Interdisciplinary Teaching of Computing to Mathematics Students: Programming and Discrete Mathematics

James H. Davenport[a] David Wilson[a] Ivan Graham[b]
Gregory Sankaran[b] Alastair Spence[b] Jack Blake[b]
Stef Kynaston[b]

Departments of Computer Science[a] and Mathematical Sciences[b],
University of Bath, Bath BA2 7AY
J.H.Davenport@bath.ac.uk

Abstract

The teaching of programming to mathematics students has been a thorny pedagogical issue for many years. Should the mathematicians do it, or the computer scientists? Here we outline Bath's solution to the issue, which is "both, in close collaboration, to an interdisciplinary syllabus". This solution (using MatLab) is now in its fifth year, and is taught to 300 students/year. It has been received well by the students, and by other lecturers who can build with confidence on the skills learned in this course.

1 Introduction

This paper describes a novel approach to the integrated teaching of computing and discrete mathematics to mathematics students, based on:

- Close collaboration, and team teaching, by the Departments of Mathematical Sciences and Computer Science;
- Development of a bespoke interdisciplinary syllabus, 50% discrete mathematics and 50% computing, rather than an "off-the-shelf" Computer Science syllabus;
- Using programming concepts as concrete instantiations of the mathematics concepts taught in the course, for example recursion as a counterpart to induction, viewing Fast Fourier Transforms as a "divide and conquer" algorithm;
- Choice of a programming delivery vehicle (MatLab) close to the immediate needs of the students;

- Attention to the pedagogy of the *craft* of programming. Many of our ideas are similar to those of Vihavainen, Paksula and Luukkaine (2011), though this postdates our early work.

2 Background

Until 1997, the University of Bath had a School of Mathematics Sciences, including a Computing Group. Programming was taught in Fortran, until in 1984 the first author led a move to C. Relevance to, and preparation for, the future computing streams was the principal criterion. Then a separate Computer Science degree, with a different first year, was introduced. This paper focuses on the evolution and practice of computing as it is taught to *mathematics* students.

Until 2009, first year Mathematics students (of which there are currently 304) took a programming course provided by the Computing Group and later by the separate Computer Science Department. The emphasis was on programming *per se* in a general-purpose language: C until 2000, then Java. The main weaknesses of this, frequently identified by students, were the lack of apparent relevance and the lack of connection with the rest of the curriculum where programming was used in later years, either in MatLab or R. In fact R is very similar to MatLab from a programming point of view. We now give students a one-page list of the differences and expect them to adapt, and they do with no fuss. Following restructuring and detailed curriculum review (though a complete curriculum review is not a necessary requirement!) the current model evolved. The course is called, and delivers, **Programming And Discrete Mathematics**. It runs thoughout the first year, as one of five streams: the others are Algebra, Analysis, Mathematical Methods and Probability/Statistics. The course is 50% Programming and 50% Mathematics, so the "programming" share of the first year is 10%, which is unchanged, but its effectiveness has greatly increased.

3 Overall Design

3.1 Aims

In practice, the aim, which underpinned all the thinking as the course was being designed, was

> The course should **be**, and be seen to be, relevant to the rest of the mathematics curriculum, and not just as "a useful skill for later on".

From this followed the fact that it could not be *just* a computing course. Certain amounts of discrete mathematics (using the term slightly loosely) were added or moved from elsewhere, and we ended up with a 50:50 mixture with the programming (in MatLab). This mathematics included orders of growth and the O-notation (which students seem to find more approachable with a concrete application), elementary graph theory, Fast Fourier Transforms, elementary coding theory and cryptography (Diffie–Hellman and RSA). The coding theory relies

on the linear algebra taught in the algebra stream: moreover, it uses and emphasises the fact that it is taught over an arbitrary field. The cryptography part is helped by the fact that the MatLab Symbolic Toolbox allows examples with realistic-sized numbers: indeed the students do two problem sheets, identical except that one has two-digit numbers for hand calculation, and the other has 60-digit numbers for MatLab-assisted computation. However, by far the most important in terms of relationship with the programming was the teaching of induction.

The first few weeks of the course are based around the thesis that the Mathematical definition of induction is *equivalent* to programming implementation by recursion.

The first example, literally in week 1, is the Fibonacci numbers, which are defined by induction, programmed by recursion, and in the next four weeks have theorems on growth proved by induction, and have these related to the O-complexity of the programs produced earlier. More specifically, three families of solutions to the Fibonacci problem are presented, with the lecturers and students proving the complexity results.

Exercise 1 (Directly recursive) *Use $F_n = F_{n-1} + F_{n-2}$ to work down to the base cases. The complexity is exponential in n: $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.*

Exercise 2 (Iterative) *Define the base cases and use $F_i = F_{i-1} + F_{i-2}$ to work up until $i = n$. The complexity is linear in n: $O(n)$.*

Exercise 3 (Matrix-based) *Therefore playing to the strengths of MatLab:*

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} \text{ hence } \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix},$$

and using "divide-and-conquer" (recursive) exponentiation we can compute F_n in time logarithmic in n: $O(\log n)$.

3.2 Reinforcing the Link

One challenge when teaching an interdisciplinary course is convincing the students of the benefit of learning a topic outside of the perceived scope of their degree. Whilst many mathematicians use programming, or the thought processes behind programming, in their daily lives, this is not obvious to first year mathematics undergraduate students.

To answer the question "why are we learning programming on a mathematics degree?" a short (five to ten minutes) talk is given fortnightly by one of the tutors, or a lecturer who doesn't otherwise teach on the unit, to the whole cohort. This is at the beginning of a lecture and the tutor/lecturer explains a little about their research, and how they use programming. These seem to have been well-received, with thoughtful questions from the students after each one.

The link between mathematics and programming is further reinforced by the whole teaching body. The professors and tutors are sourced from both

the Mathematics and Computer Science Departments, and range in specialities from Algebraic Geometry via Computer Algebra to Numerical Analysis and Computer Vision.

3.3 Course Text

After some debate, we decided to go for an, essentially mandatory, course text. Ideally we would have chosen one that fitted the aims of the course, but a fairly extensive search revealed none. We therefore opted for a portmanteau combined book (something seen often in North America, and we were agreeably surprised to discover how willing the U.K. arms of publishers were to produce one, *provided* it contained their material), containing large portions of Chapman (2013) and Epp (2011), with some additional material by ourselves. This book is sold on campus and only available to University of Bath students.

The hard part was selecting the MatLab book. While there are literally hundreds of MatLab books, nearly all of then focus on its use as a toolbox, not as a programming engine. While designing the course, we discovered Chapman (2009), which *does* focus on the programming element, and has material on program design, debugging and documentation, and is a plausible support for the programming side of the course. This tied us into the publisher of Chapman (2009), i.e. Cengage. Hence in 2009 we looked at their range of Discrete Mathematics books and chose (the previous edition of) Epp (2011). In particular, it had a chapter on O-notation in the context of algorithm efficiency, which filled the major gap in Chapman (2009) from a Computer Science point of view. For the initial custom text, we chose nearly all of Chapman (2009) and about half of (the previous edition of) Epp (2011). In 2010 we used the same, with a small amount of additional material, notably correcting out-of-date MatLab examples and screen shots from Chapman (2009). In 2011 we made no changes, and in 2012 we went to (about half of) Chapman (2013), about half of the current edition of Epp (2011), and rather more own material, notably envisaging the Fast Fourier Transform (mathematics side of the course) as a "divide and conquer" algorithm, as already used on the programming side.

4 Delivery

4.1 Overview

The course runs throughout the teaching year (October–May), and is taught on the basis of two lectures and one whole-cohort problems class per week, the same as the other streams. It is team taught, with a computer scientist (the first author) and a mathematician taking responsibility for the lectures and problems class, and tutors, typically postgraduates or final-year MMath students, taking responsibility for the practical laboratories and mathematics tutorials.

4.2 Whole-Cohort Classes

In a typical week, each lecturer gives one lecture, and both share the problems class, going over past and ongoing work. The ratio in the problems class is dynamic, and will be roughly 25% programming and 75% mathematics, unless more time is needed to respond to student queries on programming aspects.

4.3 Laboratory Classes

These are weekly and last for 50 minutes. We are fortunate enough to have a 75-seater laboratory, split into five benches of 15 machines each. Each student is assigned to a *specific* bench in a *specific* laboratory session, and the bench/session has a designated tutor. Thus each student has the same tutor for the whole semester.

4.4 Mathematics Tutorials

These are weekly and last for 50 minutes. One tutor teaches around 16 students in a small classroom, and also marks the students' problem sheets. The emphasis is on the mathematics work, and a standard tutorial consists of a 20 minute worked example, 20 minutes discussing solutions to the previous week's problem sheet, and 10 minutes discussing concepts for the coming week's problem sheet. Students attempt the worked example before a model solution is given, and questions from the students are encouraged throughout.

4.5 Virtual Learning Environment

We make heavy use of the University's Moodle VLE. Course materials and problem sheets are distributed via it, Coursework and tickable exercises are collected via it, feedback is given on coursework via it. In addition, the lecturers tend to respond to student e-mail queries by posting on the Moodle Forum, rather than replying directly, so that all students can see the (anonymised) question/answer.

5 Assessment

We gave a great deal of thought to this in the design process. Both mathematics and programming are subjects in which one learns by doing — no amount of lectures on induction will make the student into a competent author of induction proofs, and no amount of lectures on programming will make the student into a competent programmer.

Therefore, most weeks the students have two pieces of work to do for the following week.

5.1 Weekly Work — Mathematics

This is in the traditional "problem sheet" format. Until 2013, this was managed alongside the programming work, but the students commented, rightly, that this meant it got short shrift compared with the programming — when sat in front of a computer one's tendency is to pay it attention! Hence we now run separate mathematics tutorials on the same basis as the other streams. These are proving successful, with students commenting on their benefit in mid-semester feedback questionnaires.

5.2 Weekly Work — Programming

In weeks when there is no significant effort on summative practical work, there is a weekly "tickable" exercise — in practice 12 in the year. By "tickable" we mean that they are assessed as pass/fail by the laboratory tutors in the sessions, and the students (and tutors) are told that we expect students to be able to pass every exercise with reasonable diligence. For example, the first such exercise is to write a recursive MatLab function to compute Fibonacci numbers from $F_n = F_{n-1} + F_{n-2}$, by analogy (this is made explicit in the exercise) with the supplied programme for computing factorials based on $n! = n * (n-1)!$. The supplied programme was constructed in front of the students in the lecture, and hence this follows the paradigm described as *modeling/scaffolding* by (Vihavainen et al. 2011, §2.1).

Lest this seem too trivial, this is the specification of the last tickable in semester 2.

Exercise 4 *Write two MatLab functions:* `TreeAdd` *and* `traverse`*, in files of the same name.* `TreeAdd(t,str)`*, where* `t` *is a binary tree of strings and counts, and* `str` *is a string, returns the new tree with* `str` *inserted in order (or with the corresponding count increased if* `str` *was already in the tree).* `traverse(t,@f)` *should traverse such a tree* `t`*, calling the function* `f` *on each node in turn, in alphabetical order of the strings. In the parlance of [the programming] lectures, it should therefore do an* **inorder** *traversal.*

The incentive for the tickables is that the tickables develop the students' programming skills, and lead up to the summative practical work: with the motivation that failure to get 80% ticks will result in the summative coursework marks being reduced *pro rata*. In practice we have very rarely had to apply this restriction: the (very few) students who do not do the tickables fail the practical work anyway.

Many tickables from 5 onward are supported by a quiz: good students answer a quiz based on running their code, and then can submit their code using "conditional assignments" in the Moodle 2 online platform (or our own bespoke solution in Moodle 1). This is done in advance of the laboratory session, so that the tutors can look at it in advance, and spend the lab session concentrating on the weaker students who have not done the quiz.

5.3 Summative Practical Work — 50% of course marks

There are four pieces of summative practical work, which make the 50%: the first piece of work is worth 14%, and the following pieces are each worth 12%.

Coursework 1: 14% This is set in November and due at the end of teaching for Semester 1, typically mid-December. It comes in four parts: the first two are straight from previous exercises, and parts 3 and 4 are independent of each other but essentially build on part 2. Each part is worth 20% of the coursework (2.8% of the the course total), and is automatically marked, which has largely been a success: the first author has already taught the students about "black box" testing, and emphasises that marking will be tested this way. The tutors are then asked to check the results for reasonableness. For example, students who get arguments in the wrong order will typically get 0 from the automatic marking, whereas the tutor will swap the arguments and re-run the test, and report the result to the lecturer, who will allocate the "correct" marks less an appropriate penalty. In addition, the tutors allocate 20% for style and commenting.

Class Test: 12% This is done in the first week of Semester 2 (February), in front of the computers, and students are expected to use them, and submit both written answers and MatLab programs. A typical question is the following.

> **Exercise 5** *The sequence* $\{x_n\}$ *is defined by* $x_1 = a$ *and* $x_n = \frac{x_{n-1}^2 - 2}{2x_{n-1}}$ *for* $n \geq 2$. *Write a MatLab function which takes as inputs a positive integer* n *and a positive real number* a *and produces* x_n.
>
> 1. *When* $a = 50$, *what is* x_8 *correct to 10 decimal places.*
> 2. *It is known that for all* $a > 0$, x_n *converges to a limit as* $n \to \infty$. *What is that limit? Give your answer as a surd (a proof is not required).*

Coursework 2: 12% This is as coursework 1: four parts, with parts 3 and 4 independent of each other. This exercises the more advanced programming (data structures) taught at the start of Semester 2. In academic year 2013/14, for the first time, we are also going to exercise MatLab's object-oriented programming features.

Coursework 3: 12% The same structure as coursework 2, and in particular exercises functional arguments.

5.4 Examinations — 50%

There are two examinations on the course. In both of them, students are allowed (and strongly encouraged) to take the course text into examinations. It is also pointed out that rote learning of MatLab features, or indeed mathematical

definitions, will not help as these are all in the text, and therefore will not be examined.

January Worth 15% of the course total, and focusing entirely on the mathematics content of Semester 1 (October-December, with revision in January).

May Worth 35% of the course total, with 12% being on the year's Programming and 23% being on Semester 2's mathematics.

Therefore each of Programming and Discrete Mathematics gets 50% of the marks, split 12:38 and 38:12 respectively between examinations and practical work.

6 The Tutor Experience

As a tutor, interactions with students are through the weekly laboratories or tutorials. Often tutors will teach in both laboratory sessions and tutorials, allowing them to teach both aspects of the course. The tutors are PhD or Masters (MSc and MMath) students in both the Mathematics and Computer Science Departments. The professors take the career development of all tutors seriously: for example the authors of this paper include a PhD student from each Department and a PhD (ex-MMath) student from Mathematical Sciences.

Most of the interactions in the laboratory sessions are done one-on-one, whilst tutorials are more like a standard classroom environment. It is up to the tutors to reinforce the links between the mathematics and programming in these sessions, especially for those students struggling. A typical situation may be a student who has never seen a recursive program before and keeps forgetting to include base cases. Getting the student to make the link to induction proofs (something they are more familiar and comfortable with) allows them to see the necessity of the base cases, and work out how many they need. Obviously, if a student is more comfortable with programming, then this can be used to help them understand the mathematics. Due to the one-on-one nature of this style of teaching it can be difficult for tutors to allocate their time evenly across all students during laboratories. Each tutor tends to find their own way of tackling this issue, but it is one of the more challenging aspects of tutoring. Announcements to all groups at the start of a lab session explaining common programming issues and key concepts can help.

On the practical side of things, tutors need to be proficient enough in MatLab to handle standard issues, especially in the first few weeks. While this was a challenge at the beginning, we are now in the position that MMath students will themselves have taken this course in their first year. Often students arrive having never used a programming environment and it is important that any frustration or problems stemming from this are quickly identified and clarified so as not to interfere with the students understanding of the concepts. Since there are five tutors in the laboratory, at least one of whom will have tutored the course before, it is very rare for the tutors to be stuck with a MatLab issue.

Once the students are working on assessed coursework, the tutors play a different rôle. Their help is targeted at general MatLab guidance rather than helping to complete any specific task. Students are expected to start debugging their own work independently by this point and tutors will not point out specific errors beyond those in syntax. Help is therefore focused on the debugging process itself, with students being encouraged to add 'print' statements and not suppress variables at key points in the functions. Moreover, students are encouraged to pay attention to any MatLab error statements, which are often very descriptive and helpful in diagnosing runtime problems with their codes. Students who are stuck on how to proceed with specific coursework tasks are directed to relevant work (both from laboratories and tutorials) they have already completed in order to draw parallels.

Tutors assign 20% of the coursework marks for 'coding style': presentation of code and commenting. Tutors therefore can give advice to students about what criteria they may look for, but students are left to achieve this standard by themselves.

7 Outcomes

In an ideal world, we would present before/after student questionnaire results. Unfortunately, the university changed both the method (paper to online) and the questions on unit evaluations as we were introducing this course, so we cannot present such data, and the evidence is necessarily qualitative.

1. The rhetorical "why on earth are we doing this?" questions have been replaced with (fewer) genuine "why are we doing this?" questions, and the Mathematical Sciences Department has a much better answer!

2. The immediate customers of such a Year 1 course are the Year 2/3 courses, and the Numerical Analysis lecturers have noted a marked improvement in MatLab programming skills, while R programming has ceased to be an issue for Statistics lecturers.

3. It should be noted that Bath is largely a sandwich university, even in Mathematics: 84 students were in placement in 2012–13. While only a small number use MatLab directly, a common statement is

   > the programming skills they learnt in first year through MatLab and R helped them in their placement.

4. While this course is frequently discussed at Staff-Student Liaison Committee and by the Directors of Studies, the focus is always on minor improvements (2013's was Mathematics tutorials), not "was this a good idea?".

The authors believe that the interdisciplinary approach exemplified by this course is the right approach to the thorny problem of teaching programming to mathematics students.

Acknowledgements: we are grateful to the many other tutors (notably Marios Richards and Ieuan Evans for their work on the automated coursework marking) who have contributed, and to the students whose views have shaped the delivery. The MSOR Connections referees have greatly improved the paper.

References

Chapman, S.: 2009, *Essentials of MATLAB Programming*, Cengage.

Chapman, S.: 2013, *MATLAB(R) Programming with Applications for Engineers*, Cengage.

Epp, S.: 2011, *Discrete Mathematics with Applications, 4th edition*, Brooks/Cole Cengage Learning.

Vihavainen, A., Paksula, M. and Luukkaine, M.: 2011, Extreme Apprenticeship Method in Teaching Programming for Beginners, *Proceedings 42nd ACM technical symposium on Computer Science Education*, pp. 93–98.