Ten Simple Rules for Making Research Software More Robust

Morgan Taschuk[1,‡*], Greg Wilson[2,‡]

1 Genome Sequence Informatics, Ontario Institute for Cancer Research, Toronto, Ontario, Canada
2 Software Carpentry Foundation

‡ These authors contributed equally to this work.
* morgan.taschuk@oicr.on.ca

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 be run by anyone, anywhere, and thereby delight your users and collaborators.

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 improves reproducibility, but can accelerate research.

Introduction

Scientific software is typically developed and used by a single person, usually a graduate student or postdoc [1]. It may produce the intended results in their hands, but what happens when someone else wants to run it? Everyone with a few years of experience feels a bit nervous when told to use another person's code to analyze their data: it will often be undocumented, work in unexpected ways (if it works at all), 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 potential 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 is essentially a prototype, and therefore is not *robust*. The lack of robustness in published, distributed software leads to duplicated efforts with little practical benefit, which slows the pace of research [3, 4]. Bioinformatics software repositories [5, 6] catalogue dozens to hundreds of tools that perform similar tasks: for example, in 2016 the Bioinformatics Links Directory included 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 learn what they know about software development on the job or otherwise informally [1, 10]. Existing training programs and initiatives rarely have the time to cover software engineering in depth, especially since the field is so broad and developing so rapidly [4, 10]. In addition, making software robust is not directly rewarded in science, and funding is difficult to come by [1]. Some proposed solutions to this problem include restructuring educational programs, 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.

The good news is, you don't need to be a professionally-trained 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] who have adopted strong software engineering approaches, have high standards of reproducibility, use good testing practices, and foster strong user bases through constantly evolving, clearly documented, useful, and useable software. In the bioinformatics community, Bioconductor and Galaxy follow this path [12, 14]. Not all scientific software needs to be robust [15], but if you publish a paper about your software, it should, at minimum, satisfy these rules.

So what *is* "robust" software? We implied above that it is software that works for people other than the original author, on machines other than its creator's. More specifically, we mean that:

- it can be installed on more than one computer with relative ease,
- it works consistently as advertised, and
- it can be integrated with other tools.

Our rules are generic and can be applied to all languages, libraries, packages, documentation styles, and operating systems for both closed-source and open-source software. They are also necessary steps toward making computational research replicable and reproducible: after all, if your tools and libraries cannot be run by others, they cannot be used to verify your results or as a stepping stone for future work [16].

Rule 1: Use version control.

Version control is essential to sustainable software development [17, 18]. In particular, developers will struggle to understand what they have actually built, what it actually does, and what they have actually released without some mechanical way to keep track of changes. They should therefore **put everything into version control as soon as it is created**, including programs, original field observations, and the source files for papers. Files that can be regenerated at need, such as the binaries for compiled programs or intermediate files generated during data analysis, should not be versioned; instead, it is often more sensible to use an archiving system for them, and store the metadata describing their contents in version control instead [19].

If you are new to version control, it is simplest to treat it as "a better Dropbox" (or, if you are of a certain age, a better FTP), and to use it simply to synchronize files between multiple developers and machines [20]. Once you are comfortable working that way, you should **use a feature branch workflow**: designate one parallel copy (or "branch") of the repository as the master, and create a new branch from it each time you want to fix a bug or add a new feature. This allows work on independent changes to proceed in isolation; once the work has been completed and tested, it can be merged into the master branch for release.

Rule 2: Document your code and usage

How to write high quality documentation has been described elsewhere [21] and so here we only cover two minimal types: the README and usage. The README is usually available even before the software is installed, exists to get a new user started, and points them towards more help. Usage is a terse, informative command-line help message that guides the user in the correct use of the software.

Numerous guidelines exist on how to **write a good README file** [22, 23]. At a minimum, your README should:

1. *Explain what the software does.* There's nothing more frustrating than downloading and installing something only to find out that it doesn't do what you thought it did.
2. *List required dependencies.* We address dependencies in more detail in Rule 5.
3. *Provide compilation or installation instructions.*
4. *List all input and output files*, even those considered self-explanatory. Link to specifications for standard formats and list the required fields and acceptable values in other files. If there is no rigorous definition for a format, explain its parts as clearly as possible in plain English.
5. *List a few example commands* to get a user started quickly.
6. *State attributions and licensing.* Attributions are how you credit your contributors; licenses dictate how others may use and need to credit your work.

The program should also **print usage information** when launching from the command line. Usage provides the first line of help for both new and experienced users. Terseness is important: usage that extends for multiple screens is difficult to read or refer to on the fly.

Almost all command-line applications use a combination of POSIX [24] and GNU [23] standards for usage. More standard command-line behaviours are detailed in [8]. Your software's usage should:

1. *Describe the syntax for running the program*, including the name of the program, the relative location of optional and required flags, other arguments, and values for execution.
2. *Give a short description* to remind users of the software's primary function.
3. *List the most commonly used arguments*, a description of each, and the default values.
4. *State where to find more information.*

Usage should be printed to standard output so that it can be combined with other bash utilities like `grep`, and it should finish with an appropiate exit code.

Documentation beyond the README and usage is up to the developer's discretion. 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 However, it is worth noting that software that is widely used and contributed to has and enforces the need for good documentation [14].

Rule 3: Make common operations easy to control.

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, as it facilitates exploratory analysis and parameter sweeping. Programs should therefore **allow the most commonly changed parameters to be configured from the command line**.

Users will want to change some values more often than others. Since parameters are software-specific, the appropriate 'tunable' ones cannot be detailed here, but a short list includes input and reference files and directories, output files and directories, filtering parameters, random number generation seeds, and alternatives such as compressing results, use a variant algorithm, or verbose output.

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.

To make programs even easier to use, **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.

Changeable values should *never* be hard-coded: if users have to edit your software in order to run it, you have done something wrong. Changeable but infrequently-changed values should therefore be stored in configuration files. These 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. Configuration files are often created during installation to set up such things as server names, network drives, and other defaults for your lab or institution.

Rule 4: 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 this number, 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 [25] 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 old released versions continue 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 [20], 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.

Rule 5: 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 another library or 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 (whether software libraries or separate executables) introduces dependencies, which can bring their own special pain. The interface between two software packages can be a source of considerable frustration: all too often, support requests descend into debugging errors produced by the other project due to incompatible libraries, versions, or operating systems [16]. Even introducing libraries in the same programming language can rely on software installed in the environment, and the problem becomes much more difficult when relying on executables, or even on web services.

Despite these problems, software developers in research should re-use existing software provided a few 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. Reuse software that offers some measurable improvement to your project.

Second, if launching an executable, **ensure the appropriate software and version is available**. 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 your package manager. If the executable requires internet access, check for that early in execution.

Third, **ensure that reused software is robust**. Relying on erratic third party libraries or software is a recipe for tears. Prefer software that follows good software development practices, is open for support questions, and is available from a stable location or repository using your package manager.

Exercise caution especially when transitioning across languages or using separate executables, as they tend to be especially sensitive to operating systems, environments, and locales.

Rule 6: Rely on build tools and package managers for installation.

To compile code, deploy applications, and automate other tasks, programmers routinely use build tools like Make, Rake, Maven, Ant or MS Build. These tools can also be used

to manage runtime environments, i.e., to check that the right versions of required packages are installed and install or upgrade them if they are not. As mentioned in Rule 5, a package manager can mitigate some of the difficulties in software reuse.

The same tools can and should be used to manage runtime environments on users' machines as well. Accordingly, developers should **document all dependencies in a machine-readable form**. Package managers like apt and yum are available on most Unix-like 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 the pip package manager, which can check that the required software is available and install it if it is not. Whatever is used, developers should *always* install dependencies using their dependency description, 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.

Rule 7: Do not require root or other special privileges to install or run.

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, as Apache Tomcat does, that software be installed under its own user account—i.e., that `packagename` be made a user, and all of the package's software be installed in that pseudo-user's space—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 before releasing it**.

Rule 8: 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**, since "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 is an example of "convention over configuration", a principle is used by software frameworks such as WordPress and Ruby on Rails that often strikes a good balance between adaptability and consistency.

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

Every package should come with a set of tests 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 also to ensure that it works at all. This test script can also serve as a working example of how to run the software.

In order to be useful, **make the tests easy to find and run**. Many build systems will also run unit tests if provided them at compile time. For users, or if the build system is not amenable to testing, provide a working script in the project's root directory named `runtests.sh` or something equally obvious. This lets new users build their analysis from a working script. For example, with its distribution, HISAT2 includes a full set of very small files, and a 'Getting Started with HISAT2' section in its manual that leads you through the entire data lifecycle [26].

Equally, **make the test script's output 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.

Research has shown that the ease with which people can start making contributions is a strong predictor of whether they will or not [27]. 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.

Rule 10: Produce identical results when given identical inputs.

The usage message tells users what the program could do. It is equally important for the program to tell users what it actually did. Accordingly, when the program starts, it should **echo all parameters and software versions to standard out or a log file alongside the results** to increase the reproducibility of that step.

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 for each run, 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**.

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.

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.

Supporting information

S1 Checklist. Robust software checklist. A checklist summarizing these ten simple rules to apply to your own software.

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. Varoquaux G. Software for reproducible science: let's not have a misunderstanding; 2015. `http://gael-varoquaux.info/programming/software-for-reproducible-science-lets-not-have-a-misunderstanding.html`.

16. Brown CT. Replication, reproduction, and remixing in research software; 2013. `http://ivory.idyll.org/blog/research-software-reuse.html`.

17. Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, et al. Best Practices for Scientific Computing. PLoS Biology. 2014;12(1):e1001745. doi:10.1371/journal.pbio.1001745.

18. Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. Good Enough Practices in Scientific Computing. arxivorg. 2016;abs/1609.00037.

19. Noble WS. A Quick Guide to Organizing Computational Biology Projects. PLoS Computational Biology. 2009;5(7). doi:10.1371/journal.pcbi.1000424.

20. Blischak JD, Davenport ER, Wilson G. A Quick Introduction to Version Control with Git and GitHub. PLOS Computational Biology. 2016;12(1):1–18. doi:10.1371/journal.pcbi.1004668.

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

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

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

24. 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`.

25. Preston-Werner T. Semantic Versioning 2.0.0; 2016. `http://semver.org/spec/v2.0.0.html`.

26. Pertea M, Kim D, Pertea GM, Leek JT, Salzberg SL. Transcript-level expression analysis of RNA-seq experiments with HISAT, StringTie and Ballgown. Nature Protocols. 2016;11(9):1650–1667.

27. 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.