THOSE WHO WILL NOT LEARN FROM HISTORY…

By Greg Wilson

I ATTENDED A WORKSHOP ON COMPUTATIONAL SCIENCE EDUCATION LAST SEPTEMBER THAT LEFT ME FEELING PESSIMISTIC ABOUT OUR DISCIPLINE'S PROSPECTS. THE ATTENDEES THEMSELVES WERE EVERYTHING ONE COULD HOPE FOR: INTELLIGENT, INVENTIVE, AND PASSIONATE PRACTITIONERS OF THEIR CRAFT, EAGER TO SHAPE A NEW GENERATION OF scientists. But their disregard for both the lessons of history and the issues that really matter to most working scientists left me wondering how different things will actually be in 10 years' time.

The first sign of trouble came when a distinguished British academic stood up to give the opening keynote. As he described how simulation and large-scale data analysis have made computing a third branch of science, I experienced a sense of déjà vu—he gave almost exactly the same talk 20 years ago. He used "peta" instead of "giga" this time around and put more emphasis on biomedical applications, but other than that (and more gray hair on both of our heads), we could have been back in Edinburgh in 1987. At no point did he or any of the other speakers say, "Here's why we didn't change the world then, and here's what we'll do differently this time around."

Because we clearly *haven't* changed the world, at least not in any fundamental way. Yes, the number of degree programs in computational science worldwide is growing, but only slowly, and almost without exception, students in the sciences and engineering learn no more about using computers effectively than they did two decades ago. It still takes them far longer to create programs than it should, those programs are still of unknown quality, and computational results are still practically impossible to verify or reproduce.[1] This is most definitely not what the future of computational science was supposed to look like.

Déjà vu turned to frustration when the second keynote speaker (just as distinguished as the first) stood up to talk about what computer science can offer computational science. His suggestion was complexity, machine learning, and other aspects of computing theory. I asked him afterward why "building software that works" wasn't on his 12-point list of essentials. He replied that it was—as a subpoint titled "formalizing intent" under point 11 ("automating intellectual processes").

I'm all in favor of formalization, but there's a lot more to the craft of building software, scientific or otherwise, just as there's a lot more to building a radio telescope than Maxwell's equations. Most computer science departments don't even teach this craft to their own students particularly well, but I believe it would have a greater impact on most scientists' working lives than anything else.

Having spent much of the past 20 years building programs for scientists and teaching them how to do so themselves, I've come to a few conclusions about what ought to shape future discussion about computational science education. First, while computational scientists want to build useful things, computer scientists want to study computing for its own sake. The former therefore have more in common with practitioners in industry and should look to them, rather than their academic colleagues, for instruction.

Second, I agree that computational thinking "means more than being able to program a computer. It requires thinking at multiple levels of abstractions."[2] However, most people (including scientists) can't really understand modularity, indirection, recursion, and other basic computing concepts without repeatedly implementing and debugging them. This means becoming proficient at programming, which takes time—so much time that it can't be squeezed into existing curricula without displacing other significant content. Saying you'll put a little bit into every course is fudging it: five minutes per lecture still adds up to three or four courses in a four-year degree. The obvious solution is to extend programs by a year, but only a few universities have been willing to bite that particular bullet.

Third, parallel computing is a distraction for most computational scientists. Parallel programs are much harder to write than sequential ones and much, much harder to debug. In a few cases, such as global climate-change modeling, the extra cost simply must be paid, just as there are a few cases in which a CT scanner makes the difference between life and death in a hospital setting. But just as most lives are saved by simple public health measures and basic first aid, so too are the needs of most scientists best served by sequential programs running on standard desktop hardware. For a variety of reasons, though, the high-performance computing minority continues to drive most of the discussion about computational science, to the latter's detriment.

Fourth, most computational scientists simply don't know how reproducible or reliable their results are. One reason is that journal editors and grant reviewers rarely (if ever) require evidence that the computational equivalent of good laboratory practices have been followed. It's therefore difficult or impossible to earn points toward tenure for "going the extra mile" to turn a program that runs into one that can be trusted. Various proposals for addressing this problem have gone nowhere, so as scientists increasingly rely on computational results, there's a growing danger that our discipline will produce some sort of "computational thalidomide." The only way to prevent this is to raise standards, which, in turn, means raising awareness and proficiency levels.

I believe the best way to address all of these issues is to make basic software development skills the core of every computational science education program. As an example, my colleagues and I began teaching these skills to computational scientists in 1997. Our work has since grown into an open source course used by companies, universities, and national laboratories in several countries that has had more than 150,000 visitors since August 2006 (http://swc.scipy.org).

One of the most important things we've learned from this experience is that, as in industry, raising proficiency improves productivity.[3,4] This could be the key to persuading scientists to change—the senior researchers who control laboratories and university departments will politely ignore discussions of quality, but pay real attention when told that improving basic skills reliably reduces the time spent building and maintaining software by roughly 20 percent.

We've also learned that the most effective way to introduce new tools and techniques is over an extended period, in a staged fashion, in parallel with actual use. Intensive short courses are much less effective (or compelling) than mentorship from someone with several years of development experience, although they're better than nothing. We've had some success in pairing CS and non-CS graduate students in project courses, but have often had to teach the CS students almost as much about software construction as their non-CS peers.

Finally, although scientists can learn a lot from commercial software developers, they need to tailor industrial practices to their particular needs.[5] To take one example, neither traditional "big design up front" methodologies nor their currently fashionable agile competitors are a perfect fit for computational scientists, who are almost always their own customers and who often modify their programs continuously as their questions are answered. Similarly, while being able to trace deliverables back to the data and code from which they're derived is useful in industry, it is (or rather should be) a sine qua non for any computationalist who wants to call his or her work "science."

I still firmly believe that computing has the potential to revolutionize science. But I also believe that if we don't learn from past efforts to make it part of mainstream scientific education, this potential will go unrealized. Focusing on everyday skills might not give university presidents something to issue a press release about, but they'll have more impact on the quality of computational science in the coming decade than anything else we can do.

Acknowledgments

I thank Brent Gorda (Lawrence Livermore National Laboratory) and Andrew Lumsdaine (Indiana University), who were coauthors on the position paper this article is based on, for their many helpful conversations and comments, and Paul F. Dubois (well-known to readers of this magazine), Diane Kelly (Royal Military College, Canada), Rubin Landau (Oregon State University), Russ Thomas (National Research Council), and Diane Horton (University of Toronto) for their helpful feedback.

References

1. G. Wilson: "Where's the Real Bottleneck in Scientific Computing?" *Am. Scientist*, vol. 94, no. 1, 2006, pp. 5–6.
2. J.M. Wing, "Computational Thinking," *Comm. ACM*, vol. 49, no. 3, 2006, pp. 33–35.
3. R.L. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2002.
4. S. McConnell, *Rapid Development*, Microsoft Press, 1996.
5. D.F. Kelly, "A Software Chasm: Software Engineering and Scientific Computing," *IEEE Software*, vol. 24, no. 6, 2007, pp. 188–120.

Greg Wilson is an assistant professor in computer science at the University of Toronto, where his primary research interest is the application of lightweight software engineering methods and tools to computational science. Contact him at gvwilson@cs.toronto.edu.