How to be a good member of a scientific software community [Article v0.1]

Alan Grossfield[1*]

[1]University of Rochester Medical Center, Department of Biochemistry and Biophysics

This LiveCoMS document is maintained online on GitHub at https://github.com/GrossfieldLab/software_community; to provide feedback, suggestions, or help improve it, please visit the GitHub repository and participate via the issue tracker.

This version dated August 11, 2021

Abstract

Software is ubiquitous in modern science — almost any project, in almost any discipline, requires some code to work. However, many (or even most) scientists are not programmers, and must rely on programs written and maintained by others. As a result, a crucial but often neglected part of a scientist's training is learning how to use new tools, and how to exist as part of a community of users. This article will discuss key behaviors that can make the experience quicker, more efficient, and more pleasant for the user and developer alike.

***For correspondence:**
alan_grossfield@urmc.rochester.edu (AG)

1 Introduction

Most practicing scientists (even the ones who write code as part of their research) spend more time as consumers as opposed to producers of software. Moreover, we are constantly learning new skills and solving new problems, which often means learning to use new programs or new aspects of large packages. This, combined with the complex nature of scientific software (and the often low quality of the associated documentation) means we have to ask for help. Sometimes it's because we don't know how to accomplish a specific task. Others, it's because there's a feature we need that isn't implemented. Inevitably, there are bugs.

In all of these cases, it is necessary to interact with the people who develop and support the code. How you go about it has an enormous impact on your likelihood of success and how you are viewed. Asking complete strangers for help is often intimidating, especially the first time you do it. However, the best way to do so is rarely taught explicitly — at best, it's something one picks up by example, from fellow lab members or the PI. The goal of this paper is to reveal the hidden curriculum – what's expected of a software consumer, how to ask for help, how to contribute productively to a software community (regardless of whether you can write code).

This article is written from a the perspective of a developer of academic open-source scientific software, based on my personal experience as a developer and user. I released my first piece of open source code, wham [1], a tool for analyzing umbrella sampling simulation, during the summer of 2000. My group released LOOS[2, 3], a suite of tools for developing new tools to analyze molecular dynamics simulations, in 2008. Over the years, I've interacted with hundreds of people who've tried to use one package or the other. Most of these interactions have been positive, some overwhelmingly so. Others were not. This article is an attempt to distill my experiences into a concise set of recommendations.

Much of what I suggest is universal, but some points — like the reminder that the people providing the support are likely volunteers — are not. Regardless, we view the interaction between developers and users as an informal social contract, where each has obligations and expectations for the other. Obviously, every software community is unique, and many have specific conventions for interaction, but we hope that the recommendations we make are at the very least a good starting point for new users of scientific software.

2 Target Audience

This paper is primarily aimed at junior scientists who are new to performing research and inexperienced at asking for help. However, we hope it will be valuable to anyone who uses software to do their research and struggles to ask for help.

3 Good practices in asking for help

3.1 Try to solve your own problem

There's absolutely nothing wrong with needing help to figure out how to do something with a new piece of software. Perhaps your use case wasn't considered in the manual, or you're seeing behavior you don't expect. Perhaps you can't even get it to install. Asking for help is very reasonable — the developers want people to use their software, and with that comes an implied obligation to support those users. However, keep in mind that the developers of scientific software are, for the most part, volunteering their time to provide that support, and as such it's important to respect their time.

For this reason, asking the developers for help shouldn't be the *first* thing you do. Especially for software with a large community, chances are someone else has run into similar difficulties, and you can probably save yourself and the developers a lot of time if you look for solutions on your own first. Indeed, quickly debugging your use of other people's software is one of the major skills one develops when learning to do computational research.

The first step is to read the error message carefully, if there is one; this obviously doesn't apply in all cases, but it's often applicable and far too often skipped. It's true that many software packages have cryptic unhelpful error messages, but far too often people write for help when the solution is right in front of them. If the error message says "Could not open file foo.dat", it's worth checking whether there is a file called "foo.dat" and if not figuring out what was supposed to create it.

The second step is to check the manual. This might seem trivially obvious, but you'd be astonished how many emails developers get that are directly answered in the manual. If the manual is short, you can read the whole thing. If it's longer, search for relevant-seeming keywords, scan the sections that seem to pertain to your error message. If there's an FAQ (frequently asked questions) list, check that first.

Next, search the internet for the error message; this works very well for software with thousands of users, where it might very well deserve to be the first option. It is less effective with less common packages, but it's an easy thing to check, so you'd be foolish to skip it.

Finally, you can consider diving into the code. This decision depends very much on the complexity of the package, your skill level, and how urgently you need the solution. More often than not, comprehending the whole software package will be too time consuming, but there are tricks and shortcuts one can use to at least figure out where things might be going wrong. Figuring out where the error is occurring can sometimes help you figure out why it happened, so examine the stack trace if there is one. Alternatively, you can search the code base for the error message, to see what drove the error; it's not uncommon for the comments in the code to be clearer and more indicative of what could go wrong than the error message itself. To be clear: as an end user, *you are not obligated to check the code*, but it can sometimes be a good shortcut to help you either solve your problem on your own or give the developers what they need to solve it. Moreover, if you do solve the problem, it's a good idea to share your solution with the developers; most will be grateful to you for your help.

3.2 Ask for help in the right place

Once you've determined you need help, the next obvious step is to ask for it. How should you go about doing so? Email the developer? Post in a forum? Raise an issue on GitHub? Asking the question in the right venue will greatly increase your chances of getting a timely answer. Step 1 is to check the documentation: look at the manual, check the README on the GitHub repository, check the software website, etc — most of the time, at least one of these sources will say how they prefer to receive bug reports, feature requests, etc. Actually, there's a step 0: make sure you're looking at the right piece of software. This might seem obvious, but I've received multiple requests for help with software I had nothing to do with; once it took 8 emails back and forth before I figured this out (but more on that problem later).

Asking in the right place is particularly important if the software is complex or has a large user community — there may be separate forums for bug reports, usage help, and feature request, and asking in the wrong place will make it unlikely the right people will see your question. Moreover, the first impression you'll make is that you're someone who can't be bothered to read the instructions, which will make them less inclined to help you.

3.3 Write a good bug report

The next step in the process is to actually communicate your issue with the developers, generally in the form of a bug report. Doing so effectively requires striving for two somewhat contradictory goals: you must give the developer all of the relevant information about the problem, as concisely as possible.

Taking the first part first: this is hard, because as a user you're usually not an expert, *so you don't know what's relevant*. The only way to know for sure what is and isn't relevant is to develop a mental model for the problem, which requires effort. Still, as pointed out above, your prior efforts to solve the problem yourself (see 3.1) will help you here. Moreover, you may find that the act of writing the note asking for help may in itself let you solve your problem. The process is analogous to rubber duck debugging [4], in that the act of figuring out how to describe a problem clearly can lead you to a solution.

Moreover, many developers will help you with the process. For example, GitHub-based projects have the option to set up issue templates [5]; as the name implies, these templates contain a series of questions one should answer when submitting a new issue. Although they are generally customized for each project, these questions are also a good baseline for the kinds of information one should supply when asking for support elsewhere. For example, for a bug report one should

- Describe the bug, including the expected behavior and how the current behavior differs.
- Give steps necessary to reproduce the bug.
- Describe the platform you're running on (e.g. the operating system for a conventional computer, hardware if it's a tablet, etc.). If the problem is related to building or installing the software, describe the build system (e.g. compiler version) or package manager (e.g. Conda[6]).

For a feature request, you would also describe what you hope to accomplish and how you'd like to see it work, as well as any current workaround you've developed.

3.4 Contribute to the community

3.5 Treat people with respect

This should go without saying, but if you're asking for help it behooves you to do so respectfully. We're not saying that you need to be obsequious. Rather, approach every interaction with respect: respect for people's time, respect for their ideas, respect for their identity. If you're asking for help, do so with the perspective that the developer is volunteering their time to help you, with minimal payoff to them. If you're reporting a bug, you're almost certainly frustrated; try not to take that frustration out on the developers. If you're offering a new feature, keep in mind that it may not align with the developers' vision for where the software is going.

If you're interacting with fellow users in a support forum, don't mock questioners who are less experienced than you — a question may seem foolish to you, but even an uninformed or ill-directed question is an opportunity for teaching. That said, we understand that answering the same questions repeatedly is frustrating and exhausting; one of our goals for this document is that it become a resource to educate new users of expected norms without consuming developers' time and mental bandwidth.

4 Obligations of software developers

This paper has focused on what software consumers should do when asking for help. However, this does not mean that the developers of scientific software are without responsibilities. If we want users to behave in a certain way, it behooves us to tell them that. Users approaching us in good faith deserve to be treated fairly and courteously.

connect to open science: FAIR principles say developers should want code to be reusable. Publishing the code isn't enough, you need to make it usable as well. That means support.

Treating everyone with courtesy and respect, regardless of their identity, is essential if you want the software tool to have a vibrant community surrounding it. Ensuring people aren't excluded is the fair and just way to proceed — making sure that racist, sexist, anti-LGBTQ, etc language and behavior is unwelcome and unacceptable is simply the moral thing to do (it's also pragmatic — can you really afford to lose talented developers and users because they don't feel welcome?) This doesn't just mean use of epithets in messages (though that's clearly unacceptable). Keep an eye on how things are named, on jokes present in the code or messages, on whose opinions are listened to and who is ignored. Above all, if someone tells you a particular behavior or situation is harmful, *believe them*. Something that feels minor (or is just a joke) to you may hit others very differently, and part of respect is understanding and accounting for that.

5 Checklists

6 Author Contributions

The initial version of this paper was written by Alan Grossfield.

For a more detailed description of author contributions, see the GitHub issue tracking and changelog at https://github.com/GrossfieldLab/software_community.

7 Other Contributions

For a more detailed description of contributions from the community and others, see the GitHub issue tracking and changelog at https://github.com/GrossfieldLab/software_community.

GOOD COMMUNITY MEMBER

- ☐ Tries to solve problem themselves first
- ☐ Asks for help in the right place
- ☐ Writes informative bug reports
- ☐ Cites and acknowledges software appropriately
- ☐ Contributes to the community
- ☐ Treats fellow members and developers with courtesy and respect

POOR COMMUNITY MEMBER

- ☐ Doesn't read the manual or search the internet before asking for help
- ☐ Doesn't use the correct venue to ask for help
- ☐ Writes vague or unhelpful bug reports, or doesn't respond to questions
- ☐ Is rude or demanding when requesting support
- ☐ Treats fellow community members disrespectfully

A GOOD COMMUNITY

- ☐ Helps users solve their problems
- ☐ Is friendly and supportive when responding to questions
- ☐ Is receptive to suggestions and critiques, regardless of the source
- ☐ Encourages participation from users of all experience levels
- ☐ Encourages respectful treatment of all community members, and calls out disrespectful behavior

8 Potentially Conflicting Interests

Alan Grossfield serves as a consultant to two companies, Moderna, Inc. and Atelerix Life Sciences.

9 Funding Information

This work supported in part by NIH R21GM138970 to AG.

Author Information

ORCID:
Alan Grossfield: 0000-0002-5877-2789

References

[1] **Grossfield A**, An efficient implementation of the Weighted Histogram Analysis Method (WHAM), http://membrane.urmc.rochester.edu/content/wham; 2021. http://membrane.urmc.rochester.edu/content/wham.

[2] **Romo TD**, Grossfield A. LOOS: an extensible platform for the structural analysis of simulations. Conf Proc IEEE Eng Med Biol Soc. 2009; 2009:2332–2335. https://doi.org/10.1109/IEMBS.2009.5335065.

[3] **Romo TD**, Leioatts N, Grossfield A. Lightweight object oriented structure analysis: tools for building tools to analyze molecular dynamics simulations. J Comput Chem. 2014; 35(32):2305–2318. https://doi.org/10.1002/jcc.23753.

[4] **Hunt A**, Thomas D. The pragmatic programmer. Addison-Wesley;.

[5] **GitHub**, GitHub Issue Templates;. https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/configuring-issue-templates-for-your-repository.

[6] **Lumens C**, Gracik M, Kozumplik A, Sivak M, Jones P, Lane B, Cantrell D, Woods W, Vykydal R, Lehman D, Anaconda: the installation program used by Fedora, Red Hat Enterprise Linux and some other distributions.;. http://fedoraproject.org/wiki/Anaconda.