AN ENTRY-LEVEL COURSE IN COMPUTATIONAL ENGINEERING AND SCIENCE*

Joseph L. Zachary, Christopher R. Johnson, Eric N. Eide, and Kenneth W. Parker

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

In conventional science and engineering degree programs, computation is commonly treated as a narrow technical tool to be studied and applied in isolation from the traditional topics of the discipline. Students are typically required to take a programming course (usually in Fortran, C, or Pascal) to learn how to program, and perhaps a numerical analysis course (covering numerical integration, root finding, and the like) to learn ways to apply their programming skills. However, many students then go through the rest of their degree programs without ever using computation in any significant way.

This approach ignores the fact that computation has evolved into an essential way of illuminating scientific and engineering principles. Researchers now choose among the theoretical, experimental, and *computational* approaches to studying scientific phenomena. It is becoming increasingly clear that computation should be exploited in science and engineering education for its descriptive and analytical powers [1–3]. We believe that it is as unreasonable to teach science and engineering without computing as it would be to do so without calculus and differential equations.

To encourage the integration of computation into the science and engineering curricula, we have designed and taught an *introductory* course in computing expressly for science and engineering students. Our goal was to create a course to satisfy the standard programming requirement while preparing students to immediately exploit the broad power of modern computing in their science and engineering courses. Our course has at least four distinguishing features:

- We use a symbolic mathematics package (Maple) in combination with a conventional programming language (C or Fortran).
- We teach programming concepts in parallel with a scientific problem-solving methodology.
- We draw upon a variety of computational problems from the breadth of science and engineering to interest students and establish the relevance of the computational problem-solving approach.
- We have developed an extensive suite of interactive, on-line laboratory materials that students can use via any HTML viewer, such as NCSA Mosaic.

In this paper we explore the motivations behind our course, discuss its content and our experience with teaching it, and survey related efforts.

To appear in the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education, March 1995.

Motivation

The barriers to student exploitation of computational techniques were quite high in the recent past. A student had to invest a substantial amount of effort to learn the intricacies of a programming language and its compiler, a text editor, an operating system, and a moderate amount of numerical analysis before he or she could even begin to obtain useful results. Such barriers ensured that most science and engineering majors would never make effective use of computational techniques during their undergraduate years.

The emergence of computational packages such as Maple† has greatly lowered these barriers. These programs perform symbolic calculations, provide robust implementations of numerical techniques, support the visualization of two- and three-dimensional functions, and even provide animation capabilities. The computational power made available per unit of student effort is orders of magnitude higher for these packages than it is for traditional programming languages. With these tools in hand, the student can easily put into computational practice the mathematical ideas he or she has learned in a traditional course such as calculus, linear algebra, or differential equations.

Besides providing ready support for many kinds of mathematical calculations, Maple is also an effective tool for illustrating the kinds of programming concepts typically found in an introductory programming course.

---

*Work supported in part by the Department of Energy.

†Examples of similar packages include Mathematica and Matlab. We chose Maple for our work, and discuss its use in this paper, simply because our university has a site license for Maple.

Roughly speaking, Maple provides both a high-level interpreted language for doing symbolic and numerical calculations as well as an embedded procedural programming language. Without even bothering to use the embedded procedural language, it is easy to use Maple to illustrate such topics as variables and assignment, built-in and user-defined functions, functional abstraction, parameter passing, higher-order functions, arbitrary precision integer arithmetic, adjustable precision floating-point arithmetic, conditionals, iteration, the differences between numerical and symbolic computation, and function libraries. Because it is interpreted, because it is higher level, and because its capabilities more closely match the computing needs of students, we have found that Maple is far superior to a language such as C for motivating and illustrating these concepts. We have also found that a student's background in Maple is invaluable when he or she subsequently learns C.

These observations notwithstanding, we have encountered a high level of resistance on the part of many science and engineering faculty members to incorporating this new form of computing into the science and engineering curriculum. There are a number of reasons for this. Some faculty are simply not aware of the power of modern computing environments such as Maple and fail to appreciate the role that they can play in the classroom. Other faculty, regarding the computing packages as a fad, hold to a narrow view that identifies computing with Fortran or C programming. Finally, more than a few faculty view the pronouncements of computer scientists (who have encouraged them to switch from Fortran to Pascal to C to C++) with suspicion, and remain unconvinced that it is worth the effort to overhaul their approach to computing yet again.

We believe that the biggest reason for this resistance, however, is that science and engineering students are not suitably prepared by the single programming class that they typically take to exploit modern computing technology in their major classes. As a result, the students do not see the relevance of computational techniques to their undergraduate educations, and the professors in their major classes are for the most part unwilling to provide the missing background. Introductory computing classes are most often designed, not surprisingly, from a computer science perspective, which often fails to take into account the needs of students from other disciplines. In fact, we have found that the requirements for a successful introduction to computing for science and engineering majors are quite different than those for computer science majors. There is substantial overlap, to be sure, since the abstract concepts being taught are the same. However, it is important in the case of science and engineering majors to emphasize different tools and to establish relevance by using appropriate motivating examples.

From the point of view of a science or engineering major, for example, learning to write and analyze a sorting procedure in Pascal is not nearly as valuable as learning to analyze a one-dimensional cooling problem by creating and animating a discrete approximation in Maple. Both problems use arrays, exploit iteration, involve convergence, illustrate procedural abstraction, and are of comparable difficulty. The cooling problem, however, is rather unlikely to appear in a standard introductory programming class, and the animation aspect is all but impossible for a beginning programmer to code in a conventional language.

The Course

We set out in the summer of 1993 to develop our new course, which we named *Engineering Computing*. The initial impetus for developing the course came from two sources. The dean of our College of Engineering asked us to design a modern alternative to the single programming class (in Fortran or C) that nearly every student in the College is required to take. At about the same time, we became charter members of the Department of Energy's national working group on Undergraduate Computational Engineering and Science (UCES) [4]. The group's first goal was to develop a freshman-level computing course for science and engineering majors, and because of the convergence of interests our effort was adopted by UCES as its prototypical freshman course. This combination of local and national influence has been valuable to our efforts and has continued to shape our course design up to the present.

After six months of development, we offered the class for the first time during the winter quarter of 1994. After six more months of reflection and revision, we offered it again during the autumn quarter of 1994. In the remainder of this section we will discuss the original design decisions that shaped the course, the experience we gained from the initial offering, and the resulting changes we made prior to the second offering.

Language Issues. Our first and most critical decision was to use both a symbolic math package and a conventional programming language in the course. For the reasons we have already outlined, we believed from the outset that *Engineering Computing* should be centered around a symbolic math package. We chose to use Maple (instead of Mathematica or some other application) simply because our university had a site license for Maple. There is nothing in our approach that is specific to Maple, and in fact many of our course materials have been converted for use with Mathematica.

The decision to use a conventional language in addition to Maple had both technical and political motivations. Despite the power and convenience of Maple, we feel that some facility with a traditional programming language is still important for students of science and engineering. There are extensive libraries of Fortran and C routines for performing various numerical computations, and students should have at a minimum the ability to interface with these resources. Furthermore, some programming tasks are easier to do in a conventional language.

Early in the design effort we met with faculty members from each of the five other undergraduate departments in our College of Engineering. The purpose of these meetings was to solicit input into the evolving design of the course. We paid particular attention to this input because we needed to build a consensus in each department that our course should be adopted as an alternative way to satisfy their undergraduate programming requirement. While each department was supportive of the idea of using Maple to give students easy access to effective computing and to introduce programming concepts, each was also adamant that a conventional programming component be included. Furthermore, some departments insisted that the conventional language be C, while others insisted that it be Fortran.

We decided to structure the initial offering of the course as follows. In the first half we would use Maple to entice the students with its power while preparing them to learn a conventional language. In the second half we would present expressively equivalent subsets of Fortran and C in parallel. We made a virtue of this necessity by positing the idea that by learning Fortran and C in parallel, the students would quickly realize that syntactic details are superficial and would thus be better equipped to focus on the common underlying semantic ideas.

In practice, although teaching Maple in combination with a conventional language proved to be an effective strategy, treating both Fortran and C in parallel was too much. We realized this during the course's first offering, at which point we asked each of the students to pick the language in which they wished to do their homework assignments. Some chose Fortran and some chose C, but this worked out well in the laboratories since our teaching assistants could provide help with either. Unfortunately, with the students split into two camps, it became awkward to present concrete programming examples in the lectures.

For the second offering of the course, we removed Fortran and based the course entirely upon Maple and C. Our current plan is to offer Maple/C and Maple/Fortran versions of the course in different quarters during each academic year. This will allow us to satisfy the requests of the engineering faculty while avoiding the problem of overloading the course with too much material.

Interactive Laboratories. A second critical decision that we made early on was to develop a suite of interactive, on-line laboratories to support the course. We had previously met with great success in preparing such laboratories to support courses in Unix, Fortran, and C [5,6], and were understandably interested in building upon this experience. Accordingly, we structured the course to meet twice a week for one-hour lectures, and twice a week for two-hour labs.

The laboratory materials (lessons) are hypertext documents that allow the user to read sequentially, browse randomly, and display graphics. The lessons also contain hyperlinks that, when clicked by the user, will execute commands in another window. We used this feature when guiding students through Maple. This is a handy capability because it allows us to carry out demonstrations in the Maple window without requiring the user to either type or cut and paste commands, both of which are error-prone activities. During the first course offering the students viewed the lessons via a custom hypertext viewer, but prior to the second offering we converted the lessons into HTML and used NCSA Mosaic, extended with Hamlet [7] (our external viewer for handling inter-window communications), as the viewer.

Each of the original lessons [7] typically introduced some aspect of Maple, Fortran, or C, used it to solve a problem from science or engineering, and ended with a set of problems for the student to do as homework. Students were expected to work on one lesson per two-hour laboratory session and were supported in each lab session by 2 TAs for every 25 students. Our initial suite of lessons was broken down as follows:

- Six lessons on Maple with an implicit introduction to programming concepts.
- Six lessons devoted to a parallel development of Fortran and C.
- Three lessons employing a combination of Maple, Fortran, and C to solve classical numerical problems in root finding, integration, and curve fitting.
- One lesson on using NCSA Mosaic to access the Internet.

The lessons proved to be the most important means of communicating the course material. Although the students had access to background reading on Maple and the programming languages, our laboratory materials essentially constituted the textbook for the course. While student reaction to the lessons was positive, our initial offering revealed room for improvement in the following three general areas: the level of the lessons, the centrality of the motivating examples, and the encouragement of good problem-solving skills. We consequently revised the lessons extensively prior to the second offering of the course.

Most of the original lessons covered perhaps twice as much material as was appropriate. The lessons typically used too many motivating problems to illustrate too many computing concepts at too high a level of mathematical rigor. UCES colleagues who are working on

Problem	Key Concept
Area of earth	Significant digits
Horizon distance	Floating-point error
Cantilevered block stack	Built-in functions
Compound interest	User-defined functions
Ballistic trajectories	Visualization
Ship propulsion	Abstraction
Bisection method	Procedures
Weightlifting kinematics	Program structure
Weightlifting kinematics	Simple statements
Rod stacking	Functions
Sliding block	Conditionals
Root finding	While loops
Numerical integration	For loops
Heat flow	Arrays
Curve fitting	Array parameters
Visualizing heat flow	File input/output

Table 1: Summary of revised lessons

related course materials have observed the susceptibility of such material to "upward creep" in scope and difficulty. The typical revised lesson uses one motivating problem to illustrate one or two key computing or problem-solving concepts. The most complicated mathematics that we use in any of the lessons is simple integration.

Each of our original lessons introduced several new concepts and used them to solve one or two representative problems. Even so, we decided that many of the lessons were not sufficiently centered around the problems. In retrospect, this is evident from the breakdown of the lessons listed previously: they are organized around computing paradigms rather than problems. Each of the revised lessons, in contrast, is organized around (and most often titled after) a problem. After introducing the problem, we develop computing concepts and problem-solving techniques as appropriate to the particular problem.

There are sixteen lessons in the revised lesson set [7]. While the first seven are centered around Maple and the last nine around C, the partitioning is not absolute. Table 1 summarizes the motivating problems and key concepts contained in each lesson.

In our initial offering of the course, we discovered that a large number of our students had rather weak problem-solving skills. For example, many had a difficult time correctly implementing functions to perform unit conversions because they were not familiar with the idea of doing simple order-of-magnitude sanity checks. We had naively assumed that the students would be able to learn problem-solving skills by example. In the revised lessons we have been much more deliberate about teaching a problem-solving methodology. The UCES group has identified the following five-step approach to solving computational problems, around which we have carefully organized the solutions to our example problems:

- Problem. Clearly and unambiguously identify the problem to be solved.
- Model. Use simplification and abstraction to arrive at a mathematical model for the problem.
- Method. Identify a computational method for solving the model.
- Implementation. Implement the method in the appropriate language.
- Assessment. Concurrently with the other four steps, actively assess the problem-solving process to identify mistakes and unwarranted assumptions.

An Example Lesson. To illustrate our evolving sensibilities on the structure of lessons, we will describe here one of our revised lessons. This lesson is entitled "Kitty Hawk," and begins with the following narrative:

> The Wright brothers' flight was made from the highest natural point along the Outer Banks at Kill Devil Hill, which is a broad sand dune that peaks at 38 feet above Albemarle Sound. You are sitting on top of Kill Devil Hill on a calm day with no waves, watching a friend swimming across the sound toward the mainland. Because of the curvature of the earth, your friend will eventually disappear across the horizon. How far away (in feet) can your friend get before he disappears from view?

Given the problem, the next step is to come up with a model. We encourage the students to make a sketch, from which it is apparent that we can model the problem with a right triangle whose hypotenuse runs from the top of the hill to the center of the earth. Assessment reveals that we are assuming that the earth is a perfect sphere, and that we are going to be calculating line-of-sight distance to the swimmer. The method that we choose to solve the model is, not surprisingly, the Pythagorean Theorem.

At this point we make a digression to explain a bit about floating-point numbers, which is one of the two key concepts of the lesson. Although the students have been exposed to floating point in an earlier lesson, at this point we explain about the mantissa and the exponent and observe the potential for floating-point error. This process is simplified in Maple because it is possible to explicitly control the number of digits in the mantissa.

We are now ready to implement the problem-solving method in Maple. Solving for the unknown side of the right triangle involves plugging into and evaluating the expression $\sqrt{(R+h)^2 - R^2}$, where R is the earth's radius and h is the height of the hill. With a six digit mantissa, Maple calculates the answer to be zero! Increasing the size of the mantissa yields more reasonable-looking answers, but we challenge the students to determine what went wrong when there were six digits of mantissa.

The problem, of course, arises from the fact that R is very much larger than h. We show the student that by algebraically simplifying the original expression to $\sqrt{2Rh + h^2}$ before plugging in the numbers, six digits of mantissa are more than sufficient. This gives us an opening to discuss how some computations, especially those involving subtractions, are not stable, and how algebraically equivalent expressions are not generally computationally equivalent. This insight is the other key concept of the lesson.

Experience. We first offered *Engineering Computing* during the winter quarter of 1994 to approximately 70 undergraduates. Although the students pointed out a number of ways to improve the course and the lessons, they overwhelming felt that the course had been useful, that they would be able to apply what they learned in their other classes, and that they would recommend the course to friends. Based upon the students' suggestions and our own observations, we revised the lessons before offering the course again during the autumn quarter of 1994, again to 70 undergraduates. Preliminary feedback indicates that the revisions represent a tremendous improvement.

Dr. Thomas Marchioro of the Ames Laboratory and the executive director of UCES, translated our original Maple lessons into Mathematica and subsetted our original Fortran/C lessons into Fortran. These modified lessons were then used by Prof. J.C. Diaz at the University of Tulsa during the summer quarter of 1994. Individual lessons are being adapted by various members of UCES for use across the country during the 1994–95 academic year.

We will be offering *Engineering Computing* one more time during the 1994–95 academic year, and plan to largely phase out our department's service courses in Fortran and C, in favor of our new course, during the next several years.

Related Efforts

Charles Swanson has compiled a list of twelve universities across the country that currently offer undergraduate programs in computational science [8]. While this list is almost certainly incomplete, it reveals that most of the curriculum development at these twelve schools has been focused on the advanced undergraduate level. There is a preponderance of courses on numerical methods, parallel scientific programming, and high-performance computing.

A number of sites have developed on-line laboratory materials in much the same spirit, although often at a higher level, as ours. A good example is the work of Profs. Dan Warner, Steve Stevenson, and their colleagues at Clemson University [9]. They have developed a series of computational science laboratories aimed at the advanced undergraduate and beginning graduate student.

The *Engineering Computing* lessons [7] build upon the experience that we acquired by developing lessons in support of our service classes in Unix, Fortran, and C over the last three years [5,6]. Over 2000 students have taken classes taught either entirely or in large part with these on-line lessons.

The Undergraduate Computational Engineering and Science working group [4] of the Department of Energy has as its goal the development of course materials for an entire range of undergraduate-level courses in computational engineering and science.

References

[1] A National Computing Initiative. The Agenda for Leadership. Report of the Panel on Research Issues in Large-Scale Computational Science and Engineering, by H.J. Raveche, D.H. Lawrie, and A.M. Despain. SIAM Workshop, Leesburg, Virginia, 1987.

[2] Future Research Directions in Problem Solving Environments for Computational Science, S. Gallopoulos, E. Houstis, J. Rice, eds., NSF Report, 1991.

[3] J.R. Rice, Academic Programs in Computational Science and Engineering, *IEEE Computational Science and Engineering*, Vol. 1, No. 1, p. 13, Spring 1994.

[4] The UCES project's home page is available at URL `http://uces.ameslab.gov/uces`.

[5] Information about the on-line laboratory material for Unix, Fortran, and C is available at URL `http://www.cs.utah.edu/~zachary/tutor`.

[6] J.L. Zachary, Tutorial-Based Teaching of Introductory Programming Classes, *Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, March 1994.

[7] Information about Hamlet and the original and revised *Engineering Computing* lessons can be accessed at URL `http://www.cs.utah.edu/~zachary/hamlet`.

[8] Available from Chuck Swanson at Cray Research Inc., `cds@cray.com`.

[9] The Clemson CSE Program's home page is at URL `http://www.cs.clemson.edu/CSE/homepage.html`.