Embracing modern software tools and user-friendly practices, when distributing scientific codes

Remi Lehe*[1], Axel Huebl[1], Jean-Luc Vay[1], Alexander Friedman[2], Maxence Thévenet[3], Chad Mitchell[1], David Bruhwiler[4], David Grote[2], Benjamin Cowan[5], Henri Vincenti[6], Adi Hanuka[7], Brigitte Cros[8], Samuel Yoffe[9], René Widera[10], Michael Bussmann[11,10], and Auralee Linscott Edelen[7]

[1]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]Lawrence Livermore National Laboratory, Livermore, CA, USA
[3]DESY, Hamburg, Germany
[4]RadiaSoft LLC, Boulder, Colorado 80301 USA
[5]Tech-X Corporation, Boulder, CO, USA
[6]LIDYL,CEA-Université Paris-Saclay, CEA Saclay,France
[7]SLAC National Laboratory, Menlo Park, CA, USA
[8]CNRS, Université Paris Saclay, Orsay, France
[9]University of Strathclyde, Glasgow, United Kingdom
[10]Helmholtz-Zentrum Dresden - Rossendorf (HZDR), Dresden, Germany
[11]Center for Advanced Systems Understanding (CASUS), Goerlitz, Germany

1 Motivation

The development of particle accelerators relies heavily on scientific software, in particular to simulate the physics at stake in the accelerator. In this context, a number of efficient and powerful simulation codes exist, but their benefit to the community could be enhanced with greater usability and accessibility [1]. Some potential users may indeed hastily dismiss a given scientific code if it involves an intricate installation procedure, lacks an accessible documentation, or exhibits frequent breaking API changes.

As scientific software becomes increasingly important, and as scientific codes have the potential to benefit a whole community, modern software practices have a growing impact. Nowadays, many free and automated tools exist to help developers ensure code robustness and usability with minimal effort. Here we point to some of those tools and practices, that are widely used in the open-source software community. It is certainly not our view that *every* scientific code should adopt those practices. Instead these are only examples, that show how modern tools and practices can improve the quality and productivity of scientific codes.

2 Examples of modern software practices

2.1 Documentation

Due to the complexity of scientific codes, having an accessible documentation is key to the adoption of a code by new users. Although the term *documentation* can cover many aspects [2], in the case of scientific codes it would typically point the user to the literature on the scientific method implemented, and indicate how to install and use the code in practice.

It can generally be helpful to the user to have access to an online documentation. Since many scientific codes are continuously evolving, it is also important that the documentation remains up-to-date (maybe even with a separate documentation for each public release of given software). To this end, the effort of the code developers can be alleviated by tools that automatically generate some of the documentation from the actual scientific source code, e.g. Sphinx [3] or Doxygen [4]. The generated documentation can then be automatically posted online via services like GitHub-Pages [5] or ReadtheDocs [6].

2.2 Accessibility and easy installation

Another common barrier to user adoption is the installation procedure. Complex installation procedures (whereby users need, e.g., to set library paths by hand and/or install dependencies separately) can oftentimes be error-prone and time-consuming.

Fortunately, there are a number of modern, user-level package managers that automatically control the software environment and dependencies. In the scientific community, popular package managers include `spack` [7], `conda` [8] and `pip` [9]. With those tools, users often only need to type a one-line command, in order to download and install a given scientific software and its complex dependency tree. These tools

*rlehe@lbl.gov

can also help to combine different software in new environments [10]. For scientific developers, making a code available on these package managers is also relatively easy, especially if that code already adopts a standardized build system, like `CMake` [11] (for C/C++/Fortran), `setuptools` [12] and `scikit-build` [13] (for Python).

2.3 User support and communication channels

In order for scientific codes to continuously improve and meet the needs of the community, it is important to have a communication channel for users to provide feedback. This feedback may include potential bug reports, questions on installation issues, and inquiries on typical usage. Again, many tools exist, including issue trackers (e.g. GitHub issues [14]), chat rooms (e.g. Gitter [15] or Slack [16]), mailing list, forums, stack-overflow, etc. These different tools have different purposes and target different segments of the user pool. The aim is thus certainly not for a developer to embrace all of them simultaneously, but rather to select one or two channels that are best suited for their particular code. These communication channels should also be clearly identified in the documentation.

It can also be helpful for these communications channels to be openly accessible and easily searchable, so as to avoid that multiple users report the same issues. In this regard, some of the conversations in chat rooms may sometimes have to be transferred to an issue tracker, or summarized in the documentation.

2.4 Version control and software releases

It is now widely accepted, in the scientific community, that any significant software development project should be managed with a version control system (e.g. `git` [17]). However, a less common practice is that of regularly publishing new releases of the code.

Code releases are important because scientific codes are constantly evolving and improving, and, as part of this process, the code interface with the user may change, bugs may be fixed, and new bugs may occasionally be introduced. It is therefore not uncommon for a new version of the code to suddenly break a workflow that a user previously relied on. Well-documented releases (e.g. with a `CHANGELOG` document) help the user understand how the code has evolved with each release, and how to update their workflow to adapt to these changes. Archiving each release (e.g. with tools such as Zenodo [18]) may also allow the user to roll back to a previous version of the code, if a new version is altogether incompatible with a prior workflow. Also here, readily available services can speed up the generation of `CHANGELOG` documents and publication with package managers through so-called automated deployments.

2.5 Automated testing

In order to minimize the risk of new bugs, it is important to verify that the code still works as expected, whenever the source code is modified. In the case of scientific codes, this can be done for instance by comparing the results of the code against known solutions, for a number of analytically-tractable problems. This process has long been done by hand, but experience shows that it is time-consuming and may in practice be often omitted by the developers.

Instead, these tests can be done automatically and systematically, whenever the source code is modified. Automated tests usually also results in faster code development, since the different contributors to the code can be rapidly assured that their changes do not break previous functionalities. A number of free tools allow to easily setup those automated tests, including GitHub actions [19], Azure Pipelines [20] and Travis-CI [21]. Automated test on exotic hardware or HPC platforms is intricate, but addressed for instance by the E4S [22] effort within the Exascale Computing Project.

3 Conclusion

We have listed a number of modern software tools and practices that can help maximize the usability of scientific codes, and enhance the productivity of the research community as a whole. We emphasize that these practices should not be expected to be universally applied to every single scientific code. Indeed, in the early stages of development of a code, or in a broad research phase, some of these practices may be unnecessary and may stifle productivity. However, for mature and robust codes that have the potential for a large user base, these practices should be considered and encouraged. We reiterate that these practices are nowadays much easier to implement than in the past. In addition to the above mentioned tools, there are many educational resources that developers can draw upon [23, 24, 25, 26].

In order to encourage developers to consider these points, we could envision that funding proposals that cover a *significant software development effort* would include a section on e.g. code maintenance and distribution, similarly to the section on data management which is currently often required. This section would indicate whether the funded effort intends to distribute the code, and if so through which practices and tools. This section could also help justify additional funds to support code maintenance and distribution, as part of the proposal.

References

[1] Ryan Schmitz and Tom Eichlersmith. Accessibility in Physics Computing: A Snowmass Letter of Interest for the Computational Frontier. *Snowmass21 LOI*, 2020.

[2] Divio. `https://documentation.divio.com/`.

[3] Sphinx. `https://www.sphinx-doc.org/`.

[4] Doxygen. `https://www.doxygen.nl/`.

[5] Github pages. `https://pages.github.com/`.

[6] Read the docs. `https://readthedocs.org/`.

[7] Spack. `https://spack.io/`.

[8] Conda. `https://docs.conda.io/en/latest/`.

[9] The phython package installer. `https://pip.pypa.io/en/stable/`.

[10] Jean-Luc Vay, Axel Huebl, David Sagan, David Bruhwiler, Ao Liu, Cho-Kuen Ng, Ji Qiang, and Rémi Lehe. A modular community ecosystem for multiphysics particle accelerator modeling and design. *Snowmass21 LOI*, 2020.

[11] CMake. `https://cmake.org/`.

[12] setuptools. `https://pypi.org/project/setuptools/`.

[13] Scikit-build. `https://scikit-build.readthedocs.io`.

[14] Github issues. `https://guides.github.com/features/issues/`.

[15] Gitter. `https://gitter.im/`.

[16] Slack. `https://slack.com/`.

[17] Git. `https://git-scm.com/`.

[18] Zenodo. `https://zenodo.org/`.

[19] Github actions. `https://github.com/features/actions`.

[20] Azure pipelines. `https://azure.microsoft.com/en-us/services/devops/pipelines/`.

[21] Travis CI. `https://travis-ci.org/`.

[22] E4S. `https://e4s-project.github.io/`.

[23] HEP Software Foundation. `https://hepsoftwarefoundation.org/`.

[24] Ideas productivity. `http://ideas-productivity.org/resources/howtos/`.

[25] Software carpentry. `https://software-carpentry.org/`.

[26] Better scientific software. `https://bssw.io/`.