January 23, 2014

Quo Vadis, Scientific Software?

By Wolfgang Bangerth, Timo Heister

A large majority of the papers published in SIAM journals, and elsewhere in our community, show a computational result of some kind—either as the main point (e.g., in *SIAM Journal on Scientific Computing*), for backing up theoretical claims about numerical methods (e.g., in *SIAM Journal on Numerical Analysis*), or in modeling experimental observations. Indeed, the discipline that brings us these computations—computational science—is accepted today as the third leg of science, next to theory and experimentation. Yet, although we talk a lot about the underlying methods and algorithms, we almost never talk about their incarnation in real life: software.

Alas, this is an important omission, as much remains to be improved in this area.

First, whereas science prides itself on building on the shoulders of giants, this is not the case with software. Having moved past the Bourbaki movement, we readily accept a reference to a previously proven theorem rather than proving it anew.

By contrast, many of our graduate students write the codes that illustrate their algorithms essentially from scratch. These codes themselves are rarely published. This is the equivalent of publishing a theorem and claiming that a proof exists, but not including it in the paper. Grad students in pure math can't get away with that, and ours should not either!

Secondly, as a community we hold the authors of papers and books in high esteem, but rarely give credit to those who make software available or contribute to open-source projects. Few in our community can forge a career on scientific software, even if their results affect and enable the work of hundreds of other scientists. This is remarkable—writing widely usable software is a rare skill (a claim easily verified by looking at the average graduate student's programs), a skill belittled as "just coding" only by those who don't possess it.

In other words, as a community we ought to have a conversation on how to move forward with regard both to the software that underlies our work and to those who write it. This article is an attempt to start that conversation.

Software: Where We Are and Where We Are Going

The software we use today is vastly more complex than its antecedents. Papers discuss algebraic multigrid methods on thousands of processors, discretizations of the magnetohydrodynamics equations on adaptively refined unstructured meshes, and simulations of processes in geology and astronomy that require billions of unknowns. None of this is possible without a significant investment in software.

Even papers limited to one small computational aspect of a larger problem can require thousands of lines of code. Yet most of the time, authors do not make these codes available. We could think of this as merely regrettable, something that prevents others from learning from and building on these codes.

But this habit also causes serious problems for our community: If every graduate student writes the code for a new discretization from scratch, we will be stuck forever solving "toy problems" (like the proverbial "Laplace equation on the unit square"), because that's what's possible in three years of work. Unfortunately, this is no longer sufficient to convince our colleagues in the applied sciences (who moved past this stage a long time ago) that the new algorithm is also applicable to their vastly more complex problems. The risk is that we will isolate applied mathematics and numerical analysis: If we can't convince our friends in the sciences and engineering that what we do is worthwhile, using examples they can relate to, then all we really do is dabble in esoteric corners.

There are solutions to this problem. Over the past 15 years, we have seen the development of large libraries (e.g., PETSc [1,2] and Trilinos [5,6] in linear algebra; deal.II [3,4] and FEniCS [8,9] for finite elements; Clawpack [7] for hyperbolic conservation laws) that already cover many of the standard techniques. These libraries typically come with extensive tutorials that graduate students can take as the

basis for their own work. To take the example of deal.II, our own contribution: It is realistic for a graduate student to develop a new discretization and demonstrate its qualities on a nonlinear problem, using parallel algebraic multigrid solvers on adaptively refined unstructured meshes for complex geometries. The student can do this in three years of research, because all the building blocks are already there, often combined in an existing and available program in which only the discretization has to be changed.

The challenge is to change the habits of our community. We should no longer encourage or even allow our graduate students to write their programs from scratch; rather, they should build on what's already there. And they in turn should make their own codes available so that others can build on them—just as we expect students to use others' theorems and make their own proofs available.

Motivations
Part of the problem is our reward structure: Making code available to others is typically seen as a waste of time, given that it requires at least a modest amount of documentation and code cleanup. We have impact factors and citation counts for papers, but no established way to give credit for software or means for evaluating the relevance of software. Consequently, it is often overlooked in decisions of hiring, tenure, and promotion.

While this may be something of an obstacle for the leaders of widely used open-source software packages, it is a real problem for younger contributors of, say, a few thousand lines of often good code for one of these packages. This code represents a significant investment of their time, and they benefit the community by making their results available to everyone. Yet, with the exception of the principals of a project, few late-comers will generate name recognition or other tangible benefits by contributing their work. It is not that the principals want all the credit for themselves (all the projects mentioned earlier have pages that give credit to contributors, listing, in the case of deal.II, around 70 people who have made substantial contributions). The problem is that we currently have no way for others to claim this credit, beyond providing a link to a web page. Given this lack of widely accepted recognition, it is difficult to motivate the best people to put their time into making software available to everyone; in fact, they may jeopardize their careers by doing so.

Of course it is difficult for outsiders to evaluate and quantify someone's contributions to a project, especially if the project has been the work of dozens of contributors over many years. But surely, as a community we can come up with ways to give credit where credit is due, devising a system that is at least as good as citation counts and impact factors. Some projects—PETSc is a good example—provide bibtex entries for their web pages or user manuals that list all significant contributors over the years.

Solutions?
If we accept the premise that our current system of writing software is at least not optimal, then steps toward improvement must include defining better metrics and ensuring that incentives are aligned. Some often heard suggestions along these lines are the following:

- We need better ways to credit those who write software. If you use a particular open-source project in a publication, you should say so, and reference the most relevant publications describing it.
- Hiring committees and grant proposal panels need to recognize the role of software. Writing successful software is a creative process; it is also—if the authors provide user support—a lot of continuing work. Yet most departments consider it less valuable than proving theorems. This rules out academic careers for many of the best authors of scientific software. Even worse, it deprives our students of the opportunity to learn to write software, something almost all of them will need to do in their professional lives.
- We need to find better ways to credit people who contribute significant amounts of code to existing projects. Few options are now available, and creative solutions will be necessary.
- Publication of code used to obtain the results presented in a paper needs to be made the default. There has been a recent push toward *reproducibility*, as laid out recently in these pages [10]. Full reproducibility is a high hurdle, but it will already be a step forward if the code that accompanies a paper solves one of the problems shown in it.

Ultimately, as a community we need to come to grips with these issues. Otherwise, we will never move beyond solving toy problems, detached from what our applied colleagues need to see if they are to adopt the methods we are so good at developing!

References
[1] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang, *PETSc Users Manual*, Tech. Rep. ANL–95/11—Revision 3.3, Argonne National Laboratory, 2012.
[2] S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang, PETSc web page, 2013; http://www.mcs.anl.gov/petsc.
[3] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II—A general purpose object oriented finite element library*, ACM Trans. Math. Softw., 33:4 (2007), 24/1–24/27.
[4] W. Bangerth, T. Heister, and G. Kanschat, *deal.II Differential Equations Analysis Library*, Technical Reference, 2013; http://www.dealii.org/.
[5] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, and K.S. Stanley, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), 397–423.
[6] M.A. Heroux et al., Trilinos web page, 2011; http://trilinos.sandia.gov.
[7] R.J. LeVeque and M.J. Berger, *Clawpack Software 4.6*, 2013; http://www.clawpack.org/.
[8] A. Logg, *Automating the finite element method*, Arch. Comput. Meth. Eng., 14 (2007), 93–138.
[9] A. Logg and G. Wells, *DOLFIN: Automated finite element computing*, ACM Trans. Math. Softw., 37 (2010), 1–28.
[10] V. Stodden, J. Borwein, and D.H. Bailey, "*Setting the default to reproducible" in computational science research*, SIAM News, 46:5 (2013), 4 and 6.

Wolfgang Bangerth is a professor in the Department of Mathematics at Texas A&M University. Timo Heister is an assistant professor in the Department of Mathematics at Clemson University.