Better Software, Better Research

Carole Goble • *University of Manchester, UK*

We know that modern scientific research isn't possible without software, from short, thrown-together temporary scripts and the abundance of complex spreadsheets, through to the huge software enterprises behind international efforts such as the Large Hadron Collidor and the Square Kilometer Array. And it's not just research based on simulations and computational methods. Data-driven science (the so-called "fourth paradigm"[1]) wouldn't be possible without software to access and manipulate that data, and our ability to generate insights depends on software platforms.

My personal experience suggests little difference between the size of the research community and the size of the "research software community." Of 2,000 scientists Jo Hannay and colleagues surveyed online,[2] 91 percent said using scientific software is important for their own research, 84 percent said developing scientific software is important for their own research, 53.5 percent claimed to spend more time developing scientific software than they did 10 years ago, and 38 percent spend at least one fifth of their time developing software. Scientists aren't just using software; they are its prime producers. Some disciplines, such as biology, have spawned sub-disciplines, such as bioinformatics, with their own journals, funding streams, and cultures.

If software is pervasive in research, why is its vital role so often overlooked by funders, universities, assessment committees, and even the research community itself — even though the majority of researchers wouldn't be able to conduct their work without it? Mission-critical software is expected to be developed and maintained for the long term by a (temporary) untrained postgraduate during his or her coffee breaks.

Better Training, Better Production, Better Software

If your software is incorrect, so will be your science.[3] Mistakes in software happen to the best: Geoffrey Chang's discovery of the bug in his software led to his retraction of three *Science* papers.[4] We should admire him for his honest stance, and he's a better scientist for it. Many others don't even know they're wrong, or if they do, keep quiet. We can improve software quality at two major points in the research life cycle: while it's being produced and when its outcomes are subject to peer review.

Scientific software comes largely from two groups: highly trained software developers who work with research groups and are employed — typically — as postdoctoral researchers or research institute staff; and researchers self-taught in software development. Worryingly, in Hannay's survey, only 47 percent of scientists had a good understanding of testing, and just 34 percent thought any formal training was important. This is strange because presumably they wouldn't use and trust the results of a microscope or telescope that hadn't been built by qualified engineers or tested. Yet software is the most prevalent of all the instruments used in modern science.

Researcher training in software engineering practice isn't simply transposing curricula from computer science (CS) departments. We shouldn't aim to turn researchers into computer scientists or professional software developers, but should rather consider the particular context in which researchers work. Although CS often pays close attention to performance — especially in the high-performance computing area — research developers, even employing high-performance computing platforms, rank maintainability and portability of the code above performance.[5] Although developing new programming languages and environments is a significant part of CS, research developers favor

Fortran and C as programming languages, and R and Python as scripting platforms.

Consequently, best practices must be carefully tailored to meet scientists' needs.[6] Software Carpentry trains researchers in core software skills (www.software-carpentry.org). Started 16 years ago by Greg Wilson, a software engineer working with scientists, it has grown into an international network of instructors and contributors of course materials — all of them volunteers. Today's trainees become tomorrow's trainers. So far, Software Carpentry has helped more than 7,000 researchers worldwide through in situ two-day bootcamps, during which participants learn how to automate tasks, use the command line, use version control, and acquire best programming practices with, for example, Python or R. They're also introduced to software testing techniques, with particular emphasis on unit testing. In 2013, Software Carpentry became a part of Mozilla Science Lab (http://mozillascience.org).

Where software engineers are part of the team, training puts researchers in developers' shoes (and vice versa). A common area of conflict is the tradeoff between specialist applications and generic solutions. Computing professionals tend toward investment in long-term sustainable codes that users can customize, that are widely adopted, and that usually take longer to develop. Researchers often tend toward fast-return codes that specifically address them, their problem, their data, and their analysis, so they can quickly get out a result (before the competition beats them out) — but a result that might not be reproducible or reusable by anyone else. Of course, this varies between disciplines. Both views have right on their side. This means that scientific software development practice must follow the "working to working, jam today and more jam tomorrow" incremental model, while remaining cognizant of "technical debt" — that is, the work needed as a consequence of poor software design before a particular job can be completed. Unaddressed technical debt increases so-called "software entropy": as a software system is modified, its disorder, or entropy, increases.[7,8]

Better Access, Better Review, Better Software

One of my favorite overlyhonestmethods tweets (a hashtag for lab scientists) is Ian Holmes's "You can download our code from the URL supplied. Good luck downloading the only postdoc who can get it to run, though" (https://twitter.com/ianholmes/status/288689712636493824). An increasing number of journals now demand that code as well as data be openly available for review and reproducibility.[9] Researchers shy away from sharing their source code for a raft of reasons, including having to document and support it: after all, most researchers plan on building software not for others but for themselves, or people like them. Three further disincentives are embarrassment, scrutiny, and scooping.

IEEE Internet Computing
IEEE Computer Society Publications Office
10662 Los Vaqueros Circle
Los Alamitos, CA 90720 USA

Editorial. Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Internet Computing* does not necessarily constitute endorsement by IEEE or the IEEE Computer Society. All submissions are subject to editing for style, clarity, and length.
Submissions. For detailed instructions, see the author guidelines (www.computer.org/internet/author.htm) or log onto *IEEE Internet Computing*'s author center at ScholarOne (https://mc.manuscriptcentral.com/cs-ieee). Articles are peer reviewed for technical merit.
Letters to the Editors. Email lead editor Linda World, lworld@computer.org
On the Web. www.computer.org/internet/.
Subscribe. Visit www.computer.org/subscribe/.
Subscription Change of Address. Send requests to address.change@ieee.org.
Missing or Damaged Copies. Contact help@computer.org.
To Order Article Reprints. Email internet@computer.org or fax +1 714 821 4010.
IEEE prohibits discrimination, harassment, and bullying. For more information, visit www.ieee.org/web/aboutus/whatis/policies/p9-26.html.

Just like kids in kindergarten, researchers can be cruel about another researcher's code; authors are ashamed of their messy, poorly structured, and buggy software. Hence the open source semi-serious Community Research and Academic Programming License (CRAPL), which "absolves the authors of shame, embarrassment, and ridicule" (http://matt.might.net/articles/crapl/). Open software leads to much better software by potentially providing a community of contributors who are happy to improve it or at least comment on it, creating a kind of code review process. GitHub has been intensively developing several features specifically for the scientific community to support effective code sharing (https://github.com/blog/1840-improving-github-for-science).

In a system in which careers are based almost exclusively on publications, the fear that someone will find a serious bug in the released software, and all published results will be invalidated, is a strong one. Idealistically, sharing is caring; preventing research building on the top of incorrect results is good for the many (if not for the author). With the right kind of training emphasizing software testing, we should help mitigate situations in which scientific software is just plain wrong.

The fear that potential competitors will pick up the released code and "get there first" with the scientific results and a publication seems like a reasonable threat — but it could also be an urban legend. Wilson from Software Carpentry has set up an "Open Scoop Challenge," offering a t-shirt to anyone who can provide a fairly detailed story of "someone ever publishing a result you were going to, by taking advantage of software or data that you made publicly available" (http://software-carpentry.org/blog/2014/02/open-scoop-challenge.html). So far, no one has even submitted a story, let alone won a t-shirt.

Isolated development leads individuals to overestimate the ease with which others can use their software, the code's transparency, and even the possibility of locating the right version of the software (if it still exists). Setting aside issues of code portability (hard) and proprietary software licenses (tricky), getting reviewers to peer review papers is tough enough, as any editor knows. Getting them to run the codes is challenging. Getting them to scrutinize the software as a true representation of the algorithm is a leap. So we put our faith in open source, trusted (proprietary) platforms, and common libraries. Some have suggested introducing review teams of postdocs and postgrads as part of their accreditation, but little progress has been made, and this is unlikely to gain traction until we deal with the next point: recognition.

Better Recognition, Better Software

Anyone who works with software in academia will know that the level of recognition and reward for the software and those who review or develop it isn't proportional to its importance. The reward system is almost exclusively based on research publications. To get published, you must come up with something novel. Scientific software's goal is often to support the advance of research rather than being the output of the research itself. Unless you're in a particular area of CS or a sub-discipline such as bioinformatics, which has developed its own journals for reporting software, it's hard to get the research software itself published. Moreover, many activities are software maintenance — new functionalities or endless bug fixing — and hardly publishable. So, researchers must focus on creating novel, disposable code rather than providing reliable software for future use, or somehow subsidize their software development time.

Sharing software provides little merit, even if it's really useful for other researchers. You might well achieve fame within the specific research communities who use your software. You'll be rewarded with the gratitude of those who use your software to obtain results they publish in five-star journals, but this contribution is rarely rewarded by a university. The researchers you enable will progress with their careers, and you will be stuck with your amazing software that's essentially open source (unless you wanted to commercialize it). This state of affairs creates an incentive for not sharing your code, which is obviously detrimental to the research community and leads to wasted effort as researcher after researcher reinvents basically the same code that's been developed and purposefully siloed at other organizations. Mozilla Science's Code as a Research Object is a step toward getting credit for your code by archiving your GitHub code repository to figshare and receiving a citable DOI (http://mozillascience.github.io/code-research-object/). F1000Research has a similar service. Now, you just have to get people to use your DOI and cite the code they use.

The lack of recognition for software also manifests as a lack of recognition for those who develop it within academia. Let's turn to those highly trained software developers who work with research groups. A group in the UK has been campaigning to recognize these *research software engineers* (RSEs) with some success (www.rse.ac.uk). Researchers who rely on RSEs are more than aware of the value of their work. Without career paths within a university or research institute, it can be hard for RSEs to progress their career or gain reward or recognition for anything they do. Enterprising researchers hammer the square shaped peg of an RSE into any available shaped hole in an institute's employment guidelines; we can end up with software writers on a series of short-term contracts being judged on the number of papers they don't write. Unsurprisingly, retaining these talented professionals for progressing and sustaining research becomes difficult. This leads to my final point: funding sustainability.

Better Funding, Better Software

Software sustainability and the funding of software infrastructure is a recognized struggle in a research-funding environment founded on short-term bursts of (peer-reviewed) funds that are difficult to plan around. Production software is dressed in new clothes to claim novelty, and research codes are claimed to be production-quality to get service funds. We must regularly remind government, funding agencies, and investigators that investments in flashy machines aren't useful if we can't fund the means for porting software to them, and that data-generating instruments need software that can analyze their outputs. We must continually impress on funding bodies that software used as a platform to deliver services with a life beyond one reporting period or one project is a capital asset. Skilled engineers create something to fulfill scientists' long-term needs. The fact that software lacks the physical presence of a building or box doesn't make it any less "concrete."

We're making progress. The EU has funded European-wide research infrastructures for disciplines (for instance, ELIXIR for biology) and across disciplines (the European Grid Infrastructure) with emphasis on software sustainability, as well as promising centers of excellence in scientific software in its new Horizon 2020 program (http://ec.europa.eu/programmes/horizon2020/). The US has long-running software investments in infrastructure (IPlant Collaborative, DataONE). The NSF-funded Software Institutes for Sustained Innovation (S12) program has funded groups to work on specific codes and, more widely, to support particular disciplines such as water sciences, earth sciences, and computational chemistry, as well as cross-cutting concerns such as cybersecurity and trust, and science gateways. The Sloan Foundation supports international initiatives such as Software Carpentry, rOpenSci (for R), and ImpactStory (which provides alternative metrics of impact that include software production). And just recently, Phil Bourne, NIH's Associate Director for Data Science, has been proposing a "Science Commons" that emphasizes data, software, sustainability, and reproducibility.

In the UK, some funding councils have special calls for sustaining community-recognized codes, and the UK's House of Lords recently recognized that

Scientific software (e.g. meteorological and climate models, computational chemistry codes) is required to run on many generations of hardware. Software is the infrastructure and hardware the consumable.[10]

This point is fundamental: used code is long-lived code, and long-lived code decays. The need to continually nurture software won't surprise this magazine's readers. Open source can be beneficial, with "many hands helping," but as Scott McNealy of Sun Microsystems pointed out in 2005, "open source is free like a puppy is free." That is, it isn't. Someone has to look after it, and that someone needs paying. We have to educate grant holders that using open source software without thinking to support it isn't playing the game.

Of course, not all software can or should be sustained. We need new and fresh software, otherwise we will stagnate. However, the critical mass of expertise and development effort needed requires a user community to consolidate on key codes. Clustering around these requires international cooperation, national funders to support software developed elsewhere, and that grant investigators be able to reuse third-party codes — rather than reinvent their own — without prejudicing their proposals.

Better Software, Better Science

Although the problems that affect research software seem insurmountable at the moment, groups across the world are working to improve the status — and the use — of software in academia. An excellent example is the Software Sustainability Institute (www.software.ac.uk), which represents an innovative step from the UK's Engineering and Physical Sciences Research Council. The institute (I am an investigator) was founded in 2010 to "cultivate world class research with software."[11] It works across disciplines to develop exemplar research software in partnership with researchers, provide training to researchers both in person and online, foster relationships between researchers and software developers, and lobby policymakers to change software practices in research.

How can we increase the pace of change? Here are five things you can do: First, lobby your university to set up a group of RSEs whose time can be requisitioned by researchers within the institute. Not only will the RSEs receive recognition and reward for this service, they will also be retained on permanent contracts to ensure a continuity of service and retention of the best RSEs in exactly the same way that all other critical services are provided. Second, every doctoral school and PhD training program, regardless of the discipline, should incorporate basic software development training, based on courses such as those from Software Carpentry; in fact, volunteer to help this initiative. Third, lobby funding organizations, including the universities themselves, to implement policies to ensure that software is sustained if it has achieved a sufficient level of impact within the research community. Fourth, make your publicly funded software open source, and ensure that its benefit is made available to the widest possible community. Fifth, use others' software for your research and give them credit (did you use their code DOI in your paper?) and support (did you offer to contribute to its development?).

We must get software recognized as the first-class experimental scientific instrument that it is and get "better software for better research."

Acknowledgments

I'd like to acknowledge my Software Sustainability Institute colleagues Simon Hettrick and Aleksandra Pawlik for their invaluable help in preparing this article. The author photo is courtesy of Rob Whitrow (www.robwhitrow.co.uk).

References

1. T. Hey, S. Tansley, and K.M. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Microsoft Research, 2009.
2. J.E. Hannay et al., "How Do Scientists Develop and Use Scientific Software?" *Proc. ICSE Workshop Software Eng. for Computational Science and Eng.*, 2009, pp. 1–8.
3. Z. Merali, "Computational Science: ... Error ... Why Scientific Programming Does Not Compute," *Nature*, vol. 467, 2010, pp. 775–777.
4. G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, 2006, p. 314.
5. V.R. Basili et al., "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," *IEEE Software*, vol. 25, no. 4, 2008, pp. 29–36.
6. G. Wilson et al., "Best Practices for Scientific Computing," *PLoS Biology*, vol. 12, no. 1, 2014, e1001745.
7. A. Prlić and J. Procter, "Ten Simple Rules for the Open Development of Scientific Software," *PLoS Computational Biology*, vol. 8, no. 12, 2012; doi: 10.1371/journal.pcbi.1002802.
8. D. De Roure and C. Goble, "Software Design for Empowering Scientists," *IEEE Software*, vol. 26, no. 1, 2009, pp. 88–95.
9. V. Stodden, G. Peixuan, and M. Zhaokun, "Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals," *PLoS One*, vol. 8, no. 6, 2013, e67111.
10. "House of Lords Select Committee on Science and Technology Consultation on Scientific Infrastructure," HL paper 76, 21 Nov. 2013, paragraph 34.
11. S. Crouch et al., "The Software Sustainability Institute: Changing Research Software Attitudes and Practices," *Computing in Science & Engineering*, vol. 15, no. 6, 2013, pp. 74–80; doi: 10.1109/MCSE.2013.133.

Carole Goble is a full professor in the School of Computer Science at the University of Manchester, UK. Her research interests include e-Science, distributed computing, scientific workflows, Semantic Web, social computing, scholarly communication, and software engineering practices in science. Goble works in a range of science disciplines, notably biomedicine, systems biology, biodiversity, and social sciences. She's a fellow of the Royal Academy of Engineering and a recipient of Microsoft's Jim Gray award for e-Science. She is a co-investigator and co-founder of the UK's Software Sustainability Institute. Contact her at carole.goble@manchester.ac.uk.