Ten Simple Rules for Making Research Software More Robust

Morgan Taschuk[1,‡*], Greg Wilson[2,‡]
1 Ontario Institute for Cancer Research / morgan.taschuk@oicr.on.ca
2 Software Carpentry Foundation / gvwilson@software-carpentry.org

‡ These authors contributed equally to this work.
* Corresponding author.

Abstract

Software produced for research, published and otherwise, suffers from a number of common problems that make it difficult or impossible to run outside the original institution, or even off the primary developer's computer. We present ten simple rules to make such software robust enough to run anywhere, and inspire confidence in your reproducibility, and thereby delight your users and collaborators.

Notes

Author Summary

Many researchers have found out the hard way that there's a world of difference between "works for me on my machine" and "works for other people on theirs". Many common challenges can be avoided by following a few simple rules; doing so not only accelerates research, but also improves reproducibility.

Introduction

Typically, scientific software is developed and used by a single person, usually a graduate student or postdoc, and produces the intended results in their hands [1]. But what happens when someone else wants to use their software? Everyone with a few years of experience feels a bit nervous when told to use another person's code to analyze their data. Often, that software will be undocumented and work in unexpected ways (if it works at all). It will often rely on nonexistent paths or resources, be tuned for a single dataset, or simply be an older version than was used in published papers. The new user is then faced with two unpalatable options: hack the existing code to make it work, or start over. Being unable to replicate results is so common that one publication refers to it as "a rite of passage" [2].

The root cause of this problem is that most research software isn't *robust*. The difference between running and being robust is the difference between "works for me on my machine" and "works for other people on a cluster I've never used". The lack of robustness also leads to many duplicated efforts, slowing the pace of research [3, 4]. In bioinformatics, repositories such as the ELIXIR tools and data registry [5] and the Bioinformatics Links Directory [6] catalogue dozens to hundreds of tools to perform the same task. In the Bioinformatics Links Directory as of 2016, there are 84 different

multiple sequence aligners, 141 tools to analyze transcript expression, and 182 pathway and interaction resources. Some of these tools are legitimate efforts to improve the state-of-the-art, but often they are difficult to install and run [7,8], and are effectively abandoned after publication [9].

This problem is not unique to bioinformatics, or even to computing [2]. Best practices in software engineering specifically aim to increase software robustness. However, most bioinformaticians come from related disciplines and are trained on-the-job or otherwise informally [1,10]. Even the existing training programs and initiatives rarely have the time to cover software engineering in-depth, especially since the field is so broad and rapidly developing [4,10]. Regardless of whether or not the skills and knowledge are taught, developing robust software is not directly rewarded in science and funding is difficult to come by [1]. Some proposed solutions to this problem, in addition to restructuring educational programs, include hiring dedicated software engineers [4,11], partnering with private sector or grassroots organizations [1,5], or using specific technical tools like containerization or cloud computing [12,13]. Each of these requires time and, in some cases, institutional change.

You don't need to be a full-time programmer to write robust software. In fact, some of the best, most reliable pieces of software in many scientific communities are written by researchers [3,11]. The most widely-used software packages tend to adopt strong software engineering approaches, have high standards of reproducibility, good testing practices, and inter-institutional collaborations that grow into strong user bases. In the bioinformatics community, Bioconductor and Galaxy follow this path [12,14].

That said, not *every* coding effort needs to be engineered to last. Code that is used once to answer a specific question related to a specific dataset doesn't require comprehensive documentation or flexible configuration, and the only sensible way to test it may well be to run it on the dataset in question. Exploratory analysis is an iterative process that is developed quick and revised often [4,11]. However, if a script is dusted off and run three or four times for slightly different purposes, is crucial to a publication or a lab, or being passed on to someone else, it may be time to make your software more robust.

In particular, robust software:

- works for users other than the original author;
- is kept under version control;
- can be installed on more than one computer or account with relative ease;
- has well-defined input and output formats;
- has documentation that describes what its dependencies are, how to install it, and what the options are; and
- comes with enough tests to show that it actually runs.

These simple rules are also necessary steps toward creating a reusable library that can be shared and reused through a site like CPAN or CRAN. They are generic and can be applied to all languages, libraries, packages, documentation styles, and operating systems for both closed-source and open-source software. Whether the aim is as simple as sharing the code with collaborators or as complex as using the software in a production analysis environment, increasing the robustness of your software decreases headaches all around.

1 Have a README that explains in a few lines what the software does and what its dependencies are.

The README is the first stop for most new users. At a minimum, it needs to get a new user started and point them towards more help, if they need it. Numerous guidelines exist on how to write good READMEs [15, 16]. We summarize the crucial points here.

Explain what the software does: At the beginning of the README, explain what the software does in one or two sentences. There's nothing more frustrating than spending the time to download and install some software only to find out that it doesn't do what you thought it did.

List required dependencies: Often, software depends on specific versions of libraries, modules, or operating systems. This is entirely reasonable as long as it is properly documented. Include the name and version number for each dependency. Several methods exist for acquiring required packages: either provide the commands necessary to download the dependencies; link to the software homepage; or (our preference) use a dependency manager. We address dependencies in more detail in rule 5.

Provide compile/installation instructions: If the software needs to be compiled or installed, list those instructions in the README. Also mention if you recommend they use a pre-compiled binary instead through a system such as pip, yum or apt.

List input and output files: All possible input and output files should be listed in this section. Do the files conform to a particular standard, an extension of an existing format, or is it your own format? If using a standard format, link to the specification and version. If you extend the standard or have your own format, define it explicitly, listing all the required fields and acceptable values. (You get bonus points if you include a script to convert between a standard format and your file format). If there is no rigorous format (such as with log files), show an example file, or the first few lines, and explain what the sections mean.

Input files and their formats are included in most documentation, but intermediate, auxiliary and log files are often missing. *All* files should be listed in the README, even those considered self-explanatory. Log files are often full of valuable information that can be mined for the user's specific purpose. If your users might need to know, "Does this program report the percentage of reads trimmed to remove adapter sequences?" they should be able to check the README and confidently say, "Yes, it is in the log file".

State attributions and licensing: Attributions are how you credit your main contributors; licenses are how you want others to use and credit your software. READMEs are usually available *before* the software is installed (or even downloaded). Leave no question in anyone's mind about whether your software can be used commercially, how much modification is permitted, and how other software needs to credit you. If your software is not open source, state that clearly. Attributions can also contain a list of "expert" users that can be contacted if new users have problems with the software.

The README file for khmer[1] is a good model: it explains the software's purpose, tells readers where to get help, and includes links to a CITATION file (explaining how to cite the project) and the license file.

[1] `https://github.com/dib-lab/khmer/blob/master/README.rst`

2 Print usage information when launching from the command line that explains the software's features.

Usage information provides the first line of help for both first-time and experienced users of command-line applications. Ideally, usage is a terse, informative command-line help message that guides the user in the correct use of your software. Terseness is important: usage that extends for multiple screens is a nuisance, especially when printed to standard error instead of standard output (where it can easily be paged).

Usage should provide all of the information necessary to run the software. It is invoked either by running the software without any arguments; running the software with incorrect arguments; or by explicitly choosing a help or usage option. More standard command-line behaviours are detailed in [8].

Almost all command-line applications use a combination of POSIX [17] and GNU [16] standards for usage. An example of good usage is GNU's `mkdir` command, which makes new directories:

```
$ mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as needed
  -v, --verbose     print a message for each created directory
  -Z, --context=CTX  set the SELinux security context of each created
                       directory to CTX
      --help     display this help and exit
      --version  output version information and exit

Report mkdir bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
For complete documentation, run: info coreutils 'mkdir invocation'
```

The syntax for running the program: This includes the name of the program and defines the relative location of optional and required flags, arguments and values for execution. Arguments in [square brackets] are usually optional. An ellipsis (...e.g. "[OPTION]...") indicates that more than one value can be provided.

Description: Similar to the README, the description reminds users of the software's primary function.

Most commonly used arguments, a description of each, and the default values: Not all arguments need to appear in the usage, but the most commonly used ones should be listed here. Users will rely on this for quick reference when working with your software.

Where to find more information: Whether an email address, web site or manual page, there should be an indication where the user can go to find more information about the software.

Printed to standard output : So that it can be piped into `less`, searched with `grep`, or compared to the previous version with `diff`.

Exit with an appropriate exit code: When usage is invoked by providing incorrect arguments, the program should exit with a non-zero code to indicate an error. However, when help is explicitly requested, the software should not exit with an error.

3 Version your releases

Software evolves over time, with developers adding or removing features as need dictates. Making official releases stamps a particular set of features with a project-specific identifier so that version can be retrieved for later use. For example, if a paper is published, the software should be released at the same time so that the results can be reproduced. Most software has a version number composed of a decimal number that increments as new versions are released.

There are many different ways to construct and interpret the version, but most importantly for us, a particular software version run with the same parameters should give identical results no matter when it's run. Results include both correct output as well as any errors. **Increment your version number every time you release your software to other people**.

Semantic versioning[2] is one of the most common types of versioning for open-source software. Version numbers take the form of *MAJOR.MINOR[.PATCH]*, e.g., 0.2.6. Changes in the major version number herald significant changes in the software that are not backwards compatible, such as changing or removing features or altering the primary functions of the software. Increasing the minor version represents incremental improvements in the software, like adding new features. Following the minor version number can be an arbitrary number of project-specific identifiers, including patches, builds and qualifiers. Common qualifiers include `alpha`, `beta`, and `SNAPSHOT`, for applications that are not yet stable or released, and `-RC` for release candidates prior to an official release.

The version of your software should be easily available by supplying `--version` or `-v` on the command line. This command should print the software name and version number, and it should also be **included in all of the program's output**, particularly debugging traces: if someone needs help, it's important that they be able to tell whoever's helping them which version of the software they're using.

While new releases may make a program better in general, they can simultaneously create work for someone who integrated the old version into their own workflow a year or two ago, and won't see any benefits from upgrading. A program's authors should therefore **ensure that every released version continues to be available.** A number of mechanisms exist for controlled release that range from as simple as adding an appropriate commit message or tag to version control, to official releases alongside code on Bitbucket or GitHub, to depositing into a repository like apt, yum, homebrew, CPAN, etc. Choose the method that best suits the number and expertise of users you anticipate.

4 Reuse software (within reason).

In the spirit of code reuse and interoperability, developers often want to reuse software written by others. With a few lines, a call is made out to the other program and the results are incorporated into the primary script. Using popular projects reduces the amount of code that needs to be maintained and leverages the work done by the other software.

Unfortunately, reusing software introduces dependencies, which can bring their own special pain. The interface between two software packages can also be a source of considerable frustration: all too often, support requests descend into debugging errors produced by the other project. In addition, every package someone has to install before being able to use yours is a possible (some would say "likely") source of frustration for some potential user. The way the second program is invoked can throw up errors. The

[2] `http://semver.org/`

program may not be in the user's path, or it may be an older or newer version and produce results different than expected. Windows users will be frustrated if you invoke `bash` or `sh` explicitly or use shell-specific conventions like '*' expansion. Even Linux-standard functions available vary slightly between installs. For example, GNU `sort` is available on almost every *nix distribution, but sorts differently depending on locale.

Despite these problems, software developers in research should re-use existing software wherever possible and reasonable to avoid "reinventing the wheel". We fully support the reuse of software between projects provided three additional guidelines are adhered to.

First, **make sure that you really need the auxiliary program**. If you are executing GNU sort instead of figuring out how to sort lists in Python, it may not be worth the pain of integration.

Second, **ensure the appropriate software and version is available**. Early in execution, check whether the dependency is executable and whether the version is compatible. Either allow the user to configure the exact path to the package, distribute the program with the dependent software, or download it during installation using a dependency management system (see Rule 5).

Finally, to ensure support on as many different operating systems as possible, **use native functions for starting other processes**, such as Java's `Runtime.exec` call, Python's subprocess module, and Perl's system command, and be sure to capture and report the output of the subprocess's standard error to facilitate debugging.

5 Use a build utility and package manager.

In all but the simplest of scripts, **we highly recommend using a build automation utility to compile code and/or manage your software workflow**. The benefits of using a build utility like Make, Rake, Maven, Ant or MS Build vastly outweighs the drawbacks associated with the learning curve. Most importantly, a build system lets someone else build your software the same way you would. Most build utilities can also be configured to run tests during every build and re-generate documentation, and can be hooked into continuous integration, cutting releases, and deploying the software to other environments.

Build utilities rely on any dependencies being present in the environment for a successful build, and so developers should **document *all* dependencies, preferably in a machine-readable form**. Package managers like apt and yum are available on most *nix systems, and application package managers exist for specific languages like Python (pip), Java (Maven/Gradle), and Ruby (RubyGems). These package managers can be used together with the build utility to ensure that dependencies are available at compile/run time. For example, it is common for Python projects to include a file called `requirements.txt` that lists the names of required libraries, along with version ranges:

```
requests>=2.0
pygithub>=1.26,<=1.27
python-social-auth>=0.2.19,<0.3
```

This file can be read by pip package manager, which can check that the required software is available, and install it if it is not. Whatever is used, we recommend that developers should *always* install dependencies using this mechanism, especially on their personal machines, so that they're sure it works.

Conversely, developers should **avoid depending on scripts and tools which are not available as packages**. In many cases, a program's author may not realize

that some tool was built locally, and doesn't exist elsewhere. At present, the only sure way to discover such unknown dependencies is to install on a system administered by someone else and see what breaks. As use of virtualization containers becomes more widespread, software installation can also be tested on a virtual machine or container system like Docker.

6 Do not require root or other special privileges.

Root (also known as "superuser" or "admin") is a special account on a computer that has (among other things) the power to modify or delete system files and user accounts. Conversely, files and directories owned by root usually cannot be modifed by normal users.

Installing or running a program with root privileges is often convenient, since doing so automatically bypasses all those pesky safety checks that might otherwise get in the user's way. However, those checks are there for a reason: scientific software packages may not intentionally be malware, but one small bug or over-eager file-matching expression can certainly make them behave as if they were. Outside of very unusual circumstances, **packages should not require root privileges to set up or use**.

Another reason for this rule is that users may want to try out a new package before installing it system-wide on a cluster. Requiring root privileges will frustrate such efforts, and thereby reduce uptake of the package. Requiring that software be installed under its own user account (e.g., that `packagename` be made a user, and all of the package's software be installed in that "user's" space (see: Apache Tomcat)) is similarly limiting, and makes side-by-side installation of multiple versions of the package more difficult.

Developers should therefore **allow packages to be installed in an arbitrary location**, e.g., under a user's home directory in `~/packagename`, or in directories with standard names like `bin`, `lib`, and `man` under a chosen directory. If the first option is chosen, the user may need to modify her search path to include the package's executables and libraries, but this can (more or less) be automated, and is much less risky than setting things up as root.

Testing the ability to install software has traditionally been regarded as difficult, since it necessarily alters the machine on which the test is conducted. Lightweight virtualization containers like Docker make this much easier as well, or simply **ask another person to try and build your software**.

7 Eliminate hard-coded paths.

It's easy to write software that reads input from a file called `mydata.csv`, but also very limiting. If a colleague asks you to process her data, you must either overwrite your data file (which is risky) or edit your code to read `otherdata.csv` (which is also risky, because there's every likelihood you'll forget to change the filename back, or will change three uses of the filename but not a fourth).

Hard-coding file paths in a program also makes the software harder to run in other environments. If your package is installed on a cluster, for example, the user's data will almost certainly *not* be in the same directory as the software, and the folder `C:\users\yourname\` will probably not even exist.

For these reasons, users should be able to **set the names and locations of input and output files as command-line parameters**. This rule applies to reference data sets as well as the user's own data: if a user wants to try a new gene identification

algorithm using a different set of genes as a training set, she should not have to edit the software to do so.

A corollary to this rule is **do not require users to navigate to a particular directory to do their work**. "Where I have to be" is just another hard-coded path.

In order to save typing, it is often convenient to allow users to specify an input or output *directory*, and then require that there be files with particular names in that directory. This practice, which is sometimes called "convention over configuration", is used by many software frameworks, such as WordPress and Ruby on Rails, and often strikes a good balance between adaptability and consistency.

8 Allow configuration of all useful parameters from the command line.

Useful parameters are those values that a user will need to modify to suit their computer, dataset or application. The full list of useful parameters is software-specific and so cannot be detailed here, but a short list includes providing input and reference files and directories, output files and directories, filtering, tuning, random number generation seeds, and any alternatives that you've built-in, e.g., compress results, use a different calculation, verbose output, etc.

Providing parameters on the command line increases the flexibility and usability of the program. You may have determined early in development that 0.58 is an optimal seed for your original dataset, but that doesn't mean that is the best seed for every case. Being able to change parameters on the fly to determine if and how they change the results is important as your software gains more users, facilitating exploratory analysis and parameter sweeping. **Choose reasonable defaults where they exist**, and **set no defaults at all when there aren't any reasonable ones**.

You can set reasonable default values as long as any command line arguments override those values. Configuration files should be used in preference to hard-coding the defaults directly. Configuration files can be in a standard location, e.g. `.packagerc` in the user's home directory or provided on the command line as an additional argument. Only values that are unlikely to change between runs belong in the config file, such as dependencies, servers, version numbers, network drives, and any other defaults for your lab or institution. Use a default configuration file and specify all other parameters on the command line.

When the software starts, it should **echo all parameters and software versions to standard out or a log file alongside the results**. This feature supports greater reproducibility because any result can be replicated with only the previous output files as reference. Finally, if the software is configurable in any way, it should **check that all input values are in a reasonable range at startup**. Few things are as annoying as having a program announce after running for two hours that it isn't going to save its results because the requested directory doesn't exist.

9 Include a small test set that can be run to ensure the software is actually working.

Every package should come with a small test script for users to run after installation. Its purpose is not only to check that the software is working correctly (although that is extremely helpful), but rather to ensure that it will work at all. This test script can also serve as a working example of how to run the software.

In order to be useful, **this test script must be easy to find and run**. Many build systems will also run unit tests if provided them at compile time. If the build system is not amenable to testing, a single file in the project's root directory named `runtests.sh` or something equally obvious is a much better solution than documenting test cases and requiring people to copy and paste them into the shell.

Equally, **the test script's output must be easy to interpret**. Screens full of correlation coefficients do not qualify: instead, the script's output should be simple to understand for non-experts, such as one line per test, with the test's name and its pass/fail status, followed by a single summary line saying how many tests were run and how many passed or failed. If many or all tests fail because of missing dependencies, that fact should be displayed once, clearly, rather than once per test, so that users have a clear idea of what they need to fix and how much work it's likely to take.

Studies have found that the ease with which people can start making contributions is a strong predictor of whether they will or not [18]. By making it simpler for outsiders to contribute, a test suite of any kind also makes it more likely that they will, and software with collaborators stands a better chance of surviving in the busy field of scientific software.

10 Produce identical results when given identical inputs.

Given a set of parameters and a dataset, **a particular version of a program should produce the same results every time it is run** to aid testing, debugging, and reproducibility. Even minor changes to code can cause minor changes in output because of floating-point issues, which means that getting exactly the same output for the same input and parameters probably won't work during development, but it should still be a goal for people who have deployed a specific version.

Many applications rely on randomized algorithms to improve performance or runtimes. As a consequence, results can change between runs, even when provided with the same data and parameters. By its nature, this randomness renders strict reproducibility and therefore debugging more difficult. If even the small test set (#9) produces different results, new users may not be able to tell whether the software is working properly. When comparing results between versions or after changing parameters, even small differences can confuse or muddy the comparison. And especially when producing results for publications, grants or diagnoses, any analysis should be absolutely reproducible.

Given the size of biological data, it is unreasonable to suggest that random algorithms be removed. However, most programs use a pseudo-random number generator, which uses a starting seed and an equation to approximate random numbers. Setting the seed to a consistent value can remove randomness between runs. **Allow the user to optionally provide the random seed as an input parameter**, thus rendering the program deterministic for those cases where it matters. If the seed is set internally (e.g., using clock time), echo it to the output for re-use later. If setting the seed is not possible, **make sure the acceptable tolerance is known and detailed in documentation and in the tests**.

What We Left Out

Any short list of rules for developing robust software will necessarily be incomplete. A few of our more significant omissions include:

Services The rules we present above are necessary, but not sufficient, for software that needs to interact with services such as database and web servers. Such software is often stopped and started under automatic control, may need to authenticate, etc., all of which are out of scope of this paper.

Documentation We think it is very important for developers to document their work, but our experience is that people are unlikely do it during normal development, and we don't want to recommend practices that people won't actually adopt. However, it is worth noting that software that is widely used and contributed to has and enforces the need for good documentation [14]. Numerous resources exist for writing high quality documentation [19] and so we do not cover it here.

Conclusion

There has been extended discussion over the past few years of the sustainability of research software, but this question is meaningless in isolation: any piece of software can be sustained if its users are willing to put in enough effort. The real equation is the ratio between the skill and effort available, and the ease with which software can be installed, understood, used, maintained, and extended. Following the ten rules we outline here reduce the denominator, and thereby enable researchers to build on each other's work more easily.

References

1. Prins P, de Ligt J, Tarasov A, Jansen RC, Cuppen E, Bourne PE. Toward effective software solutions for big biology. Nature Biotechnology. 2015;33(7):686–687. doi:10.1038/nbt.3240.

2. Baker M. 1,500 scientists lift the lid on reproducibility. Nature. 2016;533(7604):452–454. doi:10.1038/533452a.

3. Prabhu P, Jablin TB, Raman A, Zhang Y, Huang J, Kim H, et al. A Survey of the Practice of Computational Science. In: State of the Practice Reports. SC '11. New York, NY, USA: ACM; 2011. p. 19:1–19:12.

4. Lawlor B, Walsh P. Engineering bioinformatics: building reliability, performance and productivity into bioinformatics software. Bioengineered. 2015;6(4). doi:10.1080/21655979.2015.1050162.

5. Ison J, Rapacki K, Ménager H, Kalaš M, Rydza E, Chmura P, et al. Tools and data services registry: a community effort to document bioinformatics resources. Nucleic Acids Research. 2016;44(D1):D38–D47. doi:10.1093/nar/gkv1116.

6. Brazas MD, Yim D, Yeung W, Ouellette BFF. A decade of web server updates at the bioinformatics links directory: 2003-–2012. Nucleic Acids Research. 2012;doi:10.1093/nar/gks632.

7. Stajich JE, Block D, Boulez K, Brenner SE, Chervitz SA, Dagdigian C, et al. The Bioperl Toolkit: Perl Modules for the Life Sciences. Genome Research. 2002;12(10):1611–1618. doi:10.1101/gr.361602.

8. Seemann T. Ten recommendations for creating usable bioinformatics command line software. GigaScience. 2013;2(1):15. doi:10.1186/2047-217X-2-15.

9. Nekrutenko A, Taylor J. Next-generation sequencing data interpretation: enhancing reproducibility and accessibility. Nature ReviewsGenetics. 2012;13(9):667–72. doi:10.1038/nrg3305.

10. Atwood TK, Bongcam-Rudloff E, Brazas ME, Corpas M, Gaudet P, Lewitter F, et al. GOBLET: The Global Organisation for Bioinformatics Learning, Education and Training. PLoS Computational Biology. 2015;doi:10.1371/journal.pcbi.1004143.

11. Sanders R, Kelly D. Dealing with Risk in Scientific Software Development. Software, IEEE. 2008;25(4):21–28. doi:10.1109/ms.2008.84.

12. Afgan E, Baker D, van den Beek M, Blankenberg D, Bouvier D, Čech M, et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. Nucleic Acids Research. 2016;44(W1):W3–W10. doi:10.1093/nar/gkw343.

13. Howe B. Virtual Appliances, Cloud Computing, and Reproducible Research. Computing in Science Engineering. 2012;14(4):36–41. doi:10.1109/MCSE.2012.62.

14. Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, et al. Bioconductor: open software development for computational biology and bioinformatics. Genome Biology. 2004;5(10):1–16. doi:10.1186/gb-2004-5-10-r80.

15. Johnson M. Building a Better ReadMe. Technical Communication. 1997;44(1):28–36.

16. Free Software Foundation. GNU Coding Standards; 2016. `https://www.gnu.org/prep/standards/standards.html`.

17. The IEEE, The Open Group. The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008. 12. Utility Conventions; 2016. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html`.

18. Steinmacher I, Silva MAG, Gerosa MA, Redmiles DF. A systematic literature review on the barriers faced by newcomers to open source software projects. Information and Software Technology. 2015;59:67 – 85. doi:http://dx.doi.org/10.1016/j.infsof.2014.11.001.

19. Karimzadeh M, Hoffman MM. Creating great documentation for bioinformatics software; 2016. `http://hdl.handle.net/1807/73111`.