Balancing Symbolic and Computational Thinking: Reinforcing Engineering Math Via Programming

Rahul Simha
Department of Computer Science
The George Washington University
Washington DC 20052
`simha @ gwu.edu`

Abstract

This position paper will describe an approach that uses programming to teaching core concepts in mathematics useful to engineering, such as calculus, probability, and linear algebra. The approach is based on the premise that programming is a different and often effective way to *learn*, both as a form of active learning in class as well as a platform for encouraging explorative thinking via tinkering with code. Because the representation of quantitative concepts in code is markedly different, students get to see very concretely many of the details that are often hidden in the symbolic approach. Combining the two modes, symbolic and computational, and balancing them is the central goal of our approach. The paper and accompanying ASEE talk will describe various topics in continuous mathematics (calculus, probability, linear algebra) taught with the combined symbolic-computational approach, including lessons learned and resources available for others.

1.0 Introduction

One way to prompt a long list of complaints is to ask any engineering faculty about the math skills of their undergraduates. The more thoughtful complaints often center around poor conceptual understanding and an overreliance on formula-driven plug-and-chug problem solving. Such concerns about the degree to which students really understand mathematics are shared by researchers in mathematics education. Schonfeld (2009) summarizes the different issues and schools of thought on mathematical “sensemaking,” tracing the arc from early heuristic approaches to problem solving (Polya, 1945) to modern studies that now drive the growing body of research on mathematical thinking. This paper seeks to contribute a new approach to strengthening students’ conceptual understanding of mathematics: using computation to reinforce intuition and sensemaking in mathematics. The details, however, matter. The mere use of sophisticated computational tools such as Matlab or Mathematica, we argue, is no better than its widely criticized counterpart in traditional mathematics instruction: teaching students formulas and how to apply them. The main purpose of this paper, therefore, is to also contrast the proposed approach from the current uses of computation in engineering curricula.

One important practical rationale for an alternative approach is to reach students who learn differently, and who merely “survived” the traditional math courses. In the traditional symbolic approach, a student’s mental model of quantitative concepts centers on the algebraic relationships between key symbols representing problem parameters. For many students not fully facile with symbolic manipulation and interpretation, conceptual understanding is often replaced with merely “cranking out” the math, or plugging values into a recipe-driven problem-solving approach. In contrast, students who are comfortable enough with programming can see the same mathematical concepts through a different lens, and one in which they see the mathematics “in action” through the code they write. Because the representation in code is constructive and algorithmic, students get to see very concretely many of the details that are often hidden in the symbolic approach. Combining the two modes, symbolic and computational, and balancing them is the central goal of our approach.

2.0 Some illustrative examples

Since our focus is on post-secondary mathematics, we begin with one example each from calculus and probability.

Example problem #1: evaluate $\int_0^{0.5} \sin(2\pi t)\, dt$	

Symbolic approach:
(1) Look up the integral:
$\frac{-\cos(2\pi t)}{2\pi}$
(2) Plug values:
$\frac{-\cos(\pi)}{2\pi} - \frac{-\cos(0)}{2\pi}$
(3) Write as symbol or use calculator:
$\frac{1}{\pi} = 0.31831$
	Computational approach: (Java)
	(1) Write code
	`double delT = 0.05;`
	`double sum = 0;`
	`for (double t=0; t<=0.5; t+=delT) {`
	` sum += Math.sin(2*Math.PI*t) * delT;`
	`}`
	`// Print sum (code not shown)`
	(2) Edit, compile, execute, debug, as needed
	(3) Experiment with delT

Example problem #2: An unbiased coin is flipped 10 times. What is the probability of observing 7 or more heads?	

Symbolic approach:
(1) Look up the formula:
$P[X \geq i] = \sum_{k=i}^{n} \binom{n}{k} p^k (1-p)^{n-k}$
(2) Plug values:
$P[X \geq 7] = \sum_{k=7}^{10} \binom{10}{k} 0.5^k (1-0.5)^{10-k}$
(3) Use calculator: 0.172…
	Computational approach: (only step (1) shown)
	`for (int trial=0; trial<numTrials;`
	`trial++){`
	`    int numHeads = 0;`
	`    for (int i=1; i<=N; i++) {`
	`         if (uniform() < p) {`
	`              numHeads ++;`
	`         }`
	`    }`
	`    if (numHeads >= k) {`
	`         numSuccesses ++;`
	`    }`
	`}`
	`// Print numSuccesses/numTrials`

What do the examples tell us about the two approaches? Cosmetically, it's obvious that they look very different and so, at the very least, the instructor could draw connections between the two, for example by comparing the inner for-loop with the summation in the second problem.

Next, let's contrast the *actions* of the student in each approach. In the symbolic approach to Example 1, the student is likely to either look up the integral or use an online resource like Wolfram-Alpha (which directly provides both symbolic and numerical answers). Then, the values are plugged in and the result is calculated. Similarly, in the second case, the student finds a similar problem already solved (almost every probability textbook has an example similar to the one above), and plugs in the parameters from the problem description. In contrast, in the computational approach, the student writes code, which may not be correct code in the first attempt, and then goes through a cycle of execution and debugging.

Conceptually, however, the differences are striking. In the integration example, the student who looks up the result is far removed conceptually from the concept of integration. However, the student punching out the code, is at least confronted with the meaning of integration via the addition of areas. Similarly, the code in Example 2 clearly outlines the experiment: it's obvious that there two types of repetitions, one involving the experiment, and the other within the experiment (of multiple coin flips). It is very easy, on the other hand, for student doing formula-lookup or problem-pattern matching in the symbolic approach to completely miss the point.

The differences in mistakes and false starts are equally illustrative. In the symbolic approach for Example 1, possible mistakes include wrong lookup, improper substitution of values, incorrect sign, and incorrect calculation. Which of these really help with understanding the concept of integration? We argue that none of them have anything to do with the concept. All of them fall in the category of "this is how it's done; I'd better get the steps right." Similarly, with the probability example, errors include incorrect parameter substitution, using "greater-than" instead of "greater-than-or-equal", and incorrect calculation of the final answer. Again, we point out that none of these ultimately relate to how the Binomial distribution works.

On the other hand, consider the potential mistakes made in the computational approach and what the student might learn. First, we'll assume basic proficiency in computing, well past the issues with the programming language, compilation and syntactic errors. Similarly, even thought the student may err in implementing the function (say, by writing Math.sin(t) in the first problem), we assume that finding such a mistake is easy and falls in the realm of purely programming errors. The more interesting type of error is a conceptual error. For example, a delT value that's too large will produce a result that's slightly off, which when the student experiments with will help them understand why the summation is an approximation and gets better in the limit. Computational mistakes in the second example are likely to be more helpful to the student. For example, if the loop order were changed, the student would have to confront a fundamental misunderstanding of what an experiment is.

3.0 Caveat #1: about (too) powerful tools

It is critical to note that our proposed approach has the student writing code at the lowest possible level, directly in a programming language like C or Java, as opposed to merely calling some function in a scientific package like Scilab. We argue that replacing the detailed code with a single call to a external function that does-it-all more or less voids the advantages we've outlined in the prior section. All the important detail would be buried "under the hood" leaving the student with few opportunities to grapple with conceptual detail. Furthermore, because the powerful tools make it too easy to solve simple problems, the instructor is forced to devise artificially complex problems to challenge the student, which then moves away from basic math concepts to computational concepts or mere programming skill development.

4.0 Features of the computational approach

We discuss a few features that we believe make best use of our proposed approach:

- ***Promotes exploration, sometimes to more advanced ideas.*** The computational approach allows student to experiment quite easily with minor edits to the code. For example, we use simple Euler integration to have students explore applications with non-linear ODEs (a rather large class of applications) that would be hopelessly out of reach symbolically. All of these programs are about 10-20 lines long, very easy to implement, and let the students directly interact with applications from population dynamics to molecular reactions. Similarly, students can solve harder physics problems (such as the brachistochrone) or engineering control problems (robot dynamics) that would quite difficult in an undergraduate course.
- ***Enables self-discovery***. We use the computational approach to let students discover important results. For example, consider how the all-important central limit theorem is taught in a standard symbolic-based undergraduate course: it is simply stated as a result because, after all, how could students possibly sit through a detailed proof in class, much less derive the central limit theorem on their own? Yet, computationally, it is very easy for students to write code to form visual histograms based on generated data, then scale the histograms by the measured standard deviation, and watch the central limit theorem emerge in front of their eyes. This is in fact how we teach many core concepts in the computational approach: to have students, with appropriate scaffolding, discover them on their own.
- ***Helps students build an internal model***. Because the approach is directly constructivist, students build an internal, if computational, model of mathematical concepts. The ideal way to build on this model is to also have the students grapple with the symbolic equivalent and connect the two models. We find this to be the most appealing way to get at deeper concepts. For example, in teaching linear algebra, students must work through traditional pen-and-paper examples of Gaussian elimination, while implementing solutions in code, all of which enables them to very clearly see how row reduction works.

- ***Easily aligns with active learning***. The fact that small edits to code are simple and quick, allows the instructor to build learning activities in class. The two courses we teach generally intersperse learning activities every 5-10 minutes.
- ***Facilitates the use of real data***. One advantage of the computational approach is that students can work with real applications and data. For example, in demonstrating the use of the singular value decomposition to text analysis, we use actual text (news articles). Contrast this with the small, somewhat artificial examples in a textbook example. Students are generally more motivated if they connect the concepts to real applications.

5.0 Caveat #2: about the balance between symbolic and computational

We wish to be clear that, although the computational approach has several advantages, it should not replace the symbolic approach. In our courses, we do not shy away from proofs and the full use of traditional mathematical symbolism. We believe that students benefit from seeing both side by side and grappling with the errors, pitfalls, and misconceptions in each. Thus, homeworks and assignments often consist of a pen-and-paper part (traditional), and a programming part. Similarly, many active-learning exercises in class ask students to explain why something is true. The latter is often possible only through the symbolic approach.

6.0 Curricular options

How can a computational approach fit into an already-full engineering curriculum? We do not address the "what to remove" issue here, but instead focus on course modules and where they may fit. We have developed one course, consisting of two bodies of material: (1) calculus and differential equations; (2) probability. And we are currently developing a full course on (3) linear algebra. In more detail:

- **CS-4341: Continuous Algorithms**. Overview of structures in continuous mathematics from a computational viewpoint. Main topics include simulation, computational modeling, machine learning, neural networks, text classification, statistical language processing, robot control algorithms.
- **CS-4342: A Computational Introduction to Linear Algebra**. Linear algebra applied to computational problems in computer science and engineering. Topics include points, vectors, matrices, and their programming abstractions; 3D transformations, pose and viewpoint estimation; linear equations; algorithms for matrix decompositions, dimension reduction, computation with large matrices, under- and over-determined systems; applications to big data, computer vision, text processing.

Computational prerequisites. For both of these courses, we assume the level of programming proficiency of a strong student coming out of a two-course programming sequence. The actual programming challenge in our two courses is in fact mild compared to a programming-intensive computer science course. However, the proficiency matters because we don't want students hung up on minor programming issues that would distract from developing mathematical concepts. Such a student would get frustrated with programming, and will not be able to keep up with the in-class active-learning exercises. At the moment, both courses use Java as the programming language. However, because the advanced features of Java do not play a role, it is relatively straightforward to rewrite the code in other languages.

Mathematical prerequisites. For the Continuous Algorithms course, we require the first calculus course. This is mainly so that we can refer to differentiation and integration. It is possibly to do without calculus but that would mean going at a slower pace to accommodate definitions and some practice with these concepts. For Linear Algebra, high school algebra is sufficient but some mathematical maturity beyond that is helpful.

Where do such courses fit into a curriculum? Clearly, students need to first learn programming through the standard two-course programming sequence, and at the very least the programming language that will be used in these courses. Second, they would need first-semester calculus for the Continuous Algorithms course. It helps if the students have had additional mathematics, because they are then better able to use computation to reinforce concepts.

Should students take the traditional equivalents first? We have argued against replacing the traditional with the computational. Because repetition is so central to learning, we believe it is best if students take full courses in both approaches, first taking the traditional and then following that up with the computational. Because our courses combine symbolic and computational, it serves both to reinforce concepts learned earlier while providing a fresh perspective.

7.0 Lessons Learned

With the linear algebra course under development and three offerings of the Continuous Algorithms course completed, what lessons have been learned? One of the most important is that students really need to see basic concepts again and again. We have repeatedly seen countless examples of cases where a fundamental concept was entirely misunderstood in an earlier course and which only came to light because the same concept was explored and revisited in our courses. Modern curricula are unfortunately structured to pack as much different material as possible, leaving almost no room for repetition.

Another important lesson is that students are quite different in how they learn. There is a certain type of student, including the author, that learns quite efficiently from hands-on implementation. Over and over, students have said "I finally understood [X] when I implemented the code." Somehow, grappling with the detail at the level of code helps with building understanding. We don't yet understand why, or what the epistemological implications are, but it is clear from student feedback that they value the opportunity to learn in this manner.

Another way in which students markedly differ is in their preparation coming into college, and college-level courses. For some, algebraic and therefore symbolic manipulation is a sheer challenge. For others, the connection with geometry is tenuous. For yet others, the skill of proof is far from developed. Because programming can be learned after K-12, it can sometimes even the playing field and let students with low math-self-esteem feel motivated and able to learn mathematical concepts.

Finally, students have also commented positively about the connection with applications. The "application" examples in traditional math courses are often toy examples that leave the students unmotivated, or are not fleshed out in sufficient detail.

Resources

Code, course material, and active learning exercises in Java are available for the two courses described above: http://www.seas.gwu.edu/~simhaweb/

Acknowledgements

The author would like to acknowledge the support from NSF grant 1347516.

REFERENCES

G. Polya (1945). How to solve it. Princeton: Princeton University Press.

A.H.Schoenfeld (1992). Learning to think mathematically: Problem solving, metacognition, and sense-making in mathematics. In D. Grouws (Ed.), Handbook for Research on Mathematics Teaching and Learning (pp. 334-370). New York: MacMillan.