How To Be a Good Member of a Scientific Software Community [Article v1.0]

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 24, 2022

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. 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 often the missing or inaccurate 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 your 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

Received: 18 October 2021
Accepted: 27 January 2022

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

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, I 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 computational research and inexperienced at asking for help. However, we hope it will be valuable to anyone who uses software written by others to do their research.

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 or 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 too 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; asking an FAQ, especially in a public forum, is likely to lead to grumbling.

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. Search for the whole text of the error (excluding any verbiage that looks like it depends on the compiler, etc), with and without the package name. The latter can be important in the case where the problem isn't with the package per se, but with something else in your setup.

One valuable strategy is to construct a minimal set of circumstances that causes the problem. Even if you initially encountered the problem while analyzing a 2 TB data set, that's not going to be very helpful for diagnosing the problem (and you won't easily be able to

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

share the data set with the developers so they can diagnose it). Can you reproduce the problem using only a small portion of the data? Is the problem present with some kinds of data (e.g. some frames of the trajectory) but not others, and if so what is different about those frames? Can you make the problem appear and disappear with small changes in the settings? Testing these things out may lead you to a solution, but even if they don't they'll help you describe the problem more precisely to the developers.

Along the same lines, it is often valuable to construct a fake or minimal data set where the correct answer is known (or easily computed outside the program). This is particularly important when your concern is that the program is producing incorrect answers due to a coding error or assumptions incompatible with the input data. The more you precisely you can narrow down what's going wrong, the easier the problem will be to diagnose.

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, and they have comments or improvements suggested improvements. If the package is hosted on GitHub, consider opening a pull request so your solution can be merged into the main branch.

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. The challenges in writing an effective bug report have been discussed before in the context of free software [4, 5]. Doing so 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 anal-

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

ogous to rubber duck debugging [6], 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 [7]; 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. Include any input data required.
- Give the version of the code you're using. If you've changed anything, say what you've changed. If there are compile-time options, say what you chose.
- 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[8], Homebrew [9]).
- For a command line tool, provide the command line and configuration information in plain text, *NOT* as a screen shot. A screen shot is often harder to read, and at a bare minimum forces the developer to retype your command instead of copying and pasting, which makes it harder for them to spot possible typos, etc.

Following this format (or something similar) makes it much easier for the developer to diagnose your problem, and the easier you make it the more likely they are to solve your problem quickly. While it's not ideal behavior, busy developers (particularly those for whom software support is a side activity) may shy away from starting work on a problem that looks hard or uninviting, while a clear concise description of the problem is more likely to spur immediate action.

Sometimes, the developer will be able to answer your question based on the first email. However, their reply will often contain questions or suggestions for things to test. *It is crucial that you read this message carefully, and answer all of the questions to the best of your ability when you reply.* As a rule, these questions aren't asked idly; the person providing support is trying to help you! Far too often, users don't read the reply email carefully, don't answer the questions, try only the first suggestion, or refuse to indicate what it is they were actually doing. If you do this, the best case is that it will take longer for you to get to a solution. More likely, you will frustrate the person trying to help you, and they may become reluctant to work on your issue. In my experience, few things are more frustrating than trying to help someone who won't help me help them.

Feature requests are similar: you need to describe what you hope to accomplish and how you'd like to see it work, as well as any current workaround you've developed. Keep in mind that many features sound easy, but there can be deep design issues or technical nuances that make their implementation difficult. Moreover, it is possible that the developer won't be enthusiastic about your idea, even if it is doable — for example, it might not coincide with their vision for the software. Or, as discussed above, academic developers are generally not compensated for their work (obtaining NIH funds to support software development is extremely difficult) and simply might not have the time to work on new features. For these reasons, do not be surprised or insulted if the developers suggest you work on a particular solution yourself and submit a pull request. This doesn't mean they dislike your idea, just that they're busy – they'll help guide you as best they can, but they can't do the work themselves.

3.4 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 (including among other things race, ethnicity, country, native language, sex or gender, or career stage). 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

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

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.

3.5 Contribute to the community

Once you've begun using a piece of software, it's worth looking for ways to contribute back. This can take many forms, not all of which involve writing code (though of course contributing bug fixes or new features is extremely valuable). You don't have to be particularly expert – sometimes the best documentation comes from the inexperienced or newly experienced users, who vividly remember not understanding something in the manual and can suggest revisions that would make it clearer. Many packages hosted on GitHub have the documentation in the same repository, so the same pull request mechanism can be used, but even if the docs are separate (or if you find the idea of a pull request intimidating) you could write a new paragraph or two and send them to the developer. If you developed a toy example as you were figuring out how to use the software, consider sharing it. If one of the provided examples had to be tweaked to work, share the tweaks. If the software has an online forum (e.g. mailing list, GitHub discussions or issues, etc), you might be able to contribute by answering questions from less experienced users.

Unlike the other recommendations, this one isn't mandatory — it's a "nice to have" rather than a "must have". However, there are a number of advantages to doing it. First and foremost, you'll be making a package that's valuable to you stronger, which will help you with your own scientific projects, while getting feedback from leading experts in the field. However, participating productively in a software community is also a good way to build your scientific communication skills. Finally, helping others is an excellent way for a relatively junior scientist to build their reputation in the community, which in turn can lead to future collaborations or even job opportunities. At the very least, it's always valuable when a lot of people in your field have positive associations with your name.

4 Obligations of software developers

This paper has focused on what software consumers should do when asking for help. I suppose it could even be read as a list of complaints about bad behavior by users (though that's not how I intend it). However, each behavior described above has a corresponding expectation for the developers. If we expect the users to try to solve their own problem, it behooves us to write good documentation, give examples, and (most importantly) keep them up to date. If we want to be contacted via GitHub only, we need to make that information clear and easy to find. If we are frustrated by bug reports that don't give us the information we need, we should tell people what kinds of information we need (e.g. by providing bug report templates). In order to receive proper credit, we need to document the appropriate way to cite the software. All of this information needs to be clearly written out and easy to find.

Doing these things is valuable from a practical perspective, as discussed elsewhere [10–12]; making it easy for people to interact with us efficiently, even if it costs us some effort up front, will make those interactions more pleasant. Even setting that aside, it's the right way to do science — FAIR principles for software development [13] prioritize not just releasing code, but making it useable. That means documenting it well and supporting people who try to use it.

Users approaching us in good faith deserve to be treated fairly and courteously. Treating everyone with courtesy and respect, regardless of their identity, is essential if you want to build a vibrant effective community. If the community is just you, the sole developer, then the solution is obvious — when you interact with users, do so in ways that make them feel valued and respected.

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

As the user and developer community grows, other people will get involved in supporting users. It's still your obligation as a developer to maintain those standards. Making sure that racist, sexist, anti-LGBTQ, etc language and behavior is unwelcome and unacceptable is simply the moral thing to do. 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.

It's also important to recognize that even while English has become the default language for science, not everyone is a native speaker, which means that some jargon, abbreviations, and idioms may be incomprehensible to a fair portion of your community. Moreover, an awkwardly worded question often reflects lack of familiarity with the language rather than laziness or ignorance. It's important to keep accessibility in mind while creating documentation or interacting with users in a forum.

Along the same lines, a developer who berates a user for asking a stupid question, or is rude and dismissive in a technical discussion about a new feature, is actively damaging the community, *regardless of the technical quality of their contributions.* It's not enough for you as a developer to treat people with respect. To the extent that there's a community around your software, you must attempt to make sure that community is respectful; as developer, your words and deeds carry significant weight. If you call out bad behavior (or even marginal behavior), that will go a long way towards setting a good tone for your community. If you let it slide, chances are others will assume you implicitly endorse it.

If the justice perspective doesn't persuade you, consider the practical consequences. Bad behavior on the part of developers of your project (or your community in general) will drive users and developers away from you. If interacting with your community is unpleasant, folks will either use alternative codes or use your code but not cite you. The next person with an idea for an innovative new feature will add it to a different package, or won't do it at all (hurting the whole community). Ask yourself: can your project really afford to drive away talented hard-working people eager to help? Is it worth it, just so a few people don't have to engage with basic courtesy?

Sometimes, code is published that is not particularly intended for use. For example, it is good practice to release the scripts used to analyze data in connection with the paper publishing the data, but many of these scripts are one-offs, and the authors have little incentive to polish them to the point of general useability. Rather, the scripts are essentially there as documentation, not turn-key programs for others to use. In this case, the authors have less of an obligation to provide support, although at a minimum laying out the software necessary to run it (e.g. versions of python and numpy, or a Makefile and compiler version) should be required. That said, we recognize that some bit rot is expected with academic code; once the student who wrote the code graduates and moves on, it's very possible that there is no one else to maintain it. Solving the long-term support and maintenance problem for academic software is a major problem, well beyond the scope of this manuscript.

5 Summary

Given the importance of software to modern science and the challenges in writing high quality code, virtually all working scientists will need to ask for help from time to time. The keys to doing so effectively are remarkably simple: ask your questions succinctly and only after putting in some effort to solve the problem yourself, be considerate of other people's time and effort, and look for opportunities to contribute back to the community. Developers should make it easy for users to behave this way by providing clear documentation (including the appropriate venues to ask for help), encouraging participation, and insisting on respectful behavior from users and developers alike.

6 Checklists

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

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 bad behavior

7 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.

8 Other Contributions

Dr. Tod D. Romo reviewed the document for style and content. Dr. Justin A. Lemkul made numerous helpful suggestions, particularly regarding language accessibility. Dr. David H. Mathews provided several editorial suggestions. Dr. Jeffrey Lewis had suggestions regarding searching for help.

The cover image was created by artistfymo (https://linktr.ee/ArtistFrmYO).

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.

9 Potentially Conflicting Interests

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

10 Funding Information

This work supported in part by NIH R21GM138970 to Alan Grossfield.

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.

https://doi.org/10.33011/livecoms.3.1.1473
Living J. Comp. Mol. Sci. 2022, 3(1), 1473

[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] **Raymond ES**, How To Ask Questions The Smart Way;. http://catb.org/~esr/faqs/smart-questions.html.

[5] **Tatham S**, How to Report Bugs Effectively;. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html.

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

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

[8] Conda;. https://www.anaconda.com/.

[9] Homebrew;. https://brew.sh.

[10] **Bangerth W**, Heister T. What makes computational open source software libraries successful? . ; 6:015010. https://doi.org/10.1088/1749-4699/6/1/015010.

[11] **Dall'Olio GM**, Marino J, Schubert M, Keys KL, Stefan MI, Gillespie CS, Poulain P, Shameer K, Sugar R, Invergo BM, Jensen LJ, Bertranpetit J, Laayouni H. Ten Simple Rules for Getting Help from Online Scientific Communities. . ; 7:e1002202. https://doi.org/10.1371/journal.pcbi.1002202.

[12] **Elofsson A**, Hess B, Lindahl E, Onufriev A, van der Spoel D, Wallqvist A. Ten simple rules on how to create open access and reproducible molecular simulations of biological systems. . ; 15:e1006649. https://doi.org/10.1371/journal.pcbi.1006649.

[13] **Lamprecht AL**, Garcia L, Kuzak M, Martinez C, Arcila R, Martin Del Pico E, Dominguez Del Angel V, van de Sandt S, Ison J, Martinez PA, McQuilton P, Valencia A, Harrow J, Psomopoulos F, Gelpi JL, Chue Hong N, Goble C, Capella-Gutierrez S. Towards FAIR principles for research software. . ; 3(1):37–59. https://doi.org/10.3233/DS-190026.