IDEAS IN ECOLOGY AND EVOLUTION 8: 50–54, 2015
doi:10.4033/iee.2015.8.8.f
© 2015 The Author. © Ideas in Ecology and Evolution 2015
Received 10 April 2015; Accepted 8 June 2015

Future of Publishing

Best publishing practices to improve user confidence in scientific software

Timothée Poisot

Timothée Poisot (tim@poisotlab.io, @tpoi),Département de sciences biologiques, Université de Montréal, Pavillon Marie-Victorin C.P. 6128, succ. Centre-ville Montréal, Québec H3C 3J7 Canada, and *Québec Centre for Biodiversity Sciences, Montréal, CANADA*

Introduction

The practice of science is becoming increasingly reliant on software—despite the lack of formal training (Hastings et al. 2014, Wilson et al. 2014), upwards of 30% of scientists need to *develop* their own. In ecology and evolution, this resulted in several journals (notably *Methods in Ecology & Evolution*, *Ecography*, *BMC Ecology*) creating specific sections for papers describing software packages. This can only be viewed as a good thing, because the call to publish software in an open way has been made several times (Barnes 2010), and is broadly viewed as a way towards greater reproducibility (Ince et al. 2012). In addition, by providing a peer-reviewed, journal-approved venue, this change in editorial practices gives credit to scientists for whom software development is a frequent research output.

Nevertheless, these papers are at best only pointers to the code, which is itself not tracked in a way that benefits software developers. While the long history of scientific publishing resulted in established (although debatable) measures for the impact of a scientist on the basis of its contribution to the literature, these is no similar effort or standard to do the same with software (Howison and Bullard 2015)—as a result, while there exists an incentive to write good papers, there is no clear incentive to *write good software*. Some initiatives, such as impactstory.org's metrics for code reuse, are a good start, but for code to be valued as a first-class research output, wider adoption of existing technologies and standards of quality is needed.

Most concerning is the fact that, despite the recognized need for quality control (Baxter et al. 2006), we do not currently have a set of community-wide best practices for how code should be released. While it is clear that code is on the verge of being perceived, valued, and tracked as a research output just like data and papers, understanding *how* it should be released and vetted, is still the great unknown. In a survey, Joppa et al. (2013) reported "troubling trends" in the use of software: only 30% of ecologists using species distribution modeling software justified their use by comparison with other tools, or previously published methods. My impression as a reviewer, and reader of the literature, is that software use tends to follow trends, whereby if a package is picked up by a few authors, it will rapidly gain traction regardless of its actual *quality*.

One of the recommendations by Joppa et al. (2013) is that code should be reviewed. Yet as almost anyone that wrote or attempted to review scientific software will attest, this implies a tremendous effort. First, not all software is written in the same language. This simple fact makes looking for reviewers with the expertise to handle a paper incredibly difficult. In an already over-burdened peer-review system, restricting the search for referees to only people with the technical know-how to perform code review is sure to lengthen the review process. Second, even for the most trivial pieces of software, experienced code reviewers are not able to follow all possible paths of program flow (Uwano et al. 2006); and, it can easily be argued that very few ecologists are *experienced* code reviewers—nor should they be.

In order to make scientific software better, which is to say, to both increase the confidence of users and minimize the chances of introducing hard to detect bugs—which is a pre-requisite for making sure that writing code is valued in the same way that writing papers is, these two challenges must be addressed. The good thing is that solutions are *already* in place, and all that is needed is to increase their adoption by a broader share of the community.

Proposed best practices

In the following sections, I will outline how a few steps can be used to make software more reliable, easier to reuse, and discoverable. These are not meant to be the end-all solution to software-related issues, but rather to stimulate a discussion between software producers, software users, and journal editors. These steps are in addition to already well-established best practices: the use of free and open-source license (Morin et al. 2012) (so that the raw source code can be used and improved upon by all users), and the use of publicly available repositories (so that the history of changes can easily be consulted).

Test your software

A test suite is a series of situations that test how the code responds to known inputs. For example, if one were to write a function to measure the mean of a series of number, a good test suite would ensure that the mean of 3.0 and 4.0 is 3.5 (subtleties of floating-point arithmetic notwithstanding). An excellent test suite would ensure that the mean of 3.0 and the character "a" is not defined, and send a message to the user explaining what went wrong, and how it can be fixed. Designing good test suites is somewhat of an art form, but Hunt and Thomas (1999) have a good description of how it can be done.

Writing explicit (and well-documented) test suites and releasing them alongside the code has the potential to significantly reduce the reviewing effort. While reading through code requires the reviewer to be familiar (and even proficient) with a language, test packages usually involve a self-explanatory syntax. For example, python's assertEqual(mean(2, 3), 2.5), julia's @test mean(2, 3) 2.5, and R's expect_equal(mean(2, 3), 2.5) are all easy to understand, even if the details of how mean is implemented are not. Reading tests is also orders of magnitude faster than reading the code itself, and if the code is sufficiently covered, this should be enough to evaluate the robustness of the software.

Inform users of the test coverage

Test engines, when running, collect information about which lines were tested, how often, and which lines were not. This is important information, because it (roughly speaking) informs users of what proportion of the features they are about to use are known to perform as they should. Yong Woo (2003) gives good evidence that code coverage analysis, over time, improves software quality.

From a publishing and reviewing point of view, coverage analysis is an incredibly powerful tool. While reviewing the entirety of a source code is difficult and not foolproof, it is much easier to look up what fraction of the code is covered (services like coveralls.io go as far as color coding the page when the code is not properly covered), then to evaluate whether the test suite is exhaustive enough. Based on this information, reviewers can easily make recommendations about where code revision is needed.

Let the cloud work for you

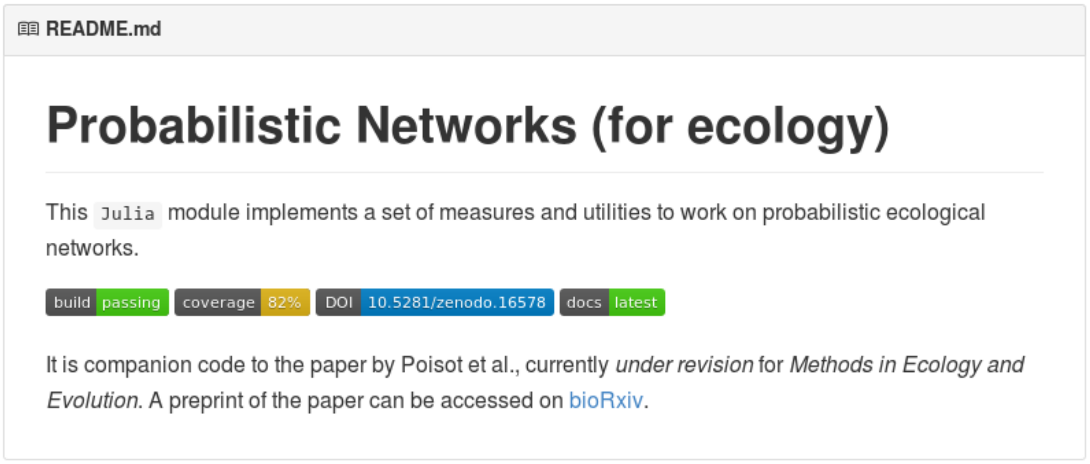
While software developers will run test suite and coverage analyses on their machines, it is important to (i) report the results to the users and (ii) ensure that the software works on "clean" machines. This can be done, in a single step, using cloud-based services known as Continuous Integration (CI) engines. Continuous integration (Duvall et al. 2007) is the practice of committing every change to a source code to a service that will test whether or not the software still works.

This is usually done by coupling a continuous integration (CI) engine (such as, *e.g.*, travis-ci.org) to a version control system such as git or SVN (Ram 2013). When a new change is sent to the central copy (hosted on the git server), the CI engine will run (roughly speaking) two steps. First, it will set up a new environment with the minimal amount of software and libraries needed to run the code. Then, it will run a user-specified series of steps (usually the test suite and coverage analysis), and if none on them fail, will report that the build (the latest version of the code) is "passing". If not, the build will be "failing."

Using CI engines serves two purposes. First, it proves that the software runs on other machines and configurations (most CI engines allow to run, e.g., different versions of R), and (most importantly) that the dependencies are known and can be installed without effort. Second, it serves as a hub for other services. Most cloud-based solutions are well integrated to one another: sending a new version of the code to *GitHub* will trigger a build on *TravisCI*, which will perform the coverage analysis for *Coveralls* to report, and both will then send the results back to *GitHub* for the users to see. Not only does it publicly disclose two obvious measures of code quality, it does so in a way that is effortless for the developer.

Release code in a citable way

As mentioned in the introduction, while software papers include links to the code, the code itself is not tracked, and is difficultly citable. This has the major


README.md

Probabilistic Networks (for ecology)

This `Julia` module implements a set of measures and utilities to work on probabilistic ecological networks.

It is companion code to the paper by Poisot et al., currently *under revision* for *Methods in Ecology and Evolution*. A preprint of the paper can be accessed on bioRxiv.

Figure 1. Example of a GitHub hosted repository, with indication of the build status, current fraction of code covered by tests, and a link to the DOI provided by Zenodo. This informations helps users (and reviewers) evaluate how robustly the code has been tested, and whether it can easily be cited.

disadvantage of not giving credit to software developers for their code (as opposed to for their papers describing the code). In addition, although papers are usually published once, software undergo many iterations ("releases"), and each of them should be cited as a separate entity. Citing code releases could be a leap forward for reproducibility. If a specific version of the code is used and cited, it becomes possible to reproduce the analysis in similar conditions (this assumes that the version number of dependencies is given too). Should a version, or range of versions, or a software be affected by a bug, this also provides a way to rapidly identify which papers can have been affected.

Zenodo (zenodo.org) recently partnered with *GitHub*, to offer researchers the opportunity to get DOI (Digital Object Identifiers) for their code. Every time a new *release* of the code is created, it receives a new DOI, and can be cited as any other scientific document (see Figure 1). This is a necessary step if we want to fully understand the impact of code on the scientific literature. *Zenodo* (hosted by the CERN Data Centre) offers independent and redundant copies of every published version, so one can decide which release to download and cite.

Write documentation, publish use-cases

Looking at recently published software papers in any journal, it is clear that there is no consensus on how these should be written; which is not necessarily a bad thing, but suggests that the community is trying to find its marks in this new practice. To some extent, software papers are a form of documentation. Yet, using them to document *how the program works*, as opposed to *what the program does*, feels like a missed opportunity. Most modern languages offer the possibility to extract formatted "docstrings" to compile a manual from the code itself. The readthedocs.org service does it automatically for python, and there are solution for R (roxygenize), julia (Docile.jl), and others languages. Since the "technical manual" can be extracted directly from the code, software papers are the place to show-case what the software can do. For example, the description of the taxize package for R (Chamberlain and Szöcs 2013), rather than giving a rundown of the different functions, emphasizes how the package can be used for actual research questions, and links to the more extensive documentation.

Conclusion

The use of most of the tools mentioned (a summary of which is given in Table 1) can easily be made public. The shields.io service, for example, provides templates that can be copied/pasted into any web page, giving an up-to-date status information of CI builds, code coverage, link to the documentation, and DOI of the current release. With most journals moving to online-only, it would not be unreasonable to suggest that these, or similar, badges, be presented on the online version of the paper. This would give *readers* the insurance that the software they are going to read about has been tested, and is most likely to be robust than software about which nothing is known. While this can currently be done on the webpage of the project (see Figure 1), moving this information in the paper itself would send a strong signal that using these tools is actively encouraged.

If anything, the importance of code and software in day-to-day scientific practice will only increase, and this

Table 1. Summary of the different tools, along with URL and a short description of their purpose. All of them can be used at no cost for open source projects, and many also provide educational discount for which scientists are eligible.

Service	URL	Purpose
GitHub	github.com	Version control
GitLab	gitlab.com	Version control
BitBucket	bitbucket.org	Version control
TravisCI	travis-ci.org	Continuous integration (Linux)
Appveyor	appveyor.com	Continuous integration (Windows)
Jenkins	jenkins-ci.org	Continuous integration (multi OS)
Coveralls	coveralls.io	Code coverage analysis
Codecov	codecov.io	Code coverage analysis
Zenodo	zenodo.org	DOI provider (*GitHub* integration)
Shields	shields.io	Badges to inform on code status
ImpactStory	impactstory.org	Information on code impact
ReadTheDocs	readthedocs.org	Easy generation of documentation

is a good thing (since it implies that most researchers are adopting state-of-the-art methodology). It is only important to remember that software is written by people, and people make mistakes. Taking simple precautions to make sure that the software works will undoubtedly accelerate the review process, and increase the overall quality of code. In parallel, adding unique identifiers on code, and focusing in describing what it does rather than how it does it, will make it easier to find, easier to cite, and easier to adopt (Howison and Bullard 2015). In most fields in biology, reporting the controls is as important as reporting the outcome of the experiment—there is no reason for software not to be held to the same standard, since the application of code is nothing if not an experiment.

With any standard comes the question of how, and by whom, it should be enforced. Ideally, any researcher writing code will recognize the benefits of these practices, and use them. In any case, there are two categories of people who can have a direct impact of their adoption. The first is, as often, reviewers and editors. Should information about testing, code coverage, and/or licenses be presented in an easy-to-access way, it would be easy for reviewers to treat this information as another measure of the quality or suitability of a paper, or to recommend adjustments. Editors can implement journal-level policies, which spell out the requirements for a manuscript to be accepted. Finally, readers and software users can "vote with their feet"; if a piece of software does not give enough information to trust it, the rational decision would be *not* to use it, and to reach out to the maintainers and request that the best practices are followed.

None of the above practices are an unreasonable time sink, and they have the ability to make scientific software better and more reliable. Just as we want to have a high degree of confidence in the equipment we use in research, so too should we have high standards for the code we use to produce and analyze results. There are outstanding grass-root initiatives (*Mozilla Science Lab*, http://mozillascience.org; *Software Carpentry*, http://www.software-carpentry.org/) that are aimed at improving the computational literacy, and awareness of best practices, among researchers. Writing code is not a niche occupation for ecologists anymore, and existing training opportunities should be used to reflect this change in practices.

Acknowledgments

This paper was prepared when putting together notes for a workshop on code discoverability, for the Canadian Society of Ecology and Evolution annual meeting 2015, in Saskatoon, and largely inspired by group discussions in the Stouffer lab, University of Canterbury. TP is funded by a starting grant from the Université de Montréal, and a Discovery grant from NSERC. Thanks are due to Ethan White and Jeffrey Hollister for comments on the initial submission of this manuscript, as well as Robert Davey for suggestions.

Referees

Ethan P. White – ethan@weecology.org
University of Florida and Utah State University

Jeffrey W. Hollister – jeff.w.hollister@gmail.com
U.S. Environmental Protection Agency

References

Barnes, N. 2010. Publish your computer code: it is good enough. Nature 467:753. CrossRef

Baxter, S.M., Day, S.W., Fetrow, J.S., and S.J. Reisinger. 2006. Scientific software development is not an oxymoron. PLoS Computational Biology 2:e87. CrossRef

Chamberlain, S.A., and E. Szöcs. 2013. taxize: taxonomic search and retrieval in R. F1000Research. CrossRef

Duvall, P.M., Matyas, S., and A. Glover. 2007. Continuous integration: improving software quality and reducing risk. Pearson Education.

Hastings, J., Haug, K., and C. Steinbeck. 2014. Ten recommendations for software engineering in research. GigaScience 3:31. CrossRef

Howison, J., and J. Bullard. 2015. Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature. Journal of the Association for Information Science and Technology. CrossRef

Hunt A, and D. Thomas. 1999. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional.

Ince, D.C., Hatton, L., and J. Graham-Cumming. 2012. The case for open computer programs. Nature 482:485–488. CrossRef

Joppa, L.N., McInerny, G., Harper, R., Salido, L., Takeda, K., O'Hara, K., et al. 2013. Troubling trends in scientific software use. Science 340:814–815. CrossRef

Morin, A., Urban, J., and P. Sliz. 2012. A quick guide to software licensing for the scientist-programmer. Lewitter F, editor. PLoS Computational Biology 8:e1002598. CrossRef

Ram, K. 2013. Git can facilitate greater reproducibility and increased transparency in science. Source Code for Biology and Medicine 8:7. CrossRef

Uwano, H., Nakamura, M., Monden, A., and K-i Matsumoto. 2006. Analyzing individual performance of source code review using reviewers' eye movement. Proceedings of 2006 Symposium on Eye Tracking Research & Applications. CrossRef

Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., Guy, R.T., et al. Best practices for scientific computing. PLoS Biology 12:e1001745. CrossRef

Yong Woo, K. 2003. Efficient use of code coverage in large-scale software development. Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative research.

Response to Referee

I agree with the points made by White (2015) in his review of this paper. Citation is bound to be an issue because software changes over time—and highly successful software is expected to change more, both because increased use will discover more issues, and because it will have to introduce new features to keep up with user demand. Citing each version separately may have the effect of diluting the impact of the software (although most citation trackers are able to aggregate citations received by different versions of an object, the way *Google Scholar* does for the preprint and published version of a paper). No single individual or group of people are going to solve this issue; of all the points I raise in this paper, this is most likely the one that will require a concerted effort to reach a consensus. Ultimately, the citations practices will be determined by what the journals allow. Software developers should advocate to editors and publishers to make sure that the journals guidelines are explicit about the way to cite software—and should do so by expressing their needs, and the best way they think their work can be credited.

White, E.P. 2015. Some thoughts on minimal best practices for publishing scientific software. Ideas in Ecology and Evolution 8: 55–57. CrossRef