An Introduction to Computing for Engineers: New Approaches to Content and Pedagogy

Joseph L. Zachary
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

zachary@cs.utah.edu

Abstract

The descriptive and analytical powers of computation must be exploited in engineering education. Unfortunately, the introductory programming classes that are taken by engineering majors often suffer from two failings. First, they frequently ignore the context in which programming is actually done. Second, they are most often designed for students planning to major in computer science. We have developed a new introductory programming course designed expressly for engineering majors. It is supported by the recently-published textbook Introduction to Scientific Programming *and by a Web site containing interactive online material.*

Introduction

The descriptive and analytical powers of computation must be exploited in engineering education. Researchers now choose among the theoretical, experimental, and *computational* approaches to studying scientific phenomena. As a result, computation has evolved into an essential way of illustrating scientific and engineering principles [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. Unfortunately, the introductory programming classes that are taken by engineering majors often suffer from two failings.

First, introductory programming classes often ignore the context in which programming is actually done. When the goal is to teach students how to write programs, the temptation is great to tell the students exactly what programs to write. This means, however, that the person who makes up the assignments ends up doing what is often the most interesting part: reducing a real-world problem to a programming problem.

Students can become so used to having someone else do this creative work that they end up with no idea how to apply their programming skills to problems that arise elsewhere. As a result, they tend not to make use of computing in their science and engineering classes unless they are explicitly told to do so. Even then, many students are completely lost until someone tells them what program to write and how to go about writing it.

Second, introductory programming classes are most often designed for students planning to major in computer science. Even when the problems that inspire computational solutions are highlighted, they are often not particularly relevant to engineering students. More fundamentally, the technical slant on programming that is appropriate for computer science majors is not always appropriate for engineering majors.

The treatment of floating-point numbers is an excellent case-in-point. Conventional introductory courses in computer science make almost no use of floating-point numbers; scientific programming, in contrast, is all but impossible without them. Computer science students might be taught about the IEEE standard for representing floating-point numbers; engineering majors must be taught the computational properties of floating-point numbers and their role in representing inexact physical measurements.

We have developed a new introductory programming course, *Engineering Computing*, designed expressly for engineering majors. This course satisfies the standard programming requirement while better preparing students to exploit the broad power of modern computing in their engineering courses. Our course, which we have taught seven times since 1994, has four distinguishing characteristics.

- It uses a symbolic mathematics package (Maple) in combination with a conventional programming language (C). The computational power that Maple makes available per unit of student effort is orders of magnitude higher for Maple than it is for conventional languages. Furthermore, learning Maple provides a

solid foundation for learning conventional languages such as C.

- It teaches programming concepts in parallel with a computational problem-solving methodology. Students not only learn how to write programs, but how to reduce problems to the point where they can be solved with a program. They also learn how to interpret the results those programs produce.
- It draws on a variety of computational problems from the breadth of science and engineering. This piques the interest of students and establishes the relevance of the computational problem-solving approach.
- It relies not just on lectures, but on an extensive suite of custom-designed online laboratory materials delivered via the Internet. These materials exploit Java applets as well as other innovative online technologies that we have developed.

Our course is supported by the recently-published textbook *Introduction to Scientific Programming* [4] and by a Web site containing the online material [5].

Programming Languages

There is no single best language for either beginning or advanced programmers to use. Programming languages are nothing more than sophisticated tools. The appropriate choice of programming tool depends on such factors as the nature of the programming problem at hand, the availability of solutions to similar problems, the programmer's background, and the computing environment in which the programmer is working.

In recognition of this reality, we use Maple in the first half of *Engineering Computing* and C in the second. Maple and C are representatives of two broad classes of tools that are widely used for scientific programming. Maple is a symbolic algebra system, and C is a conventional programming language.

Maple is a system for doing numerical, symbolic, and graphical mathematics, and it is also the name for the programming language that is used within that system. We chose to use Maple instead of C in the first half of the course for a number of reasons. First, Maple is easier to use and to understand than is C. Second, it is possible to do interesting things in Maple with only a modicum of training. Our students have almost always found immediate applications for Maple in their other classes. Finally, most of the basic concepts that a student must master to understand Maple carry over to the process of learning C. We exploit this fact in the second half of the course.

Maple is not the only symbolic algebra system. Its major competitors are Mathematica and Macsyma, and we could have equally well used either in place of Maple. We chose Maple simply because the University of Utah has a Maple site license.

C is a conventional programming language that is widely used for all types of programming, including scientific programming. Despite the power of symbolic algebra systems such as Maple, proficiency in conventional languages remains an essential skill for scientific programmers. Conventional languages have more than 40 years of development behind them, and a large body of software written in such languages as Fortran and C is available for reuse. Conventional languages are more flexible than symbolic algebra systems, and for large applications are also considerably faster.

There are a large number of conventional programming languages, but the two that are most often used for scientific programming are C and Fortran. We arbitrarily chose C over Fortran, but we could have developed essentially the same course using Fortran in place of C. The truth of the matter is that Fortran and C—at least when used for scientific programming—are much more alike than they are different.

Despite the fact that it features two programming languages, the course is not made up of two unrelated halves. We use Maple to examine such issues as the computational properties of numbers, the use of operators and functions to build expressions, the exploitation of function libraries, and the organization of programs around programmer-defined functions. All of this material from the Maple half carries over to the C half. This means that instead of starting over from scratch when we introduce C, we are able to pick up where we left off in our discussion of Maple.

We do not attempt to cover all of either Maple or C. Maple is far too large to even imagine covering in our course, and C contains a number of aspects that are either irrelevant to scientific programming or are easily picked up from any C reference after the material in the course is covered. Our idea is to show a representative and useful subset of each language in the context of computational problem solving.

Problem-Solving Methodology

The process of solving a problem in computational science does not begin with writing a computer program. In fact, much creative work must be done before the first line of a program can be written. In *Engineering Computing* we idealize the process of computational problem solving into five steps, and we emphasize the use of these steps throughout the course.

The first step is to thoroughly explore the problem that is to be solved. Our students must be sure they know what question is being asked, what theoretical and experimental knowledge they can bring to bear, and what kind of an-

swer is required. They must also be sure that the problem is amenable to a computational solution. Many of the problems that we use in our course are of the kind encountered in high school or college physics classes. They variously involve mass, velocity, acceleration, force, gravity, friction, drag, springs, and heat.

The second step in the problem-solving process is to create a mathematical model of the problem being solved. Our students must reduce the problem as originally stated into one expressed in purely mathematical terms, using mathematical expertise to extract the essentials from the underlying physical description of the problem. The modeds that arise in our course make extensive use of algebra, geometry, and the trigonometry of right angles, and much more limited use of the law of cosines, differentiation, and integration.

Once a problem has been described in mathematical terms, the third step is to settle on a method for actually calculating the solution. It is usually easier to set up the solution to a problem mathematically than it is to follow through and find the answer. In the course of identifying a method our students must often employ elementary numerical analysis. We assume no particular background in numerical analysis, but along the way we treat such topics as floating-point numbers, significant digits, interval arithmetic, numerical root finding, numerical integration, and the finite-element method.

Once a computational method for dealing with the mathematical model has been identified, the fourth step is to implement and carry out the method on a computer. Our students use Maple and C as appropriate for all of the programming in *Engineering Computing*. We assume no particular background in programming beyond a very general form of computer literacy, but to succeed the students must be willing to experiment.

The fifth and final step is to assess the solution. In many respects assessment is the most open-ended and difficult of the five steps involved in solving computational problems. Numerical answers can never be taken at face value. Errors and assumptions, both intentional and unintentional, inevitably creep into the problem, model, method, and implementation. Our students must ask themselves how large an error is likely to be included in the solution, try to fix the problems that caused them, and quantify those errors that are unavoidable.

Motivating Problems

We never introduce a new programming or numerical analysis topic without first motivating it with a problem. This shows our students that what they are learning is indeed practical and suggests ways in which they might apply computing in their science and engineering classes. We have summarized below some of the motivating problems that we use in *Engineering Computing*.

- The problem of determining the average population density of the earth is the starting point for showing how to use arithmetic expressions to do simple calculations, and for illustrating the differences between Maple's rational and floating-point numbers.
- The geometrical argument that Eratosthenes used to determine the circumference of the earth in 225 B.C. leads into a discussion of the uncertainty inherent in calculations involving physical measurements, an explanation of the methods of significant digits and interval arithmetic, and an introduction of the notions of variables and assignment statements.
- Finding the line-of-sight distance to the horizon from the top of a hill leads to two different computational methods. The first leads to an implementation that suffers from a catastrophic loss of significance. The second uses an algebraic simplification to avoid the difficulties of the first method, and also leads to an implementation that exploits programmer-defined functions.
- Solving the problem of predicting the population growth of the United States requires using Maple's symbolic programming capabilities to develop the models of simple, compound, and continuous interest.
- Analyzing the ballistic motion of a projectile leads naturally into the problem of visualizing trajectories, which leads in turn to the study of Maple's plotting and animation capabilities.
- Calculating the power required to move a ship along a one-dimensional trajectory through the water entails learning about Maple's support for both differentiation and Boolean relational expressions. It also involves writing fairly involved programmer-defined functions.
- Finding the radius of an arc that bisects a circle leads to a simple transcendental equation that defies symbolic solution and motivates the numerical approach to root finding. The implementation of the bisection method that is used to solve this equation illustrates procedural programming in Maple.
- The problem of kinematically controlling the ankle, knee, and hip joints of a simple robot leads through a trigonometric model to a straight-line program that illustrates the basics of C programming.
- Calculating the motion of a block on an inclined ramp in the presence of friction leads into a discussion of the variants of the C `if` statement and to an introduction of programmer-defined functions in C.

- The problem of determining the center coordinates of each rod in a stack of cylindrical rods leads to a modestly complicated model that illustrates the role of functional decomposition in program design, implementation, and testing.
- Determining the position of a beam that is laid against a cubical box leads to a fourth-order polynomial equation, a discussion of Newton's root-finding method, and an exploration of `while` and `do` loops in C.
- Finding the flattened length of a corrugated steel sheet requires evaluating an integral numerically. This leads into a discussion of the rectangular and trapezoidal methods for numerical integration, and from there into a discussion of C `for` loops, the possibility of passing functions as parameters, and the advantages of dividing large programs into multiple files.
- Finding the temperature of an insulated rod as it undergoes heating from its ends leads to a discussion of the finite-element method and of C arrays. Visualizing how the temperature varies with time leads into a discussion of array parameters and file input/output in C. File output is used to create a file containing a Maple command that, when read into Maple, animates the thermal behavior of the rod.

Online Materials

We teach *Engineering Computing* with a combination of two lectures and two laboratories per week. We tend to do everything in lecture at a fairly high level of detail, using a chalkboard to discuss problems, models, and methods and a computer screen projection system to present implementations. In the smaller laboratories, which we conduct in a workstation lab, the students interact with our online instructional material to learn the fine points of the lecture material.

When we first offered *Engineering Computing* in the winter of 1994, we did not use a textbook. Instead, we developed what amounted to a complete online textbook [6] that the students used via a custom hypertext viewer. We also developed Maple worksheets that allowed the students to create animations that illustrated curve fitting and numerical integration techniques.

The Maple animations were a big success, but we didn't get very far into the quarter before the students began requesting hard copies of the online textbook. Over the next two academic years we reworked our material into an online component and a set of printed notes that evolved into our current textbook. The online component consisted of an expanded set of Maple worksheets that illustrated such concepts as the difference between rational and floating-point numbers, roundoff error, interval arithmetic, the semantics of function calls, and Newton's method. We also developed a set of tutorials on C that were viewable with any Web browser [6].

For the 1996–97 academic year, we are developing a third generation of online material. We are developing Java applets to illustrate the concepts previously covered in the Maple worksheets described above as well as to bring to life most of the illustrative problems used in the course. We are also developing a Java applet that will let students interactively trace the execution of all of the C programs used in the course [5]. By doing this in Java, a student with an Internet connection will be able to use our material whether or not they have an implementation of Maple or a C compiler.

Acknowledgments

This work was supported in part by the Department of Energy through the Undergraduate Computational Engineering and Science project and by a Curriculum and Course Development grant (DUE-9455478) from the National Science Foundation. Chris Johnson, Eric Eide, and Ken Parker collaborated in the development of the original course material for *Engineering Computing*; James Clingenpeel, T.J. Giuli, Brandon Harvey, Elizabeth Odekirk, and Joseph Turner are developing Java applets for the current generation of online material.

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] J.L. Zachary, *Introduction to Scientific Programming: Computational Problem Solving Using Maple and C*, TELOS/Springer-Verlag, 1996.

[5] An extensive collection of online material that supports our class is available at the Internet http site `www.cs.utah.edu/~hamlet/IntroSciProg.html`.

[6] Online material that is used for a variety of classes at the University of Utah is available at the Internet http site `www.cs.utah.edu/~hamlet`.