 well-documented functions, assures their correctness by close to 200 unit tests and amasses all in all short of 2000 lines of code, many workflows and features are incomplete. I hope to make progress in three areas:

1. Building an infrastructure for other workflows,
2. explore "continues integration" (CI) for research projects, and
3. enable the leverage of high-performance computing (HPC) clusters and cloud infrastructure.

Because the user interface of `repro` was modelled after the `usethis` package, `repro` also inherited `usethis`' modular structure. This design decision was made more by accident than by intention but is in hindside more than useful. Because of its modularity, `repro` can be extended easily and can serve as an infrastructure package. This use-case will be explored in collaboration with Caspar van Lissa, who proposes a workflow, called `worcs` (Van Lissa et al., 2020), which is similar in spirit, but uses different tools. `worcs` also utilises RMarkdown and Git, but in place of Docker, it uses `renv` and instead of Make it relies on a highly standardised file structure. Hence, only two "moduls" or rather functions have to be added to `repro` to support the workflow, `use_renv()` (which replaces `use_docker()`) and `use_worcs_template()` (which replaces `use_repro_template()`). Also, `worcs` has other features like automatic codebook creation and synthetic data generation, which may be excellent supplements for a `repro` project. To ensure interoperability between the packages, it is planned that much of the backend is moved from `worcs` to `repro` in a more modular form. However, the users will not notice much of a difference, except that they may fuse both workflows into one.

Such a Lego system of reproducibility tools, where the users can decide which tools they want to include may be especially useful when some features are not strictly necessary for reproducibility. In its current form, no tool is optional in the sense that only in unison they guarantee reproducibility. However, some features may be useful but not necessary for reproducibility. For example, is it useful to have an online service such as GitHub Actions that asserts reproducibility of each change, but it is by no means necessary. As previously discussed there currently exists no easy solution to verify reproduction automatically. The key difficulty is to detect changes in the results that potentially alter the conclusion, from those which carry no meaning, e.g. if the date is updated. To address this challenge, I currently work on a prototype of an R package, which allows the researcher to assert that some objects remain unchanged upon reproduction, by wrapping them into `unchanged()`. This package keeps track of thus marked objects and throws an error if they accidentally change. If such a solution existed, there would be no hurdle to incorporate continues integration into research projects. Continues integration could vet all contributions automatically before they are incorporated and update the results, e.g. the preprint or the additional material on osf.org. Then projects could have a badge that signifies that a third party is able to reproduce it. Such continues updates do not only ease collaboration but are also especially interesting for some continues data sources, e.g. long-running developmental studies or meta-analysis, where new data becomes repeatedly available even after publication.

Conveniences such as the above are possible because the here presented conception of reproducibility is highly automated. Hence, an analysis may be moved to another computer without manual intervention. This functionality also opens some exciting possibilities unrelated to reproducibility, concerning the management of computational resources. It is often the case that an analysis requires substantial amounts of computational resources, more than a single computer may deliver. In such a case, a here described analysis can easily be moved to a more powerful computer or even spread across hundreds. The use of containers is the de facto standard of cloud computing providers but also becomes increasingly common for high-performance clusters. Hence it is a task that can easily be accomplished with `repro`. However, these functions are still in development and not tested well enough to be published. A typical problem, when utilising distributed computation, is the division of tasks between the computing instances. However, this is pretty straightforward because of the deployed dependency management. Make can distribute tasks across thousands of nodes while making sure that none of the dependencies collide. Initially, this functionality was available in `repro` for TORQUE, an HPC task scheduler, but because TORQUE will no longer be maintained I will phase out this set of functions.

`repro` strives to make reproducibility more attractive by lowering the barriers to advanced reproducibility tools. To my knowledge, no standard is as comprehensive as the here described combination of Git, RMarkdown, Make and Docker. Despite these efforts to make reproducibility easier, I think it is worthwhile to simplify the approaches further, to appeal to users that are more comfortable using Microsoft Office than RMarkdown and hesitant to use a formal version control system. Hopefully, `repro` is the first step to make reproducibility easier to achieve.

References

Andersen, H., & Hepburn, B. (2016). Scientific method. In E. N. Zalta (Ed.), *The stanford encyclopedia of philosophy* (Summer 2016). https://plato.stanford.edu/archives/sum2016/entries/scientific-method/; Metaphysics Research Lab, Stanford University.

Announcement: Reducing Our Irreproducibility. (2013). *Nature*, *496*(7446), 398–398. https://doi.org/10.1038/496398a

Boulton, G., Campbell, P., Collins, B., Elias, P., Hall, W., Laurie, G., O'Neill, O., Rawlins, M., Thornton, J., & Vallance, P. (2012). Science as an open enterprise. *The Royal Society*.

Brandmaier, A. M., von Oertzen, T., McArdle, J. J., & Lindenberger, U. (2013). Structural equation model trees. *Psychological Methods*, *18*(1), 71–86. https://doi.org/10.1037/a0030001

Claerbout, J. F., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. *SEG Technical Program Expanded Abstracts 1992*, 601–604. https://doi.org/10.1190/1.1822162

Deutsche Forschungsgemeinschaft. (2019). *Leitlinien zur Sicherung guter wissenschaftlicher Praxis*. https://www.dfg.de/download/pdf/foerderung/rechtliche_rahmenbedingungen/gute_wissenschaftliche_praxis/kodex_gwp.pdf

Doorn, P., & Tjalsma, H. (2007). Introduction: Archiving research data. *Archival Science*, *7*(1), 1–20.

Epskamp, S. (2019). Reproducibility and replicability in a fast-paced methodological world. *Advances in Methods and Practices in Psychological Science*, *2*(2), 145–155. https://doi.org/https://doi.org/10.1177/2515245919847421

Gigerenzer, G., Krauss, S., & Vitouch, O. (2004). The Null Ritual: What You Always Wanted to Know About Significance Testing but Were Afraid to Ask. In D. Kaplan, *The SAGE Handbook of Quantitative Methodology for the Social Sciences* (pp. 392–409). SAGE Publications, Inc. https://doi.org/10.4135/9781412986311.n21

Gilbert, S. W. (1991). Model building and a definition of science. *Journal of Research in Science Teaching*, *28*(1), 73–79. https://doi.org/10.1002/tea.3660280107

Heilbron, J. L. (Ed.). (2004). The Oxford Companion to the History of Modern Science. *Reference Reviews*, *18*(4), 40–41. https://doi.org/10.1108/09504120410535443

Hutson, M. (2018). Artificial intelligence faces reproducibility crisis. *Science*, *359*(6377), 725–726. https://doi.org/10.1126/science.359.6377.725

Ince, D. C., Hatton, L., & Graham-Cumming, J. (2012). The case for open computer programs. *Nature*, *482*(7386, 7386), 485–488. https://doi.org/10.1038/nature10836

Ioannidis, J. P. A. (2005). Why Most Published Research Findings Are False. *PLOS Medicine*, *2*(8), e124. https://doi.org/10.1371/journal.pmed.0020124

Jacobucci, R., Brandmaier, A. M., & Kievit, R. A. (2019). A Practical Guide to Variable Selection in Structural Equation Modeling by Using Regularized Multiple-Indicators, Multiple-Causes Models. *Advances in Methods and Practices in Psychological Science*, *2*(1), 55–76. https://doi.org/10.1177/2515245919826527

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255–260. https://doi.org/10.1126/science.aaa8415

Kraker, P., Leony, D., Reinhardt, W., Gü, N., & Beham, nter. (2011). The case for an open science in technology enhanced learning. *International Journal of Technology Enhanced Learning*, *3*(6), 643. https://doi.org/10.1504/IJTEL.2011.045454

Maitra, S., Shanker, M., & Mudholkar, P. K. (2011). Disaster recovery planning with virtualization technologies in banking industry. *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, 298–299. https://doi.org/10.1145/1980022.1980089

Martin, R. C. (2011). *The clean coder: A code of conduct for professional programmers / Robert C. Martin* (1. print.). Prentice Hall.

Maxwell, S. E., Lau, M. Y., & Howard, G. S. (2015). Is psychology suffering from a replication crisis? What does “failure to replicate” really mean? *American Psychologist*, *70*(6), 487.

Meehl, P. E. (1990). Appraising and Amending Theories: The Strategy of Lakatosian Defense and Two Principles that Warrant It. *Psychological Inquiry*, *1*(2), 108–141. https://doi.org/10.1207/s15327965pli0102_1

Meehl, P. E. (1978). Theoretical risks and tabular asterisks: Sir Karl, Sir Ronald, and the slow progress of soft psychology. *Journal of Consulting and Clinical Psychology*, *46*(4), 806–834. https://doi.org/10.1037/0022-006X.46.4.806

Müller, K. (2017). *Here: A simpler way to find your files.* https://CRAN.R-project.org/package=here

Nosek, B. A., Ebersole, C. R., DeHaven, A. C., & Mellor, D. T. (2018). The preregistration revolution. *Proceedings of the National Academy of Sciences*, *115*(11), 2600–2606. https://doi.org/10.1073/pnas.1708274114

Nuijten, M. B., Hartgerink, C. H. J., van Assen, M. A. L. M., Epskamp, S., & Wicherts, J. M. (2016). The prevalence of statistical reporting errors in psychology (1985–2013). *Behavior Research Methods*, *48*(4), 1205–1226. https://doi.org/10.3758/s13428-015-0664-2

Ooms, J. (2019a). *Credentials: Tools for managing ssh and git credentials*. https://CRAN.R-project.org/package=credentials

Ooms, J. (2019b). *Gert: Simple git client for r*. https://CRAN.R-project.org/package=gert

Open Science Collaboration. (2015). Estimating the reproducibility of psychological science. *Science*, *349*(6251), aac4716–aac4716. https://doi.org/10.1126/science.aac4716

Parasuraman, R., & Mouloua, M. (2018). *Automation and Human Performance: Theory and Applications*. Routledge.

Pashler, H., & Wagenmakers, E. (2012). Editors' Introduction to the Special Section on Replicability in Psychological Science: A Crisis of Confidence? *Perspectives on Psychological Science*, *7*(6), 528–530. https://doi.org/10.1177/1745691612465253

Peikert, A., & Brandmaier, A. M. (2019). *A Reproducible Data Analysis Workflow with R Markdown, Git, Make, and Docker* [Preprint]. PsyArXiv. https://doi.org/10.31234/osf.io/8xzqy

Peikert, A., Brandmaier, A. M., & van Lissa, C. J. (2020). *Repro: Automated setup of reproducible workflows and their dependencies*. https://github.com/aaronpeikert/repro

Popper, K. R. (1962). Some comments on truth and the growth of knowledge. In E. Nagel, P. Suppes, & A. Tarski (Eds.), *Logic, Methodology and Philosophy of Science Proceedings of the 1960 International Congress* (Vol. 155). Stanford University Press.

R Core Team. (2020). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Rodgers, J. L. (2010). The epistemology of mathematical and statistical modeling: A quiet methodological revolution. *American Psychologist*, *65*(1), 1–12. https:

//doi.org/10.1037/a0018326

The Linux Foundation. (2017, October 25). *2017 Linux Kernel Report Highlights Developers' Roles and Accelerating Pace of Change.* https://www.linuxfoundation.org/blog/2017/10/2017-linux-kernel-report-highlights-developers-roles-accelerating-pace-change/

The Turing Way Community, Arnold, B., Bowler, L., Gibson, S., Herterich, P., Higman, R., Krystalli, A., Morley, A., O'Reilly, M., & Whitaker, K. (2019). *The Turing Way: A Handbook for Reproducible Data Science.* https://doi.org/10.5281/zenodo.3233986

Tichý, P. (1976). Verisimilitude redefined. *The British Journal for the Philosophy of Science*, *27*(1), 25–42.

Van Lissa, C. J., Brandmaier, A. M., Brinkman, L., Lamprecht, A.-L., Peikert, A., Struiksma, M., & Vreede, B. (2020). *WORCS: A Workflow for Open Reproducible Code in Science* [Preprint]. PsyArXiv. https://doi.org/10.31234/osf.io/k4wde

Wickham, H., & Bryan, J. (2020). *Usethis: Automate package and project setup.* https://CRAN.R-project.org/package=usethis

Williams, J. M. (2017). *Style: Lessons in clarity and grace* (Twelfth Edition). Pearson.

Xie, Y., Allaire, J. J., & Grolemund, G. (2019). *R Markdown: The definitive guide.*

Yarkoni, T., & Westfall, J. (2017). Choosing Prediction Over Explanation in Psychology: Lessons From Machine Learning. *Perspectives on Psychological Science*, *12*(6), 1100–1122. https://doi.org/10.1177/1745691617693393