Math with Programming – Shaken or Stirred?

Miroslav Lovric, McMaster University*

Introduction

The idea of using programming (i.e., writing a computer code) to engage with mathematics is nothing new. As a matter of fact, certain activities involved in this approach—some are discussed in this paper—are essential components of deep learning, and are as old as mathematics itself. List of competencies that students have an opportunity to attain by using programming to solve mathematics problems is quite long, and includes: self-motivation to do and to explore mathematics, development of mathematics intuition, critical reflection on problems, approaches to solving and on the results obtained, working with abstraction and different representations, and many other (Buteau et al, 2014a).

In this paper I discuss teaching and learning situations which emerged in a problem-solving mathematics course I was teaching (in what follows I will refer to it as "the course"), especially in reference to using computer programming. The objectives of the course were to learn basics of programming, and—more important—to investigate problems in mathematics by using computer code. Needless to say, experimentation is a very important component of learning mathematics. Through working with computer code, students look for patterns, discover interesting facts, investigate given conjectures, and create their own hypotheses and test them.

Benefits of using programming to explore mathematics are well known and documented (see, for example, Howson & Kahane, 1986, King et al, 2001, or Buteau et al, 2014b). Unlike in most of these papers, the approach in a part of this paper is microscopic in that I discuss certain details related to programming. As well, more globally, I comment on the interaction between programming and mathematics.

Learning math with programming has two aspects: learning programming and learning math by using programming. Neither of the two tasks are easy or trivial. As a matter of fact, programming turns out to be quite challenging to learn (Robbins et al, 2003)

> Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates.

Similar views are presented in (Ma et al, 2011). We find important insights into students' experiences with programming in (Kinnunena and Simon, 2012). In this paper we discuss the second goal: learning mathematics by using programming.

* Hamilton, Ontario, Canada. Contact: lovric@mcmaster.ca

Learning Programming

At the start of the course, students work on activities that help them understand the very basics of computer programming. As in a generic mathematics course, students came into my course with a huge diversity in programming skills. Some had already taken a course in programming, and some had experience working with (mostly one-line commands in) Maple or Matlab. However, about a half of students in the course had no previous programming experience.

Students were free to work with any programming language they knew, or were comfortable with, or were willing to learn. However, in order to have a common context within which I could discuss programming issues, I decided to use a pseudocode. Vaguely modeled on Maple, the pseudocode was introduced to the students through a list of statements, and basic examples they had to examine. In terms of statements, we used input (enter) and output (write) statements, a loop (for *i* from *a* to *b* … end loop) and a conditional statement (if *A* then *B* else *C* end if). For instance, as one of the first examples of pseudocode we introduced a loop

```
enter N
for i from 1 to N
write i
end loop
```

and described the way in which computer understands it and executes it. Next, students were asked to produce the output of the following code which introduced a conditional statement

```
enter N
for i from 1 to N
if i is even then write i else write “not even” end if
end loop
```

Although this code looks fairly simple, it nevertheless presented difficulties for some students: on the one hand, they had to adopt the grammar and the vocabulary of this new language, and on the other, they had to think about the structure of the algorithm represented by this code.

The dual role of the equals sign was another initial challenge: it took some time to internalize the fact that the equal sign in the conditional statement if a=b then … and the equals sign in the assignment a=b+c have different roles. Students often internalized the former as checking whether or not a number is a solution of a given equation, and the latter as computing the value of a given b and c. The assignment structure a=a+1 (used often to count occurrences of some event of interest), was something new to students as well.

However, with a good amount of practice, students were able to understand and internalize all of the challenges mentioned here. This is not at all surprising, given

the fact that in order to understand any piece of mathematics, one needs to spend time thinking about it and working with it.

The idea of introducing pseudocode was to focus on the essential components of mathematics and programming; in particular:

- translation of a mathematical problem into a computer code that would help to solve or to investigate the problem; in other words, students had to modify a given math problem into a form of an algorithm
- understanding the algorithmic and logical structures of a computer program

The challenge that I did not foresee was that students wanted to work on their laptops/computers right away. For them, the fact that we were dealing with computer-related tasks meant that they should be using a computer from the start. The idea of writing code on paper (i.e., to proceed as if solving a math question) did not come naturally to them.

As a consequence, instead on focusing on important aspects, students shifted their focus on working immediately in the language of their choice (Maple, MATLAB, Python and C++), and were eager to make the computer do what they wanted it to do. This approach presented a number of difficulties to some students, and considerably slowed down their learning of programming.

Learning Mathematics By Programming

There are numerous benefits stemming from introducing programming into mathematics curriculum at all levels. (In this paper we focus on tertiary education experiences.) For instance, students get a chance to investigate mathematics problems which, using formal approach, might be too difficult for their level of education. Note that we said "investigate" and not "prove."

In this course, using simple computer code, students were able to generate the first hundred thousand prime numbers and investigate the gaps between consecutive prime numbers. As well, they were able to sketch the graph of the prime-counting function $\pi(x) =$ number of prime numbers less than or equal to (a real number) x, and compare it to Gauss' approximation $x/\ln x$.

As example of an unsolved math problem, we studied Collatz conjecture (also known as the $3n+1$ conjecture, or Ulam's conjecture): start with a positive integer n; if n is even, then divide it by 2, and if n is odd, then compute $3n+1$; repeating this calculation (starting with many values of n) leads to the infinite cycle 1, 4, 2, 1, 4, 2, 1.... Collatz conjecture states that this is true for all positive integers (in its usual form, the conjecture states that this process, for every n, eventually reaches 1). Several simple computer codes enabled my students to find out interesting facts about this conjecture. For instance, one student discovered that there is a sequence

of 17 consecutive numbers which require exactly the same number of steps to reach 1 (all numbers between, and including, 7083 and 7099 need exactly 57 iterations to reach 1).

Beyond obvious benefits, the experience students gained will help them when they take our courses in number theory, and revisit some of the problems they worked on in a more formal way. It is a great learning experience to contrast numeric evidence with proof – since a proof usually uses an approach completely different from numeric investigation.

Another important investigation in my course related to a random walk. Students had to write a code that simulates the 2-dimensional random walk (some students investigated higher dimensions as well) and figure out (among other questions) the relationship between the number of steps taken and the average distance from the starting point. This project enriched students' experience when learning statistics and diffusion in physics.

Paper-and-Pencil and Computer: Different Media, But All is Mathematics

Since the course included paper-and-pencil problem-problem solving activities as well, I was able to contrast students' thinking, approaches to solving problems and attitudes in using two distinct media: pencil and paper and computer.

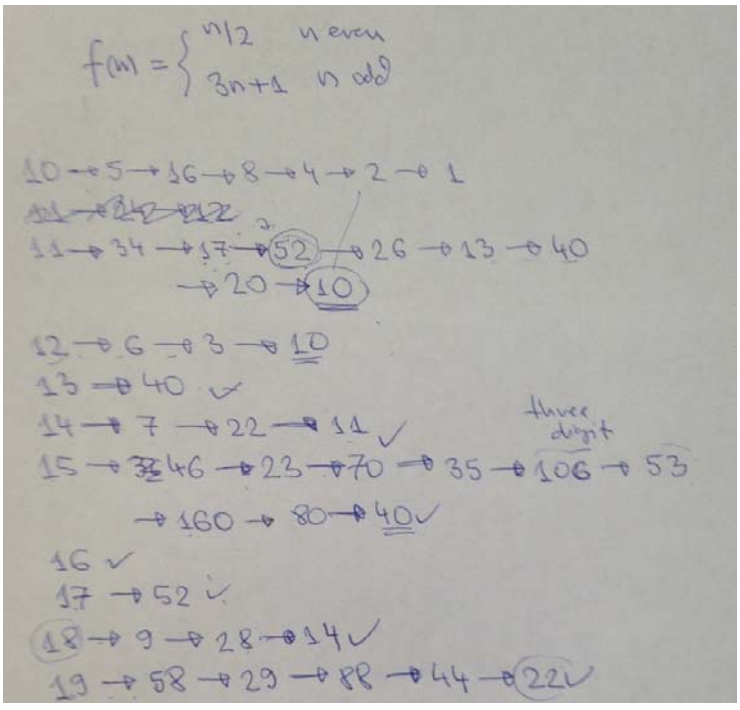

Figure 1: Investigation of Collatz conjecture on paper: iterations for numbers from 10 to 19 have been investigated, with a realization that subsequences from one sequence of iterations appear in other sequences.

Perhaps the largest obvious difference is a record (or an absence) of students' work. With a pencil-and-paper approach, quite often there is a preserved history of attempts, decisions, experiments, and/or alternatives considered (Figure 1).

However, when working on a computer, students routinely overwrite erroneous code, or delete attempts that do not yield a satisfactory result (Figure 2).

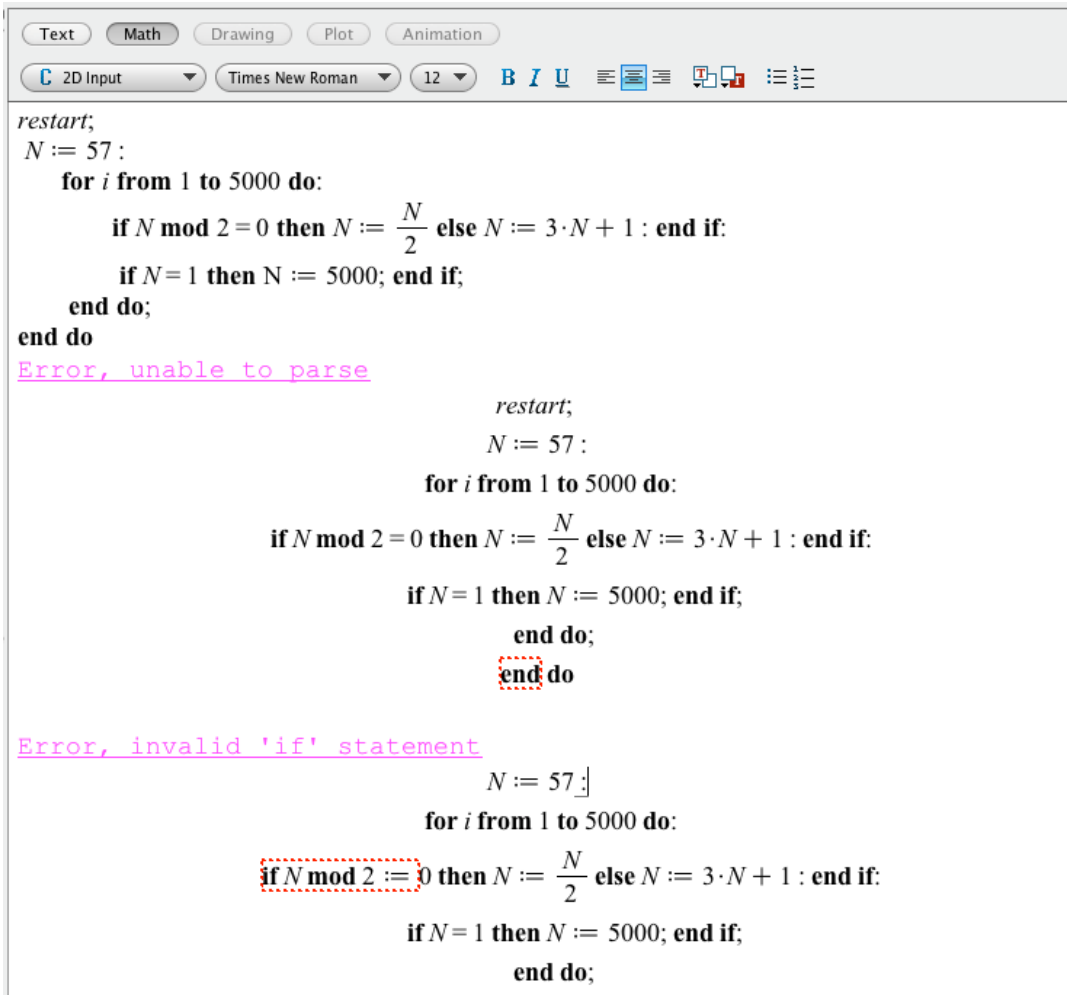

Figure 2: Attempts at coding Collatz conjecture; the bottom third is a Maple diagnostic from the previous, overwritten, version of the code; the middle part is a Maple feedback to the latest version of the code, which is in the upper third of the screenshot.

Needless to say, this absence of history of attempts at a problem presents a true challenge for a formative evaluation. On the other hand, paper record of student's work gives an opportunity of continuous dialogue between a course instructor and a student (Figure 3).

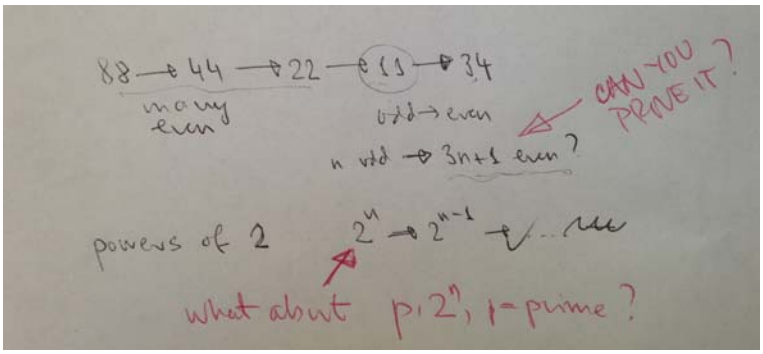

Figure 3: Discovering facts about Collatz conjecture on paper, which would be harder to notice by running a computer code and examining its output

Working with computer code in front of a computer and working with code on a piece of paper are two different cognitive experiences.

In pencil-and-paper thinking about an algorithm, students have an opportunity to visualize the program structure (Figure 4, left). Thinking about the double loop, they realize that the requirement that the loops must be nested has a useful visual representation: the arcs which connect the start of the loop to its end are not supposed to cross! A conditional statement is visualized as a fork (Figure 4, right). Of course, visual imagery can be (and is) realized on a computer as well—some commonly used programs have a visual interface, which replaces typing the actual commands.

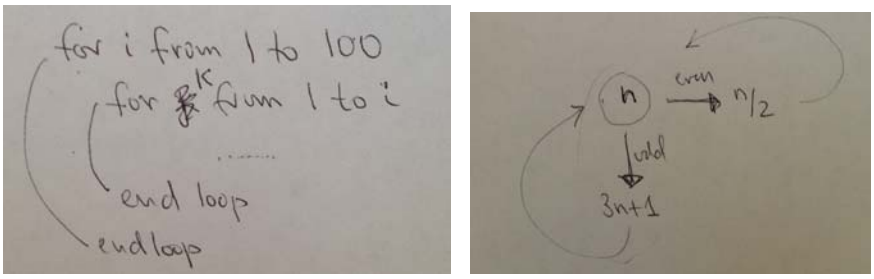

Figure 4: Coding on paper: visual representations of a double loop (left) and of a conditional statement (right)

The code which we enter into a computer does (hopefully) what we want it to do—but nothing more than that. A pencil-and-paper investigation of a problem can only enrich the experience; for instance (Figure 3) student discovered patterns "playing" with the Collatz conjecture: an odd number is always followed by an even number (thus, a sequence of iterations of any number cannot have two or more odd numbers in a row) ; powers of 2 have a straightforward decreasing sequence to 1.

Lessons Learned from Practice

Very early in the course it became obvious that learning programming is in many ways similar to learning mathematical routines and algorithms. Course instructor needs to provide numerous opportunities for practice, and that practice needs to be designed so that the students advance only when they have mastered previous steps. For instance, it is important that students understand exactly how a loop works in simple cases (such as the first code in section *Learning Programming*) before more complicated structures are introduced (such as the conditional statement within a loop in the second code in section *Learning Programming*).

In spite of availability of online manuals and numerous websites with examples, it turned out that students reacted most enthusiastically to the usual teaching (i.e., interactions with a "live" instructor), as they have experienced in their math classes.

Learning computer programming takes time, and a significant amount of practice is needed—so that students have a true opportunity to understand the structure of a computer program, and are capable of judging whether the answer obtained using that program makes sense. This could be accomplished by studying examples of a given code and figuring out what it does, and then adjusting it to answer related questions. In discussing novice vs. expert programmers, Rist (1995) wrote

> Expertise in programming should reduce variability in three ways: by defining the best way to approach the design task, by supplying a standard set of schemas to answer a question, and by constraining the choices about execution structure to the 'best' solutions. (Rist, 1995, p. 552)

Furthermore (Robbins et al, 2003)

> Many of the characteristics of expert programmers are also characteristics of experts in general, as explored, for example, in other fields such as chess or mathematics.

Although pseudocode is useful, it is ultimately the programming language whose syntax and grammar need to be learned. In this, one needs experience as well: for instance, some students initially confused an operation on the whole array with an operation on a particular element of the array. As well, some programs allow 0 as array location, and some don't.

Math and programming work together best when both paper and pencil and computer media are used. Before starting to work at a computer, students need to make sure that they understand and internalize the problem they are working on, as well as understand the program design—namely the algorithm they are about to use. Great care needs to be taken so that students learn how to prepare a math problem for exploration using programming.

Logical reasoning behind computer programming is very close to logical reasoning needed in daily work with mathematics. Thus, one enforces the other. However, there are small differences: many theorems in mathematics are if-then statements, but—unlike computer code—do not contain the else part (i.e., the "else" part in a theorem would always say "the statement of the theorem might or might not be true").

Perhaps the most important benefit of learning to program within math is the fact it provides ample opportunities to practice truly important skills for math, such as: posing one's own questions and formulating conjectures, experimenting, considering alternatives, and indentifying patterns.

Epilogue

This paper is based on my actual experiences in teaching the course, reflections on my teaching practice, and with the help of a dozen papers (some of which are listed in the *References* section). Although there is a general feeling that mathematics and computer programming go hand in hand, this interaction needs to be researched from many viewpoints, spanning the spectrum from theoretical to actual classroom practice.

References

Buteau, C., Muller, E., and Marshall, N. (2104a). Competencies Developed by University Students in Microworld-type Core Mathematics Courses. PME 2014, preprint.

Buteau, C., Muller, E., and Marshall, N. (2104b). Learning Mathematics by Designing, Programming, and Investigating with Interactive, Dynamic Computer-based Objects. International Journal of Technology in Mathematics Education, Vol 21, No 2.

Howson, A. G. and Kahane, J. P. (eds). (1986) The influence of computers and informatics on mathematics and its teaching, *ICMI Study Series* (Vol. 1), Cambridge, UK: Cambridge University Press.

King, K., Hillel, J., and Artigue, M. (2001) Technology – A working group report, in Holton, D. (ed) *The Teaching and Learning of Mathematics at University Level: An ICMI Study*, Dordrecht: Kluwer Academic Publishers, 349-356.

Kinnunena, P., and Simon, B. (2012). My program is ok – am I? Computing freshmen's experiences of doing programming assignments. Computer Science Education, Vol. 22, No. 1, pp. 1-28.

Ma, L., Ferguson j., Roper M., and Wood M. (2011). Investigating and improving the

models of programming concepts held by novice programmers. Computer Science Education, Vol. 21, No. 1, March 2011, pp. 57–80.
Rist, R.S. (1995). Program structure and design. Cognitive Science, 19, 507–562.

Robbins, A., Rountee J., and Rountree N. (2003). Learning and Teaching Programming; A Review and Discussion. Computer Science Education, Vol. 13, No. 2, pp. 137–172.