



2018-06-01

Real-Time Feedback for In-Class Introductory Computer Programming Exercises

Ariana Dawn Sellers
Brigham Young University

Follow this and additional works at: <https://scholarsarchive.byu.edu/etd>



Part of the [Engineering Commons](#)

BYU ScholarsArchive Citation

Sellers, Ariana Dawn, "Real-Time Feedback for In-Class Introductory Computer Programming Exercises" (2018). *All Theses and Dissertations*. 7434.

<https://scholarsarchive.byu.edu/etd/7434>

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

Real-Time Feedback for In-Class Introductory
Computer Programming Exercises

Ariana Dawn Sellers

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

John L. Salmon, Chair
Mark B. Colton
Steven K. Charles

Department of Mechanical Engineering
Brigham Young University

Copyright © 2018 Ariana Dawn Sellers
All Rights Reserved

ABSTRACT

Real-Time Feedback for In-Class Introductory Computer Programming Exercises

Ariana Dawn Sellers
Department of Mechanical Engineering, BYU
Master of Science

Computer programming is a difficult subject to master. Introductory programming courses often have low retention and high failure rates. Part of the problem is identifying if students understand the lecture material. In a traditional classroom, a professor can gauge a class's understanding on questions asked during lecture. However, many struggling students are unlikely to speak up in class. To address this problem, recent research has focused on gathering compiler data from programming exercises to identify at-risk students in these courses. These data allow professors to intervene with individual students who are at risk and, after analyzing the data for a given time period, a professor can also re-evaluate how certain topics are taught to improve understanding. However, current implementations do not provide information in real time. They may improve a professor's teaching long term, but they do not provide insight into how an individual student is understanding a specific topic during the lecture in time for the professor to make adjustments.

This research explores a system that combines compiler data analytics with in-class exercises. The system incorporates the in-class exercise into a web-based text editor with data analytics. While the students are programming in their own browsers, the website analyzes their compiler errors and console output to determine where the students are struggling. A real-time summary is presented to the professor during the lecture. This system allows a professor to receive immediate feedback on student understanding, which enables him/her to clarify areas of confusion immediately. As a result, this dynamic learning environment allows course material to better evolve to meet the needs of the students.

Results show that students in a simulated programming course performed slightly better on quizzes when the instructor had access to real-time feedback during a programming exercise. Instructors were able to determine what students were struggling with from the real-time feedback. Overall, both the student and instructor test subjects found the experimental website useful.

Case studies performed in an actual programming lecture allowed the professor to address errors that are not considered in the curriculum of the course. Many students appreciated the fact that the professor was able to immediately answer questions based on the feedback. Students primarily had issues with the bugs present in the alpha version of the software.

Keywords: engineering education, programming education, learning analytics, data visualization

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my advisor, Dr. John Salmon, for supporting me in researching something I am passionate about. His enthusiasm and motivation greatly aided my efforts as I completed my research.

I would also like to thank Dr. Colton and Dr. Charles for their helpful comments and questions throughout the research process.

Jonathan Sadler spent untold hours helping me develop and maintain the website used in this research. Without his assistance and his impressive programming skills, I would not have made it this far in this project. I would also like to thank Landon Wright for helping me maintain compliance with the IRB by collecting and scrubbing data that I was not allowed to see.

To my CAD lab friends, who have spent hours listening to me think out loud, being my guinea pigs, and proof reading my papers, thank you for the fun times we have had together. You managed to make sitting in a basement lab for hours every day a memory I will always cherish.

To the ME 273 students who were willing to put up with my constant experimenting, thank you for your patience and diligence.

And finally, thank you to my family: my parents, who have always encouraged me to do my best and further my education, and my husband Cameron, who never let me give up.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
NOMENCLATURE	x
Chapter 1 Introduction	1
1.1 Problem Statement	1
1.2 Research Objectives	4
1.3 Thesis Overview	5
Chapter 2 Background	7
2.1 Alternatives to the Lecture-based Approach	7
2.1.1 Flipped Classrooms	9
2.1.2 Live Coding	12
2.1.3 Collaborative Coding	14
2.2 Learning Analytics in Programming Education	14
2.2.1 Student Code Analysis	15
2.2.2 Student Error Analysis	18
2.2.3 Student Behavior Analysis	21
2.3 Live Feedback Systems	23
2.3.1 Student Feedback	23
2.3.2 Instructor Feedback	25
2.4 Programming and Numerical Methods Education in Mechanical Engineering	29
Chapter 3 Methods	31
3.1 Proposed System	31
3.2 Browser-Based IDE	32
3.2.1 User Interface	32
3.2.2 Website Development	41
3.3 Evaluation of the Browser-Based IDE	46
3.3.1 Preliminary Tests	46
3.3.2 Case Study Design	47
3.3.3 Simulated Course Experimental Design	48
3.3.4 Teaching Assistant Study	57
Chapter 4 Results	59
4.1 Case Study Results	59
4.1.1 Observations	59
4.1.2 Survey Results	60
4.1.3 Exploratory	61
4.2 Simulated Course Results	70

4.2.1	Observations	70
4.2.2	Quiz Results	71
4.2.3	Points of Confusion Results	72
4.2.4	Survey Results	73
4.3	TA Experiment Results	82
4.3.1	Points of Confusion Results	82
4.3.2	Evaluation of Website Metrics and Graphics	83
4.3.3	Ideas for Improvement	84
4.3.4	Teaching Assistant Insight on ME 273	85
4.4	BYU Survey Results	86
4.5	Cost Benefit Analysis	88
Chapter 5 Conclusion		90
5.1	Limitations	90
5.1.1	Human Subjects	90
5.1.2	Technology	91
5.2	Future Work	92
5.2.1	Feedback from Research Participants	93
5.2.2	Ideas from the Researchers	94
5.3	Conclusions	96
5.3.1	Case Study Conclusions	97
5.3.2	Simulated Course Experiment Conclusions	98
5.3.3	Teaching Assistant Study Conclusions	99
5.3.4	ME 273 Survey Conclusions	100
5.3.5	Overall Conclusions	100
REFERENCES		103
Appendix A Experimental Procedures		110
A.1	Simulated Course Experimental Procedures	110
A.1.1	Proctor Instructions	110
A.1.2	Instructor Experimental Procedure	110
A.2	Teaching Assistant Study Experimental Procedures	111
A.2.1	Proctor Instructions	111
A.2.2	Website Explanation	112
Appendix B Survey Questions		114
B.1	ME 273 Winter Semester Survey	114
B.1.1	Consent Document	114
B.1.2	Student Survey	115
B.2	Simulated Course Experiment Surveys	115
B.2.1	Programming Skill Survey	115
B.2.2	Student Survey	116
B.2.3	Instructor Survey	117
B.3	Teaching Assistant Study Survey	118

B.4	BYU ME 273 Survey	121
Appendix C Simulated Course Materials		125
C.1	Lecture Slides	125
C.1.1	Topic 1: C++ Basics	125
C.1.2	Topic 2: C++ Flow Control	133
C.1.3	Topic 3: C++ Functions	138
C.1.4	Topic 4: Root Finding Methods	142
C.1.5	Topic 5: C++ Static Arrays	146
C.1.6	Topic 6: Numerical Integration	149
C.1.7	Topic 7: Least Squares Regression	155
C.1.8	Topic 8: Initial Value Problems	159
C.2	Quiz Questions	163
C.2.1	Quiz 1: C++ Basics	163
C.2.2	Quiz 2: C++ Loop Control	164
C.2.3	Quiz 3: C++ Functions	167
C.2.4	Quiz 4: Root Finding Methods	172
C.2.5	Quiz 5: C++ Static Arrays	173
C.2.6	Quiz 6: Numerical Integration	174
C.2.7	Quiz 7: Least Squares Regression	175
C.2.8	Quiz 8: Initial Value Problems	175
C.3	Exercises	176
C.3.1	Exercise 3	176
C.3.2	Exercise 4	176
C.3.3	Exercise 6	178
C.3.4	Exercise 8	180
Appendix D IRB Documents		183
D.1	IRB Approval	183
D.2	Recruiting Script	185
D.3	Other Relevant Documents	186

LIST OF TABLES

3.1	Compiler Metrics and Definitions	37
3.2	Course Topics and Experimental Assignments	49
3.3	Errors in Example Error Finding Exercise	51
3.4	Programming Skill Survey Questions	56
4.1	Total Errors for Case Study	63
4.2	Student Requests for Website Enhancements	77
4.3	Instructor Rating Usefulness of Metrics	80
4.4	Instructor Ranking of Graphics	81
4.5	Instructor Ratings of Adequacy of the Data for Different Topics and Exercises	82
4.6	Percentage of Confusion Areas Identified by Each Teaching Assistant	83
4.7	Teaching Assistant Ranking of Graphics	84
4.8	Teaching Assistant Ranking of Difficult Subjects in ME 273	85
5.1	Participant Suggestions, Feasibility, and Hours to Implement	94
5.2	Conclusions for Hypotheses Based on the Simulated Course Experiment	99
5.3	Conclusions for Hypotheses with Supporting Experiments listed	101

LIST OF FIGURES

2.1 Overview of Topics Discussed in the Background Section	8
2.2 Instructor View Dashboard from [1]	15
2.3 4-gram Visualization of Student Paths from [2]	19
2.4 Distribution of Student Problems by Final Grade recreated from [3]	20
2.5 Class Layout View from [4]	27
2.6 Student Panel View from [4]	28
3.1 Main Website Page	32
3.2 Administrator Page	33
3.3 Assignments Page	33
3.4 Assignment Menu	34
3.5 Student Text Editor	34
3.6 Instructor Create Assignment Page	35
3.7 Instructor Assignment Menu	35
3.8 Instructor Edit Assignment Page	36
3.9 Instructor Compiler Data Page Part 1	38
3.10 Instructor Compiler Data Page Part 2	39
3.11 Instructor Compiler Data Page Part 3	39
3.12 Instructor Compiler Data Page Part 4	40
3.13 Example of an Error Finding Exercise	51
3.14 Example of a Program Writing Exercise	52
3.15 Example of a Type 1 Quiz Question	53
3.16 Example of a Type 2 Quiz Question	54
3.17 Example of a Type 3 Quiz Question	55
4.1 Student Responses to “The website was easy to use”	60
4.2 Student Responses to “The website was enjoyable to use”	62
4.3 Student Responses to “The website made it easy to do in-class exercises”	63
4.4 Student Responses to “The instructor changed his lecture based on the feedback”	64
4.5 Student Responses to “The instructor was able to address student confusion based on the feedback”	65
4.6 Mean Compiles vs. Success For Exercise 1	67
4.7 Mean Compiles vs. Success For Exercise 4	70
4.8 Mean Quiz Scores by Group	72
4.9 Student Responses to “The website was enjoyable to use” by group	74
4.10 Student Responses to “The website was enjoyable to use” by group	75
4.11 Student Responses to “The website made it easy to do in-class exercises” by group	76
4.12 Student Responses to “The instructor addressed student issues during the exercise” by group	77
4.13 Student Responses to “The instructor was aware of how the students were doing on the exercise” by group	78
4.14 Student Responses to “I enjoyed doing exercises on the website” by group	79

4.15 Student Responses to “Doing in-class exercises helped me understand the material” by
group 80

NOMENCLATURE

<i>IDE</i>	Integrated Development Environment
<i>API</i>	Application Programming Interface
<i>CAD</i>	Computer Aided Design
<i>STEM</i>	Science Technology Engineering and Math
<i>CS</i>	Computer Science
<i>ME</i>	Mechanical Engineering
<i>GUID</i>	Globally Unique Identifier
<i>UUID</i>	Universally Unique Identifier
<i>TA</i>	Teaching Assistant
<i>RSP</i>	Run Success Percentage

CHAPTER 1. INTRODUCTION

1.1 Problem Statement

In the digital age, programming has crept into every facet of life. Commonplace objects like water bottles and lamps are being enhanced by sensors and data analytics. Purely mechanical devices are becoming increasingly rare; meanwhile people are carrying computers in their pockets and wearing them on their wrists. Programming has become so important that New York City recently started a program to introduce computer science courses in all their middle schools [5]. Coding is going to be taught alongside the traditional math, science, English, and history courses. It is rapidly becoming a fundamental part of our society, and the trend looks set to continue, with the federal government making programming education a priority in 2016 [6].

The engineering industry has become increasingly reliant on programming as well. Data acquisition and control systems, common in mechanical engineering, already required a working knowledge of programming. With the advent of smart devices, even more mechanical systems are being integrated with programmable microcontrollers. The ability to program allows mechanical engineers to work with and design these systems appropriately [7].

Industry has also become dependent on computer-aided engineering programs for complex structural, fluids, and heat transfer analyses. Much of this software is most useful when it can be customized for the company's needs, which often requires programming. Furthermore, an understanding of how these computer-aided engineering programs work is essential to using them correctly and efficiently [7]. While not every mechanical engineer will need extensive coding experience, programming is a skill that can set a mechanical engineering student or employee apart from their peers.

The need for engineers who can program is clear. Unfortunately, introductory programming is widely regarded as a very difficult course [8,9], with a low retention rate and high failure rate. Watson and Li found that only 67.7% of students pass introductory programming courses after

evaluating 161 courses in 15 countries. They also found that this rate has not improved over time, despite significant research in improving introductory programming [10]. There is a high learning curve for computer programming; it is very similar to learning a completely new language. Because learning programming is difficult and time consuming, it can have a bad reputation among students, similar to math anxiety in elementary and high school students [11–13]. Math anxiety is the phenomenon where a student’s performance in math suffers due to their fear of math, rather than their lack of ability. When students believe that math is too difficult, they perform poorly; when a student enters a course already feeling defeated, it is very difficult for them to succeed. A similar situation could be happening with programming students.

While programming is difficult to learn, it is also challenging to teach [4]. For the most part, introductory computer programming courses have been organized like other science, technology, engineering, and math (STEM) courses, with a lecture session, laboratory assignments, and homework assignments. Students listen, take notes, and then attempt to solve coding problems on their own. This can be compared to a language course, where students are required to speak often in class and collaborate with one another. Language courses are fundamentally interactive; introductory programming courses should be similar.

Course sizes for introductory programming have increased as more students seek STEM degrees and more non-computer science students enroll. This leaves instructors with far more than the ideal number of students for one-on-one interaction, even with generous office hours. It is very difficult for an instructor to understand how such a large class is understanding the course material. Researchers have observed that the learning process of students in programming courses is fairly opaque to instructors [4]. Instructors know who asks questions in class, what questions are asked, and what grades are given. However, struggling students are unlikely to ask questions in class, especially if they feel that the rest of the class knows more than they do. This means that questions asked in class are often more advanced than what the class needs to know. While grades are a good form of feedback, they are often delayed by one or more weeks. If an instructor relies on homework grades to determine what students are struggling with, it will be weeks before they can understand and address the issue. A similar problem occurs with teaching assistant feedback. Teaching assistants generally help students throughout the week and report to the instructor any

issues the students are having. Again, this process can take a long time and the help the students need is delayed.

Mechanical engineering students and instructors face additional challenges with computer programming courses. Most mechanical engineering programs have long, intense courses of study. Unlike electrical engineering, which relies heavily on computer science courses, other engineering disciplines typically teach their own computer programming courses [14]. Programming material is sometimes coupled with numerical methods, control systems, or mechanical-electrical systems, requiring students to both learn basic programming and discipline specific applications in a short period of time.

Despite these challenges, there is minimal research on teaching programming to mechanical engineering students. Mechanical engineering programming courses vary widely between different universities, which may contribute to the lack of research. A few researchers have tested out using MATLAB in courses that do not traditionally use computer programming [15]. Researchers did a comprehensive survey of which languages are taught in mechanical engineering in 2002 [14]. One researcher has examined a new method for teaching numerical methods [16]. Much of this research is based on anecdotal evidence rather than empirical studies. As the need for engineers who can program increases, more research on teaching introductory programming and applications to mechanical engineering students is needed.

While mechanical engineering educators have not thoroughly researched programming education, computer science and electrical engineering educators have. There are two main areas researchers have focused on: 1) making lectures more interactive and 2) gathering data on student programming behavior. To improve lectures, researchers have investigated the flipped classroom method, live coding, and pair programming with generally positive results. The flipped classroom method involves students reviewing lecture material at home and completing assignments and activities with the instructor during class [17]. Live coding describes a situation where an instructor or students program during class instead of showing static, prepared examples [18]. With pair programming, students are put in groups of two to complete assignments [19]. One student will program while the other watches and provides input. Pair programming has great potential and has become very popular in introductory programming courses. While these methods have improved interaction in lecture, they have not made the students' status clearer to instructors.

To help instructors better understand what students are struggling with, researchers have gathered data on student programming behavior. Earlier researchers had teaching assistants record what students had problems with over a period of time [3, 20, 21]. More recently, researchers have created integrated development environments (IDEs) and learning tools that automatically record student data, analyze it, and visualize it for both instructors and students [1, 4, 22–26]. Learning tools for students provide automated feedback on their programs without having to wait for an instructor’s help. Visualizations for instructors show course wide trends in errors, time to completion, lines of code, etc. Additionally, visualizations can make it easier for instructors to examine large amounts of student code. While many of these tools and IDEs have been effective, few provide immediate feedback. Most only show trends over time and take significant computational power [1, 25]. Instructors were able to use the data to identify individual students who were struggling and improve lecture content for the next semester [23]. This is beneficial; however, the data was not used to help a large portion of the current class. The few tools that provided live feedback were used during open office hours or lab sessions [4], rather than in a lecture environment; again, this limited the ability of the instructor to use the data to help the entire class quickly and obtain a better idea of how students understand lecture material. Time can be saved by taking care of the most common issues when the class is together. The additional time could then be spent helping individual students.

1.2 Research Objectives

The overall objective of this research is **to improve introductory programming courses for mechanical engineering students by introducing live feedback on student code during programming lectures**. The following research questions will be investigated:

1. **RQ1:** How can live feedback be implemented to improve programming lectures?
2. **RQ2:** How can live feedback be adapted for a mechanical engineering programming course?

This research aims to answer these questions by accomplishing the following research objectives:

1. **RO1:** Develop a live compiler feedback system suited for use during a lecture

2. **RO2:** Evaluate the live compiler feedback system
3. **RO3:** Recommend future improvements for the live compiler feedback system

To determine whether or not the live feedback improved the effectiveness of programming lectures, two factors were considered: 1) student performance and 2) instructor understanding of student status. Students took quizzes to measure their performance and fill out surveys to record what they struggled with. The instructor took a survey recording what they believe the students struggled with. To determine the effectiveness of live feedback for mechanical engineering courses specifically, numerical methods topics were tested along with programming topics.

The following hypotheses were made regarding the research objectives:

1. **H1:** Students will perform better on quizzes when the instructor has access to live compiler feedback
2. **H2:** The instructor will more frequently know what students struggled with on an assignment when they have access to live compiler feedback
3. **H3:** When error finding exercises are given, the live feedback will have a more significant effect on quiz scores for programming topics
4. **H4:** When program writing exercises are given, the live feedback will have a more significant effect on quiz scores for numerical methods topics
5. **H5:** The instructor and students will find the live compiler feedback useful

1.3 Thesis Overview

Chapter 2 of this thesis will provide an in-depth literature review of flipped classrooms, live-coding, pair programming, learning analytics, compiler data, and live feedback. Chapter 3 will propose a solution to the problem statement. It will then describe and explain the design of live compiler feedback system developed by the researchers. The preliminary testing performed will be explained. The case study and experiments used to evaluate the live compiler feedback system will be discussed in detail. Responses from a survey performed by the BYU ME Department will

be evaluated. Chapter 4 will present the results of the case studies, experiments, and survey and provide an analysis. Chapter 5 will conclude this research and discuss limitations, future work, and conclusions.

CHAPTER 2. BACKGROUND

There has been extensive research over the last decade on improving both computer programming education and engineering education at the university level. Traditional lectures and a lack of data have been identified as major issues with current methods. This chapter will review past research in improving lectures and gathering data in programming courses. Previous implementations of compiler data gathering systems and live feedback systems will be discussed in detail. An overview of the topics discussed is shown in Figure 2.1.

2.1 Alternatives to the Lecture-based Approach

Using the traditional lecture format as the only method of university-level teaching has many shortcomings. While lectures allow for a large amount of material to be transferred to many students in a short amount of time, they are often ineffective [27]. The traditional lecture format also prevents professors from receiving feedback from students except when the students ask questions. Research has shown that while some students are motivated enough to ask questions, most struggling students are not [3]. This means that a professor will not know how most students are struggling based on questions asked during a traditional lecture. Furthermore, students often think they understand the course material during lecture, and then find themselves completely lost during a lab or homework assignment. Since these problems only manifest themselves when the student is outside of the lecture, the professor does not generally hear about it for a significant amount of time. It is typical for a professor to receive weekly updates on what students are struggling with, either from teaching assistants or from homework grades. By this point, the troublesome lecture topic was at least a week ago and the ideal time to address the issue has passed.

Programming courses pose additional issues. Learning to program is similar to learning a foreign language; it comes with a massive learning curve. Memorization of syntax and logical reasoning are both necessary to write even a simple program. A single typo can cause a program

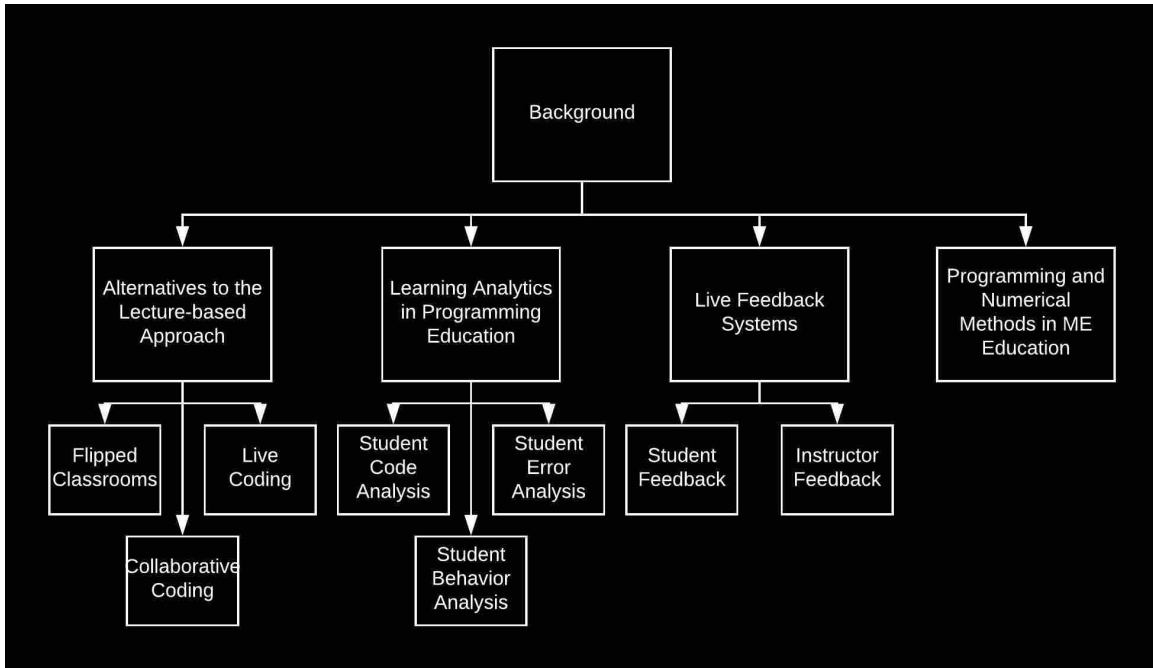


Figure 2.1: Overview of topics discussed in the background section

to fail completely, often leading to lengthy lists of compiler errors for one mistake. This can be incredibly discouraging for students already tentative about learning programming [28]. Students may be unable to complete assignments due to these small errors, and therefore struggle to fully understand the material. Since so many small things can cause errors, from missing semi-colons to bracket placement, it is difficult for an instructor to evaluate the class’s understanding without spending time with each student. Additionally, many students do not program during lecture, further reducing the instructor’s ability to evaluate their errors and areas of confusion. As with other courses, an instructor generally finds out about student issues at a weekly meeting with teaching assistants, who can make observations after a week of assisting students with laboratory and homework assignments. By this point, students may be a week behind where they should be in terms of understanding the lecture topics.

While these problems are generally recognized, lectures still prevail because they are easier to implement and maintain. First, the current university system favors the traditional lecture format, so no radical changes are required. Second, professors are incredibly busy, and lectures are relatively easy to develop when compared to experimental methods. However, experimental methods are becoming more common in undergraduate programs due to their effectiveness. The rest of

this section will focus on several experimental methods that are currently being implemented, and therefore are promising examples for this research.

2.1.1 Flipped Classrooms

Overview

The flipped classroom is a method that has been implemented across many fields, with varying degrees of success. The method was developed by two high school instructors trying to help students who missed class. When they posted lecture videos online for these students, they found that students who had not missed class also used the videos to help them remember the course material. After observing the benefits of recording lecture material, they decided to try teaching in a different way [27].

The premise is to provide students with more hands-on experience with assistance from the instructor. Instead of having students sit through a lecture and attempt to do homework and labs on their own, professors provide lecture material in the form of videos or slides. Students go through the material before class, and then spend class working on assignments.

Advantages

There are many advantages to the flipped classroom model that have been thoroughly investigated. Students can learn at their own pace, work with classmates in class, and avoid excessive periods of frustration without aid. Professors can work more directly with students, improve student attitudes towards the material, and help students solve more open-ended problems. Overall, it provides a very flexible approach to learning for both the students and professors. Instructors can enable collaboration between students more easily than when students perform their assignments individually at home. Education research in general is moving more towards flexible learning schemes, as it has become abundantly clear that everyone learns differently. As such, this is a very timely method and is becoming more and more accepted [27].

Disadvantages

Unfortunately, there are also several downsides to the flipped classroom approach. The most significant issue is student involvement. The success of the method hinges on students reviewing the lecture material before class. If students do not review the material, they have no questions in class and cannot proceed on the assignments with the professor lecturing. Researchers have found that students are generally willing to do the preparatory work early on in a semester. However, as the semester continues and the workload increases, preparatory work drops off. Additionally, the amount of preparation done varies student by student. A successful student will spend significantly more time reviewing the preparatory material. Student preparation is more difficult to regulate when not in a traditional lecture environment [27].

The second most important issue is the preparation required by professors. Instead of preparing slides, the professors need to prepare to actively assist with homework and lab assignments, or come up with interactive assignments. This requires both more time and effort, and is sometimes infeasible. A professor has to be motivated to successfully implement a flipped classroom [27].

There are several other disadvantages identified by the research, including technical issues, an informal learning environment, and keeping the class moving. As with any method or system that relies on the internet, there will invariably be problems where someone cannot access the lecture materials, the formatting becomes distorted, someone does not have access to the internet, etc. An informal learning environment can also develop when students are watching lecture material on their own, rather than in a controlled setting. They may fail to take notes or decide not to pay attention. The additional assistance in class may make them less motivated to understand the material on their own. Lastly, when each student is reviewing the material at their own pace, it is more difficult to keep everyone on the same page. Students in one group might figure something out with the professor that is not shared with the rest of the class [27].

Implementations in Engineering

In 2015, Kerr wrote a survey paper of 24 studies where the flipped classroom model was applied to engineering education. Overall, the studies indicated that students were satisfied with

the flipped classroom method and improved their performance [17]. She determined that while the flipped classroom can be effective, additional improvements are needed to increase the chances of success. These measures generally involved adding accountability and credit for students completing work outside of class, either through quizzes or rubrics. Other factors that can impact the success of a flipped classroom are class size and the type of course. While previous research has shown that various class sizes and course types have been effective with a flipped classroom, there are not sufficient data to draw any conclusions. Overall, the flipped classroom method has potential and is generally successful in improving problem-solving skills, understanding, retention, and satisfaction of students [17].

Implementations in Computer Programming

Maher used a flipped classroom method for four different computing courses over two years, ranging from introductory to graduate courses [29]. She developed several strategies over those two years, most of which revolved around collaboration. She had students participate in pair programming and designed all in-class activities to be collaborative.

Students found the videos to be especially useful for programming topics, since they could follow along with the video at their own speed. Additionally, they could refer back to the videos when completing other assignments. However, there were a few downsides to the videos, including occasional poor quality, lack of conceptual information, inability to ask questions, and isolation while watching the videos [29]. With quality, well-balanced videos, it is likely that these issues could be remedied.

Along with the videos, students enjoyed having a hands-on experience in class, explaining things to other students, and the approach overall. Many students were initially concerned about the new approach, but by the end of the semester had become appreciative of the method. Overall, the flipped classroom method was successful across a variety of computing courses, but more research is needed to refine the strategies for maintaining that success.

Harding also implemented a flipped classroom strategy, having his students view tutorials, lecture videos, and readings before class [30]. He then used a collaborative, web browser-based IDE for in-class programming exercises. The professor had the ability to upload an instructor version of a piece of code. Students then each had access to a copy of that code that they could

individually edit without affecting anyone else. This enabled the professor to easily share code with the class during lecture, enabling in-class exercises. Students were able to practice what was being taught immediately and ask detailed questions.

Along with the main student-instructor modes, there were several other modes for website users: everyone, student leader, watching or “lurking”, and group mode. The “everyone mode” put all of the students into one IDE, like Google Drive for programming. Student leader allowed one student to make changes while others watched. Lurking allowed students to view their classmates’ code, generally to assist one another with a problem. Lastly, group mode allowed students to operate in a collaborative, Google Drive type environment with only specified students.

These different modes enabled several useful flipped classroom features. Students were able to show their own code to the class and take turns teaching each other and explaining topics. The instructor could easily view any student’s code, make changes, and explain them. Outside of the classroom environment, the instructor was able to help students collaboratively in real-time on their at-home assignments. Students could also help each other collaboratively in this way. While the instructor aid was not used in this situation, it is a potential benefit of the collaborative, online IDE system [30].

Overall, both students and the instructor found the flipped classroom format useful for their introductory programming course. Students found the format helpful compared to following the instructor’s code on slides. However, no quantitative data were presented. It is therefore difficult to actually measure the effectiveness of the in-class exercises from this study.

2.1.2 Live Coding

One solution to some of the issues facing the flipped classroom is to only partially flip a classroom, generally by implementing interactive activities, exercises, or quizzes during lecture. Live coding is one of these methods. Some instructors use live-coding along with a traditional lecture [31], while others use live coding in a flipped classroom [18].

Traditionally in programming courses, pre-written programs are included in lecture slides. Live coding removes these pre-written programs and has either students or the instructor write code from scratch during lecture. This allows students to practice or watch every step of the programming process, including debugging [18]. Live coding also prevents students from memorized

canned solutions, encouraging them to actually understand how the solutions work. Some past research has shown that students may learn faster when live coding is used during class [18].

Rubin performed a live-coding experiment with four sections of an introductory programming course over a semester at the Colorado School of Mines [31]. He measured final grades at the end of the semester, comparing students who had live-coding examples with those who had static examples. He found that student final grades were statistically the same at the end of the semester. However, students in the live-coding sections performed significantly better on the final project [31]. Project scores generally indicate the ability of students to apply what they have learned to a new, more complex problem. Rubin concluded that live-coding was at least as effective as static examples, and potentially more effective. Live-coding made it easier for the professor to show students how to debug.

Shannon examined the effect of live coding on in-class quiz scores in a flipped classroom environment to determine whether there were short term benefits to live coding over static examples [18]. She split four main topics into subtopics, and used a live coding example for one and a static example for the other. Students watched a lecture video before class. In class, students were able to ask questions before going through the live coding or static example. After the example, students were given a 10 minute quiz.

Shannon did not find any significant difference in quiz scores between the live coding and static examples. However, there was a difference in score distribution. After a live coding example, quiz scores demonstrated a normal Gaussian distribution. After a static example, quiz scores exhibited a binomial distribution, with students either performing very well or very poorly [18]. It is possible that the live coding example allowed for the quiz scores to more accurately represent the students' understanding of the topic.

Overall, live coding has some proven benefits in terms of student performance. Additionally, it makes logical sense; showing students each step of writing a program can help them see the thought process behind each step. It decreases the cognitive load a student experiences when faced with trying to comprehend the entire solution at once. Live coding is a flexible method that can be applied for an entire lecture period, or for only a few minutes for an exercise.

2.1.3 Collaborative Coding

Another common solution to issues with the traditional lecture approach is collaborative coding, more commonly referred to as pair programming. Pair programming is a specific teaching model where students are placed into pairs. There are many different ways to assign pairs, from skill level to personality tests [19, 32]. Each person will take turns acting as the driver or the viewer. The driver uses the mouse and keyboard to write code, while the viewer looks on and catches mistakes, discusses algorithms, etc [33]. While not always implemented in lecture, pair programming allows students to receive help quickly despite a professor not having the time to individually assist each student in a large course. Pair programming is becoming more popular and has been found to be effective in many studies, improving student retention and attitudes about programming [19, 34–37]. However, pair programming is only effective when pairs are formed carefully [33]. Extensive research has been performed to improve how pairs are formed and how best to implement pair programming in various courses, ranging from introductory programming to senior level courses and distributed learning environments [21, 33, 37–40]. Collaboration has been shown to particularly help women in programming courses [41].

Several researchers have created collaborative IDEs to further enable pair programming. Some are plug-ins and others are browser-based, but nearly all of them are language specific. The browser-based IDEs are very similar to Google Docs, but with compiling capabilities. These IDEs can also enable better interaction in distance learning environments. Most researchers have found their collaborative IDEs to be effective, but the amount of empirical evidence varies greatly [34, 42–47].

2.2 Learning Analytics in Programming Education

Over the past several years, computer science and engineering education researchers have been gathering data to help tackle the high failure rates in introductory programming courses [10]. These data can provide a detailed view into the concepts with which students struggled in programming assignments. The data have been used both to improve course materials and lectures over an extended period of time and to aid teaching assistants in real-time. There are three main areas the research has focused on: student code, student errors, and student behavior. Researchers



Figure 2.2: A mock-up of the instructor view dashboard from [1]. The instructor view shows number of students passing various percentages of reference tests, the amount of development time by quartile, and a summary of student grades, classes, branches, and comments.

have developed systems that analyze student code structure, compiler errors, and student behavior, includes number of compiles, number of lines written, number of questions asked, etc.

For a thorough review of learning analytics in programming education, see [48] and [25]. Issues with data analytics in learning are discussed in [49].

2.2.1 Student Code Analysis

Luke developed a program that continuously collects student data while they program [1]. Events were captured when students performed a variety of actions, such as compiling or running their code, by a plugin for Eclipse. A snapshot of the current code whenever an event was triggered was stored as well. These data were analyzed and presented in a visual format for both the instructor and students, shown in Figure 2.2.

The instructor received a class overview for a specific assignment. This included two primary plots: a plot of the number of students passing various percentages of reference tests by date, and the development time by quartile. Lastly, a summary table with data for each student was presented. This table included each student's submission status, grade, percentage of reference tests passed, percentage of code completed, number of commented lines, number of classes, number of methods, and solution time [1].

Because Luke was experimenting with a computer science course, he was able to utilize unit testing to gather further information about student code. Most engineering courses, aside from computer engineering, do not teach or utilize unit tests due to time constraints and quantity of course material. Some of these additional measures are cyclomatic complexity and code coverage percentage. Luke analyzed the cyclomatic complexity over time by quartile. He defined cyclomatic complexity as “the number of logical branches in solution code divided by the total number of methods” [1]. Code coverage, the percentage of the solution run when all unit tests are used, measures how much students are using unit tests. While these features are interesting, they may be less useful in an engineering environment.

Luke also used comment percentage, code churn, code velocity to visualize his data. Comment percentage is simply the percent of the lines of code that are comments. This can indicate how well students are documenting their code. Code churn is how many lines of code changed each day over existing code. This allows the instructor to see when students are getting stuck, and when they are making progress. Code velocity refers to the velocity of edits. In conjunction with code churn, code velocity can indicate when students are programming a lot but making very little progress [1].

While Luke developed a very thorough program for gathering live student data, he has not yet published research evaluating the program. The data collected was based on ease of gathering it and what might be useful. Furthermore, these data were gathered for at-home assignments.

Glassman studied two main compiler data systems, OverCode, and Foobaz. OverCode creates a visualization of solution variation for a specific assignment. The solutions are tested for correctness by an autograder so only “correct” solutions are used for the visualization. This allows the instructor to focus on program structure and content instead of simple syntax errors.

OverCode was meant to help teachers find common misconceptions, create rubrics, and choose good examples to show the class [24].

The user interface allows the instructor to analyze a function or section of code. Common lines of code and the number of submissions they appeared in are shown in a window. These lines can be filtered by the number of submissions. Instructors also have the option to combine similar lines and ignore unimportant variations using a rewrite rule. This again allowed the instructors to focus on important differences in student code.

To pre-process the student code, Glassman used a Python package to remove style-based inconsistencies. She then executed each solution on a set of test cases while recording a program trace. The values of each variable as the tests are executed are recorded. Common variables were then identified as variables with the same values throughout a program. All of the variables were then renamed for consistency; common variables were all given the same name. Stacks are then formed from solutions that share an identical set of lines of code. Extensive work was done to avoid variable naming conflicts [24].

To evaluate OverCode, Glassman chose three problems from an introductory Python course to gather data from. She found that the code ran quickly enough to be used easily, with a running time of 15 minutes for 3875 student solutions. A user study was conducted with twelve subjects to evaluate user satisfaction with OverCode. Users were asked to use OverCode to look over student solutions, and then write a page about various ways the students solved the problem. Users found OverCode “less overwhelming, easier to use, and more helpful for getting a sense of students’ understanding.” Subjects were able to determine that students were having issues with control flow. However, they were not always clear on what the student was misunderstanding [24].

A second user study was performed to determine how quickly instructors could look through student solutions using OverCode. The results showed the instructors could look through more solutions using OverCode when the problems were more complicated. Instructors were also able to provide more feedback when using OverCode on the most complicated test problem. Lastly, with the most complicated test problem, subjects were more confident that their feedback would be relevant to many students after using OverCode [24].

Glassman performed a similar study focusing on Foobaz, a compiler analytics program that gathered data on student variable names and allowed instructors to provide feedback. The process

and results were very similar to the study on OverCode. Generally, users found the feedback to be useful. Instructors felt like they were able to help many students at the same time, greatly increasing their efficiency. The number of responses they were able to make confirmed the increase in efficiency. The software also enabled instructors to create custom quizzes based on the student's variable names. Students with similar issues in their code received the same quiz. Students found that the quizzes made them think more carefully about variable names. For both OverCode and Foobaz, clustering algorithms were essential in simplifying a large set of student code into something easily understandable [50].

Wang et al. created a visualization tool called Path Viewer to help instructors understand student strategies and areas of confusion. Paths are created based on test cases; a sequence of zeros and ones is assigned to each run based on which tests the code passed. A plot is created with student paths over time, from one sequence of tests passed to another. Paths with more than five students are shown on an aggregate plot. An example plot is shown in Figure 2.3. Paths that result in successful code are colored green, while incorrect programs that pass the test cases are colored blue. Example code is shown when the user selects a path. For example, students could be going back and forth between passing two different test cases. This indicates that their solution to one problem is causing another problem, demonstrating a lack of understanding [2].

After using Path Viewer during a course, the instructor realized that the test cases given were often inadequate for the assigned problems. Additionally, the instructor recognized that students were not utilizing the principles of recursion, and were instead relying on lengthier algorithms [2].

2.2.2 Student Error Analysis

Garner et al. collected data on student problems by having teaching assistants report the problems after helping a student. He categorized student issues by topic, including basic structure, program design, arrays, loops, data flow, constructors, etc. Garner found that the number of problems per student was normally distributed with the student's final grade in the course [3], as shown in Figure 2.4. Students achieving average grades had the most problems according to teaching assistants, while low and high performing students did not report as many issues. Garner hypothesized that high performing students asked for less help, and therefore had fewer problems

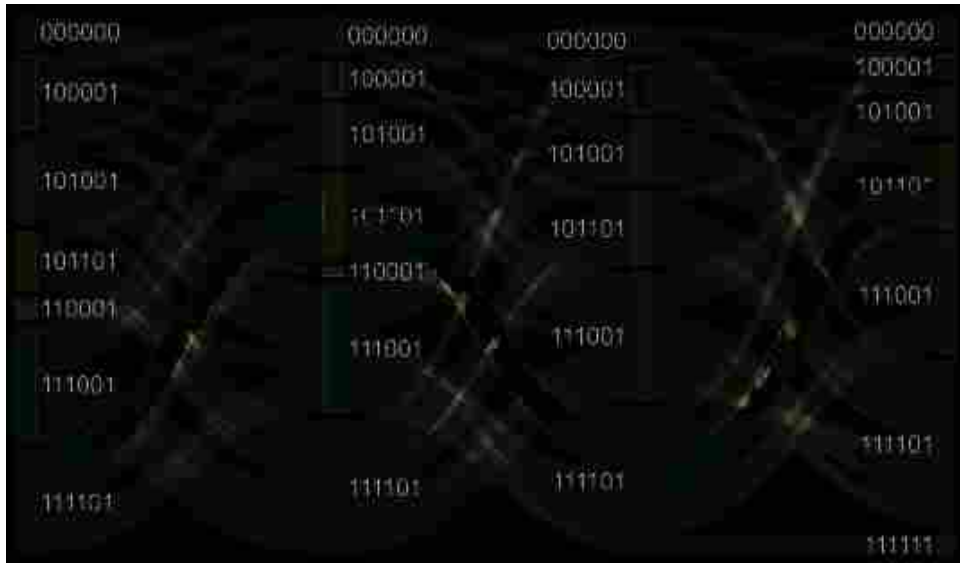


Figure 2.3: A 4-gram visualization of student paths while performing programming exercises from [2]. A 1 indicates a test case being passed; a 0 indicates a failed test case. Students with oscillating paths are likely going between two different errors repeatedly.

reported. Low performing students also likely asked for less help, had fewer problems reported, but still struggled in the course.

Garner also found that arrays, data flow, loops, constructors, and control flow were the most problematic topics, which was consistent with other studies. In the end, Garner concluded that many simple issues persisted throughout the entire semester, suggesting that students were not actually learning the subject material [3].

Robins also found that simple areas such as understanding the task, program design, and mechanics had very high frequencies of mistakes [20]. The authors took this as evidence that the basic design principles of programming can be more important than language specific syntax. They cautioned that learning analytics must be evaluated very carefully and never generalized, as there are so many factors that could affect the data.

The previous studies, [3, 20], focused on students performing programming assignments individually. Hanks replicated [20] with pair programming to determine the difference when collaboration was involved. While students had the same number of errors overall as those in studies without pair programming, they asked for less help, indicating that they were able to solve more problems on their own [21].

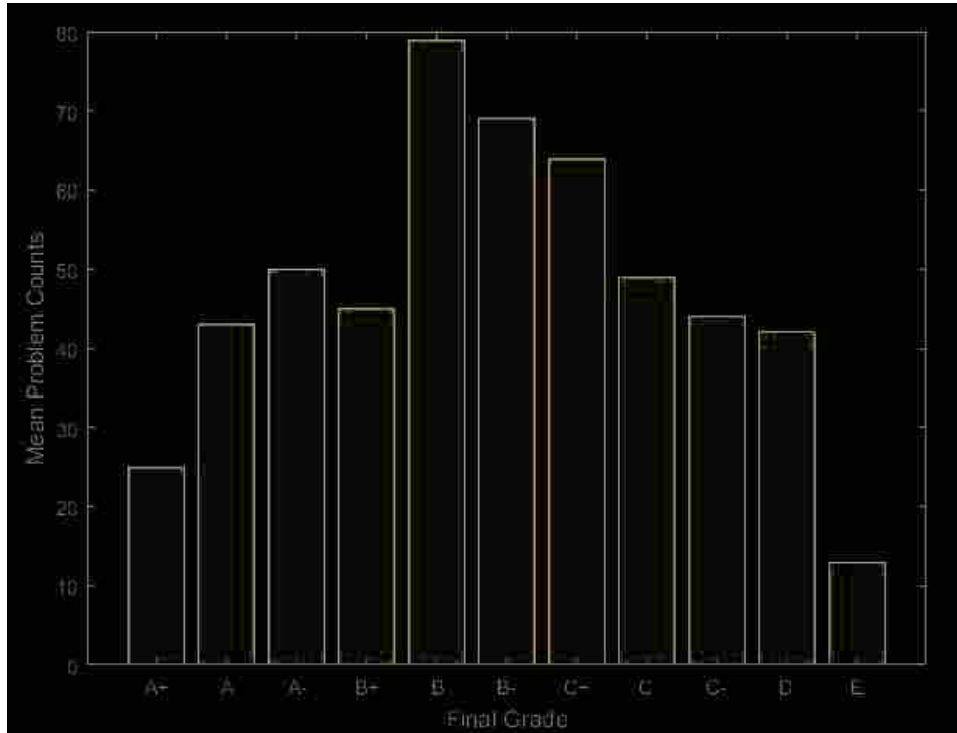


Figure 2.4: The number of student problems by final grade recreated from [3]. The distribution appears to be similar to a normal Gaussian distribution, with average students having many problems. High and low performing students had fewer problems.

Altadmri recently analyzed one of the largest sets of student compilations over the course of a year [22]. He was able to classify the most common errors, the types of errors that are most common, and how repeatable errors are. How often a student repeats an error shows whether the mistake was a simple syntax problem or a deeper lack of understanding. Examples mentioned by Altadmri as repeating issues include using the wrong type in a function call or missing a return statement.

The data showed that run-time errors are much more difficult for novice programmers to solve than compiler errors, as would be expected. This was evaluated by looking at the time to resolve an error. The authors also looked at the frequency of mistakes made over a semester. The number of syntax errors generally decreased through a semester. However, the number of semantic and type errors increased. Altadmri concludes that additional time should be spent on semantic and type problems [22].

2.2.3 Student Behavior Analysis

Instead of having teaching assistants manually record student issues, Toll recorded student programming sessions with an online programming environment [51]. He was able to determine how much time students spent actively programming. Additionally, he found that students who turned assignments in early were much faster at completing assignments than students turning in assignments later. He suggested using these data to inform which students need more instructor attention [51].

In 2015, Lin et al. tracked student eye movement while students looked at C programs to find bugs. They found that high performing students looked through programs logically, rather than line by line. Lower performing students went through programs line by line, often starting at the error and moving backwards. Lin demonstrated a difference between high and low performing students based on their debugging behavior, which could potentially help instructors teach debugging more effectively [52].

In 2016, Estey and Coady developed a programming course tool to record compiler and interaction data for an introductory programming course. The programming course tool allowed students to ask for hints, ask for a repeated hint, and attempt questions. Along with those data, the number of submissions and compiles were also recorded. The study was performed over several semesters. Estey found a significant difference between students at the top, middle, and bottom of the course. Successful students attempted more questions, compiled more, and submitted more assignments. Students who failed the course asked for more hints and repeated hints, submitted less, compiled less, and tried fewer questions [53]. These trends were repeated over three semesters and were visible within the first two weeks of the semester. Estey was able to identify 30% of students at risk of failing within the first two weeks of the semester. By the end of the semester, the metrics from the collected data recognized 81% of the students who failed [53].

Using patterns in the interaction data, Estey was able to identify students who would eventually fail the course, even if they began the course with a high grade. Initial success on simple, early assignments did not indicate success in the course. Instead, study habits directly affected success. Attempting more practice problems, compiling multiple times, and refraining from relying on hints lead to success. The more a student did this, the more successful they were. Relying on hints, especially repeated hints, without trying to compile was the most sure sign of failure in the

course. Estey suggests that these students do not understand that relying on hints is an ineffective way to learn. These data could be used to help students adjust their perception of effective learning methods [53].

With the large amount of compiler data that has been gathered, multiple models have been created to predict student performance based on the data. Both the Error Quotient and Watwin Score models are based on the differences between a student's compilation attempts [54,55]. They can account for between 30 and 40% of the variation in student final grades. Carter, Hundhausen, and Adesope created a model incorporating semantic correctness along with syntactical correctness called the Normalized Programming State Model (NPSM) [56]. They defined four states: syntax and semantics correct, syntax correct but semantics incorrect, semantics correct but syntax incorrect, and both incorrect. Semantic correctness was determined only by the lack of run-time exceptions, which was an acknowledged weakness of the work. These and a few other factors were used to create the NPSM model. Carter et al. compared their method with the Error Quotient and Watwin Score, and it performed significantly better in predicting student final grades. As demonstrated by [56] and [53], compiler data can be used to predict student performance. More work in this area can be found in [57, 58].

Olivares and Hundhausen have created a learning analytics platform called OSBLE+ that allows instructors to visualize various data regarding student performance [59]. OSBLE+ is a Visual Studio plug-in that gathers student programming data and implements a social media platform [26]. Students can make and reply to posts, which are also recorded by the plug-in. The instructor has a dashboard that shows the number of active students, lines of code written, time spent on the assignment, number of compiles, number of errors per compile, number of debug executions, number of breakpoints, number of runtime exceptions, number of posts, number of replies, etc [60]. Hundhausen has found that being active in the social media aspect of the tool correlates with success in the course [61–64]. Olivares and Hundhausen hope to use these data to automate interventions for struggling students by prompting them to participate in the social media aspect [60]. Olivares et al. are still in the process of testing OSBLE+ and have not yet, as of the time of writing, published their results.

Olivares and Hundhausen have also studied and recommended a process model for IDE-based learning analytics. They suggest that three research questions need to be answered as re-

searchers continue to use data analytics for computational learning: “What learning data should be collected within an IDE in order to provide a foundation for improving student learning? How should the learning data be analyzed in order to provide useful information on student learning? Based on the learning data, what interventions should be delivered through an IDE in order to benefit student learning?” [25]. Their process model involves collecting data, analyzing data, designing interventions, and delivering interventions. This model has not yet been evaluated [25].

2.3 Live Feedback Systems

Many automated feedback systems produce displays of data over a period of weeks or days and thus provide long-term feedback. Live feedback is instantaneous and generally applies to only one specific exercise or problem. Live feedback has been experimented with in several forms, from compiler error feedback for students to clicker quizzes to automated feedback for instructors. All three types will be discussed in the following section.

2.3.1 Student Feedback

IDEs automatically provide feedback in the form of compiler errors. An interesting factor with compiler errors is the negative psychological affect it has on novice programming students. Attempting to run a program and finding an extensive list of seemingly incomprehensible errors is very demotivating, and likely contributes to the high failure and low retention rates in introductory programming courses [28]. Additionally, one error can cause dozens more, often making the problem seem worse than it actually is.

Matsuzawa investigated how compiler errors should be visualized to improve students’ view of compiler data and improve students’ self-assessment. He created a compiler error viewer that allowed students to see the number of errors they had fixed over time, how long it took them to fix errors, and percentage of working time used to fix compiler errors. This was able to help rectify false student perceptions of debugging taking up all of their time. In the end, the viewer decreased student fear of compiler errors for 44% of students and 77% of students found the viewer useful for learning how to correct compiler errors [28].

Along with presenting compiler errors for students, several researchers have worked on automatic feedback systems for students. This allows students to get help at any time and any place necessary. Keuning performed a review of papers on automated feedback generation for programming exercises in 2016 and found that most feedback systems simply provide information about test failures from “black-box” problems, generally based on compiler errors [65]. Automated feedback for an incorrect solution or poor code quality is much more rare. Unless a feedback system is programmed with specific problems, they have trouble providing students with help on how to proceed. Additionally, these pre-programmed feedback systems do not allow for multiple solution variations; instead, they nudge the student towards the default solution. This can inhibit students who think differently, as well as stop students from learning different methods. Keuning concluded that more research is necessary to improve automated feedback systems, as well as more thorough and empirical research [65].

In 2016, Le classified the different analysis techniques used in automated feedback systems [66]. The first method, Library of Plans and Bugs, attempts to guess which strategy a student is using and then finds discrepancies between the student’s program and the stored program. These discrepancies are matched with potential errors for that specific problem. This method requires an extensive database of previous student strategies and errors and can be computationally expensive. However, it is also more broad than some other methods [66].

The second method, Program Transformation, uses only one reference program. It abstracts the student’s program and compares high level steps with the reference program. It then compares the details of the program. This method can help students with both large and small picture issues [66]. However, it is also limited to one reference solution, which can cause issues as discussed in [65].

The third method is the Weight Constraint-Based Model. This method was developed by Le and involves creating a semantic table that contains different solution strategies. Weights were calculated for a well-formed program and compared to the weights for the student’s program. The method would then attempt to predict the student’s strategy and analyze the solution accordingly. This method is very flexible if implemented correctly [66].

In 2017, Parihar combined automated feedback with automated program grading [67]. Common errors were identified and matched with a typical compiler message. Feedback was

developed for the compiler message including a simple explanation and examples of both valid and invalid statements. These common errors covered about 78% of the total errors experienced by students during the course of the study. Parihar's study showed that students found the more detailed feedback useful [67]. This provides evidence that compiler errors are confusing and often unhelpful to introductory programming students. A simple adjustment can make the errors much more useful.

2.3.2 Instructor Feedback

Live feedback for instructors has generally taken two forms: clicker quizzes and compiler data. Clicker quizzes can quickly show how the class understands a topic based on multiple choice questions, while compiler data can show student errors and outputs. Live feedback is useful for instructors to quickly determine whether or not the class understands a concept. When live feedback is given, instructors can respond immediately and attempt to remedy any issues.

Rawat tested a more general form of live feedback in 2008. Rawat found that using mobile tablets in an undergraduate electrical engineering course enabled instructors to instantaneously poll students and review their work, which improved student performance [68]. Purdue University implemented a program called Signals that automatically emails students with suggestions on how to improve their performance when the real-time data indicate they are struggling. They found improvements in the number of students earning higher grades, as well as more students seeking help earlier on in the semester [69]. This indicates that many more students would seek help if they realized they were struggling and were given suggestions on what to do to improve.

Clicker Quizzes

Clicker quizzes are multiple choice quizzes given during class that students respond to through a hand held device or computer. The results for a clicker quiz are instantaneously available and are generally viewed in a bar graph or similar chart. In 2006, Chen examined rapid feedback in a mechanical engineering statics course. He compared using PDAs to flashcards and found no difference between the two immediate feedback methods. Students found the feedback helpful in both cases. The instructors found that they were able to determine which concepts were difficult

for students based on the rapid feedback. Lastly, student grades improved in the following course, dynamics and mechanics, implying that rapid feedback had a long term effect on the students, rather than a fleeting effect [70].

Lantz and Stawiski performed a clicker quiz study in a more controlled setting to eliminate potentially confounding variables present in a traditional classroom, using video lectures. They found that subjects in the immediate feedback group performed significantly better on quizzes during the lecture. Additionally, subjects in the immediate feedback group performed significantly better on follow-up quizzes two days later. Furthermore, their scores improved on the follow-up quizzes compared to the initial quizzes. They found no difference in results based on the timing of the quizzes during the video lecture [71].

Compiler Data

In 2016, Char used Cody Coursework, produced by Mathworks [72], to automatically grade and record results for student solutions written in Matlab in an introductory computing course in electrical engineering. Instructors write tests for the automatic grading and can view detailed reports online. Data were not gathered on student debugging; students were asked to debug their code before submitting what they think is correct to Cody Coursework. Char believes Cody Coursework will be useful and is performing further studies [73].

In 2012, Chang created a mobile instructor interface connected to a collaborative, web-based programming IDE called CollabodeTA. The instructor interface was used by teaching assistants in lab sessions, which are common in introductory programming courses. The interface gave teaching assistants an overall idea of how the class was performing on the assignment in real-time as well as individualized information for the students they were helping [4].

The interface includes four key components: class layout view, console output view, student panel view, and a help queue. The class layout shows each student's image in their corresponding seat in the classroom, shown in Figure 2.5. Each student image also contains a summary of their status on the current assignment. The console output view shows summaries of current outputs students were producing during the lab session. The student panel view shows more detailed information for each student, shown in Figure 2.6. The help queue shows which students were in the queue, as well as how many errors they had [4].



Figure 2.5: Class layout view from [4]. 1) shows the student cards, each with a student image and an indicator showing whether or not they are on the help queue. 2) shows the toolbar, allowing the user to select a view. 3) shows the help queue. 4) shows the area left for additional information if needed.

Each student panel contained a time line of how often the student had run their code as well as whether or not the runs were successful. The panel also included an event stream with output from each run, including whether or not it was successful, errors, and run time exceptions. If any other students got the same errors or exceptions, it is noted on the panel next to the error. The user can check the other students' similar errors or view the current student's code causing the error. The user can also see whether or not the student is currently in the help queue [4].

Chang used cluster optimization to combine similar errors and exceptions and increase the usefulness of the data and speed up the visualization process. Clusters are formed using fuzzy string comparisons between outputs, errors, and exceptions. Each cluster is then stored in the database and mapped to a unique identifier. This made it easier for Chang to store and retrieve the data [4].



Figure 2.6: Student panel view from [4]. 1) shows the student’s image and name. 2) shows the event stream, including time stamps, errors, and exceptions. 3) shows a timeline indicating when successful and unsuccessful runs occurred. 4) indicates whether or not the student is on the help queue.

To validate the interface, Chang used recorded data from a semester of an introductory programming course at the Massachusetts Institute of Technology (MIT). The recorded data were run through CollabodeTA and shown to teaching assistants from that semester to simulate an actual lab. Teaching assistants were given a list of questions to consider while viewing the data to evaluate how useful the interface was and whether or not they could determine the issues students were having. All of the study participants felt that the console output view and classroom view were useful. They appreciated the mobile aspect because it was easy to use. They were able to answer all of the questions asked them by the study. No quantitative data were provided in the dissertation [4].

Researchers also used the recorded data to analyze each lab and examine student behavior. They found that students did not run their code often and did not iterate on each assignment. This

suggests students were not testing as they went along and did not improve their algorithms once they passed initial tests. No quantitative data were provided on these studies in the dissertation [4].

Chang suggested several additions for CollabodeTA, including adding compiler metrics, unit testing, and non-fatal errors. However, it was acknowledged that gathering these types of data would be more difficult. Additional visual cues could be added to the student cards in classroom model, including progress bars and colors indicating struggling students. She suggests using fuzzy comparison to eliminate unnecessary clusters [4]. Chang demonstrated the potential of a live feedback system in an introductory programming course.

2.4 Programming and Numerical Methods Education in Mechanical Engineering

There is minimal research on programming and numerical methods education in Mechanical Engineering, despite the growing prominence of programming in the field [14]. In 2002, Hodge and Steele felt that with the advent of Mathcad and Matlab, the need for structured programming languages would become less necessary over time. They sought to examine the current state of programming courses in mechanical engineering programs across the country. They found that most programs require more than one programming language, generally C++ and Matlab. Despite this, C++ is rarely required in future courses, while Matlab is. Hodge and Steele were surprised by the number of courses still involving a structured programming language like C++, when Matlab is easier to use. They hypothesized that as time goes on, structured programming languages will be replaced completely with Matlab [14]. However, as noted by [7], that has not yet occurred.

Most research involving programming in mechanical engineering focuses on introducing a programming tool, generally Matlab, in a freshman or sophomore level engineering course [15]. These additions are typically effective based on anecdotal and survey evidence, with little quantitative data. More recently, Aden-Buie examined different types of final exams for a numerical methods course, finding that a multiple choice test with partial credit was the most appropriate [74].

Coller performed one of the few studies on improving the teaching of numerical methods in mechanical engineering in 2009. He created a video game based on a race car and had students implement various numerical methods to control the race car. Coller found that students were much more engaged, spending additional time on the projects, and had a better comprehension of the course material as determined by concept mapping. Students working on the video game

project felt that both numerical methods and computer programming were much more important than students in a traditional numerical methods course [16].

CHAPTER 3. METHODS

This chapter will introduce the proposed system to integrate live compiler feedback with in-class exercises. The tools used to develop the feedback system and the final result will be described. All of the experiments used to evaluate the feedback system will then be discussed, including a case study, a simulated course experiment, and a teaching assistant study.

3.1 Proposed System

The traditional educational feedback method involves teaching assistants gathering information from student help sessions and reporting back to the instructor. This process typically takes at least one week. By this point, students have had multiple additional lectures that build on that previous topic. If students were confused with the original material, they are likely more confused with the new material. The cycle of feedback for students can be even slower; assignments can take weeks to be graded, especially in large courses. Students are often unsure if they really understood the material until they receive their grades.

This work attempts to inject feedback for the instructor into an interactive lecture environment to 1) increase student involvement and 2) improve instructor understanding of student comprehension. The interactive lecture environment was accomplished by incorporating a live-coding exercise, shown to be effective by [31]. Live-coding is defined as the instructor coding an exercise from scratch in front of students, rather than presenting students a completed solution. To improve instructor understanding, feedback based on student compiler data was provided as suggested in [1,4,23,26]. Unlike previous studies, the feedback in this study was given live during lecture as students completed an in-class exercise.

Previous research suggests that a browser-based integrated development environment (IDE) can enable in-class interaction and compiler data gathering [4, 30]. A website with compiler capabilities was developed that would continuously gather student compiler data and display it for

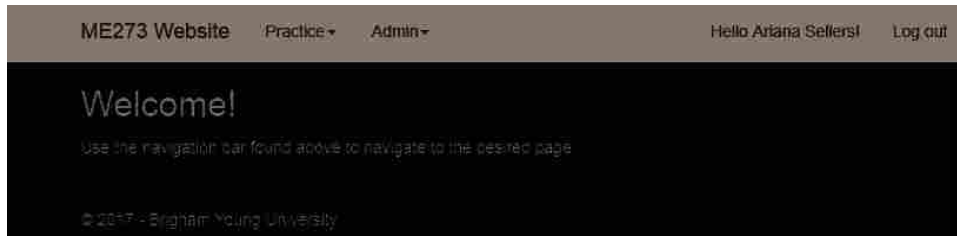


Figure 3.1: The welcome page of the website for an administrator.

the instructor. A detailed discussion of the website is included in Section 3.2. A website was chosen in part because students could program on their own laptops without having to download any additional software, minimizing the barriers to adoption.

3.2 Browser-Based IDE

The user interface of the browser-based IDE will be described in detail in the current section. The development of the website and packages used will then be discussed.

3.2.1 User Interface

The basic user interface includes a log-in and welcome page, shown in Figure 3.1. Additional pages are restricted based on the user's authorization and appear in pull down menus at the top of the screen. There are four levels of authorization: student, teaching assistant, instructor, and administrator. The student has access only to the student interface, while the teaching assistant, instructor, and administrator have access to the instructor interface. Administrators also have access to an Admin tab, shown in Figure 3.2, that allows them to change the authorization of any user and update passwords.

Student Interface

The student interface includes only one tab, Practice. This tab takes the student to the assignments page, shown in Figure 3.3. This is a list of assignments created by the instructor. A student can select an assignment and choose to start the assignment from the menu shown in Figure 3.4. The student is taken to a text editor with Save, Compile, and Run buttons, shown

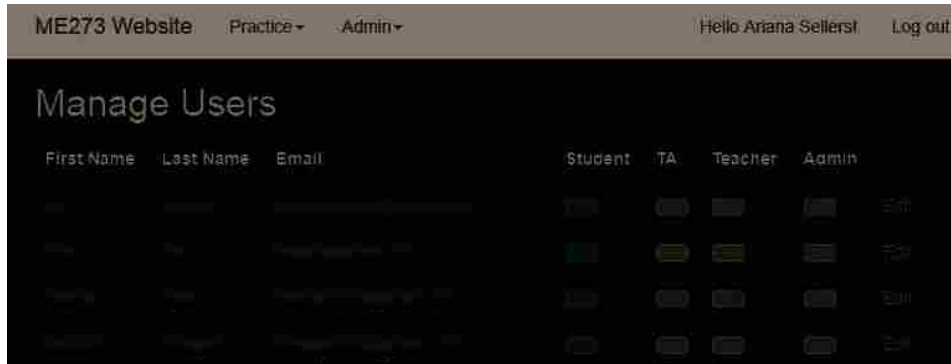


Figure 3.2: The administrator page of the website.

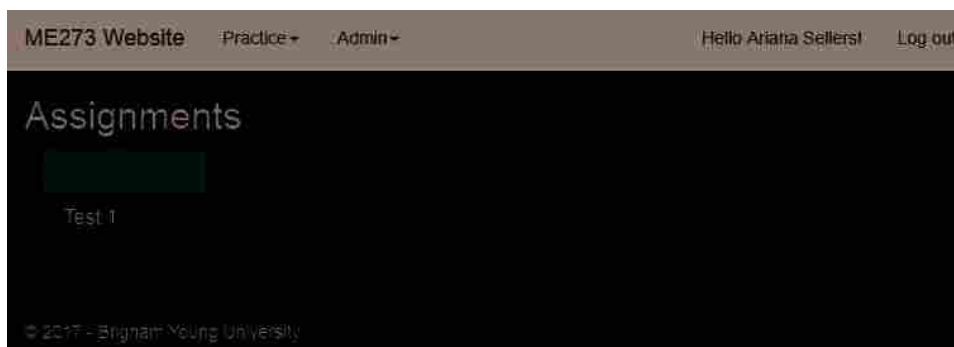


Figure 3.3: The assignments page of the website.

in Figure 3.5. The text editor was created by CodeMirror and can recognize language specific syntax [75]. If the instructor saves the assignment with code already written, that code will appear in the text editor. Students can write code and save, compile, and run their programs. Any compiler errors or console output are shown in the bottom left of the screen, seen in Figure 3.5.

There are a few limitations to the student interface. Students cannot use the C++ *cin* command, as there is no console for users to provide input. This was deemed irrelevant to the research because file input and output is only one short topic in a programming course, and therefore was not included; however, it could be added in the future. Additionally, common lines of code used in ME 273 such as *system("pause")* cannot be used, as the code is compiled on a Linux server rather than through a Windows system. *System pause* is unnecessary for the website, as it was used to keep the output window open after the program finished executing. With the website, the output remains on the screen until the next compile attempt. If *cin* or *system pause* are used, the server times out after 15 seconds and sends a message to the user explaining that their code is incorrect.

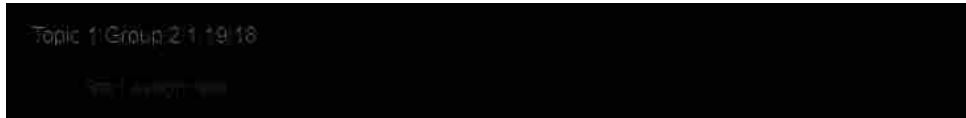


Figure 3.4: The student assignment menu.

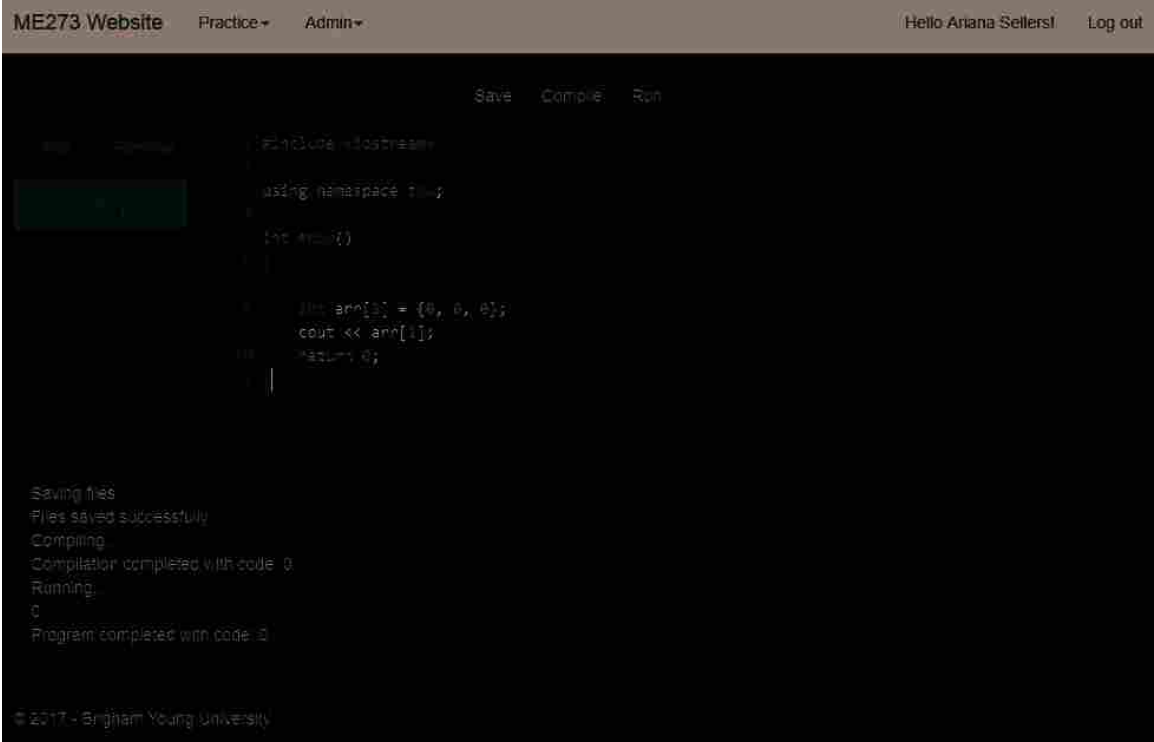


Figure 3.5: The student text editor.

Instructor Interface

The instructor interface has the same assignments page as the student interface shown in Figure 3.3. Instructors have access to the blue “Add Assignment” button, which takes them to the “Create Assignment” page, shown in Figure 3.6. Students do not have access to this button or page. The assignment is created by entering a name and clicking the submit button. This takes the instructor back to the assignments page. Clicking on an assignment allows the instructor to edit, start, and view data for any assignment through the menu shown in Figure 3.7.

The edit feature allows the instructor to give students a starting point or code solution if desired. The edits are made on the page shown in Figure 3.8. The instructor writes code in the text editor and saves the new file with the save button. The Add and Remove buttons shown

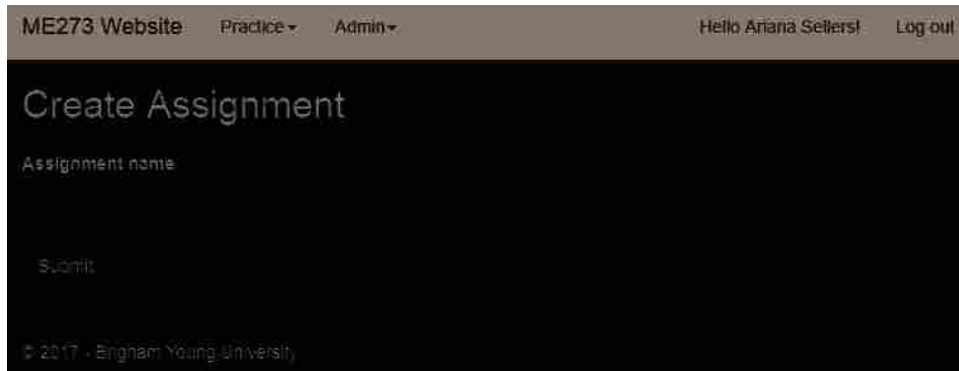


Figure 3.6: The instructor create assignment page.

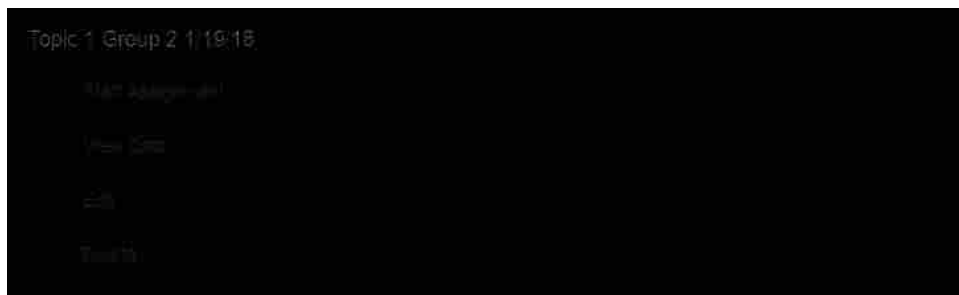


Figure 3.7: The instructor assignment menu.

in Figure 3.8 are there for future functionality and will allow the instructor to add and remove files from the programming project. This would be useful for experimenting with object oriented programming. If the instructor has edited an assignment, when students first start an assignment, they will receive the instructor's edited version. After students work on the assignment, their own changes are saved over the instructor's on their local version.

An instructor can start an assignment just as a student can. The instructor's text editor page is the same as the student's shown in Figure 3.5. However, the instructor's data will not be included in the visualization.

The view data option takes the instructor to the Compiler Data page, shown in Figures 3.9 - 3.12. The purpose of the Compiler Data page is to help the instructor visualize how the class is performing on an in-class assignment. The included metrics were chosen after reviewing previous research and discussing potential metrics with three mechanical engineering professors who have taught introductory programming. The six metrics are: 1) persistence of errors, 2) current outputs, 3) total errors, 4) total outputs, 5) times compiled, and 6) run success per-

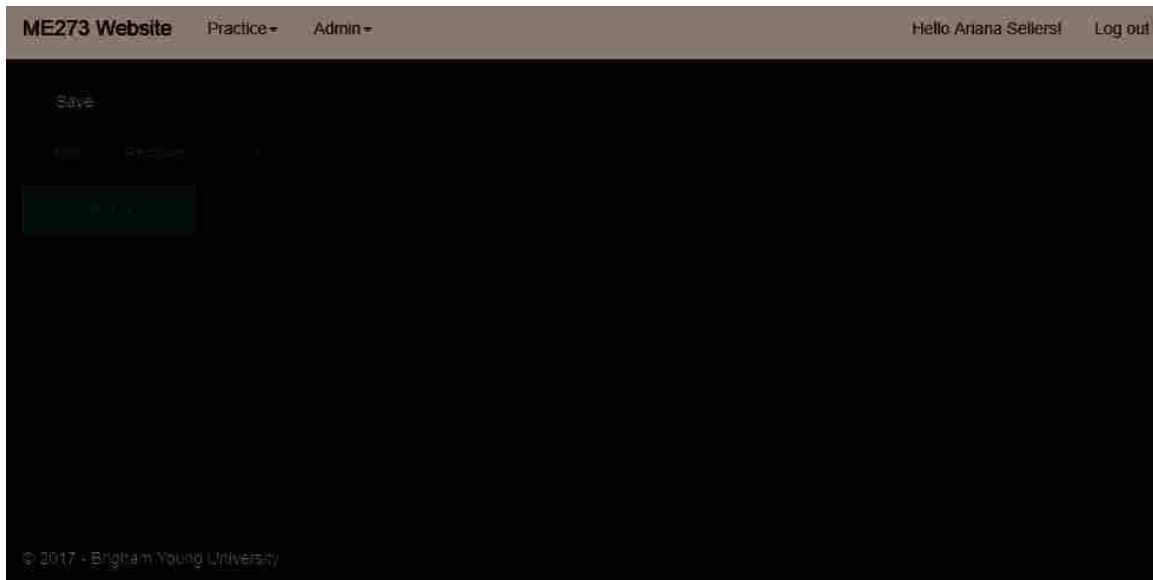


Figure 3.8: The instructor edit assignment page.

centage. A brief definition of each metric is provided in Table 3.1. Each metric will be described in more detail in the following paragraphs. Previous work by Luke involved several in-depth metrics in his compiler data feedback system [1]. However, these measures were displayed with respect to time over the course of several days. Since the current proposed system needed live feedback, simpler measures were chosen. The simpler the measure, the more quickly the instructor can interpret the data and take action. Additionally, Luke’s proposed measures had not yet been tested, so their effectiveness was unknown. Chang used both output and compiler errors in her live feedback system [4], and found them to be effective and informative. Number of times compiled and run success percentage were suggested by the consulted professors.

The Compiler Data page consists of six bar graphs and two tables divided into four sections, shown in Figures 3.9-3.12. The first section shows the current state of student solutions overall, with persistent errors and current outputs. The second section shows the total errors and outputs throughout the exercise. The third section shows data for individual students, with the number of times compiled and the run success percentage for each student. The final section contains two tables, one for current errors and one for total errors.

The first bar graph shows “Persistent Errors”, as seen in Figure 3.9. Persistence is defined here as the current occurrences of an error over the total occurrences of that error for the exercise.

Table 3.1: The definitions for each compiler metric used for the website

Metric	Definition
Persistence of Errors	The number of current occurrences of an error over the total number of occurrences of that error for all students combined
Current Outputs	The number of occurrences of the most recent output printed to the console by each student
Total Errors	The number of occurrences of each error that occurred during the exercise
Total Outputs	The number of occurrences of all output printed to the console by the students
Times Compiled	The number of times each student compiled
Run Success Percentage	The percentage of runs that were successful for each student

This provides a value between 0 and 1 describing how often the error has been fixed; or alternatively, how infrequently the error has been fixed. For example, if a “variable is not declared” error occurred five times during the exercise, but only two instances occurred on the most recent compile for each student, the persistence would be $2/5 = .2$. This calculation is shown in Equation 3.1.

$$persistence = \frac{current_occurrences}{total_occurrences} \quad (3.1)$$

This statistic is calculated for the class as a whole. An error with high persistence has rarely been resolved, while an error with low persistence may occur often but is fixed quickly. The y-axis of the bar graph is the persistence, while the corresponding error messages are displayed along the x-axis. Due to space limitations, only the top five persistent errors are shown.

The second bar graph shows “Current Outputs”, as seen in Figure 3.9. Output refers to the text students are printing to the console. In the future, controls could be added to only include specific types or formats of output to better filter the outputs included. This provides the instructor with information about the current state of student solutions, such as how many students are getting the correct answer. Intermediate outputs can be seen as well. The number of occurrences is shown on the y-axis, with the outputs on the x-axis. Again, only the topic five most frequent outputs are included, but this can be adjusted.

The third bar graph shows “Total Errors”, as seen in Figure 3.10. Total errors refers to all of the errors encountered by students throughout the exercise. This provides the instructor with

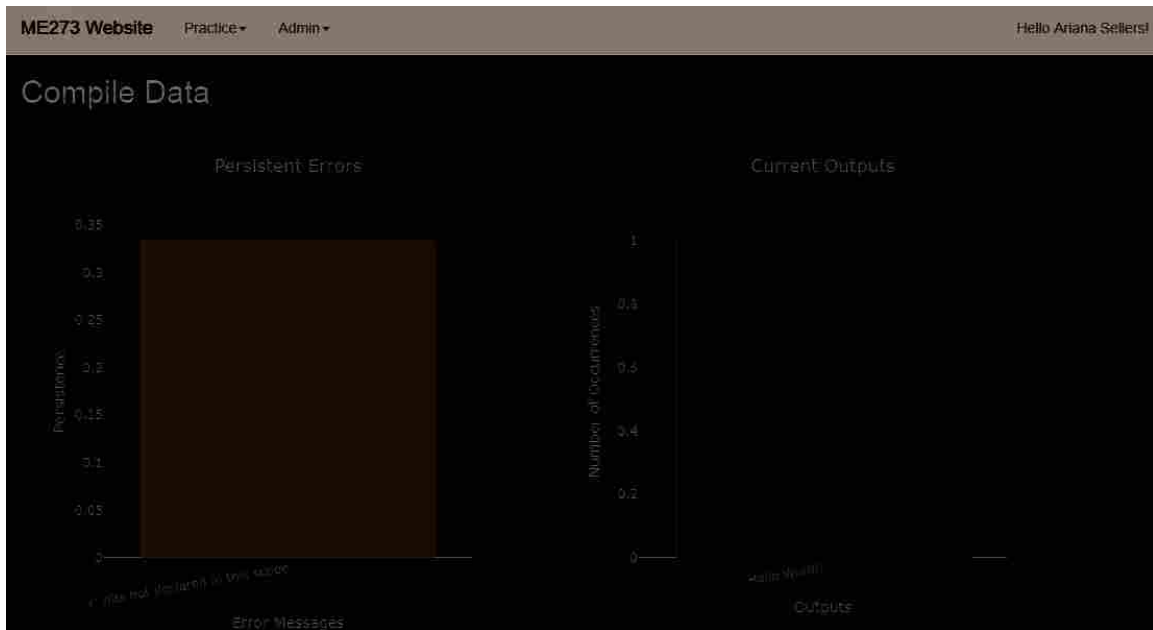


Figure 3.9: The first section of the instructor Compiler Data page showing persistent errors and current outputs.

an idea of how many errors are encountered and which ones are most common. A comparison between this graph and the Persistent Errors graph can emphasize which errors need more attention and which are common but easy to fix. The number of occurrences is shown on the y-axis, with the error messages on the x-axis. Only the top five most frequent errors are included.

The fourth bar graph shows “Total Outputs”, as seen in Figure 3.10. Total outputs refers to everything written to the console by students throughout the exercise. By comparing the total outputs to the current outputs, the instructor can see how student solutions changed during the exercise, as well as what percentage of solutions were correct. The number of occurrences is shown on the y-axis, with the outputs on the x-axis. Only the top five most frequent errors are included.

The fifth bar graph shows “Times Compiled”, as seen in Figure 3.11. This graph depicts how many times a student ran or compiled their code (running includes compiling the code). Times compiled allows the instructor to see how often students are testing their code. The number of times compiled has been correlated with success in an introductory programming course [56]. If an instructor sees that students are not compiling frequently, he or she can recommend that they test their code more often and explain that compiling more often can help the students become

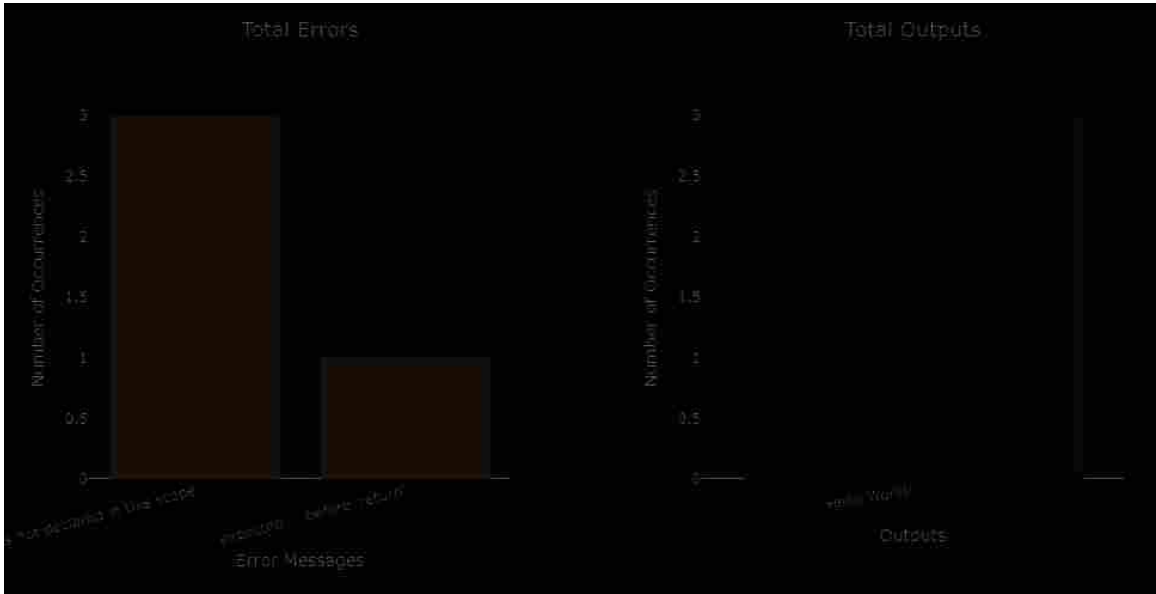


Figure 3.10: The second section of the instructor Compiler Data page showing total errors and total output.

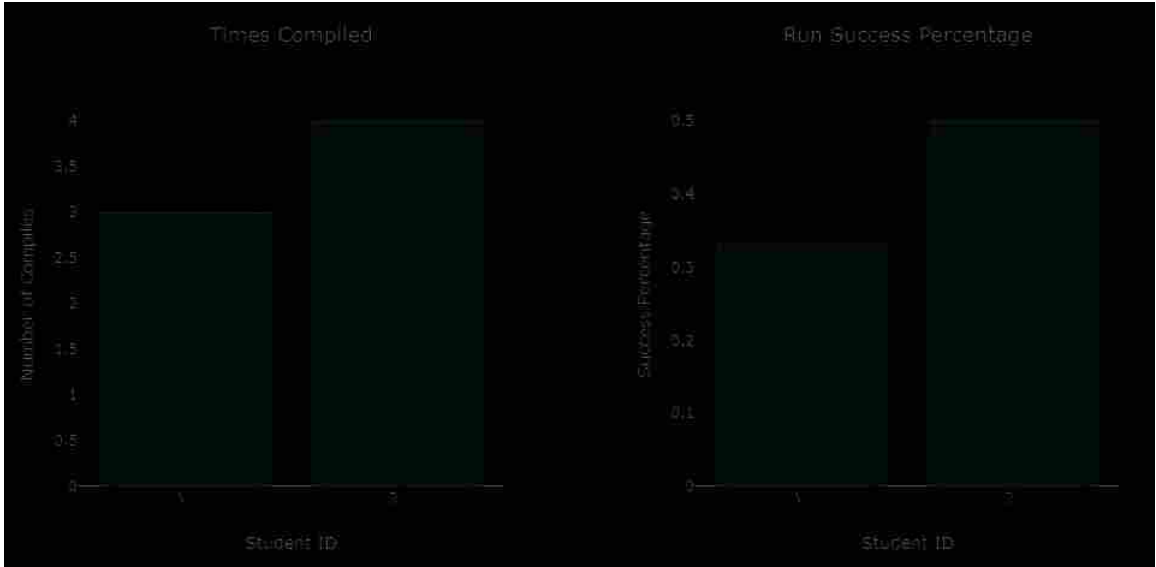


Figure 3.11: The third section of the instructor Compiler Data page showing times compiled and run success percentage for each student.

better programmers. The number of compiles is shown on the y-axis, while an ID indicating the student is shown on the x-axis. This is currently just an integer from 1 to the number of students. It could easily be replaced with student names or other identifiers if desired.

Current Errors	
Error	Count
'sum()' was not declared in this scope	10
expected primary-expression before ')' token	5
expected identifier before numeric constant	2

Total Errors	
Error	Count
'a2' was not declared in this scope	75
expected ',' or ';' before 'int'	17
expected identifier before numeric constant	6
expected unqualified-id before ')' token	4
expected primary-expression before ')' token	4
request for member 'size' in 'x', which is of non-class type 'int' [5]	1
no match for operator '>' (operands are 'std::ostream' (aka 'std::basic_ostream') and 'float')	2

Figure 3.12: The third section of the instructor Compiler Data page showing tables for the current and total errors.

The sixth and final bar graph shows “Run Success Percentage”, as seen in Figure 3.11. The Run Success Percentage (RSP) is calculated and shown for each student. The RSP is defined as the number of successful compiles over the number of total compiles. An RSP of one indicates that the code ran without errors every time. An RSP of zero indicates that the student still has errors in their code or has not yet attempted to run their code. This graph allows the instructor to see how many students have eliminated errors from their programs. Because the times compiled and RSP graphs are next to each other, the instructor can also compare how the number of times compiled relates to success. The run success percentage is shown on the y-axis, while a number indicating the student is shown on the x-axis.

The final section of the Compiler Data page consists of two tables, one for Current Errors and one for Total Errors, as shown in Figure 3.12. Each error message is displayed next to the number of times it occurred. While these tables show the same data as two of the bar graphs, the tables were considered useful because of the extra space for the message being displayed. The bar graphs required the error messages to fit in a small space along the x-axis, occasionally cutting off part of a longer error message. The tables allow for the full error messages to be displayed. Additionally, the tables hold all of the errors, rather than the five most common errors. The tables were primarily used when the bar graphs indicated a problem and the instructor wanted more

information. Because the tables are a secondary source of information, they are located at the bottom of the Compiler Data page.

3.2.2 Website Development

The website was developed using several different languages and packages that will be discussed in this section. The server, database, and text editor used will also be described. The processing used to sort the compiler data and visualize them will then be explained.

Programming Languages Used

The browser-based IDE was developed using C#, HTML, and JavaScript. HTML was used to design the web pages and JavaScript was used to allow for interaction. The Bootstrap library was used for the visual style of the website, to keep the formatting consistent and neat. C# was chosen for the rest of the website, including the server and data analysis. C# was injected into JavaScript as needed by means of Razor pages on the server. C# was selected due to the developer's familiarity with the language and its useful querying functions.

Server

The server was created using a C# Core Web Application template. This provided default code for database access, user log in, web pages and associated controllers, and all necessary backing code for running the site. The server sets up an HTTP connection to listen for requests and respond to them.

The server attempts to follow a REST (Representation State Transfer) design pattern. This allows the system to be highly scalable as it removes the need for the server to store the current state of each of the users. It also allows all resources to be requested easily by means of a simple URL (Uniform Resource Locator).

Each part of the website is managed by a different controller. The controllers determine what content should be returned and whether the user has the correct privileges to request the information.

The server is also responsible for compiling student code. This was achieved by using *gcc*, a free, open source compiler. When the server received code from the students, it started a new process which compiled the code. It would then verify if the program compiled correctly by looking at the exit code. If it compiled correctly, the server would then start a process to run the student code. All output was intercepted by the server and processed so it could be returned to the student.

Database

The database was created using Microsoft Entity Framework. This handles the access to the database automatically regardless of the chosen SQL implementation. SQLite was chosen as the back end due to its simplicity as a file based implementation of SQL, which means it does not require the installation of a SQL server. A server based SQL implementation such as PostgreSQL, MySQL, or Microsoft SQL server could be used in a future implementation as they perform better at handling data races.

Information was broken into tables which handled user information, assignments, assignment files, student files, and compiler/program output messages. User information stores the user-name and password information for students. Passwords were handled by means of salt and hash. Assignments and assignment files store information about the assignment and any files provided by the instructor. Files submitted by the student are stored in student files and are broken into what are referred to as sessions. A session is all the files submitted by the student for a single compile attempt.

Any messages generated by either the compiler or the student code are stored in messages and are also tracked in terms of sessions. Messages that refer to errors are flagged so they can be analyzed.

Text Editor

As suggested by [30], CodeMirror was used as the text editor for the website. CodeMirror is an open source text editor built for browser applications [75]. It is implemented using JavaScript and can be specialized for a variety of different programming languages, including C++, the lan-

guage taught in the experimental course. CodeMirror handles color-coding of reserved characters in a programming language, tabbing, line numbers, and other formatting generally taken care of by an IDE. It also has an extensive API that can enable auto-completion, search and replace functionality, and other useful features of IDEs. While these additional features are not implemented at the present time, they could be integrated in the future to improve the website.

Processing the Compiler Data

Compiler data were stored in the database for each student every time they compiled or ran their code. The data consisted of a session, compiler messages, whether a compile was performed, whether a run was attempted, and whether a run succeeded. The compiler messages contained error messages, console output, and default compiler messages such as “Compilation complete with code: ” or “Program completed with code:”. The data were sorted and processed to produce the outputs for the instructor’s Compiler Data page.

First, the default compiler messages were removed, as they referred to neither errors nor output. Dictionaries were created for total error, current errors, total output, current output, successful runs, failed runs, and compiles. A dictionary is a data type that connects a key and a value. In this case, the key was either the error or the output. The value was the number of times each occurred. This data type was chosen because it allowed the number of occurrences of an error or output to be looked up by the corresponding error message, and similarly with outputs. With runs and compiles, the student identifier was connected with the number of runs and compiles, again allowing for easy look up.

Old sessions were stored in a list of globally unique identifiers (GUIDs). Both sessions and student identifiers were stored as GUIDs. A GUID, sometimes referred to as a universally unique identifier (UUID), is a 128-bit integer that can be considered unique because it has a very low chance of being duplicated [76]. GUIDs were used as identifiers both to maintain unique identifiers and to remove personally identifying information from the database and protect student privacy.

Error messages were identified by looking for a string containing the word “error.” The error messages were then trimmed by a custom function that removed extraneous text, leaving only the error message. A fuzzy string comparison [77] was then used to compare the error message

to previously stored error messages. If a match was found, the count was increased for that error message in a dictionary. If no match was found, the error message was added to the total errors dictionary. If the session was the most recent session for the user, the same process was followed for the current errors dictionary.

Output was identified by a property in the message indicating that it was not an error message. Since the default compiler messages had already been removed, this indicated that the message was console output. The console output was checked against the dictionary of total outputs in the same manner as the errors, using a fuzzy string comparison. The current outputs were extracted in the same way as the current errors, again with the fuzzy string comparison.

To gather data for each student individually, namely compiles, failed runs, and successful runs, properties of the session were queried. If the session indicated a compile performed, the student ID was checked for in the compiles dictionary. If not present, it was added with a count of one. If already present, the compiles count was incremented for the corresponding student ID. The same method was used to collect runs performed and runs attempted (i.e. failed runs). If a run was successful, it was not checked if the run was attempted since it must be attempted to be successful. Therefore, the only runs marked as failed were those attempted but not successful.

After all of the data were gathered into the correct dictionaries, the dictionaries were sorted by the counts from highest to lowest. The data were stored in instances of classes created specifically for errors, output, and run information. A list was created with instances of each class. The RSP for each student was calculated at this point and stored in the correct class. A number was assigned to each student and stored in the class as well. These lists were then added to a data structure containing all of the data for the Compiler Data page.

The only compiler metric not calculated at this point is the persistence of each error. Because of how the class was structured, this is calculated when the displayed data are updated. The code iterates through each current error and finds the corresponding error in the total errors list using fuzzy string comparison. It then divides the number of occurrences of the current error over the total number of occurrences and stores this value.

A fuzzy string algorithm was used to compare errors and outputs because similar errors often vary by only a few words. For example, if a student forgot to declare the variable `x` before using it, the error message might read, "Var `x` not declared." If another student named that same

variable `y`, the error message would read “Var `y` not declared.” These error messages would not be evaluated as equal, despite the type of error being the same. This complicates the visualization because the top five errors could all be the same error with just slightly different wording. Fuzzy string algorithms calculate approximate equality between strings. The use of a fuzzy string algorithm was recommended by [4] and implemented by [24]. There are other ways of combining similar errors, such as using clustering algorithms [78] or machine learning [57]. However, these are more computationally intensive, and this system aimed to provide live feedback. Fuzzy string algorithms allowed the system to quickly combine similar errors. Console outputs had the same problem, so the fuzzy string algorithm was applied to outputs as well.

There are several fuzzy string algorithms that were investigated for implementation in the website. A C# library called FuzzyString [77] was used. It provided access to eleven different fuzzy string algorithms, including the Hamming distance, Jaccard distance, Levenshtein distance, longest common subsequence, longest common substring, overlap coefficient, etc. After testing several of these algorithms, a combination of longest common subsequence, longest common substring, and overlap coefficient was used with a normal tolerance, as defined by the package. More complex calculations like the Levenshtein distance took too long to calculate for long strings, such as the error messages, and therefore were impractical. The fuzzy string algorithms were tested on a few common error messages to make sure they were not equating errors that were significantly different from one another. They were also tested on a few output strings for the same reason. While the algorithms did not give perfect results, they correctly combined most messages and were considered effective and sufficient to meet the research objectives. This allowed the most common errors and outputs to be identified.

Visualizing Compiler Data

To visualize the compiler data collected, bar graphs were created using Plotly [79]. Plotly is a powerful visualization tool with a JavaScript API. The bar graph was chosen because it is simple and still effectively displays the data, although other types of graphs were available. Since Plotly was used, the instructor has the option to export any bar graph if desired. Plotly also automatically adds some interactivity, like mouse-over actions and zooming. In the end, Furthermore, Plotly was

chosen because of its simple implementation and powerful capabilities and it was easy to integrate into the website.

Tables were created in HTML using Bootstrap for consistent formatting.

3.3 Evaluation of the Browser-Based IDE

To make sure the website worked correctly, several preliminary tests were performed. These served the purpose of an initial load test on the database and server, as well as checking for bugs.

To test the website in both realistic and controlled environments, case studies and a simulated course experiment were performed. The case studies allowed an instructor to use the website as they wished, with few restrictions. The experiment allowed the researchers to control variables and test the impact of each. To further evaluate the effectiveness of the website's feedback, another study was run with several ME 273 teaching assistants. The case studies are described in Section 3.3.2 and the simulated course is described in Section 3.3.3.

For all of these experiments, the website was tested with course material from Brigham Young University's (BYU's) ME 273 course. ME 273 includes material on introductory computer programming as well as numerical methods, an engineering specific topic, and therefore was ideal for testing the live feedback system in an engineering environment. Additionally, ME 273 was taught by the same instructor for consecutive semesters, allowing for consistency between experiments.

3.3.1 Preliminary Tests

Preliminary tests were performed for load testing in the CAD Lab. Load testing allowed the researchers to determine how many users the website could handle simultaneously. A basic version of the website with only the tables on the Compiler Data page was used. Fourteen volunteers logged onto the website at the same time and attempted to program a Newton-Raphson algorithm to find the root of a simple polynomial function. The website and server were monitored for issues. General trends and bugs were observed and recorded. It was immediately evident to the researchers that the experimental subjects primarily programmed in Python and not C++; the

most common errors were missing semi-colons and not declaring variable types, two significant differences between Python and C++. While not statistically significant, this indicated that the data could potentially be used to determine what students were struggling with.

Further tests were performed during Spring Term 2017 in the ME 273 course. The class consisted of around thirty students. Students were asked to bring a laptop to class and log into the website. They then performed an exercise created by the instructor. At this point, editing an assignment was not enabled and the assignments were emailed to the students. Several issues were resolved as a result of both of these tests, including a CodeMirror integration problem and the website's ability to deal with changing window sizes. If the website crashed, it was either restarted immediately or the experiment was ended.

3.3.2 Case Study Design

Case studies were used to test the website in a realistic environment. The instructor and students were given access to the website and the instructor was free to use the website in any way they wanted. This allowed for observation of how the website might be used by an instructor, and what training might be needed for it to be effective. Tests were planned during Fall Semester 2017 in the ME 273 course, taught by the same instructor as Spring Term. The class contained about 90 students. Students were asked to bring a laptop to class to participate. While not every student did this, many were able to contribute. Instead of fuzzy string matching, the most common error messages were replaced with a predefined string. For example, any error message containing the phrase "variable not defined" was rewritten to say "undefined variable."

Data were collected in three forms: notes taken during the lecture period, student code recorded in the website, and a student survey at the end of the semester. Because student data were involved and because one of the researchers was a teaching assistant for the course, approval was obtained from the Internal Review Board (IRB) before the experiment began. The IRB approval documents and application are included in Appendix D. Students gave consent for their data to be used by taking the survey at the end of the semester. A research assistant unrelated to the course was assigned to remove identifying data from the survey and website before being analyzed by the researchers. No data were analyzed until grades were finalized and submitted for the semester. For additional protection of student rights, the researcher did not participate in grading for the duration

of the experiment and did not know which students had consented to be part of the experiment. This ensured that students' grades would not be affected by their participation in the experiment.

Student Code Data

Student code was collected for each assignment given during the experiment. Any errors encountered and output written to the console were stored, as well as the number of times compiled and run. Data about the success of the run were stored as well. The data were initially stored in a SQLite database. A research assistant parsed through the data to remove data for students who had not given consent.

Student Survey

At the conclusion of the semester, students were asked to complete a survey. The full survey is included in Appendix B.1. As discussed in Section 3.3.2, the survey fulfilled the dual purposes of gathering data and obtaining student consent to participate in the experiment and have their data used. The survey thoroughly explained the student's rights and available communication outlets should they need assistance.

The survey then asked students to rate their experience with the website and how well the professor used the website. They were then requested to provide any additional feedback.

3.3.3 Simulated Course Experimental Design

After the case studies were performed, a more thorough experiment was performed to obtain more data and evaluate the effect of several different variables. The experiment aimed to test four different aspects of the in-class exercises and the website:

1. The effect of having the website feedback on student quiz scores
2. The effect of having the website feedback on how well the professor knows what the students are struggling to understand
3. How error-finding exercises vs. program-writing exercises affect the usefulness of the website feedback

4. How useful the website feedback is for programming concepts vs. numerical methods problems

Simulated Course Design and Overview

A simulated ME 273 course was designed for the experiment. Eight lecture topics were chosen from ME 273: four on C++ and programming and four on numerical methods. The topics, shown in Table 4.5, were chosen based on which would lend themselves to shorter in-class programming exercises and which would work well with the limitations of the website.

The simulated course took place over the course of one month, with one or two experiments each week depending on scheduling conflicts. Twelve volunteers were recruited to be students, and one volunteer was recruited to be an instructor. The instructor was chosen from a group of former ME 273 teaching assistants, while the students were former ME 273 students. Recruiting will be discussed further in Section 3.3.3. The twelve volunteer students were separated into control and experimental groups based on schedules. The same number of graduate and undergraduate students were included in each group to maintain balance.

Each simulated course period took less than thirty minutes. The first five to seven minutes were spent in lecture, in which the instructor would review an ME 273 topic. Students were allowed to ask questions during this part of the experiment. After the lecture, students were given an in-class exercise between five and ten minutes long. Students were asked to stop after the time was up. Random assignment was used to determine the type of exercise. To simulate a large course

Table 3.2: The topics used in the simulated ME 273 course. The topic types and assigned exercise types are also shown.

Topic	Topic Type	Exercise Type
C++ Basics	Programming	Program writing
C++ Flow Control	Programming	Program writing
C++ Functions	Programming	Error finding
Roots of Equations	Numerical methods	Error finding
C++ Static Arrays	Programming	Program writing
Numerical Integration	Numerical methods	Error finding
Least Squares Regression	Numerical methods	Program writing
Initial Value Problems	Numerical methods	Error finding

where few students will ask questions, students were not allowed to ask questions during the exercise. With the experimental group, the instructor used the website to see live compiler feedback while students programmed. The instructor was allowed to address any issues he identified through the exercise. With the control group, the instructor did not have access to any feedback. Because of this, the control group's experiments were always the day before the experimental group's experiments. This ensured that the instructor was not using any information from the compiler data for the control group.

After the in-class exercise, students took a three question quiz about the day's lecture topic. The students and instructor then both took a five minute survey indicating what they were confused about during the exercise, and what the instructor believed students were confused about during the exercise. No questions were allowed during the quiz or survey.

Lecture Development

Lectures were adapted from ME 273 course materials. Since students had already taken the course, the number of slides was decreased and the material simplified to be a refresher rather than an introduction to the material. Unessential topics and examples were removed. Lectures were shortened to five to seven minutes, rather than 90 minutes.

Exercise Development

In-class exercises were developed primarily from examples provided in the ME 273 lecture slides. Modifications were made if the exercise was deemed too easy or too difficult for the five to ten minute exercise time limit.

Two types of exercises were developed: error-finding and program-writing exercises. Error-finding exercises involved giving the students already-written code with several errors, including syntax, semantic, and logic errors. Students were asked to remove the errors and provided the correct final output for comparison. Program-writing exercises asked students to write code from scratch, or nearly from scratch.

```

#include <iostream>
using namespace std;
int main ()
{
    int x = 5;
    int y = 4;
    cout << average(x y) << endl;
    return 0;
}
int average (num1, num2)
{
    return (num1 + num2) / 2;
}

```

Figure 3.13: The error-finding exercise used for Topic 3, C++ Functions. Students were told to find and fix the errors in the exercise. The corrected code should accurately find the average of 5 and 4.

Figure 3.13 shows the error-finding exercise used for Topic 3, C++ Functions. Students were given code meant to use a function to find the average of two numbers. This code contains a number of errors, shown in Table 3.3.

Figure 3.14 shows the program-writing exercise used for Topic 7, Least Squares Regression. Students were provided with a basic template and instructions and asked to write a program.

The exercise type for each lecture period was assigned at random before the experiment began, except for the last lecture period. The last period was assigned to keep an even number of each exercise type. The randomization device used was the website Just Flip a Coin [80].

Exercises were written on the website using the instructor’s “Edit Assignment” feature. Notes about limitations of the website were included with each exercise.

Table 3.3: The four errors in Figure 3.13 by line number.

Line Number	Error
N/A	No function prototype even though function is below main function.
14	Missing comma in average function call.
17	Wrong return type. Int will truncate the average.
17	No variable types given for input parameters.

```
Write a program that finds all angles for the following data points
x = [2, 4, 6, 8, 10], y = [25, 35, 4, 20, 5, 1]
// Write your code here and use the terms of y and x
#include <math.h>
using namespace std;
int main()
{
    // Write your code here
    return 0;
}
```

Figure 3.14: The program-writing exercise used for Topic 7, Least Squares Regression. Students were asked to find the linear regression coefficients for a set of points.

Quiz Development

Quizzes were developed from the ME 273 slides and were written in Google Forms. There was one quiz for each lecture topic, and each had three questions. For programming topics, these questions followed a simple pattern: Q1) Choose the valid syntax, Q2) What is the output of this code segment, Q3) What is the error in the following code. Pictures of code written in Visual Studio were provided for the output and error finding questions. Because the compiler provides visual feedback indicating where errors occurred, the images were edited using Microsoft Paint to remove those visual indications. For numerical methods topics, questions involved identifying the correct method and selecting strengths and weaknesses of a method.

Recruiting

The instructor was recruited from a set of former ME 273 teaching assistants, ensuring that they had adequate exposure to the material and was accustomed to explaining it.

Students were recruited from a group of researchers in the BYU CAD Lab due to their programming experience; in order to gather sufficient data on errors, students had to be competent enough to attempt the in-class exercises after only a brief refresher. The CAD Lab students include undergraduate, masters, and doctoral students. All had taken ME 273 before participating in the

Which of the following is valid syntax for a do while loop?

```
#include <iostream>

using namespace std;

int main()
{
    int i = 0;

    do
    {
        cout << i++ << endl;
    } while (i < 5)

    return 0;
}
```

Option 1

```
#include <iostream>

using namespace std;

int main()
{
    int i = 0;

    do
    {
        cout << i++ << endl;
    } while (i < 5);

    return 0;
}
```

Option 2

```
#include <iostream>

using namespace std;

int main()
{
    int i = 0;

    do {
        cout << i++ << endl;
    }
    while (i < 5)

    return 0;
}
```

Option 3

Figure 3.15: A type 1 quiz question for Topic 2, C++ Flow Control. Option 2 is the correct answer. A semi-colon is needed after the while statement.

experiment. Students were recruited through an email explaining the time commitment required and the overall purpose of the experiment. Students were then selected based on availability.

Twelve students were recruited overall. Six were placed in the control group, and six in the experimental group. Group assignment depended solely on scheduling availability. There were two masters students in each group and four undergraduate students.

What is the output of the following code?

```
#include <iostream>

using namespace std;

int main()
{
    int x = 5;
    int y = 17;

    for (int i = 5; i > 1; i--)
    {
        if (x < y && x + y <= 25)
        {
            cout << x << " " << y;
            x += 1;
            y -= 3;
        }
    }

    return 0;
}
```

5 176 147 118.89 5

5 176 147 118 8

5 176 147 11

5 176 14

Figure 3.16: A type 2 quiz question for Topic 2, C++ Flow Control. Option 3 is the correct answer. The conditional statement is not true when x and y both equal 8.

One student was excluded due to his extremely high skill level in computer programming. It was determined that he would not be reasonably representative of the target level of programming experience.

Data Collection

Before the experiment began, participants were asked to complete a Qualtrics survey evaluating their programming skill. Feigenspan and others have found that self-evaluation is as accurate as testing students to determine their programming skill [81]. The survey used was derived from [81] and is shown in Table 3.4. A few questions were modified from [81] to better fit a mechanical engineering environment, including asking about MATLAB, Python, and object-oriented programming instead of logic programming.

What is wrong with the following code?

```
#include <iostream>
using namespace std;

int main()
{
    int x = 5;

    switch (x)
    {
        case 0:
            cout << "0";
        case 1:
            cout << "1";
        case 2:
            cout << "2";
        default:
            cout << "Default";
            break;
    }
    return 0;
}
```

break is missing from cases 0, 1, and 2.

there's no semi-colon at the end of the switch statement

parentheses are missing: case(0): instead of case 0:

Figure 3.17: A type 3 quiz question for Topic 2, C++ Flow Control. Option 1 is the correct answer. Break statements are needed in each case in a switch statement.

Feigenspan found that questions 3 and 8 were the best indicators of programming skill and performed a regression with those questions [81]. Since he validated his findings, his regression parameters were used. Equation 3.2 shows the regression used to estimate student programming skill.

$$skill = .441Q3 + .286Q8 \quad (3.2)$$

The audio of each lecture was recorded using a cell phone. With the audio recorded, differences in the lectures between groups could be determined and accounted for.

OCAM, a free screen capture tool, was used to record each student's screen as they performed the programming exercises [82]. OCAM was also used to record an additional computer

screen with the website Compiler Data page. No audio was captured in these videos as students were working separately and not allowed to communicate with one another.

During the lecture, a Google Sheets spreadsheet was used to record any questions the students asked. Questions were organized by topic.

Students took a quiz after each exercise using Google Forms. The link to the Google Form was sent out before each class period in an email.

After the quiz, both students and the instructor took a Qualtrics survey to capture “points of confusion,” or what the students struggled with during the class period. The responses were collected in free response form.

After the entire experiment was over, participants were asked to take another Qualtrics survey rating the website. The instructor was also asked to rank the graphs provided on the website Compiler Data page and give feedback on additional outputs and features that could be implemented. Students were asked about the instructor’s behavior to determine if there was a difference with the live feedback.

Hypotheses

The simulated course experiment tested four hypotheses.

1. Student quiz scores will be higher in the experimental group, when the instructor has access to the website feedback

Table 3.4: Survey questions from the programming skill self-evaluation survey.

	Question	Type
1	For how many years have you been programming?	Text
2	How do you estimate your programming experience?	1 to 10
3	How do you estimate your programming experience compared to your classmates?	1 to 10
4	How experienced are you with C++?	1 to 10
5	How experienced are you with MATLAB?	1 to 10
6	How experienced are you with Python?	1 to 10
7	If you are very experienced with any other languages, please list them here.	Text
8	How experienced are you with object-oriented programming?	1 to 10

2. The instructor will better predict what students are struggling with when the instructor has access to the website feedback
3. When an error finding exercise is paired with a programming topic, the difference in student quiz scores will be more significant between the experimental and control groups than when error finding exercises are paired with a numerical methods topic
4. When a program writing exercise is paired with a numerical methods topic, the difference in student quiz scores will be more significant between the experiment and control groups than when program writing exercises are paired with a programming topic

3.3.4 Teaching Assistant Study

A third study was conducted to further evaluate the effectiveness of the website feedback. Since the simulated course experiment only involved one instructor, the teaching assistant study was designed to involve multiple instructors to gather more data.

Experimental Design

Former and current ME 273 teaching assistants (TAs) were recruited. TAs were sent an email explaining the experiment and asking them to volunteer for 30 minutes. Seven teaching assistants participated in the experiment.

Experiments were completed on an individual basis and lasted 30 minutes. Participants were given 10 minutes to go over two exercises given during the simulated course experiment. Participants could complete the exercise, check their solutions, and ask the proctor questions. After the ten minutes, participants were given a brief overview of the Compiler Data page of the website. They then watched recorded video of the Compiler Data page from the simulated course experiment. After watching the video, the participants were given access to the website with the final data on the Compiler Data page. The participants were asked to complete a Qualtrics survey. The survey combined questions from the skill evaluation survey, the points of confusion survey, and the instructor survey from the simulated course experiment. Additionally, participants were asked about their experiences teaching ME 273.

Exercises

Two exercises were selected from the eight used in the simulated course experiment. Exercises were chosen to allow for comparison between a C++ topic using an error-finding exercise and a numerical methods topic using a program-writing exercise.

The first exercise lasted for approximately five minutes and dealt with C++ Functions. Students were given code that used a C++ function to find the average of two numbers and were asked to remove any errors. Four errors had been added by the researchers. Table 3.3 shows each of the errors, and Figure 3.13 shows the exercise.

The second exercise lasted for approximately ten minutes and dealt with Least Squares Regression, a numerical methods topic. Students were given a set of points and asked to determine the coefficients for a linear regression. They were given the formula, but had to translate it into code. The template is shown in Figure 3.14. The correct coefficients were provided to the TAs.

Videos

The videos used in the experiment were recorded using screen capture software. Because the video quality was not perfect, the TAs were also allowed to use the Compiler Data page on the website if they could not clearly make out words in the video.

Survey

The participants were asked to complete a survey evaluating their programming skill level using the same questions discussed in Section 3.3.3. The survey then asked the participants to explain what students struggled with during the exercises. The survey then had the participants evaluate the different metrics and graphics of the website and provide feedback on their experiences in ME 273. A list of all survey questions is provided in Appendix B.3.

CHAPTER 4. RESULTS

4.1 Case Study Results

The experimental website was used in an ME 273 lecture several times during the Fall 2017 Semester. The instructor, Dr. Salmon, was allowed to use the website however he saw fit to explore how an actual instructor might use the website. Observations were noted by the researcher, who attended these lectures. After the semester ended, students who agreed to participate in the experiment filled out a survey about the website.

4.1.1 Observations

During one class period when the experimental website was used, students were asked to write a short program with a FOR loop that added a small number to a variable 50 times. While the students worked on the exercise, the instructor observed the Compiler Data page of the website. The instructor identified the most common errors, one of which was “variable is not declared in scope.” After allowing the students to work on the exercise for a few minutes, the instructor paused the exercise and discussed variable scope. Variable scope is not explicitly covered as a topic in ME 273, although it is mentioned when loops and control statements are discussed. The website data allowed the instructor to quickly identify that students did not fully grasp the concept of scope. He was then able to further explain the concept.

The instructor used the website in two ways. He generally provided an initial template or example with which students could start. He then either had students follow along as he completed the exercise or had students attempt the exercise themselves. Sometimes these methods were combined, with the instructor providing some of the code, and then allowing students to complete a short section on their own.

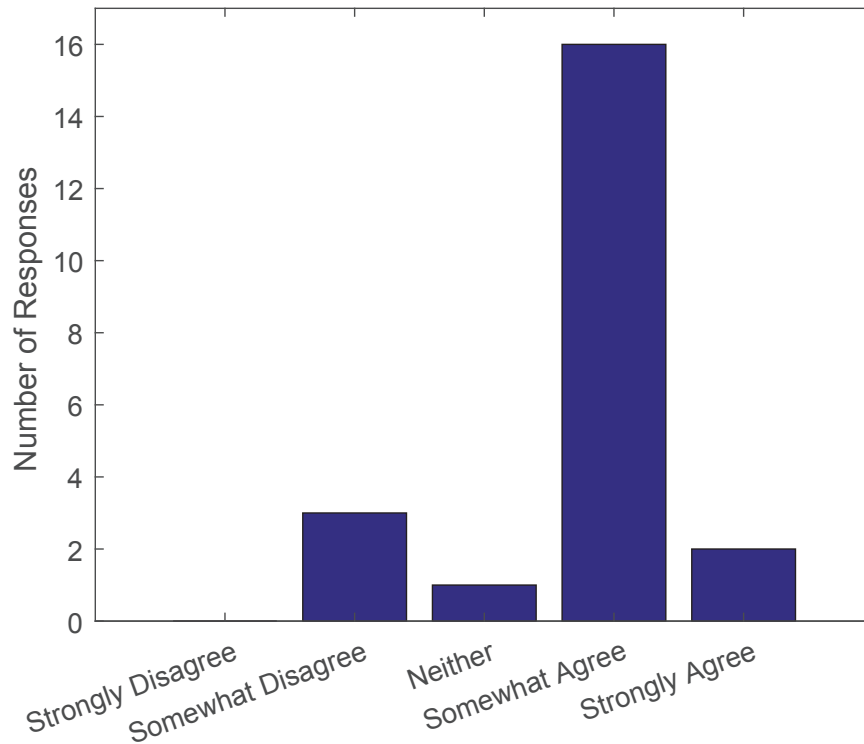


Figure 4.1: Student responses to the question “The website was easy to use”.

One student happily exclaimed “That’s me!” when a specific error was discussed. He was pleased to have personal interaction during a large lecture and excited to have his problem addressed. Individualizing education is often considered the ideal, but is nearly impossible in large courses. Having data from each student can start to address this issue.

4.1.2 Survey Results

28 students responded to the survey. 22 of those students indicated that they used the website during the course. 81% of students surveyed either somewhat or strongly agreed that the website was easy to use, shown in Figure 4.1. Making the website easy to use was important to reduce the barriers to implementing it in a real course. Only 36% of students found the website enjoyable to use, with 45% indicating no opinion, shown in Figure 4.2. Whether or not the website is enjoyable is less significant than its usefulness.

54% of students found that the website made it easy to do in-class exercises, as shown in Figure 4.3. The comments section of the survey indicates that students who disagreed did so due to the bugs and technical issues encountered during the course of the experiment. These technical issues have been resolved in later versions of the website, which could increase the percentage of students believing that the website makes it easy to do in-class exercises. As this is one of the website's primary purposes, this is an important statistic.

63.6% of students indicated that the instructor changed his lecture based on the website feedback, shown in Figure 4.4. This indicates that the majority of students felt that the instructor was responding to the website feedback. The instructor was therefore able to glean some meaning from the website feedback. However, only 54.5% of students felt that the instructor was able to address confusion in class using the website feedback, shown in Figure 4.5. More work needs to be done to translate the instructor's observations from the website feedback into effectively addressing the issues in class.

Overall, the results indicate that the website is easy to use and can enable in-class exercises, especially when improvements have been made to remove bugs. The instructor did noticeably change his lecture when he used the website feedback. However, whether or not he was able to address student confusion is less clear. More students agreed than disagreed, but more than a third of students indicated neither agreement nor disagreement. For all five questions, only one student indicated "strongly disagree," and it involved bugs that have been remedied in the later versions of the website. These results suggest that the website is easy to use, lowering the barrier to implementation, and potentially useful. The usefulness could be improved through instructor training and further use.

4.1.3 Exploratory

Each student's code for each exercise, each compile attempt, its success, error messages, console outputs, and time stamps were recorded in a database. These data were explored to get a better picture of how ME 273 students were performing on the in-class exercises.

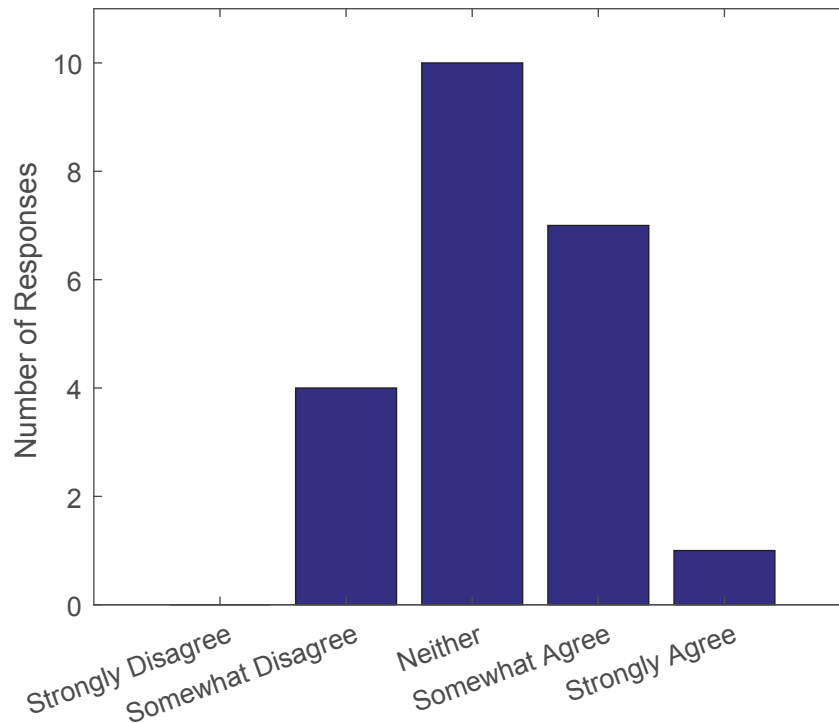


Figure 4.2: Student responses to the question “The website was enjoyable to use”.

Total Errors

Overall, there were 784 compiler errors that occurred during the five different exercises. The most common errors are shown in Table 4.1. 392, or 50%, of these errors dealt with variable scope, or the variable not being declared where it was being used. The next most common error type was missing semi-colons, at 87 instances or 11% of total errors. Declaring variables and consistently placing semi-colons are two fundamentals of C++ that are not necessary in Matlab. Because the students study both languages as part of the current curriculum, it is possible that these differences are causing them to commit more errors.

“Expected primary expression” took up about 7% of the total errors. This error generally occurs when the user incorrectly uses a control statement, such as an *if-statement* or *if-else-statement*. This indicates a lack of understanding of the syntax used for control statements.

“Unqualified-id” errors were 4.8% of the total errors. This often occurs when a user puts a semi-colon in an incorrect location, as opposed to omitting a semi-colon. Remembering where to

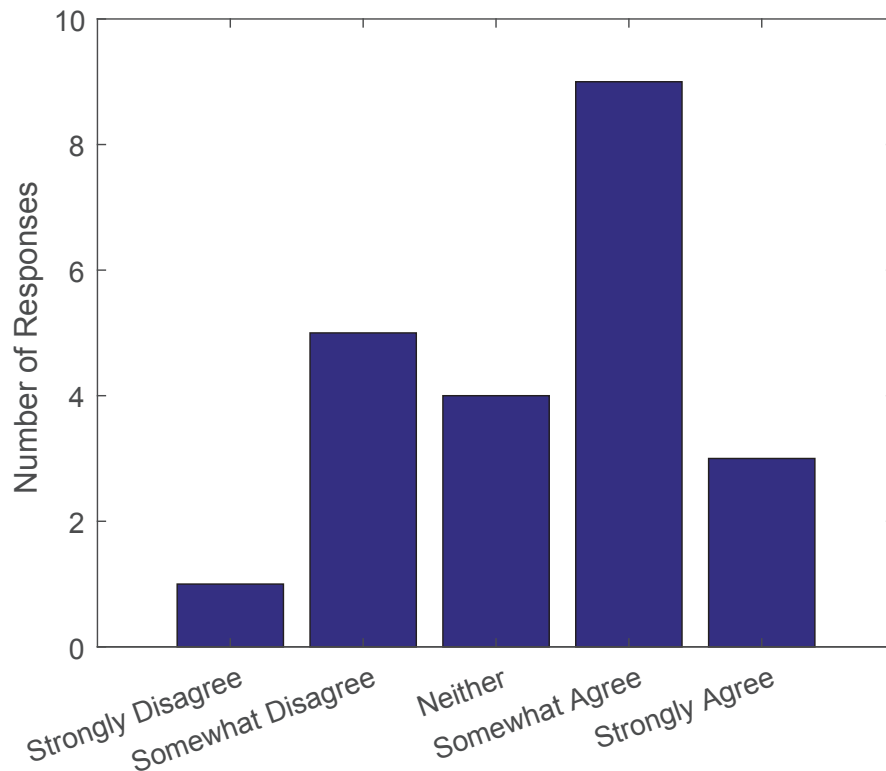


Figure 4.3: Student responses to the question “The website made it easy to do in-class exercises”.

place semi-colons is one of the more difficult parts of C++ syntax for a beginner, since semi-colons are required on all statements but not after certain curly brackets or pre-processor directives.

Omitting a curly bracket caused 4.5% of total errors. This may have been less present in a Visual Studio or professional IDE environment since they generally handle automatically adding brackets. However, it is still a common error.

Table 4.1: Total errors for the case study. The top five errors are listed with their counts and percentages of the total errors.

Error	Count	Percentage
Scope	392	50
Semi-colons	87	11.1
Expected primary expression	55	7
Unqualified-id	38	4.8
Missing curly brackets	35	4.5

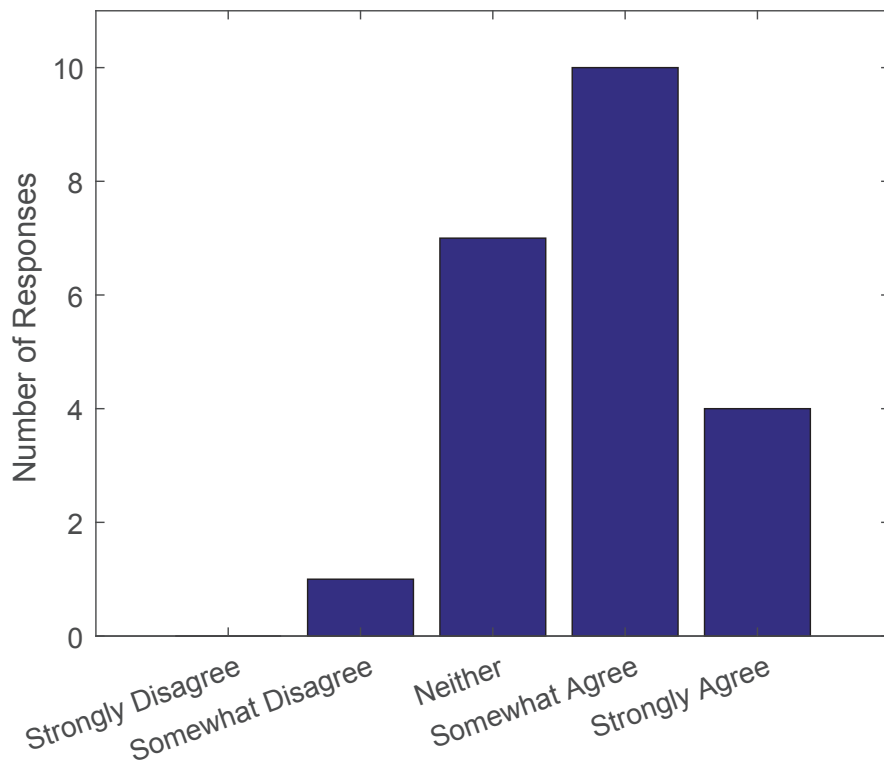


Figure 4.4: Student responses to the question “The instructor changed his lecture based on the feedback”.

In 2006, Jadud recorded the compilation behavior of students in an introductory computer science course (CS1) over two years. CS1 was taught in Java, which is fairly similar to C++ in terms of syntax [54]. He found that the three most common errors were “unknown variable”, “semi-colon”, and “bracket expected”, in that order. These results are very similar to those observed in ME 273. The most common error, dealing with variable scope, is the same as “unknown variable” because they both deal with undeclared variables. The difference in wording is likely due to the different programming languages. The second most common error is the same, dealing with semi-colons.

The third most common errors found by Jadud were bracketing errors, generally dealing with a missing bracket. “Expected primary expression” is different from a missing bracket, but generally involves something being misplaced in relation to a bracket. While omitting a curly bracket is slightly less common than an unqualified-id error, when all bracketing errors are combined (parentheses and square brackets included), they take up 6.25% of all errors, becoming the

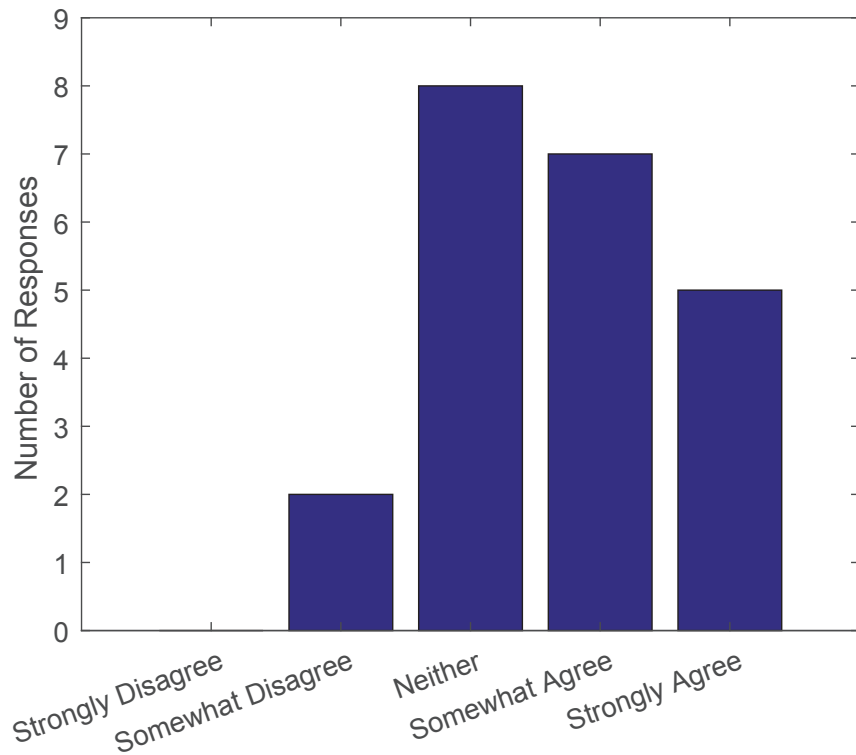


Figure 4.5: Student responses to the question “The instructor was able to address student confusion based on the feedback”.

fourth most common error. It is possible that Jadud classified unqualified-id errors as semi-colon errors, which would similarly place bracketing errors in fourth place.

This suggests that mechanical engineering students generally face the same issues as introductory computer science students, despite potentially having different backgrounds and skill sets. It appears that novice programmers struggle with the same syntax issues regardless. This information could be motivational to students who feel they are struggling alone. Most students struggle with these concepts.

The main difference between these results and Jadud’s is the “expected primary expression” error. Primary expression errors generally have to do with control statements, loops, brackets or semi-colons. It is possible that these errors would have been classified differently in Jadud’s system. However, it is also possible that mechanical engineering students struggled more with the syntax involved with if statements and loops than computer science students. This would need further analysis to prove if there is actually a difference.

Errors by Exercise

Four exercises were performed in the ME 273 course using the website. Each exercise will be explained and the errors for each will be discussed.

Exercise 1: FOR Loops In the first exercise, the instructor asked students to write a FOR loop adding a very small number to the initial value of one 50 times. No template code was provided for this exercise. 16 of the 22 students participated in this exercise.

84 errors occurred during the exercise. 31% of the errors, or 26 instances, were scope errors, where a student either forgot to define a variable before using it or defined it in the wrong location. This mirrors the overall results from all of the exercises, where the most common errors were scope related. Seven different students contributed to the 26 scope errors. 28.6% of errors were missing semi-colons. Eight of the 14 students had this error at some point during the exercise.

One student struggled with defining the counter variable inside the FOR loop, with eight instances of the error before it was fixed. There were nine “unqualified-id” errors, all of which dealt with declaring *int main()*. Two students either forgot to declare *int main()* or declared it incorrectly, without curly brackets. There were five primary-expression errors, all of which related to one student using commas instead of semi-colons in a FOR loop. Two students declared the same variable twice, causing a re-declaration error.

Overall, students struggled the most with basic syntax. Simply remembering to declare a variable before using it appears to be a barrier to running code correctly. Remembering semi-colons and using them in the appropriate places is the next most significant issue.

Of the 16 participants, eight were able to fix their errors while eight were not. The average number of compiles is 4.31 per student. Successful students compiled 5.875 times on average while unsuccessful students compiled on average 2.75 times. Using a Welch’s t-test to account for unequal standard deviations results in a two-sided p-value of .056, showing suggestive but inconclusive evidence that there is a difference between the successful and unsuccessful groups based on how many times they compiled. Figure 4.6 shows the average compiles by whether or not the student was successful, with the standard error shown for each group.

The two most common errors were experienced by approximately 50% of the participating students. Errors after the two most common applied to only one or two students in this sample. Further research would be required to determine how common an error needs to be for it to be

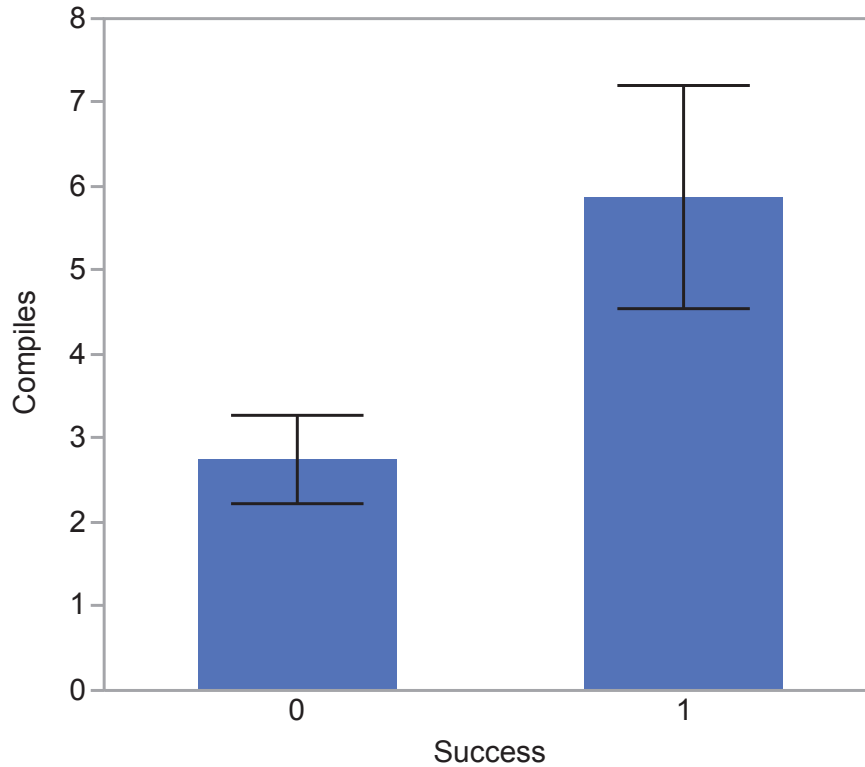


Figure 4.6: Mean compiles vs. success for exercise 1 with standard error shown

addressed in a group setting. A double bar graph indicating the number of occurrences of the error alongside the number of users experiencing the error could potentially address this problem.

Exercise 2: Root Finding Exercise 2 was a root finding problem. The instructor provided pseudocode that was not syntactically correct but provided a framework to solve the problem. The code included a function and the algorithms for the bisection and false position root finding methods. Students followed along as the instructor programmed for much of the exercise, explaining why many of the students have the same errors. 13 of the 22 users participated in this exercise.

In total, 407 errors occurred during this exercise. The high number of errors is likely due to the students following along with the instructor, who made some errors on purpose to explain how students should debug. Additionally, there were errors included in the pseudocode provided by the instructor, an unfinished *if-statement* and an undeclared variable.

287, or 70.5% of the total errors that occurred during this exercise were scope errors. 11 of the 13 students experienced scope errors. Generally, the students added new variables without

declaring them first. The next compile sometimes showed the students fixing this problem. Even more scope errors then occurred when the next variables were added. 10.3% of errors were primary expression errors, likely due to an empty *if-statement* included in the pseudocode. Despite the fact that it was already present, only nine students experienced this error, indicating that four students were able to fix the error. There were 22 unqualified-id errors accounting for 5.4% of total errors, which six students experienced. There were 15 missing semi-colon errors at 3.7% of total errors. Only four students had semi-colon errors.

The students who compiled the most, with 9 compiles, were closest to completing the exercise. Each only had one or two errors left to fix and had steadily decreased their number of errors throughout the exercise.

Exercise 3: Arrays Exercise 3 was an array problem. The instructor provided a template with an array and instructions to print out the array, print it out in reverse, replace the spaces with dashes, and print out the final array. 16 of the 22 participants completed the exercise.

There were 70 total errors during this exercise. There were 13 scope errors at 18.6% of the total errors for the exercise, with seven students experiencing the scope errors. The next most common error related to the *cin* and *cout* operators, with 11 occurrences. However, these errors were all produced by one student. The student eventually fixed one instance of the error but never remedied the second. The next most common error type was the primary expression error, with eight instances and four students experiencing the error.

Other common errors included empty character constant, unqualified-id, brackets, and semi-colons. With the exception of the empty character constant errors, these errors have been common throughout all of the exercises. The empty character constant errors dealt with incorrectly assigning a character to the array. Since this exercise related directly to arrays, the topic at hand, it is only logical that this error occurred primarily in the array exercise.

On average, students compiled 6.6 times, ranging from one time to 17 times. All three students who compiled the most, with 10, 11, and 17 compiles, completed either two or three parts of the exercise.

15 of the 16 participants were able to remove all errors from their program at some point, allowing for an analysis of which parts of the exercise they were able to complete based on the outputs. 15 of the 16 participants were able to print out the array. Six participants were able to

print out the array in reverse, and five were able to replace the spaces with dashes. Three students were able to complete all parts of the exercise, and five were able to complete two of three parts of the exercise.

Exercise 4: Numerical Integration The exercise provided a function and two vectors, one with x values and another with function values. Students were asked to numerically integrate the function between the first and last provided x values using both the trapezoidal and Simpson's rules. 15 of the 22 participants completed Exercise 4.

A total of 222 errors occurred during this exercise. There were 66 scope errors, again the most common error at 29.7% of total errors. Nine of the 15 participants had a scope error at some point. 20.7% of errors were missing semi-colon errors, with 46 occurrences and eight students experiencing them. The next three most common errors all had 18 occurrences from five students: incomplete line, vector class, and missing parentheses. The average number of compiles per student for this exercise was 8.3, ranging from one to 17.

Again, the most common errors line up with the overall most common errors with the exception of vector errors. This is once again logical, as this is the first exercise where vectors were used. The incomplete line error refers to missing components of lines.

There is a statistically significant difference using a rank-sum test between the number of compiles for successful and unsuccessful students, with a p-value of .0015. Successful is used here to define students who were able to remove syntax errors from their program and obtain an output, not necessarily the correct outputs. Figure 4.7 shows the average number of compiles based on success with the standard error shown.

These results are similar to those for Exercise 1. Exercises 2 and 3 were not analyzed due to no students completing Exercise 2 and nearly every student completing Exercise 3. Exercise 2 was more lengthy than the other exercises and ended early due to limited time during class. It appears that students who fail to remove errors compile infrequently, with a low standard deviation between students. Students who successfully remove errors have a higher standard deviation of the number of times they compile, but a higher average number of times compiled. It appears that a higher number of compiles can indicate success, while a lower number most likely means failure.

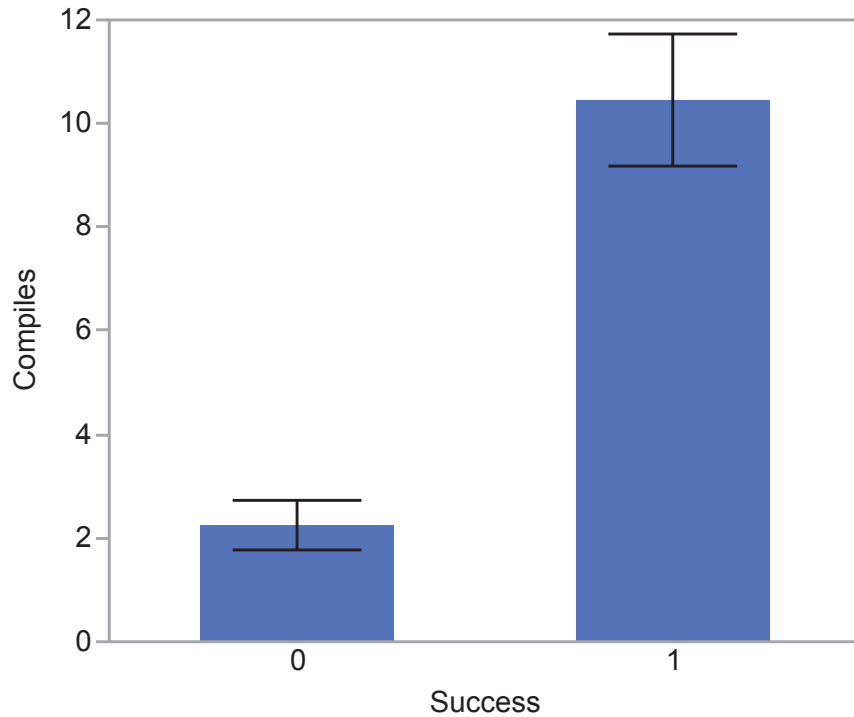


Figure 4.7: Mean compiles vs. success for exercise 4 with standard error shown

4.2 Simulated Course Results

The website was used in a simulated ME 273 course experiment. Students were divided into two groups, the experimental group and the control group. One subject was chosen as an instructor. The instructor gave the same lecture to each group. Each group then performed a short, 5-10 minute exercise using the website. With the experimental group, the instructor used the website feedback to address problems the students were having. With the control group, the instructor did not have access to the website feedback.

After the exercise, students took a brief quiz and answered survey questions about what they struggled with during the experiment.

4.2.1 Observations

Notes were taken on the questions asked by each student during the lecture, as well as the number of suggestions made by the instructor while using the website feedback. For Topics 3 and

5-8, the instructor was able to make suggestions to the class based on the website feedback. This feedback generally included gentle reminders to use semi-colons, define variables in the correct scope, and declare the size of an array. As the experiments went on, the instructor became more comfortable giving suggestions to the students. He also began looking up errors online when he could not immediately determine the likely cause of the error.

One student admitted that he felt uncomfortable admitting he had produced an error when asked by the instructor. This may have been due to the fact that the students and instructor are peers, rather than students and their superior. Additionally, the test subject students already knew how to program and likely felt that they should know the material. The uncertainty of students sharing their errors could be addressed through training or an option to opt out of a session.

4.2.2 Quiz Results

Students took a three question quiz after each exercise. The quiz scores ranged from zero to three, with one point allocated for each question. The average score for the experimental group was higher than the average score for the control group by .2, or a .06% difference. As shown in Figure 4.8, there is a difference between the two groups, but not a significant one. Additionally, a difference of .06% is not significant in practice. A larger sample size would be needed to demonstrate significance. This result addresses Hypothesis 1, “Students will perform better on quizzes when the instructor has access to live compiler feedback.”

A regression of the quiz scores and student skill in each group showed a correlation between score and skill. This indicates the quizzes accurately represented the students’ pre-existing skill level, providing evidence that the quiz scores are not random. The distribution of skill between groups was not equal, as students were assigned to groups based on schedule and not randomly, as well as the limited size of each group. Skill is somewhat normally distributed in the control group, but skewed or binomial in the experimental group.

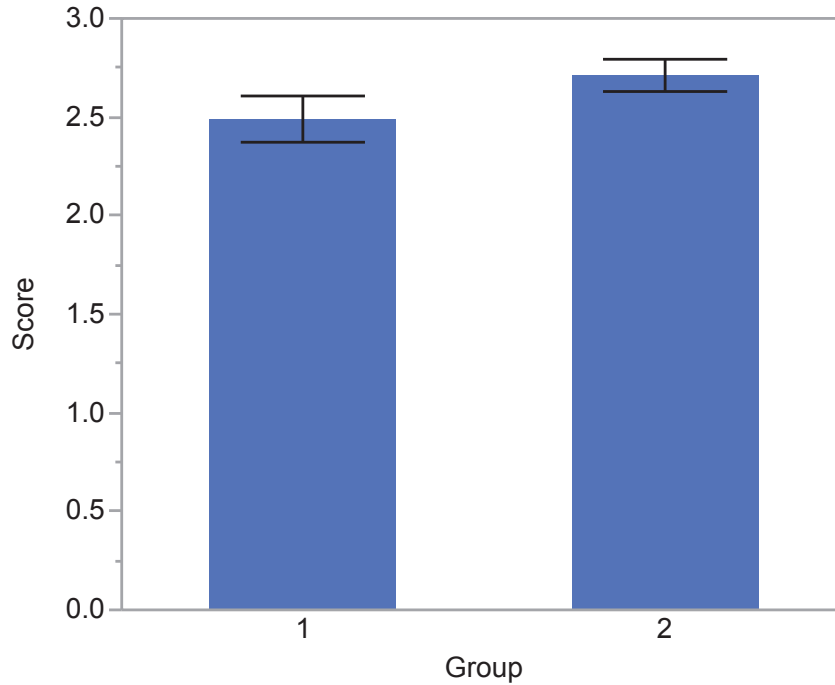


Figure 4.8: Mean quiz scores by group with standard error shown

4.2.3 Points of Confusion Results

After each quiz, students indicated what they were confused about during the exercise on a survey. The instructor also indicated what he believed the students were confused about during the exercise on a survey.

Several students misunderstood the survey question and responded with anything they were confused about during the experiment, rather than the in-class exercise. Due to this issue, the results were analyzed both from the survey results and from watching each student's recorded programming session and collecting issues.

Overall, the average percentage of issues the instructor identified was higher for the experimental group. However, the difference is not statistically significant using a Student's t-test.

For program writing exercises, the percentage identified was higher for the experimental group, but not significantly, with a two-sided p-value of 0.29. For error finding exercises, the percentage was slightly higher for the control group, but extremely insignificantly with a two-sided p-value of 0.55.

For C++ questions, the experimental group had a higher percentage of issues identified than the control group by 20%. However, this is insignificant with a two-sided p-value of 0.32. For numerical methods questions, the percentage correct is not significant.

There is only one data point for a C++ topic matched with an error finding exercise. However, in this instance for the control group the instructor predicted 0% of the students errors, but for the experimental group he predicted 100% of the students errors. For C++ with program writing, there are three examples. In each case, the instructor identified more issues with the control group. This could suggest that error finding exercises are better suited for C++ topics, as hypothesized.

There is also only one data point for a numerical methods topic matched with a program writing exercise. There is a large difference here as well, with 60% of issues identified for the experimental group to 33% of issues identified for the control group. For a numerical methods topic with an error finding exercise there are three examples. For the first exercise, the percentage of issues identified is the same for both groups. For the second, the experimental group has a slightly higher percentage of issues identified. For the last, the control groups has a significantly higher percentage of issues identified.

4.2.4 Survey Results

Both the students and the instructor took a survey at the end of the exercise to evaluate the website. The results are summarized in the following sections.

Student Responses

There was no statistically significant difference in student responses based on group for any survey question.

The average score on a scale of 1 to 7, 1 being strongly disagree and 7 being strongly agree, was 6.083 for “The website is easy to use”. A score of 6 corresponds with “agree”, indicating that on average, students believed the website was easy to use. This agrees with the results from the ME 273 case study. The responses are shown in Figure 4.9.

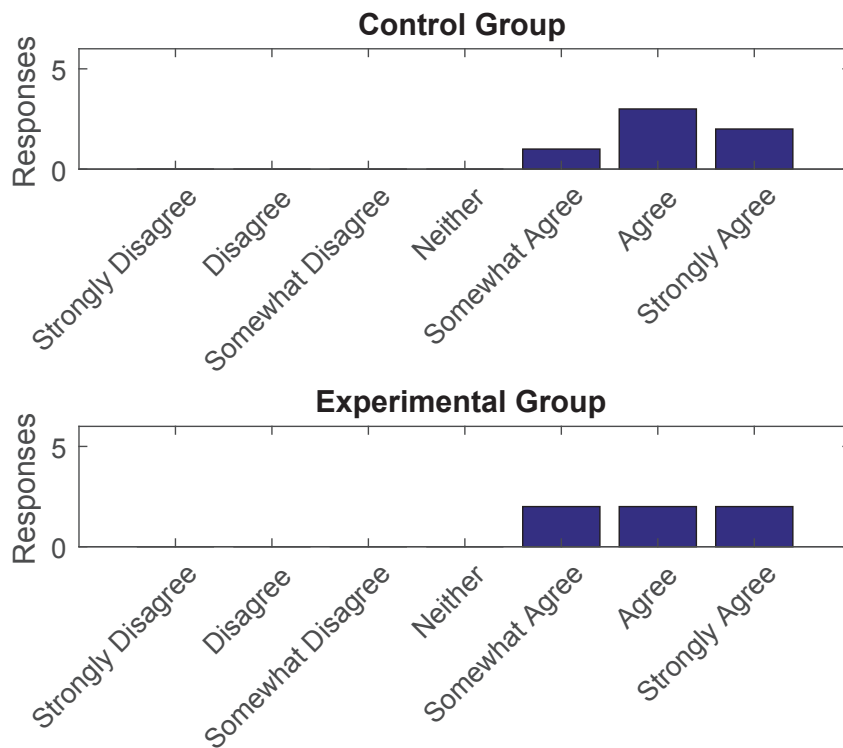


Figure 4.9: Student responses to the question “The website was easy to use” by group.

The average score for “The website is enjoyable to use” was 5.25, falling somewhere between “somewhat agree” and “agree”. Students somewhat enjoyed using the website. As shown in Figure 4.10, two students, one from each group, did disagree.

The average score for “The website made it easy to do in-class exercises” was 5.83, indicating “agree”. Enabling in-class exercises is one of the primary purposes of the website, and these results suggest that the website is successful in doing so. The responses are shown in Figure 4.11 and show that all participants either agreed or had no opinion on the statement.

The average score for “The instructor addressed issues during the exercise” was 5.167, a little higher than “somewhat agree”. The control group was the only group that had any students disagree with the statement, as shown in Figure 4.12. Allowing the instructor to address student issues during an in-class exercise was another primary purpose of the website, and these results suggest that the website has the potential to do so. The instructor improved as the course went on, indicating that training may improve the instructor’s ability to address issues in class based on the website feedback. Additional features and enhancements could also improve this score.

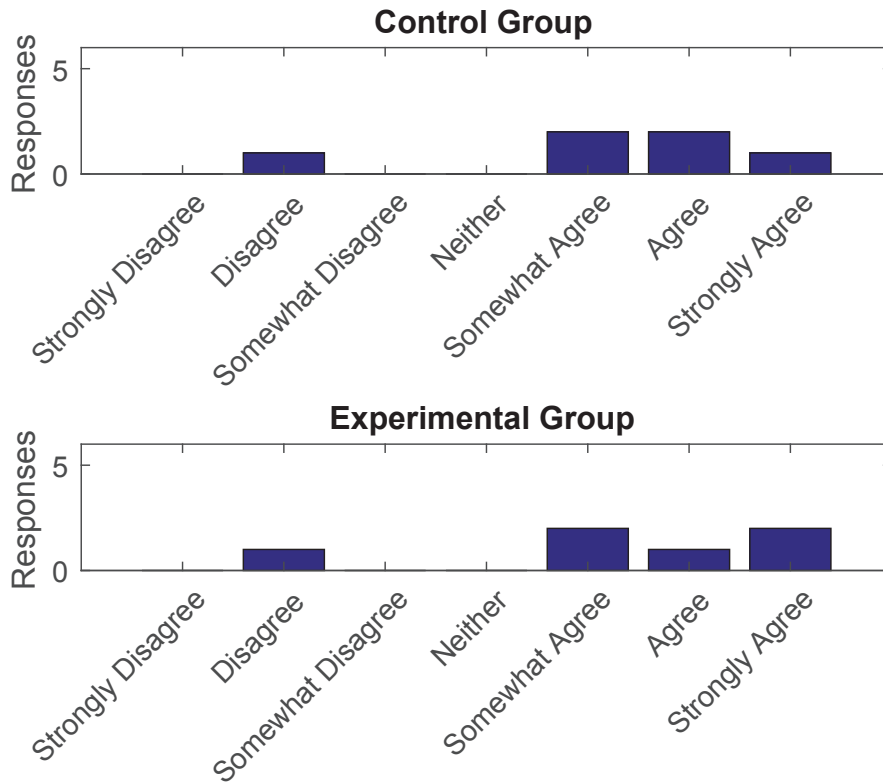


Figure 4.10: Student responses to the question “The website was enjoyable to use” by group.

The average score for “The instructor was aware of how the students were doing on the exercise” was 4.83, approximately a “somewhat agree”. The website was intended to improve instructor awareness of how the students were performing real-time, during class. Interestingly, while there was no significant difference between groups, the average for the control group was slightly higher. One more student disagreed in the experimental group than in the control group. A larger sample size would likely be needed to validate these results. The responses are shown in Figure 4.13.

The average score for “I enjoyed doing exercises on the website” was 5.5, halfway between “somewhat agree” and “agree”. The results are shown in Figure 4.14. One of the original hypotheses was that students would enjoy using the website. This has been somewhat confirmed.

The average score for “Doing in-class exercises helped me understand the material” was 5.83, approximately an “agree”. Combining this information with the comments in the open re-

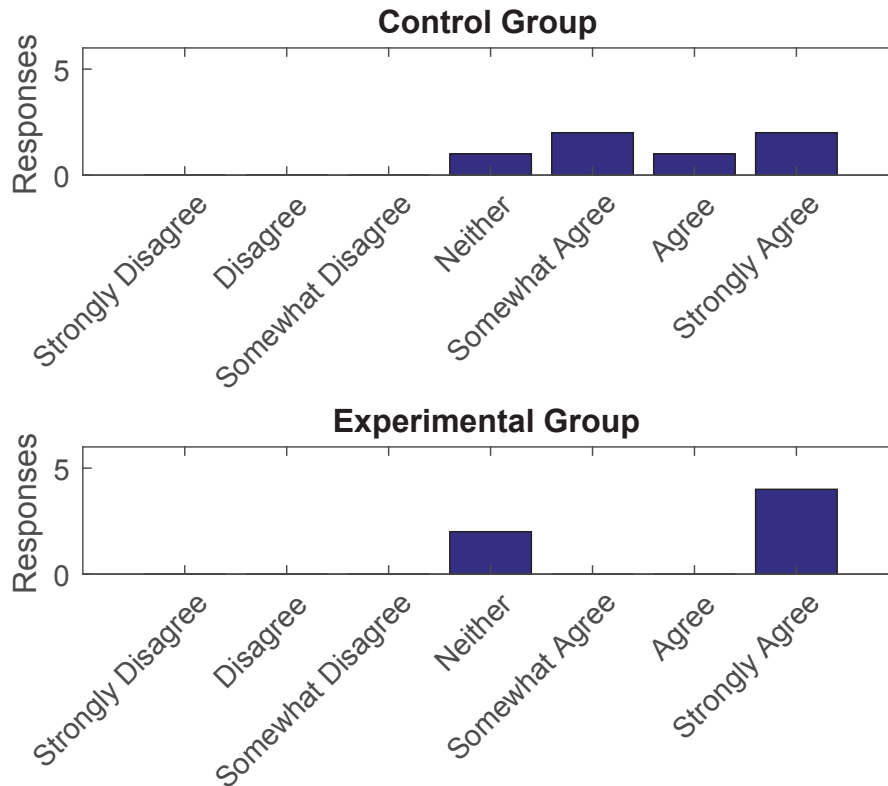


Figure 4.11: Student responses to the question “The website made it easy to do in-class exercises” by group.

response question in the survey, students generally appreciated doing the in-class exercises. The responses are shown in Figure 4.15. Only one student disagreed with the statement.

Students were asked what features they would like to be added to the website to improve it. The requests are summarized in Table 4.2. The most common requests asked for features common in professional IDEs, such as keyboard shortcuts, autocomplete, and the highlighting of syntax errors. CodeMirror, the text editor used for the website, has the ability to implement some of these features, such as autocomplete and highlighting errors. However, the implementation varied between browsers and therefore have not yet been implemented.

The next most popular request was for hints when students get stuck. This feature has been explored in other literature [23, 65] and can be helpful. Students also requested a debugger. The other requests were only made by one student but will be discussed further in Chapter 5.

At the end of the survey, students were asked for any further input. A participant from the experimental group said,

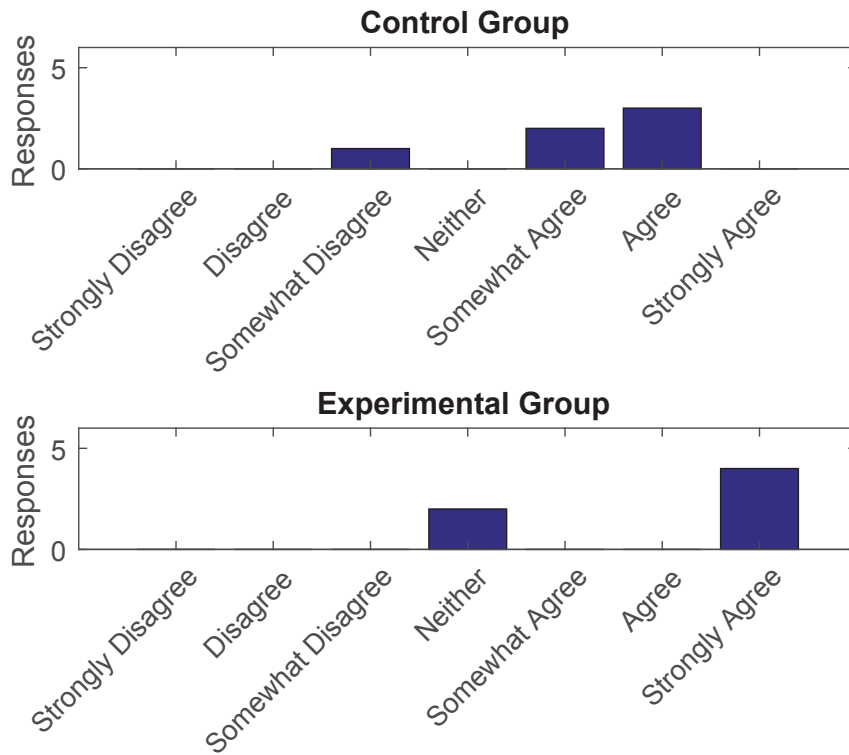


Figure 4.12: Student responses to the question “The instructor addressed issues during the exercise” by group.

“I thought this experiment was really beneficial. Other engineering classes can have a different approach - a student can go to class and comprehend the main principles and concepts, then take them home and successfully apply them to their assignments. However, in programming, a student can understand principles and concepts, but so

Table 4.2: Student requests for website enhancements

Students	Request
4	IDE enhancements (hotkeys, autocomplete, highlighting syntax errors)
3	hints when errors are not fixed
3	debugger
1	user input
1	data sent between compiles
1	larger code window
1	allow students to post questions
1	allow students to see others' errors

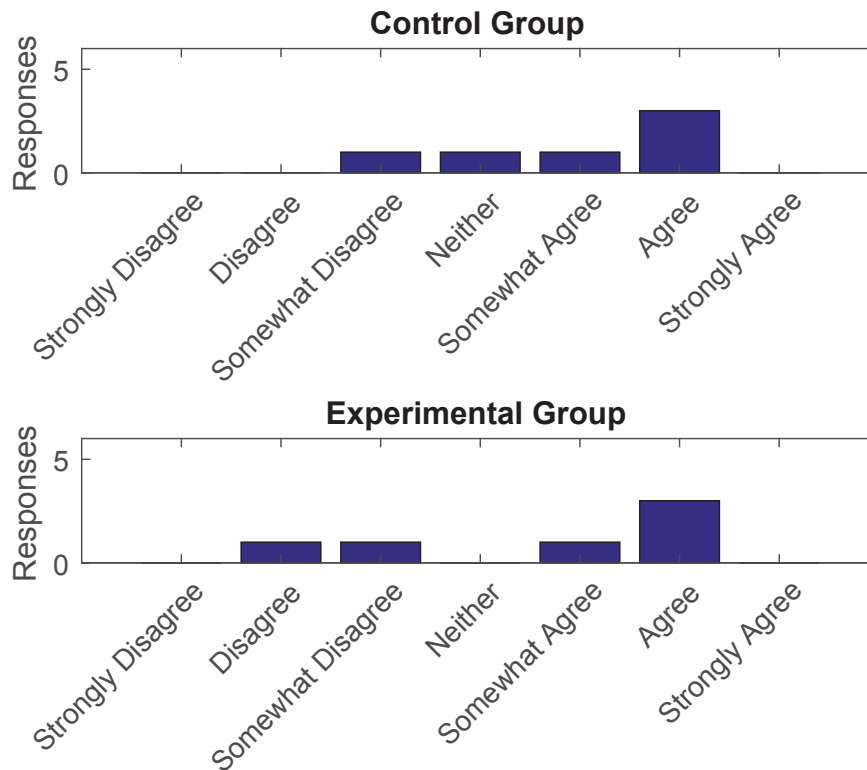


Figure 4.13: Student responses to the question “The instructor was aware of how the students were doing on the exercise” by group.

much of the difficulty in programming is in the details, like syntax errors. Learning the principles is important, but students learn programming by actually doing it... And with instant instructor feedback, the students can correct their small errors so that later they will have more success in implementing the principles in their assignments.”

A participant from the control group stated that it was “*helpful being able to try the code rather than staring at it.*” A participant from the experimental group said that “*This should become a thing. I would have loved learning this way. It would allow for much more interaction.*” All of these comments validate the hypothesis that the website will be helpful and that students will appreciate it.

Another response from the experimental group was “*The software was easy to use, if the teachers gave more helpful responses to my issues though, that would have been nice.*” The student acknowledged that the instructor responded to his/her issues, but would have liked better responses.

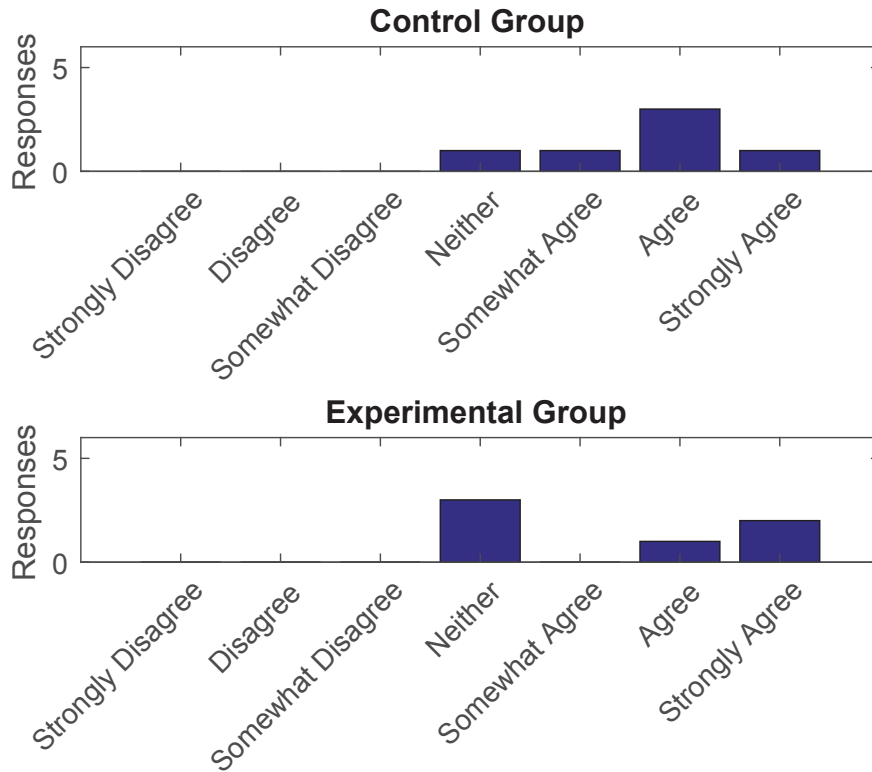


Figure 4.14: Student responses to the question “I enjoyed doing exercises on the website” by group.

Overall, the feedback from the students was positive. This supports Hypothesis 5, “The instructor and students will find the live compiler feedback useful.”

Instructor Results

The instructor agreed that website was easy to use. He slightly agreed that the website was enjoyable to use and agreed that the website made it easy to do in-class exercises.

The instructor then answered questions about the Compiler Data page of the website. The usefulness of each chart was rated from useful to not useful. The results are shown in Table 4.3. The Persistent Errors metric was by far the most useful graph. Current Output, Number of Compiles, and Run Success Percentage were all somewhat useful. Both Total Errors and Total Output were neutral. While the instructor did not find these results helpful as they did not help immediately address issues, collecting the total errors and outputs can be helpful in the long run addressing the most common issues throughout a course.

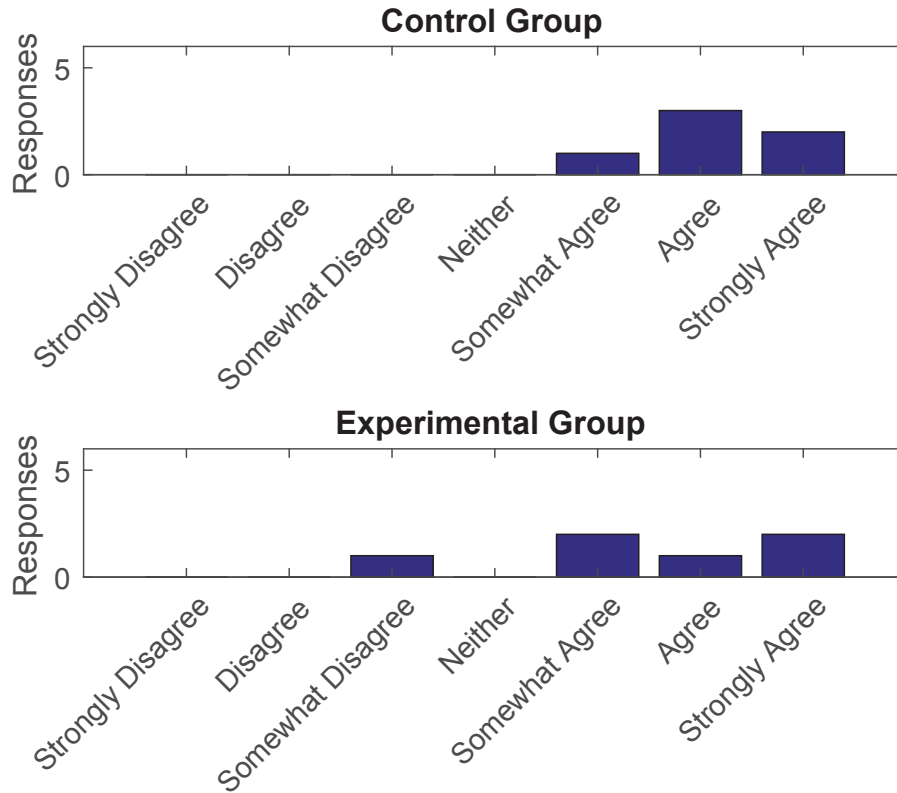


Figure 4.15: Student responses to the question “Doing in-class exercises helped me understand the material” by group.

The instructor then ranked the graphics, including the total and current errors tables. The results are shown in Table 4.4. Both tables scored above their corresponding bar graphs, likely due to the better readability in the tables. Readability will be improved in future implementations. Both the current errors and outputs were more helpful than the total errors and outputs. This is likely due to the live feedback nature of these experiments. Again, having the picture of the entire

Table 4.3: Instructor ratings for the usefulness of the website metrics

Metric	Usefulness
Persistent Errors	Useful
Current Output	Somewhat useful
Total Errors	Neutral
Total Output	Neutral
Number of Compiles	Somewhat useful
Run Success %	Somewhat useful

exercise may be more helpful for long term changes, while having a current snapshot can help the instructor address immediate problems within a lecture. The emphasis in this experiment was placed on immediate problems, likely influencing these results.

The number of compiles was not found to be particularly useful by the instructor. However, it was mentioned by professors at BYU when questioned about potential metrics for the website. As the instructor was a teaching assistant, he may not have been as aware of the impact of the number of compiles on student success as a professor may have been. Previous research has found that students who compile frequently as they code are more successful than those who write many lines of code before compiling [56].

The instructor was asked to rate the adequacy of the data for C++ topics, numerical methods topics, error finding exercises, and program writing exercises. The results are summarized in Table 4.5. The instructor found the data to be more adequate for C++ topics and program writing exercises. This was contrary to the hypothesis, which postulated that C++ topics and error finding exercises would be more useful with the website feedback.

The instructor somewhat agreed that the website helped him understand what students were struggling with. He also somewhat agreed that the website helped him assist the students with their exercises. While not a strong result, this does indicate that the website has potential. This result is also limited to only one person, and more data would be needed to verify it.

Lastly, the instructor commented on useful additional outputs and features. The instructor suggested having access to the code snippets that caused the errors appearing on the Compiler Data page, the ability to view student code, and a metric for how many students the persistent

Table 4.4: Instructor ranking of the different graphics used in the website

Rank	Graphic
1	Current Errors Table
2	Current Output Graph
3	Persistent Errors Graph
4	Run Success Percentage Graph
5	Total Errors Table
6	Total Errors Graph
7	Total Output Graph
8	Number of Compiles

errors came from. An additional feature he requested was the ability to share screens between users, either to show students his solution or to view their solutions.

4.3 TA Experiment Results

Seven former and current teaching assistants from ME 273 were asked to evaluate the website. They reviewed two exercises used in the Simulated Course Experiment and watched a video recording of the Compiler Data page taken during the exercises. They took notes on their observations and filled out a survey. The survey focused on their evaluation of the website, what students struggled with during the recorded exercises, and their perceptions of ME 273.

4.3.1 Points of Confusion Results

The percentage of points of confusion identified by the test subjects that matched those listed by the students are shown in Table 4.6. The skill level of the TA is somewhat correlated with the percentage of errors identified, with an r-squared value of .41 and a two-sided p-value of .11. More data points are needed to show any real significance.

Five of the seven TAs were able to identify 50% or more of the students' errors just by watching the videos and looking through the website data. The TAs had a higher success rate for the first exercise (an error-finding exercise dealing with C++ functions) with percentages ranging from 17% to 100%, with an average of 71%. With the second exercise, a program writing exercise dealing with least squares regression, the TAs identified 61.9% of errors on average, with a low of 0% and a high of 100%.

Table 4.5: Instructor ratings of adequacy of the data for different topics and exercises

Topic/Exercise	Adequacy
C++	Moderately adequate
Numerical Methods	Slightly adequate
Error Finding	Slightly adequate
Program Writing	Moderate adequate

Overall, the teaching assistants were successful in identifying the students’ areas of confusion. More skilled TAs may be even better at identifying errors. One area of confusion was excluded because it did not relate to programming, but to how an equation was presented.

4.3.2 Evaluation of Website Metrics and Graphics

The TAs were asked to evaluate the different metrics gathered for the website. The average response to whether or not the metric was useful for persistent errors, current outputs, total errors, compiles, and run success percentage was between “agree” and “somewhat agree”, leaning towards “somewhat agree”. Total output was rated at a “neither agree nor disagree”, with the lowest average.

The TAs then ranked the different visualizations from one to eight. The average, median, and mode responses were calculated. The average and median results yielded the same order, and were used as the final result, shown in Table 4.7. However, it is important to note that the standard deviation for most of the visualizations was close to two or three, which is rather large. Therefore, determining these orders through medians may not be very accurate.

Interestingly, these results differ from those in the Simulated Course Experiment shown in Table 4.4. This may be due in part to the lack of similarity between TA responses. Additionally, the instructor in the Simulated Course Experiment used the website for 16 days, while the TAs used the website for less than an hour. However, there are some similarities. Both experiments show that the total outputs graph is less useful. Current outputs and persistent errors are generally more useful. The run success percentage graph is ranked fourth in both experiments.

Table 4.6: The percentage of confusion areas identified by each teaching assistant by exercise

TA	Pct. Errors Identified	Pct. Errors Ex. 1	Pct. Errors Ex.2	Skill
1	17	33	0	3.659
2	67	33	100	6.257
3	67	100	33	5.244
4	50	100	0	3.945
5	67	33	100	7.27
6	100	100	100	6.257
7	100	100	100	5.53

On average, the TAs found the data to be between extremely and moderately adequate, leaning towards moderately adequate, for C++ topics. They found the data to be moderately adequate for numerical methods topics. They found the data to be between moderately and slightly adequate, leaning towards moderately adequate, for error finding exercises. They found the data to be moderately adequate for program-writing exercises. Overall, the data was adequate for most types of topics and exercises. It was slightly better for C++ topics. This makes sense, as syntax errors most often relate directly to C++ topics.

The TAs on average agreed that the website helped them determine what the students were struggling with. The TAs also on average agreed that the website would have helped them assist students. No response was less than a somewhat agree, emphasizing that the TAs believed the website would be effective. Interestingly, the TAs believed the website would be more slightly more useful than the instructor in the Simulated Course Experiment.

4.3.3 Ideas for Improvement

The TAs suggested labeling the axes on the graphs and the ability to look at the students' code when they get an error. Additionally, they wanted to know how many students were getting each error.

Table 4.7: Teaching assistant ranking of graphics by usefulness

Rank	Graphic
1	Persistent Errors Graph
2	Total Errors Graph
3	Current Outputs Graph
4	Run Success Percentage Graph
5	Number of Compiles
6	Total Outputs Table
7	Current Errors Table
8	Total Outputs Graph

Table 4.8: Teaching assistant ranking of difficult subjects in ME 273

Rank	Topic
1	Initial Value Problems
1	Boundary Value Problems
2	C++ Flow Control
2	C++ Classes
2	Linear Systems Solvers
3	C++ Functions
3	Numeric Integration
4	C++ Basics
5	C++ File Input/Output
5	Numeric Derivatives
5	Least Squares Regression
6	C++ Arrays
6	Numeric Root Finding

4.3.4 Teaching Assistant Insight on ME 273

On average, the TAs agreed that ME 273 is a difficult course. As the teaching assistants performed well in the course, this means that it's not only students who do poorly who believe the course is difficult. The TAs strongly agreed that students perceive ME 273 to be a difficult course. Their reasons mostly consisted of students lacking previous programming experience, the large amount of course material, and the fear students have of programming.

The TAs then indicated which subjects the students struggled with the most. Initial and boundary value problems received the most votes, with six out of seven TAs indicating that students struggled with these subjects. Linear systems solving, C++ classes, and C++ flow control received five votes. The rest of the course subjects are listed in Table 4.8. It is interesting that C++ flow control, C++ functions, and C++ basics are listed at two, three and four respectively. All of these subjects are discussed early on, and yet students struggle with them throughout the semester. As evidence by the most common errors in all of the exercises, simply declaring variables, placing semi-colons, and correctly defining if statements and loops cause the most problems throughout the semester. This suggests a misunderstanding of variable declarations and syntax rules.

Initial and boundary value problems were most often selected to be difficult for students. These topics are taught at the end of the semester and relate to differential equations, a course that

some students have not yet taken when they are in ME 273. This is likely why these topics are considered the most difficult.

Most teaching assistants recommended having students take an introductory programming course before ME 273 to lessen the learning curve. Other suggestions included having in-class exercises and using templates to lessen the burden on students.

4.4 BYU Survey Results

The Department of Mechanical Engineering at Brigham Young University conducted a survey in March 2017 to evaluate the current status of ME 273, Introduction to Computational Methods. 232 students participated in the study, including both undergraduate and graduate students. Questions focused on student performance, student perceptions of the course, and suggestions for improvement.

68.1% of survey participants either agreed or strongly agreed that ME 273 prepared them for the programming required in courses they took after ME 273. Despite this, several students complained that C++ was overemphasized compared to MATLAB, the primary language used in the following ME courses. The emphasis placed on C++ or MATLAB can vary between professors and semesters, which could be causing this disparity.

Only 33.74% of survey participants either agreed or strongly agreed that ME 273 prepared them for the programming required in jobs they had after ME 273. 43.58% were neutral, potentially indicating that these participants did not have a job where programming was required. 22.7% of participants either disagreed or strongly disagreed that they were prepared for the programming required in their jobs. If the students without a programming job are removed, 92 students remain for analysis. 40.2% of those students did not believe ME 273 prepared them for a job requiring programming, while 59.8% did. While more students agreed than disagreed, 40.2% is a significant number of students to not feel prepared for a job after a course. Since students felt prepared for the following courses but not for jobs, it is possible that the course did not go into enough depth for the coding required for a job.

28.16% of study participants had more than 2 months of programming experience before starting ME 273. Only 5.34% of participants had over a year of programming experience. However, 43.21% of students felt that more than 30% of their classmates had programming experience

before starting the course. About 30% of students underestimated how many of their classmates had experience, and about 30% of students correctly estimated how many of their classmates had some programming experience.

71.69% of study participants felt disadvantaged at the beginning of the course because others had more prior programming experience. At the end of the course, 41.35% of participants still felt disadvantaged because others had more prior programming experience. Based on the actual skill level of students before the course, this perception is largely incorrect and needs to be remedied. In 2016, Harvey Mudd found that decreasing the number of in-class questions by students with previous experience improved female students' experiences and increased the rate at which they signed up for more programming courses [83]. Experienced students were asked to save their questions for office hours to take up less lecture time. Based on comments in the survey results, several students at BYU not only felt disadvantaged but felt that the course was being taught to too high a level of experience, potentially because students asking questions had more experience and therefore gave the false perception that the students overall had more experience.

The survey also investigated potential improvements and changes to the course. 76.62% of students with previous programming experience agreed or strongly agreed that the programming portion of ME 273 was beneficial even though they had prior programming experience. Additionally, several students mentioned in comments that they appreciated the help and expertise of the more experienced students. Despite this, 61.04% of experienced students believe that students with significant programming experience should be allowed to test out of the programming portion.

76.68% of study participants agreed or strongly agreed that the course needs to be taught in a computer classroom so all students can immediately, in-class, implement and practice what is being taught. However, only 18.4% of participants always used a laptop in class to practice, and 25.77% sometimes used a laptop in class. Over 55% of participants either rarely or never used a laptop for practice. A participant mentioned in the comments that students without access to a laptop are disadvantaged when in-class examples are used interactively. They reinforced the idea of holding lecture in a computer classroom. A few other comments indicated that a lecture on programming syntax was not very effective without interactive examples. This has been discussed in previous literature [4, 8, 9, 18, 31].

The most difficult aspects of the ME 273 course were the cumulative load of the course, followed by numerical methods and C++. 64.81% of students highlighted the cumulative load as one of the most difficult aspects of ME 273. The next most common answers are the core subjects of the class: numerical methods and C++, with 32.1% and 29% respectively. While most students felt that the amount of material was the most difficult part, about a third of all participants felt that the two main topics in the course were the most difficult parts.

Interestingly, the most difficult parts of the course were also some of the more enjoyable parts of the course, with 49.69% of students selecting C++ as one of the most enjoyable aspects of the course and 29.56% of students selecting numerical methods. Matlab was by far the most enjoyed topic, with 60.38% of students selecting it. Labs were tied with numerical methods at 29.56%, showing the appreciation for more hands on applications. It is worth noting that in the free response section, seven students specifically pointed out projects as the most enjoyable aspect, something listed by five students as the most difficult aspect of the course. Alternatively, five students enjoyed nothing about the course.

Overall, the most common suggestions to improve the course were splitting the course into two blocks to separate programming and numerical methods and adding online resources, from 53.21% and 35.26% respectively.

4.5 Cost Benefit Analysis

The results show some of the potential benefits of using a live feedback system in a lecture environment. An instructor can immediately see what errors and outputs students are getting, and can potentially address widespread problems quickly, in lecture. Additionally, the simplified interface of the website text editor, compared to a full IDE, may lower the cognitive load required of students as they learn how to program. Students do not have to download any software to use the website as long as they have a web browser. Overall, the interface is simple and has many potential benefits.

There are always costs associated with implementing something new in a course. The first is the learning curve required. Both the students and the instructor will need to learn how to use the website. The website has been designed to be simple to use, but may still require some time to adjust to. Using the website will also require instructors to plan interactive examples ahead of

time, and to schedule enough time to get sufficient data from the students' code. There is a small amount of time required to interpret the data and form a response as well. Students will need to bring a laptop to class to use the website, or the course will need to be taught in a computer lab. The website is available in mobile format and will therefore work with smart phones, but programming on a smart phone is very difficult as typos are common.

Despite the costs, the potential benefits are significant. Implementing new software in a course is always difficult, but this website has been designed to be as simple as possible to use. The benefits include helping instructors better understand their students' difficulties, which in turn should help the students learn programming more easily. Students who are better at programming have another skill set to assist them in their future careers, making them more competent and competitive.

CHAPTER 5. CONCLUSION

Limitations, potential future work, and conclusions will be discussed in this chapter.

5.1 Limitations

There were two main categories of limitations with the experiments presented in Chapter 3. The first is dealing with human subjects in an experiment, specifically students in an educational setting. The second category is technologically based, and deals with the limits of the website implementation. Both categories will be discussed in the following sections.

5.1.1 Human Subjects

Human subjects by their nature add a certain amount of uncontrolled variables into any experiment. Gathering demographic data about the subjects is generally used to account for the impact of these uncontrolled variables. However, due to a conflict of interest where the instructor for the ME 273 course was part of the research team, demographic data were not collected to comply with the IRB. For the case study, this limited the analysis to be done for the group overall. The effects of gender, socioeconomic status, etc. on programming performance can not be determined from this study.

Another limitation of the case study was working with students in an educational setting. It is difficult to set up a controlled experiment in an already existing course with only one section. Most engineering education research working with existing courses will use one section of the course as a control and another section as the experimental group. This setup removes the ability to randomly assign subjects to a group, but allows for a control and experimental group. Students could not be forced to participate in the experiment, and therefore participants were selected on a volunteer basis, disallowing random assignment to each group. This greatly limits the inferences that can be made from any results.

A significant limitation is the sample size used for each experiment. Since students had to volunteer, the sample size for the case study was limited to 22 out of 90 enrolled students. The simulated course experiment similarly was limited to 12 students based on scheduling conflicts. A larger sample size is often necessary to show significance and to limit the effects of outliers. Only seven teaching assistants responded for the teaching assistant study, again limiting the significance of the results. Additionally, four of the seven teaching assistants had participated in the simulated course experiment, and potentially had an advantage in determining what students struggled with since they had performed the exercises previously. To compensate for this, each participant was given the same amount of time to look over and perform the exercises before evaluating student performance. Two months had also passed between the two experiments, so the information was not in the immediate short term memory of the participants.

Ideally, the simulated course experiment would have been done in an actual ME 273 course with novice students. However, completely controlling the ME 273 course would have been too difficult. For this reason, the simulated course experiment was developed. More experienced students were used because of the limited time frame available, and to ensure that some of the participants could complete the in-class exercises and provide data for the experiment. This may have provided different results than if actual novice ME 273 students had been used. Additionally, ideally an actual ME 273 instructor would have been used for the simulated course experiment. However, a teaching assistant was chosen instead, again due to scheduling conflicts. Some of the results may have been different if the instructor had more experience both with the subject matter and with teaching.

Lastly, a few of the simulated course experiments were affected by scheduling problems. Three students missed a lecture period during the experiment. The instructor missed a lecture period twice. This was remedied by either reading directly from the slides or recording a lecture beforehand.

5.1.2 Technology

The website itself had several limitations. Some of these limitations will be discussed in Section 5.2 as future improvements. The most serious limitations will be discussed here.

The first limitation is the lack of user input from the console, a concept frequently used in ME 273. Students were unable to use the *cin* command as the website only contained a text editor, not a console. While this is frequently used in ME 273, it was easy to change the in-class exercises to use default values instead of asking for user input.

System pause is a command commonly used in ME 273, but not in the programming world at large. Because the website runs student code on a Linux server and directly prints out any compiler messages and output, there is no need for *system pause*, which tells Windows to pause the execution of a program until a keystroke is received. This is used when students are using the console for their output, and need to pause the program to see the output before the program completes. Using *system pause* actually throws an error when used with the website.

The website did not originally have exception handling. When a student ran a program with a run-time exception, rather than a compiler error, the website would crash. This problem was originally remedied by having students program within a try-catch statement, and has since been resolved in the current version of the website. For the first two exercises in the simulated course experiment, students crashed the website a few times before the end of the experiment. The website server was restarted each time, allowing the experiment to continue with minimal interruptions.

Another limitation of the website was due to high levels of traffic. A load test was performed with 30 students, and the website was able to handle all 30 students compiling and running their code simultaneously. However, the website was not able to handle 90 students without slowing down. Some students were unable to load the website when there were too many users. This is likely due to a limitation of the server hosting the website, and could be resolved by moving the website to a more powerful server.

5.2 Future Work

Potential future work was derived from two sources, feedback from research participants and ideas from the researchers. Both are discussed, with the feasibility of each and an estimate of how many hours they would take to implement.

5.2.1 Feedback from Research Participants

Several changes have already been made to the website based on participant feedback from all of the experiments performed. The suggestions that have not yet been addressed have been discussed with the website developer, and the feasibility of each has been evaluated. Future research projects building off of this work were also discussed and evaluated with the website developer.

Small, aesthetic issues with the website have been fixed, such as a button that was the incorrect size in specific browsers, adding axis labels on the Compiler Data page, changing the graphs to be easier to read, and increasing the size of the text editor on the Assignments page. Additionally, exception handling has been added to increase the robustness of the website. The ability for the instructor to delete assignments has been added.

Study participants made many suggestions for additions or improvements to the website. The most common suggestions are listed in Table 5.1. Future work could involve implementing these changes. The most common suggestion was the ability to link errors to student code, either allowing the instructor to click on an error and see the student's entire program or providing the snippets of code causing the error. Many participants felt it would be much easier to determine what students were struggling with if they could see the actual code. All of the code and corresponding errors are saved each time a student compiles, so implementing this change would mostly involve changing website pages to show the student's code and making database queries. The time estimate to complete this change is between 10 and 15 hours by a skilled programmer.

The most difficult change to implement would be adding a debugger to the website. While technically feasible, this would require interacting with the process running the student's code. Currently, student code is copied into a folder on the server and a process is started programmatically to compile and run the code. After the run command, no interaction is had with the process until any messages are returned. This would need to be changed, and a method for how to interact with the process would need to be developed. Additionally, it would need to be decided which debugger features are desired, such as breakpoints, watch windows, etc. The time estimate to implement the changes is 25-30 hours by a skilled programmer. However, this estimate does not include the time necessary to determine what features to enable and assumes the features are possible to enable.

The simplest change is to allow students to see the Compiler Data page. One participant suggested this to boost students' self-esteems and allow them to see that other students are struggling too. While this would be relatively easy to fix, the question remains as to whether or not it is necessary. The instructor could simply show the Compiler Data page in class. Additionally, it may actually be discouraging for a student to see that they have more errors than anyone else. A potential solution would be to show a student if another student has the same error, and enable them to work together to solve the problem.

5.2.2 Ideas from the Researchers

As the website was developed and used, the researchers had several ideas for improving the website and potential future research projects. The researchers felt that the Compiler Data page could be improved significantly. Most of the teaching assistants and the simulated course instructor did not find the total errors and total outputs metric to be very useful. However, they might be useful to an instructor looking for trends over the entire exercise or a semester. To remedy this, the Compiler Data page could become customizable, allowing an instructor to select which metrics he/she wants to view. Customizing the Compiler Data page would likely take 5-10 hours, depending on how much customization was desired.

For the metrics that study participants did find useful, such as current output and persistent errors, it would be useful to know how many students produced each error. Errors that have several

Table 5.1: Participant suggestions, feasibility, and hours to implement by a skilled programmer

Suggestion	Feasibility	Hours
Link errors to student code	feasible	10-15
Allow students to post questions	feasible	10-15
Add a debugger	somewhat	25-30
Allow user input	feasible	15-20
Enable reading/writing to files	feasible	10-15
Add hints when errors occur	feasible	20-25
Allow students to see others' errors	feasible	1-5
Allow multiple files for a project	feasible	5-10
Add IDE features (autocomplete, etc.)	feasible	10-15

occurrences but are all coming from one student may be better addressed in an individual session, while widespread errors could be resolved more easily in a lecture setting. To fix this, a second bar for the number of students experiencing the error or output could be placed next to each bar in the bar graphs. Adding the number of students would take some refactoring in how the data are collected for the visualizations. It would likely take 5-10 hours to implement.

The output graphs were problematic because there was no control over the desired output or how many outputs from a compile were displayed. If a student output a number from a for loop 100 times, it would show up 100 in the output graph. Again, a bar indicating the number of students producing the output could partially remedy this problem. However, more control is likely needed to gain real information from the output graphs. For example, an instructor could indicate the correct output and could then see which students were producing the correct answer. Outputs could be filtered to one per student, or students could use a tag to indicate their final output or solution. A time estimate for how long this would take to implement depends on what strategy is chosen to handle student outputs.

Statistics for the course overall, or over a period of time rather than an exercise, have been explored by others [1]. Estey has even used certain metrics to predict whether or not students will pass the course within the first two weeks [23]. While these metrics are not the focus of this research, they could easily be implemented from the current collected data. The number of compiles and the changes between compiles for each student can be calculated from the database. Student compilation behavior has been shown to predict student outcomes in programming courses [56].

Four future longer term research projects were also suggested by the researchers. The first was modifying the website to work with Matlab. Matlab can be run from another program, making this feasible. Additionally, CodeMirror, the text editor used in the website, supports Matlab syntax. However, there are also several complicating factors. Matlab is not a compiled language, but a scripting language. It is therefore slower than C++, which could impact the website's performance. Most Matlab errors are run-time errors, since Matlab is not compiled. The current website is set up to receive messages from a C++ compiler; this would need to be changed to handle Matlab. Matlab is also expensive and requires a license, which would need to be available on the server hosting the website. Whether or not the website can handle multiple run attempts at once with only one

license would need to be investigated. Furthermore, Matlab has extensive plotting abilities. The website currently does not have the capability to determine if a student's plot is correct, or even to store that type of data. There are many questions that could be investigated while integrating Matlab into the website.

Another question discovered throughout the experiment is related to cybersecurity. In a normal IDE, users are prevented from going out of the bounds of an array. With the g++ compiler used in the website, users simply access garbage data, rather than being blocked. A hacker would be able to use this loophole to access data on the server. How to prevent users from accessing data through arrays is an interesting research question. One idea is to find anywhere an array is used and replace it with the C++ array class, which prevents arrays from going out of bounds. This could be done with a regex statement or something similar. There are likely many other methods that could be successful in improving the security of the website.

One student requested access to other students' errors. As discussed briefly in Section 5.2.2, the effectiveness of this is unknown. Some researchers have experimented with activity streams for programming homework assignments and found them to be useful [61]. However, having live feedback in class might be different than while working at home, alone. The effectiveness of students understanding how the class is doing as a whole should be investigated to further improve the usefulness of the website.

Lastly, a longer term study in an actual ME 273 course should be done using the website. Repeating the experiment over several semesters and years is the only way to truly evaluate its usefulness in a realistic learning environment. As the website is improved, it will be better suited for use in class and should be easy to integrate into a lecture.

The four research projects are just a few of the possibilities. Further studies could be done to evaluate the changes made to the website and determine what training is necessary for instructors and students. These could be combined with the suggested longer term study.

5.3 Conclusions

The hypotheses addressed by each experiment will be discussed in the following sections. Any discoveries made in the exploration of the data will also be addressed. Conclusions derived from the experiment as a whole will then be given. The five hypotheses are listed here for reference.

1. **H1:** Students will perform better on quizzes when the instructor has access to live compiler feedback
2. **H2:** The instructor will more frequently know what students struggled with on an assignment when they have access to live compiler feedback
3. **H3:** When error finding exercises are given, the live feedback will have a more significant effect on quiz scores for programming topics
4. **H4:** When program writing exercises are given, the live feedback will have a more significant effect on quiz scores for numerical methods topics
5. **H5:** The instructor and students will find the live compiler feedback useful

5.3.1 Case Study Conclusions

The case study primarily addressed Hypothesis 5, “The instructor and students will find the live compiler feedback useful”, focusing on the students. 54% of participating students found that the website made it easy to do in-class exercises. Those who did not think the website made it easy to do in-class exercises listed website bugs as their issue with the software. Many of these issues have since been resolved. Future work would be required to determine if these fixes would increase the number of students believing that the website makes it easier to do in-class exercises.

64% of students indicated that the instructor responded to the website feedback by changing his lecture. This also addresses Hypothesis 5, as it supports the idea that the instructor was able to use the website. 55% of students felt that the instructor was able to address confusion in class using the website feedback, indicating that a slight majority of students found the instructor’s changes to be helpful. While not as impressive as a larger percentage, at least half of the students felt they were aided by the instructor using the website. This supports Hypothesis 5, suggesting that the students found the live compiler feedback useful.

Analyzing student code and compilation behavior revealed that on average, students who compile more frequently are more successful at removing errors from their programs. This corroborates other researchers’ findings, such as those in [56]. The standard deviation of the number of compiles for unsuccessful students was much lower than the standard deviation for successful

students. Additionally, it was discovered that the number of students producing an error varies between errors. This information should be included in future iterations of the website. The most common errors over four in-class exercises in an ME 273 course closely matched the most common errors experienced in CS1 courses that have been analyzed [54]. This suggests that ME students struggle with many of the same concepts as computer science students, and methods used to aid computer science students might also help ME students.

5.3.2 Simulated Course Experiment Conclusions

The simulated course experiment addressed all five hypotheses. A summary of the conclusions is shown in Table 5.2. Hypothesis 1, “Students will perform better on quizzes when the instructor has access to live compiler feedback”, is somewhat supported by the results of the simulated course experiment. The average quiz score is higher for the experimental group. However, the difference is not statistically significant. The evidence is merely suggestive, and would need to be replicated with a larger sample size to improve results.

Hypothesis 2, “The instructor will more frequently know what students struggled with on an assignment when they have access to live compiler feedback”. Again, the average percentage of errors identified by the instructor was higher with the experimental group. However, the results were not statistically significant. The evidence is therefore only suggestive, and would need further experimentation to provide a conclusion.

Hypothesis 3, “When error finding exercises are given, the live feedback will have a more significant effect on quiz scores for programming topics”, was not supported by the results of the simulated course experiment. There was no impact for group when error finding exercises were paired with programming topics. Additionally, due to random assignment of experimental conditions, these two conditions only overlapped once. More experiments would be needed to determine the effect, if any, of these two conditions together.

Hypothesis 4, “When program writing exercises are given, the live feedback will have a more significant effect on quiz scores for numerical methods topics”, was not supported by the results of the simulated course experiment. There was no impact for group when program writing exercises were paired with numerical methods topics. Similar to Hypothesis 3, due to

random assignment of experimental conditions, these two conditions only overlapped once. More experiments would be needed to determine the effect of these two conditions together.

Hypothesis 5, “The instructor and students will find the live compiler feedback useful”, was supported by the results of the simulated course experiment. The students, on average, agreed that the website made it easy to do in-class exercises. They somewhat agreed that the instructor addressed issues during the exercise, and somewhat agreed that the instructor was aware of how the students were doing on the exercise. The instructor somewhat agreed that the website helped him understand what the students were struggling with and that the website helped him assist the students with their exercises. Both the students and the instructor found the live compiler feedback useful.

5.3.3 Teaching Assistant Study Conclusions

The teaching assistant study focused on answering Hypothesis 5, the instructor finding the website feedback useful. On average, the TAs agreed that the website helped them determine what the students were struggling with. The TAs also on average agreed that the website would have helped them assist students. No response was less than a somewhat agree, emphasizing that the TAs believed the website would be effective. This supports the hypothesis that the instructors would find the website feedback useful.

Additionally, the study sought to determine whether or not the instructors could determine what the students were struggling with. This does not entirely address Hypothesis 2, as a control was not performed with TAs identifying errors without the website feedback. However, it is still useful to note that on average, the TAs were able to identify 67% of student areas of confusion

Table 5.2: Conclusions for hypotheses based on the simulated course experiment

Hypothesis	Conclusion
1	Suggestive but inconclusive
2	Suggestive but inconclusive
3	Inconclusive
4	Inconclusive
5	Supported

overall, just by watching a video of the Compiler Data page recorded while students programmed two exercises. This cannot entirely support Hypothesis 2, but it does provide evidence that the website feedback allows the instructors to determine the majority of what students are struggling with. With training, enhancements, and more skilled instructors, this percentage could increase, as skill was slightly correlated with the percentage of errors identified.

5.3.4 ME 273 Survey Conclusions

The ME 273 survey was not aimed at supporting a specific hypothesis, but sought to better understand the current status of ME 273 and potential improvements. One of the problem areas identified in the survey was the lectures. Several students commented that they struggled in lecture and requested more hands on learning. 76.68% of study participants agreed or strongly agreed that the course needs to be taught in a computer classroom so all students can immediately, in-class, implement and practice what is being taught. However, only 18.4% of participants always used a laptop in class to practice, and over 55% of participants either rarely or never used a laptop for practice. The idea presented in this research of using live compiler feedback for in-class exercises lines up well with what the students are suggesting in the survey.

71.69% of study participants felt disadvantaged at the beginning of the course because others had more prior programming experience. At the end of the course, 41.35% of participants still felt disadvantaged because others had more prior programming experience. The live compiler feedback has the potential to show students that they are not the only ones getting errors, hopefully dispelling the myth that many other students are more skilled than they are (only 5% of students have significant programming experience before the course). Overall, the survey results showed a need for a development like the live compiler feedback website.

5.3.5 Overall Conclusions

The hypotheses are summarized in Table 5.3, with the supporting experiments listed. Hypotheses 1 and 2 have some evidence supporting them, but are not conclusively supported. Hypothesis 2 has some additional support from the TA study. Hypotheses 3 and 4 have no evidence supporting them. Hypothesis 5 is supported by three different experiments, the case study, the

simulated course experiment, and the TA study, and has the strongest support. Overall, instructors and students found the website feedback to be useful.

The following proposed research objectives were accomplished.

1. **RO1:** Develop a live compiler feedback system suited for use during a lecture
2. **RO2:** Evaluate the live compiler feedback system
3. **RO3:** Recommend future improvements for the live compiler feedback system

RO1 was completed by developing a website that allowed students to program and collected real time data from each compile. The data were presented to the instructor using the same website. The website is discussed thoroughly in Chapter 3. RO2 was accomplished by running three different experiments, a case study in the ME 273 course, a simulated course experiment, and a TA study. The case study evaluated whether or not the website was easy to use, enjoyable to use, and whether or not the instructor changed their lecture material effectively based on the website feedback. The simulated course experiment evaluated student performance when the instructor had access to the website feedback as opposed to having no feedback. Additionally, the simulated course experiment asked for the instructor’s input on the usefulness of the data. The TA study evaluated the instructor’s performance using the website by recording how many areas of confusion the instructor could identify by looking at the website data.

RO3 was completed by asking all test subjects to provide input on improvements for the live compiler feedback system. These results were coded and the most common suggestions were analyzed by the researcher and the website developer for feasibility and time to implement. More suggestions were gathered from the researcher’s observations throughout the experiments and the website developer’s input. The suggestions are listed in Section 5.2.

Table 5.3: Conclusions for hypotheses with supporting experiments listed

Hypothesis	Conclusion	Experiment
1	Suggestive but inconclusive	Simulated Course
2	Suggestive but inconclusive	Simulated Course, TA Study
3	Inconclusive	Simulated Course
4	Inconclusive	Simulated Course
5	Supported	Case Study, Simulated Course, TA Study

The following research questions were answered.

1. **RQ1:** How can live feedback be implemented to improve programming lectures?
2. **RQ2:** How can live feedback be adapted for a mechanical engineering programming course?

To answer RQ1, live feedback can be implemented to improve programming lectures by using a web-based compiler to collect data on in-class exercises. This creates a more interactive environment and allows the instructor to receive feedback on the students' progress and areas of confusion. Overall, the website was found to be effective in helping instructors understand student confusion. Students generally appreciated the chance to learn with in-class exercises and instructors found the data to be useful.

To answer RQ2, live feedback may not need to be adapted much for a mechanical engineering programming course. Overall, the most common errors produced by ME 273 students match those in other studies based on introductory computer science courses [54]. Based on user feedback, the data were slightly less useful for numerical methods topics. Additionally, the Current Output graph could be modified to show student output better by including the student who produced the output and/or customizing which outputs are shown. These improvements could adapt the live feedback to better suit a mechanical engineering course that involves numerical methods as well as programming topics.

While not every hypothesis has significant supporting evidence, this work demonstrated the potential of using live compiler feedback on in-class exercises in an introductory mechanical engineering programming course. Methods that have been developed for computer science courses can be applied to mechanical engineering programming courses. Future work could improve the website and make it even more useful, and longer term experiments could better evaluate its usefulness. This tool has the potential to improve an instructor's ability and efficiency in addressing student confusion during lecture, freeing up more time for teaching or helping individual students.

REFERENCES

- [1] Luke, J. A., 2015. “Continuously collecting software development event data as students program.” PhD thesis, Virginia Tech. viii, 4, 15, 16, 31, 36, 95
- [2] Wang, Y., White, W. M., and Andersen, E., 2017. “Pathviewer: Visualizing pathways through student data.” In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ACM, pp. 960–964. viii, 18, 19
- [3] Garner, S., Haden, P., and Robins, A., 2005. “My program is correct but it doesn’t run: a preliminary investigation of novice programmers’ problems.” In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pp. 173–180. viii, 4, 7, 18, 19, 20
- [4] Chang, A. N., 2012. “A mobile instructor interface for collaborative software development education.” PhD thesis, Massachusetts Institute of Technology. viii, 2, 4, 26, 27, 28, 29, 31, 36, 45, 87
- [5] No author About CS4All - CS4All NYC Department of Education. 1
- [6] Smith, M., 2016. Computer science for all, Jan. 1
- [7] Cheng, H. H., 2011. C as Part of a Mechanical Engineering Curriculum. 1, 29
- [8] Jenkins, T., 2002. “On the difficulty of learning to program.” In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, Vol. 4, pp. 53–58. 1, 87
- [9] Beaubouef, T., and Mason, J., 2005. “Why the high attrition rate for computer science students: some thoughts and observations.” *ACM SIGCSE Bulletin*, **37**(2), pp. 103–106. 1, 87
- [10] Watson, C., and Li, F. W., 2014. “Failure rates in introductory programming revisited.” In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE ’14, ACM, pp. 39–44. 2, 14
- [11] Beilock, S. L., Gunderson, E. A., Ramirez, G., and Levine, S. C., 2010. “Female teachers’ math anxiety affects girls’ math achievement.” *Proceedings of the National Academy of Sciences of the United States of America*, **107**(5), Feb, pp. 1860–3. 2
- [12] Ashcraft, M. H., 2002. “Math anxiety: Personal, educational, and cognitive consequences.” *Current Directions in Psychological Science*, **11**(5), oct, pp. 181–185. 2

- [13] Jackson, C. D., and Leffingwell, R. J., 1999. “The role of instructors in creating math anxiety in students from kindergarten through college.” *The Mathematics Teacher*, **92**(7), pp. 583–586. 2
- [14] Hodge, B., and Steele, W. G., 2002. “A survey of computational paradigms in undergraduate mechanical engineering education.” *Journal of Engineering Education*, **91**(4), pp. 415–417. 3, 29
- [15] Brake, M. L., 2009. “Matlab as a tool to increase the math self-confidence and the math ability of first-year engineering technology students.” *The Scholarship of Teaching and Learning at EMU*, **1**(1), p. 5. 3, 29
- [16] Coller, B. D., and Scott, M. J., 2009. “Effectiveness of using a video game to teach a course in mechanical engineering.” *Computers & Education*, **53**(3), pp. 900–912. 3, 30
- [17] Kerr, B., 2015. “The flipped classroom in engineering education: A survey of the research.” In *Interactive Collaborative Learning (ICL), 2015 International Conference on*, IEEE, pp. 815–818. 3, 11
- [18] Shannon, A., and Summet, V., 2015. “Live coding in introductory computer science courses.” *Journal of Computing Sciences in Colleges*, **31**(2), pp. 158–164. 3, 12, 13, 87
- [19] Salleh, N., Mendes, E., and Grundy, J., 2011. “Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review.” *IEEE Transactions on Software Engineering*, **37**(4), pp. 509–525. 3, 14
- [20] Robins, A., Haden, P., and Garner, S., 2006. “Problem distributions in a cs1 course.” In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52, ACE '06*, Australian Computer Society, Inc., pp. 165–173. 4, 19
- [21] Hanks, B., 2008. “Problems encountered by novice pair programmers.” *Journal on Educational Resources in Computing (JERIC)*, **7**(4), p. 2. 4, 14, 19
- [22] Altadmri, A., and Brown, N. C., 2015. “37 million compilations: Investigating novice programming mistakes in large-scale student data.” In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ACM, pp. 522–527. 4, 20
- [23] Estey, A., 2016. “An exploration of learning tool log data in cs1: how to better understand student behaviour and learning.” PhD thesis, University of Victoria. 4, 31, 76, 95
- [24] Glassman, E. L., 2016. “Clustering and visualizing solution variation in massive programming classes.” PhD thesis, Massachusetts Institute of Technology. 4, 17, 45
- [25] Hundhausen, C., Olivares, D., and Carter, A., 2017. “Ide-based learning analytics for computing education: a process model, critical review, and research agenda.” *ACM Transactions on Computing Education (TOCE)*, **17**(3), p. 11. 4, 15, 23
- [26] Olivares, D. M., and Hundhausen, C. D., 2016. “Osble+: A next-generation learning management and analytics environment for computing education.” In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ACM, pp. 5–5. 4, 22, 31

- [27] Du, S.-C., Fu, Z.-T., and Wang, Y., 2014. “The flipped classroom: Advantages and challenges.” In *International Conference on Economic, Management and Trade Cooperation, April*, pp. 12–13. 7, 9, 10
- [28] Matsuzawa, Y., Hirao, M., and Sakai, S., 2016. “Compile error collection viewer: Visualization of compile error correction history for self-assessment in programming education.” *INTERNATIONAL JOURNAL OF ENGINEERING EDUCATION*, **32**(3), pp. 1117–1127. 8, 23
- [29] Maher, M. L., Latulipe, C., Lipford, H., and Rorrer, A., 2015. “Flipped classroom strategies for cs education.” In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ACM, pp. 218–223. 11
- [30] Harding, T., 2014. “Experiences of using a collaborative programming editor in a first-year programming course.”. 11, 12, 31, 42
- [31] Rubin, M. J., 2013. “The effectiveness of live-coding to teach introductory programming.” In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, ACM, pp. 651–656. 12, 13, 31, 87
- [32] Salleh, N., Mendes, E., Grundy, J., and Burch, G. S. J., 2010. “An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model.” In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, ACM, pp. 577–586. 14
- [33] Bevan, J., Werner, L., and McDowell, C., 2002. “Guidelines for the use of pair programming in a freshman programming class.” In *Software Engineering Education and Training, 2002.(CSEE&T 2002). Proceedings. 15th Conference on*, IEEE, pp. 100–107. 14
- [34] Zacharis, N. Z., 2011. “Measuring the effects of virtual pair programming in an introductory programming java course.” *IEEE Transactions on Education*, **54**(1), pp. 168–170. 14
- [35] Chaparro, E. A., Yuksel, A., Romero, P., and Bryant, S., 2005. “Factors affecting the perceived effectiveness of pair programming in higher education.” In *Proc. PPIG*, Citeseer, pp. 5–18. 14
- [36] McDowell, C., Werner, L., Bullock, H., and Fernald, J., 2002. “The effects of pair-programming on performance in an introductory programming course.” *ACM SIGCSE Bulletin*, **34**(1), pp. 38–42. 14
- [37] Baheti, P., Williams, L., Gehringer, E., and Stotts, D., 2002. “Exploring pair programming in distributed object-oriented team projects.” In *Educator’s Workshop, OOPSLA*, pp. 4–8. 14
- [38] Preston, D., 2006. “Using collaborative learning research to enhance pair programming pedagogy.” *ACM SIGITE Newsletter*, **3**(1), pp. 16–21. 14
- [39] Tsompanoudi, D., Satratzemi, M., and Xinogalos, S., 2016. “Evaluating the effects of scripted distributed pair programming on student performance and participation.” *IEEE Transactions on Education*, **59**(1), pp. 24–31. 14

- [40] Estler, H. C., Nordio, M., Furia, C. A., and Meyer, B., 2013. “Collaborative debugging.” In *Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on*, IEEE, pp. 110–119. 14
- [41] Berenson, S. B., Slaten, K. M., Williams, L., and Ho, C.-W., 2004. “Voices of women in a software engineering course: reflections on collaboration.” *Journal on Educational Resources in Computing (JERIC)*, **4**(1), p. 3. 14
- [42] Goldman, M., Little, G., and Miller, R. C., 2011. “Collabode: collaborative coding in the browser.” In *Proceedings of the 4th international workshop on Cooperative and human aspects of software engineering*, pp. 65–68. 14
- [43] Nieminen, A., Lautamäki, J., Kilamo, T., Palviainen, J., Koskinen, J., and Mikkonen, T., 2013. “Collaborative coding environment on the web: A user study.” *Developing Cloud Software Algorithms, Applications, and Tools*,(60), pp. 275–300. 14
- [44] Natsu, H., Favela, J., Morán, A. L., Decouchant, D., and Martinez-Enriquez, A. M., 2003. “Distributed pair programming on the web.” In *Computer Science, 2003. ENC 2003. Proceedings of the Fourth Mexican International Conference on*, IEEE, pp. 81–88. 14
- [45] Boyer, K. E., Dwight, A. A., Fondren, R. T., Vouk, M. A., and Lester, J. C., 2008. “A development environment for distributed synchronous collaborative programming.” In *ACM SIGCSE Bulletin*, Vol. 40, ACM, pp. 158–162. 14
- [46] Ho, C.-W., Raha, S., Gehringer, E., and Williams, L., 2004. “Sangam: a distributed pair programming plug-in for eclipse.” In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, ACM, pp. 73–77. 14
- [47] Kumar, G. D., and Sam, R. P. “Different approaches to construct collaborative environment and real time collaborative editing.”. 14
- [48] Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S. H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K., et al., 2015. “Educational data mining and learning analytics in programming: Literature review and case studies.” In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ACM, pp. 41–63. 15
- [49] Madhavan, K., and Richey, M. C., 2016. “Problems in big data analytics in learning.” *Journal of Engineering Education*, **105**(1), pp. 6–14. 15
- [50] Glassman, E. L., Fischer, L., Scott, J., and Miller, R. C., 2015. “Foobaz: Variable name feedback for student code at scale.” In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM, pp. 609–617. 18
- [51] Toll, D., Ohlsson, T., Ericsson, M., and Wingkvist, A., 2015. “Fine-grained recording of student programming sessions to improve teaching and time estimations.” In *IFIP TC3 Working Conference” A New Culture of Learning: Computing and Next Generations” July 1-3, 2015, Vilnius, Lithuania*, pp. 264–274. 21
- [52] Lin, Y.-T., Wu, C.-C., Hou, T.-Y., Lin, Y.-C., Yang, F.-Y., and Chang, C.-H., 2015. “Tracking students cognitive processes during program debugging an eye-movement approach.”. 21

- [53] Estey, A., and Coady, Y., 2016. “Can interaction patterns with supplemental study tools predict outcomes in cs1?.” In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, pp. 236–241. 21, 22
- [54] Jadud, M. C., 2006. “Methods and tools for exploring novice compilation behaviour.” In *Proceedings of the second international workshop on Computing education research*, ACM, pp. 73–84. 22, 64, 98, 102
- [55] Watson, C., Li, F. W., and Godwin, J. L., 2013. “Predicting performance in an introductory programming course by logging and analyzing student programming behavior.” In *Advanced learning Technologies (ICALT), 2013 IEEE 13th international conference on*, IEEE, pp. 319–323. 22
- [56] Carter, A. S., Hundhausen, C. D., and Adesope, O., 2015. “The normalized programming state model: Predicting student performance in computing courses based on programming behavior.” In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, ACM, pp. 141–150. 22, 38, 81, 95, 97
- [57] Becker, B. A., 2016. “A new metric to quantify repeated compiler errors for novice programmers.” In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, pp. 296–301. 22, 45
- [58] Beck, L. L., Chizhik, A. W., and McElroy, A. C., 2005. “Cooperative learning techniques in cs1: design and experimental evaluation.” In *ACM SIGCSE Bulletin*, Vol. 37, ACM, pp. 470–474. 22
- [59] Olivares, D., 2015. “Exploring learning analytics for computing education.” In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, ACM, pp. 271–272. 22
- [60] Olivares, D. M., and Hundhausen, C. D., 2017. “Supporting learning analytics in computing education.” In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, ACM, pp. 584–585. 22
- [61] Hundhausen, C. D., and Carter, A. S., 2014. “Facebook me about your code: an empirical study of the use of activity streams in early computing courses.” *Journal of Computing Sciences in Colleges*, **30**(1), pp. 151–160. 22, 96
- [62] Hundhausen, C. D., and Carter, A. S., 2014. “Supporting social interactions and awareness in educational programming environments.” In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, ACM, pp. 55–56. 22
- [63] Hundhausen, C. D., Carter, A. S., and Adesope, O., 2015. “Supporting programming assignments with activity streams: an empirical study.” In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ACM, pp. 320–325. 22
- [64] Carter, A. S., and Hundhausen, C. D., 2016. “With a little help from my friends: An empirical study of the interplay of students’ social activities, programming activities, and course success.” In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ACM, pp. 201–209. 22

- [65] Keuning, H., Jeurig, J., and Heeren, B., 2016. “Towards a systematic review of automated feedback generation for programming exercises.” In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, pp. 41–46. 24, 76
- [66] Le, N.-T., 2016. “Analysis techniques for feedback-based educational systems for programming.” In *Advanced Computational Methods for Knowledge Engineering*. Springer, pp. 141–152. 24
- [67] Parihar, S., Dadachanji, Z., Singh, P. K., Das, R., Karkare, A., and Bhattacharya, A., 2017. “Automatic grading and feedback using program repair for introductory programming courses.” In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, pp. 92–97. 24, 25
- [68] Rawat, K. S., Riddick, G. B., and Moore, L. J., 2008. “Work in progress-integrating mobile tablet-pc technology and classroom management software in undergraduate electronic engineering technology courses.” In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, IEEE, pp. S4D–23. 25
- [69] Pistilli, M. D., and Arnold, K. E., 2010. “In practice: Purdue signals: Mining real-time academic data to enhance student success.” *About Campus*, **15**(3), pp. 22–24. 25
- [70] Chen, J. C., Whittinghill, D. C., and Kadlowec, J. A., 2006. “Using rapid feedback to enhance student learning and satisfaction.” In *Frontiers in Education Conference, 36th Annual*, IEEE, pp. 13–18. 26
- [71] Lantz, M. E., and Stawiski, A., 2014. “Effectiveness of clickers: Effect of feedback and the timing of questions on learning.” *Computers in Human Behavior*, **31**, pp. 280–286. 26
- [72] No author, 2018. MATLAB Cody Coursework: Grade MATLAB programming assignments automatically. 26
- [73] Char, B. W., Daulagala, I., Kandasamy, N., and Hewett, T. T., 2016. “Using automatic matlab program testing for a first-year engineering computation course.” In *2016 ASEE Annual Conference & Exposition*. 26
- [74] Aden-Buie, G., Kaw, A., and Yalcin, A., 2015. “Comparison of final examination formats in a numerical methods course.” *INTERNATIONAL JOURNAL OF ENGINEERING EDUCATION*, **31**(1), pp. 72–82. 29
- [75] No author Code Mirror. 33, 42
- [76] No author, 2018. GUID Structure. 43
- [77] No author, 2018. Approximate String Comparison in C#. 43, 45
- [78] Combéfis, S., and Schils, A., 2016. “Automatic programming error class identification with code plagiarism-based clustering.” In *Proceedings of the 2nd International Code Hunt Workshop on Educational Software Engineering*, ACM, pp. 1–6. 45
- [79] No author, 2017. Plotly. 45

- [80] No author, 2018. Just Flip a Coin. 51
- [81] Feigenspan, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S., 2012. “Measuring programming experience.” In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, IEEE, pp. 73–82. 54, 55
- [82] No author, 2018. oCam Free Download - Easy & Powerful Screen Recorder. 55
- [83] Staley, O., 2016. Harvey Mudd College took on gender bias and now more than half its computer-science majors are women. 87

APPENDIX A. EXPERIMENTAL PROCEDURES

A.1 Simulated Course Experimental Procedures

A.1.1 Proctor Instructions

1. Set up OCAM
 - (a) Make sure videos are being saved to the right folder
 - (b) Use name structure: group#_lastName_topic#
 - (c) Check which screen is being captured on OCAM
 - (d) Start OCAM at the beginning of the experiment
2. Check that website is running
3. OCAM website on my computer
4. Record Derek talking
5. Introduce rules to students at the beginning of the first experiment
6. Write down any questions the students ask

A.1.2 Instructor Experimental Procedure

Rules:

- You can only use the View Data page for Group 2 (Wednesday Friday). You may address issues the students are having if you infer them from the View Data page.
- You can answer questions during the lecture

- You cannot answer questions during the exercise
- Do not change slides during the exercise
- You can answer questions after the exercise, but only if they ask questions (Group 1). Do not prompt them for questions. With Group 2, you can address issues based on the data collected by the website.
- Do not allow students to take notes

Schedule:

- If W/F, sign in to website at me273vm.et.byu.edu
- Lecture - less than 10 minutes
- Exercise - give students 5 minutes unless they all finish early
- Quiz - students will have 3 minutes to take a quiz that has been emailed to them
- Survey - students will have 2 minutes to take a survey that has been emailed to them. You will also take this survey.

A.2 Teaching Assistant Study Experimental Procedures

A.2.1 Proctor Instructions

1. Set up account for subject at me273.vm.et.byu.edu
2. Provide exercise code for subject.
3. Set timer for 10 minutes. Instruct subject to look through the exercises, attempt to do the exercises, and think about problems as if they were going to use the exercise as an example in class. Subjects may ask questions during this time.
4. Open website Compile Data page for Exercise 1. Open video for Exercise 1. Provide TA Experiment Website Instructions and answer any questions. Instruct subject to take notes on what students are struggling with.

5. Open website Compile Data page for Exercise 2. Open video for Exercise 2. Answer any questions. Instruct subject to take notes on what students are struggling with.
6. Have subject take Qualtrics survey.

A.2.2 Website Explanation

Students were asked to complete 5-10 minute programming exercises using this website. The website collected compiler and output data as the students programmed. This data is displayed visually on the Compiler Data page.

The Compiler Data page consists of six graphs and two tables.

1. Persistent Errors

- (a) Persistent errors shows only current errors divided by the total number of occurrences of that error. For example, if an error occurred five times over the course of the experiment, and currently one person still has that error, the persistence would be 0.2. This is intended to help you identify how difficult an error is to fix.

2. Current Outputs

- (a) Current outputs shows the top five most common current outputs. Output is defined as console output, so whatever was written to the screen using “*cout*”. Current means the most recently compiled version for each student. This is intended to help you identify if students are getting the correct solution.

3. Total Errors

- (a) Total errors shows the total number of errors at any given time.

4. Total Outputs

- (a) Total outputs shows the total outputs at any given time.

5. Times Compiled

- (a) The number of times each student has tried to compile or run their code. This is intended to show you how often students are attempting to solve the problem.

6. Run Success Percentage

- (a) The number of successful runs over the number of total attempted runs. A success percentage of 1 means that the students code ran every time without errors. A success percentage of 0 means none of the runs were successful. This is intended to show you the progress students are making.

7. Current Errors Table

- This table allows the instructor to see all of the current errors more clearly.

8. Total Errors Table

- (a) This table allows the instructor to see all of the errors more clearly.

APPENDIX B. SURVEY QUESTIONS

B.1 ME 273 Winter Semester Survey

B.1.1 Consent Document

My name is Ariana Sellers, I am a graduate student at Brigham Young University and I am conducting this research under the supervision of Professor Salmon, from the Department of Mechanical Engineering . You are being invited to participate in this research study of an In-Class Exercise Software Tool. I am interested in finding out about your perceptions of the utility and usefulness of the software.

Your participation in this study will require the completion of the attached survey. This should take approximately 2 minutes of your time. Your participation will be anonymous and you will not be contacted again in the future. You will not be paid for being in this study. This survey involves minimal risk to you. The benefits, however, may impact society by helping increase knowledge about teaching programming to engineering students.

You do not have to be in this study if you do not want to be. You do not have to answer any question that you do not want to answer for any reason. We will be happy to answer any questions you have about this study. If you have further questions about this project or if you have a research-related problem you may contact me, Ariana Sellers, at pedersenariana@gmail.com or my advisor, Dr. Salmon, at johnsalmon@byu.edu.

If you have any questions about your rights as a research participant you may contact the IRB Administrator at A-285 ASB, Brigham Young University, Provo, UT 84602; irb@byu.edu; (801) 422-1461. The IRB is a group of people who review research studies to protect the rights and welfare of research participants.

The completion of this survey implies your consent to participate. Your consent gives me permission to publish the results of in-class exercises and follow-up quizzes anonymously and in

aggregate (only averages for the entire class will appear), as well as the anonymous results of this survey. If you choose to participate, please click the button below and complete the survey by June 21st, 2017. Thank you!

Please provide your name for consent purposes. This allows us to only include data for the participants who consent to participate. NOTE: The researchers, Ariana Sellers and Dr. Salmon, will not see your name in the survey results. A research assistant has been employed to anonymize the survey data to prevent a conflict of interest. If you wish to remove your consent at any time, you may email landon.wright91@gmail.com and request that your information be removed.

B.1.2 Student Survey

1. Did you use the experimental software?
2. Rate your agreement with the following statements:
 - (a) The experimental software was easy to use.
 - (b) The experimental software was enjoyable to use.
 - (c) The experimental software made it easy to do in-class exercises.
 - (d) When the experimental software was used, the instructor changed his/her lecture to address the results of the in-class exercise.
 - (e) When the experimental software was used, the instructor was able to address student confusion during lecture.
3. Please include any other comments about the experimental software.

B.2 Simulated Course Experiment Surveys

B.2.1 Programming Skill Survey

1. What is your name?
2. For how many years have you been programming?
3. On a scale from 1 to 10, how do you estimate your programming experience?

4. How do you estimate your programming experience compared to your classmates?
5. How experienced are you with C++?
6. How experienced are you with MATLAB?
7. How experienced are you with Python?
8. If you are very experienced with any other languages, please list them here.
9. How experienced are you with object-oriented programming?

B.2.2 Student Survey

1. Were you in group 1 or group 2?
2. Rate your level of agreement with the following statements:
 - (a) The experimental software was easy to use.
 - (b) The experimental software was enjoyable to use.
 - (c) The experimental software made it easy to do in-class exercises.
3. Rate your level of agreement with the following statements:
 - (a) The persistent errors graph was useful.
 - (b) The current output graph was useful.
 - (c) The total errors graph was useful.
 - (d) The total output graph was useful.
 - (e) The number of compiles graph was useful.
 - (f) The run success percentage graph was useful.
4. Rate your level of agreement with the following statements:
 - (a) The instructor addressed issues the students were having.
 - (b) The instructor knew how the class was doing on the exercises.

- (c) I enjoyed doing exercises on the website.
 - (d) Doing exercises after the lecture helped me learn the material.
5. What improvements or additional features would have made the experimental software more useful?
6. Provide any other comments on the experiment or the software here.

B.2.3 Instructor Survey

1. Rate your level of agreement with the following statements:
- (a) The experimental software was easy to use.
 - (b) The experimental software was enjoyable to use.
 - (c) The experimental software made it easy to do in-class exercises.
2. Rate your level of agreement with the following statements:
- (a) The persistent errors graph was useful.
 - (b) The current output graph was useful.
 - (c) The total errors graph was useful.
 - (d) The total output graph was useful.
 - (e) The number of compiles graph was useful.
 - (f) The run success percentage graph was useful.
3. Rank the types of data visualizations in order of usefulness.
- (a) Persistent errors
 - (b) Current output
 - (c) Total errors
 - (d) Total output
 - (e) Number of compiles

- (f) Run success percentage
 - (g) Current errors table
 - (h) Total errors table
4. Answer the following with extremely adequate to extremely inadequate:
 - (a) How useful was the data for C++ topics?
 - (b) How useful was the data for numerical methods topics?
 - (c) How useful was the data for error finding exercises?
 - (d) How useful was the data for non-error finding exercises?
 5. What additional outputs would have been useful?
 6. What additional features would have been useful?
 7. Rate your agreement with the following statements:
 - (a) The experimental software helped me understand what the students were struggling with.
 - (b) The experimental software helped me assist the students with their exercises.
 8. Provide any other comments on the experiment or the software here.

B.3 Teaching Assistant Study Survey

1. For how many years have you been programming?
2. On a scale from 1 to 10, how do you estimate your programming experience?
3. How do you estimate your programming experience compared to your classmates?
4. How experienced are you with C++?
5. How experienced are you with MATLAB?
6. How experienced are you with Python?

7. If you are very experienced with any other languages, please list them here.
8. How experienced are you with object-oriented programming?
9. What do you think the students struggled with during the in-class exercise you watched?
10. Rate your agreement with the following statements:
 - (a) The persistent errors graph was useful.
 - (b) The current output graph was useful.
 - (c) The total errors graph was useful.
 - (d) The total output graph was useful.
 - (e) The number of compiles graph was useful.
 - (f) The run success percentage graph was useful.
11. Rank the types of data visualizations in order of usefulness.
 - (a) Persistent errors
 - (b) Current output
 - (c) Total errors
 - (d) Total output
 - (e) Number of compiles
 - (f) Run success percentage
 - (g) Current errors table
 - (h) Total errors table
12. Answer the following with extremely adequate to extremely inadequate:
 - (a) How useful was the data for C++ topics?
 - (b) How useful was the data for numerical methods topics?
 - (c) How useful was the data for error finding exercises?

- (d) How useful was the data for non-error finding exercises?
13. What additional outputs would have been useful?
14. What additional features would have been useful?
15. Rate your agreement with the following statements:
- (a) The experimental software helped me understand what the students were struggling with.
 - (b) The experimental software helped me assist the students with their exercises.
16. What improvements or additional features would have made the experimental software more useful?
17. Provide any other comments on the experiment or the software here.
18. Rate your agreement with the following statements:
- (a) ME 273 is a difficult course.
 - (b) The students perceive ME 273 as a difficult course.
19. Why do students find ME 273 to be difficult?
20. What topics or concepts did students find difficult? Select all that apply.
- (a) C++ Basics
 - (b) C++ Flow Control
 - (c) C++ File Input/Output
 - (d) C++ Functions
 - (e) C++ Static Arrays
 - (f) C++ Classes
 - (g) Numerical Derivatives
 - (h) Numerical Integration

- (i) Root Finding Methods
- (j) Least Squares Regression
- (k) Initial Value Problems
- (l) Boundary Value Problems
- (m) Linear Systems Solvers

21. What should be done to improve ME 273?

B.4 BYU ME 273 Survey

This survey and the data were provided by Dr. Charles.

1. Are you a graduate student or an undergraduate student?
2. How much programming experience did you have before enrolling in ME 273?
 - (a) None
 - (b) Hours
 - (c) Days
 - (d) Weeks
 - (e) Months
 - (f) Years
3. How did you learn to program?
 - (a) Self-taught
 - (b) High school course
 - (c) High school club
 - (d) University level course
 - (e) Off-campus job
 - (f) On-campus job

- (g) Other
4. What languages did you learn on your own?
 - (a) Java/Javascript
 - (b) Python
 - (c) Arduino
 - (d) Other
 5. In your opinion, how many of your classmates in ME 273 had programming experience before enrolling in ME 273?
 6. Rate your agreement: The programming portion of ME 273 was beneficial even though I had prior programming experience
 7. Rate your agreement: This course needs to be taught in a computer classroom so all students can immediately (during class) implement and practice what is being taught
 8. Rate your agreement: ME 273 prepared me for the programming required in courses I have taken after ME 273
 9. Rate your agreement: ME 273 has prepared me for the programming required in jobs that I have had after ME 273
 10. Rate your agreement: Students with significant programming experience should be allowed to test out of the programming portion.
 11. How should C++ and MATLAB be taught?
 - (a) Interweaved
 - (b) C++ first, then MATLAB
 - (c) MATLAB first, then C++
 - (d) Other
 12. Did you use a laptop in class to practice what was being taught?

- (a) Always
- (b) Sometimes
- (c) Rarely
- (d) Never

13. What were the most difficult aspects of ME 273?

- (a) C++
- (b) MATLAB
- (c) Numerical methods
- (d) Homework
- (e) Labs
- (f) Cumulative load
- (g) Other

14. What were the most enjoyable aspects of ME 273?

- (a) C++
- (b) MATLAB
- (c) Numerical methods
- (d) Homework
- (e) Labs
- (f) Other

15. How could the experience in ME 273 be improved?

- (a) Reduced load
- (b) Split into two blocks
- (c) Projects over labs
- (d) Online resources

(e) Other

16. Rate your agreement: Any comments or suggestions

17. Rate your agreement: When the course began, I felt at a disadvantage because others had more prior programming experience

18. Rate your agreement: In hindsight, I really was at a disadvantage in this course because others had more prior programming experience

19. What should be done to level the playing field?

(a) Nothing

(b) Extra sessions for those without prior programming experience

(c) Require extra sessions for those without prior programming experience

(d) Require all students to take a CS class

(e) Provide more TA help for homework and labs

(f) Provide more online resources

(g) Allow those with significant prior programming experience to test out of the programming portion

(h) Other

APPENDIX C. SIMULATED COURSE MATERIALS

C.1 Lecture Slides

C.1.1 Topic 1: C++ Basics

Topic 1: C++ Basics

Figure C.1

Simple Example

- Declaring variables
- Math operations

```
1 // Simple Example: Adding and Subtracting Two Integers
2 // Author: Scott Stiles
3 // Description: This program adds two integers, outputs the
4 //              sum, and then outputs the difference.
5 // Requirements:
6 //   None
7 //   Input: Two integers
8 //   Output: Sum, Difference
9 //
10 // Compile and Run: g++ simple.cpp -std=c++11 -c -o simple.o
11 //                  g++ simple.o -std=c++11 -o simple
12 #include <ostream>
13 using namespace std;
14
15 int main()
16 {
17     // Variable Declarations
18     int num1, num2, sum, diff;
19
20     // Prompt User to Enter Two Integers
21     cout << "Enter two integers" << endl;
22     cin >> num1 >> num2;
23
24     // Compute the Sum and Difference
25     sum = num1 + num2;
26     diff = num1 - num2;
27
28     // Output the Results
29     cout << "The sum is: " << sum << endl;
30     cout << "The difference is: " << diff << endl;
31
32     return 0;
33 }
```

Figure C.2

Program Structure

```
1 // Program Structure: A Template for a C++ Program
2 // Author: Scott Stiles
3 // Description: This program illustrates the structure of a C++ program.
4 //              It shows the placement of preprocessor directives,
5 //              variable declarations, statements, and the return value.
6 // Requirements:
7 //   None
8 //   Input: None
9 //   Output: None
10 //
11 // Compile and Run: g++ structure.cpp -std=c++11 -c -o structure.o
12 //                  g++ structure.o -std=c++11 -o structure
13 #include <ostream>
14 using namespace std;
15
16 // Preprocessor Directives
17
18 int main()
19 {
20     // Variable Declarations
21
22     // Statements
23
24     // Return Value
25 }
```

Figure C.3

Comments

- Comments are ignored by the compiler
- There are two forms:
 - `//comment`
 - `/* comment */`

Figure C.4

Preprocessor Directives

- Give instructions to the compiler that are performed before the program is compiled
- Statements are not executed at runtime
- Start with #
- No semicolon needed
- Ex:
 - `#include <iostream>`
 - `#include "myClass.h"`

Figure C.5

Preprocessor Directives: Namespace

- Namespaces help keep identifiers unique in large programs
- Tells the compiler to use library file names declared in a specific namespace
- We will use the following:
 - `using namespace std;`

Figure C.6

```

// Sum and Difference of Two Numbers
// Author: Pooja Mishra
// Description: This program adds two integers, subtracts the
//             second from the first, and prints the results.
// Compiler: g++
// Date: 20/09/2021
// File Name: sum_diff.cpp
// Version: 1.0

#include <iostream>
using namespace std;

int main()
{
    //Declare variables
    int num1, num2, sum, diff;

    //Prompt the user to enter two integers
    cout << "Enter two integers" << endl;
    cin >> num1 >> num2;

    //Perform the operations
    sum = num1 + num2;
    diff = num1 - num2;

    //Display the results
    cout << "The sum is: " << sum << endl;
    cout << "The difference is: " << diff << endl;

    return 0;
}

```

Figure C.7

Main Function

- Each C++ program has only one main function
 - This is where the program execution starts
 - The “int” specifies that an integer will be returned
- Functions are contained within curly brackets – { }
- The main function contains declarations and statements

• Ex:

```
int main()
{
    return 0;
}
```

Figure C.8

Declaration Statements

- All identifiers need to be defined before they can be used
- Three purposes:
 - Give names to variables
 - Define memory locations
 - Initialize values
- Variable declarations require a type (int, double, etc.)

• Ex:

- float x;
- float x = 5.0;
- float x(5.0);

Figure C.9

C++ Data Types

- int
- double
- bool
- char
- string (must #include <string> to use)

Figure C.10

Simple Input/Output

- Use cin to read from the keyboard
 - cin >> x;
- Use cout to print to the console
 - cout << "Hello!" << endl;

Figure C.11

Variable Naming Rules

- Start with a letter or underscore
- Can consist of letters, digits, and underscores
- Cannot be a reserved word (new, return, continue, etc.)
- Case sensitive

Figure C.12

Exercise 1

- Write a C++ program that:
 - Asks the user to input 4 temperatures in degrees Celsius
 - Converts each temperature from Celsius to Fahrenheit
 - $F = (9/5) * C + 32$
 - Prints the converted temperatures to the console
- Test your program using these numbers : 0, 100, 15.4, -40

Figure C.13

Topic 2: C++ Flow Control

Figure C.14

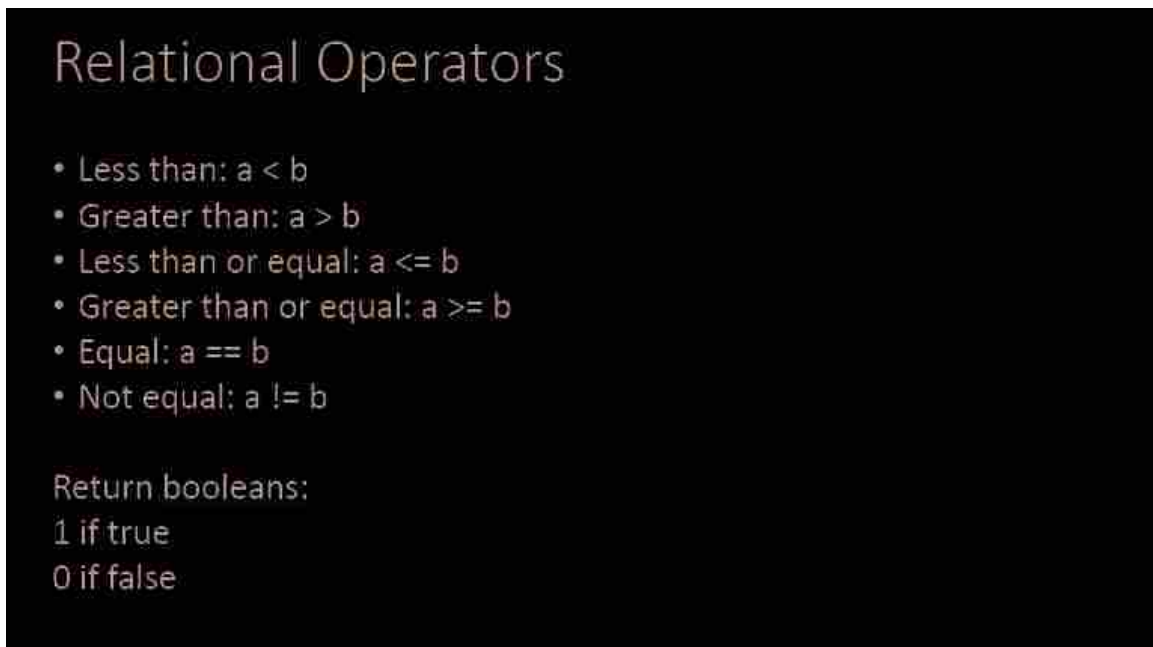


Figure C.15

What is the output?

- `int a(1), b(2);`
- `cout << (a == b) << endl;`

Figure C.16

Logical Operators

- `!` (NOT)
- `&&` (AND)
 - if all are true, then true
- `||` (OR)
 - if any are true, then true

Figure C.17

Selection Statements

```
if (expression)
{
    statements;
}
```

For example:

```
if (a == b)
    cout << "a = b" << endl;
```

Figure C.18

If-else statements

```
if (expression)
{
    statements;
}
else if (expression)
{
    statements;
}
else
{
    statements;
}
```

Figure C.19

Switch Statements

```
int score = 3;
switch (score)
{
    case 3:
        grade = 'A';
        break;
    default:
        grade = 'F';
        break;
}
```

Figure C.20

While Loop

```
while (expression)
{
    statements;
}
```

A while loop's expression is evaluated before any statements are executed. A while loop is generally used when you don't know how many iterations are needed.

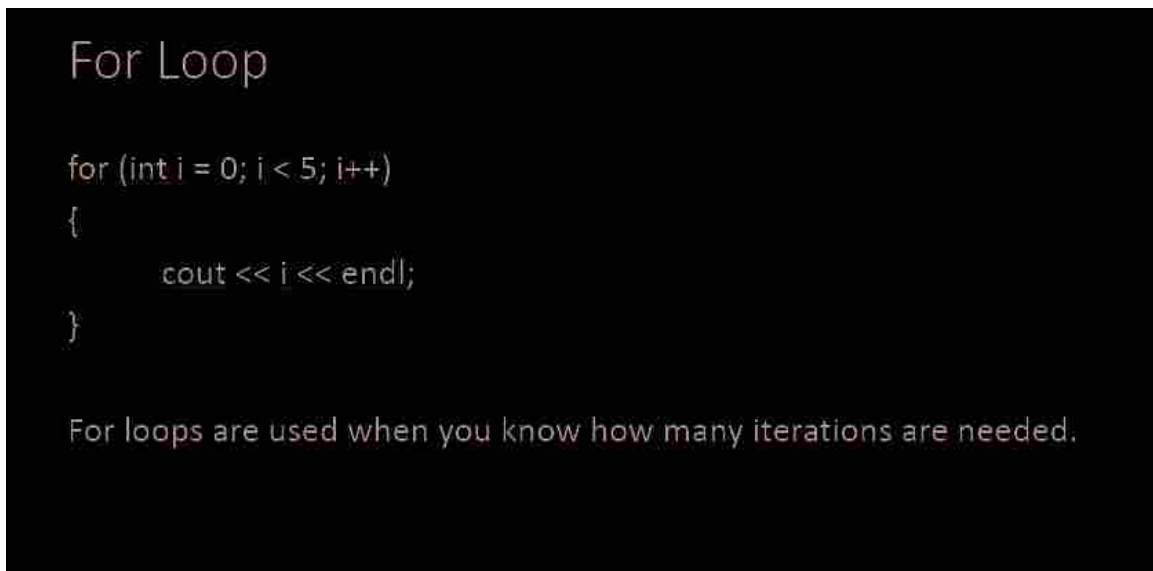
Figure C.21

Do-While Loop

```
do  
{  
    statements;  
} while (expression);
```

Statements are executed at least once before the expression is evaluated. Otherwise, this is the same as a while loop.

Figure C.22



```
For Loop  
  
for (int i = 0; i < 5; i++)  
{  
    cout << i << endl;  
}
```

For loops are used when you know how many iterations are needed.

Figure C.23

Exercise 2

- Write a program checking properties of two numbers
 1. Equality
 - If they're equal, print out a message saying "The numbers are equal"
 - If not, print out a message saying, "The numbers are not equal"
 2. Multiple conditions
 - If both numbers are less than 100 and the sum is less than 100, or the product is less than 0, print out "Success"
 - If not, print out "Failure"
 3. Use loops
 - Put the program in a do while loop and have it repeat 2 times.

Test the program with these numbers: 59, 40

Figure C.24

C.1.3 Topic 3: C++ Functions

C++ Functions

Figure C.25

Functions

- Functions can be used to separate chunks of code
- Functions are especially useful for a task that needs to be repeated
- Functions are also useful for organizing code and making it more readable

Figure C.26

Function Declaration

```
int add(int x, int y)
{
    return x + y;
}
```

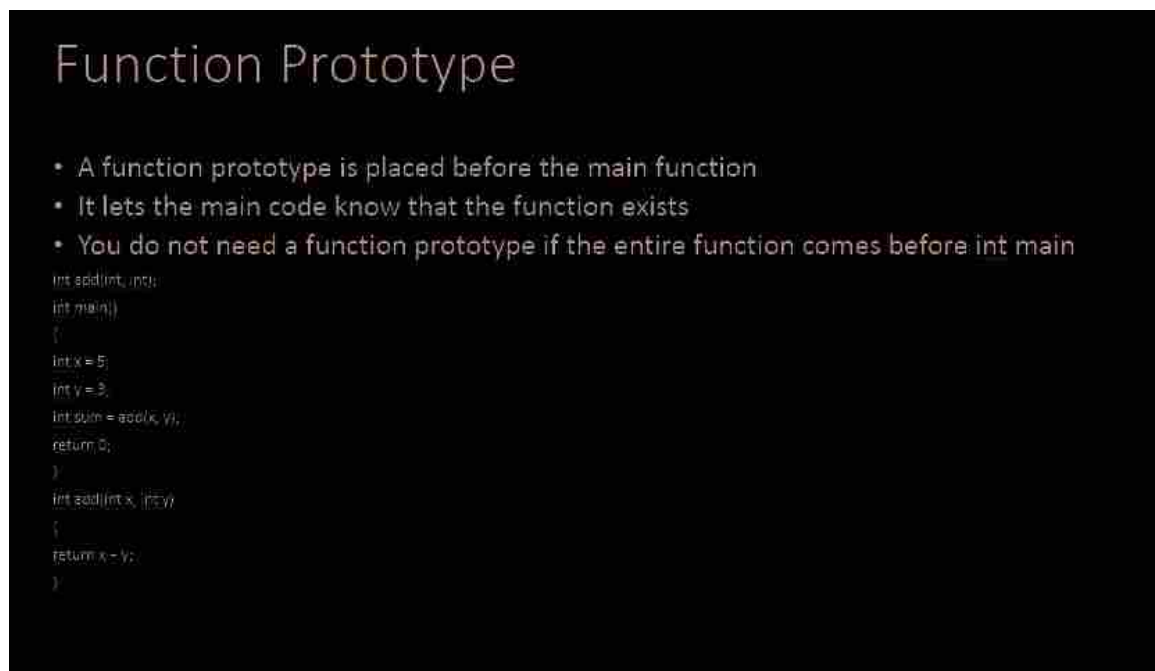
- return type
- function name
- input parameters
 - The names of the input parameters do not have to match anything in the main function

Figure C.27

Scope

- In general, a variable exists only within the brackets where it was declared.
- If you declare a variable in a function, it only exists in that function. If you try to use it in the main function, it will not work. Similarly, you cannot declare a variable in the main function and use it in the function.

Figure C.28



Function Prototype

- A function prototype is placed before the main function
- It lets the main code know that the function exists
- You do not need a function prototype if the entire function comes before int main

```
int add(int, int);
int main()
{
    int x = 5;
    int y = 3;
    int sum = add(x, y);
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

Figure C.29

Return Types

- You can return any variable type: int, double, float, string, etc.
- You can only return 1 variable
- You can also return nothing by using the keyword void:

```
void print(int x)
{
    cout << x << endl;
    return;
}
```

Figure C.30

Pass By Value vs. Reference

- An input parameter is passed by value unless otherwise specified
 - This creates a copy of the variable and it no longer has any link to the original variable.
- Passing by reference passes the address of the variable rather than the value. When you change a variable passed by reference, the value of the variable will change in the main function as well.
 - `int add(int& x, int& y);`
 - `add(5, 4);`
- This is sometimes used when you need multiple outputs from a function

Figure C.31

Exercise

- Navigate to [www.khanacademy.org](#) and log in
- Open the assignment “Topic 3 Group #” with the correct group number
- Fix the errors in the code already in the assignment

Figure C.32

C.1.4 Topic 4: Root Finding Methods

Root Finding Methods

Figure C.33

Purpose

- To find the root of an equation numerically
- To find a maximum or minimum numerically
- To find an intersection of two equations numerically

Figure C.34

Bracketing Methods

- Bisection
 - Given valid bounds, bisection always converges
 - Bisection is one of the slowest methods in terms of iterations
- False Position

Figure C.35

Bisection Method

- Start with known bounds
- Check if the middle value has the same sign as the lower bound
- If it does, make the middle value the new lower bound. If not, make the middle value the new upper bound
- Repeat until solution is within a chosen tolerance

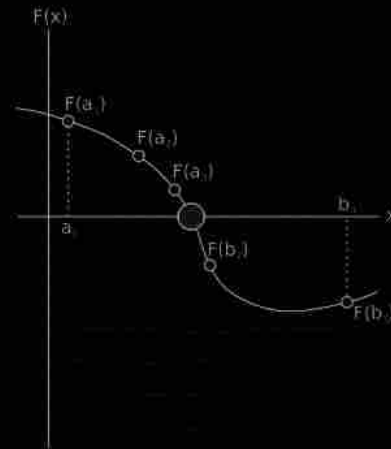


Figure C.36

Iterative Methods

- Newton-Rhapson
 - much faster than bisection
 - doesn't always converge – issues with different slopes, hitting a point with zero slope, etc.
 - requires the derivative of the function you're finding the root of
- Secant
 - much faster than bisection
 - doesn't always converge
 - does not require the derivative of the function
 - requires two initial guesses

Figure C.37

Newton-Raphson

- Start with an initial guess
- Iterate using the function below until the value of the function at x_n is very close to zero, within a chosen tolerance

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Divergence of Newton-Raphson

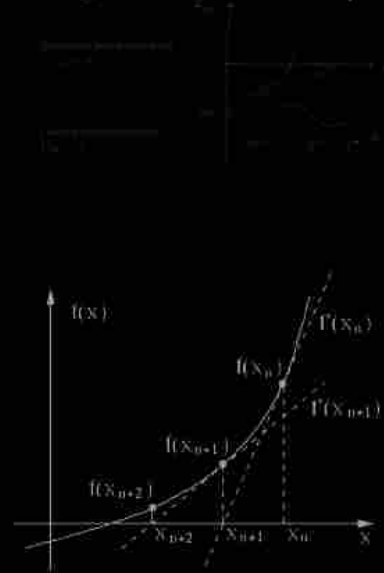


Figure C.38

Exercise

- Navigate to me273vm.et.byu.edu
- Go to the exercise for Topic 4 for your group
- Fix the errors in the provided code. There are syntax errors and algorithmic errors/typos. Fix both of these.
- The solution should be 1.7027 when everything has been fixed.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Figure C.39

C++ Static Arrays

Figure C.40

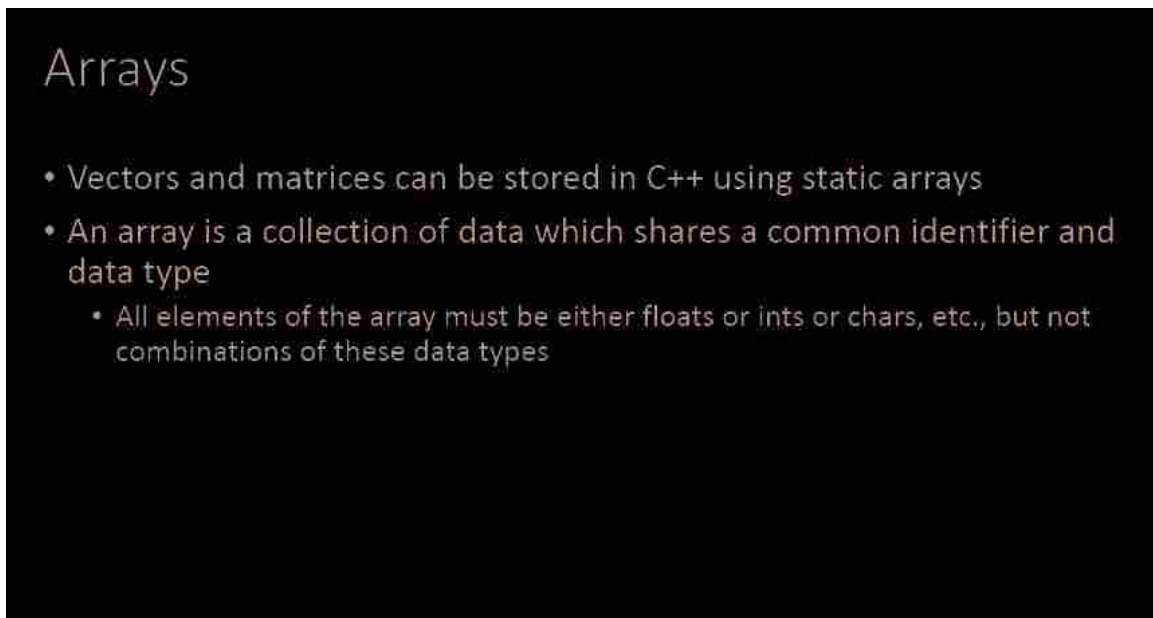


Figure C.41

Array Elements

- Individual elements of the array are specified using subscripts or indices
- In C++, indices start with 0
- Indices are specified in square brackets: []
- The index can be thought of as a number that tells the compiler which element of the array we are referring to
 - `My_array[0]` is the 1st element of `My_array`
 - `A[13]` is the 14th element of `A`
- Behind the scenes, the subscript actually describes how many memory locations the element is away from the base memory location

Figure C.42

One Dimensional Arrays

- A one-dimensional array can be visualized as a list of values arranged in either a row or a column (in C++ it doesn't matter which way you picture it)

2	<code>v[0]</code>
4	<code>v[1]</code>
1	<code>v[2]</code>
6	<code>v[3]</code>
10	<code>v[4]</code>

0.5	0.0	0.1	0.2	0.15	0.3
<code>s[0]</code>	<code>s[1]</code>	<code>s[2]</code>	<code>s[3]</code>	<code>s[4]</code>	<code>s[5]</code>

Figure C.43

Declaring Arrays

- `type array_identifier[number_of_elements]`
- The number of elements must be specified when the array is declared
- Arrays can be initialized when declared
 - `char vowels[5] = {'a', 'e', 'l', 'o', 'u'};`
 - `float gpa[] = {3.79, 2.84, 3.25, 3.97, 3.16, 2.93};`
 - `char word[] = "Hello";`

vowels:	a	e	l	o	u	
gpa:	3.79	2.84	3.25	3.97	3.16	2.93
word:	H	e	l	l	o	

Figure C.44

Accessing Array Elements

- Example - What's store in memory?

```
double m[6];  
for (int i=0; i<6; i++)  
    m[i] = i + 0.5;
```

m: 0.5 | 1.5 | 2.5 | 3.5 | 4.5 | 5.5

- Example - What's the output?

```
double gpa[] = {1.27, 3.25, 3.98, 2.75, 3.67};  
cout << gpa[2];  
3.98
```

- Subscripts may be any integer expression
(i.e. like `m[i]` above)

Figure C.45

Exercise

- Write a program that does the following:
 - Initializes an array with the following data: {3.5, 2.13, 6.78, 4.5, 9}
 - Finds the maximum value in the array and prints it out
 - Finds the minimum value in the array and prints it out
- You must write your own algorithm, you cannot use a max or min function created by someone else

Figure C.46

C.1.6 Topic 6: Numerical Integration

Numerical Integration

Figure C.47

Trapezoidal Rule

We want to find the integral of $f(x)$ of the area under the blue curve.

$$I = \int_a^b f(x) dx$$

We can approximate the function $f(x)$ with a linear function $f_1(x)$

$$f_1(x) = \frac{f(b) - f(a)}{b - a} (x - a) + f(a)$$

The approximate integral is then the integral of the approximate function

$$I \approx \int_a^b f_1(x) dx = \frac{b-a}{2} (f(a) + f(b))$$

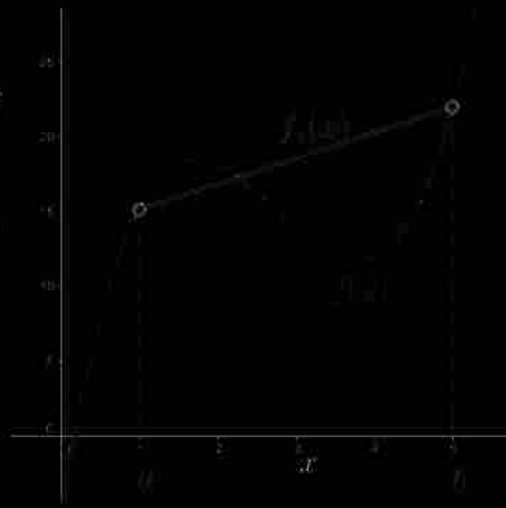


Figure C.48

Trapezoidal Rule

Therefore the derivation results in

$$I \cong (b - a) \frac{f(a) + f(b)}{2}$$

Results:

$$I_{\text{exact}} = 85.384$$

$$I_1 = 111.55$$

$$E = 30.6\%$$

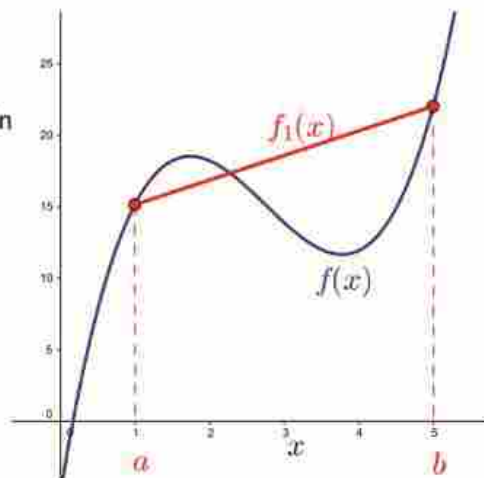


Figure C.49

Trapezoidal Rule

We can get a better approximation by breaking up the interval (a,b) into two strips of equal width h

$$I \approx \frac{h}{2}(f(a) + f(b))$$

Now, the approximate integral is just the sum of the two strips:

$$I \approx \frac{h}{2}(f(a) + f(x_1)) + \frac{h}{2}(f(x_1) + f(b))$$

Or,

$$I \approx \frac{h}{2}(f(a) + 2f(x_1) + f(b))$$

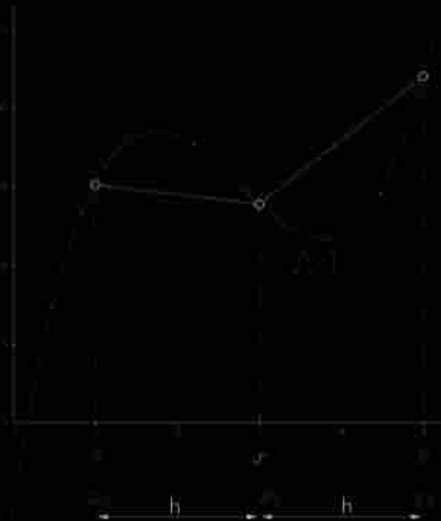


Figure C.50

Trapezoidal Rule

The general formula for $n+1$ data points, equally spaced between (a,b) is

$$I \approx \frac{h}{2}(f(a) + 2f(x_1) + 2f(x_2) + \dots + 2f(x_{n-1}) + f(b))$$

where,

$$h = \frac{b-a}{n}$$

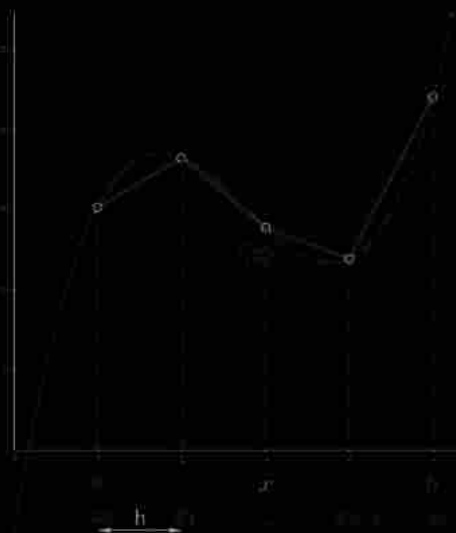


Figure C.51

Trapezoidal Rule

- Shortcomings:
 - must be equally spaced points (constant h)
 - error reduces rather slowly

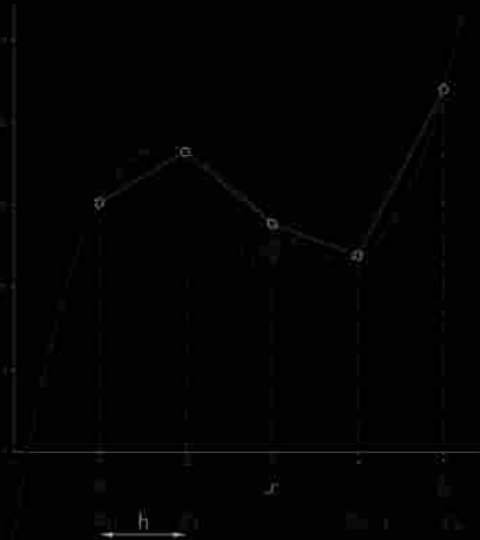


Figure C.52

Simpson's 1/3 Rule

We want to find the integral of $f(x)$, or the area under the blue curve.

$$I = \int_a^b f(x) dx = \int_a^b f(x) dx$$

We can approximate the function $f(x)$ with a second-order interpolating polynomial $f_2(x)$

$$f_2(x) = \frac{f(x_0)(x_1 - x)(x - x_2) + f(x_1)(x - x_0)(x - x_2) + f(x_2)(x - x_0)(x - x_1)}{(x_1 - x_0)(x_1 - x_2) + (x_2 - x_0)(x_2 - x_1) + (x_0 - x_1)(x_0 - x_2)}$$

The approximate integral is then the integral of the approximate function

$$I \approx \int_a^b f_2(x) dx = \int_a^b f_2(x) dx$$

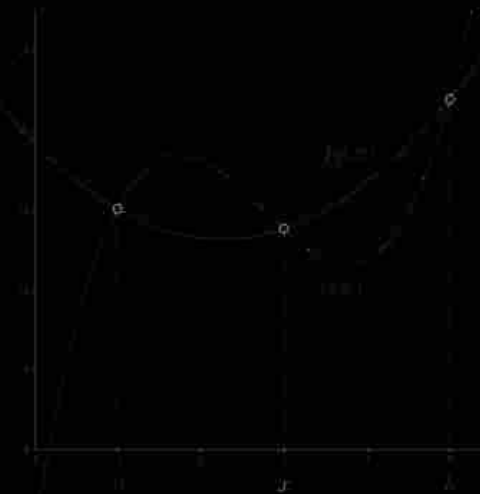


Figure C.53

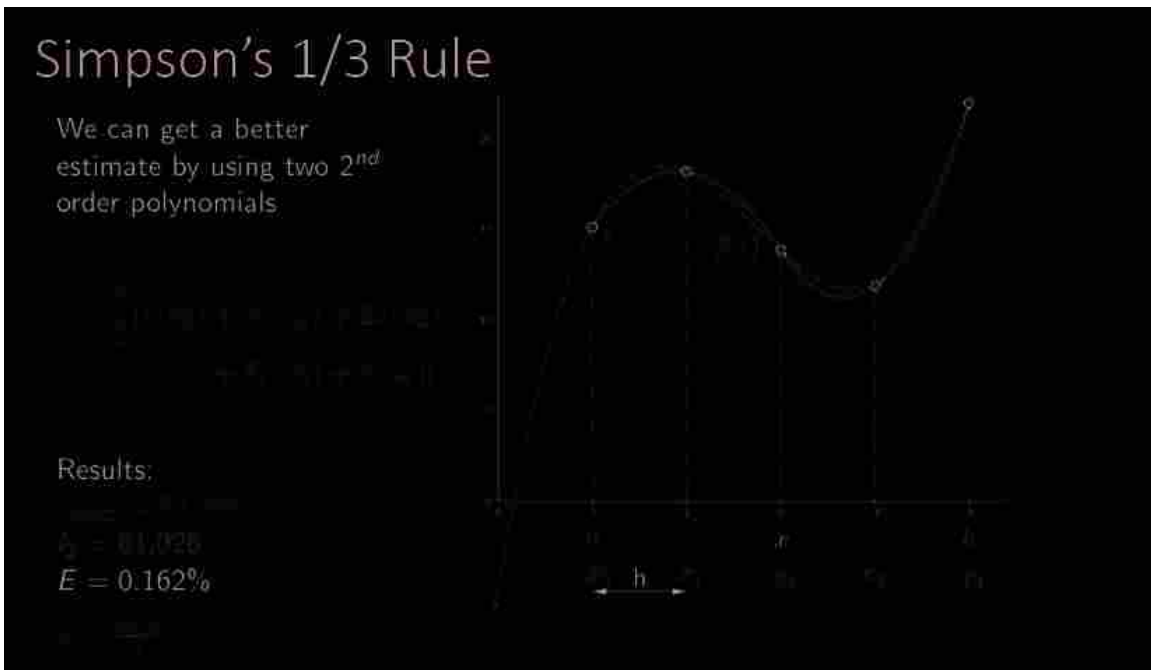


Figure C.54

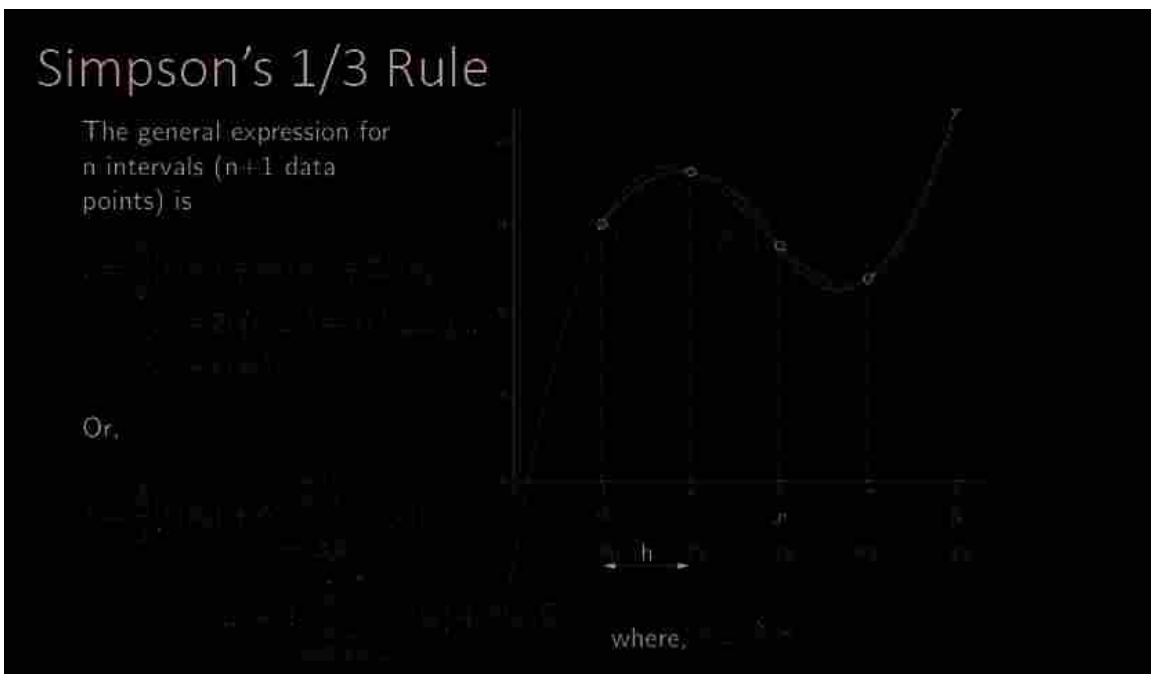


Figure C.55

Simpson's 1/3 Rule

- Shortcomings:
 - must be equally spaced points (constant h)
 - Only an even number of segments (odd number of data points) are allowed.
- Advantages:
 - Accuracy is what we would expect for a 4-point stencil even though we only use a 3-point stencil.

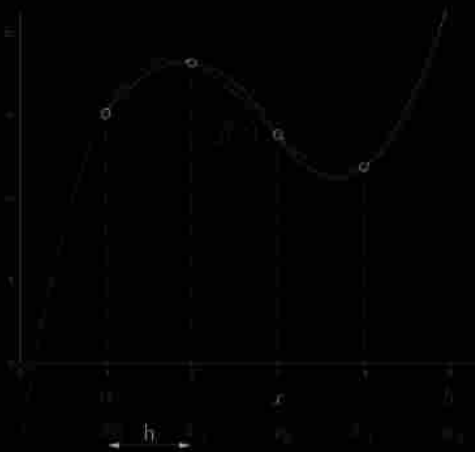


Figure C.56

Exercise

- Navigate to me273vm.et.byu.edu and log in
- Start OCAM!!!
- Start exercise Topic 6 Group #
- Fix the errors in the code. These may be algorithmic or syntax errors.

The general formula for $n+1$ data points, equally spaced between (a,b) is

$$I \approx \frac{h}{3} [f(x_0) + 4f(x_1) + 2f(x_2) + \dots + 4f(x_{n-1}) + f(x_n)]$$

where,

$$h = \frac{b-a}{n}$$

Figure C.57

Least Squares Regression

Figure C.58

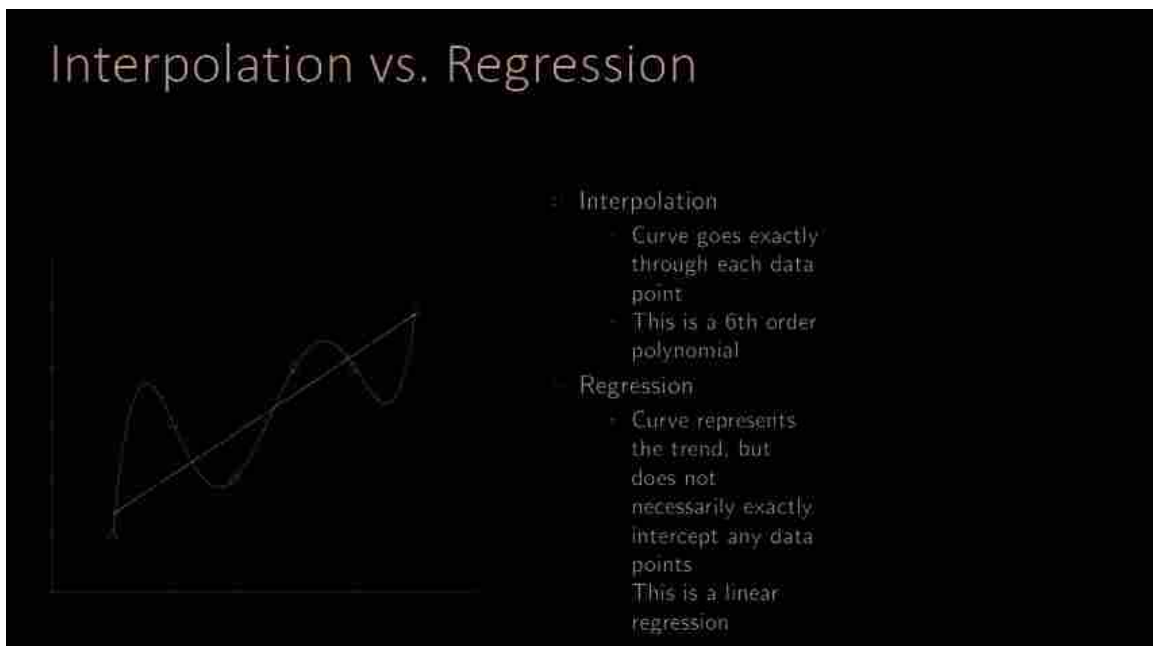


Figure C.59

General Notes



- We will focus on polynomial regression, although virtually any function form can be used
- Higher order regressions do not *always* give better representations of the trend
- Plotting the data points and the regression curve is essential to verify the appropriateness of k

Figure C.60

Basic Statistics

- > Mean (average)
 - A measure of the "center" of the data points

$$\bar{y} = \frac{\sum y}{n}$$

```
sum = 0.;
for(i=0; i<n; i++)
    sum += dat[i];
ave = sum/(double)n;
```

- > Standard deviation
 - A measure of the "spread" of the data points

$$s = s_x = \sqrt{\frac{\sum (y_i - \bar{y})^2}{n-1}}$$

```
S_t = 0.;
for(i=0; i<n; i++)
    S_t += pow(dat[i]-ave, 2);
stddev = sqrt(S_t/(double)(n-1));
```

Figure C.61

"Best Fit" Linear Regression

- **Residual:** Vertical distance between a data point and the "best fit" line.
- Minimize
 - *Sum of residuals*
 - Any line passing through the midpoint satisfies minimum
 - *Sum of absolute values of residuals*
 - Also does not yield a unique best fit
 - Any line falling between the dashed lines yields the same minimum
 - *Maximum error of any individual point*
 - Gives too much influence to single "outlier" points



Figure C.62

Least Squares Regression

- The previously mentioned shortcomings can be overcome by minimizing the sum of the squares of the residuals: S_r .
- This is called "least squares"

$$S_r = \sum (y_{\text{actual}} - y_{\text{predicted}})^2$$

- For linear regression,

$$y_{\text{predicted}} = a_0 + a_1(x)$$

$$S_r = \sum (y_{\text{actual}} - a_0 - a_1(x))^2$$

Figure C.63

Least Squares

$$S_1 = \sum (y_{i=1:n} - a_0 - a_1 x_i)^2$$

- In order to minimize, we take the partial derivatives with respect to a_0 and a_1 , then set equal to zero:

$$\frac{\partial S_1}{\partial a_0} = -2 \sum (y_{i=1:n} - a_0 - a_1 x_i) = 0$$

$$\frac{\partial S_1}{\partial a_1} = -2 \sum (y_{i=1:n} - a_0 - a_1 x_i) x_i = 0$$

Figure C.64

Least Squares

This results in two equations and two unknowns, which can be solved for a_0 and a_1 :

$$a_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

$$a_0 = \bar{y} - a_1 \bar{x}$$

Figure C.65

Exercise

- Navigate to me273vm.et.byu.edu
- Start OCAM
- Go to the Topic 7 exercise for your group
- Write a program that finds a_0 and a_1 for the following data points:
- x : [0, 2, 4, 6, 9, 17]
- y : [28, 35, 6, 10, 5, 1]

Figure C.66

C.1.8 Topic 8: Initial Value Problems

Initial Value Problems

Figure C.67

Initial Value Problems

- An initial value problem is a differential equation with a given initial condition. A differential equation is any equation with a derivative in it.
- Solving a differential equation means coming up with the original function $y(t)$

$$\frac{dy}{dt} = 2y + 5$$

$$y' + 4y - 2 = t$$

Figure C.68

Initial Conditions

- One initial condition is needed for each order or degree of the equation
- For example, a differential equation with only first derivatives and below is a first order equation and only needs one initial condition:
 - $y(0) = \text{something}$
- A differential equation with a second derivative but nothing higher needs two initial conditions:
 - $y(0) = \text{something}$
 - $y'(0) = \text{something}$

Figure C.69

Euler's Method

- Euler's method approximates the solution to a differential equation by using the slope at a point and small time steps, starting with the initial condition.
- This method generally overshoots and is not incredibly accurate, especially at large step sizes.
- Smaller step sizes generally improve accuracy
- Equation:
 - $y(i+1) = y(i) + f(x(i),y(i)) * \text{deltax}$
 - where $f(x(i),y(i)) = dy/dx$

Figure C.70

Step Size Convergence

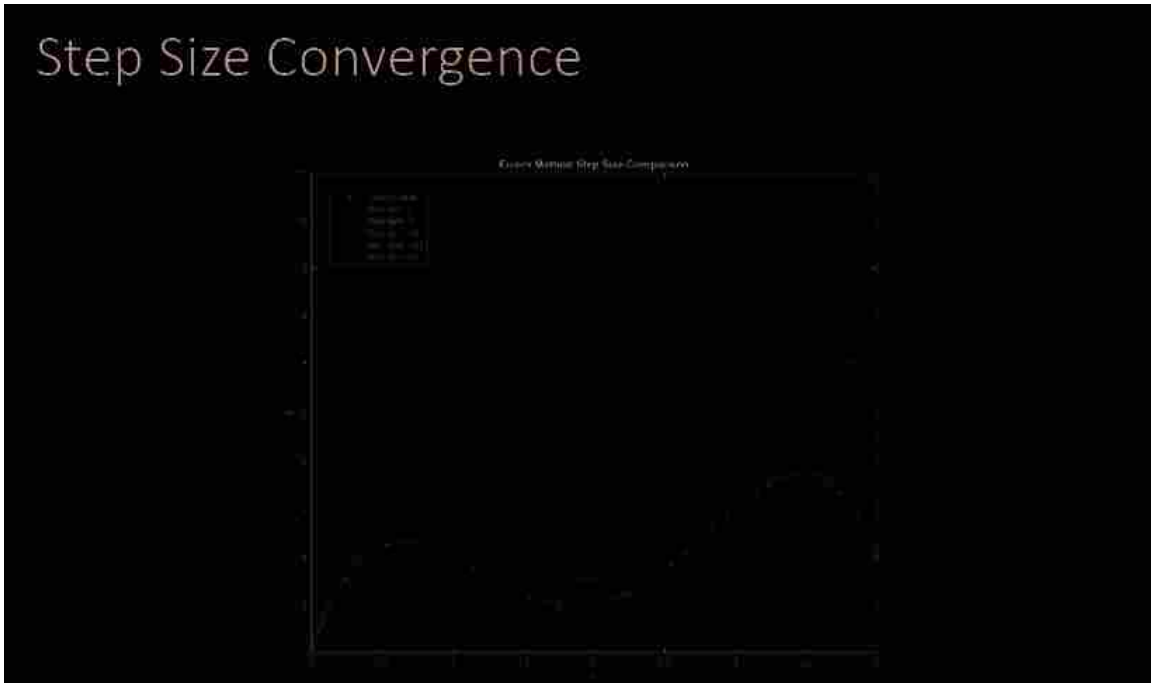


Figure C.71

Heun's Method (2nd order Runge Kutta)

- More accurate than Euler's method because it uses an average slope at two points to predict the next point

$$y_{i+1}^0 = y_i + f(x_i, y_i)h$$
$$y_{i+1} = y_i + \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^0)}{2}h$$
$$(\%) = \left| \frac{y_{i+1}^0 - y_{i+1}}{y_{i+1}} \right| (100\%)$$
$$L_i = \frac{f''(\xi_i)}{12}h^2$$

Figure C.72

Exercise

- Navigate to me273vm.et.byu.edu and log in
- Start the Topic 8 exercise with your group number
- Start OCAM
- Find the errors in the code. There may be syntax and algorithmic errors.

Figure C.73

C.2 Quiz Questions

C.2.1 Quiz 1: C++ Basics

1. Which of the following is a valid declaration statement?

- (a) `x = 10;`
- (b) `int x = 1.75;`
- (c) `double x(1.75);`

2. What is the output of the following code?

```
#include <iostream>

using namespace std;

int main()
{
    int x = 5;
    int y = 4;

    cout << x / y << endl;

    return 0;
}
```

Figure C.74

- (a) 1.25
- (b) 1
- (c) 1.5
- (d) 2

3. What is wrong with the following code?

```
#include <iostream>
using namespace std;
int main()
{
    int x;

    cout << "Please input a number: " << endl;
    cin << x;
    return 0;
}
```

Figure C.75

- (a) <<going the wrong direction for cin
- (b) Cin used instead of cout
- (c) Missing semi-colon

C.2.2 Quiz 2: C++ Loop Control

1. Which of the following is a valid syntax for a do while loop?

- (a) 1

```
#include <iostream>
using namespace std;
int main()
{
    int i = 0;
    do
    {
        cout << i++ << endl;
    } while (i < 5)
    return 0;
}
```

Figure C.76

- (b) 2

```
#include <iostream>
using namespace std;

int main()
{
    int i = 0;

    do
    {
        cout << i++ << endl;
    } while (i < 5);

    return 0;
}
```

Figure C.77

(c) 3

```
#include <iostream>
using namespace std;

int main()
{
    int i = 0;

    do {
        cout << i++ << endl;
    }
    while (i < 5)

    return 0;
}
```

Figure C.78

2. What is the output of the following code?


```
#include <iostream>

using namespace std;

int main()
{
    int x = 5;
    int y = 17;

    for (int i = 5; i > 1; i--)
    {
        if (x < y && x + y <= 25)
        {
            cout << x << " " << y;
            x += 1;
            y -= 3;
        }
    }

    return 0;
}
```

Figure C.79

- (a) 5 176 147 118 89 5
- (b) 5 176 147 118 8
- (c) 5 176 147 11
- (d) 5 176 14

3. What is wrong with the following code?

```
#include <iostream>
using namespace std;

int main()
{
    int x = 5;

    switch (x)
    {
    case 0:
        cout << "YAY!";
    case 1:
        cout << "BOO!";
    case 2:
        cout << "what?";
    default:
        cout << "Whassup?";
        break;
    }

    return 0;
}
```

Figure C.80

- (a) break is missing from cases 0, 1, and 2
- (b) there's no semi-colon at the end of the switch statement
- (c) parentheses are missing: case(0): instead of case 0:

C.2.3 Quiz 3: C++ Functions

1. Which of the following is a correct way to define and use a function?

- (a) 1

```
#include <iostream>

using namespace std;

int main()
{
    double x = 5;
    double a = 2;

    cout << exponent(x, a);

    return 0;
}

//does the operation x^a for two doubles
double exponent(double x, double a)
{
    double product = x;
    bool isNeg = false;

    if (a == 0)
        return 1.0;
    else if (a < 0)
    {
        a = -a;
        isNeg = true;
    }

    for(int i = 1; i < a; i++)
    {
        product = product * x;
    }

    if(isNeg)
        return 1/product;
    else
        return product;
}
```

Figure C.81

(b) 2

```
#include <iostream>

using namespace std;

int main()
{
    exponent(5, 2);

    return 0;
}

//does the operation x^a for two doubles.
double exponent(double, double)
{
    double product = x;
    bool isNeg = false;

    if (a == 0)
        return 1.0;
    else if (a < 0)
    {
        a = -a;
        isNeg = true;
    }

    for(int i = 1; i < a; i++)
    {
        product = product * x;
    }

    if(isNeg)
        return 1/product;
    else
        return product;
}
```

Figure C.82

(c) 3

```
#include <iostream>

using namespace std;

double exponent(double, double);

int main()
{
    double x = 5;
    double a = 2;

    cout << exponent(x, a);

    return 0;
}

//does the operation x^a for two doubles
double exponent(double x, double a)
{
    double product = x;
    bool isNeg = false;
    if (a == 0)
        return 1.0;
    else if (a < 0)
    {
        a = -a;
        isNeg = true;
    }

    for(int i = 1; i < a; i++)
    {
        product = product * x;
    }

    if(isNeg)
        return 1/product;
    else
        return product;
}
```

Figure C.83

2. What is the output of the following code?

```
#include <iostream>
using namespace std;
void iterate(double&);
int main()
{
    double x = 5;

    cout << x << " ";

    iterate(x);

    cout << x;

    return 0;
}

void iterate(double& y)
{
    y += 17;
    y *= 2;
}
```

Figure C.84

(a) 5 44

(b) 5 4

3. What is wrong with the following code?

```

#include <iostream>

using namespace std;

void fun(double);

int main()
{
    double x = 5;

    fun(x);

    return 0;
}

void fun(double& y)
{
    cout << y++ << endl;
    return;
}

```

Figure C.85

- (a) The function prototype doesn't indicate the double is passed by reference
- (b) The function prototype doesn't specify the variable name
- (c) fun(x) needs to be fun(&x) to pass the variable by reference

C.2.4 Quiz 4: Root Finding Methods

1. Which of the following methods is a bracketing method?
 - (a) Bisection
 - (b) Secant
 - (c) Newton-Rhapson

2. What is a benefit of the Newton-Rhapson method?
 - (a) It always converges
 - (b) It converges quickly
 - (c) It doesn't require a derivative calculation

3. Do one iteration of the Newton-Rhapson method for the following function with an initial guess of 5. Report the new guess. $f(x) = x^2$

C.2.5 Quiz 5: C++ Static Arrays

1. Which of the following is a valid way to initialize an array?
- (a) `string names[5] = "Percy","Fred","George","Bill","Charlie";`
 - (b) `int nums[] = 1, 2, 3, 4;`
 - (c) `double points[3] = [5,4,3];`
2. What is the output of the following code?

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string friends[] = { "Hermione", "Neville", "Luna", "Ginny", "Ron" };
    string yesorno[2] = { " ", " not " };

    cout << friends[3] << " is" << yesorno[1] << "Harry Potter's best friend." << endl;

    system("pause");
    return 0;
}
```

Figure C.86

- (a) Ron is Harry Potter's best friend.
 - (b) Ginny is not Harry Potter's best friend.
 - (c) Luna is not Harry Potter's best friend.
3. What are the errors in the following code?


```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string creatures = {"Buckbeak","Fang","Fluffy","Norbert"};

    for (int i = 0; i < 4; i++)
    {
        cout << creatures[i] << endl;
    }

    system("pause");
    return 0;
}

```

Figure C.87

- (a) Did not declare the size of the array
- (b) Used curly brackets instead of square brackets
- (c) Didn't use square brackets after creatures

C.2.6 Quiz 6: Numerical Integration

1. What is a shortcoming of the trapezoid rule?
 - (a) the points must be equally spaced
 - (b) the number of points must be even
2. Which algorithm involves summing up odd and even segments?
 - (a) Simpson's 3/8
 - (b) Trapezoid
 - (c) Simpson's 1/3
3. Which method has less error in general?
 - (a) Simpson's 3/8
 - (b) Trapezoid
 - (c) Simpson's 1/3

C.2.7 Quiz 7: Least Squares Regression

1. True or false: a higher order regression is always a better representation of the trend
 - (a) true
 - (b) false
2. True or false: interpolation goes through every point, while regression shows a general trend
 - (a) true
 - (b) false
3. a_0 and a_1 are:
 - (a) the intercept and slope of the regression line
 - (b) the interpolation parameters
 - (c) the intercepts of the regression line

C.2.8 Quiz 8: Initial Value Problems

1. Decreasing step size generally makes Euler's method more accurate
 - (a) true
 - (b) false
2. Heun's method is more accurate than Euler's method because:
 - (a) it does more iterations
 - (b) it uses two points to calculate an estimated slope
 - (c) it uses a smaller step size
3. How many initial values does a second order differential equation need?
 - (a) 3
 - (b) 2
 - (c) 1

C.3 Exercises

C.3.1 Exercise 3

```
//do not use cin
//do not use system pause

//Exercise: Fix the errors in the following code.
//This program should find the correct average of two numbers using a function.
```

```
#include <iostream>

using namespace std;

int main()
{
    double x = 5;
    double y = 4;

    cout << average(x y) << endl;

    return 0;
}

int average(num1, num2)
{
    return (num1 + num2) /2;
}
```

C.3.2 Exercise 4

```
//do not use cin
```

```

//do not use system pause
//do not use arrays
//You can use cmath, but not math.h

//Fix the errors in the following code.
//There are both syntax and algorithmic errors

(typos that may not throw an error message).

//This program uses the Newton-Rhapson method to find the root of a function
//Newton-Rhapson algorithm:  $x_{(n+1)} = x_n - f(x_n)/f'(x_n)$ 
//The correct answer is  $x=1.707$ 

#include <iostream>

using namespace std;
//the function we're finding the root of
double f(x)
{
    return x*x*x + 5*x*x - 12*x + 1;
}
//the derivative of the function we're finding the root of
double fprime(x)
{
    return 3*x*x + 10*x;
}

int main()
{
    double error;

```

```

double guess = 3;
while(error > .0001)
{
    guess = guess + f(guess)/fprime(guess);
    error = f(guess);

    cout << error << endl;
}
cout << endl << guess << endl;
return 0;
}

```

C.3.3 Exercise 6

```

//do not use cin
//do not use system pause
//write your code in the try catch statement

//Fix the errors in the following code. These may be logical/algorithmic
errors or syntax errors.
//The following code uses the trapezoid rule to estimate the integral of
the function
//f(x) = x2 + 5x - 20 between a = -4 to b = 2 with n = 5 segments

//The trapezoid rule formula: I = h/2 * (f(x0) + 2*sum from 1 to n-1
of f(xi) + f(xn))
//where h = (b-a)/n

```

```

//The exact answer is I = -126

#include <iostream>
#include <string>

using namespace std;

double f(double x)
{
    return x*x + 5*x - 20;
}

int main()
{
    try
    {
        //put your code here
        int a = -4;
        int b = 2;
        n = 5;

        int h = b-a/n;

        double sum = h/2*(f(a) + f(b));
        double x = a + h;

        for(int i = 0; i < n; i++)
        {
            sum = 2*f(x);

```

```

        x += h;
    }
    cout << sum << endl;
}
catch(exception ex)
{}
return 0;
}

```

C.3.4 Exercise 8

```

//Do not use cin
//Do not use system pause

//Fix the errors in this program. There may be both syntax and
algorithmic errors.

//Solve the given differential equation function using Euler's method with
the following parameters:
//y(0) = 1
//dy/dx = -2x^3 + 12x^2 - 20x + 8.5
//deltax = .25
//x0 = 0
//xn = 4 (from x = 0 to x = 4)

//Euler's method:
//y(i+1) = y(i) + dydx(x(i))*deltax

//Print out the value of y(4). The correct answer is 3, but with your given

```

step size it should be close to 5.

//You can try testing smaller step sizes and see if it converges to 3.

```
#include <iostream>
```

```
using namespace std;
```

```
int dydx(double x)
```

```
{
```

```
    return -2*x*x*x + 12*x*x - 20*x + 8.5;
```

```
}
```

```
int main()
```

```
{
```

```
    try
```

```
    {
```

```
        double step = .25;
```

```
        double lb = 0;
```

```
        double ub = 4;
```

```
        double num = ub-lb/step;
```

```
        //set up y values
```

```
        double yvals[num];
```

```
        //initial condition
```

```
        yvals[1] = 1;
```

```
        //set up x values
```

```
        double xvals[num];
```

```
        xvals[1] = 0;
```



```
for(int i = 1; i < num; i++)
{
    xvals[i] = xvals[i-1] + step;
}

//Euler's method
for(int i = 0; i < num; i++)
{
    yvals[i+1] = yvals[i] + dydx(xvals[i])*step;
}

//Print solution for last value
cout << yvals[num-1] << endl;
}
catch(exception ex)
{}

return 0;
}
```

APPENDIX D. IRB DOCUMENTS

D.1 IRB Approval

Memorandum

To: Ariana Pedersen

Department: ME

College: E&T

From: Sandee Aina, MPA, IRB Administrator

Bob Ridge, PhD, IRB Chair

Date: May 24, 2017

IRB#: E17213

Title: Live In-Class Compiler Feedback in Programming Exercises

Brigham Young Universitys IRB has approved the research study referenced in the subject heading as exempt, category 2.

The approval period is from May 24, 2017 to May 23, 2018. Please reference your assigned IRB identification number in any correspondence with the IRB.

Continued approval is conditional upon your compliance with the following

requirements:

CONTINGENCY: Submit the classroom announcement script.

A copy of the informed consent statement is attached. No other consent statement should be used. Each research subject must be provided with a copy or a way to access the consent statement.

Any modifications to the approved protocol must be submitted, reviewed, and approved by the IRB before modifications are incorporated in the study.

All recruiting tools must be submitted and approved by the IRB prior to use.

5. In addition, serious adverse events must be reported to the IRB immediately, with a written report by the PI within 24 hours of the PI's becoming aware of the event.

Serious adverse events are (1) death of a research participant; or (2) serious injury to a research participant.

6. All other non-serious unanticipated problems should be reported to the IRB within 2 weeks of the first awareness of the problem by the PI.

Prompt reporting is important, as unanticipated problems often require some modification of study procedures, protocols, and/or informed consent processes.

Such modifications require the review and approval of the IRB.

A few months before the expiration date, you will receive a continuing review form.

There will be two reminders. Please complete the form in a timely manner to ensure that there is no lapse in the study approval.

IRB Secretary
Office of Research and Creative Activities
A 285 ASB
Brigham Young University
(801)422-3606
irb@byu.edu

D.2 Recruiting Script

Classroom Script: Students, Dr. Salmon and Ariana Pedersen are doing an experiment to improve how people teach computer programming to mechanical engineers. They have created a website that gathers data on students programs as they code. This data includes any errors that occurred, number of lines, how many times a user tries to run the program, and program output. Dr. Salmon receives anonymous summaries of this data to see how the class is doing overall. Using this website is required for in-class exercises during the course. You will also be required to complete 5-10 minute at home quizzes as a control. These quizzes will be graded on participation only and the accuracy of your answer will not affect your grade at all. You are not

required to allow your data to be used and published as part of this experiment. Any data that is published will be completely anonymous and only reported in aggregate, in the form of averages, standard deviations, etc. Allowing your data to be used in the experiment will be very helpful to the researchers, and will hopefully improve how programming courses are taught and make the course easier for students. I am going to pass a piece of paper around. If you're interested in giving your consent or want to find out more about the experiment, please write down your email address. I will send you a survey that explains the experiment a little more and asks whether or not you consent to having your data used. Dr. Salmon and Ariana will not know which students have consented to participate in the experiment. I (Landon) am in charge of anonymizing all of the data and keeping track of which students have given their consent. The website is also set to exclude Dr. Salmon and Ariana from seeing any identifying information. If you want to withdraw your consent at any point during the experiment, you can email me and I will remove your data within 24 hours. Are there any questions?

D.3 Other Relevant Documents

The consent document is included in Appendix B.1.1 as part of a survey.