The Gestalt of Scientific Programming: Problem, Model, Method, Implementation, Assessment

Subject areas: non-majors courses, instructional software, scientific programming, introductory programming

Abstract

The process of solving a problem in computational science neither begins nor ends with designing and writing a computer program. The process entails isolating the problem, devising a mathematical model, identifying a computational method, producing an implementation, and assessing the solution. Unfortunately, the introductory programming courses taken by science and engineering students frequently focus only on implementation issues. As a result, such students are often ill-equipped to solve computational problems.

We have created a course and written an accompanying textbook that present an introduction to scientific programming. Both are organized around the process for solving computational science problems sketched above. In this paper we illustrate this approach by describing two representative problems from the course and textbook. Along the way, we present Java applets that were designed to illustrate some of the ideas that underlie the two problems.

Introduction

The process of solving a problem in computational science neither begins nor ends with designing and writing a computer program. Unfortunately, this reality is often overlooked in the introductory programming courses that are taken by science and engineering students. As a result, these students are ill-equipped to apply computing in their other classes and, ultimately, in their jobs.

Solving a problem in computational science entails identifying the problem; posing the problem in terms of a mathematical model; identifying a computational method for solving the model; implementing the computational method on a computer; and assessing the results in the context of the implementation, method, model, and problem. In practice, of course, the boundaries between these five steps can become blurred, and for specific problems one or two of the steps may be more important than the others. Only by having this strategy firmly in mind, however, can science and engineering students fruitfully apply their computing skills to solving real problems.

We have created an introductory course in scientific programming (*Engineering Computing* [6]), written an accompanying textbook (*Introduction to Scientific Programming* [5]), and developed a suite of support software (including Maple worksheets, C programs, HTML-based laboratories, and Java applets [3]). While the course and the textbook present an introduction to programming using Maple and C, both are organized around solving problems from science and engineering. The solution of each problem presented in the course and textbook is organized around the five steps of problem, model, method, implementation, and assessment. Before going any further, we will describe these five steps in slightly more detail.

The first step is to thoroughly explore the problem that is to be solved. We must be sure we know what question we are answering, what theoretical and experimental knowledge we can bring to bear, and what kind of answer we wish to obtain. All of these considerations require knowledge about the science that underlies the problem.

The second step in the problem-solving process is to create a mathematical model of the problem we are solving. We must reduce the problem into one expressed in mathematical terms, and this involves using mathematical expertise to extract the essentials from the underlying physical description of the problem. Only when the problem is expressed mathematically can it be solved computationally.

Once a problem has been described in mathematical terms, the third step is to settle on a computational method for actually calculating the solution. It is often easier to develop a mathematical model than it is to devise a computational strategy for efficiently and accurately computing the answer. Doing this may require exploiting concepts from numerical analysis.

This work was supported in part by a Course and Curriculum Development Grant from the Division of Undergraduate Education of the National Science Foundation.

Once we have a computational method for dealing with a mathematical model, the fourth step is to implement the method on a computer. Doing this, of course, requires knowing something about programming. This step is the exclusive focus of many introductory programming courses, and it is also the primary focus of both *Engineering Computing* and *Introduction to Scientific Programming*. We use Maple to gently introduce the essentials of programming, primarily at the level of expressions and functions, before moving on to a study of statements and procedures in the context of C.

Even after the problem is posed, a model created, a method developed, and an implementation crafted to obtain a solution, we are not done. 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. We must ask ourselves how large an error is likely to be included in the solution. As we identify errors, we must try to fix the problems that caused them and come up with an improved solution. When we finally identify the best solution possible, we must find a good way to communicate it to others.

We believe that it is crucial that the students who take our class and who use our textbook learn about and appreciate the context in which scientific programming is done. We have not lost sight, however, of the fact that both our course and our textbook are first and foremost introductions to *programming*. The challenge that we faced was to prevent the science, mathematics, and numerical analysis issues that arise from the problems, models, and methods from dominating the implementation issues. We adhere to our problem-solving approach as a means of putting computing in its proper context; we do not want our course or our textbook to be about physics, calculus, or numerical analysis.

We have avoided this potential pitfall in two ways. We have carefully chosen the physical problems, the mathematical models, and the computational methods that drive our course so as to not tax the backgrounds of our students. The problems are simplified versions of the ones our students encounter in their introductory physics classes; the models require no mathematical sophistication beyond geometry, algebra, trigonometry, and rudimentary calculus; and the computational methods we use are no more sophisticated than such elementary ideas from numerical analysis as interval arithmetic and numerical root finding. All of this material is well within the reach of typical science and engineering majors.

We have not taken the backgrounds of our students for granted, however. We have developed a collection of Java [1] applets that graphically illustrate the problems, models, and methods that arise in our course and textbook. These applets are simulations of the problems, interactive realizations of the models, and animations of the methods that further our students' understanding of the context in which they

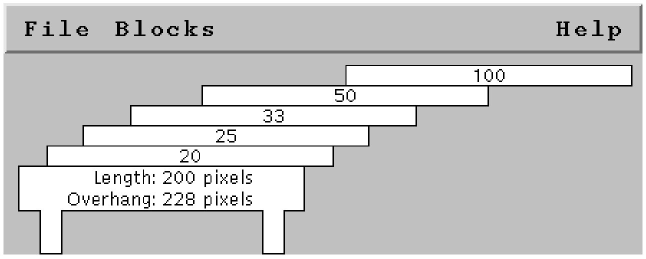

Figure 1: A stack of five blocks balanced on the edge of a table. Each block is 200 pixels long; the right edge of the top block is 228 pixels beyond the edge of the table. From the top, the fraction of a block length by which each successive block extends beyond its support corresponds to the first five even terms of the harmonic series $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{10}\right)$.

are learning to program. We use Java so that the applets can be accessible across the Internet and usable on all platforms.

In this paper we will illustrate our approach by describing a pair of problems from computational science that we have used in our course and that are featured in our textbook. We will organize our discussion of each problem along the lines of the five steps discussed above, and we will present as appropriate the Java applets that are designed to accompany the problems. The first applet that we consider, in Figure 1, illustrates a problem; the second, in Figure 2, a method; and the third, in Figure 3, a model.

Stacking Blocks

Problem. Imagine that we have a supply of identical blocks of uniform density, each 1 foot long. Our job is to arrange a stack of blocks on a table so that the stack extends out beyond the right edge of the table. When building the stack, the bottommost block must be placed so that it extends beyond the right edge, and every other block must be placed so that it extends further right than the block below it. How far out beyond the edge of the table can the right edge of the topmost block extend before the stack collapses? [2]

We have found that students have a hard time understanding both the statement of and the solution to this problem, and an even harder time believing the implications of the solution. Even after we sketch diagrams on the board and do demonstrations with real blocks, these misunderstandings persist. Consequently, we developed the applet pictured in Figure 1 to illustrate the problem and its solution.

Using this applet, students can add blocks to the top or bottom of the stack, remove blocks from the top of the stack, and slide blocks back and forth. The extension of each block beyond its supporting surface is displayed, as is the amount by which the rightmost block extends beyond the edge of the table. The color of a block changes from green to red when it becomes overbalanced. A completely green stack is balanced; a stack containing a red block is not.

Model. The model for this problem is based on the physical concept of center of gravity. Because of the way that the blocks must be arranged, we need only determine centers of gravity in one dimension, along the axis of the table. The center of gravity of one block is at its geometric center; the center of gravity of a stack of blocks is at the average of the centers of gravity of the individual blocks. For a stack to be balanced, the combined center of gravity of each block and all those above it must lie to the left of the right edge of that block's support.

By considering stacks with one, two, three, and four blocks, we are able to argue that the maximum extension possible with n blocks is the sum of the series

$$\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2n}$$

Although the argument that we use suggests an inductive proof, we do not actually prove that our derivation is correct.

The sum that governs maximum extension is composed of the first n even terms of the harmonic series. This sum diverges as n increases, which means that with a sufficient number of blocks we can obtain an arbitrarily large extension. This result is counterintuitive, but to within the limits of the computer display a student can verify it with the applet from Figure 1.

Method. Once we have created a model, it immediately becomes evident that the problem of determining the maximum possible extension for a stack of n blocks can be reduced to the computational problem of summing the first n even terms of the harmonic series. The simplest computational method for doing this, at least for small n, is to add up the terms directly.

Even with so simple a method, there is still the choice of whether to do the sum using inexact floating-point or exact rational number arithmetic. At the point of the course and textbook where we introduce this problem we are studying arithmetic expressions in the context of Maple. Maple supports both floating-point and arbitrary-precision rational numbers. We want our students to understand both, so we end up exploring both methods.

Our students, not surprisingly, do not come into the course understanding the computational nuances of floating-point numbers. We begin our explanation by drawing an analogy between floating-point numbers and the scientific notation with which they are familiar from their math and science classes. We then point out that the difference is that mantissa lengths are fixed and exponent ranges are restricted in a floating-point number system.

As a result, we point out, in any system of floating-point numbers there are maximum and minimum positive (and negative) values. Only those real numbers that lie between the two extremes can be closely approximated with a floating-point number. The distance between the extremes depends on the range of allowable exponents; the quality of the approximations within this range depends on the length of the mantissas. These crucial characteristics are best explained graphically, which is the purpose of the applet that appears in Figure 2.

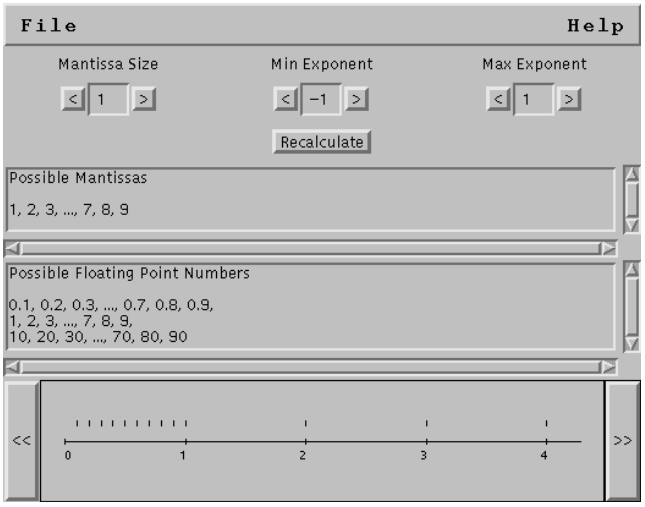

Figure 2: A graphical illustration of the distribution of the positive floating-point numbers between 0 and 4 when there is a one-digit mantissa, a minimum exponent of -1, and a maximum exponent of 1.

Using this applet, a student can design his or her own floating-point number system by specifying a mantissa length, a minimum exponent, and a maximum exponent. The applet then displays the valid mantissas, the valid positive numbers, and a graph of the distribution of the positive numbers. The student can then zoom in and out to develop a better understanding of how floating-point numbers are distributed along the real number line.

The valid values in a rational number system depend on the largest and smallest allowable integers. We also have an applet, similar to the floating-point applet from Figure 2, that allows the student to design a rational number system and to study the distribution of its values.

Implementation. At the point of the course where we discuss the block-stacking problem, we have not yet begun to discuss loops or recursion. As a result, the only means available to the students to sum the even terms in the harmonic series is to write sums directly, as in

```
1./2. + 1./4. + 1./6. + 1./8. + 1./10.;
```

which is a Maple floating-point expression.

Even though they are limited to simple expressions, the students are able to learn much about the computational properties of numbers. They learn that rational number calculations produce exact results, but that floating-point arithmetic is subject to roundoff error. Because Maple allows the programmer to specify the number of digits to include in the mantissas of floating-point numbers, the students can also learn how the magnitude of the roundoff error depends on the size of the mantissa.

As part of the implementation phase of this problem, we also introduce the notion of a library function and provide our students with two. One sums the first n even terms of the harmonic series using rational number arithmetic, the other using floating-point arithmetic. This makes it easy for the students to compute sums for large values of n.

Using the two library functions, the students begin to see how roundoff error can accumulate during long sequences of floating-point operations, and how an entire sum can eventually degrade completely if enough operations are carried out. They see how the time required to sum n terms increases with n, giving them their first taste of complexity analysis. They see how the time required to sum n terms is larger for floating-point numbers with longer mantissas, and how it is even longer for exact rational numbers.

Later in the course, when we have studied iteration in the context of C, the students return to this problem and implement their own versions of the floating-point library function. They can then better appreciate the execution speed advantage of C over Maple, which is substantial on a machine with floating-point hardware.

Assessment. In the assessment phase the students confront the fact that neither the rational nor the floating-point method for summing the harmonic series will work for truly large values for n. Trillions of terms are required to achieve sizeable extensions of the block stack. Based on their experience with the implementation, they appreciate the fact that neither method is sufficiently fast to deal with a trillion terms. Further, they understand that even if the floating-point method could be made fast enough, roundoff error would accumulate to the point where the sum would be useless. Consequently, they have not yet solved the original problem in a computationally effective way.

At this point we reveal that the first n even terms of the harmonic series can be calculated with the formula

$$\frac{1}{2}(\Psi(n+1)+\gamma)$$

where Ψ is a function and γ a constant built into Maple. The students quickly discover that an implementation based on this method is blazingly fast, even for astonishingly large values for n.

Directly summing the terms of the harmonic series may have been the simplest solution, but it was not even close to being the best solution. The final lesson that the students take away from the block-stacking problem, then, is that no matter how clever a programmer is, a bad computational method will lead to a bad program. This reinforces our position that scientific programming cannot be done in isolation from its context.

Population Growth

Problem. According to the Census Bureau, the population of the United States on January 1, 1994, was 259,167,000, and it had increased to 261,638,000 by January 1, 1995. These figures are only approximate, and the Census Bureau even revises them periodically after the fact. For the purposes of this problem, we will amuse ourselves by pretending that they are exact. Under this assumption, what was the population of the United States at any instant of time during 1994?

This is an open-ended problem that has no exact solution, which makes it more representative of the kinds of problems that arise in science and engineering than was the block-stacking problem. Before we can even begin to consider possible models, we must recast the problem into a form that is more amenable to computational attack. Consequently, we will assume that the population grew smoothly during 1994. Since we don't know exactly *how* it grew, we consider several different possible models and take the one that best matches the characteristics of natural population growth.

Model. We attack this population growth problem by making an analogy to something with which students are presumably more familiar: the growth of the balance in an interest-bearing bank account. We develop three different models by considering three different ways in which interest accumulates in bank accounts: simple interest, compound interest, and continuous interest.

When we first taught this material, we were surprised to discover that many of our students did not understand the workings of bank interest. (In retrospect, a better analogy might have been the growth of the balance due on a student loan!) We subsequently constructed the applet pictured in Figure 3 to give our students a way to visualize and experiment with the effects of the three forms of interest.

This applet allows a student to plot the growth of up to three different investments. For each investment the student can select the type of interest, the initial balance, the interest rate, and (for the case of compound interest) the number of compounding intervals per year. The student can also adjust the interval of years over which the investments are tracked.

The students come to appreciate graphically that simple interest is the same as compound interest with one compounding interval, and that continuous interest is the same as compound interest as the number of compounding intervals tends to infinity. They can also see the large difference that even a small interest rate can make over a long period of years.

Method. This is an example of a problem in which the model, method, and implementation are intertwined. We use Maple's symbolic computing capabilities to help develop the three models. We derive the simple interest formula in part by using Maple to do algebraic simplifications; we derive the compound interest formula by applying the simple interest formula to symbolic constants; and we derive the continuous interest formula by using Maple to take the limit of the compound interest formula as the number of compounding intervals goes to infinity.

As we develop each interest model, we try to determine the interest rate that would allow the U.S. population to grow

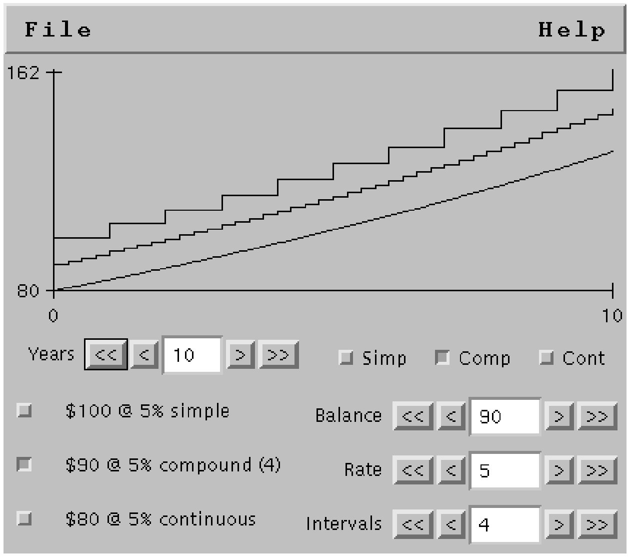

Figure 3: The comparative growth over ten years of $100 invested at 5% simple interest (large steps), $90 invested at 5% interest compounded quarterly (small steps), and $80 invested at 5% continuous interest (smooth).

according to the Census Bureau's data. We do this by using Maple's equation-solving capability to solve equations involving the interest formulas. It turns out that the equation involving compound interest cannot be solved symbolically, which foreshadows the discussion of numerical root finding that the students encounter later in the course.

The primary computational method that we use in this problem is symbol manipulation. This is a method that would be extremely difficult to explore in the context of C, but which is easy to address in the context of Maple.

Implementation. Part of the implementation work in this problem is actually performed during the model development, as discussed above. As we obtain each model, we implement Maple functions that take initial balance, interest rate, and investment duration as parameters and report the future balance of an investment.

Later in the course, after we discuss Maple's plotting capabilities, the students return to this problem and create plots, similar to the ones in Figure 3, that illustrate the growth of investments under different forms of interest.

Assessment. As we develop each implementation, we assess it and its applicability to our original problem: modeling the growth of the United States during 1994. We ultimately conclude that continuous interest is the best model, and we adapt it to develop a function to predict population growth. We also observe, of course, that the model is certainly incorrect, but that is probably the best we can do without better information. The lesson that our students take away is that it is important to be up-front about the limitations of the model on which an implementation is based. Otherwise, unjustified conclusions can easily be drawn.

Conclusions

Our emphasis throughout both *Engineering Computing* and *Introduction to Scientific Programming* is on scientific programming, but we are careful to always introduce programming concepts in the larger context of computational problem solving. This allows our students to learn how to program and at the same time see how to apply their developing programming skills. Our challenge was to identify problems that motivated the need for different programming capabilities while remaining accessible to our students. We did this by choosing our problems carefully and by developing Java applets to illustrate some of the more subtle issues.

Acknowledgments

George Stebbins wrote the block stacker applet and Elizabeth Odekirk wrote the floating-point number applet. T.J. Giuli, Brandon Harvey, Keith Marshall, and Joe Turner have developed other applets that are not presented in this paper.

Chris Johnson, Eric Eide, and Ken Parker collaborated with me on the development of the original versions of the online course notes that eventually led to my textbook.

Tom Marchioro and our fellow members of the Department of Energy's Undergraduate Computational Engineering and Science project [4] first outlined the five-step approach to computational problem solving that is featured in this paper.

References

[1] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison-Wesley Publishing Company, Reading, Mass., 1986.

[2] GARDNER, M. Limits of infinite series. In *Martin Gardner's Sixth Book of Mathematical Games from Scientific American*. Charles Scribner's Sons, New York, 1971, pp. 163–172.

[3] The online resources are available via the Internet at `http://www.cs.utah.edu/~hamlet`.

[4] The UCES home page is available via the Internet at `http://uces.ameslab.gov`.

[5] ZACHARY, J. *Introduction to Scientific Programming: Computational Problem Solving using Maple and C*. TELOS/Springer-Verlag, 1996.

[6] ZACHARY, J., JOHNSON, C., EIDE, E., AND PARKER, K. An entry-level course in computational engineering and science. In *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (Mar. 1995), ACM Press.