Building "Computational Intuition" in Freshman Scientists and Engineers

By Charles F. Van Loan

Undergraduate programs in science and engineering should foster the development of intuition in computational science courses taken during the freshman year, says the author in the first of a two-part series. The second part, to be published in the October issue of **SIAM News**, *will examine the important role of examples in building intuition and communicating the excitement of computational science.*

During the late 1970s I was asked to develop a "Programming for Poets" course for the non-technically oriented undergraduate. One approach would have been to water down the "serious" introductory programming course we offer to engineers and other quantitative types. But just at that time, I had the good fortune to read Donald Knuth's 1974 Turing Award lecture, "Computer Programming as an Art" [1]. Knuth points out that computer science is directly concerned with several of the seven original liberal arts,[1] an observation that started me thinking the right way about what we could achieve in "Poets". I became convinced that at the introductory level we should do more than just teach skills and I became intrigued by the following question: Given the importance of computer science, what should liberal arts students know about it?

Thinking about this question turned a potentially low-level, mundane teaching assignment into an exciting educational adventure, forcing me to think about what my colleagues and I do for a living and how it fits into the big picture.

A course with a double agenda unfolded [2]. Yes, there was a practical-skills component. The students learned a programming language, refined some of their quantitative abilities, and solved a few interesting problems. But, at another level, through history and timely examples, they came to appreciate the notion of an algorithm and the culture of computing.

Liberal education is all about the different ways that human beings can express what they know, and algorithms are right up there with painting, the novel, and other revered forms of expression. No single form of expression is by definition any better than the others. The mere possession of a computer simulation doesn't imply that a university-trained ecologist knows more about acid rain than a Native American whose livelihood is threatened by the phenomenon. Programming for Poets gave students who came from every imaginable discipline – including history, dance, and landscape architecture – the critical skills necessary to deal effectively with the self-proclaimed computer experts.

More recently I have been thinking about how we teach computing to technically oriented students. The increasingly high profile of "computational science" in the research community

[1] Logic, grammar, geometry, arithmetic, rhetoric, music, and astronomy.

led me to dwell upon a second classroom question: Given the importance of computational science, what part should it play in the freshman scientist/engineer's experience?

Thinking about the second question again led to a course with a double agenda, one that is best described in terms of the twin ideals set forth by the co-founders of Cornell University, Ezra Cornell and Andrew D. White. Ezra Cornell espoused the ideal of practical education. He wanted a university where the sons and daughters of farmers could acquire the scientific skills necessary to improve the state of agriculture. When this objective is mapped into the 21st century, it means that computer science should provide undergraduate instruction in practical programming that enables students to return to their "computational farms" and make useful contributions. This defines the first agenda, which is to teach programming, i.e., computer problem solving.

Andrew White was the 19th-century embodiment of the ideal of liberal education and the perfect counterpart to the practically minded Cornell. To White, it was not enough for the fledgling university to provide skills-based instruction in the technical arts; history and literature were as critical to the uplifting of rural life as genetics and chemistry. It is unfortunate that these liberal education ideals are not appreciated with the same vigor today. This is partly due to the pressure to specialize, a pressure that is crowding the role that mathematics and computer science should play in the education of the computational scientist. That is why we should teach introductory programming with a second agenda: to trigger a life-long interest in how these two subjects fit into the technical culture.

What Is Computational Science?

In one way, "computational science" is the ultimate triumph of terminology, coupling the respectability of science with whatever is computational. But instead of regarding the term as a marketing ploy, think of computational science as a point of view. Science can be seen as a triangle with theory, experiment, and computation at its vertices (see Figure 1). Each vertex represents a style of research and provides a window through which we can look at science in the large. The vibrancy of what we see inside the triangle depends upon the ideas that flow around the edges of the triangle. A good theory couched in the language of mathematics may be realized in the form of a computer program, perhaps just to affirm its correctness. Running the program results in a simulation that may suggest a physical experiment. The experiment in turn may reveal a missed parameter in the underlying mathematical model, and around we go again.

But interesting ideas can flow in the other direction as well. A physical experiment may be restricted in scope for reasons of budget or safety, so the scene shifts to computer simulation. The act of writing the program to perform the simulation will most likely have a clarifying influence, prompting some new mathematical pursuit. Innovative models are discovered, leading to a modification of the initial set of experiments, and so forth.

A parable will serve to clarify these interactions. Three scientists show up at the Leaning

Tower of Pisa. E is experimental, T is theoretical, and C is computational. E arrives with a 16-pound shot-put, climbs the tower, releases the weight, and times the free fall. A paper is written and appears in a journal of experimental physics. T thinks about the experiment and then develops a mathematical model that captures the essence of the phenomenon. A paper is written and appears in a journal of theoretical physics. E looks over the equations and wonders what would happen if w were set to 1600 pounds. For practical reasons, the obvious experiment cannot be performed. However, C responds to the lament of E by transforming T's mathematics into a program. The program is run with $w = 1600$, and it is discovered that the hundredfold increase in weight does not have much of an effect upon the time of free fall. A paper is written and appears in a journal of computational physics. Moral: The interactions between mathematics, physical experimentation, and computer simulation are crucial to the scientific enterprise.

If we fail to communicate the dynamics illustrated by the parable, then the technical student will graduate with the anemic, weakly connected view of how things work in science and engineering shown in Figure 2. The time to start building the required appreciation is during the freshman year, and introductory programming is an appropriate vehicle. Think of the freshman computing experience as a trip along Route 66 from Chicago to Los Angeles, with computational science being the view outside the window of the car. If the trip is a success, then the freshman arriving in L.A. cannot stop talking about the landscape: "I got my kicks on Route 66 and want to revisit all those great sights." If it is a failure, then the student arrives in L.A. and cannot stop complaining about the car problems sustained along the way: "I got kicked on Route 66 and never want to go back". We must avoid the latter situation, where the syntactic side of computing and the anomalies of the system dominate the student's experience at the expense of what is truly important.

The key is to paint the landscape in colors so brilliant that the student will be absolutely entranced with the view. Is this possible with freshman-level mathematical maturity, typically defined by a semester or two of calculus? If our ambition is to build "computational intuition", the answer is yes.

The Five Computational Senses

Much has been written by philosophers and psychologists on the subject of intuition, and just about every scientist I know has something to say about it. Instead of deliberating on the topic, let us just assume that intuition is a sense of direction, essentially no different from the sense of direction that enables you to find your way around an old childhood neighborhood without a map. *The key is that you've been there before.*

If intuition is a sense of direction, then computational intuition is a sense of computational direction. Those who have it can find their way around the computational side of engineering and science. Success requires five keen senses, so we ask the freshmen:

1. **Do you have eyes for the geometric?** The ability to visualize is very important to

the computational scientist. Of course computer graphics plays a tremendous role here, but the visualization tools that it offers do not obviate the need to reason in geometric terms. The student must be totally at home with sines and cosines, polygons and polyhedra, metrics and proximity, and so forth.

2. Do you have ears that can hear the "combinatoric explosion"? Many design and optimization problems involve huge search spaces with an exponential number of possibilities. It is important to be able to anticipate this complexity and to have the wherewithal to handle it with intelligent heuristics.

3. Do you have a taste for the random? Many important processes have a random component. Having a sense of probability and the ability to gather and interpret statistics with the computer is vital.

4. Do you have a nose for dimension? Simulation is much more computationally intensive in three dimensions than in two dimensions – a hard fact of life that is staring many computational scientists right in the face. An accurate impression of how computers assist in the understanding of the physical world requires an appreciation of this point. Moreover, being able to think at the array level is essential for effective, high-performance computing.

5. Do you have a touch for what is finite, inexact, and approximate? Rounding errors attend floating-point arithmetic, terminal screens are granular, analytic derivatives are approximated with divided differences, a polynomial is used in lieu of the sine function, and the data acquired in a lab are correct to only three significant digits. Life in computational science is like this, and you just can't fall apart in the presence of such uncertainty. A steady balance is required along the fence that separates the continuous from the discrete.

We teach our children to use their physical senses, and we can teach our freshman to use their computational senses. In both arenas, experience with examples is critical.

Intuition and Rigor

Computational intuition is built by hanging around the right set of examples, a process that should begin during the freshman year. The first year of college is not the time to stress mathematical rigor or formal program correctness proofs. Interest in computational science is jeopardized if these things are pushed before the student is ready. However, the freshman year can be used to set the stage for the precision that mathematics has to offer if the connection between intuition and formality is understood:

Formalism First = Rigor Mortis
Intuition First = Rigor's Mortise

Experience must precede abstraction. Formal methods that support the development and verification of programs are more easily learned by students who have spent a semester programming at a more informal level. Likewise, a mathematical concept that supports an interesting computation is more easily mastered if the student has quite literally played with the concept beforehand on the computer.

Critics of this philosophy tend to equate intuitive problem solving with ad hoc problem solving. This is unfortunate because it denigrates the role of intuition. There is a lesson to be learned here from the school of phonetic spelling. Many first-grade teachers find it easier to foster creative writing skills by accepting phonetic spelling. The "axioms" of correct spelling are gradually enforced in a way that does not stifle the child's originality. Correct spelling *is* important, but it is co-developed with other writing skills, not in advance of them. Likewise, the student's facility with mathematical rigor must be co-developed with other computational skills.

To illustrate these points further, I offer two examples. First, consider the path from freshman computing to complexity theory, a branch of computer science that deals rigorously with what makes certain computational problems hard to solve. A beginning student may start by benchmarking (timing) a number of programs that all solve the same problem. The sorting problem is a favorite because there are so many different methods from which to choose. After benchmarking, say, the methods of bubble sort and merge sort over a range of input lengths, the student discovers that they behave differently. The intuition acquired from the computational experience sets the stage for a discussion of running-time classifications. Merge sort is an $O(n \log n)$ algorithm, while bubble sort is $O(n^2)$. Here, as n (the number of items to be sorted) increases, the ratio of the bubble sort running time to the merge sort running time grows like $n/\log n$. By observing this experimentally, the student develops intuition about running time.

Later, in follow-up courses, the student is brought into contact with other running-time classifications: $O(\log n)$, $O(n)$, $O(n^2)$, and so forth. This prompts a curiosity about lower bounds for running time and naturally leads to questions about inherent problem-solving difficulties. In this way the student is brought gracefully to the house of complexity theory and is ready to enter.

Another instructive example involves the path from elementary array manipulation to advanced scientific computation. For most students, the presentation of arrays in introductory programming is the first time they see n things portrayed as a single object. Matrix-vector multiplication is a classic example of an array computation that can and should be introduced at this level. However, the words "matrix" and "vector" cannot be mentioned because the student has yet to study linear algebra. But this situation doesn't last long, because, in one or two semesters, the student is engaged in a formal presentation of linear algebra by the mathematicians on campus. At this point, the prior experience in programming with arrays has built a facility with subscripts and an intuition about array-level operations that free the student to think about such central mathematical concepts as basis, independence, and rank.

The matrix/vector intuition acquired "over in math" then sets the stage for a return visit to elementary numerical analysis "over in CS". But the presentation of Gaussian elimination and its analysis in CS requires the use of matrix norms, a topic not fully covered in that first linear algebra course. So back over to math the student goes, perhaps stepping into a first course in functional analysis. With that expertise the student is ready for yet another return visit to CS and a course in the numerical solution of partial differential equations. And so it goes.

In this particular shuttle view of CS/math interactions, the student is more than just a Ping-Pong ball going back and forth between the two departments. The intuition acquired in one course sets the stage for the rigors of the next. Our job is to ensure a proper tread-to-riser ratio as the student climbs this curriculum staircase. Intuition is the tread, the firm, nonslip surface that sets the stage for the next level of abstraction. Rigor is the riser, and it should be negotiated only when the firmly planted foot says "ready".

Acknowledgments

The author would like to thank David Gries and Steven Vavasis for their comments while this paper was being written.

References

[1] In *Communications of the ACM*, 17 (1974), 667-73.

[2] Charles Van Loan, *Computer science and the liberal arts student*, Educational Forum, 45(1980), 29-42.

Charles F. Van Loan is a professor of computer science at Cornell University.

Using Examples to Build Computational Intuition

By Charles F. Van Loan

My knowledge of Latin is at the motto level:

1. *exempli gratis* (by way of example);
2. *E pluribus unum* (out of many, one);
3. *caveat emptor* (let the buyer beware);
4. *semper fidelis* (always faithful).

In this article, I want to use my four-phrase Latin vocabulary to say something about building computational intuition in freshman scientists and engineers. In the last issue of **SIAM News**, I wrote about the five computational senses whose development we should foster at the freshman level. Now I want to argue that this is an achievable goal if we teach freshman computer programming *exempli gratis*.

In teaching, writing, and research, there is no greater clarifier than a well-chosen example. Above and beyond clarification, however, examples generate interest. There seems to be a positive correlation between student enthusiasm and the use of examples. In a single lecture, examples illustrate and enliven when sprinkled among the theorems and abstractions. Throughout a semester, detailed, aptly positioned examples can make a particular course memorable. During a four-year undergraduate experience dominated by the weekly problem set, it is the "project course", with its focus on a single, large example, that often best prepares the student for the outside world. No matter what the educational time scale, examples play an important role. They engage the student.

The Frisking of Examples

There is good reason to invite examples into the computational science classroom. But just because an example shows up at the door doesn't mean that you should let it enter. *Caveat emptor.* When freshman are involved, all examples need to be frisked and interrogated with the following questions:
(a) Do you complement or uplift the student's mathematical expertise?
(b) Do you symbolize what goes on in computational science?
(c) Do you engender a burning curiosity for mathematics and/or computer science?
(d) Do you squelch naive views about computer problem solving?

Let us look at some examples that, when scrutinized in this fashion, get by the security guards.

Reinforcing Examples

Consider the problem of determining whether a real number x is inside an interval $[a, b]$. One way to do this is to see if x is to the right of a and to the left of b:

```
In:=(a<=x) and (x<=b)
```

Another way is to make sure that x is neither to the left of a nor to the right of b:

```
In:=not ((x<a) or (b<x))
```

A discussion of these two alternatives amounts to a "90% proof" of De Morgan's law,

$$A \wedge B \equiv \neg(\neg A \vee \neg B),$$

a well-known theorem in propositional logic that relates the "and" and "or" operations. The formal proof of this law may be presented in a discrete mathematics course, but until then the student is well served by this proof by example.

Just as a computing problem can set the stage for future mathematical work, it can also serve as an occasion to use a bit of mathematics already learned. Consider this illustration of nested conditionals:

```
if a<=b then
    if a<=c then
       min:=a
    else
       min:=c
else
    if b<=c then
       min:=b
    else
       min:=c
```

The illustration happens to assign the smallest of `a`, `b`, and `c` to `min`, and, as such, it is a fine example of nested conditionals. But if the student knows first-semester calculus, to use this illustration is to squander an opportunity: A better illustration is to have the student compute the minimum value of a quadratic $x^2 + bx + c$ on the interval $[L, R]$:

```
L_Slope:=2*L+b;
R_Slope:=2*R+b;
if L_Slope>=0 then
    min:=L*L+b*L+c
else
    if R_Slope<=0 then
       min:=R*R+b*R+c
    else
       min:=c-b*b/4
```

This, too, illustrates nested conditionals, but it also serves to reinforce the student's calculus expertise. It makes use of the fact that the minimum of a continuously differentiable function on an interval occurs either at the endpoints or at a point where the derivative is zero.

Symbolic Examples

The symbolic example is a snapshot that illustrates some particular concern or worry of the computational scientist. Out of many such snapshots should emerge one view of the discipline. *E pluribus unum.*

For example, the Earth's surface area in square kilometers can be computed from the formula where $A = 4\pi r^2$ is the radius in kilometers:

```
c:=4.0*3.14159;
r:=6378;
A:=c*r*r
```

This looks harmless enough as a vehicle for talking about expressions and assignment. If we poke around, however, we find that there is a lot more to the example than meets the eye. Is the Earth a sphere or an oblate spheroid? (A model- error question.) Is the radius exactly 6378 kilometers? (A data-error question.) Does π equal 3.14159? (A mathematical error question.) Does the computer multiply 4 and 3.14159 exactly to get 12.56636? (A roundoff-error question.) All kinds of error beset the computational scientist, and this simple example is symbolic of these difficulties.

When are three points colinear? This seemingly simple problem can be used to introduce a basic programming skill such as the writing of Boolean-valued functions. But in a world filled with fuzzy data and inexact arithmetic, we quickly discover that life isn't so Boolean after all. To handle colinearity problems in practice, we need to develop suitable measures of near-colinearity, a surprisingly complicated problem that is guaranteed to make any curious student reflect upon the descriptive power of mathematics. Accepting that nothing is simple whenever computers are involved is the first step on the road to becoming an enlightened computational scientist. The colinearity problem dramatizes this point.

What is the design process, and how do computers fit in? "Industrial-strength" design problems are typically very complex, often involving thousands of parameters and requiring sophisticated computer environments for their solution. Still, the key ideas can be communicated with a small symbolic example. Consider the design of a mountain bike that requires the selection of three pedal sprockets and seven wheel sprockets. The problem is to choose the ten sprockets so that the 21 gear ratios are uniformly spread across the real interval [1,4] subject to constraints. One constraint, imposed by the marketing experts, could be that the lowest and highest ratios should be 1 and 4, respectively. Another constraint, imposed by the wholesale buyer, could be that the sprockets come in a limited number of sizes.

Although this example is small, it is rich enough to support the discussion of key issues, such as the notion of parameter-space dimension, the combinatoric explosion of possibilities, the choice of objective function, the role of heuristics, and the implications of constraints. A simple interactive environment can be set up to illustrate how the computer can accelerate the search for the optimal design. The 10-parameter mountain bike design problem is symbolic of all design problems.

Seductive Examples

No matter what course of study the freshman ultimately pursues, our goal should be that he or she maintain a faith in mathematics and computer science. *Semper fidelis.* Well-chosen, seductive examples have the effect of making the student hungry for computer science and mathematics by pointing to their role in the modern conduct of science and engineering.

Consider the problem of finding the smallest ellipse (in area) that encloses n given points in the plane. A simple interactive environment can be set up that facilitates the "clicking in" of trial ellipses. The student may in fact help build the environment as a programming exercise by writing, for example, a function that can test for inclusion of the point set. However, after enough experimentation, the student will begin to wonder if the search for the optimum ellipse can be automated. The tumblers will fall into place, and the example will have unlocked an interest in computational geometry.

Sobering Examples

Computational science has its share of belligerent know-it-all types who bully their way around by underplaying the realities of computing. One such reality is the chasm between formula and production software. The computer belligerent thinks that the mere transcription into code of a math book formula like

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

is all that is required to produce usable software. But a freshman-level excursion into Newton's method will expose the fallacies of this wishful thinking and have a sobering effect.

Another type of computational belligerence has to do with the finiteness of computer arithmetic. Lack of understanding in this area sets the stage for an exaggerated view of computational accuracy and makes possible tragedies like the Patriot missile disaster of the Gulf War. But the freshman who hangs around the right set of sobering examples will not be prone to pitfalls like this.

A final kind of sobering example has a counterintuitive, less-is-more theme. If a spherical Earth ($r = 6378$ km) is glazed with a one-micron layer of gold, what would the increase in surface area be? The exact increase, ΔA, is given by:

$$\Delta A \approx 4\pi((r + \Delta r)^2 - r^2)$$

and gives rise to the computation

```
r:=6378;
Delta_r:=0.000000001;
Delta_A:=4*pi*(sqr(r+Delta_r) - sqr(r)).
```

An approximation to the increase (derived using the calculus) is given by:

$$\Delta A \approx 8\pi r \Delta r$$

and leads to:

```
r:=6378;
Delta_r:=0.000000001;
Delta_A:=8*pi*r*Delta_r.
```

On many computers, however, the exact-formula method gives zero while the approximate-formula method gives an answer that is much closer to the true value. Exact, closed-form recipes are a breeding ground for unjustified confidence and set the stage for computational belligerence.

The nice thing about sobering examples is how easy they are to discover. Almost all freshman-level concepts in mathematics have subtleties when you play with them on the computer. The right combination of sobering examples can send a powerful message to the student.

Conclusion

The examples used when we teach freshman computing act as a set of "basis vectors"; everything that the student learns is in their span. Our job as professors is to choose that basis carefully to ensure the development of computational intuition.

Charles F. Van Loan is a professor of computer science at Cornell University.