Dear Editor and Reviewers,

Please find attached our revised paper for "Ten Simple Rules for Making Research Software More Robust". We have taken all reviewer comments into account in this revision and have produced a stronger paper. Each of the reviewer's concerns is addressed in-line below. Attached is also a version of the document with changes highlighted in orange.

Thank you and best regards,

Greg Wilson & Morgan Taschuk

> Reviewer 1
>
> This manuscript provides rules aimed at novice programmers to help them write more robust bioinformatics programs. The rules are all sensible, and biologists turned programmers (and some people with formal software engineering training) would do well to read them.
>
> With any set of guidelines, and especially guidelines about programming, there is room for subjectivity and differences of opinion, so here are mine. The authors are free to include or exclude these comments in their revision.
>
> Robustness
>
> The rules are framed in terms of increasing robustness.
>
> The piece starts out defining robustness in terms of portability, specifically outside the programmer's own environment - but then goes on to define it more broadly in terms of multiple criteria such as being kept under version control (which I don't think is a necessary condition for robustness - many robust programs were made before VCSs although I think using VCSs is excellent advice for a modern programmer)
>
> The definition here is somewhat different from Wikipedia's, which defines it as:
>
> "the ability of a computer system to cope with errors during execution[1][2] and cope with erroneous input."
>
> I am not sure of the best name for the property the authors are aiming at, but it seems some combination of robustness, maintainability, quality, predictability - essentially better software.

Thanks for the feedback. We have clarified and condensed our definition of 'robust' software in the introduction, and have made version control a full rule.

> Documentation and maintainability.

> I find the relegation of documentation to an additional note rather than a rule surprising. It's also telling the authors don't call out maintainability as a desirable property (except in passing in the conclusions). If software is not thrown away, it frequently needs to be updated - e.g. to accomodate changes in input file types. This maintenance task often falls on people who are not the original author. If software does not have adequate documentation, it will be harder to maintain (either by the original author or by others).

We've promoted documentation to a full rule by merging the README and Usage rules into one and explicitly mentioning other types of documentation with appropriate references.

> Good software development practice
>
> The rules do not mention anything about good software engineering practice, such as structured programming. While it's good to not be too specific and dictate specific fads or paradigms (and the intended audience is individual programmers not teams), no one would disagree that some kind of good structured programming practices are required to write programs where clarity, maintainability, extensibility and quality are of concern.
>
> In general, the program should not repeat large pieces of code with the same logic - these should be abstracted into subroutines/functions/methods/classes etc. The program should not be a giant monolithic main() method. Behavior of subroutines should not be controlled through globals. etc.
>
> While it's not necessary to replicate Knuth's three volumes, I think good software engineering practice deserves to be a rule, or at least called out.

Thanks for the feedback. We are focusing mostly on software that is preparing to be published or otherwise distributed, and so discussing the details of program flow and execution are out of scope. We do mention other good software engineering practices, such as source control, testing, and versioning as an introductory taste of software engineering.

> Reproducibility
>
> Reproducibility of software and workflows is increasingly in the spotlight. There are passing mentions of reproducibility, and it is implicit in many of the guidelines, but it would be good to call it out explicitly as a goal beyond making the software robust. For example, in the list on p2.
>
> People like Titus Brown have written a lot about this in blogs. It would be useful to reference some of this.

We've added more mention of reproducibility and citations to posts written by C. Titus Brown and by Gaël Varoquaux.

> README
>
> Excellent advice, although there is some overlap with 2.

These two rules have been combined.

> Usage info
>
> Good advice, but I wonder if 2 and 8 could be combined?

We consider usage to be part of the documentation, but a section of this rule referred to checking input parameters and echoing out parameters to the command line. These sections have been moved to the rules on controlling common operations and producing identical results.

> Versioning
>
> All good advice.
>
> As the authors note, GitHub makes this very easy. The advice is a bit succinct:
>
> "as simple as adding an appropriate commit message or tag to version control"
>
> As this article seems to be aimed as novice programmers who may not have been immersed in good, modern, open software development practice, it would be a good idea to include more details or pointers here.
>
> In fact, I would recommend every programmer learns how to use a VCS and a hosted VCS like GitHub/GitLab/Bitbucket, to commit early and commit often, to use issue trackers, etc. Not just a matter of making public releases (which is important) but also as a matter of good practice even for software not intended to be released.

We think it's important too, but it's a tough sell in some crowds. Version control is its own rule now, and we've added a reference to another educational article, "A Quick Introduction to Version Control with Git and GitHub".

> Reuse software
>
> At first I assumed this section was about reusing software libraries, but it transpired to be about reusing executable programs.

> This seems an odd guideline. While it's certainly convenient to launch a program using commands like 'system' in perl, it's mostly better to use a library (or service). In fact the authors agree with this in 5, in which case why are we recommending them do this here?
>
> The advice seems a bit confused in general. It's not really clear if the authors are saying if it's a good idea or bad idea to call external programs.
>
> I think general advice to 'take care when calling external programs' may be more appropriate.

We've re-written this section more mindful of the package manager rule to follow and more explicitly pushing back against using external executables and services. We also include web services now for consideration.

> built utility / package manager
>
> Excellent advice that is all too often ignored.
>
> As noted above the presentation in combo with 4 is confusing. Generally a package manager is used when an external library, module or package is re-used. But as noted there is generally no advice on reusing these artefacts, as 4 is about running programs.

We believe we have addressed the discrepencies between these two rules with our re-write.

> Do not require root privileges
>
> I certainly wouldn't contradict this but I wonder why it's called out. There are many possible inadvisable things we've all seen in years of programming, but maybe this could be combined with advice about running external programs (4) or making the software runnable outside a default user environment (7)

With the increasing torrent of 'big data', more and more calculations are outsourced into high performance compute clusters or cloud environments. When spawning jobs across tens to thousands of machines that may or may not share any storage, and likely only gives their users privileges in their own space, a requirement for needing root to install a package severely limits its reuse and distribution. Other people without administrative access to their machines are also cut off from using such software. Mostly, we've been bitten by this often enough to feel strongly that it should be its own rule.

> Eliminate hard-coded paths
>
> Excellent advice, and worth calling out as it is such a frequent problem

:)

> Configuration from command line
>
> Good advice, but could be combined with 7, as they are both about configuration.
> Include a small test set
>
> Good advice, but there is no advice about including software unit tests. This is good general software engineering advice, and the existence of simple cloud-based CI frameworks like Travis that integrate with GitHub make it very easy to ensure that every change to a program does not break thing.
>
> Produce identical results when given identical inputs
>
> Generally good advice, but perhaps overly specific. There is a lengthy discussion about what to do in the case of algorithms that involve random numbers. What about software that calls external services?
>
> In my opinion this is a judgment call by the software author. Certainly, if there is no deliberate randomization or calling of external services then nondetermistic output would be troubling.

If you call external services, then according to the (new!) guideline in "Reuse software", that software should itself be robust and therefore produce the same results on every run. Some services are updated to change, add or remove data or results and this is unfortunate, but expected. Two subsequent runs of the software, barring any other changes, should still produce identical results.

In addition, software that doesn't produce identical results is much more difficult to test. Depending on the amount of variation, it can be nearly impossible.

> What we left out
>
> As mentioned above, documentation is important, and some of the existing 10 could be merged to make room for this.

We merged all mentions of documentation into the first rule.

> For services, I am not sure why authentication is mentioned, even in passing. It doesn't really pertain to robustness per se.

With the other changes, this section has been removed.

Reviewer 3

Comments to the author:

This paper sets out to describe 10 simple rules which, if applied, can contribute to making software research more robust. It provides a useful set of guidelines that should generally improve the robustness of research software and it should be of interest to most graduate students who are writing software in the process of their research.

I recommend the publication of this paper but would request the addition or additional emphasis on one point which is not called out as a rule but which is mentioned in passing in several places within the paper. In particular I am referring to the recommendation to put all software into a source control system. As this is an opinion paper, I do not think this is a necessary change for publication (in my opinion it's a vital rule, that is just my opinion) but it is referenced in several places and in particular the author's definition of "robust software" includes "...is kept under version control". As this is an issue frequently ignored or addressed "too late" by graduate students I would like to see this get additional emphasis, even if it is not promoted to being a full rule.

We added Rule #1, which is version control.

The remainder of my comments are meant to provide (hopefully) useful feedback to the authors in the line of suggestions that may enhance the manuscript.

In describing successful software research projects the authors state "...who have adopted strong software engineering approaches, have high standards of reproducibility, use good testing practices, and foster inter-institutional collaborations that grow into strong user bases".. All of these points can be clearly tied to their rules with the exception of the final statement about inter-institutional collaborations. While such collaborations are desirable, particularly to establish project continuity and longevity, they are not strictly necessary for robustness and this is not further referenced in the manuscript.

We removed this line from the text.

In several cases, it seems that the paper might violate the first rule of "The 10 simple rules for writing a 10 simple rules paper" (http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003858) with multiple rules combined into a single rule.

We reviewed all the sections you mentioned. In general, we believe that most of our *bolded*

sections within each rule are refinements of the rule and not separate rules. We address each one of your concerns in-line below.

> For example rule 8 combines putting parameters on the command line with echoing all parameters to the log file to support reproducibility and provenance.

We've moved these recommendations around so that they fit better into their major rule than they did before.

> Similarly rule 5 covers both automating the build process and automating the deployment and installation.

We've clarified this section to focus mostly on build automation and dependency acquisition.

> Rule 4 also tends to stuff multiple rules (bolded) into one rule ((limit dependencies, ensure dependencies are available) these 2 are closely related), with " use native functions" which is a very different idea and not what I would expect from a 'reuse' rule.

We removed this last 'native functions' rule and replaced it with a more appropriate discussion of robust dependencies.

> Finally the 'What we left out' section implicitly defines an additional rule (documentation). In this particular case there could have been a single rule (Document your code and usage) that could encompass the points of both Rules 1 and 2 (Readme, usage) as well as the points from 'What we left out'.

We combined all mentions of documentation into a single rule.

> In rule 8, I would tend to suggest generalizing the rule somewhat to suggest externalizing parameters rather than necessarily requiring them to be available as command line options. For example if a configuration file is used to provide all parameter values, this could accomplish the task of separating the parameters out from the code as well as providing a record of the values used for a run as they describe through echoing parameters to a log file.

This section has been re-written to take your concerns into account.

Thank you again for your time and effort.