

# Best publishing practices to improve user confidence in scientific software

Timothée Poisot

Mar. 2015

## 1 Introduction

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

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

1 existing technologies and standards of quality is needed.

2 Most concerning is the fact that, despite the recognized need for quality control (Baxter et al.  
3 2006), we do not currently have a set of community-wide best practices for how code should be  
4 released. While it is clear that code is on the verge of being perceived, valued, and tracked as a  
5 research output just like data and papers, understanding *how* it should be released and vetted, is  
6 still the great unknown. In a survey, Joppa et al. (2013) reported “troubling trends” in the use of  
7 software: only 30% of ecologists using species distribution modeling software justified their use by  
8 comparison with other tools, or previously published methods. My impression as a reviewer, and  
9 reader of the literature, is that software use tends to follow a trends, whereby if a package is picked  
10 up by a few authors, it will rapidly gain traction regardless of its actual *quality*.

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

20 In order to make scientific software better, which is to say, to both increase the confidence of users  
21 and minimize the chances of introducing hard to detect bugs — which is a pre-requisite for making  
22 sure that writing code is valued in the same way that writing papers is, these two challenges must be  
23 addressed. The good thing is that solutions are *already* in place, and all that is needed is to increase  
24 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 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. 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.

## **1 Inform users of the test coverage**

2 Test engines, when running, collect informations about which lines were tested, how often, and  
3 which lines were not. This is important information, since it (roughly speaking) informs users of  
4 what proportion of the features they are about to use are known to perform as they should. Yong  
5 Woo (2003) gives good evidence that code coverage analysis, over time, improves software quality.  
6 From a publishing and reviewing point of view, coverage analysis is an incredibly powerful tool.  
7 While reviewing the entirety of a source code is difficult and not foolproof, it is much easier to look  
8 up what fraction of the code is covered (services like `coveralls.io` go as far as color-coding the  
9 page when the code is not properly covered), then to evaluate whether the test suite is exhaustive  
10 enough. Based on this information, reviewers can easily make recommendations about where code  
11 revision is needed.

## **12 Let the cloud work for you**

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

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

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

## 9 **Release code in a citable way**

10 As mentioned in the introduction, while software papers include links to the code, the code itself  
11 is not tracked, and is difficultly citable. This has the major disadvantage of not giving credit to  
12 software developers for their code (as opposed to for their papers describing the code). In addition,  
13 although papers are usually published once, software undergo many iterations (“releases”), and  
14 each of them should be cited as a separate entity. Citing code releases could be a leap forward for  
15 reproducibility. If a specific version of the code is used and cited, it becomes possible to reproduce  
16 the analysis in similar conditions (this assumes that the version number of dependencies is given  
17 too). Should a version, or range of versions, or a software be affected by a bug, this also provides  
18 a way to rapidly identify which papers can have been affected.

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

## 1 Write documentation, publish use-cases

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

## 14 Conclusion

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

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

crease, and this 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 informations about testing, code coverage, 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 gives 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.

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



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

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

 README.md

# Probabilistic Networks (for ecology)

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

build

passing

coverage

82%

DOI

[10.5281/zenodo.16578](https://doi.org/10.5281/zenodo.16578)

docs

latest

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.

# 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. Pearson Education; 2007.
- Hastings J, Haug K, Steinbeck C. Ten recommendations for software engineering in research. *GigaScience* [Internet]. 2014;3(1):31. Available from: <http://www.gigasciencejournal.com/content/3/1/31>
- Howison J, Bullard J. Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature. *J Assn Inf Sci Tec* [Internet]. Wiley-Blackwell; 2015 May;n/an/a. Available from: <http://dx.doi.org/10.1002/asi.23538>
- 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, McInerney 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.
- Morin A, Urban J, Sliz P. *A Quick Guide to Software Licensing for the Scientist-Programmer*. Lewitter F, editor. *PLoS Comput Biol*. 2012 Jul;8(7):e1002598.
- Ram K. Git can facilitate greater reproducibility and increased transparency in science. *Source Code Biol Med*. 2013 Feb;8(1):7.

- 1 Uwano H, Nakamura M, Monden A, Matsumoto K-i. Analyzing Individual Performance of Source  
2 Code Review Using Reviewers' Eye Movement. Proceedings of 2006 symposium on Eye tracking  
3 research & applications. 2006.
- 4 Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. Best Practices for  
5 Scientific Computing. PLoS Biol. 2014 Jan;12(1):e1001745.
- 6 Yong Woo K. Efficient Use of Code Coverage in Large-Scale Software Development. Proceedings  
7 of the 2003 conference of the Centre for Advanced Studies on Collaborative research. 2003.