What makes computational open source software libraries successful?

To cite this article: Wolfgang Bangerth and Timo Heister 2013 *Comput. Sci. Discov.* **6** 015010

COMPUTATIONAL SCIENCE&DISCOVERY

What makes computational open source software libraries successful?

Wolfgang Bangerth[1] **and Timo Heister**[2]

[1] Department of Mathematics, Texas A&M University, College Station, TX 77843-3368, USA
[2] Mathematical Sciences, O-110 Martin Hall, Clemson University, Clemson, SC 29634-0975, USA
E-mail: bangerth@math.tamu.edu and heister@clemson.edu

Received 25 April 2013, in final form 31 October 2013
Published 19 November 2013
Computational Science & Discovery **6** (2013) 015010 (18pp)
doi:10.1088/1749-4699/6/1/015010

Abstract. Software is the backbone of scientific computing. Yet, while we regularly publish detailed accounts about the *results* of scientific software, and while there is a general sense of which numerical methods work well, our community is largely unaware of best practices in writing the large-scale, open source scientific software upon which our discipline rests. This is particularly apparent in the commonly held view that writing successful software packages is largely the result of simply 'being a good programmer' when in fact there are many other factors involved, for example the social skill of community building. In this paper, we consider what we have found to be the necessary ingredients for successful scientific software projects and, in particular, for software libraries upon which the vast majority of scientific codes are built today. In particular, we discuss the roles of code, documentation, communities, project management and licenses. We also briefly comment on the impact on academic careers of engaging in software projects.

Contents

1. Introduction

Today, computational science is the third leg upon which scientific discovery as well as engineering progress rests, often considered on equal footing to theory and experimental investigation. The primary tool of computational science is *software*. Yet, current curricula give surprisingly little attention to teaching the software that is available and, in particular, to the process of arriving at high quality software. Likewise, discovering and documenting best practices are not often considered in our community. In this paper, we intend to discuss what we consider the primary factors determining whether open source computational software libraries—the main foundation upon which academic and also a significant part of industrial computational research rests—is successful. Examples of the kind of libraries we have in mind are PETSc [4, 5], Trilinos [27, 28], the various openly available finite element libraries such as the deal.II project we are both associated with [7], or on a more foundational level libraries such as BLAS/LAPACK and open source Message Passing Interface (MPI) implementations. Because our focus is on open source software libraries supporting computational science, we will define a successful library as one that attracts a significant user and developer community outside the immediate institution at which it is developed; it will also be long-lived, i.e. exceed the 3–5 years lifetime of a typical research project for which it may have initially been built.

Our focus here is not on the reasons for developing and releasing software as open source, nor will we discuss how to write the code that goes into such libraries[3]. Instead, we intend to shed light on the ecosystem that surrounds the actual code as experience has shown that it is not only the actual *code* that determines whether any given project is successful, but also to a large degree the community that surrounds a project, its documentation, willingness to answer questions, etc. In other words, the success of an open source project requires substantial effort and skill outside of programming. Our goal is to document some of the factors we perceive as important in evaluating whether a project will be successful, as well as what project creators should keep in mind in their development planning.

Many of the points we will make here have previously been made—and have been made more eloquently—albeit often in different contexts and with different intentions. For example, many of the observations below about how and why open source communities work mirror those made in Eric Raymond's

[3] For this, refer to the book by Oliveira and Stewart [36] and the recent preprint [44], for example. Most modern books on numerical methods have chapters on writing the actual code for the methods they discuss. It is curious to note that neither [36] nor books on numerical methods pay much attention to anything that goes beyond just writing and debugging the actual code.

seminal essay *The Cathedral and the Bazaar* first published in 1997.[4] However, Raymond's essay is essentially descriptive and analytical, not prescriptive as this paper intends to be. On the other hand, it has an excellent analysis of the social environment of open source software that encompasses that given in section 2.3. A later description of open source software projects is provided in the extensive book by Fogel in [20]. While it contains a great deal of very practical advice on running open source projects and is a worthy resource to any open source manager, it is rather abstract in the projects it talks about. In particular, it does not address at all the specifics of computational software. It is also geared at projects that are typically much larger than what we encounter in our community.

The most important difference between these sources and the current paper is that here, our focus is on *computational* software since there are important distinctions between computational software and other classes of software[5]. For one, the target audience of computational software packages is vastly smaller than for many other projects: while projects like *Firefox* or *Eclipse* may have millions of downloads per year, successful computational projects will have a few thousands[6]. There are a few exceptions, like the very popular NumPy project[7], but this can be attributed to their wide applicability in any scientific project. Furthermore, the potential number of developers of computational software packages is also much smaller because they require a more advanced and specialized skillset on top of an interest in software development: the majority of contributors to computational science projects are graduate students or beyond, with deep knowledge of the mathematical methods employed.

However the biggest difference is in the *kind of software* we will focus on and the technical consequences this has. Software can roughly be categorized into the following classes: software libraries; applications driven by input files, defined communication protocols, or similar fixed format data; and interactive applications with some kind of (graphical) user interface. Within the computational sciences, examples of the first kind would be MPI [21], BLAS, LAPACK or PETSc [4, 5], i.e. foundational libraries that provide building blocks from which users create applications. The second category is represented by many of the codes used in the applied sciences such as in computational chemistry or the computational earth sciences. The final group is exemplified by large visualization packages such as VisIt [17] or ParaView [26]. This distinction between classes of software is important because it affects how users actually interact with software.

In this paper, we will primarily focus on open source projects in the first group, software libraries, since they form the basis of much of computational sciences method development in areas where algorithms are still invented, implemented and tested—i.e. in academic research. Because users (as well as other developers or commercial companies interested in using them) interact in fundamentally different ways with software libraries than with applications that are driven through graphical user interfaces, it is clear that the projects behind such software need to take these differences into account to succeed. It is this particular angle by which this paper differs from the entire existing literature on open source projects.

The paper owes many of its insights to the many conversations we had with leaders of open source projects and their experience. However, there is no denying the fact that it is also influenced by the experience we have had with the deal.II project, a large and successful open source library that supports all aspects of solving partial differential equations via the finite element method (see e.g. [6, 7, 9, 10]) and more recently the Aspect code to simulate convection in the Earth mantle [8, 31]. In the case of deal.II, one of us has been a project maintainer with the project since its inception 15 years ago, whereas the other has graduated from being a user to one of the three current project maintainers. We are also the two principal authors of Aspect. Given this background, some of our examples will use the language of finite element methods; however, this is not intended to limit the scope of the software we want to cover here.

[4] Available as a postscript file from Raymond's web page. An amended version of the essay was later re-printed together with others in [38].

[5] For a similar conclusion that scientific software is different from other software projects, see also [29]. This paper, a case study of how scientific software is typically written, also gives a good overview of the informal nature in which software has traditionally been treated in the sciences. A concrete description of two computational astrophysics projects and how they work is also given by Turk [40]. The conclusions of this latter paper very much align with our own observations.

[6] A guide to the size of the computational sciences community is to keep in mind that even the most widely circulated mathematical journals have less than 1000 institutional subscribers worldwide.

[7] Numpy: scientific computing tools for python (www.numpy.org/).

Like most open source projects, computational science software often starts out as unfunded projects that satisfy someone's or some research group's need. Some projects may, at a much later stage, acquire external funding or institutional backing but these are typically the ones that have already proven to be successful (this is also the story of DEAL.II). In other words, what we describe in this paper as important points and best practices for open source software is most relevant to those projects that may not yet be externally funded but hope to get on a course where they may receive such funding. Of course, the points we make apply equally well to projects that are already externally supported.

In the remainder of this paper, we will first discuss what we consider to be the three primary factors for success (section 2), followed by other reasons (section 3). These sections will show that a lot more than just writing code is required to make a project successful. In section 4 we will therefore comment on the question what this means for individual researchers considering a career based on scientific software. We conclude in section 5.

2. The top three reasons for success

There are, no doubt, many reasons that determine whether a particular project is successful. Maybe the single most important one is whether a project is conceived of at the right time—but since this is something that an author can not immediately affect, we will only come back to this in section 3.1. Among the factors that a project *can* affect, the following three stand out: utility and quality, documentation and the community that supports a project. We will discuss these in turn and come back to other determining factors in section 3.

2.1. Utility and quality

For a project to be used it needs to provide a certain amount of utility. Projects that do not provide the functionality they promise frustrate users and will, sooner or later, be replaced by better software. Utility is a subjective term as it is relative to a user's needs. However, it also has objective components. For example, PETSc is a useful library to many people due to its broadly applicable object model as well as its many independent components that can be combined in myriad ways and that can interact with user written components.

Another metric that determines utility is quality, and we will focus on this aspect in the following. Among the many definitions of 'quality', let us for simplicity just discuss 'it works'—i.e. it can be installed and does what it is intended to do—and ignore criteria like performance, scalability, etc. A project that does not 'work' for a user does not provide any utility and it is this particular (negative) point of view that will guide the following discussion.

The first contact a potential new user has with a software is trying to install it after download. Given that most computational software does not have precompiled packages for major operating systems[8], installing oftentimes happens through the common `./configure; make; make install` sequence of commands in the Linux world. Sometimes, to get through this step, a host of other (optional) libraries may need to be configured, installed and supplied in the configure stage. Packages that turn out to be difficult to install already lose a large fraction of their potential user base: if a package does not readily install on my machine, there is oftentimes a similar package from a separate project that one can download. Most of us do this all the time: both of us certainly download and try to install many more packages than we end up using, often over frustration about the fact that a package does not work right out of the box.

For the developers of a project, this is a depressing reality: making software portable is difficult and requires large amounts of time; yet, it never leads to any recognition. For example, in DEAL.II, the previously second largest file—at 7500 lines of code—was `aclocal.m4`, a file that fed the `autoconf`-generated configuration scripts [41] that determine operating system and compiler capabilities as well as set up interfaces with the many external packages DEAL.II supports[9]. Despite this effort, a non-negligible fraction of questions

[8] The reason for this lack of precompiled packages is the dependence on many other non-standard libraries and the many different ways those libraries can typically be configured.

[9] Our current, `CMake`-based [35] rewrite of this functionality is at around 10 500 lines of code.

on the mailing list is about installation problems and we have little doubt that DEAL.II has lost users that simply could not overcome the hurdles of installing, say, DEAL.II with interfaces to PETSc, BLAS and MPI on an older Apple Mac laptop.

A package that does not install easily on any given system must have unique functionality for people to use it. An example is the MUMPS (MUltifrontal Massively Parallel sparse direct Solver) package [2, 3].[10,11] It requires registration before one can download it. It then also requires the installation of the BLACS and ScaLAPACK packages. For all three of these packages, one needs to find, copy and edit Makefile fragments with non-standard names and none use the configuration and build systems that virtually every other package has been using for the past 15 or more years (e.g. `autoconf`, `automake`, `libtool` or `CMake`). MUMPS would no doubt have far fewer users if there was an easier-to-install, open source project with similar functionality.

Once a potential user has gotten past the installation hurdle, a package needs to be reasonably bug free for people to continue using it. Achieving this aim requires experienced programmers (or code reviewers) that anticipate common programming mistakes as well as discipline in writing test cases for an automated test suite (see section 3.4). It is an illusion to believe that large-scale software with hundreds of thousands of lines of code can be bug-free, but there is clearly better and worse software, and software that often fails to do what a user expects will not be widely used; worse, software that has gotten a bad reputation in a community will not easily recover even if the majority of problems are eventually fixed.

We have good reasons to believe that DEAL.II contains, by and large, few bugs. We believe that this is true primarily because code is either written or reviewed by very experienced programmers and, moreover, because of an insistence on using a very large number of assertions that test for consistency of function arguments and internal state[12]. It also has a very large continuously running test suite.

Finally, we know that those part of the library not well designed in the beginning or not well covered by the test suite show up much more often in bug reports. For many of the same reasons we will discuss in the section on documentation below, software that starts out without a focus on fixing bugs as soon as they are identified will never be of high quality. Thus, quality needs to be an important aspect of development from the start. It is our experience that users generally tolerate a small number of bugs as long as they can get timely help with work-arounds on mailing lists.

2.2. Documentation

Successful software—in particular software libraries and input file-driven applications—*must* have extensive documentation, and it must have it right from the start. There is really no way around it: while one can explore programs with graphical user interfaces interactively, no practical way other than the documentation exists for programs that are not interactive. This is particularly true for software libraries that can be assembled into applications in myriad different ways.

Beyond the classical definition of documentation as a manual, tutorials and code comments, a broader and more practical definition would also include emails (in particular mailing lists and their public archives), lectures and their recordings, and even conversations. We will cover these alternative documentation forms and the associated problems at the end of this subsection.

2.2.1. Extent of documentation. Inexperienced authors often believe that they can write the documentation once functionality is completely developed and software is ready for release. But this is a fallacy because one always underestimates the amount of documentation that is necessary. As one data point, consider that of the 478 000 lines of code in DEAL.II's `include/`, `source/` and `examples/` directories, roughly 170 000 are in comments (both documenting algorithms as well as interface documentation that serves as input for the `doxygen` tool[13]), representing several years of work despite the fact that we know of many areas of

[10] Fortunately, there is an easy-to-use interface to MUMPS through PETSc.

[11] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver. http://graal.ens-lyon.fr/MUMPS/.

[12] For example, the roughly 160 000 lines of code in the `source/` directory contain 38 313 lines with a semi-colon and 4574 lines contain an assertion.

[13] Doxygen: generate documentation from source code. http://doxygen.org/.

the library that are inadequately documented. It is not possible to add even a fraction of this amount of documentation at a later time given the other demands on our work time and the need to continue implementing new features, fix bugs, write papers, etc. It also takes far longer to read through code again at a later time and document it adequately. The only way to achieve sufficient documentation is to write it concurrently with code development. In fact, it is common to write (at least part of) the documentation *before* writing the code. A more strict variation of this is the *design by contract* philosophy of software development that advocates defining formal specifications as part of the design[14].

2.2.2. Levels of documentation. Classical documentation must come at different levels. From the bottom up, well thought out projects provide the following information:

- Traditional comments within the code to explain the algorithms used.
- Function-level documentation explaining what the function does (but not how), the meaning of arguments, what the returned value represents, and pre- and postconditions.
- In object-oriented paradigms, class-level documentation that explains what a class as a whole does.
- Modules that give a bird's eye view of a whole group of classes, explaining how they act together or how they differ if they offer similar functionality.
- Complete, worked examples in tutorial form.

The latter two are frequently forgotten but they are especially important for large software libraries since it is typically difficult to establish how different parts of the library work together by just the myopic look afforded by the first three points above.

Installation instructions (e.g. in the form of a traditional README file) will complement the categories above. The extent of this list makes clear again that documentation cannot be written after the fact, simply because there needs to be so much. We will come back to this in section 3.4 when discussing how to manage the complexity of large software systems.

2.2.3. Alternative forms of documentation. Alternative forms of documentation, such as private emails, lectures, conversations and mailing lists[15] (and their public archives), have several disadvantages over the forms discussed earlier. The most obvious problems are the lack of accessibility and applicability. This is also true for mailing lists to some extent, because the answers are typically tailored to the question and it is sometimes hard to find the right information in mailing list archives. In other words, documentation in these forms is produced for a smaller number of people. Because documenting is such a time consuming effort, it is better spent in writing more accessible forms. A second problem is that traditional documentation can be updated continuously or at least with each release. This is much harder on a mailing list as what has been archived as an answer to a question cannot easily be modified again at a later time.

In recent years, new forms of written communication to replace mailing lists are gaining popularity. The tremendous success of StackOverflow, which also follows the question/answer form of emails with the main difference of allowing the community to edit and score answers and questions, can be attributed to the fact that it improves on accessibility (rephrasing questions, marking questions as duplicates, tagging, etc) and allows updating answers over time. We are not aware of computational science projects that currently use community-editable platforms to replace mailing lists, but they will certainly find their place in the future.

One sometimes hears that inadequate documentation can be compensated by well functioning mailing lists or forums where users can ask things that are otherwise unclear. We believe that this is not true because the number of people who can answer questions—and the amount of time they have—does not scale with the number of users once a project becomes successful. If every one of the several hundred DEAL.II users had only one question per month, it would be impossible for any of the developers to do anything but answer questions. In other words, mailing lists do not scale as well as the primary form of documentation.

[14] See, for example, http://en.wikipedia.org/wiki/Design_by_Contract.

[15] We include discussion boards, web forums, etc, in the term mailing list, because they provide the same form of communication.

The only strategy to avoid this scenario is therefore to provide documentation that is written once and then accessible to all. While there will always be questions on mailing lists, a common strategy is in fact not to answer it right away but to realize that behind every question there is likely a missing piece of documentation—then identify the proper location, write the documentation and provide a link to it as an answer to the original question.

A second observation is that many users apparently do not like to ask questions in a public forum, maybe from a false sense of fear of asking 'trivial' questions. While we always encourage students, for example those in the DEAL.II classes and short courses one of us frequently teaches, to sign up for mailing lists, only a small fraction does; we also have far more (by a factor of 50) downloads than registered users on our mailing lists. On the other hand, we very frequently find papers using DEAL.II whose authors are not on the mailing lists and who have never asked us questions. Clearly, for these users, mailing lists as an alternative to sufficient documentation would not have worked and this reality needs to be taken into account when thinking about how adequate documentation needs to be designed.

Nevertheless, mailing lists are an irreplaceable tool together with traditional documentation. Interestingly, one can get an impression about the quality of the documentation (e.g. the manual) by looking at the kind of questions asked on these lists. On the DEAL.II lists, we traditionally have very few questions about specific things that could and are addressed in the documentation. Instead most questions are high-level, conceptual questions (e.g. how one would implement some advanced feature). Unfortunately, as discussed in section 2.1, we also get a significant number of questions about the installation process—a sign that more work should be done in this regard to better streamline and document the installation.

2.2.4. Technical aspects of documentation. Traditionally, software has been documented in one large monolithic document or book. For example, PETSc still offers (excellent) documentation in this format. However, today, the more widely used format appears to use hyper-linked and shorter documentation in many separate pieces, as it is easier and faster to read, write and update. It is also often easier to search.

There appear to have been two primary reasons for this switch: (i) the realization that it is almost impossible to keep documentation synchronized if it resides in a file separate from the actual code; think, for example, of the difficulty of keeping documentation up to date when adding a defaulted argument to an existing function; (ii) the emergence of tools that can extract documentation from the source code and process them in a way that produces a large number of cross-referenced, well formatted and searchable websites. For code written in C/C++, the primary tool for this today is DOXYGEN (see footnote 13). Programs of this kind extract the relevant information from specially formatted comments placed immediately next to the entity being documented; it is thus easy to keep in sync.

While the creation of documentation will always reside primarily with the author of the code, users can play an important role in expanding it by adding examples, use cases, or general context. An interesting approach to this end has been taken by the SciPy/NumPy group: In an effort to increase the quality and quantity of the documentation, the 'SciPy documentation project' (see http://docs.scipy.org/doc/ and the report [18]) focuses on coordinating the creation of documentation as a community project. Its authors have realized a workflow that allows users (and not only developers) to discuss, correct and write documentation from the official website (in the style of a wiki). The changes are reviewed and, after a final check by developers, applied to the repository[16].

2.2.5. Summary. The length of this section and the diversity of the topics touched upon represents how important we believe it is for projects to offer documentation at many different levels and in many different forms. Primarily, this is a consequence of the fact that there is no other way by which a small number of developers can efficiently train a much larger number of users. It is also a reflection of the complexity of dealing with large-scale software (a topic we will come back to in section 3.4). In effect, it is our firm belief that a software can not survive beyond the time span of the project for which it was initially written (and the

[16] In this project, the documentation is also generated from comments in the source code (similar to DOXYGEN). A script automatically generates patches to the source files from the edits on the website. A developer will then ultimately review and apply the patches.

natural turnover of people associated with it) or across large geographic distances if it is not well documented at many levels.

2.3. Building a community

The prototypical open source project starts as the software a single graduate student writes for her thesis, or maybe as that of a small group of people in one lab. It is also almost universally true that projects in scientific computing are led by a very small group of people—three, four, five—that handle most of the interactions with users, write a very significant fraction of the code, maintain websites, develop tests, take care of documentation and many of the other things that keep a project running from an organizational viewpoint. We will call this group of people the *community of maintainers* of the project for lack of a better name. It often contains those who started the project.

Around this small set of maintainers grows, in the best case, a *community of users*. A point that may not be entirely obvious to those starting a project and thinking about its future is that these communities are not static and that they can not survive long term without a third group: a *community of contributors*. Even more importantly: these communities do not just happen—they are, and must be, engineered: they *require* conscious, sustained efforts to create and grow. Ignoring these two points will lead to the eventual death of a project.

Let us comment on the first point—that communities are dynamic—first. As mentioned, many projects start out with graduate students who have few other obligations and a significant amount of time that they can spend on developing software. As a project matures, developers grow to become maintainers of a project, spending more time on the infrastructure of a project and less on the actual development of the code base. This trend continues as maintainers move along in their careers, spending time teaching, supervising other students or on committees. Some maintainers may also move on to jobs or projects that no longer allow them to work on the software. A project can therefore not survive if it is not able to regenerate its ranks of developers from the user community and, eventually, also its maintainers from the developer community.

Thus, maintainers need to take care to grow and nurture both their user and developer communities, and to be willing to share control of the project with new maintainers. New users are often very hesitant to ask questions on mailing lists or directly. To keep a user community vibrant therefore requires an attitude that is inviting: questions should be answered in a timely manner and with an encouraging, not disparaging, attitude. This requires patience and time, or, as stated in [19]: humility, respect and trust. Similar points are made also in [20].

2.3.1. Lowering the bar. Arguably the most difficult and crucial step is the jump from being a user to being a developer. It is critical to 'lower the bar' to entry into a project, to increase the chance of that jump happening. Without an incentive and without encouragement from the community, only few users are interested enough in contributing back to the project. There are many ways to encourage contributions:

- Make it easy to submit patches, fixes, documentation and bug reports (also see the next paragraph about accepting contributions).
- Promote the mailing lists and the options for contributions.
- Encourage contributions whenever possible, be it on your projects' front page or in replies on a mailing list.
- Be friendly, open and social.
- Provide, help with and highlight incentives.

Incentives for users can come in a variety of ways, both intrinsically as well as extrinsically, and they can be stimulated by the developer community. Firstly, users have practical reasons to contribute: reporting bugs increases the chance that they are fixed by someone else; submitting patches takes away the burden of maintaining them by yourself across releases[17]. A user's new feature will also be better integrated and

[17] This is how the second author got into the development of DEAL.II.

improved upon: with more people using a contributed small feature or tool, improvements are made that can help with the original author's project that required this feature in the first place.

Secondly, some incentives can be provided by the community: appreciation of the contributions (on mailing lists, websites, release notes), invitations to workshops or something simple like free T-shirts[18]. The goal here is to make the contributors realize that their work matters and is appreciated.

Thirdly, there are many advantages to being a contributor. One gains respect in the community and gets 'known'. Project maintainers can make sure that the 'street cred' that comes from participation in a project leads to involvement in research projects, invitations to speak, publications, citations, jobs, funding and influence in general[19].

While these are technical suggestions, lowering the bar ultimately requires addressing the psychology of users. Project leaders must make users feel that the mere fact that they are using a software is something of value that they can contribute—their experience when answering questions on the mailing lists, when improving the documentation, or when pointing out typos or broken links[20]. Of course, the important point is not just to tell users that their contributions are valued in abstract cases, but to show them that this is so by helping get their patches accepted quickly.

2.3.2. Accepting contributions. Growing a developer community also involves questions of control over a project and 'how the code should be written'. To start with, we can observe that occasionally, users simply send pieces of code they have developed for an application of their own for inclusion into the project. We frequently encourage this sort of behavior when replying to questions how to do *X* by saying 'There isn't anything in the project to do *X* right now, but it would not be hard to implement. To do it, follow steps A, B and C. We'd be glad to accept a patch implementing anything along these lines!'

The question is what to do if a user does, in fact, send a patch. Most programmers feel a strong sense of ownership in their code, their programming style and their designs. Patches by new contributors almost never satisfy a project's style guide, nor do they have sufficient documentation; it is even rarer that they satisfy a maintainer's sense of 'how it should be done'. It is then easy to tell someone who has just contributed her first patch to modify it to fit the existing style[21]. Our experience is that this is a safe way to ensure that a significant number of patches will never be submitted in final form. Furthermore, rejecting patches by inexperience developers on purely formal grounds is not an encouraging response and not conducive to growing a developer community. While a large project may be able to afford the luxury of turning away contributions and contributors, most computational software projects cannot.

On the other hand, it is clear that the quality of the code base needs to be maintained (see sections 2.1 and 3.4) and most patches by newcomers are simply not ready to be accepted as is. Our solution to this problem is to spend a significant amount of our time rewriting other people's patches, writing accompanying test cases and then patiently mentoring them why we made these changes. Doing this a few times with a contributor, she will typically pick up on the things that are necessary for high quality patches and start doing them with their next patch. As an additional incentive, we have been rather liberal with granting write access to our code repository after someone has sent in only a few patches. Many of the patches so committed require some reworking and thus a close eye on what is happening in our code base. They also occasionally step on code that the original author is particular proud of and we need to resist the urge to undo the damage done to this marvel of programming art. However, we feel that the gratification of being able to commit patches under a contributor's own name is an important incentive to grow the community of developers. All of this of course goes hand-in-hand with Weinberg's observation that software written by developers that are not territorial about their work tends to improve dramatically faster than elsewhere [43].

[18] The SciPy documentation project gave out T-shirts for the top documentation writers.

[19] The last three postdocs in the first author's group all got their jobs this way. Other contributors have had the opportunity to have a paid for intercontinental research visit.

[20] We can back this up with experience: at least twice in the history of DEAL.II, users have sent small patches with typo fixes; having seen their patches accepted quickly and with gratitude, they came back with scripts that spell-checked the *entire* collection of in-code comments and both later contributed significant new functionality.

[21] As an example of such behavior, the GCC. (GCC) project regularly turns away patches because the accompanying entry to the CHANGELOG file is not appropriately formatted. No doubt this has cost the project a significant number of potential developers.

There are other ways by which one can encourage active participation in a project. For example promoting openness by discussing design decisions on public forums rather than in private mails, making the code repository with the current development sources readable for everyone and avoiding 'inner circle' conversations provides a more welcoming atmosphere to those interested in contributing to a project[22].

2.3.3. Summary. A summary of these views is that a project needs a community of developers but that such a community does not happen by itself: it requires active grooming of contributors and a set of strategies likely to encourage users to contribute code, and to encourage patch authors to continue offering patches. Leading an open source software project to success therefore requires much more than just being a good programmer: it also requires considerable social skills in gently encouraging users inexperienced in the open source ways to participate in a project. Ultimately, small niceties and politeness matter a great deal and can help bring people into a project, while a lack thereof almost certainly turns newcomers away. Furthermore, being polite costs very little, unlike 'material' incentives such as T-shirts or workshop invitations.

3. Other reasons determining success

While we feel that the three topics outlined in the previous section are the most important indicators of whether an open source project will be successful or not, there are clearly other questions that enter the equation. Those are addressed in the following.

3.1. Timing

An interesting point made in Malcolm Gladwell's book *Outliers: The Story of Success* [24] is that people are successful if their skills support products in a marketplace that is just maturing and where there is, consequently, still little competition. The same is certainly true for open source software projects as well: Projects that pick up a trend too late will have a difficult time thriving in a market that already supports other, large and mature projects.

This is not to say that it is always the first project that wins the competition: For example, the Parallel Virtual Machine [23] project predated MPI [21] by some two years (1989 versus 1991), but the advantages of the latter quickly allowed it to supplant the former. Yet, despite the immense expansion of parallel computation both in the number of machines available as well as in the number of cores per parallel machine since then, no other parallel programming paradigm has replaced MPI—even though it is universally acknowledged that MPI is a rather crude way of programming these machines and that MPI might not be successful for machines much larger than the ones available today. The reason for MPI's dominance lies in the fact that it appeared on the market at just the right time: it offered a mature, portable and widely available technology when the scientific computing community started to use parallel computing on machines that were suddenly available to almost everyone. Other upstart projects that might have offered better functionality could not compete with this head start in functionality and size of community, and consequently remain confined to niches; an example might be Charm++.[23]

Other examples for this in the field of scientific computing abound:

- The PETSc library [4, 5] for parallel sparse linear algebra (and other things) dates back to the mid-1990s when distributed parallel computing became widely available for the first time. There have been any number of other libraries with similar aims since; however, with the possible exception of Trilinos [27, 28] (which, like PETSc, has a significant institutional backing from the US Department of Energy), no other library has been able to build as large a community as PETSc's.
- The first widely used libraries upon one can build solvers for partial differential equations were PLTMG [11] and DiffPack [16, 32]. However, they were difficult to adapt to the new mantra of adaptive mesh refinement (in the late 1990s, say after the publication of the influential book by Verführt [42])

[22] This is a frequently made point, most notably found in the influential 1997 essay 'The Cathedral and the Bazaar' already mentioned above. Developing and discussing development in the open corresponds to Raymond's preferred *bazaar* model.

[23] The Charm++ parallel programming system manual. http://charm.cs.illinois.edu/manuals/html/charm++/manual.html.

and all the widely used libraries that today provide functionality for modern finite element methods date from around the year 2000: Cactus [25][24], libMesh [30], Getfem++ [39], OOFEM [37], DUNE [13] or the DEAL.II library [7]. While one can find an almost infinite number of other projects in the same direction that have been started since then, the only widely used recent addition to this field is FEniCS/DOLFIN [33, 34], primarily due to its ability to interface with the Python programming language and the resulting ease of use.

All this, of course, does not come as a surprise: why would one use a small start-up project if extensive, mature, well-tested libraries with significant communities already exist, unless the existing projects fail to incorporate important, new algorithms? In other words, a project can only be successful if it serves a market that is not yet well served by existing projects. Just writing yet another C++ finite element library, even if well thought out, will not produce a project that can create a thriving community around its original developers.

At the same time, the important metric is not necessarily pure functionality but how it works from a user's perspective. An example may be the fact that the GNU compiler collection (GCC) lost a significant share of its (potential) developer community to the much smaller LLVM project: while the latter may have far fewer targets or optimizations, its much clearer code structure and more inclusive and welcoming policies have made it more attractive to new developers.

3.2. Getting the word out

A point that may be obvious is that even good projects will not be successful if nobody knows about them. There are of course many ways to avoid this. The ones we believe are important are well-structured, informative websites that give an overview of what the project does and provides, and release announcements on mailing lists widely read by those in the relevant fields. Furthermore, mentioning and referencing a project in talks and publications—by name and with a link to a website—ensures that people see what a project is used for in a concrete setting, alongside with results they are hopefully impressed by.

3.3. Making a project usable

As discussed in sections 2.1 and 2.2, a project must have sufficiently high quality and be well documented. This is especially true for software libraries and applications driven by input files. However, there is more to it: applications built on such libraries need to be developed and they may need to be maintained for a potentially long time as the underlying software basis grows and changes. Thus, there are both short-term (developing software based on a project) and long-term needs (maintaining it in view of changes in the underlying project) that need to be addressed. Note that only the latter point is relevant to projects that provide user interface-driven applications.

3.3.1. Support for developing with a project. Developing software based on large libraries (or for applications with complex input files) can be a difficult task. This is particularly true when using programming languages with relatively poor support for error messages (such as the template heavy use of C++ in many of today's computational libraries). In order to support their users, projects therefore do well to think about ways that make development easier, beyond providing adequate documentation (section 2.2) and managing complexity (section 3.4).

In particular, we have found the following strategies particularly useful:

- *Catalog common use cases.* While libraries may offer a large number of parts that can be put together in various ways, it is not always trivial to figure out how exactly that should happen. An example would be to build a multigrid solver for a coupled diffusion–advection-reaction problem using multiple processors coupled via MPI: the classes dealing with the mesh, the finite element discretization, the parallel matrices and vectors, the linear solvers and the multigrid components may individually be well documented, but how they interconnect and talk to each other may be harder to determine. A solution to this is to catalog

[24] Cactus code. http://cactuscode.org/.

common use cases by providing tutorials that demonstrate the solution of prototypical problems—in the current case maybe the sequential multigrid solution of a scalar diffusion problem, the solution of a coupled problem on a single processor, and the demonstration of a solver working in practice. This provides worked out examples from which users can copy and paste code fragments to assemble their own programs, and that already demonstrate the connections between different classes.

This appears to be a common approach: both DEAL.II and PETSc have extensive and well documented sets of tutorial programs, and so do many other libraries. For input-driven applications, a similar approach is possible. For example, the ASPECT code [8, 31] has 'cookbooks' that consist of model input files for a variety of situations and that are discussed extensively in the manual. Similar cookbooks exist in other well-written codes (see e.g. [1]).

- *Extensive frequently asked questionss.* Certain questions come up frequently and the answers to some of them do not immediately fit into the documentation of a particular piece of a software library or application. An example is how to debug a particular kind of bug—say, a deadlock in a parallel computation. Such questions can be collected on frequently asked questions (FAQ) pages. These are typically much more visible from a project's homepage than the details of a technical documentation and therefore provide an easy-to-access forum for common questions.
- *Help with debugging.* We all spend far more time debugging code than actually writing it. Support for debugging is therefore an important contribution toward making a library usable. In the case of DEAL.II, we have found that having a large number of assertions checking function arguments for validity throughout the code (see the footnote in section 2.1) and printing a backtrace whenever such an assertion is triggered allows us to find the vast majority of bugs in user code with minimal effort. PETSc follows a similar approach.

  For better or worse, many novice users—even those with prior programming experience—are not used to using debuggers. Furthermore, they are typically entirely unprepared to finding the causes of performance bugs or bugs in parallel programs. To help such users, we provide significant help in our FAQs and through other training modules.

3.3.2. Backward compatibility. In developing software, one is frequently torn between maintaining backward compatibility and moving past things one now recognizes as misconceived or in the way of progress in other areas. The price one pays for breaking compatibility is breaking user programs and other libraries (in the case of software libraries), input files (in the case of programs that are driven by input files) or user workflows (in the case of programs that are operated interactively, such as visualization programs). Software libraries are arguably the most difficult to keep backward compatible because users can interact with them through so many more entry points than, for example, a program with a graphical user interface.

There are different approaches to this problem employed in the community. If a project accepts breaking compatibility often, less of the developers' time is spent on maintaining deprecated features or compatibility wrappers. The result is a project with a smaller, cleaner and often better interface. Conversely, if backwards compatibility is important, the price to pay for sticking with existing interfaces for too long are crufty interfaces and a larger effort implementing new functionality on old, unsuitable interfaces. The risk in this trade-off when moving too quickly is users who either get turned away from the project or hesitate to upgrade to the most recent version.

In DEAL.II, we try very hard to maintain backwards compatibility between releases for a long time. We mark features as 'deprecated' in the documentation and the interface (using language features), to give our users a less painful way to upgrade their code. Often, functions get removed only after having been marked deprecated for years (typically in major x.0 releases). There appears to be little disagreement within our user base with this approach. It has also, over the years, not required an incredible amount of work to maintain older interfaces.

Other projects have taken different approaches. On the one hand, TRILINOS [27, 28] has even codified their approach and many other aspects into a software development strategy, the Tribits Lifecycle Model [12], that strongly encourages maintaining backward compatibility. Linux follows a similar model, with its project leader Linus Torvalds declaring 'So any time any program (like the kernel or any other project), breaks the user

experience, to me, that's the absolute worst failure that a software project can make'.[25] On the other extreme, PETSc [4, 5] has traditionally leaned much more in favor of fixing old warts in their interfaces[26].

3.4. Making a project maintainable: managing complexity and correctness

Writing large pieces of software is much more than just writing the code and making sure that it is correct at that moment. It also requires to think about how what one does today affects writing code and ensuring correctness in the future.

This requires managing the complexity of software, and in particular to keep it as low as possible. Complexity can be described in many ways, and examples may be the best way to deal with this to be concrete:

- *Standardize.* Code is easiest to understand for a reader if it uses clear conventions. For example, with few exceptions, functions in DEAL.II that return the number of cells, degrees of freedom, rows in a matrix, etc, all start with the common name prefix `n_`; variables denoting such numbers of somethings have type `unsigned int`; input arguments of functions come before output arguments; input arguments of functions are marked as `const`; class names use `CamelCase` whereas function names use `underscore_notation`. A slightly more obscure example would be to consistently refer to the names of design patterns [22] in the documentation whenever applicable. The point of such conventions is not that they are better than any other convention; it is simply that following *any* convention consistently makes the code easier to understand because using it is a form of implicit documentation.
- *Modularize.* Separating concerns is a widely used technique to manage complexity. Without going into more detail, methods to achieve this are to minimize the public interfaces of classes, to use inheritance to separate interfaces from particular implementations or to use namespaces to make clear which groups of classes are meant to interact with each other.
- *Avoid duplication.* Since performance is so important in scientific software, one is often tempted to implement code with similar but not exactly the same functionality more than once, specialized to their purpose, rather than implementing it once in a more generic but likely slower way. A trivial example would be a function that interpolates boundary values onto a finite element space for just those parts of the boundary with *one* particular boundary indicator versus a separate function that takes a *list* of boundary indicators. The first of these functions could be implemented by calling the second with a one element list; the second could call the first repeatedly; or they could simply be implemented twice for maximal computational efficiency.

  While one may be tempted to go with the third way in a small project, such duplication will quickly turn out to be difficult to manage as a project grows and as the code needs to be adjusted to new functionality. For example, in DEAL.II we had to adjust this kind of function when we introduced meshes that can be stored in parallel and where each processor no longer knows all cells. If a function is duplicated, this implies that one has to find more places where this adjustment has to be made, with the potential for more bugs and the potential for divergence between duplicated functions if the adjustment is forgotten in one of them.

  The price to pay for avoiding duplication is often slightly slower code. However, it is our experience that the long-term cost of duplication in terms of maintainability of a code base is much higher in all but the most computationally expensive functions.
- *Do it right, right away.* One is frequently tempted to only implement a narrow case one is currently interested in. For example, in the finite element method one can use polynomials of different degree. The first implementation of a new feature typically assumes linear polynomials. However, software libraries are used in myriad and often completely unexpected ways and one can be almost certain that someone will want to use the function in question for a more general case at one point, using polynomials of higher

[25] A summary of how Linux handles backward compatibility is provided at http://felipec.wordpress.com/2013/10/07/the-linux-way/.

[26] To the point where the name `KSPCHEBYCHEV` was changed into `KSPCHEBYSHEV` between PETSc 3.2 and 3.3 without providing any kind of backward compatibility. PETSc has also occasionally changed the number, types or order of function arguments between versions.

degree. While there is no sense in trying to be too generic, there is also a cost to having to extend a design at a later time, possibly requiring changes to the interface and, in the worst case, having to offer different interfaces for the simple and the more general cases. It is our experience that it is well worth investing the time to think about what a proper interface would be for the general case before implementing a narrower one. If it is not feasible to implement the general case right away, a common strategy is to design the interface for the general case but in the implementation abort the program if it is called for a case for which the algorithm has not been written yet, providing a note that it has not been implemented yet. This way, the *structure* of what needs to be implemented is already there and designed, even if the *body* of the code is not yet.

All of this results from the realization that in large software projects, no single person can have a complete overview of the entire code base. Code must therefore be structured in a way so that individual developers can work on their area of expertise without having to know about other modules, and for new developers to be able to be productive without first having to get a global overview.

The second issue about maintainability is to manage correctness in the long term. It is of course possible to verify the correctness of a new function by applying it to the case it was written for. But if it interacts with other functions, or extends an earlier one, does previous functionality continue to work as expected? The only way to ensure this consistently is to write test suites that are run periodically and that verify that functionality that existed at the time a test was written still exists unaltered. This turns out to be far easier for software libraries than for input-file driven applications and, in particular, than for interactive applications. Many libraries therefore have extensive test suites covering most of the existing functionality. As an example, for DEAL.II we run some 2700 tests with every change to the code repository and they have uncovered many bugs and changes in functionality we have inadvertently introduced in patches.

Like documentation, test suites covering a large code base cannot be written after the fact—they must be written concurrently with the implementation. In fact, it is often helpful to write the test *before* the implementation of the functionality it verifies—a process often referred to as 'test-driven development' [15] and commonly accepted as good software development practice in many software development communities. In either case, waiting to write tests until a later time is not an option: even if it were to take only 20 min to write a test and verify that its output is correct, then 2700 tests represent an investment of 900 h, i.e. 6 months of work for someone with no other obligations at all. Nobody can invest even a fraction of this much time in addition to other academic duties.

3.5. License

Which license a project chooses is an often discussed topic among *developers*, but it turns out to be one of little actual interest to the vast majority of *users* in academia and other research settings as long as it falls under the 'open source' label[27]. Nonetheless, in the long run choosing the right license is an important decision, primarily for two reasons:

- Changing the license a software is under at a later time is an incredibly difficult undertaking. This is because every contributor up to that time—several dozen, in our case—has provided their contribution under the old license and will need to be contacted and convinced to re-license the code under the new license. In the case of DEAL.II, the process of changing from the Q Public License (QPL, an open source license popular in the late 1990s when the field of widely used licenses was much larger than it is today) to the GNU Lesser General Public License (LGPL) took more than 2 years.
- The available open source licenses in essence differ in how much control the authors retain over their source code. For example, the QPL requires everyone modifying the software to license their changes back to the project. The GNU General Public License (GPL) does not require this but still requires that those building software on top of a GPL-licensed library make their own software available under the GPL as well. The LGPL requires none of this and thereby allows for closed-source, commercial use. There are also more permissive licenses with minimal restrictions, most notably the various BSD licenses. The

[27] As established, for example, by the Open Source Initiative, see http://opensource.org/.

question of which license to choose is therefore also a question of how much control one wants to retain over how one's software is used.

This last point makes it clear that the question of license is much more important for libraries than for applications because they are meant to serve as the basis for further development.

Most software developers are attached to the fruits of their labor, as are we to our project. Thus, our initial instinct was to go with the more restrictive license (the QPL) because it is, psychologically, difficult to give up control. However, retaining rights can be a hollow victory if it does not translate into tangible benefits. For example, in the 15 years of work on DEAL.II we have not gotten a single patch as a result of the QPL license. Consequently, it would not have made a practical difference had we gone with the GPL instead. Worse, we have had maybe a dozen inquiries over the years from companies who would like to use DEAL.II but did not see how to create a business model given the current license. While one could argue that that is their loss, upon closer thought it is ours, too: these companies would have provided a job market for people with experience in our software, and they might have provided back to the project in the form of code, feedback on development direction, resources or co-financing for workshops. Thus, in the end, our insistence on retaining rights guaranteed by the QPL got us the worst of all outcomes: all abstract rights, no concrete benefits.

As a consequence of this realization, we recently switched the license under which DEAL.II is distributed to the more liberal LGPL—an arduous process, as explained above.

4. Is it worth it? Academic careers with open source

Most computational open source software is written either at universities or at large research facilities such as the national labs in the United States. In the latter, forging a career writing software is certainly a possibility. In academia, most software is written by graduate students or postdocs, and we are occasionally asked whether making their projects open source is a worthwhile career consideration.

We would like to answer this question with an enthusiastic *yes* since we, along with many of our colleagues, believe that science is better served with open source software—in much the same way as we insist that a theorem is so much better if the proof is also provided to the public—as both a way to verify the correctness of a statement and, more importantly, as the basis for further experiments that would otherwise need to start from scratch again. At the same time, for junior scientists with ambitions for a career in academia, this is of course only an option if it provides material for a resume that allows finding a faculty position and later getting tenure.

As laid out in the previous sections, it is not sufficient—by a very large margin—to simply put a piece of software onto a website. Such software will of course be available, but it will not be widely used and its author will be able to derive only little credit from it. Rather, a very significant time investment is necessary to make a project successful and, no doubt, papers will remain unwritten over it[28]. Given the factors explained above, there is also rarely a guarantee that a project will make it in the long term whereas the tenure review is guaranteed to come at one point. Trying your hand at your own open source project is therefore a high risk–high reward, or maybe even just a high risk–doubtful reward, proposition.

Whether a career based on software can work out is difficult to predict. Between the two of us, we only have one sample so far, and the following is therefore a personal account of the first author. When I started DEAL.II in 1997, I was blissfully ignorant of the risks. Since then, I have gone from undergraduate to full professor and I have learned a bit about what counts for a resume at a major research university—and writing software counts for little, even in a case like DEAL.II with several hundred users from practically every corner of the world and well-documented impact[29]. Of course, this is not new and a case for a software-based career track at universities has occasionally been made [14].

[28] We estimate that we spend 30 h per week on working on DEAL.II between the two of us. Of this, at most 10 h is spent on developing new functionality. The remainder is on answering questions, setting up and maintaining the various web servers, regression test scripts, and other services as well as re-writing patches sent in by others.

[29] For example, we have tried very hard to find as many publications written with the help of DEAL.II; see the list of around 400 referenced at www.dealii.org/developer/publications/index.html. A world map showing the locations of downloads indicates practically every major research center as well as many locations one could not have predicted (e.g. French Polynesia, Mauritius and Guam).

For most of my career, I have therefore tried to have more than this one leg to stand on. While I have been tremendously lucky that what I do with DEAL.II was certainly one factor in getting a job and getting promoted—and that I enjoy, more than almost any other aspect of my job, writing software and interacting with the community that has sprung up around it—I am under no illusion that this alone may have sufficed to forge a career. Anecdotal evidence from others in the field points in the same direction. As a consequence, I am usually rather muted in my support to students asking whether they should make their project open source. There are certainly safer ways to plan an academic career, even if my personal opinion is that there are few that are more rewarding.

5. Conclusions

Despite the common perception that the success of a software package is determined by (i) whether the application it is written for is interesting and (ii) whether the authors are good programmers, we hope to have shown that there are, in fact, many more factors involved. In particular, they involve understanding how users interact with a piece of software (through installation, learning to use it through documentation and continued development in view of a base that is changing over the years), and that software projects are embedded in a set of communities of people that are dynamic and that can be actively engineered. Creating and maintaining a successful piece of scientific software therefore requires skills that extend far beyond being a good programmer. It is also an enormous amount of work in addition to just writing the code itself.

Given the importance software has in computational science, it is interesting to realize how little weight we give it in designing our curricula as well as in our decisions on hiring, tenure and promotion. We believe that this is largely because there are few concrete metrics to measure software (e.g. the number of papers written about a software is typically relatively small; however, the number of papers written *with the help* of a software is likely a good indicator of its impact; see also [29]). Likewise, until recently, non-mission-specific funding agencies did not typically fund projects with the explicit goal of creating or extending open source software. It will be interesting to see whether academic and funding perceptions on computational software will change in the future.

Acknowledgments

While we have written this paper in the first person, there is no question that a lot of our insight into what makes open source projects successful has come from discussions with others involved in DEAL.II, in particular our co-maintainers—and friends—Guido Kanschat (to date), Ralf Hartmann (for the first half of the project's life), and the many other users and developers we have met over the years. Many thanks! I (WB) would also like to acknowledge the many pleasant and interesting conversations I have had with Matt Knepley, Mike Heroux, Ross Bartlett, Peter Bastian and others that have shaped my ideas on how open source communities work and what makes them successful. We may not always have the same opinion but I have always valued your insights. WB also thanks jetlag for moments of nocturnal clarity that provided the spark for this paper. The authors also acknowledge the supportive and constructive comments of the two anonymous reviewers.

Parts of the work discussed herein have been funded by the National Science Foundation under award OCI-1148116 as part of the Software Infrastructure for Sustained Innovation (SI2) program; by the Computational Infrastructure in Geodynamics initiative (CIG), through the National Science Foundation under award no. EAR-0949446 and The University of California—Davis; through award no. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST); and an Alfred P Sloan Research Fellowship.

References

[1] Aagard B *et al* 2013 PyLith: a finite element code for the solution of dynamic and quasi-static tectonic deformation problems (www.geodynamics.org/cig/software/pylith)

[2] Amestoy P R, Duff I S, Koster J and L'Excellent J-Y 2001 A fully asynchronous multifrontal solver using distributed dynamic scheduling *SIAM J. Matrix Anal. Appl.* **23** 15–41

[3] Amestoy P R, Guermouche A, L'Excellent J-Y and Pralet S 2006 Hybrid scheduling for the parallel solution of linear systems *Parallel Comput.* **32** 136–56
[4] Balay S, Brown J, Buschelman K, Eijkhout V, Gropp W D, Kaushik D, Knepley M G, McInnes L C, Smith B F and Zhang H 2012 PETSc users manual *Technical Report* ANL-95/11, revision 3.3, Argonne National Laboratory
[5] Balay S, Buschelman K, Gropp W D, Kaushik D, Knepley M G, Curfman McInnes L, Smith B F and Zhang H 2013 PETSc Web page (www.mcs.anl.gov/petsc)
[6] Bangerth W, Burstedde C, Heister T and Kronbichler M 2011 Algorithms and data structures for massively parallel generic adaptive finite element codes *ACM Trans. Math. Softw.* **38** 14/1–28
[7] Bangerth W, Hartmann R and Kanschat G 2007 Deal.II—a general purpose object oriented finite element library *ACM Trans. Math. Softw.* **33** 24/1–24/27
[8] Bangerth W *et al* 2013 Aspect: advanced solver for problems in Earth convection (http://aspect.dealii.org/)
[9] Bangerth W, Heister T and Kanschat G 2013 Deal.II: differential equations analysis library *Technical Reference* (www.dealii.org/)
[10] Bangerth W and Kayser-Herold O 2009 Data structures and requirements for *hp* finite element software *ACM Trans. Math. Softw.* **36** 4/1–4/31
[11] Bank R E 2012 *PLTMG: Software Package for Solving Elliptic Partial Differential Equations* Users' Guide 11.0 (Philadelphia: SIAM)
[12] Bartlett R A, Heroux M A and Willenbring J M 2012 TriBITS lifecycle model, version 1.0 *Technical Report* SAND2012-0561, Sandia National Laboratory
[13] Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Kornhuber R, Ohlberger M and Sander O 2008 A generic grid interface for parallel and adaptive scientific computing: Part II. Implementation and tests in DUNE *Computing* **82** 121–38
[14] Baxter R, Chue Hong N, Gorissen D, Hetherington J and Todorov I 2012 The research software engineer *Digital Research Conference* (*Oxford, September 2012*)
[15] Beck K 2003 *Test-Driven Development: By Example* (Reading, MA: Addison-Wesley)
[16] Bruaset A M and Langtangen H P 1997 A comprehensive set of tools for solving partial differential equations; DiffPack *Numerical Methods and Software Tools in Industrial Mathematics* ed M Dæhlen and A Tveito (Boston, MA: Birkhäuser) pp 61–90
[17] Childs H, Brugger E S, Bonnell K S, Meredith J S, Miller M, Whitlock B J and Max N 2005 A contract-based system for large data visualization *Proc. IEEE Vis.* **2005** 190–8
[18] Van der Walt S J 2008 The Scipy documentation project (technical overview) *SciPy Conf.* (*Pasadena, CA, 19–24 August*) p 27
[19] Fitzpatrick B and Collins-Sussman B 2012 *Team Geek: A Software Developer's Guide to Working Well with Others* (Sebastopol, CA: O'Reilly Media)
[20] Fogel K 2009 *Producing Open Source Software* (Sebastopol, CA: CreativeSpace Independent Publishing Platform) (available for free at http://producingoss.com/)
[21] Message Passing and Interface Forum 2009 MPI: a message-passing interface standard (version 2.2) *Technical Report* (www.mpi-forum.org/)
[22] Gamma E, Helm R, Johnson R and Vlissides J 1994 *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley)
[23] Geist A, Begeulin A, Dongarra J, Jiang W, Manchek R and Sunderam V 1994 *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing* (Cambridge, MA: MIT)
[24] Gladwell M 2008 *Outliers: The Story of Success* (New York: Little, Brown & Company)
[25] Goodale T, Allen G, Lanfermann G, Massó J, Radke T, Seidel E and Shalf J 2003 The Cactus framework and toolkit: design and applications *VECPAR 2002: 5th Int. Conf. on Vector and Parallel Processing* (Berlin: Springer)
[26] Henderson A, Ahrens J and Law C 2004 *The ParaView Guide* (Clifton Park, NY: Kitware)
[27] Heroux M A *et al* 2005 An overview of the Trilinos project *ACM Trans. Math. Softw.* **31** 397–423
[28] Heroux M A *et al* 2011 Trilinos web page (http://trilinos.sandia.gov)
[29] Howison J and Herbsleb J 2011 Scientific software production: incentives and collaboration *CSCW'11: Proc. ACM Conf. on Computer Supported Cooperative Work* (*ACM, 2011*) pp 513–22
[30] Kirk B, Peterson J W, Stogner R H and Carey G F 2006 `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations *Eng. Comput.* **22** 237–54
[31] Kronbichler M, Heister T and Bangerth W 2012 High accuracy mantle convection simulation through modern numerical methods *Geophys. J. Int.* **191** 12–29
[32] Langtangen H P 2003 Computational partial differential equations: numerical methods and diffpack *Programming, Texts in Computational Science and Engineering* (Berlin: Springer)

[33] Logg A 2007 Automating the finite element method *Arch. Comput. Meth. Eng.* **14** 93–138
[34] Logg A and Wells G N 2010 DOLFIN: automated finite element computing *ACM Trans. Math. Softw.* **37** 1–28
[35] Martin K and Hoffman B 2010 *Mastering CMake* 5th ed (Clifton Park, NY: Kitware)
[36] Oliveira S and Stewart D 2006 *Writing Scientific Software* (Cambridge: Cambridge University Press)
[37] Patzák B and Bittnar Z 2001 Design of object oriented finite element code *Adv. Eng. Softw.* **32** 759–67
[38] Raymond E S 1999 *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (Sebastopol, CA: O'Reilly Media)
[39] Renard Y and Pommier J 2006 Getfem++ *Technical Report* INSA, Toulouse (www-gmm.insa-toulouse.fr/getfem/)
[40] Turk M 2013 How to scale a code in the human dimension, arXiv:1301.7064v1
[41] Vaughn G V, Ellison B, Tromey T and Taylor I L 2000 *GNU Autoconf, Automake and Libtool* (Indianapolis, IN: Sams Publishing)
[42] Verfürth R 1994 *A posteriori* error estimation and adaptive mesh-refinement techniques *J. Comput. Appl. Math.* **50** 67–83
[43] Weinberg G 1998 *The Psychology of Computer Programming: Silver Anniversary Edition* (New York: Dorset House)
[44] Wilson G *et al* 2012 Best practices for scientific computing, arXiv:1210.0530v3