Caron, F., Floyd, S., & Lovric, M. (2023). Coding, Computational Modelling & Equity in Math Education Working Group B Secondary/University Report. In Online Proceedings of the *Coding, Computational Modeling, and Equity in Mathematics Education Symposium*, St. Catharines (Canada), April 2023.

Coding, Computational Modelling & Equity in Math Education Working Group B Secondary/University Report

WG leaders
France Caron, Steven Floyd, Miroslav Lovric

WG members

Chantal Buteau, Sarah Castle, Amenda Chow, Andy diSessa, Francis Duah, Wendy Forbes, Krista Francis, George Gadanidis, Bogumila Gierus, Oh Nam Kwon, Kehinde Ladipo, Brian Lawler, Gabriel Lecompte, Haobin Liu, Elise Lockwood, Neil Marshall, Ariane Masuda, Simon Modeste, Eric Muller, Mohammad Reza Peyghami, Elena Prieto-Rodriguez, Ana Isabel Sacristán, Jessica Sardella, Pam Sargent, Marie-Frédérick St-Cyr

Introduction

Mathematics is used in designing and coding algorithms (Modeste, 2016), and coding is often used to expand the class of mathematical problems that can be solved, or the set of concepts that can be explored (Buteau et al, 2020; Lovric, 2018). Mathematics and coding can also be used jointly with principles from other disciplines to model real-world situations (Giabbanelli & Mago, 2016; Caron, 2019).

In this working group, we examined coding and computational modelling in secondary and university mathematics by working with, and developing, hands-on activities for students and educators. We explored ready-made programs related to secondary and university mathematical concepts, along with tasks that have been, or could be, built from these programs, and then considered the characteristics of effective program-task combinations.

Important concepts, practices and ideas associated with computational thinking, computational modelling, mathematics, or computer science were identified in the effective tasks and activities. As we developed our ideas in discussing activities and teaching, we aimed at connecting them to, and contrasting them with, existing research and associated frameworks (Weintrop et al., 2016; Grover & Pea, 2017; diSessa, 2018; Modeste, 2018; Bråting & Kilhamn, 2021; Dohn, 2020). The intent was to identify key elements that could be made part of a foundation to create new activities, as well as envision improved integration of mathematics, coding, and computational modelling instruction.

We invited participants to bring, if they wished, their own samples of coding or programming tasks.

Participants and Prepared Activities

Prior to the working group meeting, a form was sent out to the working group registrants seeking background information. The data received through this form indicated that participants were a diverse group that included high school and adult education teachers, college and university instructors, researchers, and graduate students. When asked to describe their experience with coding and computational modelling, their responses ranged from limited and beginner to strong and "I have taught coding and computational mathematics". In terms of experience with specific platforms and languages, 52.9% of respondents had some or a lot of experience with the Scratch programming platform, while 63.2% had some or a lot of experience with the Python programming language. The responses from the form also indicated a variety of experiences with different programming languages and platforms including R, Matlab, Visual Basic, Java, SQL, Lego Robotics, C++, Mathematica, SAS, SPSS, Octave, and others.

In preparation for the Brock meeting, the facilitators created a set of over 25 mathematics and computer programming based activities, to serve as examples and starting points for discussions. Some of the activities addressed contemporary, real-world problems such as large language models and encryption in cyber security applications.

As well, the facilitators organized an excel file that categorized and summarized the various activities, so that the participants could access those that were most relevant to their work, and those that they were most comfortable with, in terms of mathematics and computer programming. In order for participants to decide which activities were most relevant to them, the excel file included information related to the programming language used, the mathematical topic and the approximate school level of mathematics where it could be used, as well as the levels of programming involved from L0 (simply observing the results of the running program) to L5 (designing and writing all components of the program) based on Broley et al.'s (2017) categories.

When selecting a particular activity, participants had access to more detailed information which included, as shown in the table below, the mathematical and computational concepts as well as some habits of mind associated with either discipline that could be mobilized in the activity. Many of these habits of mind have been identified in research papers (Weintrop et al., 2016; Grover & Pea, 2017; diSessa, 2018; Modeste, 2018; Bråting & Kilhamn, 2021; Dohn, 2020), which were made available to the participants; using them helped convey the idea that a single activity could benefit both mathematical thinking and computational thinking, while offering potential for connecting ideas and concepts from both disciplines.

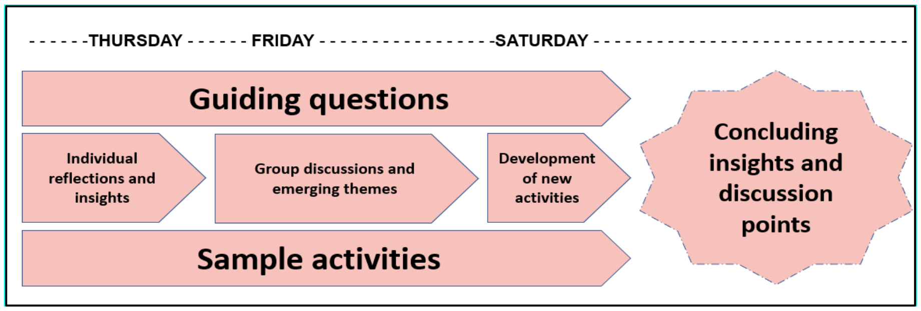
Example: Learning scope of the Linear Combination Activity

	Mathematical	Computational
Concepts	Vectors, points, translation Linear combination Linear independence Uniform distribution (probability)	Loop Random generator Variable types and identifiers Trace

Thinking / Habits of mind	Generalizing Explaining	Debugging Generating test cases Validating

The activities were developed using a variety of "computational tools", including Excel, GeoGebra, Insight Maker, Python, Scratch, and Excel. All activities that were developed included guiding questions and extension activities (or the next steps) so that participants could complete the initial activity, and also extend their thinking to see how the activity could be modified or enhanced.

The group met for five working sessions, from 90-120 minutes each, over the span of three days, and culminated in a final, sixth, session where the work was presented and summarized for all those in attendance at the symposium. The diagram below outlines the dynamics of our working group activities over the three days of the Symposium:


------THURSDAY ------ FRIDAY ----------------SATURDAY ----------------------------
Guiding questions
Individual reflections and insights
Group discussions and emerging themes
Development of new activities
Sample activities
Concluding insights and discussion points

Objects (and questions) to Think (about what) With

In Mindstorms, Papert (1993) discusses how physical gears were important in his early development as a mathematician, as these served as a way to help mathematics enter into his life at a young age. He explains how the gears served as models, carrying abstract mathematical ideas, and how they allowed for Piaget's (1952) notion of assimilation to take place, whereby new information fit into existing cognitive schemas within his mind. He describes these gears as "objects-to-think with".

When developing his physical, programmable turtle for students, which was controlled using LOGO programming code, Papert was attempting to create another "object-to-think with", that could serve the same role for students as the physical gears had done served for him: "My interest is in the process of invention of "objects-to-think-with," objects in which there is an intersection of cultural presence, embedded knowledge, and the possibility for personal identification" (Papert, 1993, p. 11).

In our working group, participants were provided with a number of small mathematics and programming activities that served as "objects-to-think with" during the three days we were together. The activities included a range of complexity in terms of mathematics and programming concepts, and they also were developed to present a variety of digital tools that can be used (e.g., python or scratch programming language, Insight Maker). The activities were presented to participants at the start of the first session, and participants had time to work with the activities, discuss them, and extend them. The following day, participants were invited to share their own activities, or to share how they extended provided activities.

Much like the gears of Papert's childhood, or his physical, programmable turtle, the activities in this working group served as "objects-to-think with" as they included an intersection of cultural presence, embedded knowledge, and the possibility for personal identification. Participants were able to manipulate the activities by altering the code, providing an opportunity for assimilation to take place, whereby this new information (mathematics, programming, or teaching related) could potentially fit into existing cognitive schemas within the participant's minds.

As an example, participants were provided with a short snippet of Python programming code related to the story of Anno's Seeds (Anno, 1995). This story, and associated computer program, is summarized and discussed in Gadanidis, Hughes, Namukasa, and Scucuglia (2019). The story involves Anno being given two magic seeds that, when eaten, can sustain him for a year. He decides to eat one of the seeds, and plant the other. A year later, the planted seed has produced two new seeds. He again eats one of the seeds, and plants the other, and repeats this year after year.

After writing and running the code that models this story, participants were then asked to alter the code, so that instead of eating one of the seeds at the start of the year, Anno eats one of the seeds after they have been planted and after they have yielded new seeds.

As participants worked in pairs or in small groups in altering the code, conversations took place that focussed on the syntax, purpose and components of each line of code (programing aspects), the potential growth in the number of seeds (mathematical aspects), and in how a program like this could be used to support student learning (pedagogy and andragogy aspects). Participants were also asked whether there were other mathematics stories, similar to Anno's seeds, that could be modeled with code.

As participants worked to alter and extend the Anno's seeds example, the conversations revolved around the small snippet of code itself, thereby allowing it to serve as an "object-to-think with". After all participants worked with the Anno's seeds example, they were then given the opportunity to explore over 15 other activities, each serving as a basis for conversations and learning, and ultimately each serving as an "object-to-think-with". The thinking did not limit itself to the mathematics involved. For instance, a participant created a sequence that could generate negative numbers of seeds, in which case she wanted the sequence generation to stop. Moving from mathematical thinking to computational thinking, and knowing that a go to statement is not the best for readability and adaptability, she created a while loop that checked for the negative value condition.

When considering how to best plan future coding and mathematics working groups or experiences for educators, we would recommend providing participants with coding and mathematics activities that serve as "objects-to-think with". We found that this facilitated conversations and discussions throughout the time together, as participants referenced specific code, mathematics, or instructional components from the activities. It also provided the group with a wide range of hands-on, learning activities upon which to build.

Case studies

We discuss two of the activities that the working group participants engaged with. In the *One-dimensional random walk* activity, we use simulation and computing to investigate a mathematical idea involving randomness, and to arrive at an important conclusion. In *Cryptography*, our mathematical knowledge of modular arithmetics guides us in creating and modifying a Python code that is used to encrypt and decrypt messages, thereby showing an application of mathematics in computer science.

Investigating one-dimensional random walk activity was designed to encourage and invoke questions about mathematics, which can then be verified using code (as is, or modified).

Brief description: A particle starts at x=0 on a number line. In each step, with equal chance, it either moves left or right for one unit (thus, after the first step, the particle is either at x=-1 or at x=1; if it is at x=1, in the next step it will move to x=0 or to x=2 with equal chance). Describe what happens after 2, 3, 4, …, n steps of this random walk.

The motivation behind this question is important. Albert Einstein used the random walk model (in two and three dimensions; also known as Brownian motion) to describe a diffusion process, and to convince "reluctant physicists to accept the existence of atoms" (*Annus mirabilis papers*, Wikipedia). In our model, the left-right movements of a particle are due to collisions with other particles.

A student or a teacher could suggest another situation where this model could be applied. Indeed, one of the members of our WG said: "The random walk example can be analyzed by students, and then they can develop a simpler program to simulate rolling two die. They can then graph the distribution of the rolls. All of the code needed is included in the random walks example. This is perhaps reverse engineering." (This remark was given in the context of an interesting suggestion: "We often provide students with a simple program, and then have them extend and build upon it. What about working the other way? Providing a relatively complicated program, and have them use the code within the program to build something relatively simpler.")

The activity sheet encouraged participants to work with a pencil and paper (or a notebook) first, before coding. This is an important step, as thinking about the problem will suggest important mathematical questions, such as: What is the farthest location from the origin a particle can be after 2, 3, 4, …, n steps? What is the range of the ending locations after n steps? Is there a pattern? Are all ending locations equally likely to occur? Moreover, deep(er) understanding of the mathematical problem at hand suggests a logical structure for the code to solve it (which is an essential step in coding!).

Once a student understands the code and starts using it, many doors of investigation open. The activity sheet suggests some: Pick a certain number of steps (say, 20) and run the simulation 10, 1000, and 10,000 times to obtain a distribution of ending locations. Is it what you expected it to be? Investigate what happens when you modify the probabilities (so that the particle is 'biased,' i.e., there is a higher chance of the particle going in one direction than in the opposite direction). Can you predict what the distribution will look like in this case? Modify the code appropriately and run the simulations to see if your prediction was accurate.

Here is the code that was given to our WG participants:

```
 1  # random walk in 1D
 2  # import relevant modules
 3  
 4  import matplotlib.pyplot as plt
 5  import random as ra
 6  %matplotlib inline
 7  
 8  # simulating one-dimensional random walk
 9  
10  numsteps = 6
11  
12  x = 0 # initial location
13  
14  for i in range(0,numsteps):
15      a = ra.randrange(1,3)
16      if a==1:
17          x=x+1
18      if a==2:
19          x=x-1
20      #x = x + (-1)**a      CHALLENGE - what does this line of code do?
21      print("time:", i+1,"location:",x)
22  
23  print()
24  print("The final location of particle after", numsteps, "steps is", x)
25  
```

The participants who engaged with this code were able to go through it and gain basic understanding; for instance, anything in a line that starts with a hashtag is a comment (documenting code is important, as it facilitates its understanding!) and is ignored by Python; certain modules (accepted here as black boxes) have to be imported, as they contain routines and commands that will be used; the word 'for' at the start of a line indicates the start of a loop, used to repeat certain actions; etc.). Certain commands needed to be explained; for instance, it is not at all obvious (nor logical) that ra.randrange(1,3) generates a random number from the set {1,2}, where one number is equally likely to be selected as the other. Once this is resolved, the sequence of two 'if' commands is clear.

An interesting math question is the following: can you make the code shorter, by removing the 'if; statements? That was the point of the challenge in line 20, which, unfortunately, our participants did not engage with.

Here is a sample run of the code:

```
time: 1 location: 1
time: 2 location: 0
time: 3 location: 1
time: 4 location: 0
time: 5 location: 1
time: 6 location: 2

The final location of particle after 6 steps is 2
```

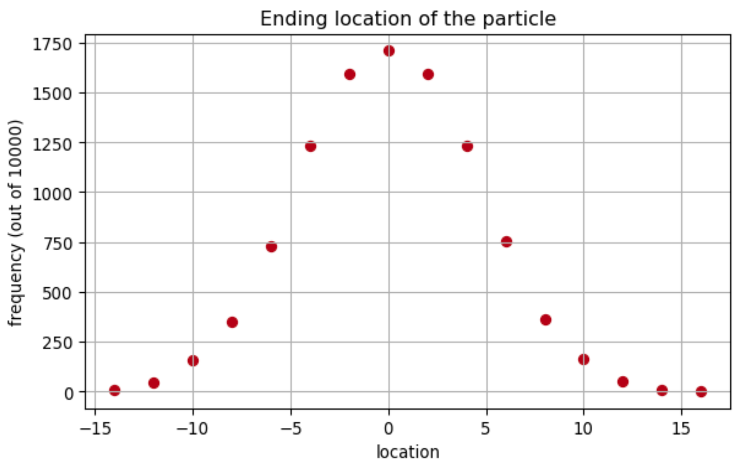
Of course, the most natural thing to do is to keep running the code and observing what happens. The next extension, suggested by the activity sheet, is to make this process automatic. At this point, a student has already seen one basic element they need in terms of coding, i.e., the loop. In our view, this is an important step in learning to code: you have seen something work, how can you use it again? One new element is needed: every time a random walk is run, we would like to record the final location. For this purpose, a list is used. How does a novice programmer learn that there are lists, and how to work with them? Here, we argue that some course instruction can help, or, a hint such as “look up lists in the Python manual that we have been using.”

Here is one way to automatise repeated random walks:

```python
# simulating multiple random walks, all with the same number of steps

numsteps = 25
numwalks = 20
xend = [] # to save final locations of the particle

for walk in range(0, numwalks):
    x = 0
    for i in range(0,numsteps):
        a = ra.randrange(1,3)
        x = x + (-1)**a
    xend.append(x)

print('Final locations of particle:', xend)
print('Farthest location on the right is', max(xend))
print('Farthest location on the left is', min(xend))
```

```
Final locations of particle: [11, 5, 5, -9, -1, -3, 1, -1, -11, 1, -3, 3, -5, -7, 7, -1, 3,
1, 3, -5]
Farthest location on the right is 11
Farthest location on the left is -11
```

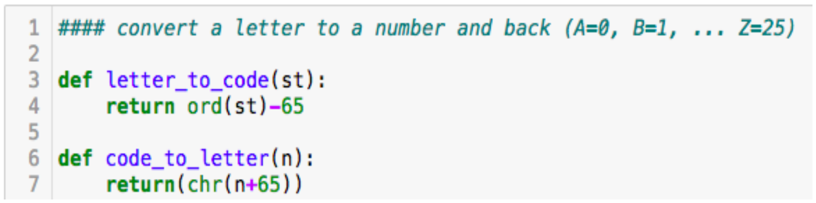
Final locations are recorded in a list, and then, with the help of statistics and graphing (initially accepted as black boxes, but later, as one learns more about coding, some of their features should be revealed) Python generates the following diagram of ending locations of 10,000 random walks with 20 steps each:



As a further extension (“wide walls”), we find a suggestion from a participant: “This code can be used to teach students about all systems in which a transmission from one state to another one is completely random. To add a bit fun to this topic, I would design a system for which the random walk can be examined in the class. For example, assume that seats are the states of a system and I ask everyone to run the code for a number of steps (let say 10). Assuming that the initial position is the student’s current seat, we set up some rules to change the seat based on the code’s output. For example, the movement in the simple case would be stepping forward (+) or backward (-). Then, we can ask them to identify their location at the end of the day. We can further change the design to add more randomness to this walk by for example alternating the movement between up/down and right/left." The last sentence suggests modelling a two-dimensional random walk!

The activity **Cryptography** was designed to engage students with a fun objective (to create and then to break a code), together with working on a mathematical topic of *modular arithmetic*. Basic *encoding* and *decoding* techniques were considered: translation (also known as the Caesar code) and encoding with a key, based on the arithmetic modulo 26. A unique number needs to be assigned to each letter of the alphabet (to simplify, we use uppercase letters). The most convenient way is this: A=0, B=1, C=2, …, Z=25. In Caesar code, each letter is moved the fixed number of locations in the alphabet to the right. Coding with a key (also known as Vigénère code) involves "adding" a letter and the corresponding value of a key.

As this activity is - in terms of coding - somewhat demanding, participants were given the entire code to start with (of course, they needed to code extensions and their own ideas). Several participants commented on the following piece of code, which establishes the mapping A=0, B=1, C=2, …, Z=25:



```
#### convert a letter to a number and back (A=0, B=1, ... Z=25)

def letter_to_code(st):
    return ord(st)-65

def code_to_letter(n):
    return(chr(n+65))
```

For instance: "I enjoyed exploring the Cryptography task. I would say that I tried to dig into the code a bit to understand it and explore it. I was at first unsure about the need to subtract 65 in the conversion between letters and numbers, but I googled (with the help of a partner) the ord() command to learn about Unicode and the fact that A is 65 in Unicode."

Another participant echoed this remark: ""Playing with the code was very fun and interesting. I understand why you give "ready-made" programs to work with, as many "tricky" things are used in some parts, that could be technical and difficult (and not very interesting) for students (examples: The chr(n+65) trick using ASCII codes; clearly it is quicker than creating a correspondence table or a converting function with 26 cases …)"

About stimulating thinking, and switching between coding and modular arithmetic, one participant wrote: "The task was a joy. It challenged me mathematically and with coding. I appreciated having to think in terms of modular arithmetic. [...] really helped me frame my thinking. I was fortunate to have two teachers [...] to help me with working back and forth between the mathematical concepts and translating them into coding a solution to the problem."

One participant commented on the virtues of working with a colleague (another important approach to learning in general, not just coding): " … and I discussed with a partner why the mod 26 was necessary and what it was doing. It was nice to discuss that with a partner and to think about where the letters are in unicode and why the mod 26 was necessary (we also discussed why the mod would work regardless of subtracting or adding some shift)."

Some participants added their own comments, etc, to make the code textbook-like. Some analyzed the underlying mathematics, drew a circle to represent the periodic behaviour when calculating modulo 26.

Considering and Combining Multiple Computational Approaches

As reflection of where we are (or where we appear to be heading) with respect to coding integration in mathematics education, there was a keen interest from most participants in engaging in math learning activities with Python. About half of the participants had coding experience with it; those who did not had an opportunity to learn some of its elements through our activities. We thus had a substantial amount of activities in Python.

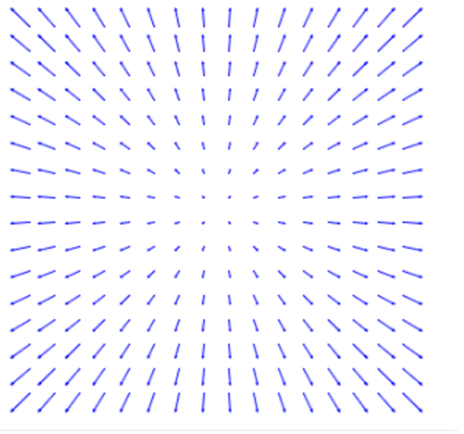
As we wanted to enlarge the discussion to other programming environments, derive similarities, point to specific uses and strengths, and even revisit what we mean by coding, we included activities that made use of Scratch, Excel, GeoGebra and Insight Maker. In the same way that multiple representations have been valued to learn mathematics, we aimed at showing the complementary contribution of multiple computational approaches for looking into a common situation or problem.

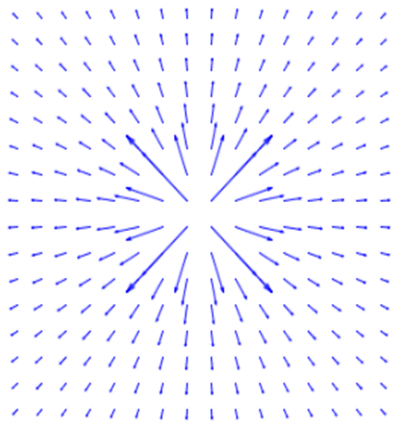
One of the key computational concepts is iteration (or loop), which can be linked quite naturally to the mathematical concept of a sequence, which itself can be connected to the concept of a function, as a discretized version of it. This was brought to light with the first introductory Python activity (Anno's seeds), where a simple change in the order of operations led to creating two very different sequences (and functions). The generation of these sequences leads to data structure consideration and thus makes us move to a type of thinking that is more computational than mathematical.

The first version of the code had each sequence generated within a single scalar variable where a value, once generated and printed, would be replaced with the next value of the sequence (with the troubling-at-first use of the equals sign in assignment instructions such as x = x + …). Graphing a sequence however requires keeping in memory, and in the right order, the different values of the terms of the sequence. As could be seen in the second version of the code, this can be done by generating two lists: one for the ranks (corresponding to the years here), and one for the values of the corresponding terms in the sequence. Such lists can be dynamically extended by appending new elements. A simple call to the plot function of the pyplot module, with those two lists as parameters, will generate the graph. Format options can be specified.

List generation by appending successive items can also be done with Scratch. However, graphing in Scratch requires to manage at a lower level the association between the rank and the value of its associated term (or between the x and y values of a function) and to direct the software to "stamp" a "dot" for each of these pairings in the 2D cartesian space provided (or to move with the «pen down» between two "consecutive" points of a function). This may in turn require scale conversion to adjust the target graph to the available space. In an interesting way, programming in Scratch may help unpack the black box that graphing can be to students and have them appreciate at a deeper level both the connection between a table of values and a graph and the necessary discretization and iterative process for graphing a function. As testified by one participant, young students may spontaneously elect to "look inside": given a graphing tool, they may prefer to figure out how to store the data of a graph and use the idea for their own work.

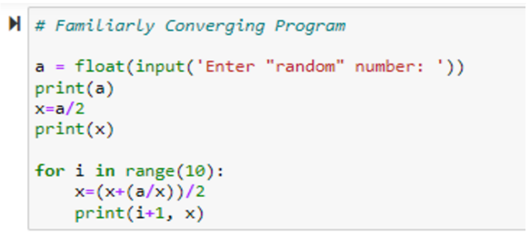
As was suggested with one of our activities, the approach can be extended to graphing vector fields, with embedded loops for running through a discretized 2D space. Given the Scratch code that produced the first figure below (on the left), a team of participants challenged themselves to produce a vector field that could represent a repulsive force whose magnitude decreases as distance increases, as shown in the second figure on the right.



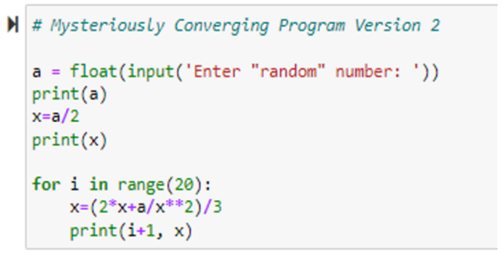


Producing the code for generating similar arrows (where the length and the head size are proportional to the norm of the vector) that will point in the right direction can be an interesting exercise of vector geometry and linear algebra, which can be done without requiring angle computation and use of trigonometry. Prototyping it in GeoGebra (or having to explain a ready-made GeoGebra construction protocol for that purpose) can prove a useful preparatory step. It may also suggest that such a sequence of construction steps, applicable to an arrow of any size and direction, could be considered a form of code.

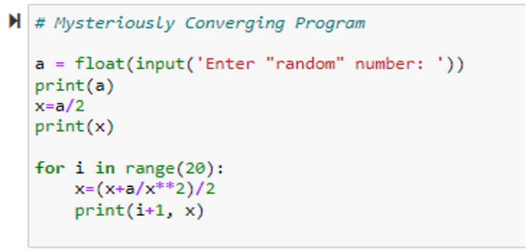
Building on the ideas of iteration and sequences, a set of four short programs had some participants move into the realm of numerical methods, through exploration of converging sequences defined by recurrence relations.



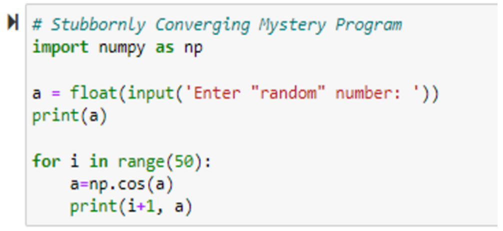
```
# Familiarly Converging Program

a = float(input('Enter "random" number: '))
print(a)
x=a/2
print(x)

for i in range(10):
    x=(x+(a/x))/2
    print(i+1, x)
```



```
# Mysteriously Converging Program Version 2

a = float(input('Enter "random" number: '))
print(a)
x=a/2
print(x)

for i in range(20):
    x=(2*x+a/x**2)/3
    print(i+1, x)
```



```
# Mysteriously Converging Program

a = float(input('Enter "random" number: '))
print(a)
x=a/2
print(x)

for i in range(20):
    x=(x+a/x**2)/2
    print(i+1, x)
```



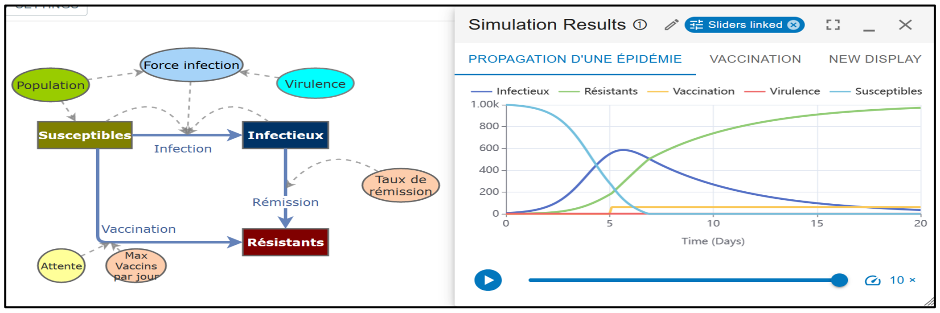
```
# Stubbornly Converging Mystery Program
import numpy as np

a = float(input('Enter "random" number: '))
print(a)

for i in range(50):
    a=np.cos(a)
    print(i+1, a)
```

By running each program a few times with different input values, participants could try to infer the connection (if any) between that input value and the number towards which the output sequence appears to be converging. As these sequences are all defined by recurrence relations of the form $x_{n+1} = f(x_n)$, despite being written as $x = f(x)$, the key strategy for finding and explaining the limit of the sequence is actually to express (and possibly simplify) the solution of equation $x = f(x)$. Unveiling what these programs do, and mainly how they do it, allows students to envision algorithms that can serve as viable alternatives to the black boxes associated with magical function keys of their calculator (or preprogrammed functions of a programming language) such as square root. Running the two versions of the "mysteriously converging program" shows two algorithms converging at different speeds to the same limit, and can open to the concept of rate of convergence. The "stubbornly converging mystery program" can serve as an entry point into cobweb diagrams.

In playing with these programs, one participant raised a highly relevant question: "What is the added value of Python here when compared to Excel?" There may not be a single answer to that question, as it may very well depend on what the students already know, the learning objective and the connections that can be made in moving from one environment to the other. Despite the smaller distance with mathematics that the Python syntax and reference to variables may appear to have when generating sequences, the access to the explicit definition of each cell in Excel and the ease of replicating relative references may provide a more concrete handle at first, from where to build generalization and understand the idea of iteration.

Maintaining the possibility of comparing affordances for learning, participants were invited to explore dynamical complex system simulations, epidemics in particular, with different environments: Insight Maker, NetLogo, Scratch, Excel and Python.

Like similar software, such as Stella and Vensim, Insight Maker supports a system dynamics approach to modelling complex systems, with an icon-base interface for defining stocks and flows. Stocks are used to represent variables (e.g. number of infected persons) that accumulate over time (as integrals) and are given an initial value, while a flow changes a stock at every time step, either by adding to the stock (inflow) or subtracting from the stock (outflow). The flow (e.g. the rate of infection) corresponds to a rate of change (or derivative) that can evolve over time, depend on other variables, and be defined with functions and even, as is the case with Insight Maker, with programming instructions. For example, in the model below, the vaccination flow (that has people move directly from the susceptible to the resistant group) has been modeled crudely with a conditional statement that assigns it a constant daily value if the number of days since the beginning of the epidemics is greater than the waiting period (before a vaccine becomes available and effective) and leaves it to zero if not. Students can be invited to think of more sophisticated models that could better reflect reality.



As has been observed in studies with students (Doerr, 1998) and with our participants, modelling with the stock-flow approach can be unsettling at first. In order to facilitate the understanding of the hydraulic metaphor behind this approach, a team of participants envisioned an activity where students would describe, compute and compare the amount of water used in a shower and a bath. Spending time to model and implement a filling bathtub (and envision a way to avoid overflow with a conditional statement) has also been found to be more than useful in making secondary school students experience a first success with Insight Maker and be more open to debugging when a modelling mistake or syntax error prevents their model from functioning adequately (Gonczi et al. 2022).

Once students have made sense of the stock-flow dynamics, it can help them expand radically the class of situations and scenarios that they can model and simulate. Although it may not be perceived that way by students that learn to model with it in their secondary science class, the approach provides a visual representation of a system of differential equations, which can be solved (or numerically integrated) using the simple Euler's method. This is the black box upon which the simulation is based and that must be opened at least once for students to maintain control over such software. This can be done by using Excel or Python to implement the simple numerical scheme to generate the sequences associated with the evolution over time of each of the variables. Examples of that were provided to the participants. One of them commented: "I can see uses for this in my first year university calculus course with connections to derivatives, especially related rates and introduction to differential equations. Of course it is also very useful for a differential equations course."

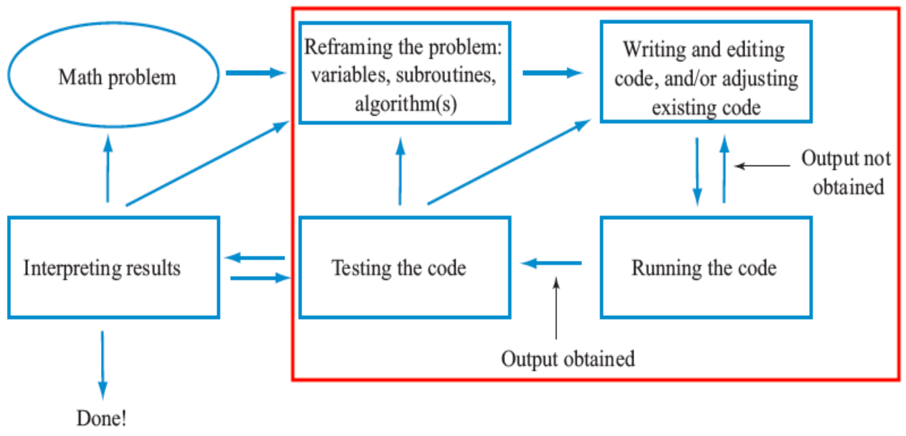
Dynamical complex systems can also be described with agent-based models (Caron, 2019). In such models, agents are programmed to move in a space, interact together, and change state depending on their interactions and the time elapsed. Elements of randomness are typically included in such descriptions. NetLogo is a multi-agent programmable modelling environment, designed to be "low threshold and no ceiling". It has been used with students, and is increasingly being used by scientific researchers with a limited programming background. Scratch and Insight Maker also support agent-based modelling. Short programs in each of these environments were offered to participants to experiment with agent-based computational models (of epidemics and flocks of birds), to see the effects of different parameters and infer the meaning of the coding instructions in those environments. Those who tried them felt that it was rather difficult to understand the modelling code in a programming language that they did not know, even with the possibility of comparing with a similar program in another language.

The fact that a model is often built on principles that come from a discipline other than mathematics and computer science certainly adds to the difficulty. We recognize that learning math, modelling, and programming at the same time involves a steep learning curve and that teaching should try to provide appropriate scaffolding. Yet, one should not neglect the inherent difficulty of adapting to a new programming language or paradigm (e.g. from procedural to object-oriented programming). This was also expressed in our introductory activity by a participant who was new to Python and the coding environment that we used.

Python notebooks as a coding environment

There is a wide variety of platforms available for coding, varying from the ones for beginners to those for professional coders. The platforms used for educational purposes are often easy to access, and straightforward to use. For Python, a convenient way is to use notebooks. One such setup is Jupyter notebooks (https://jupyter.org/), made available to researchers and students at Canadian institutions through the syzygy.ca service, created in collaboration between PIMS, Compute Canada and Cybera in 2017. A notebook is a collection of cells, where each cell can be executed on its own.

Perhaps the most appealing feature of the notebooks is that we can create, test, and run small parts of a code in one cell, and when we are sure that the code is working, we can move it to another cell where (often) a larger code is built. In terms of a coding model (Lovric, 2018; p. 689), working with individual cells within a notebook facilitates the important steps in coding, in particular the ones within a red rectangle:


Math problem
Reframing the problem: variables, subroutines, algorithm(s)
Writing and editing code, and/or adjusting existing code
Output not obtained
Interpreting results
Testing the code
Running the code
Output obtained
Done!

One of our WG participants echoed this comment: **"**I also played around with just doing one of the programs at a time (in particular, being able to enter the encoded text and decoding it, and that presented some interesting opportunities for debugging)."

Another participant asked: "One of the questions that I had was about the jupyter notebooks we were given. Were these originally separate scripts?" The answer is - yes, originally these were separate scripts. Whenever faced with writing a complex code, a good way to break it down into simpler, smaller parts, and write a code for each part (modular approach). The most important reason why we do this is testing: if we write the entire code and Python returns an error, or the output is not correct, it is often not easy to find the source of the problem.

Notes: The values defined in one cell are "remembered" by Python until they are altered (by another assignment command) somewhere else in the notebook. Once a notebook is closed, all values are forgotten. A fully functioning piece of code (a module, created in a cell) can be (and often is) used, with confidence, in coding other problems. Python allows for comments (either inside a cell with code, or in a separate, text-only cell), which is very important, as we need to document the code we're working with.

Considerations related to how much code is provided

In teaching coding we need to make important decisions, including answering the following questions: Should we provide code (in exercises, assessments), and if so, how much of it? What pieces of code should we include? What do we expect our students will do with that code? In the absence of research results about these questions, we rely on our personal/professional, classroom, and other experiences.

Perhaps the best answer is - how much (if any) code is revealed (given to students) depends on the situation (topic(s) covered, teaching approach, pedagogical considerations, and so on). If we wish to provide a "low floor" approach, and/or work with novice coders, giving some code is a good idea. For instance, it could be a functioning code that needs to be refined and/or modified, or several starting lines of a code which define the variables or suggest a logical structure.

Seeing sample code helps students with the non-intuitive, confusing at-first elements of coding (and an initial barrier to learning!), such as learning what the names of commands are, what the commands do, and the syntax. For instance, to ask Python to show the graph of a function, we need to use plot rather than draw; the command range(0,4) generates the list of integers 0,1,2,3 which includes zero but does not include 4; a list is indexed starting from 0 (i.e., we refer to the first element in a list L by L[0] and not L[1]); a code will not work if the for statement opening a loop does not end with a colon).

Providing code (as well as appropriate comments documenting the code) helps our students learn from professionals, and do as professionals often do, which is a productive way to code. When we teach proofs in mathematics (perhaps the most difficult concept for students to learn), we start by showing to our students how proofs work, and explain each proof step-by-step. We direct their attention to the logical structure of a proof, to the narratives that are used to help the reader follow the argument, and so on. This is a challenge to absorb at once, it takes time to master. As students advance through their studies, they slowly start to create their own proofs, first with assistance, and later on their own. Learning to code is, in many ways, like learning proofs (or mathematics in general). This process is mimicked in the so-called PRIMM (predict-run-investigate-modify- make) approach to learning to code, where a learner moves from a code which is not theirs (predict, run, and investigate), to the one which is partially theirs (modify), to the one that they create (make).

On the other hand, our anecdotal evidence suggests that providing code can be counterproductive, as it might obscure the coding process and reasoning. For instance, a student might miss a reason why a certain variable was coded as a list, or why a certain loop was used to implement an algorithm. Similarly, a math proof that starts by announcing a method used ("We will prove this statement by contradiction"), does not give an opportunity for a student to figure out why that particular method is used.

On this theme, our participants offered their own thoughts and asked questions:

- "We often provide students with a simple program, and then have them extend and build upon it. What about working the other way? Providing a relatively complicated program, and have them use the code within the program to build something relatively simpler."
- ""Also with the examples a lot of these have us modifying the coders initial thoughts. But why couldn't students first start with the problem such as cryptography and the caesar key and go through the process of building up this formula?"
- "But clearly, we see that this very simple cryptographic principles, Caesar and Vigénère codes, are not so easy to implement and you really get confronted with programming issues in order to implement them, and from a blank page, students would be in difficulty. But, at the end, creating a cryptography program from scratch is what we would like students to be able to do, isn't it?"

  In summary, there are two types of decisions we need to make when teaching coding:
- Code: Do students begin with a ready made code? Do we encourage them to copy and paste pieces of code, or write their own? Or, do we give them a starting part of a code, where we define the variables to be used and suggest the approach? Or, do we start with an empty cell/ empty notebook?
- Complexity: How do we stimulate students to break up a complex problem into smaller, easier-to-code pieces (modular approach)? Does it make sense to give to students a relatively complex program, and then use parts of the code to build something simpler?
- Of course, these decisions are of dynamic nature - they need to be made or modified on a daily basis, depending on the students' progress, and the content which is taught.

Conclusion

Integrating coding in the learning of mathematics opens possibilities for exploring, understanding and modelling, but it does require addressing and resolving tensions that may arise when making decisions on the programming language, the coding environment, and the code made available. As time is a limited resource in any class, the time spent on learning a new language or even in writing original code may appear as time taken away from learning mathematics. This (and the impression that it was probably hard to convince math instructors that coding has a power to enhance the understanding of mathematics) is why programming in the math classes had been progressively replaced in the 90s by user-friendly applications. Although such applications have provided new means for teaching, learning and doing mathematics, they often came as black boxes with a limited scope of use.

Learning to program can help open these black boxes and extend the scope of application of the mathematics we learn. To fully reap such potential, we may have to consider not only *coding for mathematics* as another way of learning mathematics, but also *coding with mathematics* for modelling and understanding real-world situations. This might bring together more mathematicians, computer scientists and others scientists in helping address the current challenges of our world.

References

Anno, M. (1995). *Anno's seeds.* Paperstar Book (UK).

Bråting, K., & Kilhamn, C. (2021). Exploring the intersection of algebraic and computational thinking. *Mathematical Thinking and Learning*, 23(2), 170-185. https://doi.org/10.1080/10986065.2020.1779012

Broley, L., Caron F., & Saint-Aubin, Y. (2018). Levels of Programming in Mathematical Research and University Mathematics Education, *International Journal of Research in Undergraduate Mathematics Education*, 4(1), 38-55 https://doi.org/10.1007/s40753-017-0066-1

Buteau, C., Gueudet, G., Muller, E., Mgombelo, J., & Sacristán, A. I. (2020). University students turning computer programming into an instrument for 'authentic' mathematical work. *International Journal of Mathematical Education in Science and Technology*, 51(7), 1020-1041. https://doi.org/10.1080/0020739X.2019.1648892

Caron, F. (2019). Approaches to investigating complex dynamical systems. In *Lines of inquiry in mathematical modelling research in education* (pp. 83-103). Springer, Cham. https://link.springer.com/content/pdf/10.1007/978-3-030-14931-4_5

diSessa, A. A. (2018). Computational literacy and "the big picture" concerning computers in mathematics education. *Mathematical thinking and learning*, 20(1), 3-31. https://doi.org/10.1080/10986065.2018.1403544

Doerr, H. M. (1996). Stella ten years later: A review of the literature. *International Journal of Computers for Mathematical Learning*, 1, 201-224.

Dohn, N. B. (2020). Students' interest in Scratch coding in lower secondary mathematics. *British Journal of Educational Technology*, 51(1), 71-83. https://doi.org/10.1111/bjet.12759

Gadanidis, G., Hughes, J. M., Namukasa, I., & Scucuglia, R. (2019). Computational modelling in elementary mathematics teacher education. In: *International Handbook of Mathematics Teacher Education: Volume 2* (pp. 197-222). Brill.

Giabbanelli, P. J., & Mago, V. K. (2016). Teaching computational modeling in the data science era. *Procedia Computer Science*, 80, 1968-1977. https://doi.org/10.1016/j.procs.2016.05.517

Gonczi, A. L., Palosaari, C., Urban, N., & Mayer, A. (2022). From Simulation User to Creator: Helping Students See Inside the "Black Box" with Insight Maker. *The science teacher*, 89(3).

Grover, S., & Pea, R. (2017). Computational thinking: A competency whose time has come. In S. Sentance, E. Barendsen, & C. Schulte (Eds.). *Computer science education: Perspectives* *on teaching and learning* (pp. 19–38). Bloomsbury Academic. https://10.5040/9781350057142.ch-003

Lovric, M. (2018). Programming and Mathematics in an Upper-Level University Problem-Solving Course, *PRIMUS*, 28 (7), 683–698. https://doi.org/10.1080/10511970.2017.1403524

Modeste, S. (2016). Impact of Informatics on Mathematics and Its Teaching. In: Gadducci, F., Tavosanis, M. (Eds). *History and Philosophy of Computing. HaPoC 2015. IFIP Advances in Information and Communication Technology*, vol 487. Springer, Cham. https://doi.org/10.1007/978-3-319-47286-7_17

Modeste, S. (2018). Relations between mathematics and computer science in the French secondary school: a developing curriculum. In: I*CMI STUDY 24*. https://hal.archives-ouvertes.fr/hal-03184712/document

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25(1), 127–147. https://doi.org/10.1007/s10956-015-9581-5