CASE STUDY

Developing computational mathematics provision in undergraduate mathematics degrees

Richard Elwes, School of Mathematics, University of Leeds, Leeds, UK.
Email: r.h.elwes@leeds.ac.uk. https://orcid.org/0000-0002-6752-5501.
Rob Sturman, School of Mathematics, University of Leeds, Leeds, UK.
Email: r.sturman@leeds.ac.uk. https://orcid.org/0000-0001-7299-9931.

Abstract

Over the last ten years we have comprehensively embedded computational mathematics, and in doing so programming, into the undergraduate mathematics degree programmes at the University of Leeds. This case study discusses some of the practical, organisational and pedagogical issues we encountered, and how we addressed them.

Keywords: computational mathematics, Python, employability.

1. Introduction

Although this special issue is about "programming in the mathematics curriculum", this article is about the more specific topic of computational mathematics. Any definition of "computational mathematics" (beyond "the study and practice of solving mathematical problems with a computer") might involve terms such as numerical analysis, algorithmic thinking, symbolic manipulation, computer-assisted proof or complexity theory. Certainly each of these topics contains sufficient mathematics, and computation, to fill a course for undergraduates. They might, or might not, also require a student to learn to program a computer.

A review of computer programming provision (Sangwin & O'Toole, 2017) within UK mathematics programmes states that computing is "*more popular amongst applied mathematicians and statisticians*", with "*very few explicit pure mathematics courses*" within their sample of programming modules. Our view is that undergraduate computational mathematics should be as broad as possible. We argue to our students that there are two good reasons to turn to a computer when confronting a mathematical problem: when you know exactly what you want to do, and when you don't. The first of these is typically applied mathematics and numerical methods – we have a set of routine calculations to perform, and lack only the time to do these ourselves. The second covers more open-ended problems, possibly with no known solution, where we use a computer to seek evidence, visualise possible solutions, reveal mechanisms and aid our understanding.

Our approach is best illustrated with a typical pair of questions:

1. Write a function that performs Euclid's algorithm on a pair of positive integers $a > b$, returning their greatest common divisor.
2. Investigate the following results of Lamé and Heilbronn:
    a. The number of steps required by Euclid's algorithm is at most 5 times the number of digits of b.
    b. On average the number of steps required is equal to $\frac{12\ln 2}{\pi^2}\log_{10} a$ for all pairs a, b.
    c. The number of steps is maximised when a and b are consecutive Fibonacci numbers.

Notice that the question requires the students both to convert a familiar algorithmic process into computer code, and to explore for themselves mathematical facts that are likely to be beyond their

current mathematical sophistication to prove. We found it relatively easy, and quite liberating, to devise such questions of this nature, including topics such as the Collatz conjecture, elementary number theory (e.g. Goldbach's conjecture and Euler bricks), continued fractions, iterated function systems, sorting algorithms and cryptography.

Thus our purpose in developing a set of modules on computational mathematics at the University of Leeds was: to introduce mathematical problems which are appropriate for solving with a computer; to see why some problems are *not* suitable for solving in this way; to present some fundamental techniques in mathematical computation; to encourage an investigative spirit in attacking problems which are not intended to be solved completely. We promote this independent approach to learning by scheduling only one hour of lecture time per week for material delivery, leaving the bulk of student time to smaller workshop sessions staffed by a team of demonstrators.

"doing my own research made me feel truly more independent and free to let my mathematical creativity flow, which can't always be the case" (Anonymous student feedback, 2019)

Clearly, studying computational mathematics in this sense inevitably involves learning rudimentary programming techniques. Thus, computational mathematics provides students with an opportunity to acquire valuable transferrable skills, much as delivering oral presentations on mathematical topics provides offers both narrow curricular and broader life-skill benefits. Both activities provide mechanisms to broaden students' range of experience without diluting core mathematical content. Indeed our syllabus often supports the rigorous mathematical content of other modules.

"*I enjoyed the freedom to learn many things through one, gaining a deeper understanding of mathematics, increasing my ability to solve problems, and learning how to program"*
(Anonymous student feedback, 2018)

In Leeds, and likely in many places, our introduction of this module was not the very first step towards introducing computation into the mathematics curriculum. There were already a small number of computational components in the form of worksheets using proprietary mathematical software such as Matlab, Maple, and Mathematica. But these were not the best use of limited resources (financial or time) and mutually incompatible. Thus, introducing computational mathematics was not wholly new, but rather the opportunity to replace legacy teaching methods with expanded modern ones.

This case study describes challenges and (our) solutions in devising a practical computational mathematics module (initially), and then a suite of modules. In the spirit of programming, we describe these challenges as bugs, and give our fixes to these problems in the context of a large UK university mathematics department. Of course, such problems may differ in their reproducibility across different institutions, as may the viability of solutions.

2. Planning the module

Bug summary: Colleagues hate the idea.

Bug description: There is distrust among academic colleagues in mathematics that programming is something we *should* introduce into our curriculum. Don't many students choose a degree in Mathematics precisely because they *don't* want to work with computers? Isn't this diverting student (and staff) time away from our core task of learning (and teaching) mathematics?

Fix: There were four arguments we made to colleagues, and the combination was compelling.

(i) Our external advisory board (which comprises local and national employers in a range of industries and services) had given a strong steer that they would value greater computational skills in mathematics graduates. See CBI/Pearson (2019) for development of this point.

(ii) In an environment in which we were encouraged, and keen, to develop our provision of "employability" skills within the mathematics curriculum, programming expertise seemed the natural way to achieve this whilst retaining mathematical content.

(iii) With growing student numbers, and final year projects becoming more individual and resource-intensive, having a student cohort who were confident at investigation was clearly desirable. We argued that computational mathematics formed the ideal place to introduce problems which may not be soluble by an undergraduate mathematician, but which are amenable to experimentation and investigation.

(iv) Importantly, we appealed to a wide cross-section of academic colleagues. This module was never intended to simply be numerical analysis. Both pure and applied mathematicians became convinced that the content of the computational mathematics module was relevant to their field. Elementary number theory provides a wonderful pool of problems for investigation, while automatic theorem proving, an area of pure mathematics in which Leeds has research expertise, has been developed within our undergraduate programming syllabus.

Bug summary: We don't know what language to choose.

Bug description: Institutions may have reasons to choose different languages (the need to prepare students to use particular software in later years; existing proprietorial software deals; the expertise of whoever is developing and teaching the module).

Fix: Several factors led us to choose Python as the single language for the module. Firstly, we wished to move away from proprietary mathematical software. Sangwin & O'Toole (2017) found Matlab to be the most popular language in compulsory year 1 modules mathematics degrees in the UK (of those institutions who responded). Maple is also well represented, and had previously been in use at Leeds within first year modules, albeit in a somewhat peripheral capacity. A comparison between Python and Matlab (and R) in a pedagogical setting is (Ozgur et al., 2017). However, and regardless of any financial incentive, we saw a real opportunity in switching to a general purpose programming language, in that it would allow us to offer our students a genuine transferrable skill, valued by employers (we will return to this point below).

"*I feel I have learnt a lot of practical skills to take into the workplace"*
(Anonymous student feedback, 2019)

As well as being free and general-purpose, we wanted something with a quick entry time for first-time programmers. Python proved well-suited to this, in our case via the open-source distribution Anaconda and its interface Spyder. We found this straightforward to install on institution-wide cluster machines, and quick and easy for individual students on their own Windows, Linux and Mac OS X machines. Crucially all such installations looked and operated in essentially identical ways.

Python is generally recognised as being a good choice for first-time programmers, a judgement with which we agree: it is dynamically-typed, "duck-typed" (meaning that objects generally behave in accordance with naïve expectation), it has a relatively small vocabulary of keywords most of which are simple English terms, the use of indentation aids readability, it compiles on-the-fly. Thus the journey to the execution of a first "hello world" programme is as quick and simple as for any language (and much more so than, say, C or Java).

A relevant example of Python's duck-typing can be seen by entering "3/2" (dividing the integer 3 by the integer 2). Python 3 automatically outputs a float (1.5). It is easy to take this silent type-change for granted, and in a mathematical setting we often find it convenient to do so (in other cases we deploy integer division 3//2). However, it is also worth spending a moment on the powerful pedagogical point here, that a computer is only ever representing the idea of mathematical quantity, and it may usefully do so in different ways (as illustrated by Python 2 interpreting "/" differently).

Thus we have found that Python represents a better balance than say C or Java, in terms of the relative prominence of syntactic versus mathematical or algorithmic concerns for beginner programmers. This is also illustrated by the kinds of error that such students make. All programming languages are liable both to syntax errors (such as, in Python, confusing "=" and "==") and logic/semantic errors in which (roughly speaking) the code runs or compiles but does not behave as intended. However, our experience is that the relative prevalence of these types of error differs: in C & Java syntactic errors predominate, while in Python beginning students spend longer grappling with semantic/logic errors. We view this as positive, as these errors have greater mathematical/algorithmic educational value.

Python, we argue, allows mathematical and algorithmic aspects to come to the fore rather than being concealed behind technicalities. Consider Figure 1, a Python programme for implementing Euclid's algorithm (and thus a possible solution to question 1 in Section 1). The coding can be done in four short lines using a small number of simple commands, allowing the student to focus on the (significantly more challenging) mathematical matter of understanding how this programme achieves its goal:

```
def gcd(a, b):
    while b>0:
        a, b = b, a % b
    return abs(a)
```

Figure 3: A simple Python function to compute the greatest common divisor of integers *a* and *b*.

Note the single line in which *a* and *b* are re-assigned to the values of *b* and (*a* mod *b*) respectively. Single-line multiple assignment is possible in Python, but not in C or Java, where either the introduction of a temporary variable, or some other method, would be required. Features of this sort, such as list comprehensions, dictionaries and function decorators are typical in Python, and we have found them highly appropriate for computational mathematics. Notice, for instance, the code `Y = [myfunction(x) for x in X]` which illustrates list comprehension is highly reminiscent of applying a mathematical function to every object in a domain `X`, to obtain its range `Y`. Thus `Y = [x**2 for x in range(10)]` is, we claim, not only more elegant than writing a loop, but is more mathematically intuitive. The "Zen of Python" (Peters, 2004) contains several guiding principles which we believe also encapsulate mathematical attitudes: beautiful is better than ugly; explicit is better than implicit*; simple is better than complex; complex is better than complicated.

* We are grateful to the anonymous reviewer for fairly objecting that the earlier remarks about dynamical typing could be seen to contradict this proverb. We might respond by drawing distinctions between explicitness at the level of code (which

Python also carries the benefit of being increasingly popular in the industries in which our graduates typically seek employment. The TIOBE Company maintains the *TIOBE Programming Community Index*, a popularity metric for programming languages as measured by a variety of internet search tools. Figure 2 puts Python firmly in third place, behind Java and C, but growing significantly in popularity in recent years.

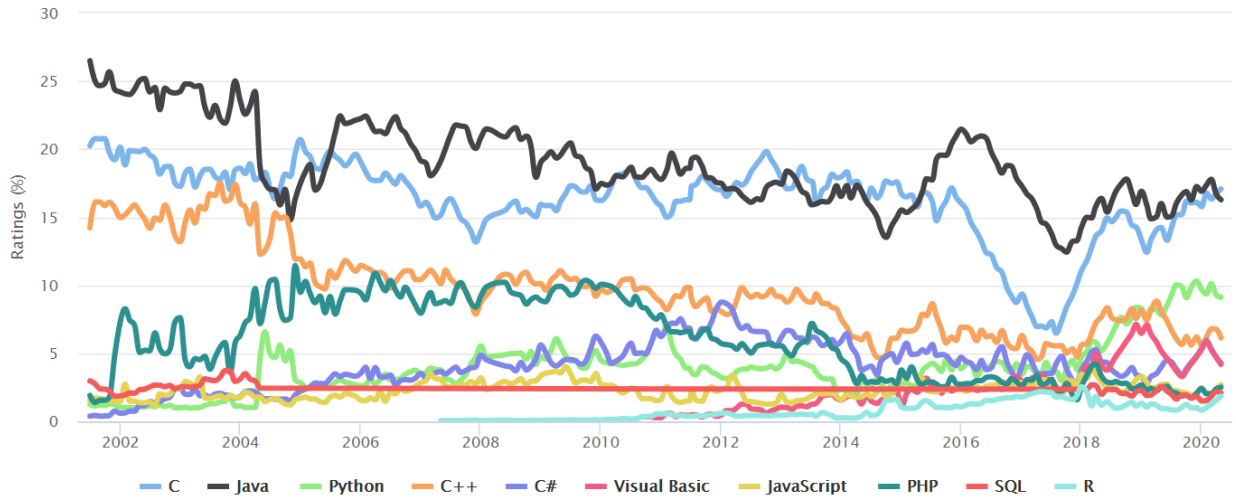

Figure 4: Popularity ratings of various languages from the TIOBE Programming Community (TIOBE, 2020)

Bug summary: I worry about accessibility and inclusivity.

Bug description: The particular software used may not be fully accessible and care must be taken to meet the needs of all students.

Fix: Of course solutions need to be personalised to meet the individual needs of students. We give a particular example of a visually impaired student who was worried about being able to use the software efficiently and effectively. After listening to her describe her particular requirements, we were able to help her set up her interface, using a combination of control key-strokes and magnification. She found Spyder more accessible than notebooks, and said:

"*Firstly, I'd set my screen up so that it was split vertically with the python console on one side and the editor window on the other side. I found this was the best way to fit the most information on the screen when you'd enlarged the text. To enlarge the text I would [use key-strokes] to zoom in to specific sections of the code, [together with] the standard Windows magnifier [to navigate the software features]. Using these two magnification techniques I managed to navigate my way around Spyder and the specific code quite effectively*".

is desirable) versus at the level of the language itself (where the suppression of declaration of variables is convenient), and between explicitness in describing procedures versus that describing objects. However, we do not have the space to develop such arguments further.

We also note that there are communities working towards the inclusivity and accessibility of Project Jupyter's software, including iPython notebooks (GitHub, 2019).

3. Running the module

Bug summary: Marking students' work is time-consuming

Bug description: With a class of over 300 students, each submitting 10 pieces of coursework (3 summative, 7 formative) through the semester, our module undeniably entailed a great deal of marking.

Fix: Recruiting extra staff to help with the marking is likely an unavoidable part of the solution (our module has a team of 5 or 6 markers, either postgraduate students or tutorial assistants). Nevertheless, the use of appropriate technology has allowed us to bring the problem down to manageable proportions (and we hope for further improvements to come).

In 2019, we worked with colleagues Craig Evans and Samuel Wilson from the School of Electronic and Electrical Engineering, to adapt their automatic feedback and assessment platform (Evans and Wilson 2019) to the specific needs of our module. This system tests the Python functions within submissions against specified inputs, generating a feedback text file for each student. It is thus automatic to judge functions as being correct, and to identify common errors and award partial marks accordingly. Human input is concentrated in places where it is essential: in setting up the marking system, in identifying uncommon errors (and hopefully thereby improving the system), in making sense of error-strewn submissions, and most valuably in assessing styles of questions which cannot be done automatically. These include plotting graphs, commenting on the results of experiments, performing open-ended investigations such as question 2 in section 1. (We remark that solutions to such questions are also much harder to plagiarise, which forms an important part of our strategy in that regard.)

Another benefit of automatic marking has been that students get accustomed to working under tighter instructions: in our system functions must be given specific names, they must accept inputs of a given type, and they must output their results in a specified format. Any deviation from these requirements will cause the automatic marker to fail, and students are now used to marks being lost in such cases.

4. Developing computational mathematics further

Bug summary: After the module, students want to do more computational mathematics.

Bug description: This should be welcomed as a success, of course. Nevertheless, there is a risk of either disappointing enthusiastic students or creating additional pressure on already crowded timetables.

Fix: Although it would be possible to create an entire follow-on module entitled "Advanced Computational Mathematics", there was little appetite for this among staff. Rather our approach has been to embed further computation in pre-existing modules, in two ways.

The computational mathematics module we have discussed runs in the first semester of second year. We created computational 'add-ons' to existing second year (second semester) modules. There are currently three such, that have been brought in at different times. In each case, a pre-existing 10 credit module was supplemented with a new 5 credit computational part. Students can either study the original 10 credit theoretical module or the new 15 credit "with computation" variant.

We have such modules in Numerical Analysis, Discrete Mathematics, and Logic. In each there are twin aims: to apply computational methods in the given setting, and to develop Python techniques for which there was little or no time in the Computational Mathematics module. For instance, in Discrete Mathematics with Computation, students learn to build recursively defined functions; in Numerical Analysis with Computation students construct new classes for rational numbers and learn how to handle vectors and matrices efficiently.

All Leeds students undertake a research project which runs through both semesters of their final year. Third year mathematicians select their project from a published list of over 50 titles. These vary widely in subject-matter (pure, applied, statistical, financial) and in the tools and techniques required. However, several such projects require computation of assorted kinds, in either major or minor ways. Indeed, one of the goals of the introduction of our Computational Mathematics module was to allow the scope of such projects to be broadened. For example, there are projects on discrete random processes, automatic puzzle-solving, and numerical equation solving. Students' prior knowledge of Python has undoubtedly extended the possibilities here. To broaden the range still further and cater to the most enthusiastic students, we initiated a dedicated project-title on "Computational Applied Mathematics" in which students select from several topics including implementing Fast Fourier Transforms and modelling a nonlinear pendulum.

5. References

CBI/Pearson, 2019. *Education and learning for the modern world*. Available at: https://www.cbi.org.uk/media/3841/12546_tess_2019.pdf [Accessed 29 May 2020].

GitHub, 2019. *jupyter / accessibility*. Available at: https://github.com/jupyter/accessibility [Accessed 29 May 2020].

Ozgur, C., Colliau, T., Rogers, G., Hughes, Z. and Myer-Tyson, B., 2017. MatLab vs. Python vs. R. *Journal of Data Science*, *15*(3), pp.355-372. Available at: http://www.jds-online.com/volume-15-number-3-july-2017 [Accessed 29 May 2020].

Evans, C.A., and Wilson, S.S., 2019. Development of an automated feedback and assessment platform. *Horizons In Stem Higher Education Conference, Kingston University, 3 July 2019*. Available at: https://ukstemconference.files.wordpress.com/2019/06/horizons-conf-19-abstract-booklet-.pdf [Accessed 29 May 2020].

Sangwin, C.J. and O'Toole, C., 2017. Computer programming in the UK mathematics curriculum. *International Journal of Mathematical Education in Science and Technology*, 48(8), pp.1133-1152. https://doi.org/10.1080/0020739X.2017.1315186.

Peters, T., 2004. *The Zen of Python*. Available at: https://www.python.org/dev/peps/pep-0020/ [Accessed 29 May 2020].

TIOBE, 2020. *TIOBE Programming Community Index*. Available at: https://www.tiobe.com/tiobe-index/ [Accessed 29 May 2020].