A minimal set of best practices for scientific software

Timothée Poisot

Mar. 2015

Introduction

The practice of science is becoming increasingly reliant on software — despite the lack of formal training (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 package. This can only be viewed as a good thing, since 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 not tracked in a bibliometrically measurable way. Some initiatives, such as `impactstory.org`'s metrics for code re-use, are a good start, but for code to be valued as a first-class research output, wider adoption of existing technologies are 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 perceived as a research output just like data and papers, understanding *how* it should be released, vetted, and tracked 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 they 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 a 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 are written in the same language. This simple fact make 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 make the delays increase. Second, there are evidences that, 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 increase the confidence of users, and make the act of producing it recognized in the same way that writing papers is, these two challenges must be addressed. The good thing is that solution 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 safer, easier to re-use, 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.

Use continuous integration

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 "pushed", the CI engine will run a user-specified series of steps, 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".

This serves as a first obvious step to evaluate the robustness of a code. Requesting that code submitted for publication, as part of a software paper or otherwise, have a "passing" status would make sense. Note also that most CI engines allow to test the software on several platforms, or versions of *e.g.*, `R`, `python`, etc, and therefore can be used to ensure that the code will be available to users with various configurations.

Use automated testing, and report the results

One of the most interesting uses of CI engines is to use them conjointly with a robust *test suite*. 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. An excellent test suite would ensure that the mean of 3.0 and the character `"a"` is not defined, and sends a message to the user explaining what went wrong. 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.

Most CI engines offer the possibility to run the tests, and subsequently report how much of the code has been tested. For example, `coveralls.io` has a good interface to see which files, functions, lines are not covered, and whether the coverage changed over time. Yong Woo (2003) gives good evidence that code coverage analysis 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. Focusing the reviewing effort on the test suite also has the advantage of not requiring the reviewer to be familiar with the language the code is written in, since test packages usually use a simple 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.

Release code in a citeable way

As mentioned in the introduction, while software papers include links to the code, the code itself is not tracked, and is difficultly citeable. This should not be the case. The `zenodo.org` service 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. This is a necessary step if we want to fully understand the impact of code on the scientific literature.

More importantly, this is a major leap forward for reproducibility. If a specific version of the code is used, and can be cited, it becomes possible to reproduce the analysis in the exact same conditions — this is particularly true since *Zenodo* (hosted on the CERN Data Centre) offers independent and redundant copies of every published version.

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 showcase what the software can do. For example, the description of the `taxize` package for `R` (Chamberlain and Szöcs 2013), rather than giving a run down 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 on 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 is a good thing. It is only important to remember that software is written by people, and people make mistake. 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.

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.

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
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
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 web pages

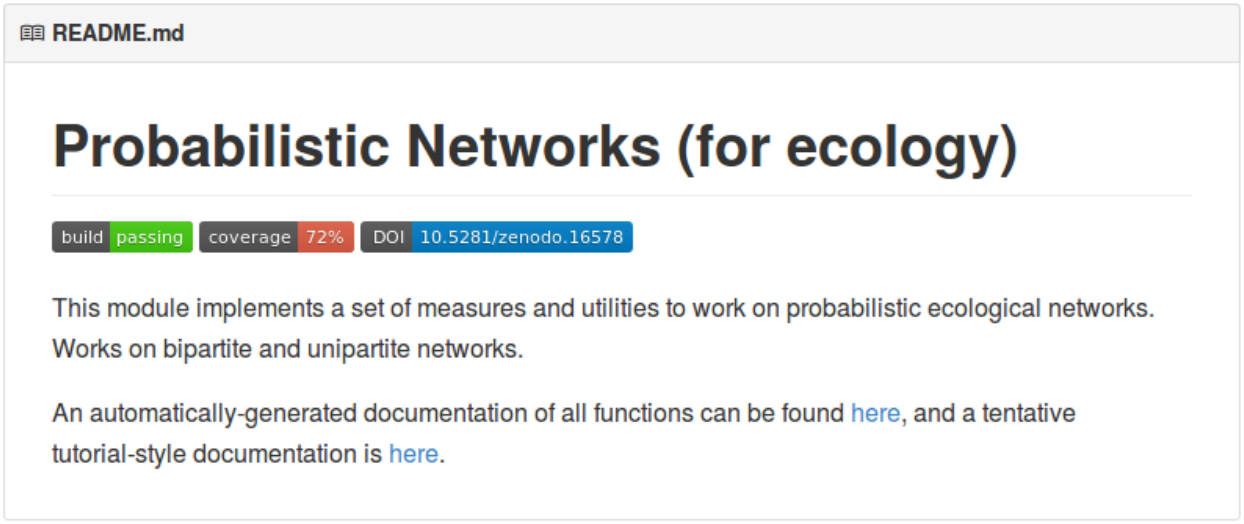

Figure 1: Example of a *GitHub* hosted repository, with indication of the build status, current fraction of code covered by tests, and a link to the DOI provided by *Zenodo*. This informations helps users (and reviewers) evaluate how robustly the code has been tested, and whether it can easily be cited.

References

Barnes N. Publish your computer code: it is good enough. Nature. 2010 Oct;467(7317):753.

Baxter SM, Day SW, Fetrow JS, Reisinger SJ. Scientific Software Development Is Not an Oxymoron. PLoS Comput Biol [Internet]. Public Library of Science (PLoS); 2006;2(9):e87. Available from: http://dx.doi.org/10.1371/journal.pcbi.0020087

Chamberlain SA, Szöcs E. taxize: taxonomic search and retrieval in R. F1000Research. 2013 Oct;

Duvall PM, Matyas S, Glover A. Continuous integration: improving software quality and reducing risk. InPearson Education; 2007.

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

Ince DC, Hatton L, Graham-Cumming J. The case for open computer programs. Nature. 2012 Feb;482(7386):485–8.

Joppa LN, McInerny G, Harper R, Salido L, Takeda K, O'Hara K, et al. Troubling Trends in Scientific Software Use. Science. 2013 May;340(6134):814–5.

Ram K. Git can facilitate greater reproducibility and increased transparency in science. Source Code Biol Med. 2013 Feb;8(1):7.

Uwano H, Nakamura M, Monden A, Matsumoto K-i. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In Proceedings of 2006 symposium on Eye tracking research & applications. 2006.

Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. Best Practices for Scientific Computing. PLoS Biol. 2014 Jan;12(1):e1001745.

Yong Woo K. Efficient Use of Code Coverage in Large-Scale Software Development. In Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research. 2003.