Workshop on Computational Education for Scientists – Position Paper

Devika Subramanian
Computer Science Department, Rice University

I am deeply interested in the problem of computational education for scientists for two reasons.

(1) Computation has pervaded all scientific and engineering endeavors. Six of the top ten discoveries in Science last year were enabled by computation. For example, the T-cell signaling network was computationally reconstructed *ab initio* from data. Design in engineering disciplines involves significant amounts of computational prototyping. For example, the design of the Pringle chip was done using super computers.

(2) We as computer scientists have a unique opportunity to contribute to the training of the next generation of computational scientists. I do not believe we are doing the best we can with our current approach. Even more importantly, at my institution, the engineers do not believe that we are training their students well, and have taken over the training process through courses they teach their own students. This has become possible because newly hired faculty in sciences and engineering are able to design new courses in computational modeling and high performance computing that are specialized to their own disciplines. We are at a crossroads right now, and we need to decide what role we, as faculty in academic CS departments, intend to play /can play in the education of scientists and engineers.

Before outlining what I believe are key components of training for the next generation of computational scientists, let me share an anecdote here from a class in computational biology that I taught last Spring. Students had to build first-order Markov models of CpG islands drawn from human Chromosome 22. I had supplied the start and end points of these islands on the chromosome, and the chromosome itself is read into a giant array. I received an irate note from a student (senior in bioengineering) saying that the assignment was impossible to complete because it was taking way too long to just read the islands. Here is a snippet of the Matlab code I received from the student.

```
 k =0;
 while(k <= start_island[i]) {
   k++;
 }
while (k <= end_island[i]) {
  // counting code here
 k++;
 }
```

The student had no conceptual understanding of an array and thought of it as a singly linked list. Since chromosome 22 is several hundred thousand base pairs long, this conceptual misunderstanding increased code execution time significantly. Most introductory CS courses for scientists and engineers focus on the mechanics of languages like Java or C or Scheme or Matlab and teach very little problem-solving, data structures and algorithm design. As former chair of the Curriculum Committee for the

Computer Science Department at Rice University, I got an opportunity to examine our "service" course offerings to the other departments in the School of Engineering. In a series of town hall meetings with students and faculty in the ECE department, which is our "major" customer, I got to understand that we were simply not meeting the needs of the students as well as the faculty. Students complained that our introductory course in Scheme focused on recursion which they never encountered in their engineering design classes, and the professors complained that students could not handle real languages like C and Fortran and had no clue how to integrate and extend large existing codebases.

We have to get science and engineering disciplines to accept the fact that a one or two course sequence is simply not sufficient to prepare their students. They need to treat CS like mathematics in the undergraduate curriculum and get used to having at least four courses (one or two every year starting from freshman year). We cannot use the same courses that we train our CS majors on, for this purpose. We have to work with the disciplinary scientists to produce courses that prepare them to use computation as a tool for problem solving in their fields. A first course on computational modeling working with high-level scripting languages like Mathematica/Matlab and using significant examples from their own area is key. Students are not impressed by abstract examples that are not connected to their other experiences. Rich examples drawn from mathematics and physics that they take in their freshmen year seems to help them connect with the material better. We need to then move into lower levels of abstraction exposing them to languages that are closer to the machine level than Mathematica and Matlab (e.g., C/Java/Fortran/C#/C++). We need to introduce them to programming in the large, software architectures and software engineering. In their junior/senior years we need to equip them with tools to handle distributed and parallel computing to solve problem of scale. Somewhere in between we need to give them rudiments of hardware architectures, compilers and OS. For many disciplines, graph theory and algorithms are very important (analyzing infrastructure networks in Civil Engineering requires detailed knowledge of graph theory and graph algorithms, for example), as is dynamic programming for discrete optimization problems. Matrix algebra and matrix computations which students typically get from the mathematics department should be tied closely with computation, so students understand how to compute with these structures. Statistical data mining is also becoming an important data analysis tool in many engineering disciplines and exposure to tools and the principles behind them should be added. For the first course in computation suitable for scientists and engineers there are at least two successful models – E77 at UC Berkeley based on Matlab, and John Browne's course based on Mathematica. MIT's computation course for civil engineers (Java-based) is a reasonable follow-on course.

I believe that computation can be introduced into the K-12 curriculum as early as the 5th grade, when algebra is (or should be!) introduced. There are many programming frameworks (MIT's Scratch is popular, and one can imagine constructing a framework based on Mathematica) that can be used for this purpose. Revising the current AP curriculum so CS is not confused with programming will go a long way to improving the perception of computing among future scientists and engineers.

We will know we are succeeding in our educational efforts when our colleagues in science and engineering understand that computational modeling is a distinct intellectual endeavor that can be

learned and that it is crucial for successful use of computation to solve problems in science and engineering. Discrete models of systems are important to have, and are not generally covered in mathematics courses that are taught to science and engineering students. Software failure modes are much more complex than failure modes of most physical systems. Most physical systems do not generally have a combinatorial number of operating “regimes”. In addition, we need to make our colleagues in science and engineering appreciate that designing good software systems requires a unique set of engineering skills that can be taught to them by computer scientists. They cannot build robust systems by writing out specifications and throwing them over the wall to a CS graduate.