A Preliminary Study on Teaching Programming Through Physics: Development of a C# Code Library on Classical Mechanics

Lance Owen C. Gulinao[1], Emilio B. Marmol[1], Gabriel Paulo A. Rayo[1], Aedan Red E. Tiu[1], and Shirley Chu[2,*]

[1]De La Salle University Integrated School, Manila, Philippines
[2]De La Salle University, Manila, Philippines

*Corresponding Author: shirley.chu@dlsu.edu.ph

ABSTRACT

Current programming pedagogy mostly revolves around the concept of syntax first. This can result in students lacking the ability to form detailed mental models on programming. A problem-solving-first method is suggested, wherein students can "imagine" solutions before writing one. Consequently, this makes classical mechanics a possible education medium for programming, as classical mechanics is a neurologically intuitive science. However, abstraction is needed to allow students to write code that solves classical mechanics problems. One possible solution is through code libraries. Currently, there are few classical mechanics code libraries available; thus, the researchers created a code library implementing concepts such as gravity, thrust, and torque. Grade 12 senior high school students were invited to participate in the study. Due to the work requirement of the study and restrictions caused by the COVID-19 pandemic, only a few people responded to the invitation. The participants were asked to construct programs to solve given classical mechanics problems using the provided code library. Through their insights and submitted source code, a qualitative approach was used to test the hypothesis. Results mostly support the suitability of classical mechanics as programming pedagogy; however, due to the limited participant pool, further reinforcement of the idea may be done by reimplementing the research on a larger scale of participants and a more thorough experimental design involving a control group to concretely ascertain if exposure to the code library improves one's programming abilities.

Keywords: education, computer programming, programming pedagogy, code library, classical mechanics

INTRODUCTION

In today's age of human exploration and development, technology has been at the forefront of leading global change and innovation. It comes together in providing ease and comfort to the lives of the everyday man. Computer programming plays an integral role in the development of economy, healthcare, and infrastructure among others. In 2019, Fedorenko et al. explained that computer programming has developed into a crucial skill in various fields and disciplines.

Considering this, programming education is therefore in high demand, and efforts to reevaluate teaching methods have been made in recent years. The current methods of programming pedagogy still focus on syntax (Tran & Torrisi-Steele, 2022). This method of teaching programming has its disadvantages (Robins et al., 2003). Mainly, syntax-first instruction tends to produce programmers that struggle to translate their knowledge of syntax into meaningful program structures. Additionally, the resulting students exposed to this pedagogy tend to use the imperative approach in programming. Some methods have been devised to abstract syntax from problem-solving, allowing students to focus on the program structure. One such method is block-based programming. However, as Allen et al. (2022) have mentioned, it is possible that students will experience a loss of confidence when eventually switching towards syntax-heavy programming.

Deek et al. (1998) suggest that a problem-solving-first method is used instead. This method demands students to come up with solutions first, and only then will relevant syntax be taught in the context of said solutions. In this method, emphasis is placed on students' formation of detailed mental models prior to writing code. In other words, students must be able to "imagine" a solution before they write one. Robins et al. (2003) further suggest that the most optimal teaching method for novice programming is one that focuses not on the features of the programming language but on the application and combinations of these features in relation to a problem. Therefore, there is value to be found in teaching programming through physics, particularly classical mechanics.

Classical mechanics, or Newtonian mechanics, is a field of physics that deals with the three basic laws of motion. It has been continually discussed that classical mechanics is intuitive and easily learned by empirical means. One such proof is the ability of a human eye to predict movement (Fischer et al., 2016). Athletes, as an example, continually practice their ability to predict movement, as per classical mechanics, without needing to calculate using any formula.

Therefore, instead of completely abstracting syntax from students, programming pedagogy can revolve around solving physics problems that students can already "imagine" the solution to. However, writing physics simulations is another difficulty entirely. It is only the essence of intuitive classical mechanics that is beneficial for instruction. This area is where abstraction should be applied. This can be done using code libraries. Kumar (2020) described code libraries as methods for locating existing resources without having to construct from scratch. Additionally, Xie et al. (2019) stated that code templates can bridge the gap between syntax learning and problem-solving using abstractions of programming knowledge, which is similar to how code libraries operate. Similar to physical libraries, code libraries abstract information that is not relevant to the user. They ease workload and improve time efficiency and quality by effectively storing relevant functionality.

Although there are already existing code libraries available online such as on the Astrophysics Source Code Library (ASCL), libraries dedicated to classical mechanics concepts are not publicly available. Furthermore, there are little to no code libraries that focus on abstracting functionality for instructional purposes.

As such, this research focuses on exploring the possibility of using classical mechanics to bridge syntax learning and problem-solving, whilst providing a classical mechanics code library to be used as a learning tool targeted towards beginner programmers.

MATERIALS AND METHODS

The Code Library

The goal of the code library is to allow beginner programmers to simulate physics in hopes of training their programming skill; thus, the code library is constructed in a way that is usable in an educational level of programming. Additionally, the code library was designed with intuitiveness in mind. This is to adhere to Deek et al.'s (1998) methods in which the syntax should only be introduced in the context of a problem that needs it. The code library was designed in a way that the syntax matches with what is required to be solved, therefore ensuring that language/syntax features (i.e., loops and lists) are not introduced unnecessarily early into a problem-solving process. This is done in hopes of guiding the student's learning process through the methods explained by prior research. Furthermore, the code library aims to be modular to allow beginner programmers to easily dissect and understand how the code library functions. The code library intends to serve as a model on what "clean and maintainable code" should look like.

The concepts of classical mechanics that are implemented were chosen due to their usefulness and intuitiveness. Gravity, thrust, and torque were implemented as they are the most useful concepts for beginner programmers to work with and since these are the concepts that will be primarily used for most basic applications that utilize the code library. Aside from the aforementioned concepts, collision detection was also added as it is generally useful when dealing with movement. C# will be utilized as the programming language; for that reason, it can be imported into any C# code project and into any Unity Game Engine project. Furthermore, the code library in this research is distanced from other libraries that are made to be used in a professional context. Instead, it merely functions as a pathway for beginner programmers to understand abstract programming concepts.

To achieve the desired modularity of the code library, it revolves around the extension of a base class such that each subclass will only contain the methods it needs. To organize and prevent the code library from interfering with other libraries, all its code is stored inside the **SpaceSimulation** namespace. Thus, to use the code library, one could either prepend the namespace or import it into the file by adding a using directive.

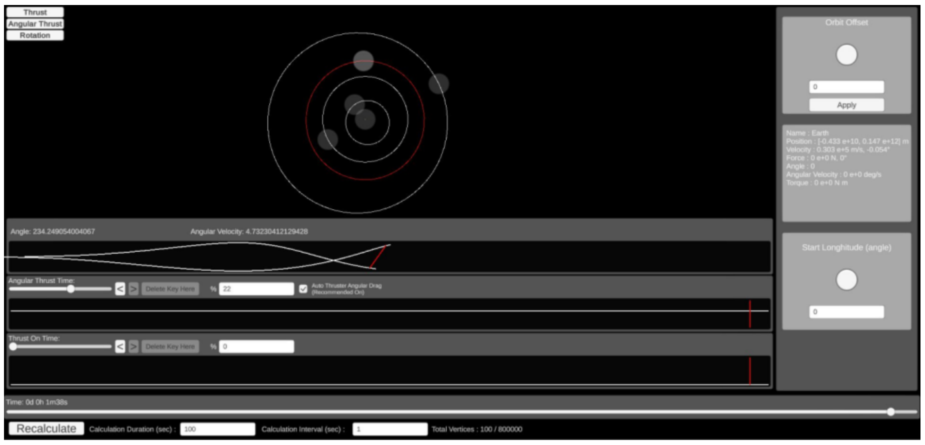
- **TrajectoryData** struct—The TrajectoryData struct contains various information about the trajectory of an object at a given time.
- **TrajectoryBody** class—The TrajectoryBody class represents any object in space and thus contains all the methods and properties common to all objects. It is an abstract class, meaning it cannot be instantiated;

rather, a subclass must inherit from it so that it can be instantiated.

- **ThrustBody** class—The ThrustBody class is a concrete class that extends from the TrajectoryBody class. It represents any object in space that can move under its own influence (or thrust).
- **CelestialBody** class—The CelestialBody class is a concrete class that extends from the TrajectoryBody class. It represents any celestial object in space.

Code Library Usage

The code library was meant to be used mostly by beginner programmers; special attention was given to minimize the amount of code and syntax that they will have to write to perform relatively trivial tasks. As such, both a high-level and low-level application programming interface (API) were implemented in the code library through method overloading such that they could use the high-level API, which abstracts the complexity of the low-level API. Aside from applying abstraction, documentation for both the high- and low-level APIs was also provided to aid users in understanding the code library and developing their own applications. One such application is an orbital rocket simulator, which was made by the researchers as a proof of concept or as a showcase of the code library's capabilities.



Name : Earth
Position : [-0.433 e+10, 0.147 e+12] m
Velocity : 0.303 e+5 m/s, -0.054°
Force : 0 e+0 N, 0°
Angle : 0
Angular Velocity : 0 e+0 deg/s
Torque : 0 e+0 N m

Angle: 234.249054004067 Angular Velocity: 4.73230412129428

Angular Thrust Time: % 22 ✓ Auto Thruster Angular Drag (Recommended On)

Thrust On Time: % 0

Time: 0d 0h 1m38s

Calculation Duration (sec) : 100 Calculation Interval (sec) : 1 Total Vertices : 100 / 800000

Figure 1. Orbital rocket simulator.

The process of creating physics simulations is abstracted through the code library in the following general steps:

1. Users can import the **SpaceSimulation** namespace into the file using either of the following methods:
    a. through using a directive or
    b. by calling the classes and structs directly through the **SpaceSimulation** namespace keyword.
2. Create the **TrajectoryBody** instances that are to be included in the simulation.
    a. Generate the initial trajectory data of the object.
        i. Instantiate a new **TrajectoryData** instance <with the status of the object that will be used>.

   b. To instantiate a **ThrustBody**:
      i. Generate the linear and angular movement keys of the **ThrustBody** by creating a new double array and populating each element with the thrust percentage at a given second.
      ii. Call the **ThrustBody** constructor method and pass in all the required parameters.
   c. To instantiate a **CelestialBody**:
      i. Call the **CelestialBody** constructor method and pass in all the required parameters.
3. Combine all the generated **CelestialBody** instances in a single array.
4. Calculate the next **CurrentTrajectoryData** of the **ThrustBody** using the **CalculateNext()** method.
   a. Define a loop that calls the **CalculateNext()** method for every movement key defined.
   b. Optionally, print the calculated **CurrentTrajectoryData** by calling its **PrintToConsole()** method.
5. If needed, the user can fetch data from the computed trajectory by any of the following:
   a. fetching the entire **TrajectoryList** of the object by calling the **GetTrajectoryList()** method or
   b. fetching a specific piece of data at a specific moment in time using the abstract getter methods.

It is through these five primary steps that the code library attempts to ease students into learning only the required syntax that is required in the context of a problem, therefore adhering to the methods of Deek et al. (1998).

```
using System;
//Step 1
using SpaceSimulation;

namespace MyDemonstrationProgram
{
        class Program
        {
                static void Main(string[] args){
                        //Step 2.a
                        Double2 startingPos = new Double2(0, 0); // position of the rocket
                        TrajectoryData startingTrajectoryData = new TrajectoryData(10000, startingPos, Double2.Zero,
45, 0);
                        //Step 2.b.i
                        double[] thrustKeys = { 0, 0, 0, 0, 0 };
                        double[] angularThrustKeys = { 0.1, 0.0, -0.2, 0.0, 0.1 };
                        //Step 2.b.ii
                        ThrustBody rocket = new ThrustBody(startingTrajectoryData, thrustKeys, angularThrustKeys, 1,
1000, 1000);
                        //Step 2.a
                        Double2 planetStartingPos = new Double2(0, -10000); // position of the planet
                        TrajectoryData planetTData = new TrajectoryData(7.5 * Math.Pow(10, 20), planetStartingPos,
Double2.Zero, 0, 0);
                        //Step 2.c
                        CelestialBody earth = new CelestialBody(planetTData, 4000);
                        //Step 3
                        CelestialBody[] planets = { earth }; // create the planets array with only one planet as the
item
                        //Step 4.b
                        Console.WriteLine("Initial TrajectoryData: ");
                        startingTrajectoryData.PrintToConsole();
                        //Step 4.a
                        for (int currentTime = 0; currentTime < thrustKeys.Length; currentTime++) {
                                Console.WriteLine("Currently on iteration: " + currentTime);
                                rocket.CalculateNext(planets);
                                rocket.CurrentTrajectoryData.PrintToConsole();
                        }
                }
        }
}
```

```
Initial TrajectoryData:
Velocity: [0, 0]
Position: [0, 0]
Angle: 45
AngularVelocity: 0

Currently on iteration: 0
Velocity: [0, -500.5725]
Position: [0, -500.5725]
Angle: 732.549354156988
AngularVelocity: 687.549354156988

Currently on iteration: 1
Velocity: [0, -1055.29032561037]
Position: [0, -1555.86282561037]
Angle: 1420.09870831398
AngularVelocity: 687.549354156988

Currently on iteration: 2
Velocity: [0, -1757.32154795294]
Position: [0, -3313.18437356331]
Angle: 732.549354156988
AngularVelocity: -687.549354156988

Currently on iteration: 3
Velocity: [0, -2876.83235325745]
Position: [0, -6190.01672682075]
Angle: 45
AngularVelocity: -687.549354156988

Currently on iteration: 4
Velocity: [0, 0]
Position: [0, -6190.01672682075]
Angle: 45
AngularVelocity: 0
```

Figure 2. Sample console application program and output.

Experimental Design

A qualitative experimental approach was used to evaluate the code library as a learning tool. A group of seven senior high school students responded to the invitation to participate in the study. Chosen participants had no external programming expertise other than what was taught to them by the required curriculum.

The participants were subject to take a preliminary form containing a diagnostic test. After two days, an orientation meeting with the seven participants was held. The meeting was held online through Zoom meetings. In this meeting, a brief introduction about the research was given, followed by a tutorial on the installation of the code library and simulator. An overview of the documentation was given, so that the students had a grasp of the basic syntax of C# to be able to use the code library's API. It was in this meeting that the students were assigned one individual task each that they needed to complete by the creation of a C# program.

Table 1. Task Pool

Task #	Task
1	Suppose a rocket whose initial position is at the origin and a planet whose initial position is 10,000 m to the right of the rocket. The rocket has a mass of 7.5 * 10^20 kg and a radius of 4,000 m. Determine the position of the rocket and if it has collided with the planet after 6 seconds if the rocket has no thrust for the entire duration.
2	Create a rocket whose initial position is at the origin, if the rocket has 100% thrust for 7 seconds and mass flow rate of 2,200 kg/s and an exhaust velocity of about 2,900 m/s.
3	Create a rocket whose initial position is at the origin; if the rocket has 100% positive angular thrust for 7 seconds and max torque of 200,000 newton meters, what is its final angular velocity?

The students were given one week to complete their assigned tasks. A second meeting was scheduled for a group interview. During this interview, questions regarding the strengths and weaknesses of the code library were asked. The group interview was conducted following Mazza and Berre's (2007) methodology that consists of two parts: general impressions and specific cognitive tasks involved in using the library. In the interview, for each answer given, every participant was asked if they agreed or disagreed with that particular answer.

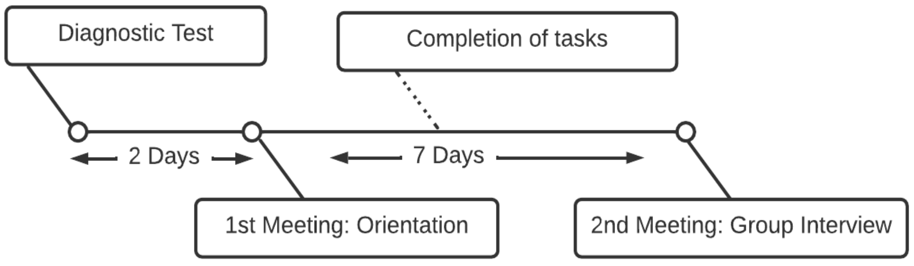


Figure 3. Participant activity timeline.

Notes and transcripts from the second meeting were categorized to gain consensus on the quality of the library based on strengths, weaknesses, opportunities, and threats (SWOT) factors. On the other hand, the submitted code of the students were analyzed and viewed in the context of their unfamiliarity with C# and programming in general. Four aspects were considered in the evaluation of the submitted code:

1. The code is first checked for the absence of compilation errors.
2. Then, if the code compiles, it is checked if the correct solutions were applied. Then, the resulting final answer is also checked for correctness.
3. Finally, the code is checked for the presence of code comments and their substance.
4. Any form of assistance given to the participant will be listed, if there is any.

To provide reasoning on why code comments are checked, it is generally agreed upon that the presence of code comments contributes towards better readability and clarity, albeit not always indicative (Buse & Weimer, 2010). Subsequently, clarity has been found to have a positive correlation to programming experience (Fernandes et al., 2017). Particularly with novice programmers, it can be implied that the optional nature of code comments makes their existence significant. An assumption can be made about the correlation of a students' understanding of their written code with the amount of code comments they put in.

Issues Encountered

Due to the work requirement of the current study, only a handful of people responded to the invitation. Furthermore, due to restrictions because of the COVID-19 pandemic, the study could not be conducted in a physical context, which could factor into students' unwillingness to participate. This issue hindered the study in terms of its scale.

Additionally, although a diagnostic test on physics problems was administered, it was later discovered that the scores of the students on this test are irrelevant to the current study. This is due to the goals of the research later shifting focus from physics education to programming education. Presentation materials for the orientation meeting and interview questions for the second meeting were changed and updated as needed. This shift was made in consideration of the existing nature of the code library and in what use case it would be better suited for at its current design. Furthermore, it is more practical and realistic for precollege students to learn programming through existing physics knowledge, rather than learn physics through nonexisting programming knowledge.

RESULTS AND DISCUSSIONS

Results

Seven senior high school students accepted the invitation to participate in the current study.

Table 2. Participant Profile

Participant #	SHS Strand	Curriculum Programming Language Used
1, 7	STEM	Python
2	ICT	C#
3, 4, 5, 6	ICT	Python

Note. SHS = senior high school.

Upon acquisition of their profile, it is known that only two of these students are from the Science, Technology, Engineering, and Mathematics (STEM) strand, while the rest come from the Information and Communication Technology (ICT) strand. The programming language that their curriculum uses, however, differs. Six participants learned Python in their senior high school curriculum, while only one learned C#.

Table 3. Programming Task Results

Participant #	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Compiles?	Yes	Yes	Yes	Yes	Yes	Yes	No
Correct?	Yes	Yes	Yes	Yes	Yes	Yes	No
Presence of code comments	Problem statement Labeled formulas Labeled variables	N/A	N/A	Problem statement Labeled formulas Labeled variables	Labeled variables	Labeled variables	N/A
Assistance?	N/A	Variable placement	Syntax	N/A	N/A	N/A	N/A
Task # assigned	Task #1	Task #1	Task #3	Task #1	Task #1	Task #2	Task #3

Six out of seven participants wrote code that both compiled successfully and produced correct answers. Four of these submissions contained comments on their code to label variables. Two submissions (Participants #1 and #4) contained comments on much of the written code and included a proper problem statement as well. It is worthy to note that Participant #1, a STEM student having no prior knowledge on the C# language, wrote significantly better code than Participant #2, who is an ICT student who has a background in C#. It is also noteworthy how the participants who put comments on their code are also the participants who required no midweek assistance.

Table 4. Interview Summary

Question	**Theme/Answer**	**Participants Answered**
Did you have any issues with the code library and the processes you must take to use it?	- No. Extensive documentation helped guide me. Even as a beginner, the code was easy to understand.	All
	- No, but I did experience considerable difficulty that may be considered user-error.	2, 3, 4, 5, 6
	- No, but my difficulty came from a difference in operating system.	7
Would you consider using the library for personal projects?	- Yes, I would consider using it in game development.	3, 6
	- Yes, I would consider using it in academic simulations.	1, 2, 4
If you had anything to change about the code library, what would it be?	- None.	1, 5, 6
	- A more complete documentation and guide.	2, 3, 4, 7

All participants agree that the code library was beginner friendly, as proven by their performance on their assigned programming tasks. It is interesting to note that five participants experienced difficulties that they considered as "user-error"; however, only two of these participants reached out to request assistance. One participant had difficulty in installation since the entirety of the tutorial and guide targets Windows users. Two participants expressed desire to use the code library in game development, while three others agreed on the idea of the library being used in academic simulations. Four participants commented on the need for more complete documentation and guide. This can also be extended to the idea of making installation guides for other operating systems.

Discussions

Six out of seven participants had submitted code that both compiled without error and answered respective tasks correctly. However, due to the small sample size and lack of a control group, this may not be decisively indicative of classical mechanics being suitable to bridge syntax learning and problem-solving. Nonetheless, it is significant to note the performance of each participant in the context of their background and their answers in the group interview.

Participant #1 was one of the two students who thoroughly wrote comments on their submitted code. Although their background is in Python and they are not an ICT student, they still displayed an understanding of C# concepts and was able to follow proper syntax with no assistance. Furthermore, this particular participant was not one of those who attested to having major difficulty. This specific case is interesting when compared to the participants who agreed that they had difficulty, because it is these participants who have a background in ICT. A presumptive argument could be made about STEM students having a better intuition on classical mechanics, therefore possessing a more detailed mental model of the problem given. If this is indeed the case, it would be consistent with the discussion of Robins et al. (2003), wherein emphasis was placed on the ability of programmers to form detailed mental models prior to writing code.

Participants #4, #5, and #6 are also notable cases. These participants also submitted code that compiles and is correct; however, despite being ones who attested to having difficulty while completing the tasks, they still did not request any midweek assistance. This implies that these three students, though not having any C# background, managed to debug their own code properly and add comments as well. This particular case may be used as a relatively stronger argument towards the suitability of classical mechanics as an alternative method in programming instruction, since it shows how students can be left on their own and yet still understand unfamiliar language features (i.e., syntax) through a detailed mental model formed with the assistance of physics.

Participant #2 is an interesting case. Though a student from ICT with a background in C#, they needed a clarification on where to put a particular value (planet mass, on Task #1) in relation to their code. This case can be contrasted with Participants #1, #4, and #5, who were students also assigned to Task #1, but unlike Participant #2, they did not need assistance. This case could be an example of what Robins et al. (2003) warn about syntax-first instruction. Students exposed to this pedagogy will tend to rely on memorized facts rather than a mental model. Therein lies the problem when students eventually forget what they had tried to memorize.

To understand the sole failure of Participant #7, the case of Participant #3 can be considered. These two students both were assigned to accomplish Task #3, which is undeniably a more difficult task when compared to Task #1, for instance. However, unlike Participant #7, Participant #3 requested assistance. Presumably, this difference could have been the deciding factor on whether Participant #7 could have made working and correct code. However, it is unlikely due to the sheer length of the time given to the students, which therefore increases the number of possible factors that could affect their completion. Nevertheless, if this was indeed the case, then this scenario emphasizes the importance of introducing or reinforcing language features (i.e., syntax) to students after the problem has been thoroughly introduced to the student, as done by Deek et al. (1998).

CONCLUSIONS AND FUTURE WORK

This study has explored the possibility of using physics, particularly classical mechanics, as a tool to bridge the gap between syntax learning and problem-solving learning. To do this, a code library was developed in the C# programming language. It implements and abstracts the concepts of gravity, thrust, torque, and collision detection. This code library is then distributed to seven participants who were assigned tasks they needed to solve using the library.

Due to the limited participant pool, a conclusion cannot be made regarding the viability of classical mechanics as a suitable means for programming education. Although some participants were able to accomplish their given tasks, it is difficult to determine if the code library aided in their abilities due to a lack of a control group. Further research is required to be able to properly evaluate the effectiveness of a classical mechanics code library as a tool for teaching problem-solving skills. Future researchers should employ a more thorough experimental design involving a control group to concretely ascertain if exposure to the code library improves one's programming abilities. Both groups should have minimal to no programming experience, with one being tutored through exposure to the code library, while the control group is tutored through traditional means of learning. It is also recommended to have a shorter time frame between the initial tutoring and the submission of the assigned task to reduce the impact of external factors on their solutions.

REFERENCES

Allen, O., Downs, X., Varoy, E., Luxton-Reilly A., & Giacaman, N. (2022). Block-based object-oriented programming. *IEEE Transactions on Learning Technologies*, *15*(4), 439–453. https://doi.org/10.1109/TLT.2022.3190318

Buse, R. P. L., & Weimer, W. R. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, *36*(4), 546–558. https://doi.org/10.1109/TSE.2009.70

Deek, F. P., Kimmel, H., & McHugh, J. A. (1998). Pedagogical changes in the delivery of the first-course in computer science: Problem solving, then programming. *Journal of Engineering Education*, *87*, 313–320.

Fedorenko, E., Ivanova, A., Dhamala, R., & Bers, M. U. (2019). The language of programming: A cognitive perspective. *Trends in Cognitive Sciences*, *23*(7), 525–528. https://doi.org/10.1016/j.tics.2019.04.010

Fernandes, E., Ferreira, L. P., Figueiredo, E., & Valente, M. T. (2017). How clear is your code? An empirical study with programming challenges. In *Proceedings of the Ibero-*

American Conference on Software Engineering: Experimental Software Engineering Track (CIbSE-ESELAW) (pp. 1–14).

Fischer, J., Mikhael, J. G., Tenenbaum, J. B., & Kanwisher, N. (2016). Functional neuroanatomy of intuitive physical inference. *Proceedings of the National Academy of Sciences*, *113*(34), E5072–E5081. https://doi.org/10.1073/pnas.1610344113

Kumar, V. (2020, October 21). *How to get started with external code libraries*. Analytics Insight. https://www.analyticsinsight.net/how-to-get-started-with-external-code-libraries/

Mazza, R., & Berre, A. (2007). *Focus group methodology for evaluating information visualization techniques and tools*. https://ieeexplore-ieee-org.dlsu.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=4271964

Robins, A., Rountree J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Tran, T., & Torrisi-Steele, G. (2022). *A conversation with my peers on learning and teaching programming*. https://doi.org/10.21125/edulearn.2022.2107

Xie, B., Loksa, D., Nelson, G., Davidson, M., Dong, D., Kwik, H., & Ko, A. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*, 1–49. https://doi.org/10.1080/08993408.2019.1565235