# Teaching Programming for Mathematical Scientists


**Jack Betteridge, Eunice Y. S. Chan, Robert M. Corless, James H. Davenport, and James Grant**

## 1 Background

This volume is part of the fast-growing literature in a relatively new field—being only about 30 years old—namely Artificial Intelligence for Education (AIEd). The survey (Luckin et al., 2016) gives in lay language a concise overview of the field and advocates for its ambitious goals. For a well-researched discussion of an opposing view and of the limitations of Artificial Intelligence (AI), see Broussard (2018). This chapter is concerned with AI in mathematics education in two senses: first, symbolic computation tools were themselves amongst the earliest and most successful pieces of AI to arise out of the original MIT labs already in the sixties,[1] and have had a significant impact on mathematics education. This impact is still changing the field of mathematics education, especially as the tools evolve (Kovács et al., 2017).

[1] For example, (Slagle, 1963), which took a "Good Old-Fashioned Artificial Intelligence (GOFAI)" approach, and concluded "The solution of a symbolic integration problem by a commercially available computer is far cheaper and faster than by man". Of course, this was from the era when people still believed in GOFAI. We are grappling with different problems today, using much more powerful tools. Yet some important things can be learned by looking at the effects of the simpler and older tools. The riposte to Slagle (1963) was the development of Computer Algebra (Davenport, 2018) as a separate discipline.

J. Betteridge · James H. Davenport · J. Grant
The University of Bath, Bath, England
e-mail: jdb55@bath.ac.uk

J. Grant
e-mail: rjg20@bath.ac.uk

E. Y. S. Chan · R. M. Corless (✉)
Western University, London, Canada
e-mail: rcorless@uwo.ca

E. Y. S. Chan
e-mail: echan295@uwo.ca



P. R. Richard et al. (eds.), *Mathematics Education in the Age of Artificial Intelligence*,
Mathematics Education in the Digital Era 17,
https://doi.org/10.1007/978-3-030-86909-0_12

Second, and we believe more important, the existence of these tools, and similarly the existence of other AI tools, has profoundly changed the affordances of mathematics and therefore *should change the content of mathematics courses, not just their presentation methods* (Corless, 2004a). That last paper introduced the phrase "Computer-Mediated Thinking", by which was meant an amplification of human capability by use of a computer. The idea itself of course is not new, and later we will give a quotation from the 1970s expressing the same thought. In Hegedus et al. (2017), we find this idea beautifully articulated and set in the evolving sequence of human methods for mediating their thinking: symbolic marks on bone, through language, to symbols on paper, to where we are today. One of our theses—surely a fact obvious to all—is that such tool use is not innate; instead, people need to be given opportunities to learn how best to use these tools. This chapter discusses how to help people to learn how to use computational tools in mathematics, and why this is a good thing.

This chapter reflects our experiences in changing mathematical course syllabi to reflect these new tools and affordances, and may serve as a reference point in discussing future curricular changes (and what should not change) owing to the ever-shifting technological ground on which we work. Our methodology is to consider mathematics education and computer programming together. Our thesis is that the effect of computational tools, including AI, is greater and more beneficial if students are taught how to use the tools effectively and even how to create their own.

The connection between mathematics and computer programming is widely recognized and profound. Indeed, most members of the general public will (if they think about it) simply assume that all mathematicians can, and do, program computers. When mathematics is used instrumentally in science, as opposed to purely for aesthetic mathematical goals, this is in fact nearly universally true. This is because computational tools are ubiquitous in the mathematical sciences. Such tools are nowadays becoming increasingly accepted in pure mathematics, as well: see, for example, Borwein and Devlin (2009). Modern tools even approach that most central aspect of pure mathematics, the notion of mathematical proof.[2] See Richard et al. (2019) and its many references for a careful and nuanced discussion of the notion of proof in a modern technologically assisted environment, and the implications for mathematical education.

One lesson for educators is that we *must* teach students in the mathematical sciences how to use computational tools responsibly. We earlier said that the public assumes that all mathematicians can program; with a similar justification, many students assume that they themselves can, too. But programming computers *well* (or even just *using* them well) is a different story. The interesting thing for this chapter is that learning to use computers well is a very effective way to learn mathematics well: by teaching programming, we can teach people to be better mathematicians

[2] Much interest in computation and proof for pure mathematics was generated by the very successful *polymath project*. Because computation has always been perceived as instrumentally important, a corresponding but much larger scale project on the Applied Mathematics side might be the Intergovernmental Panel on Climate Change.

and mathematical scientists (Papert, 1993). This thesis is well-supported by modern research: see, for instance, (Monaghan et al., 2016) or, even for earlier levels of education, the project Learning Math through Coding (Tepylo & Floyd, 2016).

To be specific, learning "iteration" helps to understand dynamical systems; learning "recursion" helps to understand mathematical induction; learning "numerical analysis" helps to understand real (and complex) analysis, including approximation theory; learning "computational linear algebra", numerical or symbolic, helps to understand linear algebra; learning "algorithms for numerical quadrature" helps to understand calculus (both integral and differential); and learning to *write a program that works, and can be shown to work*, helps to understand how to construct a mathematical proof. We have seen this happen in our students, and felt it happen in ourselves.

Of course, this works the other way, too. For a scientist to work with Machine Learning, they need to know *linear algebra*, *optimization*, and *statistics*, perhaps even above knowing calculus. Indeed, Gilbert Strang has been saying for years that our current curriculum has "Too Much Calculus" (Strang, 2001). One of the instrumental reasons that we teach mathematics is so that scientists can be effective and productive, and this is deeply connected in the modern world with programming (Wilson et al., 2014).

We used the word "must", above: we *must* teach students how to . . .. Why "must"? For what reason? We contend that this is the *ethical* thing to do, in order to prepare our students as best we can to be functioning and thinking adults. This is more than just preparation for a job, even as a scientist: we are aiming at *eudaemonia* here (Flanagan, 2009). This observation has significant implications for the current revolution in AI-assisted teaching. We will return to this observation after discussing our experiences.

Our experience includes teaching programming to mathematical scientists and engineers through several eras of "new technology", as they have flowed and ebbed. Our early teaching experience includes the use of computer-algebra-capable calculators to teach engineering mathematics[3]; calculators were a good solution at the time because we could not then count on every student having their own computer (and smartphones were yet a distant technological gleam). Some of the lessons we learned then are still valid, however: in particular, we learned that we must teach students that they are *responsible* for the results obtained by computation, that they *ought to know* when the results were reliable and when not, and that they should *understand the limits of computation* (chiefly, understand both complexity of computation and numerical stability of floating-point computation; today, we might add that generic algebraic results are not always valid under specialization, as in Camargos Couto et al. (2020)). We claim that these lessons are invariant under shifts in technology,

[3] We had intended to give the reference (Rosati et al., 1992) for this; however, that journal seems to have disappeared and we can find no trace of it on the Web, which is a kind of testimony to ephemerality. Some of the lessons of that article were specific to the calculator, which was *too advanced* for its era and would be disallowed in schools today. We shall not much discuss the current discouragingly restricted state of the use of calculators in schools hereafter.

and become particularly pertinent when AI enters the picture. This observation is consistent with the recommendations in Wilson et al. (2014).

Speaking of shifts, see Kahan (1983) ("Mathematics written in Sand") for an early attack on teaching those lessons, in what was then a purely numerical environment. A relevant quotation from that work is

> Rather than have to copy the received word, students are entitled to experiment with mathematical phenomena, discover more of them, and then read how our predecessors discovered even more. Students need inexpensive apparatus analogous to the instruments and glassware in Physics and Chemistry laboratories, but designed to combat the drudgery that inhibits exploration. —William Kahan, p. 5 *loc cit.*

Teaching these principles in a way that the student can absorb them is a significant curricular goal, and room must be made for this goal in the mathematical syllabus. This means that some things that are in that already overfull syllabus must be jettisoned. In Corless and Jeffrey (1997) and again in Corless (2004a), some of us claim that *convergence tests for infinite series* should be amongst the first to go. Needless to say, this is a radical proposal and not likely to attain universal adoption without a significant shift in policy; nevertheless, if not this, then what else? Clearly, *something* has to go, to make room!

Curricular shifts are the norm, over time. For instance, spherical trigonometry is no longer taught as part of the standard engineering mathematics curriculum, nor are graphical techniques for solving algebraic equations (which formerly were part of the *drafting* curriculum, itself taken over by Computer Aided Design (CAD)). Special functions are now taught as a mere rump of what they were, once. Euclidean geometry has been almost completely removed from the high-school curriculum. Many of these changes happen "by accident" or for other, non-pedagogical, reasons; moreover, it seems clear that removing Euclidean geometry has had a deleterious effect on the teaching of logic and proof, which was likely unintended.

We have found (and will detail some of our evidence for this below) that teaching *programming* remediates some of these ill effects. By learning to program, the student will in effect learn how to prove. If nothing else, learning to program may motivate the student wanting to *prove the program correct*. This leads into the modern disciplines of Software Engineering and Software Validation, not to mention Uncertainty Quantification. Of course, there are some truly difficult problems hiding in this innocent-seeming suggestion: but there are advantages and benefits even to such intractable problems.

We will begin by discussing the teaching of Numerical Analysis and programming, in what is almost a traditional curriculum. We will see some seeds of curriculum change in response to computational tools already in this pre-AI subject.

## 2 Introduction to Numerical Analysis

The related disciplines of "Numerical Methods", "Numerical Analysis", "Scientific Computing", and "Computational Science" need little introduction or justification

nowadays (they could perhaps benefit from disambiguation). Many undergraduate science and engineering degrees will grudgingly leave room for one programming course if it is called by one of those names. Since this is typically the first course where the student has to actually *use* the mathematical disciplines of linear algebra and calculus (and use them *together*), there really isn't much room in such a course to teach good programming. Indeed, many students are appalled to learn that the techniques of *real analysis*, itself a feared course, make numerical analysis intelligible. [Remarkably, taking numerical analysis at the same time can make real analysis more intelligible.] In this minimal environment (at Western, the course occupies 13 weeks, with 3 hours of lecture and 1[4] hour of tutorial per week), we endeavoured to teach the following:

1. The basics of numerical analysis: *backward error* and *conditioning*;
2. How to write simple computer programs: conditionals, loops, vectors, and recursion;
3. How to debug computer programs;
4. The elements of programming style: readability, good naming conventions, and the use of comments;
5. Several important numerical algorithms: matrix factoring, polynomial approximation; solving an Initial Value Problem (IVP) for Ordinary Differential Equations (ODEs);
6. How to work in teams and to *communicate mathematics*.

As for what constitutes good programming, it is useful to relate these to the eight Best Practices for Scientific Computing (Wilson et al., 2014). By emphasizing readability of code, communication, and working in teams, we encourage students to "Write programs for people", to "Document design and purpose" and to "Collaborate". The nature of exercises is to incrementally develop codes to target problems, and we encourage modularization through functions and recursion. While we do consider debugging, time limitations mean that we are not able to teach testing or introduce version control. We will return to these as they are important omissions, with significance for mathematics and AI, particularly in relation to reproducibility and ethics.

The students also had some things to *unlearn*: about the worth of exact answers, or about the worth of some algorithms that they had been taught to perform by hand, such as Cramer's rule for solving linear systems of equations, for instance.

Western has good students, with an entering average amongst the highest in the country. By and large, the students did well on these tasks. But they had to work, in order to do well. The trick was to get them to do the work.

[4] Students were enrolled in one of three tutorial hours, but often went to all three hours.

## 2.1 Choice of Programming Language

We used Matlab. This choice was controversial: some of our colleagues wanted us to teach C or C++ because, ultimately for large Computational Science applications, the speed of these compiled languages is necessary. However, for the goals listed above, we think that Matlab is quite suitable; moreover, Matlab is a useful scientific language in its own right because *development time* is minimized by programming in a high-level language first (Wilson et al., 2014), and because of that Matlab is very widely used.

Other colleagues wanted us to use an open-source language such as Python. This is quite attractive and Python may indeed eventually displace Matlab in this teaching role. Unlike Matlab or Maple, Python is free to download and supports a very large collection of standard and community-provided libraries. Indeed, Python often features in the "Top 5" most popular languages.[5] Another reason for changing is that Python is the language of choice for Machine Learning and AI applications, with all the popular Machine Learning libraries offering a Python interface. But as of this moment in time, Matlab retains some advantages in installed methods for solving several important problems and in particular its sparse matrix methods are very hard to beat.

We also used the computer algebra language Maple, on occasion: for comparison with exact numerical results, and for program generation. Matlab's Symbolic Toolbox is quite capable, but we preferred to separate symbolic computation from numeric computation for the purposes of the course.

## 2.2 Pedagogical Methods

We used *Active Learning*, of course.[6] By now, the evidence in its favour is so strong as to indicate that *not* using active learning is academically irresponsible (Handelsman et al., 2004; Freeman et al., 2014). However, using active learning techniques in an 8:30am lecture hall for 90 or so students in a course that is overfull of material is a challenge. To take only the simplest techniques up here, we first talk about *Reading Memos* (Smith & Taylor, 1995).

---

[5] Often, these lists rely on fairly baseless claims or are derived from the number of Internet searches for a language; here, we base this claim on proportion of code on GitHub as measured by https://madnight.github.io/githut/ visited on 2021-02-09.

[6] Active learning is defined, for instance, in a well-known teaching and learning website at Queen's University, Kingston, Ontario: "Active learning is an approach to instruction that involves actively engaging students with the course material through discussions, problem solving, case studies, role plays and other methods. Active learning approaches place a greater degree of responsibility on the learner than passive approaches such as lectures, but instructor guidance is still crucial in the active learning classroom. Active learning activities may range in length from a couple of minutes to whole class sessions or may take place over multiple class sessions.".

We gave credit—five per cent of the student's final mark—for simply *handing in a Reading Memo*, that is, a short description of what they had read so far or which videos they had watched, with each programmatic assignment. Marks were "perfect" (for handing one in) or "zero" (for not handing one in). Of course, this is a blatant bribe to get the students to read the textbook (or watch the course videos). Many students initially thought of these as trivial "free marks" and of course they could use them in that way. But the majority learned that these memos were a way to get detailed responses back, usually from the instructor or Teaching Assistant (TA) but sometimes from other students. They learned that the more they put into a Reading Memo, the more they got back. The feedback to the instructor was also directly valuable for things like pacing. Out of all the techniques we used, this one—the simplest—was the most valuable.

The other simple technique we used was discussion time. Provocative, nearly paradoxical questions were the best for this. For instance, consider the following classical paradox of the arrow, attributed to Zeno, interpreted in floating point (actually, this was one of their exam questions this year):

```
1  % Zeno's paradox, but updated for floating-point
2  %initial position of the arrow is s=0, target is
       s=1
3  s = 0
4  i = 0; % Number of times through the loop
5  % format hex
6  while s < 1,
7      i = i+1;
8      s = s + (1-s)/2 ;
9  end
10 fprintf( 'Arrow reached the target in %d steps\n',
       i)
```

**Listing 1** Zeno's paradox in floating point arithmetic

In the original paradox, the arrow must first pass through the half-way point, and then the point half-way between there and the target, and so on, *ad infinitum*. The question for class discussion was, would the program terminate, and if so, what would it output? Would it help to uncomment the `format hex` statement in line 5? Students could (and did) type the program in and try it, in class; the results were quite surprising for the majority of the class.

Another lovely problem originates from one posed by Nick Higham: take an input number, $x$. Take its square root, and then the square root of that, and so on 52 times. Now take the final result and square it. Then square that, and again so on 52 times. One expects that we would simply return to $x$. But (most of the time) we do *not*, and instead return to another number. By plotting the results for many $x$ on the interval $1/10 \leq x \leq 10$ (say), we see in Fig. 1, in fact, horizontal lines. The students were asked to explain this. This is not a trivial problem, and indeed in discussing this problem amongst the present authors, JH Davenport was able to teach RM Corless (who has used this problem for years in class) something new about it.

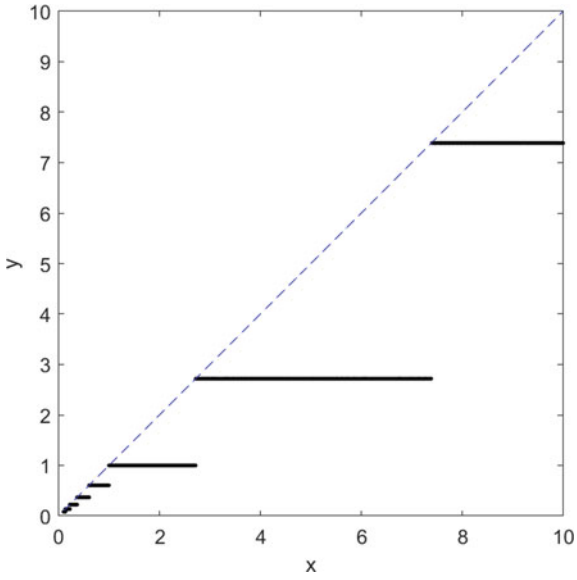


**Fig. 1** The function $y = \text{Higham}(x) = (x^{1/2^{52}})^{2^{52}}$, i.e., take 52 square roots, and then square the result 52 times, plotted for 2021 points on $0.1 \leq x \leq 10$, carried out in IEEE double precision. Students are asked to identify the numerical values that $y$ takes on, and then to explain the result. See Sect. 1.12.2 of Higham (2002) and also Exercise 3.11 of that same book, and Kahan (1980b)

We will *not* give "the answers" to these questions here. They are, after all, for discussion. [A useful hint for the repeated square root/repeated squaring one is to plot $\ln(y)$ against $x$.] We encourage you instead to try these examples in your favourite computer language and see what happens (so long as you are using floating-point arithmetic, or perhaps rounded rational arithmetic as in Derive)!

We will discuss, however, Kahan's proof of the impossibility of numerical integration (Kahan, 1980a) here, as an instance of discussing the limits of technology. This lesson must be done carefully: too much scepticism of numerical methods does much more harm than good, and before convincing students that they should be careful of the numerics they must believe that (at least sometimes) computation is very useful. So, before we teach numerical integration, we teach them that symbolic integration is itself limited, especially if the vocabulary of functions is limited to elementary[7] antiderivatives. As a simple instance, consider

$$E = \int_1^\infty \frac{e^{-y^2/2}}{y+1}\,, \tag{1}$$

which occurs in the study of the distribution of condition numbers of random matrices (Edelman, 1988). The author laconically states that he "knows of no simpler form" for this integral. In fact, neither do we, and neither do Maple or Mathematica: the indefinite integral is not only not elementary (provable by the methods of (Davenport, 1986)), it is right outside the reference books. Of course, the sad (?) fact

[7] The *elementary functions* of the calculus are not "elementary" in the sense of being simple, but instead they are "elementary" in a similar sense to the elementary particles of physics.

is, as observed in Kahan (1980a), the vast majority of integrands that occur in "real life" must be dealt with numerically. This motivates learning numerical quadrature methods.

However, it is a useful thing for a budding numerical analyst to learn that numerical techniques are not infallible, either. Consider the following very harmless function: Aphra($x$) := 0. That is, whatever $x$ is input, the Aphra function returns 0. However, Aphra is named for *Aphra Behn*, the celebrated playwright and spy for King Charles.[8] The function is written in Matlab in such a way as to *record its inputs* $x$.

```
function [ y ] = Aphra( x )
%APHRA A harmless function, that simply returns 0
%
  global KingCharles;
  global KingCharlesIndex;
  n = length(x);

   KingCharles(KingCharlesIndex:KingCharlesIndex+n-1)
   = x(:);
  KingCharlesIndex = KingCharlesIndex + n;
  y = zeros(size(x));
end
```

**Listing 2** A function named for Aphra Benn

If we ask Matlab's *integral* command to find the area under the curve defined by Aphra($x$) on, say, $-1 \leq x \leq 1$, it very quickly returns the correct answer of zero. However, now we introduce another function, called Benedict:

```
function [ y ] = Benedict( x )
%BENEDICT Another harmless function
%   But this function is not zero.
  global KingCharles;
  global KingCharlesIndex;
  global Big;
  s = ones(size(x));
  for i=1:length(KingCharles),
      s = s.*(x-KingCharles(i)).^2;
  end
  y = Big*s;
end
```

**Listing 3** A routine using information gathered by Aphra

This function is defined to be zero exactly at the points reported by Aphra, but strictly positive everywhere else: indeed the "Big" constant can be chosen arbitrarily large. If we choose Big equal to $10^{87}$, then after calling Aphra with `integral( @Aphra, -1, 1 )` first, we find the function plotted in Fig. 2. It is clearly not zero, and indeed clearly has a positive area under the curve on the interval $-1 \leq x \leq 1$.

However, asking the Matlab built-in function `integral` to compute the area under Benedict($x$) on the interval $-1 \leq x \leq 1$ gives the incorrect result 0 because the

[8] Aphra Behn 1640–1689 has one of the most interesting, if only dubiously accurate, biographies that we have read.

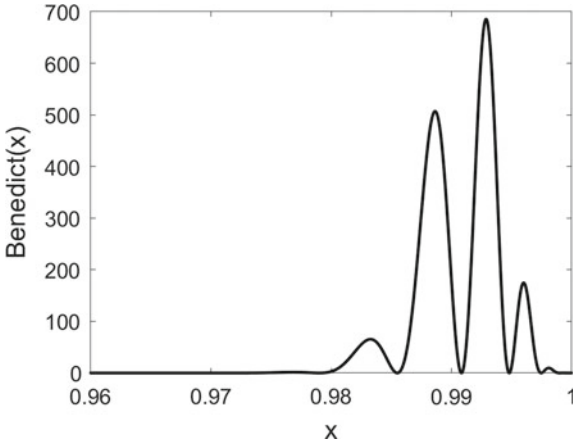


**Fig. 2** The function $\text{Benedict}(x) = K \prod_{i=1}^{150} (x - s_i)^2$ where the $s_i$ are the 150 sample points in the interval $-1 \leq x \leq 1$ reported by the function $\text{Aphra}(x)$, and with $K = 10^{87}$. We plot only an interesting region near the right endpoint of the interval. We see that the area under this curve is not zero

deterministic routine `integral` samples its functions adaptively but here by design the function Benedict traitorously behaves as if it was Aphra at the sample points (and only at the sample points). This seems like cheating, but it really isn't: finding the good places to sample an integrand is remarkably difficult (and more so in higher dimensions). One virtue of Kahan's impossibility proof is that it works for arbitrary deterministic numerical integration functions. Without further assumptions (such as that the derivative of the integrand is bounded by a modest constant), numerical integration really is impossible.

The students *do not like* this exercise. They dislike learning that all that time they spent learning antidifferentiation tricks was largely wasted, and they dislike learning that computers can give wrong answers without warning. Still, we feel that it is irresponsible to pretend otherwise.

Finally, the course was officially designated as an "essay" course. This was in part recognition for the essay-like qualities of the lab reports, but was also explicitly in recognition of the similarities between a good computer program and a good essay: logical construction, clear division of labour, and good style. It is our contention that programming and proving and explaining all share many attributes. As Ambrose Bierce[9] said, "Good writing is clear thinking made visible."

We also not only allowed but actively encouraged collaboration amongst the students. They merely had to give credit to the other student group members who helped them, or to give us the name of the website they found their hints or answers on (frequently Stack Exchange[10] but also Chegg[11] and others). Many students could

[9] Ambrose Bierce 1844–1914(?) was an American satirist, critic, and journalist, perhaps most famous for his collection of definitions published as "The Devil's Dictionary".

[10] https://stackexchange.com/ Stack Exchange is a network of websites for communities where contributors ask and answer questions and then vote on responses. Stack Overflow is for general programming but specialist communities exist, e.g., for Maths, AI, and Data Science. While generally good, the quality does vary, and is poor for computer security (Fischer et al., 2017).

[11] https://www.chegg.com/ Chegg is a homework help website.

not believe that they were being allowed to do this. The rationale is that in order to *teach* something, the student had to know it very well. By helping their fellow students, they were helping themselves more.

But modern programming or use of computers is *not* individual heroic use: nearly everyone asks questions of the Web these days (indeed, to answer some LaTeX questions for the writing of this chapter, we found the LaTeX FaQ on the Help Page on Wikibooks[12] Useful, and this even though the authors of this present chapter have *decades* of LaTeX experience). We do not serve our students well if we blankly ban collaborative tools. We feel that it is important to teach our students to properly *acknowledge* aid, as part of modern scientific practice.

## *2.3 Assessment*

But we did not allow collaboration on the midterm exam, which tested the students' individual use of Matlab on computers locked down so that only Matlab (and its help system) could be used. Examination is already stressful: an exam where the student is at the mercy of computer failure or of trivial syntax errors is quite a bit more stressful yet. To mitigate this, we gave *practice exams* (a disguised form of active learning) which were quite similar to the actual exam. The students were grateful for the practice exams, *and moreover found them to be useful methods to learn.*

Exam stress—assessment stress in general—unfortunately seems to be necessary[13]: if the students *could* pass the course without learning to program Matlab, they *would* do so, and thereafter hope that for the rest of their lives they could get other people to do the programming. Students are being rational, here: if they were only assessed on mathematical knowledge and not on programming, then they should study mathematics and leave programming for another day. So we must assess their individual programming prowess.

In contrast, the students were greatly relieved to have a final exam that "merely" asked them to (in part) write pencil-and-paper programs for the instructor to read and grade. In that case, trivial errors—which could derail a machine exam—could be excused. On the other hand, the instructor could (and did) ask for explanations of results, not merely for recitations of ways to produce them.

[12] https://en.wikibooks.org/wiki/LaTeX.

[13] Given the economic constraints of the large class model, we mean. Even then, there may be alternatives, such as so-called "mastery grading" (Armacost & Pet-Armacost, 2003). We look forward to trying that out. Exam stress is often counterproductive, and the current university assessment structures do encourage and reward successful cheating. We would like a way out of this, especially now in COVID times.

## 3 Computational Discovery/Experimental Mathematics

The courses that we describe in this section are described more fully elsewhere (Chan & Corless, 2017b, 2021). Here, we only sketch the outlines and talk about the use of active learning techniques with (generally) introverted mathematics students. The major purpose of these related courses (a first-year course and a graduate course, both in Experimental Mathematics, taught together) was to bring the students as quickly as possible to the forefront of mathematics.

> Short is the distance between the elementary and the most sophisticated results, which brings rank beginners close to certain current concerns of the specialists.
> —(Mandelbrot & Frame, 2002)

In this we were successful. For example, one student solved a problem that was believed at the time to be open (and she actually solved it *in-class*); although we were unaware at the time, it turned out to have actually been solved previously and published in 2012, but nonetheless we were able to get a further publication out of it, namely (Li & Corless, 2019), having taken the solution further. There were other successes. Some of the projects became Masters' theses, and led to further publications such as (Chan & Corless, 2017a), for example.

The course was also *visually* successful: the students generated many publication-quality images, some of which were from new Bohemian Matrix classes. Indeed some of the images on that website were produced by students in the course.

### *3.1 Choice of Programming Language*

We used Maple for this course, because its symbolic, numerical, and visual tools make it eminently suited to experimental mathematics and computational discovery; because it was free for the students (Western has a site licence), and because of instructor expertise (Corless, 2004b). For instance, Maple allows us to produce the plot shown in Fig. 3 of all the eigenvalues of a particular class of matrices. This figure resembles others produced by students in the class, but we made this one specifically for this chapter. There are 4096 matrices in this set, each of dimension 7. However, there are only 2038 distinct characteristic polynomials of these matrices because some are repeated. Getting the students to try to answer questions such as "how many distinct eigenvalues are there" is a beginning (this is not obvious, because again there are repeats: the only way we know how to answer this is to compute the Greatest Common Divisor (GCD)-free basis of the set of 2038 degree 7 polynomials, in fact). A bigger goal—in fact, the main goal of the course—was getting the students to come up with their own questions. It helped that the students were encouraged to invent their own classes of matrices (and they came up with some quite remarkably imaginative ones).

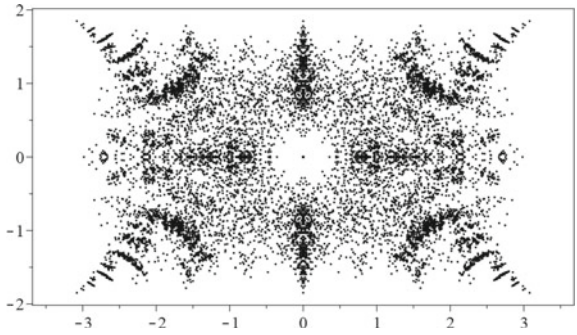


**Fig. 3** All the complex eigenvalues of all the 7-dimensional symmetric tridiagonal (but with zero diagonal) matrices with population $\{-5/3 - i, -5/3 + i, 5/3 + i, 5/3 - i\}$. There are $4^6 = 4096$ such matrices, but only about half as many distinct characteristic polynomials in the set

## *3.2 Pedagogical Methods*

This course was designed wholly with active learning in mind. It took place in the Western Active Learning Space, which was divided into six tables called Pods, each of which could seat about seven students; the tables were equipped with technology which allowed students to wirelessly use the common screens to display materials to each other. The smartboards were (in principle) usable in quite sophisticated technological ways; in practice, the varieties of whiteboards with simple coloured pens were just as useful.

Students enrolled in the first-year course were grouped with students enrolled in the graduate course. Each group benefitted from the presence of the other: the presence of the senior students was a calming factor, while the junior students provided significant amounts of energy. The grad student course also had an extra lecture hour per week where more advanced topics were covered in a lecture format.

Active learning techniques run from the obvious (get students to choose their own examples, and share) through the eccentric (interrupt students while programming similar but different programs and have them trade computers and problems) to the flaky (get them to do an interpretive dance or improvisational skit about their question). We tried to avoid the extremely flaky, but we did mention such, so that these introverted science students knew that this was within the realm of possibility.

The simplest activity was typing Maple programs that were handwritten on a whiteboard into a computer: this was simple but helpful because students learned the importance of precision, and had *immediate* help from their fellow students and from the TA.

Next in complexity was interactive programming exercises (integrated into the problems). Mathematicians tend to under-value the difficulty of learning syntax and semantics simultaneously. The amplification of human intelligence by coupling it with computer algebra tools was a central aspect of this course.

We describe our one foray into eccentricity. The paper Strange Series and High Precision Fraud by Borwein and Borwein (1992) has six similar sums. We had six teams program each sum, at a stage in their learning where this was difficult (closer to the start of the course). After letting the teams work for 20 minutes, we forced one

member of each team to join a new team; each team had to explain their program (none were working at this stage) to the new member. This exercise was most instructive. The lessons learned included

- people approach similar problems very differently.
- explaining what you are doing is as hard as doing it (maybe harder).
- basic software engineering (good variable names, clear structure, and economy of thought) is important.
- designing on paper first might be a good idea (nobody believed this, really, even after).
- social skills matter (including listening skills).

### *3.3 Assessment*

The students were assessed in part *by each other*: we used peer assessment on class presentations. The instructor informed the students that he would take their assessments and *average them with his own* because peer assessment is frequently too harsh on other students; they found this reassuring. The main mark was on an individual project, which took the full term to complete. They had to present intermediate progress at a little past the half-way point. Marks were also given for class participation.

Collaboration was encouraged. The students merely had to make proper academic attribution. While, technically, cheating might have been possible—one might imagine a plagiarized project—there was absolutely no difficulty in practice. The students were extremely pleased to be treated as honourable academics.

## 4 Programming and Discrete Mathematics

This course described in this section is also more fully explained elsewhere; see Betteridge et al. (2019). We restrict ourselves here to an outline of the goals and methods.

The course XX10190, Programming and Discrete Mathematics, at the University of Bath is both similar and dissimilar to the Western courses described above. One of the big differences is that it was designed specifically for the purpose of teaching programming to mathematical scientists by using mathematics as the proving ground. The course was designed after significant consultation and a Whole Course Review in 2008/2009. In contrast, the Western course designs were driven mostly by the individual vision of the instructor. The Bath course therefore has a larger base of support and is moreover supported by the recommendation from Bond (2018) that "every mathematician learn to program". As such, it is much more likely to have a long lifespan and to influence more than a few cohorts of students; indeed, since

it has been running for 10 years, it already has.[14] Now that RM Corless has retired from Western and the numerical analysis course has been taken over by a different instructor, the course there is already different. Conversely, all the Bath authors have moved on from XX10190, but the course is much the same. This is the differential effect of institutional memory.

Another big difference is that the course is in the first year, not the second year; moreover, it runs throughout the first year, instead of only being a 13-week course. This gives significant scope for its integrated curriculum, and significant time for the students to absorb the lessons.

However, there are similarities. The focus on discrete mathematics makes it similar to the Experimental Mathematics courses discussed above, with respect to the flavour of mathematics. Indeed, perhaps the text (Eilers & Johansen, 2017) might contain some topics of interest for the course at Bath. Although the focus is on discrete mathematics, some floating-point topics are covered and so the course is similar to the Numerical Analysis course above as well. But the main similarity is the overall goal: to use mathematical topics to teach programming to mathematical scientists, and simultaneously to use programming to teach mathematics to the same students. This synergistic goal is eminently practical: learning to program is an effective way to learn to do mathematics.

Another similarity is respect for the practical *craft* of programming: the papers Davenport et al. (2016) and Wilson (2006) discuss this further. To this end, the instructors use Live Programming (Rubin, 2013), defined in Paxton (2002) as "the process of designing and implementing a [coding] project in front of class during lecture period". This is in contrast to a counter-principle, namely the approach called "Never touch the keyboard" (during instruction), which can also have benefits. For the Western courses, an accidental rediscovery of this counter-principle proved valuable: the instructor was for several years discouraged from using keyboards owing to a repetitive strain injury, and as a consequence took to writing code on the whiteboard. This had unexpected benefits when the students would ask him to debug their code, and he would do so in a Socratic manner by asking the students to relay error messages. In doing so, the students frequently found their own solutions. However, in spite of this success, one of the most common requests from students was for live demonstrations (supplied at Western by the Tutorial Assistant): there is no question that live programming techniques can be valuable. At this moment, it is not clear which approach is more valuable, if either. We talk about some nuances of "Never touch the keyboard" in Sect. 4.2.

## *4.1 Choice of Programming Language*

A major similarity to the Western course is the choice of programming language: Matlab. As with the Western course, Matlab may eventually be displaced by Python,

[14] One British citizen in 25,000 is a graduate of XX10190.

but is an admirable first language to learn for mathematical scientists. This choice came with several unanticipated benefits, as described in Betteridge et al. (2019): for instance, the statisticians teaching subsequent courses found it simpler to teach R to students who had a working knowledge of the similar language Matlab.

## *4.2 Pedagogical Methods*

The course is fifty per cent Programming and fifty per cent Discrete Mathematics. The course is team-taught, with Lecturers and Tutors. The whole cohort have one programming lecture, one Discrete Mathematics lecture, and one Examples class per week. The roughly 300 students are divided up into tutorial groups of size roughly 10, and there is one Discrete Math tutorial per week (when budgets allow: some years this has been financially impossible, and some years these have been in groups of 20) and one Programming Lab on Fridays, after the whole-cohort classes (this apparently minor timetabling point is pedagogically very helpful, as we discovered after the first year, when we didn't have it). Management of this relatively large staff with its hierarchical structure repays attention, and the instructors have found it valuable to provide tools such as a separate mailing list for tutors. The course uses Moodle and its various electronic delivery tools.

The Lab physically holds 75 seats, divided into five tables with 15 computers each. There is one tutor for approximately ten students: students and tutors are assigned to specific groups of seats. This division allows greater and more sustained personal contact, and more active learning.

Tutors must take great care helping students in labs. The student is not just learning a language but a new logical structure, while instructors are proficient coders. When a student asks for help, it is far too easy for a tutor to "fix" the code for them, particularly when one is new to teaching. While this is the path of least resistance, because the student's priority is working code, for many not only does this do little for learning but also in fact this can be detrimental to learning. If a tutor rewrites code with no sympathy for the student's approach, this can just alienate and destroy confidence.

The philosophy of "never touch the keyboard", alluded to previously, embodies our approach. As one practices, this approach reveals subtler layers. [We have also noted that with remote teaching, although one is physically removed, practising the method is more difficult!] The philosophy applies to both instructor and student. It really means not telling students the difficulty with their draft code, but rather discovering it with them. One method is to ask what the student is trying to do, read their code with them, and try to nurture *their* creativity. It can be time- intensive, and is not easy. One needs to react to the student, taking care not to add to the student's

pain by repeating the same question[15]; methods like pseudocode and flow diagrams can be useful for withdrawing from the screen. Any suffering (on both sides) is justified when the students "get it" and the sparks of understanding light in their eyes.

### *4.3 Assessment*

Similar to the "Reading Memos" of the Western courses, the Bath course has what is called a "tickable". These are short exercises—gradually increasing in difficulty throughout the year—which are graded only on a Yes/No basis. A tickable therefore differs from a Reading Memo in that it requests some well-defined activity, whereas a Reading Memo is less well-defined and more open-ended. The similarity is in their assessment and in their encouragement of continual work throughout the course.

For instance, one tickable from midway through the first semester is given here:

**Tickable**: Write a recursive Matlab function, in the file `myexpt.m`, which will compute $A^n$ (via the call `myexpt(A,n)`) using Eq. (2), for any square matrix $A$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x \cdot x)^{n/2} & \text{if } n \text{ is even} \\ x \cdot (x \cdot x)^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases} \tag{2}$$

This tickable is then used to write another small program for quickly calculating the $n$th Fibonacci number. During lab sessions, a tutor (who has approximately 7–10 students assigned to be their tutees for the whole semester, or ideally a year) walks around the computer terminals offering help with the mathematical or programming aspects of the exercise. Students who successfully get this code running can also re-use this routine for parts of the coursework at the end of the semester.

An insufficient number (fewer than 80% of the total) of tickables marked "Yes" results in a pro rata reduction in the summative mark. This is widely perceived as fair, because there is general agreement that doing the work as you go along helps you to learn the material.

Otherwise, there is significant use of automatic assessment tools via Moodle, with tutors providing more detailed feedback on programming style.

[15] Although it's true that, sometimes, simply reading a question aloud can be surprisingly useful, of course tone matters, here. Reading the question aloud as if it were a reminder to the *instructor* can be less painful for the student.

## 5 Artificial Intelligence and Programming

The relationship between "Artificial Intelligence" and Programming is clearly deep: all forms of AI as we know it depend on programming computers. The details depend on the sort of AI being considered: much Good Old-Fashioned Artificial Intelligence (GOFAI) is programmed in LISP, Prolog, or languages based on these. The emphasis is on the manipulation of complex, often irregular, data structures containing symbolic data.

Conversely, the currently fashionable (and extremely useful in, for example, image and speech recognition) field of Machine Learning (which used to be known, perhaps more correctly, as Pattern Recognition) depends on large amounts of, generally structured, data, and a worryingly large amount[16] of numerical computation to compute the appropriate weights (anthropomorphically described as "training the network").

It is common to believe that the major languages for Machine Learning are R and Python, and in terms of the user interfaces provided, that is generally true. But we pointed out that Machine Learning is basically vast amounts of numerical computation, at which neither are particularly efficient. If one "peeks under the hood", as contributors to Croucher (2020) did, one sees a very different picture (Fig. 4). It is at this level that techniques like the use of GPUs or the more recent Tensor cores are actually deployed.

There are significant initiatives to use AI for programming now, as well; one such initiative goes by the name "AI-enabled software development". This makes the connection go both ways, just as the connection between AI and mathematics goes both ways, and just as the connection between AI and mathematics education is now beginning to go both ways. One particular area is that of "safety-critical" software. Formal proofs of safety-critical software generate vast numbers of "verification conditions" (essentially lemmas on the way to "Theorem: this software is safe") to be proved. One example (the U.K.'s National Air Traffic Services iFACTS system) is documented in Chapman and Schanda (2014, Sect. 2.6)—see Fig. 5. The 74 thousand lines of SPARK contracts (8% of the total software) generated 152,927 verification conditions, of which at the time 151,026 were proved automatically, by a combination of "Good Old-fashioned AI" and SAT/SMT checkers. A 98.76% automatic rate looks impressive, but that leaves 1901 for manual proof. Manual proof is both time-consuming and error-prone, hence the aim is to eliminate, or at least reduce, it. In automated proof theory, a relatively recent development (e.g., Kühlwein et al., 2013) is the use of Machine Learning to select the "relevant" subset of already-proved results to give the theorem-prover as a basis.

[16] Strubell et al. (2019) report that training a big model with neural architecture search can generate as much $CO_2$ as five cars during their lifetime, including fuel.

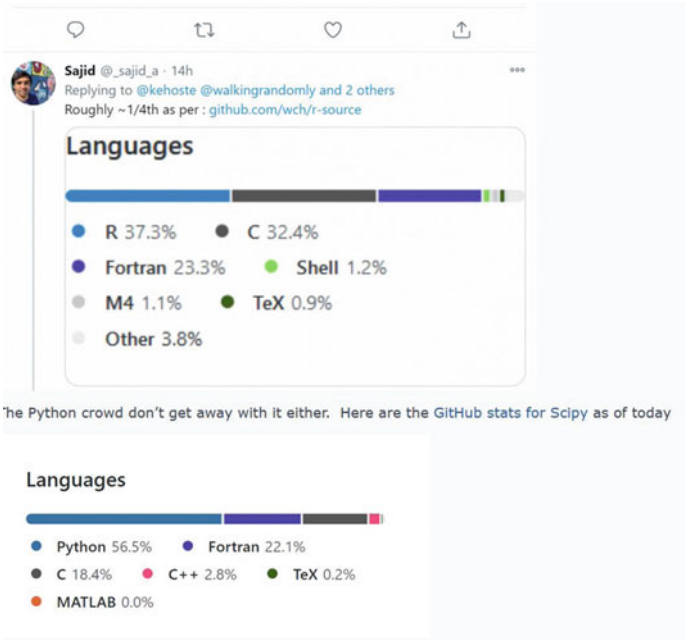


**Fig. 4** Fortran Lives! (Croucher, 2020) Of course, no SciPy means you also can't have anything that depends on SciPy including things like Keras or scikit-learn. The up-to-date figures are at https://github.com/wch/r-source and https://github.com/scipy/scipy

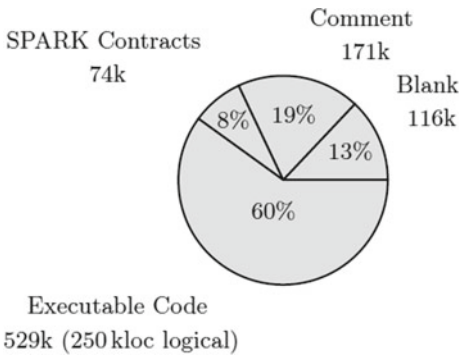


**Fig. 5** Size of NATS iFACTS (Chapman & Schanda, 2014, Fig. 1)

## 6 Outcomes

In both the Western and the Bath cases, the student surveys showed great satisfaction. For instance, the TA for the Western Numerical Analysis course twice won the "TA of the Year" award from the Society of Graduate Students. True measurement of the effectiveness of these courses is naturally difficult, but the indications pointed out in Betteridge et al. (2019), which include superior outcomes in downstream

courses, seem quite solid. There are also excellent, though again informal, pieces of evidence at Bath, which is largely an industrial placement ("sandwich" in the U.K., "co-op" in Canada) university, even in Mathematics. While this chapter was being written, one of the authors paid an industrial placement visit to a former XX10190 student spending a year as a Data Analyst at an I.T. staffing company. The student said "Programming (XX10190/MATLAB) has been really useful in my Placement year, since Power BI also requires expressing formulae to a computer system". That author had in fact never heard of Power BI before, which demonstrates the wide applicability of these skills. The company representative said "it would be great to hire another of your brilliant students". The exceptionally high employment statistics of Bath mathematics students was part of the evidence for the recommendation in Bond (2018) that "All mathematics students should acquire a working knowledge of at least one programming language".

However, since no controlled experiments were made about teaching methods—in neither case was there a control group, where different methods were used—this kind of qualitative good feeling about outcomes may be the best indication of the success that we can obtain. This clearly touches on the ethics of testing different methods of teaching, and we take this up briefly in the next section.

## 7 Ethics, Teaching, and Eudaemonia

Much published research on teaching just barely skirts rules about experimentation on humans. The "out" that is most frequently used is the *belief* on the part of the teachers that what they are doing is "best practice". It is rare to have a proper statistical design with control groups to compare the effects of innovation with mere placebo change over the status quo. The previously mentioned research on Active Learning includes some that meets this stringent standard, and as previously mentioned the evidence is so strong that it is now known to be *unethical* not to use active learning. Still, active learning is labour-intensive (on everyone's part—it's a lot simpler for a student to sit and pretend to listen in class, and then cram for an exam in the traditional "academic bulimia" model) and not everyone is willing to pay the price for good ethics.

Another significant piece of active learning is the social aspect. Humans are social animals and teaching and learning is part of how we interact in person. University students appear to value *personal contact* above nearly anything else (Seymour & Hewitt, 1997). Working against that, economics of scale means that universities want to provide certificates of learning by using only small numbers of teachers for many students; this impersonal model is already unsatisfactory for many students. This time of social isolation due to COVID-19 is making this worse, of course, in part because teaching and learning are becoming even more impersonal. One response to this pressure—and this was happening before COVID—is to try to let computers help, and to use AI to personalize instruction and especially assessment.

There is an even deeper ethical question at work, however. A teacher who taught lies[17] would be properly viewed as being unethical, even as being evil. A teacher who hid important facts from the students would be scarcely less unethical. This observation seems to be culturally universal (with perhaps some exceptions, where secret knowledge was jealously guarded, but valued all the more because of its exclusiveness). Yet, aside from idealism, what are the motivations for the teacher to tell the truth, the whole truth, and nothing but the truth?

When humans are the teachers, this is one question. We collectively know to be sceptical of the motives of people: who benefits from this action, and why are they doing this? Teaching pays, and not only in money: perhaps the most important part of our pay is the respect of those that we respect. Most of us understand that the best teachers do their jobs for the love of watching their students understand, especially seeing "light-bulb moments". But when the teacher is an app on your smartphone, the questions become different. We will take as an example the popular language app Duolingo (Von Ahn, 2013). The goals of a company that sells (or gives away—Duolingo is free by default, supported by advertising) an app to teach you something may very well be different from the goals of a human teacher. Indeed, and there is nothing hidden or nefarious about this, one of the goals of the maker of Duolingo is to *provide low-cost translation services*, essentially by distributing the translation tasks to (relatively) trusted human computers. It is an ingenious idea: make the skilled app user pay for the service of learning a new language by providing some services, more as the learning progresses, that others want. The question then becomes not "what does my teacher gain?" but rather "what does the creator of this service gain?"; more insidiously, if a teaching app became truly viral, it might be "what reproductive value does this app gain?".

The modern university system has evolved from its religious roots to provide the desired service today—namely access to the scholarship of the world—to anyone who can find a way to access the University system. We (mostly) share a belief that access to education is one of the great benefits, and provides the key to a better life, a good life, the best life possible (indeed to *eudaemonia*, in Aristotle's term,[18] although people still argue about what exactly he meant by that). It is not at all clear to us that an artificially intelligent teacher (even if humans are in the loop, as with Duolingo) would necessarily share this belief. The benefits to such a "teacher" of actively *discouraging* critical thinking are unfortunately extremely clear: one only has to look at the unbearable success of lies on social media to see the problem.

It seems clear to us that we as teachers should pay attention to the ethics of teaching by or with the help of AIs.

---

[17] Except as an important stepping stone to the real truth—see the entry "Lies to Children" in Wikipedia. Sometimes a simplistic story is the right first step.

[18] Aristotle may have done us a disservice by looking down on crafts and craftspeople; the term Software Carpentry is not likely to induce respect for the discipline in academia, for instance. We lament this prejudice.

## 8 Concluding Remarks

> Instead there must be a more serious concern with the significant ways in which computational resources can be used to improve not so much the **delivery** but rather the **content** of university courses.
> —(Abelson, 1976)

The content of mathematics courses has changed over the past few decades (this has been noted in many places, but see, for example, Corless & Jeffrey, 1997). Some of that change has been forced by the increasing number of students and their necessarily more diverse backgrounds and interests; some of that change has been deliberate abandonment of no-longer-useful techniques; and some of that change has been driven by the introduction of new tools, some needed for new applications such as AI, which in turn is being applied to teaching. We claim that an important part of that curricular change was in fact AI; remember that *symbolic computation was the first form of AI that worked.* One might consider compilers as early thin AI, or even numerical computation as AI (after all, machine computation replaced human intellectual labour)! But symbolic computation came out of the AI labs at MIT. Getting a computer to find an antiderivative was considered to be fundamentally harder than finding a derivative; that was a watershed. Symbolic computation has been affecting the mathematics curriculum for more than 60 years now. We don't think it's finished affecting the curriculum.

These tools synergistically affect one another: the development of AI changes mathematics teaching, mathematics, and programming; mathematics teaching changes mathematics, programming, and AI; and of course, advances in programming change AI, mathematics, and mathematics teaching. The cycle is dizzying.

One new tool that we have not yet talked about is WolframAlpha. This is nearly universally available, free, almost uses natural language input—it's pretty accepting, and the students find it simple to use—and produces for the most part very legible, neat, and useful answers to problems at roughly the first year university level. Its language recognition features are indeed a kind of AI. We believe that its use (or the use of similar tools) should not only be allowed in class but also encouraged. The students will still be *responsible* for the answers, and it helps to give examples where WolframAlpha's answers are wrong or not very useful, but it is irresponsible of us to ignore it. Matlab, Maple, Mathematica, Python, NumPy, and SymPy provide other tools for mathematical thinking, on a larger scale. We believe that it is incumbent on us as educators to teach our students the kinds of mathematics that they can do when using those tools.

Developing AI further depends upon mathematicians who understand the numerical methods that underpin machine learning, reinforcement learning, etc., as well as how to program. But good programming practices are not just about what code is written but the way in which it is created. While theorems need proofs and the requirement for correctness and reproducibility in mathematical sciences is paramount, academia has been slow to apply this rigorously to its codes. In software development, this is achieved with testing, validation, and version control. While comparison with expec-

tation and (better) analytic proof is adequate for validation, we have not formally taught testing or version control in our undergraduate programmes. The time pressure on curriculum cannot excuse this omission much longer. The value of adopting professional practices goes beyond those who will work as software engineers. They are vital tools for working efficiently, contributing to open software, for data scientists and AI engineers to manage data and to ensure trust in the methods that they develop and apply in their careers. These enable students to use their computational tools responsibly.

We also have not talked here about GeoGebra, which is probably now the most popular computational thinking tool for mathematics in the world. This is because we are "old guard" (well, some of us are) and GeoGebra is a newer tool, one that we have not yet used. However, it seems clear to us that the same principles that we have been using for our other tools also apply here: the students should be aware of the program's limitations; the students should know when the answer is correct and when it is not, and the students should be responsible for the answers.

And we have not talked at all about perhaps the most significant tool: the Internet itself. Collectively, we have vastly more capable memory now than we did before—mathematical definitions can be looked up instantly (indeed, Wikipedia is quite reliable for mathematics, though not perfect)—and we have nearly universal access to papers, projects, course materials, and more. The growth in mathematical videos on YouTube is astonishing: Tom Crawford ("The Naked Mathematician"), Bobby Seagull, Numberphile, Online Kyne, and many more provide wonderfully accessible mathematics. Is that AI? Given that many videos are automatically close-captioned, we would say that there is at least some AI involved. Then there are the algorithms for finding such videos automatically for you.

With the advent of modern AI tools like these for mathematics education, more questions arise. We believe that amongst the most important questions for AIEd will be about the *ethics* underlying the tools. We are not merely talking about the debates over using homework-assistance websites to cheat for class. We all know now that machine learning can easily copy our biases and prejudices, without us intending, or unintentionally alter the content of what we are conveying.[19] We also know that the goals of developers of AIEd tools may well be different than the goals of good teachers. We forbear from trying to define what it means to be a "good teacher'; this has been the subject of considerable debate since forever, in all cultures, but see, for instance, (Boynton, 1950). We take from that short, well-written article, out of many possible choice quotations, "Good teaching must be based on an unequivocal, sincere interest in the human individual." Achieving this alone will be a challenge for AIEd.

The ethics of AIEd is beginning to be studied intensively (see, for instance, the works (Aiken & Epstein, 2000; Sijing & Lan, 2018)), but clearly we are only just

[19] See also Bradford et al. (2009), which shows that computational tools can affect the basic meaning of equality: pedagogical equality is not the same as mathematical equality. It is perfectly possible for two expressions to be mathematically equal, but only one expression to be the desired student response.

scratching the surface of the issues, which include some very deep classical philosophical problems, including how to live a good life (achieve eudaemonia). The amplified human life, when humans use computers to increase their thinking capability, clearly also needs philosophical study, as we are touching directly on very classical problems. Not only philosophers, but cognitive scientists, as well as computer scientist experts in AI, will be needed to properly develop these tools.

Plus ça change, plus c'est la même chose.—Jean Baptiste Alphonse Karr, 1849

**Acknowledgements** RMC thanks the Isaac Newton Institute for Mathematical Sciences and the staff of both the University Library and the Betty and Gordon Moore Library at Cambridge for support and hospitality during the programme Complex Analysis: Tools, techniques, and applications, by EPSRC Grant # EP/R014604/1 when some of the work on this project was undertaken. RMC likewise thanks the University of Bath for an invitation to visit Bath, at which this project was started. EYSC and RMC also thank Western University for a grant to work on the project *Computational Discovery on Jupyter*, some of whose results are discussed here.

## References

Abelson, H. (1976). Computation in the undergraduate curriculum. *International Journal of Mathematical Education in Science and Technology, 7*(2), 127–131.

Aiken, R. M., & Epstein, R. G. (2000). Ethical guidelines for AI in education: Starting a conversation. *International Journal of Artificial Intelligence in Education, 11,* 163–176.

Armacost, R. L., & Pet-Armacost, J. (2003). Using mastery-based grading to facilitate learning. In *33rd Annual Frontiers in Education, 2003. FIE 2003* (Vol. 1, pp. T3A–20). IEEE.

Betteridge, J., Davenport, J. H., Freitag, M., Heijltjes, W., Kynaston, S., Sankaran, G., & Traustason, G. (2019). Teaching of computing to mathematics students: Programming and discrete mathematics. In *Proceedings of the 3rd Conference on Computing Education Practice* (pp. 1–4).

Bond, P. (2018). The era of mathematics–review findings on knowledge exchange in the mathematical sciences. Engineering and physical sciences research council and the knowledge transfer network. https://epsrc.ukri.org/newsevents/news/mathsciencereview/.

Borwein, J., & Devlin, K. (2009). *The computer as crucible: An introduction to experimental mathematics*. The Australian Mathematical Society (p. 208).

Borwein, J. M., & Borwein, P. B. (1992). Strange series and high precision fraud. *The American Mathematical Monthly, 99*(7), 622–640.

Boynton, P. L. (1950). What constitutes good teaching? *Peabody Journal of Education, 28*(2), 67–73.

Bradford, R., Davenport, J., & Sangwin, C. (2009). A comparison of equality in computer algebra and correctness in mathematical pedagogy. In J. Carette et al. (Eds.), *Proceedings intelligent computer mathematics* (pp. 75–89).

Broussard, M. (2018). *Artificial unintelligence: How computers misunderstand the world*. MIT Press.

Camargos Couto, A., Moreno Maza, M., Linder, D., Jeffrey, D., & Corless, R.M. (2020). *Comprehensive LU factors of polynomial matrices*. Mathematical Aspects of Computer and Information Sciences MACIS (pp. 80–88).

Chan, E. Y. S., & Corless, R. M. (2017a). A new kind of companion matrix. *The Electronic Journal of Linear Algebra, 32,* 335–342.

Chan, E. Y. S., & Corless, R. M. (2017b). A random walk through experimental mathematics. In J. M. Borwein (Ed.), *Commemorative Conference* (pp. 203–226). Springer.

Chan, E. Y. S., & Corless, R. M. (2021). *Computational discovery on Jupyter* (In preparation).
Chapman, R., & Schanda, F. (2014). Are we there yet? 20 years of industrial theorem proving with SPARK. In *Proceedings of Interactive Theorem Proving* (pp. 17–26).
Corless, R. M. (2004a). Computer-mediated thinking. *Proceedings of Technology in Mathematics Education*. https://github.com/rcorless/rcorless.github.io/blob/main/CMTpaper.pdf.
Corless, R. M. (2004b). *Essential Maple: An introduction for scientific programmers*, 2nd ed. Springer Science & Business Media.
Corless, R. M., & Jeffrey, D. J. (1997). Scientific computing: One part of the revolution. *Journal of Symbolic Computation, 23*(5), 485–495.
Croucher, M. (2020). No Fortran? No data science in R and Python! https://walkingrandomly.com/?p=6696.
Davenport, J. (1986). On the Risch differential equation problem. *SIAM Journal on Computing, 15,* 903–918.
Davenport, J. (2018). Methodologies of symbolic computation. In: *Proceedings AISC* (pp. 19–33).
Davenport, J. H., Hayes, A., Hourizi, R., & Crick, T. (2016). Innovative pedagogical practices in the craft of computing. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)* (pp. 115–119). IEEE.
Edelman, A. (1988). Eigenvalues and condition numbers of random matrices. *SIAM Journal on Matrix Analysis and Applications, 9*(4), 543–560.
Eilers, S., & Johansen, R. (2017). *Introduction to experimental mathematics*. Cambridge University Press.
Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., & Fahl, S. (2017). Stack overflow considered harmful? The impact of copy& paste on android application security. In *38th IEEE Symposium on Security and Privacy (SP)* (pp. 121–136).
Flanagan, O. (2009). *The really hard problem: Meaning in a material world*. MIT Press.
Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H., & Wenderoth, M. P. (2014). Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences, 111*(23), 8410–8415.
Handelsman, J., Ebert-May, D., Beichner, R., Bruns, P., Chang, A., DeHaan, R., et al. (2004). Scientific teaching. *Science, 304*(5670), 521–522.
Hegedus, S., Laborde, C., Brady, C., Dalton, S., Siller, H.-S., Tabach, M., Trgalova, J., & Moreno-Armella, L. (2017). *Uses of technology in upper secondary mathematics education*. Springer Nature.
Higham, N. J. (2002). *Accuracy and stability of numerical algorithms*, 2nd ed. SIAM, Philadelphia.
Kahan, W. M. (1980a). Handheld calculator evaluates integrals. *Hewlett-Packard Journal, 31*(8), 23–32.
Kahan, W. M. (1980b). Interval arithmetic options in the proposed IEEE floating point arithmetic standard. In *Interval Mathematics 1980* (pp. 99–128). Elsevier.
Kahan, W. M. (1983). Mathematics written in sand. In *Proceeding of the Joint Statistical Meetings of the American Statistical Association* (pp. 12–26). http://people.eecs.berkeley.edu/~wkahan/MathSand.pdf.
Kovács, Z., Recio, T., Richard, P. R., & Vélez, M. P. (2017). GeoGebra automated reasoning tools: A tutorial with examples. In *Proceedings of the 13th International Conference on Technology in Mathematics Teaching* (pp. 400–404).
Kühlwein, D., Blanchette, J., Kaliszyk, C., & Urban, J. (2013). MaSh: Machine learning for Sledgehammer. In *International Conference on Interactive Theorem Proving* (pp. 35–50).
Li, A., & Corless, R. M. (2019). Revisiting Gilbert Strang's "A chaotic search for $i$." *ACM Communications in Computer Algebra,53*(1), 1–22.
Luckin, R., Holmes, W., Griffiths, M., & Forcier, L. B. (2016). *Intelligence unleashed: An argument for AI in education*. Pearson Education.
Mandelbrot, B. B., & Frame, M. (2002). Some reasons for the effectiveness of fractals in mathematics education. In *Fractals, graphics, & mathematics education* (pp. 3–9).
Monaghan, J., Trouche, L., & Borwein, J. M. (2016). *Tools and mathematics*. Springer.

Papert, S. (1993). *Mindstorms*, 2nd ed. Basic Books.
Paxton, J. (2002). Live programming as a lecture technique. *Journal of Computing Sciences in Colleges, 18*(2), 51–56.
Richard, P. R., Venant, F., & Gagnon, M. (2019). Issues and challenges in instrumental proof. In *Proof Technology in Mathematics Research and Teaching* (pp. 139–172). Springer.
Rosati, P. A., Corless, R. M., Essex, G. C., & Sullivan, P. J. (1992). An evaluation of the HP28S calculator in calculus. *Australian Journal of Engineering Education, 3*(1), 79–88.
Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 651–656).
Seymour, E., & Hewitt, N. M. (1997). *Talking about leaving*. Boulder, CO: Westview Press.
Sijing, L., & Lan, W. (2018). Artificial intelligence education ethical problems and solutions. In *2018 13th International Conference on Computer Science & Education (ICCSE)* (pp. 1–5). IEEE.
Slagle, J. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Journal of the ACM, 10,* 507–520.
Smith, R. C., & Taylor, E. F. (1995). Teaching physics on line. *American Journal of Physics, 63*(12), 1090–1096.
Strang, G. (2001). Too much calculus. http://www-math.mit.edu/~gs/papers/essay.pdf.
Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 3645–3650).
Tepylo, D. H., & Floyd, L. (2016). Learning math through coding. https://researchideas.ca/mc/learning-math-through-coding/.
Von Ahn, L. (2013). Duolingo: Learn a language for free while helping to translate the web. In *Proceedings of the 2013 International Conference on Intelligent User Interfaces* (pp. 1–2).
Wilson, G. (2006). Software carpentry: Getting scientists to write better code by making them more productive. *Computing in Science & Engineering, 8*(6), 66–69.
Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., et al. (2014). Best practices for scientific computing. *PLoS Biology, 12*(1), e1001745.