How reproducible should research software be?

Sam Harrison*[1] [0000-0001-8491-4720], Abhishek Dasgupta[2] [0000-0003-4420-0656], Simon Waldman[3] [0000-0003-4811-398X], Alex Henderson[4], Christopher Lovell[5]

[1] UK Centre for Ecology & Hydrology, UK
[2] University of Oxford, UK
[3] University of Hull, UK
[4] University of Manchester, UK
[5] University of Hertfordshire, UK

*Corresponding author: sharrison@ceh.ac.uk

In this article, we attempt to answer this question by defining four levels of reproducibility, suggesting criteria to help you decide which level your research software should be at, and recommended practices to reach these levels of reproducibility.

Introduction

Reproducibility is central to science, as it allows replication and verification of results. This can present particular challenges for computational science, due to the complexity of some scientific software: while the process or algorithm being used can, and should, be clearly documented, the number of implementation details and dependencies involved with software means that this may not be enough to allow another researcher to exactly reproduce the “experiment”. In an attempt to address this, journals are increasingly requiring code to be provided with submissions. This is desirable for transparency, but it is probably not reasonable to expect reviewers to have the time or expertise to routinely review all aspects of a complex code submission.

We suggest that there should be three priorities in scientific software quality to promote reproducibility of results: that code is *correct,* that it is *reusable,* and that it is *documented.* Code should be *correct* so that we can have confidence that it is producing scientifically accurate results. This implies confidence that there are no significant bugs. Code should be *reusable* so that people — the original authors or others — can use it again and can build upon it in the future. Code should be *documented* to contribute to its reusability, but also to provide a clear record of the science that has been done, at a level of detail that cannot be attained in a manuscript.

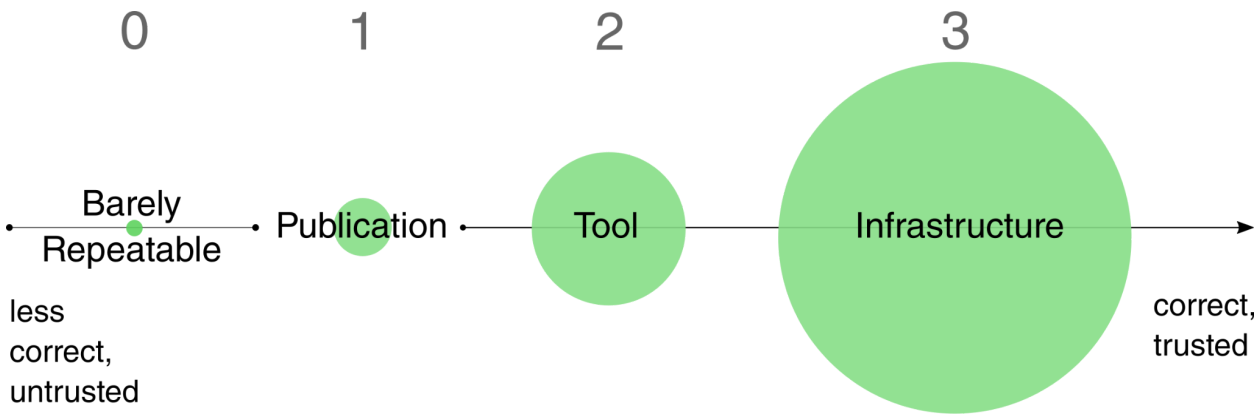
Various suggestions have been made in the past as to what practices should be adopted in scientific software to ensure correctness, reusability, documentation, and thus reproducibility (e.g. Wilson et al, Lee et al and Pawlik et al). However, these measures all have a cost, and when considering cost vs benefit it becomes clear that the appropriate level of effort in this regard varies markedly according to the intended application of the software: what is good enough for one research software project may be overkill for another, but insufficient for a third. Here, we suggest categorising software into levels and discuss the measures that may be appropriate to ensure reproducibility at each level.

Be FAIR: FAIR data, and more recently FAIR software, are emerging as key goals for any research project. FAIRness is defined as how Findable, Accessible, Interoperable and Reusable data/software are. Though FAIRness is broader than just reproducibility, the themes discussed and best practices recommended in this article are integral to achieving FAIRness.

Levels of reproducibility

Defining reproducibility by categorical levels makes it easier to target best practices and resources to help meet these best practices for each level individually. Anyone who writes any type of software, from small scripts to large-scale distributed software, could use these levels to identify how reproducible their software should be and learn how they can achieve this reproducibility. By giving structure to how reproducibility is defined, we hope to make it easier for researchers to consider the practical steps to take when developing their software, rather than - as can often be the case at present - getting lost in a myriad of definitions, best practices and resources. Here we suggest four levels of reproducibility and discuss best practices and resources to meet these levels.

Repeatability vs replicability vs reproducibility: These three Rs of reproducibility have historically been used, somewhat inconsistently, to give structure to reproducibility. However, their definitions vary, the fact they all begin with the same three letters makes them difficult to remember, and they are too broad to encompass some of the nuances of research software. Here, we prefer to give better defined structure in our four numerical levels.



Circle size indicates research software impact

Levels of reproducibility. The x-axis represents how correct the software is (i.e. how confident we are in the scientific accuracy of the results it produces), whilst the size of the circles represent the extent of research software impact (i.e. how large are the consequences of the code being incorrect).

Level 0: Barely repeatable

The absolute baseline of reproducibility is that software should be re-runnable on the same computer by the same person such that it produces the same results that are in line with what is expected. This level is for simple code intended for single or occasional use by one individual, which would not be of use to others (i.e. there is no benefit to making the code openly available). This could include simple data parsing scripts or proof of concept software (testing ideas/conceptualisations before implementing or building into other software). As soon as the software gets complex enough that you can't understand it at a glance, then you are no longer at Level 0.

Best practices:

- Ensure code is clearly written. Use best practices and conventions for the language you are writing in, e.g. PEP 8 for Python.
- Perform basic testing by validation to identify bugs in the software.
- Consider writing simple documentation. Keep this alongside the code (e.g. a README file in the same directory as the code) and as in-code documentation rather than storing in a place you're likely to forget. The primary user of this documentation will be the author's future self.
- Consider using some form of version control. This might just be taking a snapshot of the script with the output, but dedicated tools such as Git could be useful.

Level 1: Research Software for Publication

This level forms the baseline of reproducibility for what we commonly refer to as research software - anything above the simple scripts that might fall under Level 0. This level represents the bare minimum required for software used in publications. Over and above Level 0's

requirement for replicability on the same computer, software at this level should be portable to different computers, but platform-dependence is acceptable (e.g. software that only runs on Linux or Windows). There is no requirement for the software to be maintained over a long period of time as it will only be run a few times. The key goal of this level is to facilitate trust in results obtained from research software, for example when submitting results to a journal article. If the software becomes useful to more than a handful of people, if it is used with lots of different input data, or if reuse over any significant period of time is required, then it should be at a higher level of reproducibility. Examples of software at this level include data processing, analysis and visualisation scripts, which help prepare data for use elsewhere, for example input to other software or in generating results/visualisations for journal articles.

Best practices - everything from Level 0, plus:

- Code should be available, ideally under an open source license, and particularly if the code contributes to a journal article. Good open source licenses to consider include the MIT license, Apache 2.0 and GNU General Public License v3.
- Good commenting and readability of code is crucial. Write code in the knowledge that code is read more often than it is written.
- Documentation of the software should be at least to the level such that other users can easily run the software and replicate your results, but preferably deeper than this. For example, what methods/algorithms were used, and which libraries and other dependencies were used, and in which versions, for the published result?
- Version control should definitely be used. Where multiple people are making significant contributions to the code, agree upon a common workflow (e.g. Git Flow).
- Testing should be carried out, though comprehensive unit testing might not be required at this level.

Level 2: Research Software as a Tool

Software at Level 2 is probably more complex than at Level 1, and rather than being created for one-off use, e.g. for processing a single dataset for a single publication, it is intended to be a tool that is applied to different inputs or scenarios over a modest period of time. The expectation remains that the user is either the original developer, or somebody else with the skills to understand and maintain or modify the code themselves.

Best practices - everything from Level 1, plus:

- Documentation should include detailed information on architecture, build system, compiler choice and build flags. It is unrealistic to expect software to be tested on all possible systems (especially proprietary ones), so documentation needs to be explicit as to what systems it has been tested on.
- Where research publications rely on this code for significant scientific calculation, a snapshot should be archived with a DOI and cited in the paper.
- Pay attention to dependencies, keeping track of versions so you don't get caught out by upstream changes. Choose sustainable, well-supported software that is unlikely to be deprecated or disappear. Read about the left-pad debacle for inspiration as to why this is important.

- Code should be reviewed by others to minimise the chance of mistakes and bugs.
- Consider continuous integration for complex software, including regression and integration testing.
- Consider unit testing.
- Consider containerisation of code, data, configuration and architecture to create snapshots of the software and data used to create research results. Conda, pipenv and Docker are examples of tools to reproduce computing environments, and Zenodo can be used to archive these snapshots.

Level 3 - Research Software as Infrastructure

The defining feature of Level 3 is that the software will be involved in multiple outputs, by multiple groups and likely have multiple developers over an extended period of time. Rather than just code used to complete one or a few specific tasks, software at this level is broadly used by the whole community, and thus should be thought of and maintained as infrastructure. It is likely to be used as a tool by people who do not fully understand how it works and cannot fix problems themselves. Because of its broad use, a critical bug in software at this level would potentially invalidate numerous scientific papers and outputs. Correctness, reusability and documentation are fundamental. This level is broad, at its lower end including “standard tools” used in a small field and in its upper reaches encompassing ubiquitous software like NumPy and SciPy.

Best practices - everything from Level 2, plus:
Note that a lot of optional practices from lower levels are now mandatory at Level 3.

- A formal testing regime including unit testing, regression and integration testing, through a continuous integration system.
- A formal code review process. Consider pair programming for complex or critical parts.
- A formalised workflow with version control (e.g. Git Flow or similar).
- Semantic versioning should be used to allow users to adapt to breaking changes, and this should be supplemented with a changelog for the same purpose. Consider creating a release roadmap.
- Long term archiving and persistent URLs/identifiers, such as DOIs, should be used to ensure that legacy versions, on which other people’s software or publications may depend, are available. Zenodo can be used to create a DOI for GitHub repositories.
- Consider making code available in community registries such as CRAN, Pip or Conda.
- Consider making your software as platform-independent as possible. There may be exceptions to this, such as software that needs to be optimised for use on particular architectures and high performance computers, but careful consideration should be given to if this dependence is justified.
- Consider long term funding and governance from the outset - you don’t want to spend the time and effort creating a piece of software that is soon deprecated. If appropriate, the support of organisations like NumFOCUS could be considered.

It may be common for Level 2 software to move to Level 3 over time, as it becomes more widely adopted than was originally envisioned. This can present challenges in “upgrading” the practices that are applied, especially where maintenance funding is required.

Beyond Level 3, software uses and priorities broaden out into multiple directions, which is beyond the research software focus of this article. For example, this might include software where very long-term reproducibility, in the order of decades, is required (e.g. instrumentation control in space missions), or safety-critical software, where bugs are not just inconvenient but may cost lives (e.g. nuclear or medical software).

We hope that the above four levels give enough structure and definition to reproducibility to reduce it from being a daunting, confusing and open-ended task, into tangible strategies you can apply to your own research software. Of course, this article isn’t exhaustive, and these levels should be used as a foundation to be built upon with best practices more pertinent to individual research areas and software tasks.

Author contributions: SH, AD and SW wrote the article. SH, AD, AH, and CL contributed to the initial discussion session at the Software Sustainability Institute Fellows Online Selection Day 2021.